Compiler-Assisted Software Fault Tolerance for Bare Metal and RTOS Applications on Embedded Platforms

Benjamin James
Brigham Young University

Follow this and additional works at: https://scholarsarchive.byu.edu/etd

Part of the Engineering Commons

BYU ScholarsArchive Citation
James, Benjamin, "Compiler-Assisted Software Fault Tolerance for Bare Metal and RTOS Applications on Embedded Platforms" (2021). Theses and Dissertations. 8958.
https://scholarsarchive.byu.edu/etd/8958

This Thesis is brought to you for free and open access by BYU ScholarsArchive. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of BYU ScholarsArchive. For more information, please contact ellen_amatangelo@byu.edu.
Compiler-Assisted Software Fault Tolerance for Bare Metal and RTOS Applications on Embedded Platforms

Benjamin David James

A thesis submitted to the faculty of Brigham Young University in partial fulfillment of the requirements for the degree of Master of Science

Jeffrey B. Goeders, Chair
Mike Wirthlin
Phil Lundrigan

Department of Electrical and Computer Engineering Brigham Young University

Copyright © 2021 Benjamin David James
All Rights Reserved
ABSTRACT

Compiler-Assisted Software Fault Tolerance for Bare Metal and RTOS Applications on Embedded Platforms

Benjamin David James
Department of Electrical and Computer Engineering, BYU
Master of Science

In the presence of ionizing particles and other high-energy atomic sources, many electronic and computer systems fail. Single event upsets (SEUs) can be mitigated through hardware and/or software methods. Previous research at BYU has introduced COAST, a compiler-based tool that can automatically add software protection schemes to improve fault coverage of programs.

This thesis will expand on the work already done with the COAST project by proving its effectiveness across multiple platforms and benchmarks. The ability to automatically add fault protection to arbitrary user programs will be very valuable for many application designers. The results presented herein show that mean work to failure (MWTF) of an application can increase from 1.2x – 36x when protected by COAST.

In addition to the results based on bare metal applications, in this thesis we will show that it is both possible and profitable to protect a real-time operating system with COAST. We present experimental results which show that our protection scheme gives a 2x – 100x improvement in MWTF. We also present a fault injection framework that allows for rapid and reliable testing of multiple protection schemes across different benchmarks. The code setup used in this paper is publicly available. We make it public in the hope that it will be useful for others doing similar research to have a concrete starting point.

Keywords: COAST, LLVM
ACKNOWLEDGMENTS

As with any large project, there are always many people working behind the scenes that don’t get their name on the finished product. I would like to recognize some of those people for their contributions that helped make my research and writing successful.

First of all, I owe a lot to my advisor, Jeff Goeders. He helped me pick this project and has put up with my incessant questions and slowly coached me into solving problems by myself. His expertise and experience have been very instructive over the past few years, and he is one who knows how to teach by example.

I want to express my thanks to Dr. James Archibald for teaching me a lot about computer architecture, and significantly increasing my repository of technical jokes. His knowledge and humor were both helpful in my understanding and enduring the technical challenges I faced working on this project.

I would like to thank both the National Science Foundation (NSF) and Los Alamos National Laboratory (LANL) for their financial contributions that made this work possible.

My family has also been very supportive, most especially my wife Brittany. Her belief in me has been just as important as the physical help she gave in making sure I always have a meal to eat.
# TABLE OF CONTENTS

**TITLE PAGE** ................................................................. i

**ABSTRACT** ................................................................. ii

**ACKNOWLEDGEMENTS** ................................................... iii

**LIST OF TABLES** .......................................................... viii

**LIST OF FIGURES** ........................................................ ix

**PREFACE** ................................................................. 1

**CHAPTER 1** **INTRODUCTION** ........................................... 2

  1.1 Motivation ......................................................... 2

  1.2 Thesis Contributions ............................................. 3

  1.3 Thesis Organization ............................................... 4

**CHAPTER 2** **BACKGROUND** ............................................. 6

  2.1 Single Event Effects ............................................. 6

  2.2 Error Mitigation .................................................. 7

    2.2.1 Software Protection – Related Work ....................... 7

  2.3 Compilers and LLVM ............................................... 8

  2.4 COAST .............................................................. 9

    2.4.1 Testing Additional Applications of COAST ............... 11

  2.5 Fault Injection – Related Work ................................ 11

**CHAPTER 3** **EXTENDING COAST TO NEW PLATFORMS AND BENCHMARKS** 13

  3.1 Extending Support for ARM and RISC-V ....................... 13

  3.2 Verifying Correctness ........................................... 14

  3.3 Experimental Setup ............................................. 15

    3.3.1 Devices Under Test ....................................... 15

    3.3.2 Benchmarks ................................................. 17

    3.3.3 Test Methodology ......................................... 19

    3.3.4 Experimental Results ................................... 20

  3.4 Analysis of Test Results ........................................ 22

    3.4.1 Variation between benchmarks ............................ 23

    3.4.2 Variation between platforms .............................. 24

    3.4.3 Variation between cache configuration .................. 24

  3.5 Hercules – Hardware-Assisted Fault Tolerance ............. 25

    3.5.1 Experimental Setup ....................................... 25

    3.5.2 Results ...................................................... 26

  3.6 Conclusion ....................................................... 27

**CHAPTER 4** **UNDERSTANDING BENCHMARK PROPERTIES** ............. 28

  4.1 Experimental Setup ............................................. 28

    4.1.1 Device Under Test ....................................... 29
D.2 Normal Usage ......................................................... 121
D.3 Compiling with the COAST toolchain ......................... 121
LIST OF TABLES

3.1 Neutron beam test results ................................................. 20
3.2 Radiation Test Results – Hercules ........................................ 26
4.1 Neutron beam test results .................................................. 31
4.2 Linear Regression Model ..................................................... 33
5.1 Fault Injection – Evaluation ................................................ 48
5.2 Fault Injection – Rates ....................................................... 49
5.3 Profiling – nanojpeg.tmr ..................................................... 54
7.1 Fault Injection Results – rtos_kUser ........................................ 71
7.2 Fault Injection Results – rtos_mm .......................................... 71
7.3 Kernel Profiling – rtos_kUser .............................................. 74
B.1 Fault Injection Results – Stack Protection ............................. 113
C.1 Fault Injection Results – rtos_fat_demo ............................... 119
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure Number</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Diagram of LLVM Framework</td>
<td>9</td>
</tr>
<tr>
<td>2.2</td>
<td>List of Replication Rules</td>
<td>10</td>
</tr>
<tr>
<td>2.3</td>
<td>Example Code Before and After TMR Mitigation</td>
<td>10</td>
</tr>
<tr>
<td>3.1</td>
<td>COAST compiler flow for supporting new toolchains</td>
<td>14</td>
</tr>
<tr>
<td>3.2</td>
<td>Neutron beam test setup – 2018</td>
<td>17</td>
</tr>
<tr>
<td>3.3</td>
<td>Cross-section view of board placement in neutron beam</td>
<td>18</td>
</tr>
<tr>
<td>3.4</td>
<td>Matrix Multiplication kernel code</td>
<td>18</td>
</tr>
<tr>
<td>3.5</td>
<td>Benchmark Cross Sections – 2018</td>
<td>21</td>
</tr>
<tr>
<td>4.1</td>
<td>Neutron beam test setup – 2019</td>
<td>29</td>
</tr>
<tr>
<td>4.2</td>
<td>Benchmark Cross Sections – 2019</td>
<td>32</td>
</tr>
<tr>
<td>4.3</td>
<td>malloc memory chunk internals</td>
<td>35</td>
</tr>
<tr>
<td>5.1</td>
<td>Fault Injection Framework – Use Case</td>
<td>40</td>
</tr>
<tr>
<td>5.2</td>
<td>Framework Process Diagram – arrows represent communication over Unix sockets.</td>
<td>41</td>
</tr>
<tr>
<td>5.3</td>
<td>Control flow graph – fft</td>
<td>55</td>
</tr>
<tr>
<td>6.1</td>
<td>Scope of Replication – Example</td>
<td>59</td>
</tr>
<tr>
<td>7.1</td>
<td>MWTF Summary – FreeRTOS</td>
<td>72</td>
</tr>
<tr>
<td>7.2</td>
<td>Benchmark Run times – FreeRTOS</td>
<td>74</td>
</tr>
<tr>
<td>7.3</td>
<td>Benchmark Memory Usage – FreeRTOS</td>
<td>75</td>
</tr>
<tr>
<td>7.4</td>
<td>Cache Miss Rate – rtos_mm</td>
<td>76</td>
</tr>
<tr>
<td>7.5</td>
<td>Cache Operations – rtos_mm</td>
<td>77</td>
</tr>
<tr>
<td>A.1</td>
<td>Initial state of linked list</td>
<td>101</td>
</tr>
<tr>
<td>A.2</td>
<td>Linked list after insertion</td>
<td>102</td>
</tr>
<tr>
<td>A.3</td>
<td>Initial state of TMR’d linked list</td>
<td>102</td>
</tr>
<tr>
<td>A.4</td>
<td>Inserting into TMR’d list</td>
<td>103</td>
</tr>
<tr>
<td>A.5</td>
<td>Inserting into TMR’d list – revisited</td>
<td>104</td>
</tr>
<tr>
<td>A.6</td>
<td>Removing TMR’d list</td>
<td>104</td>
</tr>
<tr>
<td>A.7</td>
<td>Removing TMR’d list – revisited</td>
<td>105</td>
</tr>
<tr>
<td>B.1</td>
<td>Example Stack Frame</td>
<td>109</td>
</tr>
</tbody>
</table>
PREFACE

Many of the contributions presented in this thesis have been published in other venues, such as journals and conference proceedings.

Content from both Chapter 3 and Chapter 4 were published separately in IEEE Transactions on Nuclear Science [1], [2].

Parts of Chapter 5 have been submitted for publication in DAC. Portions of Chapters 6 and 7 were submitted to the DATE conference and are currently undergoing the review process.


CHAPTER 1. INTRODUCTION

1.1 Motivation

As the number of electronic devices in use is increasing world-wide, and as the size of the transistors on these devices is shrinking, it is becoming more and more critical to properly deal with radiation-induced errors. Although devices exist which have been manufactured to be inherently radiation-tolerant, these devices are much more expensive than their commercial off-the-shelf counterparts. The rigorous testing required to certify such devices means the base technology also lags behind, which can lead to them being orders of magnitude slower in performance. Because of the price and performance drawbacks of these radiation-hardened processors, many researchers have turned to software techniques to give the cheaper, more modern microcontrollers greater fault tolerance.

Radiation-induced errors can be of concern in terrestrial applications, though this usually applies only to very large scale projects, such as data centers or super computers. This is because there are statistically many more possible components which could experience radiation-induced upsets than in other applications. This thesis is not concerned with these types of projects; rather our focus is where microcontrollers would be exposed to higher levels of radiation than they would be if protected by the Earth’s atmosphere. This includes aircraft, satellites, high-energy physics experiments, and so forth. As the failure of a microcontroller in these applications is potentially catastrophic, it is desirable to find ways to make these devices more reliable in radiation-prone environments.

We are not the first to do research in this area of software fault tolerance; multiple methods and algorithms have been published which have been shown to add significant fault tolerance to microcontrollers and their programs. One of the main methods for fault tolerance is by instruction replication. Past work has published replication rules for optimal fault coverage [1] and created automated tools that implement various protection techniques [2], [3] based on instruction repli-
cation. The work presented in this thesis builds upon the great work that has already been done. While past work has developed tools, these were mostly closed source or dependent on a particular architecture or feature to function. Our fault tolerance tool is open-source, is independent of the target architecture, and can be used to protect arbitrary user programs.

The COAST project (Compiler-Assisted Software Fault Tolerance) began in 2016 with the goal of creating a portable and flexible tool which could be used to automatically add software fault tolerance to a wide variety of software programs. Previous publications presented the groundwork and initial testing of the COAST tool [4], [5]. In this thesis, we demonstrate not only the correct functionality of the tool through extensive testing, but also its wide applicability to multiple configurations. This includes four different target architectures and more than 10 different benchmarks, as well as 2 protected applications for a real-time operating system.

1.2 Thesis Contributions

This thesis presents major contributions made to the COAST project, and to the work of software fault tolerance in general. Building on the foundation of COAST first developed by Matthew Bohman [4], this work demonstrates that COAST does fulfill its goal of being applicable to more than one architecture. The exact ways in which COAST was used and tested to complete this goal are briefly summarized in the following paragraphs.

First, the thesis presents experimental results from two different radiation tests. The first of these tests was conducted to show the applicability of COAST to multiple target architectures. The second test was done to show that COAST can be used to apply protection to a variety of software benchmarks. As part of the analysis of these experimental results, we show how certain attributes of software programs make them more or less conducive to protection from COAST.

After this, the thesis discusses applying automated fault tolerance to a more complex system. Most work with COAST thus far has targeted bare metal benchmarks. To show that COAST is also useful for more complex systems, we apply COAST protection to FreeRTOS, a real-time operating system. We chose this particular real-time kernel because it is commonly used in embedded microcontroller applications and, if successful, would expand the potential applications of COAST to a much wider audience. A lot of work went into understanding the real-time kernel and its different subsystems in order to create a protection scheme that would provide the most benefit.
This protection scheme is described in detail as part of this thesis, along with general principles that can be applied to other complex systems.

The research on protecting FreeRTOS was largely done during the COVID-19 pandemic, which precluded any radiation testing from happening. To evaluate our protection scheme without radiation testing, we created a fault injection framework to validate the correct operation of the protection scheme. The goal of this testing was to ensure that the protection scheme prevented silent data corruption (SDC) as much as possible. This fault injector has proven very useful in many applications to prove the correctness of COAST. The design and validation of this fault injection framework is discussed as a contribution of this thesis.

A corollary to all of the work just mentioned is that the COAST tool itself is much more stable and has more features than it did before. Over the 3+ years I have been working on this project, I have updated COAST with the following features:

- Support for voting on SIMD instructions
- Functions can “return” multiple values
- Support for C++ programs
- In-code directives are more fully supported
- More fine control over replication scope and rules

The source code for the COAST tool, as well as most of the experiments that will be discussed in this thesis, are all available at the project website, https://github.com/byucc1/coast.

1.3 Thesis Organization

Chapter 2 summarizes software protection concepts, and reviews some research that has been done in this field. There is also a discussion of the basics of COAST and how it implements software protection.

The rest of this thesis is organized to present a major contribution in each chapter. Chapters 3 and 4 will cover results from two radiation experiments that were performed at the Los
Alamos Neutron Science Center. These experiments tested the applicability of COAST to multiple architectures and software benchmarks. The implications of the results from both of these experiments are discussed.

Chapter 5 presents the design of a fault injection framework, PACIFIC. Chapter 6 covers the challenges we ran into while constructing the protection scheme for FreeRTOS, including the modifications to COAST that were necessary. Chapter 7 provides analysis of the results of fault injection on the protected RTOS applications.

The thesis concludes with Chapter 8, which discusses some of the future work that is still possible on this project. There are also four appendices: Appendix A covers some of the pitfalls that often cause compilation or execution issues with COAST-protected code; Appendix B gives a brief report of investigation we made into protecting the system stack from corruption; Appendix C gives some information and results about a small experiment protecting a file system with COAST; and Appendix D gives some low-level implementation details of the fault injection framework.
CHAPTER 2. BACKGROUND

2.1 Single Event Effects

This thesis deals with mitigating errors in microcontrollers caused by ionizing radiation. High energy particles and other types of radiation can cause single event effects (SEE). There are multiple classes of events that can happen, some of which are described below.

**Single Event Upsets (SEU):** a change of state in one of the bits in the system (in the cache, register file, pipeline registers, etc). These can be transient (go away after a while) or not (require power reset to fix).

**Single Event Functional Interrupt (SEFI):** “A condition where the device stops operating in its normal mode”, but that can be solved with a power reset.

**Single Event Transient (SET):** “A current transient induced by the passage of a particle through an integrated circuit. The current can propagate to cause an output error in combinational logic” [6].

There are other types of errors described in [6], but these are usually more destructive and permanent, and so will not be discussed here. Single event upsets in data memory, often called soft errors, are more common forms of SEEs, and mitigating the effects of soft errors is the focus of this thesis.

Quinn et al. [7] have shown that the probability of SEEs occurring in a device is increasing because the size of transistors is decreasing and the total number of transistors in a component is increasing. Mitigating soft errors will continue to become more important as transistor size continues to shrink and as the number of transistors in a component continues to increase.
2.2 Error Mitigation

There are different approaches to mitigating these kinds of soft errors. One way is using processors specifically made to tolerate high levels of radiation. These radiation-hardened processors, while quite reliable, are very expensive. In addition, because of the strict requirements for these radiation-hardened devices, they often lag behind their commercial counterparts in terms of features and toolchains [8].

2.2.1 Software Protection – Related Work

Using commercial off-the-shelf (COTS) processors in place of expensive radiation-hardened processors is a way to cut down on costs, if software methods can be found that give enough fault tolerance to these COTS microcontrollers [9]. One common approach to software-based fault tolerance is duplicating or triplicating parts of a program in order to detect or correct errors. Code replication carries noticeable overhead costs, as it effectively doubles or triples the execution time and the size of the code in memory. However, the approach is still effective at reducing the error rate, and can actually net an increase in mean work to failure (MWTF) [1], [3], [10], [11]. Mean work to failure is a metric used to compare the improvement in fault tolerance, normalized with respect to the benchmark run time. Our use of this metric is discussed in more detail in Section 3.3.4.

Some of this past work has relied on modifying the assembly code by hand to add replicated instructions [3], [10]. While this is effective, it is also prone to human error. Even if the modification is done correctly, this is a one-shot approach; if the base program ever changes, the assembly will need to be modified again. Other applications may be too large to apply this technique in any manageable time-frame.

Fortunately, other past projects have built on this idea and created automated ways of adding software-based fault tolerance. Oh et al. presented the idea of error detection by duplicated instructions (EDDI) [2]. Reis et al. created a tool called SWIFT (software implemented fault tolerance) [12], which introduced software-based triplication, allowing not only error detection but also correction. The granularity of the protection schemes presented in these works may differ; some replicate on the instruction level, and others operate at the function level.
Another work that influenced the work in this thesis was done by Chielle et al. [13], who presented a set of rules that can be used to guide decisions about replication and synchronization. Other works target only specific architectures [2], [14], [15], assembly languages [1], [13], [16], [17], processor features [18], [19], or are dependent on multicore systems [20], [21], making them of limited use for future research or commercial projects. Other works have focused on protecting control-flow rather than data-flow [22]–[24]. Most of the works have focused on a server-like environment, targeting high-performance, superscalar processors, rather than embedded systems [2], [11], [12], [14], [15], [18]–[25]. Of the above described tools, none of the works provided open-source, publicly available tools. Furthermore, only four of the papers [1], [3], [10], [26] present results tested in an actual high-radiation environment, the rest only have only simulated upsets with fault injection.

2.3 Compilers and LLVM

Performing replication at the instruction level will provide better fault coverage than something more coarse-grained because it is funneling more data through a single synchronization point [4]. A key to implementing automated software-based fault protection in this manner lies in leveraging a compiler to generate the appropriate code structures. As humans, it is easier for us to think about difficult problems if the problems can be abstracted to a high level. The replication we seek is at the level of machine instruction, not a very high abstraction. Trying to understand how a user program (for example written in C) maps to the assembly the compiler generates is quite difficult for most meaningful programs, and one of the reasons that hand-modification is not an ideal solution. The complexity of trying to change the assembly code by hand after the compiler has touched it is one of the main reasons why the projects mentioned above almost always modified part of their toolchain in some way to achieve the desired instruction replication.

Some of the previous works have used GCC and modified the back end assembler for a specific processor architecture. This limits the applicability of the tool to that particular architecture. LLVM is a compiler framework that many use as an alternative GCC. The idea of LLVM is that it has multiple independent front- and back ends, which allows the central optimizer to operate on a common code representation (see Figure 2.1). For example, a user can write their program in C, then use the frontend tool clang to compile this program to the LLVM intermediate representation.
The LLVM optimizer can make all necessary code permutations at this abstracted level, and when it’s done, the appropriate back end will generate the assembly code for the selected target architecture. While GCC does operate similarly to LLVM, the LLVM IR is human-readable, and the GCC IR is not.

2.4 COAST

The project for Compiler-Assisted Software Fault Tolerance began a few years ago, and I joined the project in 2017. The COAST tool is able to add replicated instructions to user programs written in different languages (C and C++ tested so far) and target multiple architectures. This is because COAST is implemented as optimizer passes in the LLVM framework. Being able to operate on the LLVM IR means that the COAST passes can add fault tolerance on an instruction level, but a level that is still higher than architecture-specific assembly.

This research project does not aim to create any new replication procedures or rules, such as have been discussed before [1], [2]. Rather, we follow in the footsteps of others who have tested and proven various techniques and implement these rules in the form of an automated tool. Chielle et al. presented a number of different replication rules to follow to achieve the most fault coverage for the least overhead [17]. These options are reproduced in Figure 2.2. The COAST tool uses rules D2 and C3, C4, and C6, with the option to enable C5.

The code in Figure 2.3 is a simple example of the effect of applying these replication rules. The left half of the figure shows the original assembly code, without any COAST modification.
Table 18.3  Techniques and rules

<table>
<thead>
<tr>
<th>Global rules (valid for all techniques)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>G1</td>
<td>Each register used in the program has a spare register assigned as replica</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Duplication rules (performing the same operation on the register’s replica)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>D1</td>
<td>All instructions except branches</td>
</tr>
<tr>
<td>D2</td>
<td>All instructions, except branches and stores</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Checking rules (compare the value of a register with its replica)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>Before each read on the register (except load/store and branch instructions)</td>
</tr>
<tr>
<td>C2</td>
<td>After each write on the register</td>
</tr>
<tr>
<td>C3</td>
<td>Before loads, the register that contains the address</td>
</tr>
<tr>
<td>C4</td>
<td>Before stores, the register that contains the datum</td>
</tr>
<tr>
<td>C5</td>
<td>Before stores, the register that contains the address</td>
</tr>
<tr>
<td>C6</td>
<td>Before branches</td>
</tr>
</tbody>
</table>

Figure 2.2: Replication rules from [17]

<table>
<thead>
<tr>
<th>run:</th>
</tr>
</thead>
<tbody>
<tr>
<td>i&lt;sub&gt;1&lt;/sub&gt; 1d r0 = i</td>
</tr>
<tr>
<td>i&lt;sub&gt;2&lt;/sub&gt; r1 = sub r0, 1</td>
</tr>
<tr>
<td>i&lt;sub&gt;3&lt;/sub&gt; r2 = cmp r1, 0</td>
</tr>
<tr>
<td>i&lt;sub&gt;4&lt;/sub&gt; br neq r1 run</td>
</tr>
</tbody>
</table>

(a) Original code

<table>
<thead>
<tr>
<th>run:</th>
</tr>
</thead>
<tbody>
<tr>
<td>i&lt;sub&gt;1&lt;/sub&gt;&lt;sub&gt;a&lt;/sub&gt; 1d r0 = i</td>
</tr>
<tr>
<td>i&lt;sub&gt;1&lt;/sub&gt;&lt;sub&gt;b&lt;/sub&gt; 1d r10 = i_copy</td>
</tr>
<tr>
<td>i&lt;sub&gt;1&lt;/sub&gt;&lt;sub&gt;c&lt;/sub&gt; 1d r20 = i_copy2</td>
</tr>
<tr>
<td>i&lt;sub&gt;2&lt;/sub&gt;&lt;sub&gt;a&lt;/sub&gt; r1 = sub r0, 1</td>
</tr>
<tr>
<td>i&lt;sub&gt;2&lt;/sub&gt;&lt;sub&gt;b&lt;/sub&gt; r11 = sub r10, 1</td>
</tr>
<tr>
<td>i&lt;sub&gt;2&lt;/sub&gt;&lt;sub&gt;c&lt;/sub&gt; r21 = sub r20, 1</td>
</tr>
<tr>
<td>i&lt;sub&gt;3&lt;/sub&gt;&lt;sub&gt;a&lt;/sub&gt; r2 = cmp r1, 0</td>
</tr>
<tr>
<td>i&lt;sub&gt;3&lt;/sub&gt;&lt;sub&gt;b&lt;/sub&gt; r12 = cmp r11, 0</td>
</tr>
<tr>
<td>i&lt;sub&gt;3&lt;/sub&gt;&lt;sub&gt;c&lt;/sub&gt; r22 = cmp r21, 0</td>
</tr>
<tr>
<td>i&lt;sub&gt;3&lt;/sub&gt;&lt;sub&gt;d&lt;/sub&gt; r3 = cmp r2, r12</td>
</tr>
<tr>
<td>i&lt;sub&gt;3&lt;/sub&gt;&lt;sub&gt;e&lt;/sub&gt; r4 = select r3, r2, r22</td>
</tr>
<tr>
<td>i&lt;sub&gt;4&lt;/sub&gt; br neq r4 run</td>
</tr>
</tbody>
</table>

(b) TMR code

Figure 2.3: Code before and after TMR mitigation, based on figure from [5]

The right half shows what the assembly might look like after the code is modified by COAST. Each of the instructions are numbered on the left side, with subdivisions on the right to show how they are related. The extra copies of <i>i<sub>1</sub></i>, <i>i<sub>2</sub></i>, and <i>i<sub>3</sub></i> are created by following rule D2. Instructions <i>i<sub>1</sub>v</i> and <i>i<sub>2</sub>v</i> are inserted before the branch to follow rule C6. Rule C6 also includes checking before function calls.
2.4.1 Testing Additional Applications of COAST

Previous work with COAST laid the groundwork for this automated tool and tested its applicability to a simple platform, the MSP430. The work was essential in proving that implementing protection rules during program compilation was possible, and that the effect of fault tolerance was significant enough to be worthwhile. The major limitations of this work was that it was applied to only a single architecture and a very limited set of benchmarks [4]. We believe that a lot of the value of COAST comes in its wide applicability to many platforms and benchmarks. Extending the supported platforms and number of benchmarks tested is one of the main goals of the work presented in this thesis. Another major contribution is using COAST to add software protection to a real-time operating system.

2.5 Fault Injection – Related Work

A common way to test the fault tolerance of a system without the use of high-energy radiation is to artificially cause faults in the system. This is known as fault injection. Fault injection can be done either using a physical device as the target, or done entirely in simulation.

Fault injection can be used for a number of different specific purposes. It can be used to determine how good a system is at detecting and recovering from errors, to measure how effective a particular fault mitigation scheme is, and to get valuable feedback about the mitigation scheme during the development process [27]. It can be used to target values in the processor cache and main memory (as they are the largest target of radiation and often the most susceptible) and to prevent upsets from causing permanent damage to actual hardware [28]. And fault injection can be used to assess the fault detection abilities of commercial off-the-shelf microcontrollers, as these are increasingly being used in safety-critical applications [29].

Chapter 6 contains a discussion of a protection scheme for a real-time operating system. Because normal radiation test facilities were not available during the time that this part of the thesis work was being conducted, it was important for us to have a way to validate this brand new protection scheme. We desired a fault injection framework that could simulate the randomly-distributed errors that occur in radiation beam testing, as well as a support bare metal benchmarks.
Some of the interesting projects on fault injection in the last few years include FSFI, a fault injection framework for SPARC architectures built on SAM, an open-source system simulator [27]. Heing-Becker et al. created FITIn, a system built on the popular Valgrind tool, to inject faults from the abstraction level of C code variables [30]. The DrSEUS project built a fault injection framework around Simics, an advanced simulator, to get a high level of visibility on the different parts of the system [31]. Both FAIL* [32] and VarEMU [33] use QEMU as an emulation back end for fault injection.

Chapter 5 gives the details of a fault injection framework that we built to achieve our testing goals. This framework is built on the QEMU emulator as its emulation back end. Previous works have used other emulation back ends, or had to modify the QEMU source code to work. Of the papers on fault injection we surveyed, none specifically targeted bare metal applications. Most other fault injectors do either exhaustive or targeted fault injection, rather than randomly distributing the faults.
CHAPTER 3. EXTENDING COAST TO NEW PLATFORMS AND BENCHMARKS

One of the main contributions presented in this thesis is the validation of the COAST tool across multiple target architectures and benchmarks. In creating this tool, a strong motivation for building it as a compiler pass was to enable this kind of portability. As previous work with COAST had been mostly limited to variants of the MSP430 architecture, it was important for us to make sure COAST came through on its promises.

Over a couple of years we were able to run two experiments at the Los Alamos Neutron Science Center (LANSCE). The first was to test COAST on multiple platforms, and the second was to investigate what characteristics of benchmarks made them more apt for protection for by COAST. This chapter will cover the first test, and Chapter 4 will cover the second.

3.1 Extending Support for ARM and RISC-V

In order to test COAST’s performance on different platforms, it was necessary to extend COAST to support additional toolchains. In this work we extended COAST to support the Freedom SDK\(^1\) for RISC-V, as well as the Xilinx SDK\(^2\) for Xilinx’s ARM SoC parts. The approach taken for both toolchains was very similar, and thus we believe that users could easily extend COAST to support additional architectures and boards. While the full details of these modifications are beyond the scope of this discussion, a brief description of the process is included.

Figure 3.1 illustrates the build flow for both the Freedom RISC-V and Xilinx ARM platforms. Both of these toolchains utilize the GCC compiler out-of-the-box. However, COAST relies upon the LLVM compiler framework [34], an alternative to GCC. Rather than migrating the build of all platform files to LLVM, which would require significant user effort, our approach is to only compile the user’s core program using the LLVM-based COAST compiler. As shown in the diagram, the platform support files, which include the Board Support Package (BSP), continue to

\(^1\)https://github.com/sifive/freedom-e-sdk

\(^2\)https://www.xilinx.com/products/design-tools/embedded-software/sdk.html
be compiled using the existing toolchain compiler. The binaries from both flows are then linked, again using the existing toolchain linker. For the Freedom RISC-V and Xilinx ARM flows, this means that the platform files can continue to be compiled using the GCC toolchain, and the final binary will still be produced by GCC.

This approach minimizes the user effort required to utilize the COAST tool on new platforms. The main responsibility placed on the user is determining the flags that must be passed to COAST (both the Clang front-end, and the LLVM optimizer) to compile the core user program. For example, for the RISC-V platform these consist of passing "-m32 --target=riscv32" to Clang and "-march=rv32imac -mabi=ilp32 -mcpu=medany" to LLVM. Thus, any CPU architecture supported by LLVM should be targetable by COAST, with minimal user effort, provided that both the LLVM and GCC ports use compatible ABIs.

As part of this process, we developed extensible Makefiles for both of these toolchains in order to implement the build process shown in Figure 3.1. Users of COAST can support new platforms and boards by making minimal modifications to these Makefiles to connect into their existing toolchains. The source code for these makefile, and the instructions for how to use them, can be found at the project website.

### 3.2 Verifying Correctness

The LLVM passes which implement COAST perform a number of code mutations in order to add the protection and synchronization instructions. We must make sure that none of these mutations affect the correct functionality of the user’s code. We have used a couple of different methods to perform automated testing to verify COAST correctness.
The first iteration of automated testing used Buildbot\(^3\) to run several self-verifying benchmarks that had been protected by COAST. This benchmark suite consists of matrix multiply, quick-sort, CRC, AES, FFT (4 variants), \texttt{llvm-stress}, MiBench \cite{35} (6 programs), CHStone \cite{36} (12 programs), and CoreMark \cite{37} (2 configurations). We also have unit tests designed to exercise very particular use cases of the protection algorithms. Most of the unit tests were created when a specific failure mode of COAST was observed. They were useful in debugging and fixing the issue, and have proved helpful ensuring that the same issues don’t become problems again. A few other unit tests were created to test a particular feature of a language, either C or C++. All together, this is just over 30 benchmarks to test against, providing a good spread of algorithm types and code sizes.

We have recently updated the automated testing to be a part of the build flow whenever changes are made to the COAST code. We use Travis Continuous Integration\(^4\) to run a suite of tests whenever our code repository is updated. This new suite of tests is very similar to what the Buildbot system used, and now also includes automated testing of FreeRTOS benchmarks (see Chapter 6).

### 3.3 Experimental Setup

In this section we demonstrate the effectiveness of COAST protection on a number of different platforms. The test was performed at the Los Alamos Neutron Science Center (LANSCE). The test configurations were designed to both demonstrate the applicability of our tool to several different platforms, as well as gain radiation sensitivity results for a few popular and emerging platforms.

#### 3.3.1 Devices Under Test

The following platforms were tested in our experiment:

\(^3\)
\[^{https://buildbot.net/}\]
\(^4\)
\[^{https://travis-ci.org/}\]
**SiFive HiFive board (RISC-V)** This board contains a SiFive Freedom E310, a 32-bit 320 MHz 130 nm RISCV processor, with a 16 KiB L1 instruction cache and a 16 KiB SRAM scratch-pad (non-ECC). In our benchmarks, all data was contained within this scratchpad.

**PYNQ-Z1 board (A9)** This platform is based on the Xilinx ZYNQ XC7Z020 FPGA, which contains an embedded 2-core 32-bit 667 MHz 28 nm ARM A9 processor. There is a 32 KiB instruction and 32 KiB data cache per core (non-ECC). The FPGA fabric was not utilized or tested, and the benchmark ran on only one of the ARM cores.

**AVNET Ultra96 board (A53)** This platform contains a Xilinx Zynq UltraScale+ MPSoC ZU3EG FPGA, produced using the TSMC 16FinFET+ technology, and contains an embedded 4-core 64-bit 1.5 GHz ARM A53 processor. There is a 32 KiB instruction and 32 KiB data cache per core (ECC), with a 1 MiB L2 cache. The FPGA fabric was not utilized or tested, and the benchmark ran on only one of the processor cores.

All platforms were configured as a bare metal system, with only essential board support package (BSP) software. Two RISC-V boards and two PYNQ-Z1 boards were used in the beam test; however, only one Ultra96 board was used, as the JTAG design of the board prevents two boards from being configured from a single computer.

The experiment was conducted in the 30R flight path at LANSCE, as shown in Figure 3.2. The boards were spread from a distance of 81 cm (Hercules 5F board) to 99 cm (2nd RISC-V board) 83cm from the detector. This results in a 10.7–12.9% attenuation in flux versus the measurements taken at the detector, which was taken into account in our results.

The A9 and A53 platforms contain external DRAM chips. We did our best to ensure the DRAM chips were located outside the 2 in diameter neutron beam, as shown in Figure 3.3; however, the close proximity to the DUT meant that placement was barely outside of the beam radius. However, since DRAM is not as susceptible to neutron-based upsets, we expect that it had minor impact on our results.

At the same time as this experiment, we also ran another experiment looking at the fault tolerance of the Texas Instruments’ Hercules platform. This board is mentioned in the experimental setup, but the results pertaining to this separate part will be discussed later in Section 3.5.
3.3.2 Benchmarks

Most configurations were tested using a matrix multiplication (MxM) benchmark, with the core computation shown in Figure 3.4. The COAST tool was configured to enable TMR on the matrix computation, which resulted in triplication of all operations and all variables, including both input and result matrices. The input matrices contained hard-coded random values. Upon completion of the multiplication, the verification code would compute an XOR of the entire result matrix, and compare against a hard-coded, predetermined golden value. The size of the matrices were chosen such that they filled the memory in the TMR configuration (scratchpad for the RISC-V platform, L1 cache for the ARM platforms), and identical matrix sizes were then used for the unmitigated configurations. The size of the square matrix was 19 for the RISC-V, and 30 for the A9 and A53.
Figure 3.3: Cross-section view of board placement in neutron beam. 2 in diameter beam cross-section is shown as a red circle, boards are shown as the large rectangles, with DUTs shown as the colored shaded small rectangles within the beam. The black-filled rectangles show placement of chips that we tried to place outside the beam area (power regulator for RISC-V board, and DRAM chips for the Xilinx boards).

```c
void __xMR matrix_multiply ( int mat1 [][][N],
   int mat2 [][][N], int mat_out [][][N]) {
   int i, j, k;

   for ( i = 0 ; i < N ; i++ ) {
      for ( j = 0 ; j < N ; j++ ) {
         unsigned long sum = 0;
         for ( k = 0 ; k < N ; k++ )
            sum += sum + mat1[i][k]*mat2[k][j];
         mat_out[i][j] = sum;
      }
   }
}
```

Figure 3.4: Matrix Multiplication kernel code showing in-code directive for triplication

On the ARM platforms, this benchmark was tested with the caches enabled and then disabled. Since the matrix sizes were chosen to fit in the L1 cache, the result differences should mainly reflect the cross-section difference between the processor core and the L1 cache.

The SHA256 benchmark (used only on the RISC-V platform due to experiment time constraints) is designed to compute the hash of a large input string. Once again, this was sized as large
as possible to fit in the scratchpad memory for the TMR configuration, and the same length, 4000 bytes, was used for the unmitigated version. For this benchmark, the golden value is an XOR hash of the results matrix. This approach to output validation aims to minimize cases where the golden value, which isn’t protected, can be corrupted while the program executes.

3.3.3 Test Methodology

During the test, the benchmarks were setup to repeatedly execute the same computation operation, periodically printing a heartbeat message via UART. This message was actively monitored by a separate computer that was set outside the path of the neutron beam. If the computed value did not match the golden value, the program would immediately print an error message to the monitoring computer. In these cases, the controlling system would power-cycle the board and reprogram the software. Other events also triggered a reprogramming, including a malformed output, or a heartbeat timeout. Although the A9 and A53 parts contain dual- and quad-core processors respectively, the provided results are for the benchmark executing solely on one core.

Each benchmark was compiled and tested using an original unmitigated version, and a TMR’d version produced by COAST. In the following sections, the reader may notice that the TMR’d versions received more time in front of the radiation beam. This is because it is comparatively more difficult to cause upsets with these benchmark configurations.

To create the TMR’d version of each benchmark, we employed the -TMR option of COAST. This instructs the compiler pass to insert voter operations at branch points in the program. We also enable the -countErrors option, which allows for enhanced voter code that counts the number of corrected faults. Using this synchronization counter does introduce some extra overhead in run time and code size. While this extra overhead would not be desirable in a deployed system, it does provide useful data for our experiment. The synchronization counter value is part of the heartbeat message, so when faults are detected, they also trigger a power cycle of the board.
Table 3.1: Neutron beam test results

<table>
<thead>
<tr>
<th>Configuration (Board/Bench., Options)</th>
<th>Fluence (n/cm²)</th>
<th>Faults (TMR Fixed)</th>
<th>Errors (SDC)</th>
<th>Hangs</th>
<th>Invalid Status</th>
<th>Code Size (KB)</th>
<th>Run-time (ms)</th>
<th>Cross Section (cm²)</th>
<th>MWTF</th>
</tr>
</thead>
<tbody>
<tr>
<td>RISC-V</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MxM, Unmitigated</td>
<td>3.9 × 10¹⁰</td>
<td>N/A</td>
<td>80</td>
<td>58</td>
<td>101</td>
<td>104</td>
<td>5.97</td>
<td>2.0 × 10⁻⁹</td>
<td>-</td>
</tr>
<tr>
<td>MxM, TMR</td>
<td>1.6 × 10¹¹</td>
<td>700</td>
<td>3</td>
<td>189</td>
<td>375</td>
<td>110</td>
<td>17.50</td>
<td>1.9 × 10⁻¹¹</td>
<td>-</td>
</tr>
<tr>
<td>Change</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.06x 2.9x</td>
<td>105.3x 35.9x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHA256, Unmitigated</td>
<td>6.6 × 10¹⁰</td>
<td>N/A</td>
<td>132</td>
<td>91</td>
<td>146</td>
<td>105</td>
<td>1.34</td>
<td>2.0 × 10⁻⁹</td>
<td>-</td>
</tr>
<tr>
<td>SHA256, TMR</td>
<td>8.7 × 10¹⁰</td>
<td>570</td>
<td>23</td>
<td>106</td>
<td>267</td>
<td>122</td>
<td>4.99</td>
<td>2.6 × 10⁻¹⁰</td>
<td>-</td>
</tr>
<tr>
<td>Change</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.16x 3.7x</td>
<td>7.7x 2.0x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARM A9 (PYNQ-Z1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MxM, Unmitigated</td>
<td>3.7 × 10¹⁰</td>
<td>N/A</td>
<td>28</td>
<td>3</td>
<td>9</td>
<td>206</td>
<td>0.211</td>
<td>7.6 × 10⁻¹⁰</td>
<td>-</td>
</tr>
<tr>
<td>MxM, TMR</td>
<td>8.3 × 10¹⁰</td>
<td>163</td>
<td>2</td>
<td>165</td>
<td>11</td>
<td>239</td>
<td>0.666</td>
<td>2.4 × 10⁻¹¹</td>
<td>-</td>
</tr>
<tr>
<td>Change</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.16x 3.2x</td>
<td>31.7x 10.0x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MxM, Unmit., NoCache</td>
<td>8.3 × 10¹⁰</td>
<td>N/A</td>
<td>13</td>
<td>14</td>
<td>37</td>
<td>206</td>
<td>5.33</td>
<td>1.6 × 10⁻¹⁰</td>
<td>-</td>
</tr>
<tr>
<td>MxM, TMR, NoCache</td>
<td>8.1 × 10¹⁰</td>
<td>26</td>
<td>3</td>
<td>19</td>
<td>12</td>
<td>239</td>
<td>18.80</td>
<td>3.7 × 10⁻¹¹</td>
<td>-</td>
</tr>
<tr>
<td>Change</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.16x 3.5x</td>
<td>4.3x 1.2x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARM A53 (Ultra96)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MxM, Unmitigated</td>
<td>1.8 × 10¹⁰</td>
<td>N/A</td>
<td>0</td>
<td>11</td>
<td>4</td>
<td>240</td>
<td>1.02</td>
<td><strong>5.7 × 10⁻¹¹</strong></td>
<td>-</td>
</tr>
<tr>
<td>MxM, TMR</td>
<td>2.9 × 10¹⁰</td>
<td>0</td>
<td>0</td>
<td>16</td>
<td>11</td>
<td>256</td>
<td>3.05</td>
<td><strong>3.4 × 10⁻¹¹</strong></td>
<td>-</td>
</tr>
<tr>
<td>Change</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.07x 4.0x</td>
<td></td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>MxM, Unmit., NoCache</td>
<td>3.9 × 10¹⁰</td>
<td>N/A</td>
<td>1</td>
<td>19</td>
<td>17</td>
<td>240</td>
<td>54.6</td>
<td>2.6 × 10⁻¹¹</td>
<td>-</td>
</tr>
<tr>
<td>MxM, TMR, NoCache</td>
<td>4.6 × 10¹⁰</td>
<td>1</td>
<td>1</td>
<td>24</td>
<td>13</td>
<td>256</td>
<td>209.0</td>
<td>2.2 × 10⁻¹¹</td>
<td>-</td>
</tr>
<tr>
<td>Change</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.07x 3.8x</td>
<td></td>
<td></td>
<td>-</td>
</tr>
</tbody>
</table>

**No errors observed, so this is calculated given one error (assuming the worst-case, where an error could be observed in the very next instant of the experiment).**

### 3.3.4 Experimental Results

The full results of the neutron beam test are provided in Table 3.1. The first column lists the benchmark and protection configuration. The next column lists the total Fluence received for each benchmark. To compute the fluence, we used flux measurement logs from LANSCE, and correlated the timestamps in these logs with the timestamps for when each benchmark was running.

The next set of columns list the different abnormal statuses encountered during repeated benchmark execution. The Faults column lists the number of times the TMR voters in the code encountered errors. The Errors column lists the number of times the SDC voters encountered errors. The Hangs column lists the number of times the benchmark hung. The Invalid Status column lists the number of times the benchmark invalid status was encountered. The Code Size column lists the size of the code in kilobytes. The Run-time column lists the run-time of the benchmark in milliseconds. The Cross Section column lists the cross-sectional area of the benchmark in square centimeters. The MWTF column lists the mean weighted time to failure in seconds.

---

*https://coast-compiler.readthedocs.io/en/latest/passes.html#command-line-parameters*
detected and corrected a fault. *Errors* are the number of times the benchmark computed a result which did not match the golden value. This is also referred to as silent data corruption (SDC), and is the main type of error that COAST is intended to protect against. A *Hang* was recorded when the benchmark heartbeat stopped responding for a significant amount of time (about 10x the expected heartbeat interval). An *Invalid Status* was recorded any time the UART message from the benchmark did not match the expected regular expression format. Any of these unsuccessful runs triggered a reset of the board.

The columns *Code Size* and *Runtime* are for comparing the overhead required for COAST protection against the original version of the benchmark. *Code Size* is the size of the compiled ELF file, measured in KB.

The column *Cross Section* measures the error rate according to Equation 3.1, which captures the likelihood of an error occurring in a given radiation dose.

\[
\text{Cross Section} = \frac{\text{Errors (SDC)}}{\text{Fluence}} 
\]  
(3.1)
Along with cross-section, we have the indicator *Mean Work To Failure* (MWTF) that puts cross-section in the context of the run time overhead. In other words, benchmarks which run longer have more time during which they can be upset. We use the MWTF metric to make fair comparisons between benchmark configurations that take into account this difference in run time. The equation for calculating MWTF is given by Equation 3.2.

\[
\text{MWTF} = \frac{\text{amount of work completed}}{\text{number of errors encountered}}
\]  

(3.2)

The tables report the *difference* in MWTF, rather than absolute values. As it is difficult to calculate the “amount of work completed”, we compute the relative MWTF as the *cross section change* divided by the *run time change*.

One may notice that the sum of the *Faults* and *Errors* columns for the TMR’d code is much greater than the *Errors* of the Unmitigated code. This is expected, as the COAST TMR process triplicates almost all operations and program data. This makes the program roughly 3x more susceptible to single event upsets. In addition, one should note that the TMR configuration typically was tested for longer durations in the beam and accumulated a greater fluence. This was done to increase the statistical significance of errors, which occur relatively infrequently in the TMR’d version. This increase is evident in Figure 3.5, where the cross-section of faults in the TMR’d configurations (red squares) is noticeable larger than the cross-section of errors in the unmitigated designs. Accounting for the 95% confidence intervals shown in Figure 3.5, the increase is calculated as 1.6x–3.0x, 2.5x–4.2x, 0.9x–8.9x, and 0.3x–10.7x, for the first four configurations in the figure, respectively.

### 3.4 Analysis of Test Results

Based on the collected data, there are a number of significant trends. Firstly, similar to our previous radiation testing [5], we have observed that COAST provides a significant improvement to cross-section, with a reduction of 4.3x–105.3x, depending on the platform and configuration. However, this benefit comes at a significant cost: run time is increased by 2.9–4.0x. This increase is expected, as triplication of all instructions, plus voting instructions, results in a run time that is usually greater than 3x.
We did notice that the RISC-V MxM benchmark had a run time increase of 2.9x. Upon inspecting the assembly code in detail, we discovered that the compiler had vectorized some of the triplicated instructions, resulting in a run time increase of less than 3x. Another factor that could affect run time overhead is instruction reordering, or out-of-order execution. There is quite a lot of instruction-level parallelism to take advantage of in COAST-protected code.

Across the configurations, the MWTF increase ranges from 1.2x to 35.9x. Since this range is quite dramatic, we provide some analysis of what causes the difference in benefit between the different platforms and configurations.

3.4.1 Variation between benchmarks

On the RISC-V platform, we tested two different benchmarks and saw noticeably different results between the two. Further study after the radiation test revealed a significant difference in the way that COAST treats the two algorithms. By default, COAST uses the same synchronization rules as presented by Chielle et al. in [1], [13], [17], [26], and will synchronize data values on control-flow branches as well as memory stores.

Depending on the benchmark, there may be a significant variation in the number of synchronization points. For example, the MxM benchmark contains only three synchronization points, as the core compute algorithm is contained in a small portion of code, triply nested for loops. To contrast, the SHA benchmark, which does not benefit as much from COAST protection, contains 98 synchronization points. These synchronization points, by nature, contain single-points of failure, as triplicated values are aggregated and voted on. With more synchronization in the code, it is not surprising that the TMR protection does not provide as much benefit.

After further investigation into the synchronization characteristics of the SHA benchmark, we updated COAST to no longer synchronize before storing data. Eliminating some of the synchronization points helped to lower the possibility of single points of failure. However, synchronizing on data stores can still be enabled using a command line flag\(^6\).

Another difference is the nature of the computation and data access patterns between the benchmarks. The SHA benchmark is a hashing algorithm, which sweeps through the input data in a streaming fashion. In contrast, in matrix multiplication, each element in the input matrices

are accessed multiple times throughout the multiplication process. This may make M\textsuperscript{x}M more sensitive to upsets in the input data memory, compared to say, upsets in the processor pipeline. The results, as discussed in the next subsections, suggest that COAST is more effective at correcting upsets in the memory than other locations. These observations of variations between benchmarks form the primary motivation for the next radiation test, discussed in Chapter 4.

3.4.2 Variation between platforms

There are also noticeable differences in the matrix multiply benchmark results running on the RISC-V vs PYNQ platforms. It is the same benchmark, though the matrix sizes differ. There are significant platform differences that might come into play when influencing fault coverage. The primary contributor is likely the fact that on the RISC-V platform, the main memory is located on-chip and is susceptible to radiation upset, whereas the PYNQ and Ultra96 platforms have an external DRAM located outside of the beam radius. In addition, other factors likely contribute to this difference; for example, the RISC-V processor has more general purpose registers than the A9, and may have a significantly simpler processor pipeline.

Finally, we were not able to upset the A53 processor beyond a few single events, despite a week of testing. This validates previous testing that shows that, even though the A53 has a non-zero upset cross-section, the amount of silent data corruption is near zero, likely due to the extensive use of error correcting codes in the memories of this system [38].

3.4.3 Variation between cache configuration

On the PYNQ platform we tested configurations with the caches enabled and disabled. Since the main memory for the PYNQ is located in off-chip DRAM, we expect that when the caches are disabled, most upsets will be located within the processor pipeline itself (register file, functional units, etc.).

The results show that when caches are disabled, COAST still provides significant cross-section reduction, which demonstrates that COAST can help protect against upsets in the processor core. However, the benefit (4.3x vs 31.7x) is not nearly as large as when upsets are frequently occurring in the memory. When accounting for run time, the MWTF is only 1.2x.
The results suggest that COAST is not as effective at correcting faults that occur in the processor pipeline. This is likely for two reasons: first, the synchronization voter code, that contains most of the single points of failure, often access recent values that would be located in the register file. Secondly, when processor core elements are affected, the result can go far beyond simple data corruption: the program counter, stack pointer, TLB, or other special-purpose elements can be affected that cannot be corrected by the simple data-replication provided by COAST.

3.5 Hercules – Hardware-Assisted Fault Tolerance

In addition to the the platforms tested in the experiment discussed in Section 3.3, we also performed an experiment with the Texas Instruments Hercules platform. We were interested in understanding how the effectiveness of our software-only approach compares to a hardware-assisted approach. The Hercules platform has dual ARM Cortex-R5 cores which run in lock-step. To take advantage of this feature, we modified a CRC benchmark to roll back and redo portions of computation when a mismatch between cores was reported. We found that this approach offered similar radiation protection to our software-only approach, with some notable advantages and disadvantages discussed below.

3.5.1 Experimental Setup

To make use of the hardware features of the Hercules, we employ a form of software checkpointing. This technique involves periodically saving the state of the program during execution so it can be restored if an error is detected. Using the lock-step feature of the Hercules platform, we are able to roll-back immediately after a fault is detected. Similar checkpointing approaches have been used recently; for example, recent work by Oliveira et al. [39] demonstrated checkpointing methods on a Xilinx ARM A9 platform, and measured a fault coverage rate of 78%.

In our test we used a Hercules 5F launchpad board. This board contains a TMS570LC4357 chip, which has two 32-bit ARM Cortex-R5F cores that are configured to run in lock-step at 300 MHz. It has a 32 KiB instruction and 32 KiB data cache with ECC. The Hercules part has a lock-step mode that allows both Cortex-R5F cores to execute the same instructions, separated in

---

8 https://www.ti.com/product/TMS570LC4357
Table 3.2: Radiation Test Results – Hercules

<table>
<thead>
<tr>
<th>Configuration (Benchmark, Options)</th>
<th>Fluence (n/cm²)</th>
<th>Faults (C.P. Fixed)</th>
<th>Errors (SDC)</th>
<th>Hangs</th>
<th>Invalid Status</th>
<th>Code Size (KB)</th>
<th>Run-time (s)</th>
<th>Cross Section (cm²)</th>
<th>MWTF</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRC - no checkpointing</td>
<td>4.9 x 10¹⁰</td>
<td>N/A</td>
<td>22</td>
<td>59</td>
<td>1</td>
<td>244</td>
<td>11.72</td>
<td>4.5 x 10⁻¹⁰</td>
<td>-</td>
</tr>
<tr>
<td>CRC - checkpointing</td>
<td>1.5 x 10¹¹</td>
<td>62</td>
<td>0</td>
<td>174</td>
<td>1</td>
<td>244</td>
<td>23.34</td>
<td>6.5 x 10⁻¹²</td>
<td>-</td>
</tr>
<tr>
<td>Change</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.0x ↑ 1.99x</td>
<td></td>
<td>↓ 69.2x ↑ 34.8x</td>
<td></td>
</tr>
</tbody>
</table>

time by 2 cycles. A CPU Compare Module (CCM) checks the output of the processor core bus for any variation between the cores. This gives real-time error detection for almost any type of error.

In our experiment we implemented algorithm-level checkpointing and rollback. The benchmark we ran was a cyclical redundancy checker (CRC) which calculated the checksum of a stream of random numbers, generated from the same known seed every time, which was compared against the known golden value. Any execution run where the calculated CRC sum did not equal the golden value was considered an error. We used the Hercules part’s built-in error signaling module (ESM) to check for an error in the CCM. Any time an error was detected, the program would roll back to a previous checkpoint. We consider the checkpoints saved in memory to be safe, because all of the memory in the Hercules part is protected with ECC. For each iteration of the CRC, there were 4096 checkpoints saved. Each checkpoint saved would overwrite the previous one, and contained the current value of the CRC sum, the iteration count through the stream of numbers, and the value of the random number generator’s seed.

It should be noted that our version of checkpointing did not save the state of the system (registers, etc.) but rather relied on modifying the user code to insert checkpoint saving of key variables and roll-back upon error detection.

3.5.2 Results

This system was tested at LANSCE simultaneously with our COAST experiments described in Section 3.3. Table 3.2 shows results in a similar format to those presented earlier. As can be seen, our hardware-assisted approach provides a significant MWTF benefit of 34.8x, comparable to the best-case reliability of our COAST platform configurations. However, the checkpointing introduced a run time penalty to periodically saving the state, and resulted in a 2x increase to run
time. It’s entirely possible that the checkpointing frequency could be reduced to obtain better run time performance.

3.6 Conclusion

This section has demonstrated how our COAST tool, which provides automated fault-tolerant protection to user programs, can be effectively deployed on a wide range of processing platforms. The produced software executables are more tolerant of single event upsets, making COTS platforms more attractive for processing in high radiation environments. The results from the neutron beam test show that COAST provides significant increase to MWTF across a wide range of platforms and configurations.
CHAPTER 4. UNDERSTANDING BENCHMARK PROPERTIES

The results from the 2018 radiation test showed that variations in benchmark properties can lead to noticeably different changes in error cross-section, as noted in Section 3.4. In this section we aim to explore and understand what characteristics of particular benchmarks make them more apt for protection through data and instruction replication. Rather than varying the platform, as we did in the previous test, we vary the program.

Performing radiation testing on multiple benchmarks is the first step towards being able to make predictions for new benchmarks. Our goal is to get an idea of whether or not it would be beneficial to apply software protection techniques to a benchmark we have not tested before. While we would ideally apply our tool to tens or hundreds of different C programs and build an accurate model, this is not feasible. Limited access to radiation testing facilities combined with the relatively low frequency of errors means we must restrict the number of benchmarks to a small sample set.

The main contributions of this section are, experimental testing of multiple C programs at the Los Alamos Neutron Science Center (LANSCE), and an analysis of the results from the experiment. The platform under study, the 32-bit Xilinx Zynq ARM Cortex-A9, had 8 benchmarks tested on it. Across all these benchmarks, we saw reduction in cross-sections from 1.6x–54x. Using the experimental results, we draw insight into which characteristics of benchmarks make them most conducive to software-based fault protection.

4.1 Experimental Setup

This test followed the same testing methodology as in the previous year’s test, described in Section 3.3.3.
4.1.1 Device Under Test

The DUT for this experiment was the same ZYNQ part used in the previous year’s test, as described in Section 3.3.1. The 30L flight path (Ice House) was used for this experiment, and the three ZYNQ boards were placed 99, 101 and 106 cm from the neutron detector. These distances were accounted for when determining the fluence received by each board during the experiments. Each board was placed so the A9’s external DRAM chip was outside of the 2 in diameter neutron beam. Figure 4.1 provides a photo of the setup, which includes boards from several other experiments.

4.1.2 Benchmarks

We used eight different benchmarks in our test, as outlined below:
**crc32:** A 32-bit Cyclic Redundancy Check. This computes the hash of a randomly-generated stream of 0x400000 (4194304) 32-bit values.

**dijkstra:** From the MiBench test suite\(^1\), this finds the shortest path between a set of 100x100 vertices in a graph.

**matrixMultiply:** Matrix multiplication, tested with two sizes: *Fit L1*, where the matrices were sized to all fit in the L1 cache when triplicated (each matrix 30x30 words), and *Fit L2*, where they likewise fit in the L2 cache (120x120). The words in the matrices were 32 bits.

**nanojpeg:** A simple JPEG decoder\(^2\); input data is a JPEG image converted to a C array (11362 bytes).

**qsort:** Sort an array of 4000 3D coordinates by their distance from the origin. Tested in two configurations: *Library*, where we use the C standard library implementation of `qsort`, which is notably not protected by our tool; and *Custom*, which uses our own code for the sorting kernel, which allows protection to be enabled on the entire algorithm.

**sha256:** Computes the SHA-256 hash of a statically defined array of 4000 bytes.

### 4.1.3 Test Results

The results from this experiment are shown in Table 4.1 and summarized in Figure 4.2, which shows the cross-section for each of the benchmarks with 95% confidence error bars. As can be seen in the figure, COAST TMR protection reduces cross-section by 1.0x to 54x, indicating that the characteristics of the benchmark significantly influence the effectiveness of the fault mitigation.

When taking run time into consideration, it can be seen that while most benchmarks improved in MWTF (1.1x–43x), there were a couple that degraded in reliability (*nanojpeg* and *qsortLib*).

### 4.2 Analysis and Linear Regression

When we began this test, we hoped to use the data to construct a predictive model, a model which would allow us to predict what the fault coverage improvement of a program would be if we

---

\(^1\) [http://vhosts.eecs.umich.edu/mibench/index.html](http://vhosts.eecs.umich.edu/mibench/index.html)

\(^2\) based on [https://keyj.emphy.de/nanojpeg/](https://keyj.emphy.de/nanojpeg/)
were to protect it with COAST. After much analysis, we determined that there are too many factors at play, and we have too little data to develop an accurate predictive model. If we had access to radiation test facilities for weeks, and dozens of benchmarks to test, perhaps we could have created such a general-purpose model. Despite our limited data-set, through regression analysis we were able to draw some meaningful insights into how different benchmark characteristics can impact the effectiveness of automated fault tolerance.

### 4.2.1 Choosing Characteristics & Creating the Model

As part of the regression analysis, we examined many different benchmark characteristics. The intention of the model was to use measurable properties of each benchmark to predict the cross-section improvement that would be seen if the benchmark were tested in a radiation experiment. We looked at quite a few characteristics of benchmarks as potential predictors. These ranged from maximum resident memory size to usage statistics for the processor caches. Several of these either showed no meaningful correlation, or were redundant when considered with other properties. In the end, the characteristics we found to be most impactful were the following:

**Table 4.1: Neutron beam test results**

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Fluence (n/cm²)</th>
<th>Faults (TMR Fixed)</th>
<th>Errors (SDC)</th>
<th>Hangs/Invalid Status</th>
<th>Code Size (KB)</th>
<th>Runtime (ms)</th>
<th>Cross Section (cm²)</th>
<th>MWTF</th>
</tr>
</thead>
<tbody>
<tr>
<td>crc32, Unmit</td>
<td>2.41 × 10⁷</td>
<td>N/A</td>
<td>5</td>
<td>1/1</td>
<td>159</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>crc32, TMR</td>
<td>2.6 × 10⁸</td>
<td>20</td>
<td>0</td>
<td>11/1</td>
<td>191 ↑ 1.201x</td>
<td>1162 ↑ 1.241x</td>
<td><strong>3.84 × 10⁻⁹</strong> ↓ 53.99x ↑ 43.49x</td>
<td></td>
</tr>
<tr>
<td>dijkstra, Unmit</td>
<td>1.14 × 10⁸</td>
<td>N/A</td>
<td>0</td>
<td>76/2</td>
<td>171</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>dijkstra, TMR</td>
<td>6.25 × 10⁹</td>
<td>13</td>
<td>0</td>
<td>356/1</td>
<td>191 ↑ 1.117x</td>
<td>2414 ↑ 5.05x</td>
<td><strong>1.6 × 10⁻¹⁰</strong> ↓ 5.51x ↑ 1.09x</td>
<td></td>
</tr>
<tr>
<td>MxM, Unmit, L2</td>
<td>1.23 × 10⁹</td>
<td>N/A</td>
<td>24</td>
<td>12/0</td>
<td>307</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>MxM, TMR, L2</td>
<td>4.97 × 10⁸</td>
<td>101</td>
<td>7</td>
<td>47/0</td>
<td>536 ↑ 1.75x</td>
<td>640 ↑ 3.02x</td>
<td>1.41 × 10⁻⁸ ↑ 13.85x ↑ 4.58x</td>
<td></td>
</tr>
<tr>
<td>MxM, Unmit, L1</td>
<td>8.06 × 10⁸</td>
<td>N/A</td>
<td>3</td>
<td>36/0</td>
<td>209</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>MxM, TMR, L1</td>
<td>1.14 × 10¹⁰</td>
<td>14</td>
<td>1</td>
<td>519/3</td>
<td>228 ↑ 1.09x</td>
<td>2897 ↑ 1.9x</td>
<td>8.8 × 10⁻¹¹ ↑ 42.28x ↑ 22.3x</td>
<td></td>
</tr>
<tr>
<td>nanojpeg, Unmit</td>
<td>5.85 × 10⁹</td>
<td>N/A</td>
<td>47</td>
<td>324/1</td>
<td>187</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>nanojpeg, TMR</td>
<td>7.27 × 10⁹</td>
<td>119</td>
<td>22</td>
<td>329/1</td>
<td>241 ↑ 1.29x</td>
<td>2503 ↑ 7.73x</td>
<td>3.02 × 10⁻⁹ ↓ 2.66x ↓ 2.91x</td>
<td></td>
</tr>
<tr>
<td>qsortLib, Unmit</td>
<td>8.62 × 10⁹</td>
<td>N/A</td>
<td>2</td>
<td>464/0</td>
<td>290</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>qsortLib, TMR</td>
<td>6.85 × 10⁹</td>
<td>13</td>
<td>0</td>
<td>333/2</td>
<td>429 ↑ 1.48x</td>
<td>189 ↑ 2.45x</td>
<td><strong>1.46 × 10⁻¹⁰</strong> ↓ 1.59x ↓ 1.54x</td>
<td></td>
</tr>
<tr>
<td>qsortCustom, Unmit</td>
<td>5.25 × 10⁹</td>
<td>N/A</td>
<td>10</td>
<td>255/0</td>
<td>290</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>qsortCustom, TMR</td>
<td>2.29 × 10¹⁰</td>
<td>22</td>
<td>0</td>
<td>1119/0</td>
<td>429 ↑ 1.48x</td>
<td>880 ↑ 3.18x</td>
<td><strong>4.37 × 10⁻¹¹</strong> ↓ 43.61x ↑ 13.73x</td>
<td></td>
</tr>
<tr>
<td>sha256, Unmit</td>
<td>5.21 × 10⁷</td>
<td>N/A</td>
<td>4</td>
<td>2/241</td>
<td>138</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>sha256, TMR</td>
<td>2.13 × 10⁸</td>
<td>30</td>
<td>2</td>
<td>10/0</td>
<td>215 ↑ 1.56x</td>
<td>57 ↑ 4.07x</td>
<td>9.37 × 10⁻⁹ ↓ 8.19x ↑ 1.95x</td>
<td></td>
</tr>
</tbody>
</table>

**No errors observed, so this is calculated given one error (assuming the worst-case, where an error could be observed on the next neutron).**
$x_1$: **Peak Heap Usage** (in KiB)

$x_2$: **Static Memory Size** Size of global variables in memory (.data and .bss sections counted, in KiB)

$x_3$: **Sync Points/s** How many times a synchronization voter was hit per second of program execution.

$x_4$: **Cross section** (cm$^2$) of unmitigated program, determined from our experimental data.

Peak heap usage was obtained using the dynamic analysis tool `massif`, from the `valgrind` tool suite. Static memory usage was determined by inspecting the program executables using the `readelf` utility and observing the sizes of the .data and .bss sections. To count the synchronization points, we modified our compiler pass with a new option to increment a global counter variable every time a synchronization point is encountered during execution. This number was then divided by the program execution time to get the number of synchronization points hit per second. For the cross section predictor, we used the results of the unprotected benchmark from the radiation test to determine how susceptible each benchmark was to radiation, before any protection schemes were applied.
To interpret the data in a meaningful way, we used multiple linear regression (MLR) to create a model, shown in Equation 4.1.

Cross Section Factor Improvement = $-0.44x_1 + 0.31x_2 + 2.2 \cdot 10^8x_3 + 1.1 \cdot 10^{-6}x_4 - 24.7$ \hfill (4.1)

This model estimates the factor decrease to cross-section provided by using the TMR pass of COAST. The model likely has limited portability to other benchmarks and architectures, but is useful in understanding the characteristics that tend towards a program having better fault tolerance when fault mitigation is applied. Since fault mitigation comes at a significant price (3–4x program slowdown), it’s likely that designers would only want to use it in cases where the reliability improvement is predicted to be significant. The negative coefficient on $x_1$ suggests that as heap usage increases, improvement in cross-section decrease will get smaller. The positive coefficient for $x_2$ suggests an opposite correlation for static memory sizes. Section 4.3 goes into more detail interpreting each of the predictors.

<table>
<thead>
<tr>
<th>Model</th>
<th>$R^2$</th>
<th>$R^2$ – delta</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full Model</td>
<td>0.88</td>
<td>N/A</td>
</tr>
<tr>
<td><strong>Characteristic Removed</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Heap Usage ($x_1$)</td>
<td>0.11</td>
<td>0.77</td>
</tr>
<tr>
<td>Static Memory Size ($x_2$)</td>
<td>0.25</td>
<td>0.63</td>
</tr>
<tr>
<td>Cross section ($x_3$)</td>
<td>0.44</td>
<td>0.44</td>
</tr>
<tr>
<td>Sync Points / second ($x_4$)</td>
<td>0.71</td>
<td>0.17</td>
</tr>
</tbody>
</table>

To ensure that using a linear model was appropriate for this problem, we examined the residual plots for each of the characteristics. These plots indicated that the characteristics effect was linear, making linear regression a valid method for modeling these relationships. If the residuals had not been linear, we would have needed to use a non-linear model, which would have significantly increased the complexity of modeling the effects of these characteristics.
Using the characteristics from the eight benchmarks mentioned in Section 4.1.2, the model described in Equation 4.1 has an $R^2$ value of 0.88, as shown in Table 4.2. The $R^2$ value is a measure of how well the line of the equation represents the underlying data.

To evaluate the impact that each characteristic by itself has on the full model, we removed each characteristic and re-ran the regression. These values are given in the third column of Table 4.2. In addition to the characteristics listed in the table, we explored several other metrics, such as cache hit rates and fault injection testing on the register file, but found no noticeable correlations.

4.3 Explaining the Benchmark Characteristics

**Heap Usage ($x_1$)** The most influential characteristic in our benchmark set was the amount of heap memory used. In our benchmark set, only a few used the heap, with *nanojpeg* and *qsort-Library* being the largest users. Since the Static Memory Size positively affected cross-section performance, we concluded that it wasn’t actually the memory usage itself that was the primary issue, but rather the calls to *malloc*, and the way that it manages heap memory. It seems that the more often *malloc* is called, the less effective COAST is at protecting the code. There are a couple reasons we expect this is the case: 1) since *malloc* is a library function, it cannot be protected by COAST; 2) even when the memory regions are passed back to the protected code, *malloc’d* regions have special header/footer metadata sections that COAST cannot synchronize (see Figure 4.3). If a fault occurs in any of these special regions, it is likely unrecoverable. Based on this, we believe it is best to avoid using dynamic memory allocation when wanting to perform software-based fault mitigation. As an example of other research that discourages using *malloc* in safety-critical applications, see Rule 5 of the JPL Coding Standard [40].

The reader may notice that the two benchmarks that used *malloc* are also the only two for which we measured a decrease in MWTF. Though we do not have enough data to call this a correlation, it is interesting to notice.

**Static Memory Sizes ($x_2$)** The positive coefficient on this predictor indicates that we expect fault tolerance improvement to increase as the amount of memory set aside for variables increases. In our test platform, the main memory consists of a large DRAM chip, which is outside of the beam path and naturally more resistant to radiation-based upsets than SRAM [42]. How-
ever, data in the processor caches is still highly susceptible to faults, and our previous test results (Section 3.4.3) indicate that COAST is better at protecting values that reside in caches [43]. As COAST is primarily intended to protect against faults in data memory, it is not surprising that this characteristic is one of the main predictors.

To be more precise, although this characteristic is called “static memory size”, it is actually the total size of the `.data` and `.bss` sections of the ELF file. During the development of our model, we tried different combinations of ELF sections and these two proved to be the best. These sections represent the majority of the data variables, besides those that are allocated from the `.heap` section, which is taken care of in the previous predictor. So we can say that the more data variables there are, the better able COAST is to protect the program.

Some of this behavior may be due to the cache configuration of the platform under test. In the previous year’s test discussed in Section 3.4, we tested the PYNQ board with and without caches enabled. The error rate with the caches enabled was noticeably higher than the error rate with the caches disabled. This means that the errors are more likely to occur in the caches that in main memory or the processor registers. So COAST is helpful in this case because the cross section is so much higher with the caches enabled that there’s plenty of room for improvement. It is possible that a different memory hierarchy configuration would have different behavior under a high radiation environment. When application designers are considering using COAST to pro-
vide fault tolerance, they will need to take these possible discrepancies into account in making a decision.

**Cross Section** \((x_3)\) Our model is designed to predict *cross-section improvement*; here we use the *baseline cross-section*, or the cross-section of the program without TMR protection, as one of the predictors. This predictor is used to take into account inherent differences between benchmarks. If the program was already inherently fault tolerant, there may be fewer opportunities for COAST to improve its cross-section. An example of this is seen in the *quicksort* benchmarks: our golden checking code ensures that the values were sorted correctly; however, it does not actually check that no bits were flipped. Thus, many faults could be naturally masked. While this may not be desirable for an actual sorting program, it would likely arise in other programs, such as machine learning algorithms which have been shown to be somewhat fault tolerant [44].

Because algorithmic differences can affect fault tolerance, we included the raw cross section numbers from the unmitigated benchmarks, to account for inherent differences in fault tolerance between algorithms that would be difficult to quantify any other way. The positive coefficient demonstrates that as cross-section numbers get *higher* (worse), the opportunities to improve the fault tolerance increase. Conversely, programs that have a lower inherent cross-section are more difficult to improve.

**Sync Points/s** \((x_4)\) This predictor is a count of the number of synchronization points hit in the TMR’d code during one second of execution time. Our results suggest that benchmarks that synchronize more often will see more benefit from our protection techniques, and the positive coefficient in the model agrees with this.

One thing to keep in mind is that it is possible to have too many sync points. Although synchronization allows the TMR’d code to detect/correct errors, it does introduce a potential single point of failure. The code that does the voting is vulnerable to upsets, which represents a failure mode that did not exist in the original, unmitigated version of the code. Analyzing these sync points would be very difficult, as it’s not as straightforward as simply measuring the memory usage as in other predictors. The sync points can vary distinctly in quantity, type (data store vs branch comparison), and placement. However, it appears that with the benchmarks tested, we did not exceed the ratio of normal code to synchronization code that would cause it to perform worse.
4.4 Conclusion

The results in this section indicate that when protecting an application, it is important to take into account memory usage patterns, and stay away from dynamic memory allocation if possible. One must also take into account the inherent fault tolerance of the application, and the frequency and placement of voters in the protected code.

From an application design point of view, knowing what characteristics lead to better fault tolerance should influence the design from the beginning. If developers are aware how to structure memory accesses and places for synchronization, they can code accordingly and help COAST provide better fault tolerance increase for their program.

In addition to our model, the results from the radiation test have shown once again that applying COAST protection to applications can provide significant fault coverage improvement to software applications.
CHAPTER 5. BUILDING A FRAMEWORK FOR FAULT INJECTION

In this chapter we will discuss the creation of a fault injection framework. This framework was created to facilitate testing COAST-protected programs, similar to the radiation tests mentioned in Chapters 3 and 4. The ability to test the fault tolerance of COAST-protected code, without a radiation test facility was crucial for the success of the work presented in Chapters 6 and 7. We were not able to schedule time with a test facility to test this latest work because the bulk of it was done during the COVID-19 pandemic.

Our framework for simulation-based fault injection is called PACIFIC – Platform for Active Injection of Faults In a Campaign. This framework is noteworthy because it:

- Leverages custom QEMU plugins, instead of requiring modifications to the QEMU source code.
- Supports fault injection on bare metal programs.
- Approximates radiation testing with randomly distributed faults.
- Is granular to the processor-cycle level.
- Supports targeting the processor cache(s).

5.1 Fault Injection – Background

Fault injection is used by many people to classify the failure modes of systems. To “inject a fault” can mean different things depending on the context, but it is basically flipping a bit somewhere in the system. Fault injection is an attractive method of finding these failure modes because it can be done in simulation. While actual radiation experiments usually yield a more accurate measurement of fault tolerance, use of radiation test facilities is expensive, complex, time-consuming, and hard to schedule. The use of fault injection to classify failure modes is a common alternative to radiation testing. For those who do perform radiation testing, fault injection can be used to complement and prepare for the testing experience. In addition to classifying failure modes, fault
injection is often used to help validate fault mitigation schemes. If a simulation is run before taking the mitigation scheme to the radiation test facility, the fault injection results can help uncover potential weak spots in the mitigation scheme.

There are a number of fault injection frameworks in existence, mentioned in detail in Section 2.5. We have created our own framework, PACIFIC, with the hope of expanding on the features and portability of these previous projects. Past works have been limited to a specific processor architecture, used proprietary software, or only worked on Linux systems. The PACIFIC framework is built on QEMU, so it can support any architecture that QEMU can emulate. QEMU is open-source, and so is PACIFIC, so it can be used by anyone interested in performing fault injection experiments. In contrast to projects like DrSEUs [31], which depend on the target processor running Linux, PACIFIC supports injection into bare metal applications as well as Linux-based applications.

Other works focus on targeted injection, under the premise that only certain bits in the processor and program are susceptible to upsets [45], [46]. In this work, we use a random injection approach, similar to how high-radiation particles can cause randomly-distributed single event effects. We specifically choose to target the processor caches because the bits in the caches represent a significant target for radiation-induced upsets [28], [47]. Our model is based on simulating the types of errors in caches that would be caused by randomly-distributed radiation, compared to most other models we have seen, which tend to focus on intentionally targeted injections.

5.2 PACIFIC – A Fault Injection Framework

The goal of the work presented in this chapter was to create a framework for fault injection on arbitrary programs. The work done in Chapters 6 and 7 was dependent on having such a framework available to validate fault coverage. Another motivation for the creation of this framework was to be able to measure the effectiveness our COAST tool at protecting programs, before we go to the effort of testing the protected code in an actual radiation beam.

The target architecture with this project is the Xilinx PYNQ board, which has a 32-bit, dual-core ARM Cortex-A9 processor. This is the only architecture we have tested, but because we use QEMU as the emulator, this setup is theoretically extensible to any architecture supported by QEMU. See Section 5.4 for how this could be done.
QEMU works by translating the guest binaries it executes into machine code that can run natively on the host machine. It can be hosted on anything that can run Linux, and supports a wide variety of guest processor architectures. There have been other projects which use QEMU for the emulation back end [32], [33], [48], [49], but these all require modifying the QEMU source code.

Figure 5.1: Fault Injection Framework – Use Case

The goal of the PACIFIC framework is to use fault injection to approximate the effects of radiation on a system, in particular the data errors that can occur in the processor caches. We also use PACIFIC to inject errors into the processor registers. Testing is done in the form of a fault injection “campaign”, where the user specifies 1) the executable to be tested, 2) the section to be targeted, and 3) the number of faults to inject. The campaign supervisor will manage the QEMU and GDB instances and inject the specified number of faults, randomly distributed across the bits in the desired section.

An “injection” is a run of the program under test where, before execution finishes, the processor is paused and a single bit changed before execution is allowed to continue. If execution of the program does not finish, there is a watchdog which will detect if the program has gone on too long so it can be forcibly ended. The different possible results are: success, error detected, fault corrected, invalid output, and timeout. This matches the format of results from our previous tests.

The results of the fault injection campaign are a set of detailed logs files which keep track of many aspects of each run of the benchmark. The log files are in JSON format for easy parsing. Analysis of these files give detailed introspection into the fault tolerance of the program under test,
and a user should be able to identify any limitations or single points of failure in the program. A generic usage pattern for PACIFIC is given in Figure 5.1.

PACIFIC uses a GDB debugger instance to change bits during program execution. It also uses QEMU as the emulation back end, with a user plugin connected to the QEMU instance. The plugin keeps a model of the processor cache state during execution so injections can be targeted at cache-resident memory addresses. Figure 5.2 gives a high-level overview of how the pieces of the PACIFIC framework fit together.

Figure 5.2: Framework Process Diagram – arrows represent communication over Unix sockets.

There are 3 distinct processes which run as part of PACIFIC: the supervisor, a QEMU instance, and a GDB instance. The different processes communicate with each other over sockets.

**Supervisor:** The supervisor, which is implemented in Python, is a persistent process and will not exit until the fault injection campaign is finished. The supervisor uses multiple threads to communicate with each of the other framework components and to parse the output of the running application. To ensure a clean system state each time a fault is injected, the supervisor will restart the QEMU and GDB instances after each injection.

**GDB instance:** We use a version of GDB compiled for the target architecture and which supports running Python scripts. This instance is responsible for flipping bits and inspecting the system state, at the request of the supervisor.
**QEMU instance:** We require QEMU v4.2, the version in which support for user plugins was introduced. The installation must be configured to support plugins. The bare metal application is loaded into the QEMU instance and execution is paused until the debugger connects.

**QEMU plugin:** When QEMU is instantiated, it is loaded with a user plugin. The plugin we have written keeps track of the current count of processor cycles since execution began. It also tracks each memory load/store to keep a dynamic model of the memory addresses that are cache-resident.

### 5.3 Framework Features

PACIFIC is intended to approximate radiation testing, and to observe the behavior of the system under test in the presence of data errors. There are a few features that facilitate specific experiments.

**Section Targets:** There are a few options for which “section” of memory to inject the fault into. The 2 main ones are cache and registers. To inject into the cache section means that the supervisor will pick a random word in one of the processor caches, and inject a fault into the memory location association with that cache word. To inject into registers means that the supervisor will select a random processor register in which to flip a bit. There are also some “legacy” options, which include the main sections in an ELF file\(^1\).

- init
- text
- rodata
- data
- bss
- stack
- heap

Other sections can be added if desired by modifying some of the Python files that implement the Supervisor. There is also a special section called memory that will randomly pick from one of the above sections for each injection.

\(^1\)https://en.wikipedia.org/wiki/Executable_and_Linkable_Format
Forcing an Error Condition: There may be times where it is necessary to more thoroughly investigate a particular failure mode. The supervisor provides a way to set a specific address to a specific value, and at a particular time in the execution. The supervisor can take as optional input an expression of the form "set [addr] = [value]", which follows GDB’s convention for setting values. The supervisor can also be told on what iteration of the campaign to execute this command; in other words, do this on the \( n \)-th injection. The user can also specify the number of cycles to wait after execution begins before injecting this fault.

5.3.1 Modeling Cache Upsets

Previous work and our own past testing suggests that bits in a cache are more likely to be upset than bits in main memory [28]. To that end, to get a more accurate idea of the effects COAST will have on a specific program, we make modeling errors in cache the primary focus of this framework.

While QEMU does not model cache contents and behavior out-of-the-box, it is possible to extend QEMU to model caches. Our project is not the first to do this; Dung et al. used QEMU v0.12 for cache emulation to do experiments on cache utilization [50]. However, this work is no longer in active development, and the structure of QEMU has changed significantly since that time.

Release version 4.2 of QEMU introduced support for user plugins, which is one of the main features that allows our framework to function \textit{without} changing the QEMU source code. We created a QEMU plugin which subscribes to instruction translation, and instruments each load and store with a call to functions which track which values are in the cache at any point in the execution. The cache plugin keeps a model in memory that represents each block in the caches. The tag bits are stored for each block along with a flag indicating if the block is valid or dirty. The model can handle different block sizes, number of rows, levels of cache associativity, and replacement/allocation policies. Listing 5.1 shows some implementation details of our plugin.

Keeping track of the addresses of memory currently in cache allows us to inject faults specifically into these memory addresses. The plugin also maintains a count of how many processor cycles have elapsed since the program began. After the specified number of cycles have passed, the supervisor will ask the plugin for what address is associated with a random word in the cache. Using this information is what allows PACIFIC to inject randomly-distributed faults into
the caches. Note that the plugin cannot change the state of the guest, so any faults injected will
have to be done using the GDB interface. The plugin communicates with the supervisor over a
TCP socket at specific times so the supervisor can coordinate with GDB which memory addresses
to change.

5.3.2 Accurate Injection Timings

Another advantage of creating and using a QEMU plugin is the ability to get deterministic,
cycle-based “sleeping” before injecting a fault. During the design process of the PACIFIC frame-
work, we discussed using the Python function \texttt{time.sleep} to wait a random time before injecting
a fault. Adding a time delay in this manner presents a number of challenges. First, because of the
way that calls to a system sleep function work, we could not guarantee that the amount of time we
requested to sleep for would be the amount of time it actually slept. This is an inherent limitation
of using Linux as the host system, which does not have real-time task requirements. The second
challenge, which follows from the first, is that system sleeping does not have very fine granularity.
It is likely that sleeping for 0.0001 seconds (100 microseconds) and 0.0002 seconds would yield
stopping in the same place, even though those numbers are theoretically separated by (at least) several hundred instructions.

To get around this, we take advantage of the ability of the QEMU plugin to subscribe to instruction execution events. Each time an instruction is executed, the plugin will increment a counter, and then check if that counter has met the desired number. This means we can get a much more uniform distribution of cycles slept, provided our random number generator gives a nicely uniform distribution. We use Python’s `random.uniform` function to pick a cycle number on which to inject the next fault. Even when injecting into registers, where we don’t need to know the cache contents, we can still use the plugin for keeping track of how many cycles have executed.

Of course, checking cycle counts at every instruction will add quite a bit of overhead to the execution time of each test. However, we have found that since our benchmarks (for the most part) have a very short execution time running on real hardware, running on QEMU with our plugin still measures a small number of seconds for execution to complete. More concrete numbers for run time overhead costs are given in Section 5.5.

5.3.3 Limitations

One of the drawbacks of using QEMU as the emulator is that it does not emulate some of the inner functional units of the processor. There are some emulators, like Simics used in the DrSEUs project [31], that do model these inner functional blocks like TLBs. Being able to inject faults into these parts of a processor gives a more comprehensive view of the failure modes of the processor. We believe that despite these limitations, our setup is still useful for evaluating fault tolerance. The number of bits in the TLBs and other things like processor pipeline registers are dominated by the number of bits in main memory, and even just by the number of bits in the processor caches. For those interested in modeling these other functional units, it should be possible to modify the plugin to also have a model for TLBs, or even to track other kinds of instructions for additional modeling. Other emulators may have built-in support for other functional units without having to create additional plugins. However, it was an intentional design choice to use QEMU, both to keep the whole PACIFIC framework stack open-source, and to keep open the possibility of support for as many platforms as possible.
5.4 Extensibility

Because this fault injection setup is built on top of QEMU, we can theoretically support any architecture that QEMU supports. For our uses, we chose to emulate the ARM Cortex-A9, a 32-bit ARM processor, so we expect that extending this to 64-bit ARM will be relatively easy. Other processor architectures will take a little more work, but should be possible provided the following is addressed:

1. **Register Definitions:** The supervisor needs to know the names of all of the system registers.

2. **GDB Client:** A GDB client that supports the target architecture and can runn Python scripts (GDB 7.0+).

3. **Toolchain:** In addition to whatever compiler toolchain is used to create the executables being tested, PACIFIC requires a version of `objdump` compatible with the target architecture ELF files.

5.5 Evaluation

As the PACIFIC framework is primarily intended as a tool to aid in fault tolerant program design and testing, we will show its usefulness in the testing of some bare metal benchmarks that have been protected with software TMR using the COAST tool [5], [43]. We expect that running many injections on each of the benchmarks will yield a ballpark estimate for the fault tolerance increase for each of the benchmarks, as well as highlight areas of the protection scheme that may be lacking in efficacy.

5.5.1 Experiment

We used PACIFIC to evaluate the fault tolerance of a few different benchmarks that we have used in previous tests, described in Section 4.1.2. These benchmarks were crc32, dijkstra, matrixMultiply, nanojpeg, and sha256.

We injected faults in each of these benchmarks until a certain number of errors occurred, then kept injecting faults up to the next nearest thousand. Ideally we wanted at least 30 errors seen
in each configuration, but some benchmarks take a long time to error when compiled with software
TMR; in those instances we only waited for 20 errors to occur.

Each of these benchmarks do a computation, then check the result against an expected output. The crc32 and sha256 benchmarks both output a hash value, which we can directly compare against a golden value. The dijkstra and matrixMultiply benchmarks both compute results arrays; for these the code will compute an XOR hash and use that to compare with the golden value. All the benchmarks were compiled in two versions, “unmitigated”, the normal compilation process, and “TMR”, which uses the COAST tool to add TMR fault correction abilities to the program.

5.5.2 Results

The full results for the benchmarks are shown in Table 5.1. The table lists the number of faults injected, as well as the number of different results states recorded. These states are the same as reported in the tests in previous chapters. We also report the error rate and associated change for the protected version of each benchmark. The number of cycle counts reported by the plugin is also included as a measure of the overhead cost of adding TMR protection to the benchmarks. We also include the rate at which injections can be performed for each of the benchmarks, in Table 5.2.

5.5.3 Analysis

From examining the table, we can see that the different benchmarks had very different fault tolerance under injection testing. While the matrixMultiply benchmark decreased significantly in error rate, the error rate for dijkstra actually increased; the TMR’d version of the dijkstra benchmark has worse fault tolerance than the unmitigated version. We will use the detailed fault injection logs to show how a system designer would use PACIFIC to classify failure modes or change the protection scheme.

The logs for matrixMultiply show that 32 of the 39 errors reported when injecting into cache occurred in one of the three computation matrices. This indicates that the main vulnerability for this benchmark is data errors. Looking at the logs for the TMR’d matrixMultiply corroborates this theory; 88% of the faults corrected were in the matrices, and all of the 20 errors reported came
Table 5.1: Fault Injection – Evaluation

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Section</th>
<th>Runs</th>
<th>Faults</th>
<th>Errors</th>
<th>Hangs</th>
<th>Invalid</th>
<th>Error Rate</th>
<th>Cycle Count (avg)</th>
<th>MWTF</th>
</tr>
</thead>
<tbody>
<tr>
<td>crc32</td>
<td>cache</td>
<td>10000</td>
<td>N/A</td>
<td>41</td>
<td>0</td>
<td>0</td>
<td>0.41%</td>
<td>2.26 × 10⁸</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>dcache</td>
<td>8000</td>
<td>N/A</td>
<td>250</td>
<td>3</td>
<td>0</td>
<td>3.13%</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>registers</td>
<td>3000</td>
<td>N/A</td>
<td>250</td>
<td>141</td>
<td>0</td>
<td>8.33%</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>crc32</td>
<td>cache</td>
<td>59000</td>
<td>626</td>
<td>25</td>
<td>65</td>
<td>0</td>
<td>0.04% (9.68x ↓)</td>
<td>5.806x ↑</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>dcache</td>
<td>22000</td>
<td>2018</td>
<td>20</td>
<td>33</td>
<td>0</td>
<td>0.09% (34.38x ↓)</td>
<td>3.77 × 10⁸ (1.67x ↑)</td>
<td>20.625x ↑</td>
</tr>
<tr>
<td></td>
<td>registers</td>
<td>3000</td>
<td>360</td>
<td>65</td>
<td>122</td>
<td>0</td>
<td>2.17% (3.85x ↓)</td>
<td>-</td>
<td>2.308x ↑</td>
</tr>
<tr>
<td>dijkstra</td>
<td>cache</td>
<td>57000</td>
<td>164</td>
<td>657</td>
<td>0</td>
<td>0</td>
<td>0.29%</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>dcache</td>
<td>54000</td>
<td>823</td>
<td>3919</td>
<td>1</td>
<td>0</td>
<td>1.52%</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>registers</td>
<td>53000</td>
<td>1120</td>
<td>3430</td>
<td>16</td>
<td>0</td>
<td>8.33%</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dijkstra</td>
<td>cache</td>
<td>12000</td>
<td>665</td>
<td>54</td>
<td>149</td>
<td>0</td>
<td>0.45% (1.56x ↑)</td>
<td>10.637x ↓</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>dcache</td>
<td>10000</td>
<td>3229</td>
<td>169</td>
<td>612</td>
<td>0</td>
<td>1.69% (1.11x ↑)</td>
<td>4.45 × 10⁸ (6.80x ↑)</td>
<td>7.542x ↓</td>
</tr>
<tr>
<td></td>
<td>registers</td>
<td>11000</td>
<td>360</td>
<td>65</td>
<td>122</td>
<td>0</td>
<td>8.33%</td>
<td>-</td>
<td>1.258x ↓</td>
</tr>
<tr>
<td>mm</td>
<td>cache</td>
<td>1000</td>
<td>N/A</td>
<td>39</td>
<td>1</td>
<td>0</td>
<td>3.90%</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>dcache</td>
<td>1000</td>
<td>N/A</td>
<td>289</td>
<td>0</td>
<td>0</td>
<td>28.90%</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>registers</td>
<td>1000</td>
<td>N/A</td>
<td>162</td>
<td>56</td>
<td>0</td>
<td>16.20%</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mm</td>
<td>cache</td>
<td>34000</td>
<td>3353</td>
<td>20</td>
<td>234</td>
<td>1</td>
<td>0.06% (66.30x ↓)</td>
<td>22.247x ↑</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>dcache</td>
<td>51000</td>
<td>40295</td>
<td>21</td>
<td>63</td>
<td>2</td>
<td>0.04% (701.86x ↓)</td>
<td>3.53 × 10⁸ (2.98x ↑)</td>
<td>235.506x ↑</td>
</tr>
<tr>
<td></td>
<td>registers</td>
<td>20000</td>
<td>1573</td>
<td>22</td>
<td>920</td>
<td>1</td>
<td>0.11% (147.27x ↓)</td>
<td>49.417x ↑</td>
<td>-</td>
</tr>
<tr>
<td>nanojpeg</td>
<td>cache</td>
<td>3000</td>
<td>N/A</td>
<td>226</td>
<td>12</td>
<td>0</td>
<td>7.53%</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>dcache</td>
<td>1000</td>
<td>N/A</td>
<td>63</td>
<td>0</td>
<td>0</td>
<td>6.30%</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>registers</td>
<td>1000</td>
<td>N/A</td>
<td>95</td>
<td>72</td>
<td>0</td>
<td>9.50%</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nanojpeg</td>
<td>cache</td>
<td>7000</td>
<td>478</td>
<td>40</td>
<td>56</td>
<td>0</td>
<td>0.57% (13.18x ↑)</td>
<td>1.714x ↑</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>dcache</td>
<td>13000</td>
<td>1156</td>
<td>31</td>
<td>23</td>
<td>0</td>
<td>0.24% (26.42x ↓)</td>
<td>2.11 × 10⁸ (7.69x ↑)</td>
<td>3.436x ↑</td>
</tr>
<tr>
<td></td>
<td>registers</td>
<td>2000</td>
<td>40</td>
<td>35</td>
<td>93</td>
<td>0</td>
<td>1.75% (5.43x ↓)</td>
<td>-</td>
<td>1.417x ↓</td>
</tr>
<tr>
<td>sha256</td>
<td>cache</td>
<td>116000</td>
<td>N/A</td>
<td>33</td>
<td>9</td>
<td>0</td>
<td>0.03%</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>dcache</td>
<td>13000</td>
<td>N/A</td>
<td>36</td>
<td>5</td>
<td>0</td>
<td>0.28%</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>registers</td>
<td>3000</td>
<td>N/A</td>
<td>30</td>
<td>230</td>
<td>4</td>
<td>1.00%</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sha256</td>
<td>cache</td>
<td>17000</td>
<td>564</td>
<td>30</td>
<td>60</td>
<td>0</td>
<td>0.18% (6.20x ↑)</td>
<td>24.022x ↓</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>dcache</td>
<td>22000</td>
<td>5016</td>
<td>30</td>
<td>16</td>
<td>0</td>
<td>0.14% (2.03x ↓)</td>
<td>2.09 × 10⁶ (3.87x ↑)</td>
<td>1.907x ↓</td>
</tr>
<tr>
<td></td>
<td>registers</td>
<td>1000</td>
<td>153</td>
<td>33</td>
<td>46</td>
<td>0</td>
<td>3.30% (3.30x ↑)</td>
<td>-</td>
<td>12.779x ↓</td>
</tr>
</tbody>
</table>

from flipping bits in instruction memory. The system designer can then conclude that the protection scheme chosen is effective at reducing silent data corruption for this particular benchmark.

Contrast this with dijkstra, which is a benchmark dominated more by control flow changes (finding the shortest path) than data computations. The causes of errors in the unmitigated version of the benchmark are more distributed between data and instruction memory. About 58% of the errors were in data memory; of these most were in the adjacency matrix or results vectors. So we expect that the software TMR will have less of an effect protecting this benchmark. Looking at the logs for TMR, many of the faults injected into the adjacency matrix were corrected, but it
Table 5.2: Fault Injection – Rates

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Injection Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>crc32 (Unmitigated)</td>
<td>5.58/min</td>
</tr>
<tr>
<td>crc32 (TMR)</td>
<td>4.4/min</td>
</tr>
<tr>
<td>dijkstra (Unmitigated)</td>
<td>21.41/min</td>
</tr>
<tr>
<td>dijkstra (TMR)</td>
<td>4.23/min</td>
</tr>
<tr>
<td>mm (Unmitigated)</td>
<td>8.18/min</td>
</tr>
<tr>
<td>mm (TMR)</td>
<td>1.53/min</td>
</tr>
<tr>
<td>nanojpeg (Unmitigated)</td>
<td>30.69/min</td>
</tr>
<tr>
<td>nanojpeg (TMR)</td>
<td>5.13/min</td>
</tr>
<tr>
<td>sha256 (Unmitigated)</td>
<td>73.89/min</td>
</tr>
<tr>
<td>sha256 (TMR)</td>
<td>66.59/min</td>
</tr>
</tbody>
</table>

seems that as this matrix represents a smaller percentage of total bits in use by the program than the matrices in the other benchmark, the number of errors was significantly higher in this test.

However, the dijkstra benchmark actually improved the error rate when injecting into the system registers. Again, we can use the detailed information in the campaign log files to get some insight as to why the benchmark may have behaved this way. In the unmitigated version, 26% of errors occurred in data variables, and another 62% in the stack. The last 12% did not have an injection location recorded. The logs record the information the GDB instance knows about the program state when the fault is injected; for registers sometimes it is difficult to tell what variable is in the register being modified. However, the logs do contain the value of the program counter, the old and new register values, and the number of processor cycles since the program began; thus it is possible for the application designer to look at the assembly code and determine which value was in the register at the time the fault was injected. Listing 5.2 shows an entry from the log file produced by injecting faults into the “cache” section of the unmitigated dijkstra benchmark.
Comparing the unmitigated dijkstra benchmark with the TMR’d version, there were no errors in data variables, which is what we expect with our protection scheme. 84% of the errors were in the stack or heap, and the rest were unknown. Looking at the faults corrected summary, we see that most (68%) of the faults corrected were in data memory (variables or the stack and heap). The designer of this application could then infer that the protection scheme chosen is better at protecting control flow issues in this particular case, compared to protecting against data corruption in matrixMultiply. If the designer wanted to also protect against data corruption, they could modify the protection scheme, then re-run the fault injection campaign to validate their changes.
5.6 Using a QEMU Plugin to Profile the Applications

As part of our fault injection campaign, it was important to get accurate data for how long it took each function to execute. Being able to compare execution time between original and protected code allows us to make specific statements on the overhead costs and potential benefits different levels of coverage have on the application.

During the analysis of the data from fault injection campaigns we performed using PA-CIFIC, we also collected data about the execution time of each function in the benchmarks under study. This section describes the method we used to obtain this data, and how it relates to the fault injection plugin described previously.

5.6.1 Profiling Background

Profiling is taking measurements about a program, including memory usage, time, function calls, etc. In our case we want to measure the number of function calls and how long each function took out of the total execution time.

The nature of the applications is such that using a traditional profiler would have given subpar results. gprof, a popular profiler, uses statistical methods to profile applications [51]. One of the big drawbacks to this method is that it depends on long-running applications to get a statistically significant result. The applications we use run for a short period of time, and so functions which take only a few cycles to complete might not show up using this sampling method. In addition, gprof works on Linux, and our runtime setup uses QEMU, running a bare metal application at that. So we needed to come up with a different solution.

5.6.2 QEMU Plugin

We created a plugin that subscribes to a certain set of instructions being executed. The instructions are identified externally, and the plugin is given a file to read from with all relevant addresses already marked. When one of these instructions is executed, the plugin will write a message to a log file. Since we are trying to count function calls, these addresses are for the entrance and exit of functions.
**ELF File Inspection**  In order to get the correct instruction addresses for function entrance and exit, we make use of compiler instrumentation flags and some of the GNU binutils binaries. First, each source file (except assembly) is compiled with the flag

```
-finstrument-functions-after-inlining
```

After the function inlining step, each remaining function is instrumented with calls to the functions

```
__cyg_profile_func_enter
__cyg_profile_func_exit
```

The user must define these functions; we defined them to do nothing, as seen in Listing 5.3.

**Code Listing 5.3: Assembly Code for Profiling Functions**

```
.text
.arm

.global __cyg_profile_func_enter
.global __cyg_profile_func_exit

.align 4
.type __cyg_profile_func_enter, %function
__cyg_profile_func_enter:
    BX LR

.align 4
.type __cyg_profile_func_exit, %function
__cyg_profile_func_exit:
    BX LR
```

Instead of using the profiling function to output information, we use the QEMU plugin, because that has less of an effect on cycle counts, even if it might have more of an effect on run time. We use the binary tool `objdump` to find the addresses where the `__cyg*` profiling functions are called, and save that information in a file for the plugin to read.

**Plugin Output**  The plugin will output a line of the form
The arrow directions represent function entry or exit, with the * indicating that no exit point was found for the function. cycleCount is a decimal number of cycles since the program started, and returnAddr is a hexadecimal number representation of the 2nd argument to the profile function, which is the address the calling function will return to upon completion.

The plugin output is parsed using a Python script which builds a call tree representation of the program execution. Once the tree is built, the script generates an spreadsheet summarizing the call count for each function, as well as how many times it was called. The script can also generate a Graphviz file for the call graph(s) so they may be used for visual inspection of the program execution.

**Handling Context Switches** A big challenge in getting this kind of profiling to work was making it aware of the FreeRTOS kernel. The instrumented profiling functions can only detect when a function is called or it returns, but not when it is preempted. As preemption is one of the major tenets of a Real-Time Operating System, we had to address this issue for the profiling to be accurate.

The solution was to make the plugin output additional text on exit from the function vTaskSwitchContext. This function is the only one responsible for changing contexts, so we could reliably catch all context switches in this place. The plugin looks at the value of the variable pxCurrentTCB (the address of which was found by inspecting the symbol table of the ELF file), and prints out the name of the task and cycle count.

The result of handling context switches this was is that there are multiple “root nodes,” each with its own call graph. We can still summarize the information about each function called, even if it shows up in multiple graphs, though there are multiple Graphviz files, one for each graph.

### 5.6.3 Profiling Results

As most of the work done in this chapter is intended to prepare for the work done on protecting a real-time operating system, the results presented here will be minimal compared to
what is treated later on. Nevertheless, we will give a small example of what kinds of data the profiler can collect, which will become more relevant in the later chapters.

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Total Calls</th>
<th>Total cycles</th>
<th>Self cycles</th>
<th>Percent cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>njGetBits_TMR</td>
<td>29706</td>
<td>7104480</td>
<td>1812066</td>
<td>0.86%</td>
</tr>
<tr>
<td>njDecodeScan</td>
<td>2</td>
<td>70359820</td>
<td>708928</td>
<td>0.34%</td>
</tr>
<tr>
<td>njDecodeBlock_TMR</td>
<td>2640</td>
<td>69649712</td>
<td>10916500</td>
<td>5.16%</td>
</tr>
<tr>
<td>njSkipMarker</td>
<td>2</td>
<td>664</td>
<td>76</td>
<td>0.00%</td>
</tr>
<tr>
<td>njClip_TMR</td>
<td>792012</td>
<td>60169004</td>
<td>60169004</td>
<td>28.45%</td>
</tr>
<tr>
<td>njDecodeDQT</td>
<td>4</td>
<td>17820</td>
<td>16644</td>
<td>0.01%</td>
</tr>
<tr>
<td>Totals</td>
<td>1024430</td>
<td>0</td>
<td>211469546</td>
<td></td>
</tr>
</tbody>
</table>

A main feature of the profiler is its ability to count the number of calls of each function, as well as how many cycles are spent in these functions. Table 5.3 shows a subset of the output obtained by profiling the nanojpeg.tmr benchmark. This table reports not only the amount of time spent in a certain function, but also the total number of cycles spent in the function and its children. The totals column if for the whole benchmark, although only about one quarter of the rows are shown.

As part of the profiling process, a dot file is also created. This format is used to generate the Graphviz files mentioned previously. Figure 5.3 is an example of what a profiling graph might look like. This graph contains information for two iterations of the fft code. The nodes in the graph have the name of the function as well as the total number of cycles spent in that particular function call. Note that many of the dot files contain many more iterations than just two, and so generating a control flow graph will take quite some time. It may be wise to prune the dot file so it only contains one iteration.

5.7 Conclusion

The framework we have created for fault injection, PACIFIC, is a useful tool for simulating the effects of radiation on a system’s data memory. Testing applications with PACIFIC gives insight
into the failure modes of the applications, and can also be used to validate software protection schemes.

This framework has been first validated using benchmarks we have tested before in a radiation test facility. We have shown that it gives a good approximation of the results that would be seen from testing at such a facility. The construction and correct operation of this framework was essential to the success of the experiments discussed in Chapters 6 and 7.

PACIFIC can of course still be improved upon. A first step would be adding the ability to model multi-bit upsets, whether adjacent or separated in memory/time. Another feature to add to the QEMU plugin would be modeling the tag bits. And it would always be beneficial to test the framework with a different target architecture.
CHAPTER 6. PROTECTING FREERTOS WITH COAST

One of the major limitations of past work with COAST, described in Chapters 3 and 4, was that we only targeted simple bare metal applications, such as matrix multiplication or AES encryption. While this work was useful, these small applications did not really explore the limits of automated protection on complex software code.

The objective of this chapter is to explore the complexities involved in adding COAST protection to an entire operating system, and indeed to determine if this is even feasible. In attempting to implement this protection, we will discuss what challenges arise from such an endeavor, and how these challenges can be addressed. We will discuss the how the effectiveness of implementing this protection scheme compares to the bare metal applications, and measure the overhead costs of adding this protection.

In this chapter, we use COAST to apply automated fault-mitigation to the a popular real-time operating system (RTOS), FreeRTOS [54]. Doing so required enhancements to COAST and giving the tool specific configuration options to handle the complexities of FreeRTOS. Specifically, some of the challenges we encountered related to: maintaining automated protection while switching between multiple threads of execution, accessing kernel-specific objects, and the fact that part of the kernel is written in assembly code.

Our work demonstrates that automated protection tools can be used on complex systems, with some assistance from the user. The key contributions of the work presented in this chapter are:

- A demonstration of compiler-automated fault mitigation for a full real-time kernel (FreeRTOS).
- Fault injection results demonstrating 1.3x–267x improvement to MWTF (Mean Work To Failure), depending on the application and configuration.
6.1 Related Work

Past work has investigated protecting RTOSes; however, the approach in most of these works was to manually protect the scheduler code, rather than applying compiler-automated protection to the entire kernel. Different techniques include checkpointing [55], building in lots of slack to the rate-monotonic scheduling [56], dynamic scheduling [57] and others [58]–[61]. There is also a safety critical RTOS called SafeRTOS\(^1\), which is marketed for safety critical applications.

These methods focus on protecting the control-flow of the RTOS, whereas COAST attempts to protect the entire data flow of the program, and provides synchronization and correction on each control-flow branch. Additionally, the methods presented in these works operate at a coarse-grained scale, for example, treating each task as its own protection unit. Contrast this with COAST, which has voting code sprinkled throughout both application and kernel code. In summary, these efforts employ methods which are different than, and potentially complementary to, the methods discussed in this work.

Some more closely related work has been done by Borchert et al., who use Aspect Oriented programming to get automated protection of two different RTOS platforms [62], [63]. This depends on the Aspect-Oriented C++ compiler, so RTOSes written in C, such as FreeRTOS, could not be supported.

6.2 Using COAST to protect an RTOS

Previous work in Chapters 3 and 4 demonstrated that protecting bare metal programs with software fault tolerance using the COAST tool can yield great results. We want to look into protecting a more complicated and featureful system, which may be of more use to some developers than a basic bare metal application. The first step in expanding COAST to more complex systems is protecting a real-time operating system (RTOS). RTOSes vary in size, complexity, and features, and there are many that are available and open-source. We chose to do our experiments using FreeRTOS for a couple of reasons:

1. It is a commonly used RTOS in the embedded space,
2. It has a small and relatively simple kernel, making it easier to protect automatically.

\(^1\)https://www.highintegritysystems.com/safertos/
When writing application code for the FreeRTOS kernel, the developer will make calls to functions defined by the kernel. Unlike system calls on an OS such as Linux, these kernel functions are compiled into the same single binary with the application code. Having all of the code as part of a single compile chain makes it easier to use with a compiler-based protection tool like COAST.

Protecting the FreeRTOS kernel introduces additional complexities beyond a bare metal system, such as multiple threads of execution, preemptive scheduling, and multiple system interrupts. In several cases, new functionality was added to the COAST tool to correctly handle these complexities. Section 6.3 will enumerate some of the challenges associated with protecting FreeRTOS with COAST. It will also go over some of the design decisions that were made as part of this process.

### 6.3 Challenges Applying Automated Protection to FreeRTOS

Our approach to protecting an RTOS is to protect the entire software stack, both application and kernel code. Our intention was to use COAST to do a blanket TMR protection of all the user variables and kernel objects. This includes both statically declared and dynamically allocated variables. However, while attempting to implement this protection approach, we ran into several issues. This section describes the different challenges we encountered, and our solutions to overcome them.

#### 6.3.1 Scope of Replication

Before enumerating the challenges experienced implementing this protection scheme, we discuss a fundamental principle to understand when using software protection techniques, called the *Scope of Replication* (SoR). This defines which parts of a program (i.e., which functions and variables) should be replicated. Protected variables, which are those within the SoR, should only be accessed within the SoR, to ensure that replicas are kept up to date and synchronized. Unprotected variables should primarily be accessed by functions outside the SoR, though they can usually be safely accessed by functions inside the SoR. However, any time the use of a variable passes through the boundary of the SoR, there is potential for errors to happen, if not handled correctly.
```c
#include <stdlib.h>
#include <stdio.h>
#include "COAST.h"

int* glblPtr;

__xMR_FN_CALL
void incPtrVal(int* ptr) {
    *ptr += 1;
}

void middleCall() __xMR {
    incPtrVal(glblPtr);
}

int main() {
    glblPtr = (int*)malloc(sizeof(int));
    *glblPtr = 0;
    middleCall();
    printf("%d\n", *glblPtr);
}
```

Figure 6.1: Scope of Replication – Example

Figure 6.1 gives an example of code that does not properly handle the SoR boundaries. The expected result of running this code is that glblPtr will point to the value 1, but it will actually be 3 instead. This is because the function incPtrVal performs a read-modify-write operation; all the reads and modifications should take place before the writes. The other problem is that there is only one pointer, but it’s being treated as if there were three copies. More discussion on the Scope of Replication can be found in Sections 3.2.2 and 3.3.1 of Matthew Bohman’s Master’s thesis [4].

The protection scheme used with the benchmarks mentioned in Chapters 3 and 4 is very basic and the corresponding Scope of Replication includes nearly all of the variables and functions in the application. The protection scheme and SoR for FreeRTOS applications is not as straightforward as those were.
6.3.2 Overview of Challenges

Implementing blanket COAST protection was not possible due to a number of challenges involving the Scope of Replication. First, some of the code for the kernel is written in assembly code. This is not possible to protect with COAST, as architecture-specific assembly code is not part of the LLVM IR. Second, issues of object replication arose from the various ways that FreeRTOS allocates objects, and originally we were losing copies of kernel objects. We also had to deal with ensuring the correct number of copies of dynamically allocated objects are created, as well as making sure function inlining did not interfere with the Scope of Replication. Even with all of these difficulties, which are more numerous than we had experienced protecting bare metal applications, we were still able to implement the protection schemes solely during compilation with the COAST build flow. COAST required a large number of command line flags and in-code directives to properly maintain the integrity of the Scope of Replication, but we did not hand-modify any assembly code for this experiment.

This section explains most of the problems and solutions in generic terms, and also gives examples of how this applies specifically to FreeRTOS. Besides the comprehensive protection scheme, we also looked into protecting only the application code and not the kernel. This introduced its own challenges, explained in Section 6.4.2. Finally, we were interested in only protecting key parts of the kernel. Ultimately this wasn’t feasible due to problems maintaining the integrity of the Scope of Replication. This is described in more detail in Section 6.4.3.

6.3.3 Architecture Specific Code

As the COAST tool operates on LLVM IR (Intermediate Representation), it is dependent on having the source code available. Anything already at a lower level than the IR (such as assembly code) cannot be protected automatically by an LLVM-based tool. Refer to Figure 3.1 for the overview of the COAST build flow. The port-specific assembly code follows the lower path in the build flow. One can see how assembly code cannot pass through COAST during compilation. If this port-specific assembly code accesses global variables, care must be taken to ensure these globals are not included inside the Scope of Replication. COAST must be informed of this to make sure that all accesses to these globals remain consistent.
In many cases it is possible to detect these invalid access patterns by examining the list of functions and globals that are protected and comparing that with actual uses of each of the globals. We added a verification step to COAST that will tell the user if any variable accesses violate the protection rules given in the compiler invocation.

COAST must treat each access to these globals correctly. If an instruction that uses a non-replicated global does not reference any other objects, the instruction itself should not be replicated. If the instruction references both a non-replicated global and another value that is replicated, each copy of that instruction must use only the single reference to the global. While COAST was designed to do this, the complexity of a full RTOS exposed some unhandled subtle cases that required fixes and improvements to the source code of COAST.

In the FreeRTOS kernel, the part of the scheduler that saves and restores context and the interrupt handlers are written in assembly. To properly deal with this, we manually created a list of global variables which needed to be excluded from protection, in order for the integrity of the SoR to be maintained, and instructed COAST to skip replicating these variables.

### 6.3.4 Losing Replicated Objects

There are certain FreeRTOS kernel functions which create objects that will be used by the user space tasks, but are allocated in the kernel functions. If the handles (pointers) to these objects are returned directly to the calling function via the return value, copies can be lost. By default COAST inserts a voter to synchronize return values before a function returns, but this is only valid when the values are scalars, as voting on pointers is not appropriate. When the kernel objects are inside the SoR, there are three different objects created. The C language allows a single return value from each function, so only one of these kernel objects would be returned, losing the references to the replicate copies.

There are a few ways to deal with this problem. The first is by calling the function more than once, with the function being outside the SoR, and each invocation returning one of the triplicated object copies. While this works in most cases, certain functions we encountered have side affects, and calling them multiple times caused internal data corruption. In these cases, the function must be invoked only once, and the function signature must be modified to “return” multiple values. In
addition, calling the function multiple times usually means the function body and associated local variables are not protected by COAST replication.

We added functionality to COAST that allows it to accept a list of functions on the command line that need to have their return values replicated. COAST then modifies the function signature by adding two extra arguments, which allow the extra values to be returned to the caller through pointers. The call sites are also updated automatically to retrieve these replicated objects appropriately. This is used to give the impression that the function returns more than one value, even when it is only called once.

To give an illustration of this concept using functions from the FreeRTOS kernel, we look at two kernel functions, creating a message queue, and creating a timer. The operation of creating a queue simplifies to initializing a struct, so to get three copies of a queue, we instruct COAST to call the queue initialization function three times. Contrast this with the operation for creating a timer, which has side effects in the kernel. Since we cannot call the function more than once, we instruct COAST to replicate the return values.

Code Listing 6.1: Losing Replicated Objects – Example

```c
TaskHandle_t xTaskGetCurrentTaskHandle( void )
{
    TaskHandle_t xReturn;

    /* A critical section is not required as this is not called from
       an interrupt and the current TCB will always be the same for any
       individual execution thread. */
    xReturn = pxCurrentTCB;

    return xReturn;
}
```

Listing 6.1 helps to illustrate this problem. The function described therein (simpler than the timer code) can only be called once, so COAST must create extra pointers and pass them as arguments to the function. Listing 6.2 shows the LLVM IR of the function after COAST has added the extra pointers to the function signature. Notice how the last instructions before the return instruction set up the other “return value” pointers. Listing 6.3 gives part of the LLVM IR of
the function that calls the function from Listing 6.1. Note that this technique incurs additional
overhead in run time and also in stack space required for these extra pointers.

Code Listing 6.2: Losing Replicated Objects – Solution Part 1

```c
#define dso_local i8* @xTaskGetCurrentTaskHandle.RR(i8** %__retVal.DWC,
    i8** %__retVal.TMR) local_unnamed_addr #4 {

entry:
  ; save the addresses of the extra pointers
  %__retVal.DWC.addr = alloca i8**, align 4
  store i8** %__retVal.DWC, i8*** %__retVal.DWC.addr, align 4
  %__retVal.TMR.addr = alloca i8**, align 4
  store i8** %__retVal.TMR, i8*** %__retVal.TMR.addr, align 4

  ; get the return value and copies
  %0 = load volatile %struct.tskTaskControlBlock*,
    %struct.tskTaskControlBlock** @pxCurrentTCB, align 4, !tbaa !3297
  %1 = bitcast %struct.tskTaskControlBlock* %0 to i8*
  %.DWC = load volatile %struct.tskTaskControlBlock*,
    %struct.tskTaskControlBlock** @pxCurrentTCB_DWC, align 4, !tbaa !3297
  %.DWC1 = bitcast %struct.tskTaskControlBlock* %.DWC to i8*
  %.TMR = load volatile %struct.tskTaskControlBlock*,
    %struct.tskTaskControlBlock** @pxCurrentTCB_TMR, align 4, !tbaa !3297
  %.TMR2 = bitcast %struct.tskTaskControlBlock* %.TMR to i8*

  ; store to the pointers before returning
  %loadRet = load i8**, i8*** %__retVal.DWC.addr
  store i8* %.DWC1, i8** %loadRet
  %loadRet2 = load i8**, i8*** %__retVal.TMR.addr
  store i8* %.TMR2, i8** %loadRet2
  ret i8* %1
}
```
6.3.5 Handling Dynamic Memory Allocation

Functions which dynamically allocate memory are some of the most important function calls to be aware of when replicating and protecting the data of a program. In some instances it is appropriate to call these functions multiple times, and other instances they must only be called once.

As an example from FreeRTOS, allocating stack space for a task should only be done once. This is because the stack is something which is primarily used by assembly code; LLVM IR has no concept of a system stack. In contrast, when allocating the TCB (Task Control Block, a struct which contains data for each user task) we want the data to be replicated, and thus multiple memory blocks should be allocated.

Unfortunately there is no obvious way to do this automatically, and again we needed to manually build a list of locations in the code where dynamic allocation should not be replicated. These modifications to the compilation process were accomplished using some of COAST’s in-code directives.
6.3.6 Inlining

Compilers often inline the bodies of functions into their callers as an optimization. In most cases this is desirable, as it can lead to decreased run time. However, if one of the functions is supposed to be inside the SoR, and the other is not, inlining can corrupt the protection scheme. When a function is inlined, the symbol for the original function no longer exists, and so COAST cannot treat it correctly because it is no longer distinct from its caller(s).

There were a few functions in the FreeRTOS kernel where we had to instruct the compiler not to inline, otherwise the symbol would disappear before it got to the optimizer pass that implements COAST. We achieved this by using the common symbol annotation

```
__attribute__((noinline)).
```

6.4 Protection Schemes

6.4.1 Full Protection Scheme

Taking into account the difficulties outlined in Section 6.3, we created a protection scheme which covers as much of both the kernel and application code as possible. This is referred to in our experiments as the “full” protection scheme. The exact lists of functions given special treatment in this protection scheme can be found on our repository.

6.4.2 Application-only Protection

As we have seen in previous experiments, there is a significant penalty in both run time and memory usage for comprehensive TMR protection (~3-4x increase). Thus, in addition to our protection scheme which covers all of the kernel and application code, we also wanted to explore the effects of protecting only the application code, in order to better understand the possible trade-off between fault tolerance and performance cost. We refer to this protection scheme as “application-only”. The main consideration for this protection scheme is that handles to the kernel

---

2https://github.com/byuccl/coast/blob/master/rtos/pynq/Makefile
objects cannot be replicated in the application code. This includes function arguments that serve as return values.

The “application code” is the code that defines the RTOS tasks and supporting functions. We expect that this protection scheme will have a lower overhead in both code size and run time. This subsection explores some of the details of implementing this protection scheme for the benchmark rtos_kUser.

Almost all of the functions in the application part of the code are protected, with the exceptions being the functions which delete the tasks at the end of the program execution. The reason for skipping these functions is because they call a single kernel function, vTaskDelete, which is not protected, and pass in unprotected task handles. As the main benefit of COAST comes from protecting values and not instructions, we disable protecting these functions, as the added instructions would more get in the way than be of help.

The values which reside in application code that are not protected are task handles, kernel queues handles, timing values, function pointers, and some local variables used in sending/receiving data from the kernel.

**Task Handles** All of the task handles were skipped because the function that creates a new task is not protected in this experiment, meaning that only one task handle will be returned anyway. This is a similar reason to why the functions that delete tasks were not protected.

**Queues** Any queues or other kernel data structures are skipped. This is because objects like queues and event groups rely on the list system of the kernel, and the list objects are owned by the Task Control Blocks (TCBs), which are not replicated. Attempting to protect these kernel objects would result in the linked list problem discussed in Appendix A.1.

**Timing Values** There are two timestamp variables used in measuring the execution duration of the tests. In the full TMR protection scheme, these values can be protected. However, in this partial protection scheme, the function which retrieves the timestamp is not protected, and so the timestamp variables cannot be protected either. More information about the special treatment of this timing function can be found in Section 7.3.5.

**Function Pointers** The application code for both benchmarks has code which is checks to see if the benchmark has completed its work. This is discussed in more detail in Section 6.5.1. Part of
the functionality is implemented using a function pointer, which is called when the benchmark has finished. This function pointer should not be replicated, mainly to avoid side effects from calling the function it points to more than once. We discuss some of the potential problems with function pointers in Appendix A.2.

**Local Storage Variables** Kernel functions such as those which send and receive messages to/from a queue use variable references to exchange the message with the application code. The application code calls these functions, usually using the address of a task-local variable. In this partial protection scheme, since the kernel functions aren’t changed, only one address may be passed, and so these local variables are marked with the in-code directive `NO_xMR`.

### 6.4.3 Partial Kernel Protection

We also considered protecting only parts of the kernel. We created a plugin for QEMU that allowed us to obtain detailed profiling information of our benchmark applications, allowing us to identify the kernel functions that were invoked most frequently. While we initially wanted to protect just these functions, after some attempts at this, it became clear that the kernel functions and kernel objects were simply too interconnected to implement such a partial protection scheme. Accesses to the kernel objects were widespread, and as explained previously, the rules to maintain SoR correctness prevent accessing protected data from unprotected code.

### 6.4.4 Partial Protection – Conclusion

It is possible to create an application that is only partially protected, provided that it obeys certain rules. The principles all involve maintaining the integrity of the Scope of Replication, and can be applied to specific situations in different ways. For example, if you wanted to protect all of the kernel, but as little of the application as possible, some of the rules would be:

- Any time a task handle is to be used, multiple copies must be stored
- Any functions which take task handles as arguments must be called inside protected code

The generalization of these rules is, all kernel functions must be called inside of protected application functions, essentially creating a wrapper layer. This effectively expands the SoR into a
subset of the application code, but one which is clearly distinct from the rest of the code. How to do this depends very much on the shape of the application code.

6.5 Application Design

To test the FreeRTOS protection scheme, we chose applications which would accurately represent the types of use cases an application writer might work with. The first application is a modified version of the demo application distributed with FreeRTOS, intended to exercise kernel functions. The second is our matrix multiplication benchmark used in previous tests, ported to work with FreeRTOS. By using these two different applications we aim to show the difference between an application that is dominated by calls to kernel functions versus one where most of the time is spent in the application code.

For each of these applications, we implement the full protection scheme, which is denoted by adding the “.TMR” suffix to the benchmark name. We also implement the partial protection scheme, denoted by the suffix “.app.TMR”. The replication rules for the kernel were input as part of the command line invocation of COAST as often as it was possible, in order to keep the code of the FreeRTOS kernel as close as possible to the state in which it is distributed. In contrast to this, for the application code we used in-code directives as much as possible. This was done to help make it clear which parts, and in which ways, the application was being protected.

Compared to the multitude of bare metal benchmarks tested in previous chapters, only having two benchmarks for testing the FreeRTOS protection scheme seems paltry. We would have liked to have more, however it is quite difficult to locate any open-source RTOS benchmarks, especially ones independent of any particular hardware requirements.

This chapter concludes with a description of the benchmarks. In Chapter 7 we will discuss the experimental results obtained from performing fault injection testing on the benchmarks and protection schemes described in this chapter.

6.5.1 rtos.kUser

This application is based on the FreeRTOS demo application that comes as part of the installation files. This application tests the use of kernel functions, such as queue send and receive,
task notifications, etc. The code has been modified to have a well-defined “finished” metric, and to print the same kind of heartbeat message we have used in the tests described in Chapters 3 and 4.

The application contains functions which are in charge of making sure the RTOS execution results are correct. One of these functions allows registering a callback, which will be called when the work is considered “done.” This function is triggered when a certain number of queue operations have been performed in a specific task.

6.5.2 rtos_mm

In addition to the application made to exercise kernel functions, we also created an application based on our bare metal matrix multiplication benchmark. The core operation of multiplying two matrices is done by two tasks, each alternating whose turn it is to run. We chose two tasks instead of one to make sure the kernel functions were still being exercised at least a little bit. However, this application is still heavily dominated by processing in the user space code compared to the kernel functions.

Similar to rtos_kUser, this application has a “finished” condition defined by how many times the matrix multiplication has been calculated. When completed, the application prints out information about errors and/or faults detected, as well as run time.
CHAPTER 7. FREERTOS EXPERIMENTAL RESULTS

This chapter continues the discussion of protecting FreeRTOS with COAST by presenting the experimental results. This chapter also goes into detail about some of the failure modes of the protection schemes.

7.1 Test Methodology

We designed an experiment to test the validity of the protection schemes discussed in Sections 6.4.1 and 6.4.2. The full TMR protection scheme, as well as the partial application protection, are both applied to two different benchmarks. We tested these protected FreeRTOS benchmarks, along with the unprotected versions, using the fault injection framework PACIFIC discussed in Chapter 5. The goal of this experiment was to determine the increase in fault coverage afforded by the protection scheme. Runtime and memory usage overheads were also measured.

7.2 Experimental Results

Table 7.1 shows the results of injecting faults into the \texttt{rtos.kUser} benchmark, and Table 7.2 the results for the \texttt{rtos.mm} benchmark. In each table, the first column lists the protection scheme and the second column lists the region targeted for fault injection. These section targets are discussed in more detail in Section 5.3. We chose these as being mostly representative of the types of errors that would be seen in a real radiation test. The next set of columns indicate how many benchmark executions were performed, and the outcome of each execution and fault injection. An “error” is when the benchmark completes, but the calculated result value is incorrect. For \texttt{rtos.mm}, this means that the output matrix has an error in it, detected by running a CRC like was done in the bare metal version. For \texttt{rtos.kUser}, an error means that a value passed to/from a kernel object, such as a queue, did not match what was expected. A “fault” is when the voting code inserted by COAST detects and corrects an error. An “invalid” result means the benchmark output did not
match the expected format. These benchmarks have the same periodic heartbeat message format used by our bare metal benchmarks. A “timeout” is when the benchmark did not complete in the expected time (all configurations are expected to finish in less than 5 seconds, sometimes much less). We are most interested in reducing the error rate, as this represents silent data corruption (SDC), and would normally be undetectable. While reducing timeouts or invalid outputs would also be beneficial, we anticipate that in a deployed system there would be a reboot mechanism to deal with such occurrences.

Table 7.1: Fault Injection Results – rtos_kUser

<table>
<thead>
<tr>
<th>Protection</th>
<th>Section</th>
<th>Runs</th>
<th>Faults</th>
<th>Errors</th>
<th>Timeout</th>
<th>Invalid</th>
<th>Error Rate</th>
<th>MWTF</th>
<th>Size (KB)</th>
<th>Cycle Count (avg)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unmitigated</td>
<td>cache</td>
<td>5000</td>
<td>0</td>
<td>122</td>
<td>97</td>
<td>0</td>
<td>2.44%</td>
<td>-</td>
<td>824</td>
<td>$4.22 \times 10^7$</td>
</tr>
<tr>
<td></td>
<td>dcache</td>
<td>5000</td>
<td>0</td>
<td>162</td>
<td>242</td>
<td>0</td>
<td>3.24%</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>registers</td>
<td>5000</td>
<td>0</td>
<td>150</td>
<td>261</td>
<td>0</td>
<td>3.00%</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>TMR (kernel + application)</td>
<td>cache</td>
<td>10000</td>
<td>292</td>
<td>44</td>
<td>359</td>
<td>0</td>
<td>0.44% (5.55x ↓)</td>
<td>1.495x ↑</td>
<td>668 (0.81x)</td>
<td>1.56 $\times 10^8$ (3.71x ↑)</td>
</tr>
<tr>
<td></td>
<td>dcache</td>
<td>10000</td>
<td>753</td>
<td>69</td>
<td>675</td>
<td>7</td>
<td>0.69% (4.70x ↓)</td>
<td>1.266x ↑</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>registers</td>
<td>10000</td>
<td>323</td>
<td>47</td>
<td>450</td>
<td>0</td>
<td>0.47% (6.38x ↓)</td>
<td>1.721x ↓</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>TMR (app only)</td>
<td>cache</td>
<td>10000</td>
<td>40</td>
<td>45</td>
<td>237</td>
<td>0</td>
<td>0.45% (5.42x ↓)</td>
<td>3.310x ↑</td>
<td>592 (0.72x)</td>
<td>6.91 $\times 10^7$ (1.64x ↑)</td>
</tr>
<tr>
<td></td>
<td>dcache</td>
<td>5000</td>
<td>49</td>
<td>45</td>
<td>234</td>
<td>2</td>
<td>0.90% (3.60x ↓)</td>
<td>2.198x ↑</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>registers</td>
<td>5000</td>
<td>9</td>
<td>38</td>
<td>288</td>
<td>1</td>
<td>0.76% (3.95x ↓)</td>
<td>2.410x ↑</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

After the failure modes, the next set of columns provide the error rate and mean work to failure (MWTF). The error rate is the number of errors divided by the total number of times the benchmark was run. While error rate is a good indication of our effectiveness at reducing the number of errors, it is important to recognize that increases in program run time would naturally cause more faults to occur in an SEU prone environment. Thus, the MWTF column scales down...
the improvement in error rate by the increase in run time, to provide a more fair metric of total improvement. The equations from cross section (replaced by error rate in these tables) and MWTF are given in Equations 3.1 and 3.2. A summary for the MWTF for each benchmark configuration and injection target can be found in Figure 7.1.

The final set of columns give the executable size and number of simulator cycles to complete execution, as well as the relative change versus the unmitigated configuration. This is independent from fault injection location, so data is given only once per configuration. The reader may notice that the “size” column in each table indicates that the protected versions of the benchmarks are actually smaller than the original size. This is because all unused symbols were stripped out during compilation, a feature of COAST. The FreeRTOS kernel and processor BSP (Board Support Package) both contain some functions that are not used by the applications.

rtos_kUser: Both protection schemes provided a decrease in error rate and related increase in MWTF for all three target sections. Note that even though the comprehensive protection
Scheme resulted in an overall lower error rate, because of the steep penalty in run time overhead it has, protecting only the application code gave a better increase in MWTF. This indicates there are some instances where protecting only the application code could provide enough fault tolerance to meet the desired goals, especially when reducing protection overhead is important.

rtos_mm: For this benchmark, while COAST definitely helps protect against errors in the caches, errors in registers resulted in a worse error rate than without protection. Errors caused by injections into registers are usually control flow errors. Since this benchmark is essentially a triply-nested loop, any error in control flow is likely to affect the hash that is run at the end to validate the result.

It is important to keep in mind that the number of bits in the register file is significantly smaller than the total bits in all the caches. This means that, in an environment where upsets are evenly distributed between all bits in the system, the results for injecting into the cache will most closely resemble the actual fault coverage.

Tables 7.1 and 7.2 also show that the number of “timeouts” are increasing when the protection schemes are applied. When the program does not respond, this is likely due to a control flow error. These seem to become more prevalent when the data errors are mitigated.

7.2.1 Impact on Runtime

To measure impact on run time, we profiled each application to see how much time was spent in each function. This profiling was done through another QEMU plugin we created which outputs the cycle count on entry and exit of each function. A Python script parsed this data to build the data shown in Figure 7.2. The groups on the x-axis are the different configurations of benchmark and protection scheme. The colored series are the breakdown of run time increase across different parts of the code, indicated by the legend. As expected, when protecting code, run time increases by \(\sim 3-4x\), as instructions are triplicated, plus voters are inserted. As evident from the total data (yellow), rtos_kUser spends most time in kernel code, while the rtos_mm benchmark follows the reverse trend.

To give an idea of the kind of detailed information that can be obtained from the profiling plugin, we present a small sampling of the profiling results from the rtos_kUser benchmark in Table 7.3. This table shows the percent cycles used by three major kernel functions and how they
Figure 7.2: Benchmark Run times

Table 7.3: Kernel Profiling – rtos.kUser

<table>
<thead>
<tr>
<th>Function Name</th>
<th>% cycles rtos.kUser</th>
<th>% cycles .app.xMR</th>
<th>% cycles .xMR</th>
</tr>
</thead>
<tbody>
<tr>
<td>xQueueGenericSend</td>
<td>17.39%</td>
<td>8.75%</td>
<td>11.62%</td>
</tr>
<tr>
<td>xQueueReceive</td>
<td>24.02%</td>
<td>11.92%</td>
<td>14.27%</td>
</tr>
<tr>
<td>vTaskSwitchContext</td>
<td>7.62%</td>
<td>4.66%</td>
<td>8.10%</td>
</tr>
</tbody>
</table>

differ across the various configurations. Similar numbers were obtained for all of the functions in the kernel, allowing for a very detailed analysis of the run time effects of the protection schemes. These numbers are omitted for brevity.

7.2.2 Impact on Memory Usage

To evaluate impact on memory usage, we measured both static and dynamic memory allocation. Static memory usage was determined by examining the ELF symbol table. Dynamic
memory usage was calculated by manually searching for all calls to `pvPortMalloc` (the FreeRTOS kernel version of `malloc`) and using our knowledge of the applications to calculate how much memory these calls will account for. Total memory usage of each benchmark configuration, relative to the unmitigated version, is shown in Figure 7.3. We choose to combine static and dynamic in the same graph since each benchmark uses both types of memory allocation in the same way. The organization of the groups and series is the same as in Figure 7.2, with configurations across the x-axis and the colored series breaking down the user- vs kernel-space statistics. The y-axis is the relative increase in static plus dynamic memory usage compared to the unmitigated versions.

Compared to the change in run time, the difference in memory usage is not as large. Some of the memory usage that is constant across each configuration can be attributed to things like libraries for printing formatted strings. Another contributing factor is that there are certain parts of the kernel, most notably the task stack spaces, that cannot be replicated in any case.
7.2.3 Impact on Cache Usage

In this section we present some interesting statistics we gathered about how the caches are utilized during the running of each of our benchmarks. Looking at the usage of each cache should give us some increased understanding how using COAST to protect these applications affects the memory profile. We would like to know how much of the program and its data fit in the caches.

Based on some previous observations [43], we presume that the greater the cache utilization of a program, the better that COAST can protect it. In addition, being more resident in cache improves performance speed.

In order to get this cache information, we used a stripped-down version of the QEMU plugin used in PACIFIC that keeps only the cache emulation functionality. At the end of the program execution, the plugin would emit the number of accesses to each cache, separated by loads and stores. We collected this data and now present it in graphs to make it easier to visualize and compare.

Figure 7.4: Cache Miss Rate – rtos_mm

Figure 7.4 shows the cache utilization characteristics for each version of the rtos_mm benchmark. The x-axis shows the load/store usages of the three processor caches (no storing
to the instruction cache). The colored series represent the relative increase in cache miss rate for the two protection schemes of \textit{rtos\_mm}. The reader may notice that most of the bars are \textit{below} 1.0x. To understand why cache utilization is better, except for D-cache loads, we look at the data in Figure 7.5.

![Figure 7.5: Cache Operations – \textit{rtos\_mm}](image)

In Figure 7.5, the colored series are the same as in Figure 7.4, but the x-axis has slightly different cache access categories. This graph indicates that the total number of accesses to each cache is close to 3x, which is what we expect for TMR protection. The outliers here are the accesses to the L2 cache. While the number of L2 cache accesses represents a relatively small percentage of the total number of cache operations (less than 1.5% in all cases), these large bars in Figure 7.5 show that the memory accesses in the unmitigated version fit better in the L1 caches, and that adding COAST increased the total memory size just enough that the L2 cache had to be used much more frequently.
The above discussion illustrates the usefulness of the QEMU cache plugin that we have created. We were able to get accurate measurements of cache operations for each level of the cache, and draw important conclusions from the data.

7.3 Analyzing Kernel Errors

Even with COAST protection, there are still certain data errors that can occur. When protecting an OS kernel, there are more things to consider, and more “moving parts” that can be upset than in a typical bare metal application. We want to know what types of data errors can be expected after a program has received COAST protection.

After completing the fault injection experiments discussed in Section 7.2, we looked through the log files and compiled a list of benchmark runs that reported a computation error. By examining and categorizing these errors, we have identified a number of ways in which the kernel can fail. These include:

- Control Flow Errors
- Stack Frame Corruption
- Generic Pointer Corruption
- Single Points of Failure

This is not meant to be an exhaustive list of ways in which an OS kernel can fail, but it should be useful in illustrating the kinds of ways that kernels can fail, in general. The following subsections will go into more detail on each of these points.

7.3.1 Control Flow Errors

The functions that comprise the kernel represent a large number of machine instructions. In the case of FreeRTOS, the .text ELF section is about the same size as the .bss section (for our applications). This means that an upset is just as likely to occur in the middle of an instruction as it is in data memory.

When values are changed in instruction memory, it becomes very difficult to predict what will happen next. Most likely there will be some sort of control flow error, in that code execution
will branch to somewhere wrong, or chooses the wrong condition. There can also be data errors, if the fault causes an instruction opcode to change or load the wrong memory address.

**Hardware Considerations**

In ELF files, the `.text` section is marked read-only because instructions should not change at run time. In some embedded platforms, this section will be in a physically different memory location than memory which can be modified (such as `.bss`, `.data`, etc). This largely depends on the linker file and how it is set up, but it is certainly possible for the `.text` section to be in a ROM memory, such as Flash, and for the other data to live in something like DRAM.

Some studies have shown that Flash, as long as it is not being written, is less likely to be upset than standard DRAM. Fogle et al. estimates Flash to have a soft error failure rate between “3–5 orders of magnitude better than SRAM” [64], and the error rates of DRAM is only about one order of magnitude better than SRAM [42]. This discrepancy can affect the overall susceptibility of the platform and benchmark configuration by making it more likely (all else equal) for errors to occur in data memory than instruction memory.

**7.3.2 Stack Frame Corruption**

As mentioned in Section 6.3, which describes the protection scheme for FreeRTOS, although we keep multiple copies of the Task Control Block (TCB) in memory, each task has only a single stack. This is due to hardware limitations – the architecture has only a single stack pointer register. It would be beyond the scope of our project to rewrite the compiler back end to use multiple registers as stack pointers. LLVM IR, on which our compiler pass operates, does not have any notion of a stack pointer, or even pushing/popping from the stack. This all happens at the architecture assembly level.

Each task has its own stack, which is used for allocating automatic variables and saving the task context when switching between tasks. This makes this area of memory a large target for upsets, because each task will have a chunk of memory allocated for it from the heap. The default
size for a task stack in FreeRTOS is 200 bytes. The FreeRTOS website has a detailed example of context switching which can be consulted by readers who are unfamiliar with this concept\(^1\).

A stack frame is created at the beginning of each function call, where the assembly code (prologue) will push onto the stack all registers it will need to restore again later before returning. When the function finishes, more assembly code (epilogue) will restore these values using the pop instruction. This is illustrated in Listing 7.1. There is not an easy way to protect these values, because as mentioned, they don’t exist at the IR level.

**Code Listing 7.1: Example Function Prologue & Epilogue**

```
1 ; function prologue
2 push {r4, r5, r6, r7, r8, r9, sl, fp, lr}
3 add  fp,  sp, #28
4 sub   sp,  sp, #28
5 ; ...
6 ; function epilogue
7 sub   sp,  fp, #28
8 pop   {r4, r5, r6, r7, r8, r9, sl, fp, pc}
```

One error we found came from a fault corrupting the return address. When a function is called, the next address immediately after the calling instruction is placed in the link register ($\text{l}r$). This value is put back into the program counter ($\text{pc}$) at the end of the function. If this value is corrupted, then execution will not be able to return to the correct address after the function finishes. Often the compiler will use the link register for other purposes besides just holding the return address, and so the return address will be put on the stack until the end of the function. This becomes the vulnerability point for the return address.

In an example of this problem, we found an instance in the log files where the return address was corrupted halfway down a call stack, so that instead of returning to finish executing the calling function (queue sending a message), it went into executing halfway through a completely different function (a kernel function which removes an item from a list), which is clearly undefined behavior. After the first error, the rest of the effects of the faults are corrected. This was not the only instance in our experiments of execution jumping to the wrong return location. Thus we can see that the

\(^1\)https://www.freertos.org/implementation/a00020.html
stack frame is a part of the memory map that is still vulnerable to upsets, even with COAST protection.

Another error came from the address of the frame pointer being corrupted. The frame pointer is a register which contains an address on the stack that corresponds to where the stack pointer was when the current function was called. In the error we found, when the function call returned, it reset the frame pointer to be the wrong value. The memory loads and stores based off of the frame pointer were incorrect, but the ones based on the stack pointer still worked correctly. This corruption is likely to cause data errors. Similarly, if any of the register values saved in the stack frame are corrupted, this is likely to lead to data corruption. However, those kinds of errors are easier to recover from than corrupting the frame pointer or program counter.

### 7.3.3 Generic Pointer Corruption

Of course, any time there are multiple copies of a data structure in memory, and pointers are involved, there is potential for Silent Data Corruption. COAST does not vote on values of pointers, because most of the time, it is correct for them to be different addresses between the copies.

We will now give an in-depth analysis of one such example of pointer corruption that we discovered while examining the failure modes obtained from our fault injection experiments. We found a fault which corrupted one of the pointers in a queue struct. The value

```c
queue_TMR->u.pcReadFrom
```

was being set to point to something 0x100 less than its original value (we use `queue_TMR` to mean the 3rd copy, and `queue` to mean the original copy).

When FreeRTOS queues are created, they also allocate space for items that will be placed on that queue. This is a design choice by the kernel architects, so that whatever is sending data using the queue does not need to keep the value around. If dynamic memory allocation is enabled in the kernel configuration, these spaces for queue objects immediately follows the space set aside for the queue struct itself. The pointer in the queue struct, `pcReadFrom`, points to the next place in this space that should be read from; in other words, it points to the front of the queue. This address is also used by the function that copies data from the task into the queue space.
When the fault was injected into this particular variable, it was in essence telling the queue that its front was in a different location. In this instance, this happened to be a value in the original queue struct which keeps track of how big the items in the queue are, in bytes.

Actually, the value that was corrupted was fixed only a few lines later, when the code performs the check

```c
if ( pxQueue->u.pcReadFrom < pxQueue->pcHead )
```

Then the read pointer will be reset to point at the end of the queue data area.

```c
pxQueue->u.pcReadFrom = ( pxQueue->pcTail - pxQueue->uxItemSize );
```

However, even though the pointer is fixed by the kernel code, it’s too late; the size of the first copy of the queue has already been corrupted.

The reader may wonder why this would cause an error – one would think that reading the wrong value from only one of the queues would be fixed by voting. However, there is another side effect from this fault that we must be aware of. Since the size of queue has been corrupted, it also fails the check for

```c
if( pxQueue->u.pcReadFrom < pxQueue->pcHead )
```

which means that both the 1st and 3rd copies will have their pcReadFrom pointers reset. Because the queues only have one value in them at a time, when the code later reads from them, it will read 0 for 2 of the copies. And then when it votes, it will accept the wrong value.

From this discussion, we hope the reader can see how malicious silent data corruption can be, and how difficult it might be to track down some of these errors.

### 7.3.4 Application Single Points of Failure

Using the results from our fault injection experiments, we found some errors in our design of the protection scheme. These errors can be generalized as single points of failure. The following paragraphs will give some background on the development of our protection scheme, then outline some of the details of these errors.
**Background – Task parameters**  When we were first developing the FreeRTOS applications, we noticed that there was a single point of failure in passing parameters to tasks. This is because of the way that the kernel code sets up a fake initial stack frame for the task. We got around this by making a global array of parameters and passing in an index to the array. However, for one of the tasks in the application \texttt{rtos\_kuser}, the tasks which poll on a queue, the only parameter is a queue.

In previous iterations of the kernel protection scheme, we did not duplicate the kernel queues, and so didn’t bother changing this task to use a global array. In the most recent iteration, however, replicating the queues is part of the protection scheme. Unfortunately, we forgot to update this task, so this single point of failure remained. As a result, there were 6 errors that could have been prevented if we had properly corrected this weakness.

**Queue Errors**  Twice, the value \texttt{queue->pcWriteTo} was set to be much higher than the actual space allocated for the queue. In both cases, the address written to did not cause an error, but rather reading from the wrong place next, because there was only one copy of the queue. A similar error was caused by corrupting \texttt{queue->pcReadFrom}, and another from corrupting the address of the queue itself. Twice the fault caused the value pushed onto the queue to be incorrect. Because the voting was useless (only one copy of the queue), this caused an error.

**Measuring Execution Time**  Four errors were caused by injecting a fault into the value of \texttt{tStart}, which is read from the global timer prior to starting the scheduler. These errors were not reported by the application itself; rather the fault injection supervisor regarded the results as errors because the reported run time was either much too small or much too big.

When developing this application, we noticed that sometimes there would be errors caused because there were multiple copies of \texttt{tStart} and \texttt{tEnd}. We considered this was because calls to \texttt{XTime\_GetTime} did not return the same number even when called in very close succession. Because the time is such a high resolution, we could only call it once.

Unfortunately, we made the decision to not replicate the values \texttt{tStart} and \texttt{tEnd}, because there was only a single call to the function to populate them. The trick with this function is that
it modifies the variable by reference instead of returning a value directly. The solution to this is addressed in Section 7.3.5.

7.3.5 Updating the Protection Scheme

After finding all of these errors, we decided to fix the queue vulnerability and upgrade the protection scheme. Fixing the queue vulnerability was easy. Fixing the vulnerability related to the timing values took a little more work.

These fixes were implemented prior to the final results presented in the Section 7.2; this section is included because it is instructive to understand what changes were necessary to have a better protection scheme.

Replicating values after a call As we were testing the protection scheme, we realized that the values used to store the execution time represented a single point of failure. We were only calling the function once, and so the only copy of the timing values were vulnerable to being upset. We attempted to call the function XTime.GetTime more than once, but because this function reads from a system timer, each time it would return a different value. This kept the voting code from agreeing on a valid timing value, as each of the scalar values were not in agreement.

Normally the way we would solve this problem is copying the return value into multiple copies in the calling function. However, XTime.GetTime is a void function; the values that it “returns” come through a variable passed by reference that is modified before the function returns.

In order to support protecting these variables, we added the ability for COAST to copy the value of a function call parameter into its clones after the call completes. Listing 7.2 shows an example of what this might look like.

Code Listing 7.2: Example of Copy After Call

```assembly
call void @XTime_GetTime(i64* @tStart)
%loadOrig23 = load i64, i64* @tStart
store i64 %loadOrig23, i64* @tStart_DWC
store i64 %loadOrig23, i64* @tStart_TMR
```
Running more tests  After fixing these vulnerabilities, we re-ran the fault injection tests to see what would happen. The number of errors encountered in the COAST protected version was cut in half - there were 6 errors out of 1000 injections.

There were 3 times that the stack pointer saved in the stack frame was corrupted. Another time, the printing code was hit, causing the output to say that the execution time was much higher than it actually was.

The other 2 errors were caused by values in a TCB being corrupted. The first was blockQTasks_TMR[5]->xEventListItem.pxNext, though we had a hard time recreating this error. The other error was caused by changing the value of blockQTasks[5]->uxPriority. This is interesting because this value is used in this section of code:

```c
if(listCURRENT_LIST_LENGTH( &pxReadyTasksLists[pxCurentTCB->uxPriority] ) > ( UBaseType_t ) 1 )
{
    xSwitchRequired = pdTRUE;
}
```

You can see that this will cause the kernel code to read out of bounds of the array pxReadyTasksLists. We don’t really know what kind of behavior this will cause. However, the result is that the call to xQueueSendToBack in vBlockingQueueProducer fails immediately after, though only once. Many (about 1800) faults are reported at the end of the run.

It is interesting to see that the TCBs are still vulnerable, even when they are replicated in memory. All of the errors we have seen related to TCBs so far have to do with pointers.

7.4 Conclusion

While previous work has been able to automatically protect simpler bare metal applications, this work has demonstrated that protecting a full RTOS requires some extra care and intervention on the part of the application writer. Our work showed that by accounting for issues relating to assembly code, replicates of kernel objects, dynamic memory allocation and inlining, it is possible to protect a full RTOS using automated compiler techniques. Our fault injection results demonstrated a significant improvement to MWTF, with necessary overheads to run time and memory usage.
While our work targeted FreeRTOS, we feel many of these techniques could be extended to add SEU mitigation to other operating systems.
CHAPTER 8. CONCLUSIONS AND FUTURE WORK

The work presented in this thesis represents advancements in software fault tolerance that can be beneficial to a broad audience. First, the experiments in Chapters 3 and 4 show that COAST can be used to add fault tolerance to a wide variety of processor architectures and software programs. This is important as it shows that it is valuable to explore using automated tools when contemplating using a commercial microprocessor in a high-radiation environment. The knowledge gained from the analysis in Chapter 4 can help application designers know if using the COAST tool will be beneficial for their particular program, or how to reformulate their program so it can benefit more from protection by instruction replication.

Second, the fault injection framework PACIFIC described in Chapter 5 is a valuable contribution to the corpus of frameworks already in existence. What sets apart PACIFIC from other frameworks is that it is entirely open-source, it can support any platform that QEMU supports, and the injection methodology closely approximates the error distribution that often occurs in radiation tests. We hope that this framework will be used in future tests, both in place of and to augment high-energy radiation testing.

Lastly, the work described in Chapters 6 and 7 is an important example of how to use an automated tool in a mixed environment. That is, when the scope of replication is not straightforward, many rules and constraints have to be given to COAST to ensure the integrity of the SoR. These chapters go over our efforts to use the COAST tool to add protection to a real-time operating system, FreeRTOS. Beyond the example of formulating such a complex protection scheme, the analysis covers how this could be done in future projects. Experiments also show that this kind of protection is worth the effort in terms of the potential fault tolerance benefits.

As we were developing and improving the COAST tool, we identified a few avenues of research that could be interesting to pursue. The following sections will describe some ideas for how COAST can be expanded to be more featureful, and new ways in which the tool can be used.
8.1 Stack Protection

During the analysis phase of the fault injection experiments with FreeRTOS in Section 7.3.2, we discovered that a common vulnerability shared between benchmarks and configurations was the stack frame. Values in the stack frame are not known to COAST at compile time, and so they are not replicated and therefore vulnerable to upsets.

Protecting against errors in the stack has been studied a lot in the field of cybersecurity, where researchers are interested in protecting their code from malicious actors. In our case, the malicious actor is ionizing radiation. We found that there are a number of stack protection techniques that could be useful for preventing data corruption in the function call stack frames. Our initial foray into this area is described in more detail in Appendix B. Although these first tests were not successful, we believe it may be worth looking into in more detail.

The question that this work would attempt to answer is, can compiler-based techniques be used to protect values in the stack frame from corruption? Ideally this would be done in such a way that it is independent of the platform, though based on the many and varied function calling conventions that exist, that may not be possible. Regardless, being able to protect against data errors in the stack frame would help prevent control flow errors.

8.2 Compiling the C Library with COAST

During compilation, COAST is able to add replicated instructions to any functions for which it can access the function body. This necessarily precludes adding protection to library functions; in fact there is a flag for COAST that allows the user to specify how to treat such library functions\(^1\). The most common source of library functions that we used in our testing was the C standard library.

On most systems the C standard library exists in a pre-compiled state, and likely very optimized as well. This is great for most use cases. However, we thought it would be useful to recompile the C library using COAST, to add replicated instructions to these library calls as well. Of course, there are multiple implementations of the C standard library; it would make sense in

\(^1\)https://coast-compiler.readthedocs.io/en/latest/passes.html#command-line-parameters
our case to use the implementation that comes with clang, as that is our front-end compiler. The source code for the library is available as part of the large LLVM repo\(^2\).

If these functions were instead available, pre-compiled, in a protected state, we would be able to determine what kind of effect errors in library calls has on the total number of errors. Running different configurations with the normal and protected versions of the C library would make it possible to find what percentage of errors are in the application vs library calls.

If someone should attempt to compile the C library in this manner, they must be aware that often some standard library functions have been hand-optimized, and exist only as architecture specific assembly. These functions will need to be treated in a similar manner to the way we dealt with parts of the FreeRTOS kernel (Section 6.3.3). Fortunately, the C standard library is more compartmentalized than the FreeRTOS kernel, so these assembly functions should not have as large an impact on the scope of replication compared to the FreeRTOS kernel code.

8.3 Software Memory Scrubber

Hardware TMR mitigation efforts are often paired with what is called a memory scrubber. These scrubbers will go through memory and use the encoded parity bits to correct any single bit errors. There are also strategies for correcting multi-bit errors that happen using some external tool [65]. We believe it would be possible to implement something similar with software. This would need to be implemented as part of the COAST pass, because it needs access to the data structures that map original variables to their replicated copies.

Many of the platforms we have tested as part of validating COAST have more than one processor core, though we have only ever used one at a time. We believe it would be possible to use this extra processor core to act as a memory scrubber. The secondary processor would have a map that describes where all the data values are that need to be checked, and the offsets to their copies. It would continually iterate through this map, perform majority voting on each set of values, and change any that don’t agree with the majority.

There are a number of challenges that immediately present themselves when considering implementing a memory scrubber in this way. First, doing so goes against the platform-independent nature of COAST. Care must be taken so that COAST is altered in a way that makes

\(^{2}\)https://github.com/llvm/llvm-project
this scrubbing optional, and that it can be configured to support any platform that has multiple processors.

The second challenge that comes to mind is that of data coherence. Because the two processors would be reading from and writing to the same address space containing all the data variables, it is important to make sure all such accesses are atomic. For example, if the main processor is in the middle of storing recently-computed values to memory, and the secondary processor is trying to validate those values at the same time. Listing 8.1 shows an example of why coherency could be such a problem. Getting this right will probably be the most challenging part of implementing a memory scrubber.

8.4 Final Conclusion

Because of the work presented in this thesis, we now have a more comprehensive understanding of the strengths and weaknesses of automated software-based fault protection. We also now know that it is possible to add automated protection to something as complex as a real-time operating system, albeit with some configuration help from the user. We hope that the work presented here will be a starting point for others who wish to add software-based fault tolerance to their own programs.
BIBLIOGRAPHY


APPENDIX A. COMMON PITFALLS

In the course of our using and debugging code protected by COAST, we have identified a number of common pitfalls. Most of these have to do with improper consideration of pointers when the Scope of Replication does not cover the entirety of the program. Appendix A.1 contains a discussion of a problem that happens when an unprotected global variable is used with protected local variables inside a protected function. Appendix A.2 deals with problems with functions pointers.

A.1 The Linked List Problem

During the development stage of the protection scheme for the FreeRTOS applications, we ran into some difficulty maintaining the integrity of the scope of replication. The problem involved pointers being accessed both inside and outside the scope of replication. This appendix will give a brief treatment of the problem in a generalized form.

A.1.1 Linked List Basics

For this discussion, we assume a linked list with each node having a pointer to the next and previous elements in the list. The previous pointer on the head node is NULL, as is the next pointer of the last element. We will start with a linked list of two elements, shown in Figure A.1.

```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1000</td>
<td>0x1010</td>
</tr>
<tr>
<td>0x1010</td>
<td>NULL</td>
</tr>
<tr>
<td>NULL</td>
<td>0x1000</td>
</tr>
</tbody>
</table>
```

Figure A.1: Initial state of linked list
Now there is another linked list element, this one a global variable, that is inserted into the middle of the list. After the insertion, the list contents are as shown in Figure A.2.

![Figure A.2: Linked list after insertion](image)

If this element is removed again, then we expect the contents of the list to return to what they were in Figure A.1.

### A.1.2 Problems with the Scope of Replication

Section 6.3 introduces the concept of the Scope of Replication (SoR), which defines what variables and functions in a program will be replicated. The problem with this basic list operation comes when the initial linked list elements are triplicated and the global element is not. We start with three copies of the initial linked list with two elements, shown in Figure A.3.

![Figure A.3: Initial state of TMR’d linked list](image)
As before we attempt to insert the global list element from address 0x2000. The result is shown in Figure A.4.

At first it might look like everything worked fine, but close inspection of the addresses indicate that all is not correct. There is only one copy of the global element, which is pictured three times. If we remove the duplicates and adjust the arrows representing the pointers, we get the result in Figure A.5.

The next pointers on all of the first elements point to the same global element. As there is only one prev pointer, it points to the last element assigned to it; we assume the third copy is the last one written to memory. The last list element for the first two list copies have become orphaned, that is, no pointer points to those addresses. At this, only the third copy of the list is a valid list construct.

It gets even worse if we try to remove the middle element from the lists. Assuming the standard linked list removal function is something like

```c
this->next->prev = this->prev;
this->prev->next = this->next;
```
```c
this->next = NULL;
this->prev = NULL;
```
then we get the outcome depicted in Figure A.6.
Even though the middle element has been removed, there are still two head pointers that have it as the next element. Again, the third copy is correct, but the other two are very wrong. This is illustrated in Figure A.7.

One reason why this is such a problem is because COAST won’t detect a problem like this. The verification and voting code operates primarily on scalar values, not pointers. So when pointer values get out of sync, it is very hard to detect, because you couldn’t vote on them in the first place. A unit test written in C that illustrates the linked list problem can be found on the COAST repo¹.

![Figure A.7: Removing TMR’d list – revisited](image)

A.2 Function Pointers

There is a common paradigm in programming called “read-modify-write”. This is often used for making accesses to certain variables appear atomic, if there is no possibility of interruption between each of the read, modification, or write operations. Application writers must be careful when using this paradigm with COAST as it can backfire.

¹ [https://github.com/byuccl/coast/blob/master/tests/TMRregression/unitTests/linkedList.c](https://github.com/byuccl/coast/blob/master/tests/TMRregression/unitTests/linkedList.c)
Let us start with the code in Listing A.1. In this example, the function \texttt{intMath} represents an atomic operation on a value. The function \texttt{callFnPtr} calls \texttt{intMath} indirectly, and is the only function inside the Scope of Replication (SoR) in this file, indicated by the directive \_\_xMR. Although \texttt{callFnPtr} is replicated, there is only one copy of the value \texttt{x} passed into the function, as it comes from outside the SoR.

The expected output of the code is for it to say “fnX = 5”. If we use COAST to apply duplicate with compare (DWC) protection to this code, the output will actually be “fnX = 7”. This is because the function \texttt{intMath} is called twice, and the value \texttt{fnX} is updated twice, first to 5, and then to 7. We can see the problem more explicitly by examining the LLVM IR of the function \texttt{callFnPtr}, shown in Listing A.2.
If the “read-modify-write” pattern were inlined directly into this function, then COAST could reorder the instructions so that both loads happen before both stores. However, this can’t be done because the function updating the value of \( x \) is called indirectly. Even if the code wasn’t inlined, normally we could tell COAST to only call `intMath` once using the `-skipLibCalls` flag, but again, this doesn’t work because of the function pointer.

Application writers must be very careful to not introduce these kinds of patterns into their code. This is a very obvious example, when the output is not what is expected. Other times the side effects might involve data pointers, which are much harder to diagnose. We recommend that programmers avoid using function pointers if at all possible.
APPENDIX B. STACK PROTECTION

During the fault injection experiments on the FreeRTOS benchmarks, we observed several errors that were caused by corrupting the stack frame (see Section 7.3.2). We thought it beneficial to research potential methods for protecting against these kinds of stack errors.

B.1 Investigation Stack Protection

As mentioned in Section 7.3.2, the vulnerability we are trying to protect is the return address and frame pointer value, both stored in the stack frame. Figure B.1 gives a high-level description of the contents of a stack frame. The saved return address and frame pointer are single words in the frame.

Some previous research [66] used a CRC of the stack frame to check for corruption. There was also some work done using Linux threads and a watchdog with snapshots and DWC [67]. Although these methods are likely useful for the purposes for which they were designed, they don’t fit with the automated approach of COAST.

B.1.1 Clang/LLVM passes

In order to keep with the design tenets of COAST, the solutions we will explore in this section must be implemented as compiler passes. There are some passes for LLVM opt that may be useful. We will first list them, then examine their usefulness in this circumstance.

- ShadowCallStack
- SafeStack
- ControlFlowIntegrity
- -fstack-protector
- GOT protection
- FORTIFY_SOURCE
**ShadowCallStack**

This pass must be run with both the compiler and linker. It stores the return address of functions in a separate location, and restores it right before the function ends. This is intended to protect against attacks in a cyber-security context.

The only officially supported implementation for this is for `aarch64`, and our simulator is built around `ARM32`. Prior to LLVM 9.0 (COAST uses 7.0), an implementation was included for `x86_64` as well, but this was dropped because of inefficiency reasons.

This pass by itself doesn’t match with COAST, because it unconditionally replaces the return address with one stored elsewhere. The other word has approximately the same probability of being corrupted as the one in the stack frame, so some sort of voting would be necessary. Changing the ShadowCallStack pass is beyond the scope of this project, so this will not be used.

**Stack Protector**

This pass adds a special value at the beginning of the stack frame, right after the return address and stack pointer have been saved. This known value is then checked against the version
of it saved on the stack. If there is a mismatch, the stack has been corrupted. Note this will only check for corruption that affects the return address, etc, not local variables.

The effect of this pass is like applying DWC to the return address and frame pointer. However, the canary value must either be generated dynamically, or stored as a single global value. This introduces either additional run time overhead, or a single point of failure (the global value). That being said, using this pass could still be useful, though only for error detection, not correction.

Others

The SafeStack pass works by keeping the control flow part of the stack in a different memory location. This is intended to protect against buffer overflow attacks, which if protected this way, would only corrupt data values. However, it is unlikely that SafeStack will be much help against SEUs, because it's just moving some of the values that would have been in the stack to a different location; they would still have the same probability of being upset. Additionally, splitting the stack like this would introduce additional difficulties with the FreeRTOS task stack.

ControlFlowIntegrity seems to specifically guard against a vulnerability in C++ that uses virtual function operation tables. It seems this would be of limited use in our situation.

GOT protection would only be useful if we have an MMU to enforce read-only sections. But if that is available, that will prevent errors caused by trying to treat instruction memory as data memory.

Compiling with the flag `-D_FORTIFY_SOURCE=2` adds extra checks that might be able to detect buffer overflows. It seems the main way it accomplishes this is by changing calls of things like `strcpy` to `strcpy_chk`. So this could be useful, but not in many cases.

B.1.2 GCC builtin

It is possible to get the value of the current frame address & return address using a GCC builtin\(^1\). These builtins can get the frame & return address for any stack frame up from the current function call frame, but for this application, we only need the values for the current frame.

When converted to LLVM IR, the builtins map to the following functions:

\(^1\text{https://gcc.gnu.org/onlinedocs/gcc/Return-Address.html}\)
• `__builtin_return_address` – `@llvm.returnaddress`
• `__builtin_frame_address` – `@llvm.frameaddress`

We could use these builtins and give COAST a flag that will emit this code, and add in our own error checking, similar to what is used in `-fstack-protector`.

### B.1.3 Other

There is security-related research that has been done into protecting the stack that may be useful for this application [68]. It seems they enforce that all valid frame pointer values are aligned to a specific size, and if the FP ever gets unaligned, then they know there’s been an attack. This adds a 32.6% increase in memory overhead.

While this could be helpful, the implementation of this research does not seem to be publicly available, and there are likely better/less expensive ways of achieving the kind of FP protection we’re looking for.

### B.2 COAST Stack Protection

Based on the findings in Appendix B.1, we decided to implement rudimentary stack protection as an optional part of COAST. The implementation will be based on the builtin functions that get the current frame address and return address. The goal here is to increase the fault tolerance of these applications by eliminating the failure modes based on stack frame corruption.

In our recent radiation tests, we have been using a 32-bit ARM processor. The following explanation uses that architecture as an example, but most other architectures follow a similar calling convention.

When a function is called in ARM32, the assembly instruction used is `bl` (or variant). This instruction, “Branch and Link”, branches to the specified address and saves the next pc location to the Link Register (`lr`). It is common practice at the beginning of a function body (prologue) to save this register, as well as the stack pointer (`sp`), and any other registers that are callee-save, in part of the stack memory. This is much of what constitutes a stack frame (see Figure B.1).
B.2.1 Protecting a stack frame

In researching into ways to protect a stack frame, we discovered that there are a lot of ways that exist, but they are mostly targeted at cyber-security applications, not reliability. However, we did find some useful compiler intrinsics that can help us partially solve the problem.

We examined the LLVM IR generated from calling these intrinsics, and discovered that these builtins lower to LLVM intrinsics\(^2\). The one we want to use is called `llvm.returnaddress`\(^3\).

Although there exists an intrinsic to get the stack pointer of the current stack frame (`llvm.sponentry`), unfortunately it is not supported in the version of LLVM we use for our projects, and only implemented in the aarch64 back end. There is also a function for getting the address of the return address (`llvm.addressofreturnaddress`), which we will discuss later.

### Duplicate With Compare

The first thing to implement was simple Duplicate With Compare (DWC). Using the `llvm.returnaddress` intrinsic, we could read the value stored in the stack frame whenever we want. We inserted code at the entry basic block of each protected function that saves the return address for later comparing.

Then, at each return point in the protected functions, we inserted code that again calls `llvm.returnaddress` and compares the result to the stored value. If there is a difference, the code calls `FAULT_DETECTED_DWC`, which is a custom function that can be overwritten by the user. For testing we made it print an error message and then abort the program execution.

### Voting

While DWC is a good place to start, we would like this stack protection to also work the TMR protection, and so some sort of voting was necessary. The problem with the current approach is that the intrinsic function is effectively read-only; it tells you the value of the return address, but not where that value is stored. This is a problem because we would not be able to update the stored

\(^{2}\)https://llvm.org/docs/LangRef.html#intrinsic-functions

\(^{3}\)https://llvm.org/docs/LangRef.html#llvm-returnaddress-intrinsic
value, of which there is only one copy in memory that will be used, if the voting decided it was incorrect.

As mentioned before, there is another intrinsic called `llvm.addressofreturnaddress`\(^4\), which “returns a target-specific pointer to the place in the stack frame where the return address of the current function is stored.” This intrinsic is only implemented in the x86 and aarch64 back ends, so any work we do with this will be experimental and not used in any radiation tests, as it is not completely portable.

With the address of the return address, it is possible to vote on the return address of each function before returning to it. At the function prologue, we insert an additional call to `llvm.returnaddress` and store that in another automatic variable. Then at each return point we again insert instructions to compare the current and stored value of the return address. Except this time, it is possible to store directly to where the return address is located, because we have the address of the return address for the current stack frame.

### B.2.2 Experiment and Results

<table>
<thead>
<tr>
<th>Protection</th>
<th>Section</th>
<th>Runs</th>
<th>Faults</th>
<th>Errors</th>
<th>Timeout</th>
<th>Invalid</th>
<th>Error Rate</th>
<th>MWTF</th>
<th>Size (KB)</th>
<th>Cycle Count (avg)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unmitigated</td>
<td>cache</td>
<td>5000</td>
<td>0</td>
<td>122</td>
<td>97</td>
<td>0</td>
<td>2.44%</td>
<td>-</td>
<td>824</td>
<td>4.22 × 10^7</td>
</tr>
<tr>
<td></td>
<td>dcache</td>
<td>5000</td>
<td>0</td>
<td>162</td>
<td>242</td>
<td>0</td>
<td>3.24%</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>registers</td>
<td>5000</td>
<td>0</td>
<td>150</td>
<td>261</td>
<td>0</td>
<td>3.00%</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>TMR (kernel + application)</td>
<td>cache</td>
<td>10000</td>
<td>292</td>
<td>44</td>
<td>359</td>
<td>0</td>
<td>0.44% (5.55x ↓)</td>
<td>1.495x ↑</td>
<td>668 (0.81x)</td>
<td>1.56 × 10^8 (3.71x ↑)</td>
</tr>
<tr>
<td></td>
<td>dcache</td>
<td>10000</td>
<td>753</td>
<td>69</td>
<td>675</td>
<td>7</td>
<td>0.69% (4.70x ↓)</td>
<td>1.266x ↑</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>registers</td>
<td>10000</td>
<td>323</td>
<td>47</td>
<td>450</td>
<td>0</td>
<td>0.47% (6.38x ↓)</td>
<td>1.721x ↑</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>TMR + stack protection</td>
<td>cache</td>
<td>10000</td>
<td>294</td>
<td>67</td>
<td>333</td>
<td>2</td>
<td>0.67% (3.64x ↓)</td>
<td>1.055x ↓</td>
<td>734 (0.89x)</td>
<td>1.62 × 10^8 (3.84x ↑)</td>
</tr>
<tr>
<td></td>
<td>dcache</td>
<td>10000</td>
<td>763</td>
<td>47</td>
<td>644</td>
<td>4</td>
<td>0.47% (6.89x ↓)</td>
<td>1.795x ↑</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>registers</td>
<td>10000</td>
<td>195</td>
<td>49</td>
<td>360</td>
<td>0</td>
<td>0.49% (6.12x ↓)</td>
<td>1.594x ↑</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

We added stack protection to the FreeRTOS benchmark `rtos_kUser.xMR`, the version with the entire kernel and application protected. We performed fault injection on this configuration at the same time as the other FreeRTOS benchmark configurations. The results of this experiment are shown in Table B.1, which contains results from other configurations for comparison (similar to Table 7.1).

\(^4\)https://llvm.org/docs/LangRef.html#llvm-addressofreturnaddress-intrinsic
The results from this experiment indicate that the overhead required for stack protection makes the overall fault tolerance improvement worse than the configuration without stack protection. It is possible another configuration might benefit more from stack protection than this one, but it is not likely to be by very much, because of the very narrow margin seen in Table B.1.

If there were a more efficient way to implement stack protection, or if the LLVM builtin that gives the address on the stack where the return address is stored were more portable, perhaps we could see different results. As is stands, this area could benefit from more investigation, but it is not a high priority.
APPENDIX C. PROTECTING A FILE SYSTEM

During the creation of the protection scheme for the FreeRTOS applications, we also looked into protecting a file system. One of the optional add-ons for FreeRTOS is a FAT file system, referred to as FreeRTOS+FAT\(^1\). We thought that attempting to add automated protection to a file system would give more important insights about adding automated protection to complex systems. The application we used for this experiment was a demo application distributed with the source code. It was originally designed to run on the Windows port of FreeRTOS, but we modified it to run on the Xilinx Zynq A9 port.

C.1 Applications

The application, as originally written, hosts a TCP and/or HTTP server which accepts commands from an external client. These commands instruct the file system to do certain things. In our version of the application, we have removed the server control of the file system, and leave only the automated tests that are run at start up. This way we can test for correct operation of the file system without having to worry about interaction with an external client.

The current tests are being done using QEMU, and so we also disabled the part the interfaces with the SD card. All tests are done using a RAM Disk, which is a file system that exists solely in RAM, and hence is volatile. The application creates several example files, then does various operations on them and checks for correctness. This is exactly the kind of application we want to use for testing COAST, as it very quickly detects if there are any errors.

C.2 Creating a Protection Scheme

This section will give details on the protection scheme and important lessons learned from this endeavor. The goal here was to have multiple copies of each of the files stored in memory.

\(^1\)https://www.freertos.org/FreeRTOS-Plus/FreeRTOS_Plus_FAT/index.html
There were a few changes to COAST code that had to made in order to properly support this new application. These changes include removing unused functions and adding support for variadic functions. This section will also discuss replication function calls, return values, and other required considerations for replicating the correct values.

C.2.1 Removing Unused Functions

The rationale behind removing unused functions is twofold. First, because many of the functions have their names and signatures changed, both the original and protected functions will exist in the code after the main part of COAST has finished. In most cases, this is not desirable. There are some instances where a function can safely be inside and outside the Scope of Replication (SoR), but these are more rare than common. COAST should remove the original copy of the function so that only the new version remains. Any exceptions should be checked carefully.

The second reason for removing unused functions is to save on code size. The cost for COAST protection in terms of code size is quite high, at least 2-3x for each function protected. On embedded platforms, having code that fits well in memory is important. In the case of the FreeRTOS applications, many of the source files contain functions that are not used by our applications. Removing these extra “API” functions that were not used in the first place can save extra space.

It should be noted that some parts of the FreeRTOS kernel are written in assembly, and cannot be protected with COAST. Some of these assembly functions reference other functions defined in C source files, which means that COAST cannot detect their uses and will try to remove them unless instructed otherwise. We created an in-code directive, __COAST_VOLATILE, which instructs COAST to not remove the function.

C.2.2 Variadic Functions

A variadic function in C is one which can take a variable number of arguments. printf is an example from the C library of a function which can take more than one argument. Application writers can also make use of this functionality with some macros found in stdarg.h.

Up until this point, none of our tests had code with any variadic functions which needed to be protected. With the FAT file system code, the function ff_fprintf needed to be protected to
maintain the integrity of the SoR. We discovered that COAST did not copy any function arguments beyond what were explicitly stated in the function definition. It also did not respect the isVarArg property. Both of these bugs have now been fixed.

The current COAST code will only replicated named function arguments. Any additional arguments will only have one copy passed to the callee. This is because of the way that the compiler emits assembly for the va_start and va_end macros. At this time, it does not seem feasible to replicate variadic arguments.

C.2.3 Function Return Values

There were several instances where multiple values had to be returned from a kernel/file system function. In Section 6.3, we mentioned why creating tasks and creating timers are treated differently in our protection scheme. A similar rule applies here: sometimes file buffers or the like are directly returned from a function, other times they are modified by argument reference. The general rule is, if there is a file handle being returned, those return values need to be replicated.

By default, calls to malloc() and printf-style functions are not replicated. However, several times we had to make multiple calls to malloc() using the COAST directive MALLOCP_WRAPPER_CALL. In addition, sometimes a call to esprintf would be used to print a formatted string into a file. Because our protection scheme allows for multiple file copies, we had to replicate calls to this function also, using the COAST directive GENERIC_COAST_WRAPPER.

C.2.4 Replicating Function Calls

Though not common, there are a couple of functions that need to be called more than once. Previously, the protection scheme treated calls to functions such as queue creation this way, but these have since been added to the list to return multiple values instead.

The functions that need to be called more than once are library functions. The first is memcmp. It shows up quite often in the self-checking tests, and is used to make sure the contents of a file are what they are expected to be.

One of the problems with this is that it lacks the granularity that we desire with COAST. Say you have a section of memory that looks like in Listing C.1. When comparing this region of
memory against a golden copy, COAST will replicate each call to `memcmp` and then vote on the result of these calls. If there is a mismatch in the results, then a fault has been detected. However, this only reports that there’s an error in the entire memory region. Some calls to `memcmp` compare regions that are over 1000 bytes in length.

Code Listing C.2: Alternate `memcmp` implementation

```c
int my_memcmp(const void* lhs, const void* rhs, size_t count) {
    // NOTE: doesn't completely conform to POSIX
    for (size_t i = 0; i < count; i++, lhs++, rhs++) {
        if (*lhs < *rhs) {
            return -1;
        } else if (*lhs > *rhs) {
            return 1;
        }
    }
    return 0;
}
```

Contrast this to an implementation that does not call a library function, but instead compares one data value at a time, like in Listing C.2. If COAST has access to the function body, then it will insert voting instructions for each comparison (lines 3 and 5). This results in much greater granularity over which memory location it is that has the fault.

So protecting `memcmp` and `memchr` helps with the fault coverage, but still kind of masks where exactly the errors are. It may be worth looking into doing some kind of library interpositioning to replace certain C standard library calls with other implementations that COAST can protect better.
C.2.5 Other protection rules

In addition to the functions which were already marked to be protected in the other FreeRTOS applications, there are several other functions which had to be added to that list.

One such set of functions deal with storing data specific to each “thread,” which in this context is a task. These functions load and store data in the current TCB. These functions were not used in our other applications previous to this, but is part of the FreeRTOS kernel, not the FAT extension. The one that reads is marked to return multiple values, because there are multiple copies of the TCB in memory.

C.3 Results

The results presented in this section followed the same experimental procedure as outlined in Section 7.1. Table C.1 gives the results, in the same format as presented in Table 7.1.

<table>
<thead>
<tr>
<th>Protection</th>
<th>Section</th>
<th>Runs</th>
<th>Faults</th>
<th>Errors</th>
<th>Timeout</th>
<th>Invalid</th>
<th>Error Rate</th>
<th>MWTF</th>
<th>Size (KB)</th>
<th>Cycle Count (avg)</th>
</tr>
</thead>
<tbody>
<tr>
<td>rtos_fat_demo (Unmitigated)</td>
<td>cache 4000</td>
<td>0</td>
<td>82</td>
<td>112</td>
<td>2.05%</td>
<td>-</td>
<td>-</td>
<td>1258</td>
<td>2.19 \times 10^8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>dcache 4000</td>
<td>0</td>
<td>342</td>
<td>41</td>
<td>8.55%</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>registers 6000</td>
<td>0</td>
<td>33</td>
<td>346</td>
<td>0.55%</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TMR (kernel + application)</td>
<td>cache 6000</td>
<td>559</td>
<td>24</td>
<td>298</td>
<td>0.40%</td>
<td>(5.13x \downarrow)</td>
<td>1.304x \uparrow</td>
<td>1424 (1.13x)</td>
<td>5.23 \times 10^8 (3.93x \uparrow)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>dcache 97000</td>
<td>19496</td>
<td>20</td>
<td>1001</td>
<td>0.02%</td>
<td>(414.68x \downarrow)</td>
<td>105.527x \uparrow</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>registers 7000</td>
<td>485</td>
<td>29</td>
<td>302</td>
<td>0.41%</td>
<td>(1.33x \downarrow)</td>
<td>2.960x \downarrow</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TMR (app only)</td>
<td>cache 4000</td>
<td>12</td>
<td>75</td>
<td>91</td>
<td>1.88%</td>
<td>(1.09x \downarrow)</td>
<td>1.835x \downarrow</td>
<td>952 (0.76x)</td>
<td>2.67 \times 10^8 (2.01x \uparrow)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>dcache 2000</td>
<td>2</td>
<td>171</td>
<td>15</td>
<td>8.55%</td>
<td>(1.00x \downarrow)</td>
<td>2.007x \downarrow</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>registers 4000</td>
<td>1</td>
<td>24</td>
<td>224</td>
<td>0.60%</td>
<td>(1.09x \uparrow)</td>
<td>2.189x \downarrow</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The results show that it is critical to protect the kernel, and not just the application part of the code. This is different from the other FreeRTOS benchmarks tested earlier in this thesis, in that the application-only protection scheme was as good as or better than the total protection scheme.
APPENDIX D. ARM SEMIHOSTING

This appendix covers some of the low-level details required to get the PACIFIC framework mentioned in Chapter 5 working properly. One of the requirements for PACIFIC to work was to make sure that we could see the output of the programs being run. Since the emulator of choice for the fault injection setup is QEMU, and our target processor the ARM Cortex-A9, this meant that we would have to use Semihosting¹.

D.1 Rationale

Since the fault injection setup uses a GDB instance to change values in the program, it would have been possible to simply introspect the return variable’s value instead of waiting for it to be printed. However, this would then miss any invalid output that could happen as a result of a Single Event Upset (SEU). The `printf` function (and derivatives) require a decent amount of processing time to complete, and so a good target for SEUs.

Another reason to require output is so that any additional printing that happens can be captured. Some benchmarks print error messages when they occur, which gives us more precise information about the location of the error. In the case of FreeRTOS, sometimes it will print assertion errors. These are also quite useful when figuring out the effect of each fault injected.

Semihosting means “a mechanism that enables code running on an ARM target or emulator to communicate with and use the Input/Output facilities on a host computer”². The main reason we desired this was to see `printf` output, but semihosting can handle inputs as well.

“Semihosting is implemented by a set of defined software instructions that generate exceptions from program control. The application invokes the appropriate semihosting call and the debug agent then handles the exception. The debug agent provides the required communication with the host”².

¹https://static.docs.arm.com/100863/0300/semihosting.pdf
²see https://developer.arm.com/docs/100863/latest page 5
D.2 Normal Usage

gcc has a built-in way of compiling an executable for semihosting, using a spec file: 
--specs=rdimon.specs. This file specifies linking with a specific library librdimon.a, and
has specific startup code that sets up the parameters for the semihosting exception handlers.

D.3 Compiling with the COAST toolchain

The toolchain for compiling code through COAST is not as simple as gcc. We use clang
as the front-end so we can get LLVM IR emitted from the front-end going into the optimizer. While
clang does not support spec files, we do use the target specific gcc linker for the final step, so that
means that using semihosting is still possible.

Linking with semihosting libraries  In order to get semihosting to work, we have to link the
executable against a library that has the correct definitions for system calls that will work properly
with the host I/O. After digging around in the file rdimon.specs, we found that we need to
link against a specific library, librdimon.a (for static linking). To get a library compatible with
our target platform, we installed the Ubuntu package libnewlib-arm-none-eabi. This package
contains libraries for linking against bare metal ARM applications.

However, after doing this, we were still not able to see printf output. That’s when we
realized we needed to change the startup code as well.

Changing the startup code  The specs file rdimon.specs tells the linker to link against a file
called crt0.S. This file contains startup code that sets up things for the semihosting interface
to work. However, in our case, the file xil_crt0.S was already providing all of the symbol
definitions found in crt0.S. We couldn’t simply replace the file because it has other important
startup code specific to our Xilinx part.

If you look at the startup code file for ARM from newlib, you can see some of the places
where they take into account semihosting. We adopted some of the code from this file and inte-

\[\text{crt0.S on GitHub}\]
grated it into the startup code for our project, as can be seen in our version of the file. Listing D.1 shows the code we added to xil_crt0.S to get it work with semihosting.

Code Listing D.1: xil_crt0.S

```assembly
; Issue Angel SWI to read stack info.
movs r0, #0x16 ; known from disassembly
adr r1, .LC0 ; Point at ptr to 4 words to receive data.
; AngelSWIAsm AngelSWI
svc #0x123456 ; Angel Semihosting ARM mode
ldr r0, .LC0 ; Point at values read.

... ; Need to set up standard file handles.
bl initialise_monitor_handles

...

.LC0:
   .word HeapBase
.LC30:
   .word CommandLine
   .word 255
.LC31:
   .word _end

; Workspace for Angel calls.
.data
; Data returned by monitor SWI.
global __stack_base__
HeapBase: .word 0
HeapLimit: .word 0
__stack_base__: .word 0
StackLimit: .word 0
CommandLine: .space 256,0 ; Maximum length of 255 chars handled.
```

---

4xil_crt0.S on GitHub