Hardware and Software Fault-Tolerance of Softcore Processors Implemented in SRAM-Based FPGAs

Nathaniel Hatley Rollins
Brigham Young University - Provo

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

Part of the Electrical and Computer Engineering Commons

BYU ScholarsArchive Citation
Rollins, Nathaniel Hatley, "Hardware and Software Fault-Tolerance of Softcore Processors Implemented in SRAM-Based FPGAs" (2012). Theses and Dissertations. 2998.
https://scholarsarchive.byu.edu/etd/2998

This Dissertation 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 scholarsarchive@byu.edu, ellen_amatangelo@byu.edu.
Hardware and Software Fault-Tolerance of Softcore Processors Implemented
in SRAM-Based FPGAs

Nathaniel H. Rollins

A dissertation submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

Michael J. Wirthlin, Chair
Brent E. Nelson
Doran K. Wilde
Brad L. Hutchings
David A. Penry

Department of Electrical and Computer Engineering
Brigham Young University
April 2012

Copyright © 2012 Nathaniel H. Rollins
All Rights Reserved
ABSTRACT

Hardware and Software Fault-Tolerance of Softcore Processors Implemented in SRAM-Based FPGAs

Nathaniel H. Rollins
Department of Electrical and Computer Engineering
Doctor of Philosophy

Softcore processors are an attractive alternative to using expensive radiation-hardened processors for space-based applications. Since they can be implemented in the latest SRAM-based FPGA technologies, they are fast, flexible and significantly less expensive. However, unlike ASIC-based processors, the logic and routing of a softcore processor are vulnerable to the effects of single-event upsets (SEUs). To protect softcore processors from SEUs, this dissertation explores the processor design-space for the LEON3 softcore processor implemented in a commercial SRAM-based FPGA. The traditional mitigation techniques of triple modular redundancy (TMR) and duplication with compare (DWC) and checkpointing provide reliability to a softcore processor at great spatial cost. To reduce the spatial cost, terrestrial ASIC-based processor protection techniques are applied to the LEON3 processor. These techniques come at the cost of time instead of area. The software fault-tolerance techniques used to protect the logic and routing of the LEON3 softcore processor include a modified version of software implemented fault tolerance (SWIFT), consistency checks, software indications, and checkpointing.

To measure the reliability of a mitigated LEON3 softcore processor, an updated hardware fault-injection model is created, and novel reliability metrics are employed. The improvement in reliability over an unmitigated LEON3 is measured using four metrics: architectural vulnerability factor (AVF), mean time to failure (MTTF), mean useful instructions to failure (MuITF), and reliability-area-performance (RAP). Traditional reliability techniques provide the best reliability: DWC with checkpointing improves the MTTF and MuITF by almost 35x and TMR with triplicated input and outputs improves the MTTF and MuITF by almost 6000x. Software fault-tolerance provides significant reliability for a much lower area cost. Each of these techniques provides greater processor protection than a popular state-of-the-art rad-hard processor.

Keywords: FPGA, reliability, SEU, radiation effects, softcore processors, software fault-tolerance, checkpointing, consistency checks, control-flow monitoring, software indicators, duplication with compare, TMR, AVF, MTTF, MuITF, probability modeling, hardware fault-injection
ACKNOWLEDGMENTS

This document represents many years of hard work, long hours, late nights, non-existent weekends, blood, sweat, and tears. The completion of this work would not have been possible without the help of many people to whom I am sincerely grateful.

Most importantly, I am indebted to my family. Without the support and encouragement of my wife Brianna during these past few years I would not have succeeded. I thank her and our two daughters, Ella and Jacqueline, from the bottom of my heart. I am also grateful to my two sisters and their families for the many years of support since I started attending BYU. I am grateful to my father for inspiring me from childhood to be fascinated with space, and I am grateful to my mother for believing that I am smarter than I am, and for believing that I can do anything.

Many professors in the Electrical and Computer Engineering Department at BYU, have been an essential part of my success. My advisor, Dr. Michael Wirthlin, to me, has been a professor, an advisor, a mentor, and a friend over the past eleven years, during my undergraduate, Master’s, and Doctorate work at BYU. I am grateful for his guidance, insight, and patience. Drs. Brent Nelson, David Penry, Doran Wilde, Brad Hutchings, and Michael Jensen have also been great sources of assistance. I thank all of these professors.

I have had many colleagues over the years in the reconfigurable computing lab who have provided assistance, support, and friendship. In particular, I would like to thank Chris Lavin, Brian Pratt, Jon-Paul Anderson, Welson Sun (Sun Hua), Keith Morgan, Daniel McMurtrey, Eric Johnson, Derrick Gibelyou, Joshua Monson, Megan Fuller, and Kevin Ellsworth.

I also would like to acknowledge the major sources of funding for the research that went into this dissertation. Gary Swift at Xilinx and others in the Xilinx Radiation Test Consortium (XRTC) have been a great source of help and financial assistance. In particular, I am grateful to them for twice financing heavy-ion testing at the Texas A&M cyclotron. This research was also supported by the I/UCRC Program of the National Science Foundation under Grant No. 0801876 through the NSF Center for High-Performance Reconfigurable Computing (CHREC).
TABLE OF CONTENTS

LIST OF TABLES .................................................................................. vi

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

Chapter 1 Introduction ........................................................................ 1
  1.1 Motivation ...................................................................................... 1
  1.2 Processor Reliability Design-Space .............................................. 2
  1.3 Dissertation Contributions .......................................................... 4
  1.4 Dissertation Organization ............................................................. 6

Chapter 2 Protecting Softcore Processors in Space .............................. 8
  2.1 Effects of Radiation on Electronics ................................................. 9
  2.2 Single Event Effects in FPGA Configuration Memory .................. 11
    2.2.1 Failure Modes in FPGA Designs ........................................... 12
    2.2.2 Traditional FPGA Design Protection .................................... 13
  2.3 Single Event Effects in FPGA User Memory .................................. 15
  2.4 LEON3 Softcore Processor ........................................................... 16
    2.4.1 LEON3 Processor Architecture ............................................ 17
    2.4.2 LEON3 Area Costs ............................................................ 20
    2.4.3 Modifications to the LEON3 ................................................. 20
  2.5 Summary ....................................................................................... 22

Chapter 3 Traditional Softcore Processor Fault-Tolerance .................... 24
  3.1 Fault Recovery Through Checkpointing ....................................... 24
    3.1.1 Applying Checkpointing to Softcore Processors ...................... 25
    3.1.2 Endless Detection Loops: dloops .......................................... 27
    3.1.3 Probability of dloop Occurrence .......................................... 27
    3.1.4 Preventing dloops ............................................................... 28
    3.1.5 Checkpoint Interval ............................................................ 30
  3.2 Traditional Softcore Processor Mitigation Techniques ................... 31
    3.2.1 DWC with Checkpointing ..................................................... 32
    3.2.2 TMR with Checkpointing ..................................................... 33
    3.2.3 LEON3 Modifications ........................................................ 35
  3.3 Cost Comparisons for Traditional Softcore Processor Mitigation ...... 36
    3.3.1 Area Costs ......................................................................... 36
    3.3.2 Performance Costs ............................................................. 37
  3.4 Summary ....................................................................................... 38

Chapter 4 Software Detection Techniques for Softcore Processors ............ 39
  4.1 Modified SWIFT .......................................................................... 39
    4.1.1 Register File CDWC ............................................................ 40
    4.1.2 Control-Flow Monitoring ...................................................... 42
4.2 Consistency Checks ........................................ 52
  4.2.1 Functional Unit Tests ............................... 54
  4.2.2 Memory Tests ......................................... 56
  4.2.3 Costs .................................................. 58
  4.2.4 Undetected Upsets .................................. 59
4.3 Software Indicators .................................. 60
  4.3.1 Program Hangs ....................................... 60
  4.3.2 Illegal Traps .......................................... 60
  4.3.3 Costs .................................................. 61
  4.3.4 Undetected Upsets .................................. 61
4.4 BRAM Parity and Memory Scrubbing ................. 62
  4.4.1 BRAM Parity ......................................... 62
  4.4.2 Memory Scrubbing .................................... 62
  4.4.3 Costs .................................................. 63
  4.4.4 Undetected Upsets .................................. 63
4.5 Undetected Upsets .................................. 63
4.6 Summary .................................................. 64

Chapter 5  Reliability Metrics and Models for Softcore Processors .... 66
  5.1 Reliability Metrics for Softcore Processors ......... 66
    5.1.1 Architectural Vulnerability Factor (AVF) ........ 67
    5.1.2 MTTF ............................................... 69
    5.1.3 MuITF .............................................. 70
    5.1.4 RAP ................................................ 72
  5.2 A Hardware Fault-Injection Model for Softcore Processors .... 73
    5.2.1 Hardware Fault-Injection for Unmitigated Softcore Processors .... 74
    5.2.2 Hardware Fault-Injection for Fault-Tolerant Softcore Processors .... 76
    5.2.3 Benchmark Programs .............................. 78
  5.3 Summary .................................................. 81

Chapter 6  Reliability of the LEON3 Softcore Processor ............ 82
  6.1 Unmitigated LEON3 .................................... 82
  6.2 Reliability of Hardware-Based Softcore Processor Mitigation .... 85
  6.3 Reliability of Individual Software Fault-Tolerance Techniques .... 87
  6.4 Reliability of Full Software Fault-Tolerance ............. 90
    6.4.1 Reliability Comparisons ......................... 92
    6.4.2 Cost Comparisons .................................. 94
  6.5 Processor Design-Space Comparisons .................... 95
  6.6 Radiation Hardened Processor Comparisons ............. 98
  6.7 Summary .................................................. 100

Chapter 7  Conclusion ........................................ 101
  7.1 Future Work ............................................ 101
    7.1.1 Bitstream Scrubbing Detection ................... 102
    7.1.2 DWC with Parity and Software Indicators .......... 103
7.2 Concluding Remarks .................................................. 103

REFERENCES ............................................................... 106

Appendix A  FPGA Configuration Memory Failure Modes in Routing ........ 116

Appendix B  A Comparison of Fault-Tolerant Memories in SRAM-Based FPGAs . 118
   B.1 Fault Mechanisms in Memories .................................... 119
      B.1.1 Upsets in BRAMs ............................................. 121
      B.1.2 Upsets in LUTRAMs ........................................... 121
      B.1.3 Upsets in SRLs ................................................ 122
   B.2 Memory Fault-Tolerant Techniques ............................... 123
      B.2.1 TMR .......................................................... 123
      B.2.2 Duplication with EDC ....................................... 124
      B.2.3 ECC .......................................................... 126
   B.3 Memory Scrubbing ................................................. 127
      B.3.1 Scrubbing BRAMs ............................................. 128
      B.3.2 Scrubbing LUTRAMs ......................................... 130
      B.3.3 Scrubbing SRLs ............................................... 132
   B.4 Memory Fault-Tolerance Comparisons ........................... 133
      B.4.1 BRAM Fault-Tolerant Technique Comparisons ............ 135
      B.4.2 LUTRAM Fault-Tolerant Technique Comparisons ........ 137
      B.4.3 SRL Fault-Tolerant Technique Comparisons .............. 140
   B.5 Summary ........................................................... 141

Appendix C  Ideal Checkpointing Intervals for ASIC-Based Processors ........ 143
   C.1 Simplest 1st Order Approximation of an Optimal Checkpoint Interval .... 144
   C.2 Nakagawa’s Ideal Checkpointing Interval Probability Model ............ 148
       C.2.1 Finding the Optimal CSCP Interval ........................ 149
       C.2.2 Finding the Optimal CCP Interval .......................... 154

Appendix D  Ideal Checkpointing Intervals for Softcore Processors ........... 158
   D.1 Ideal Checkpoint Interval for a Duplicated Softcore Processor .......... 160
   D.2 Ideal Checkpoint Interval for a Softcore Processor Using Software Fault-Tolerance
       Techniques ........................................................ 167
       D.2.1 Finding the Optimal CSCP Checkpoint Interval ............ 167
       D.2.2 Finding the Optimal Consistency Check Interval .......... 174

Appendix E  Optimal Number of Consistency Checks .......................... 180

Appendix F  Code Duplication Overhead in LEON3 ............................ 182
   F.1 Architectural Changes Required by Assembly Code Duplication .......... 182
   F.2 Costs Of Assembly Code Duplication ................................ 183
       F.2.1 ISA Changes .................................................. 184
       F.2.2 Performance Costs .......................................... 185
## LIST OF TABLES

2.1 Memory bit percentages for the Xilinx Virtex4 FX60 FPGA. ........................................ 15
2.2 Area costs for the LEON3 processor and each of its components. ................................. 20

3.1 Area cost comparison between the unmitigated LEON3 and a LEON3 with rollback checkpointing implemented designs. ......................................................... 26
3.2 Area cost comparison for the unmitigated and traditionally mitigated LEON3 processor designs. ........................................................................................................... 36
3.3 Average performance and code size costs for the unmitigated and traditionally mitigated LEON3 processor designs. ................................................................. 37

4.1 Area costs of the unmitigated LEON3 processor, a LEON3 with rollback checkpointing, and a LEON3 with register file CDWC and rollback checkpointing. ... 42
4.2 Code increase and performance cost of control-flow monitoring. .................................. 50
4.3 Area costs of a LEON3 processor protected with control-flow monitoring and rollback checkpointing. ................................................................. 51
4.4 Functional unit tests run during a consistency check. ....................................................... 55
4.5 Memory write/read tests performed during a consistency check. ..................................... 56
4.6 Performance cost of consistency checks. ................................................................. 59
4.7 Area costs of a LEON3 processor protected with consistency checks and rollback checkpointing. ................................................................. 59
4.8 Area costs of a LEON3 processor protected with software indicators and rollback checkpointing. ................................................................................. 61
4.9 Area costs of a LEON3 processor protected with parity, memory scrubbing, and rollback checkpointing. ................................................................. 63
4.10 Area costs of a LEON3 softcore processor protected with various fault-tolerance techniques. ......................................................................................... 65
4.11 Performance cost of software fault-tolerance. ........................................................... 65

5.1 Memory usage report for each of the testbench programs .............................................. 79
5.2 Program code usage report for the benchmark programs. ............................................... 80
5.3 Instruction mix report for each of the testbench programs. ........................................... 80

6.1 Complete hardware fault-injection results for the unmitigated LEON3 processor and its components. ................................................................. 83
6.2 Architectural vulnerability factors (AVFs) and relative mean instruction to failure (rMITF) for each LEON3 processor component. ............................................. 85
6.3 Fault-injection results for the unmitigated and hardware mitigated LEON3 processors. .............................................................................................................. 86
6.4 Comparison of AVF, MTTF, and MuITF of an unmitigated LEON3 against the traditionally mitigated LEON3 processors. ......................................................... 87
6.5 The protection that TMR and DWC with checkpointing provide to each LEON3 processor component. ................................................................. 88
6.6 Fault-injection comparisons for the unmitigated LEON3 processor and LEON3 processor protected with each of the individual detection techniques and rollback checkpointing. ........................................ 89

6.7 Comparison of the architectural vulnerability factors (AVFs), relative mean time to failure (MTTF), relative mean useful instruction to failure (multi TF), and RAP of an unmitigated LEON3 against the LEON3 processor protected with each of the software fault-tolerance techniques individually. ........................................ 90

6.8 The protection that each individual fault-tolerance technique provides to each LEON3 processor component. ................................................................. 91

6.9 The four combinations of detection techniques used in this work. Each combination includes rollback checkpointing for upset recovery. ......................... 92

6.10 Comparison of fault-injection results for the unmitigated LEON3 processor and LEON3 processor protected with combinations of software detection techniques and rollback checkpointing. ........................................ 93

6.11 Comparison of the architectural vulnerability factors (AVFs), relative mean time to failure (MTTF), relative mean useful instruction to failure (multi TF), and RAP of an unmitigated LEON3 against the LEON3 processor protected with combinations of software fault-tolerance techniques. ........................................ 94

6.12 The protection that different combinations of software fault-tolerance techniques provide to each LEON3 processor component. ........................................ 95

6.13 Area costs of a LEON3 processor protected with different combinations of software detection techniques and rollback checkpointing. ......................... 96

6.14 Performance costs of a LEON3 processor protected with different combinations of software detection techniques and rollback checkpointing. ......................... 96

6.15 Failure rates of two popular rad-hard processors using an upset rate of a GEO orbit at solar minimum. ................................................................. 99

6.16 Failure rates of LEON3 softcore processors using an upset rate of a GEO orbit at solar minimum. ................................................................. 99

B.1 The cost of fault-tolerance in terms of BRAM bits. ......................... 135

B.2 Non-triplicated, non-scrubbing reliability and cost comparisons for BRAMs. ......................... 136

B.3 Triplicated, non-scrubbing reliability and cost comparisons for BRAMs. ......................... 137

B.4 Triplicated, scrubbing reliability and cost comparisons for BRAMs. ......................... 138

B.5 Non-triplicated, non-scrubbing reliability and cost comparisons for LUTRAMs. ......................... 138

B.6 Triplicated, non-scrubbing reliability and cost comparisons for LUTRAMs. ......................... 139

B.7 Triplicated, scrubbing reliability and cost comparisons for LUTRAMs. ......................... 139

B.8 Non-triplicated, non-scrubbing reliability and cost comparisons for SRLs. ......................... 140

B.9 Triplicated, non-scrubbing reliability and cost comparisons for SRLs. ......................... 141

C.1 Comparison of optimal checkpoint interval ($T^*$) with approximated optimal checkpoint interval ($\hat{T}$). ................................................................. 148

C.2 Comparison of optimal checkpoint interval ($T^*$) with approximated optimal checkpoint interval ($\hat{T}$). ................................................................. 153

C.3 Percentage overhead due to checkpointing, and percentage of execution time saved with checkpointing compared to without. ................................. 154
D.1 Comparison of optimal checkpoint interval ($T^*$) with approximated optimal checkpoint interval ($\tilde{T}$) for a duplicated softcore processor. .................................................. 165
D.2 Percentage overhead due to checkpointing in a duplicated softcore processor, and percentage of execution time saved with checkpointing compared to without. .... 166
D.3 Comparison of optimal checkpoint interval ($T^*$) with approximated optimal checkpoint interval ($\tilde{T}$) for a softcore processor. .................................................. 173
D.4 Percentage overhead due to checkpointing, and percentage of execution time saved with checkpointing compared to without. .................. 173
D.5 Comparison of optimal consistency check interval ($T_{cc}^*$) with approximated ideal consistency check interval ($\tilde{T}_{cc}$) for a softcore processor. .................. 179

F.1 Performance cost of code duplication. .................................................. 186

G.1 Comparison of fault-injection results for a software fault-tolerant LEON3 processor with and without constrained placement. ............................................ 188

H.1 Fault-injection results for protected and unprotected processors. ............ 191
H.2 Comparison of the architectural vulnerability factors (AVFs) and relative mean useful instruction to failure (rMuITF) of an unmitigated LEON3 against the mitigated LEON3 processor. ............................................ 192
H.3 Fault-injection results for protected and unprotected processors. ............ 193
H.4 Average error between modeled and measured MTTF. ........................ 194

I.1 Average detection latencies for each detection technique when upsets are detected and successfully repaired (DRU) and when upsets result in a dloop. ............ 199

J.1 MISSE-8 SEUXSE II test results .................................................. 203
## LIST OF FIGURES

2.1 SRAM bits in the configuration memory control logic and routing of a design in an SRAM-based FPGA. ................................................................. 11
2.2 Single event upsets (SEUs) can cause firm errors. Firm errors are repaired only when the FPGA is reconfigured. .................................................. 12
2.3 Triple modular redundancy (TMR) can be implemented with either a single, or triplicated voters. ................................................................. 14
2.4 DWC detects upsets, and checkpoint recovers. .................................................. 14
2.5 A BRAM protected by TMR with scrubbing. .................................................. 16
2.6 LEON3 softcore processor pipeline architecture. .......................................... 18
2.7 The current window pointer (CWP) in a windowed register file decrements when traps or function calls are made, and increments when returning from a trap or function call. .................................................. 19
2.8 The LEON3 core units used in this study. .................................................. 19

3.1 An upset occurs before a checkpoint, but is detected after the checkpoint, possibly causing a dloop. .................................................. 28
3.2 Dloops can be practically eliminated when two sets of alternating checkpoints are used. .................................................. 30
3.3 LEON3 processor with DWC and checkpointing. ........................................ 32
3.4 LEON3 processor with TMR and checkpointing. ........................................ 34

4.1 Complement duplicate with compare (CDWC) is used only on the register file as an alternative to instruction duplication. ........................................ 41
4.2 Control-flow monitoring adds signature checking to each basic block. .............. 43
4.3 Example program demonstrating the CFCSS algorithm. .................................. 46
4.4 Modified ICFCSS added to a code block in the Fhourcorners testbench program. .................................................. 50
4.5 Functional unit consistency check with an SEU in the operand register. .............. 53
4.6 Traditional consistency check performed to detect upsets in memory elements. .... 54

5.1 Traditional cycle-by-cycle hardware fault-injection methodology. ................. 73
5.2 A fault-injection model for softcore processors needs to allow the DUT to temporarily misbehave. .................................................. 74
5.3 SEAKR’s XRTC board is used to perform hardware fault-injection. .................. 75
5.4 The hardware fault-injection procedure followed for each of the 21 million bits in the DUT configuration memory. .................................................. 77
5.5 The hardware fault-injection procedure for each of the 21 million configuration bits in the DUT configuration memory. ........................................ 77

6.1 Processor design-space created by a MTTF-area trade-off. ........................ 97
6.2 Processor design-space created by a MuITF-area trade-off. ........................ 98

A.1 A 2x2 programmable interconnect point (PIP). ........................................ 116
A.2 An SEU in a PIP can cause an open or a short. ........................................ 117
B.1 Upsetting a BRAM write enable can cause entire contents to be overwritten - resulting in a critical failure. ................................. 119
B.2 Upsetting a LUTRAM write enable can cause memory contents to be overwritten - resulting in a critical failure. ......................... 122
B.3 Triple modular redundancy (TMR) can be implemented with either a single, or triplicated voters. ..................................................... 124
B.4 A memory protected by duplication and with an error detection code (EDC). ................................................................. 124
B.5 A memory protected by an error correction code (ECC). ............................................................. 127
B.6 A BRAM protected by TMR with scrubbing. ............................................................... 129
B.7 A LUTRAM protected by duplication with EDC/ECC and scrubbing. ................................................. 131
B.8 An SRL protected by duplication with EDC and scrubbing. .............................................................. 132
B.9 Fault-injection is performed using SEAKR’s XRTC board. .............................................................. 133

C.1 Program execution timeline with \( n \) equally spaced checkpoint intervals. ................................. 144
C.2 Timeline showing the terminology for the first-order probability model to select an optimal checkpointing interval for a processor ........................................ 145
C.3 Time lost due to checkpointing and reruns as a function of the checkpoint interval length \( T \) in seconds. ................................................ 147
C.4 Timeline showing the terminology for Nakagawa’s probability model to find the optimal number of CSCP checkpoints. ........................................ 150
C.5 Total execution time of a program as a function of the checkpoint interval length \( T \) in seconds. ................................................ 152
C.6 Timeline showing the terminology for Nakagawa’s probability model to find the optimal number of CCP checkpoints. ........................................ 155

D.1 Program execution timeline with \( n \) equally spaced checkpoint intervals. ................................. 159
D.2 Timeline showing the terminology to find the optimal number of SCP checkpoints for a duplicated softcore processor. ........................................ 161
D.3 Total execution time of a program on a duplicated softcore processor as a function of the checkpoint interval length \( T \) (in seconds). ................................................ 164
D.4 Timeline showing the terminology used to find the optimal number of CSCP checkpoints for a softcore processor protected with software fault-tolerance. ........................................ 168
D.5 Total execution time of a program on a softcore processor as a function of the checkpoint interval length \( T \) (in seconds). ................................................ 171
D.6 Timeline showing the terminology used to find the optimal number of consistency checks for a softcore processor protected with software fault-tolerance. ........................................ 171
D.7 Execution time of a CSCP checkpoint interval on a softcore processor as a function of the consistency check interval length \( T_{cc} \) (in seconds). ................................................ 178

E.1 Decide the number of consistency checks per checkpoint interval based on empirical fault-injection results (MTTF, MuITF, and RAP) of a LEON3 softcore processor protected with consistency checks and rollback checkpointing. ........................................ 181

F.1 SPARCv8 \texttt{add} instruction. ................................................ 184
G.1 FPGA Editor view of an unconstrained and constrained software fault-tolerant
LEON3 softcore processor. ................................................................. 188

H.1 The accelerator test setup. .............................................................. 193

I.1 Relative number of unACE, SDC, and DUE bits for an unmitigated LEON3 pro-
cessor when each of the testbench programs are run. .......................... 196
I.2 Fault mode ratios of the ACE bits in the unmitigated LEON3 softcore processor.. 196
I.3 Hardware fault-injection results for the LEON3 processor protected with DWC
and rollback checkpointing. ............................................................... 197
I.4 Hardware fault-injection results for the LEON3 processor protected with TMR and
roll-forward checkpointing. ............................................................... 197
I.5 Locations of the SDC and DUE configuration upsets in the traditionally mitigated
LEON3 processors. ....................................................................... 198
I.6 Hardware fault-injection results for the LEON3 processor protected with different
combinations of software detection techniques and rollback checkpointing. .... 200

J.1 Mission Specialist Drew Feustel deploys the MISSE-8 on 20 May 2011. (Photo
credit: NASA) ............................................................................... 202
J.2 Memory experiment performed on the MISSE-8. ................................. 203
J.3 Picoblaze processor experiment performed on the MISSE-8. .................. 203
CHAPTER 1. INTRODUCTION

1.1 Motivation

Microprocessors have been an important part of space-based applications since the first processors were used for spaceflight [1, 2]. In 1977, the RCA COSMAC 1802 was the first microprocessor selected for spaceflight when it was chosen for the Galileo probe mission to Jupiter [1, 3]. Since then, processors have been an integral part of most space missions. Currently, BAE System’s RAD750 processor, which is used on the Mars Reconnaissance Orbiter and the Jupiter Juno Launch, is the state-of-the-art ASIC-based processor for space applications [4, 5].

As processors began to be used in spaceflight, it was observed that high-energy particles could cause faults in the electronics. Upsets caused by high-energy particles are referred to as single-event upsets (SEUs). When NASA’s Voyager spacecraft was launched in 1977, upsets to electronics in space caused by SEUs had not yet been observed, thus no measures were taken to protect the hardware from the effects of SEUs [1]. But by 1980-1981, when Voyager had reached Jupiter, the problems caused by high energy sulfur ions emanating from Jupiter’s volcanic moon, Io, quickly became apparent. From these observations, Galileo Project Control Engineer Edward H. Kopf commented with regards to the Galileo mission [1], “It is not worth flying the mission if you cannot get rid of the SEU problem.” Since then, protective measures have always been taken to protect electronics used in space [6, 7].

Microprocessors used in space-based applications must, therefore, be protected from the effects of high-energy particles. To protect the electronics of the Galileo mission, Sandia National Laboratories was contracted to custom make a radiation-hardened version of the attitude control computers [1]. These radiation-hardened computers were hardened by process. Processors that are radiation-hardened (rad-hard) by process are built with special design libraries and fabrication processes that are more resilient to high-energy radiation [8–10]. These larger transistors resist high-energy particles by requiring more energy to switch. Although tolerant to radiation, rad-
hard processors are larger, slower, consume more power, and can cost hundreds of thousands of dollars [4, 11–13] when compared to a commercial processor. Additionally, rad-hard processors used in space are often one to two decades old [4, 14–16].

A newer form of radiation-hardening provides protection through radiation hardening by design (RHBD) [16, 17]. Instead of using larger, more SEU-resistant transistors, a rad-hard by design processor is protected with transistor-level redundancy. This redundancy increases the area cost of a rad-hard processor. Also, RHBD processors use older process technologies. Thus, like rad-hard by process processors, they are larger, slower, and consume more energy than commercial processors, and their implementation costs make them extremely expensive.

1.2 Processor Reliability Design-Space

Although radiation hardening has become the most popular way to protect processors in space, it is also among the most costly. Putting aside the high cost of purchasing a rad-hard processor, this work considers cost in terms of area and performance. The processor design-space is considered along three axes: reliability, area, and performance. This work shows that there are less costly ways to protect a processor that can provide reliability which is comparable to that of rad-hard processors.

As an alternative to an older expensive rad-hard processor, processors can be implemented in the logic of a field-programmable gate array (FPGA). A processor implemented in an FPGA is called a softcore processor. A softcore processor is a microprocessor that is completely described in a hardware description language (HDL), such as VHDL or Verilog. Softcore processors have the advantage of being very flexible and customizable. Architectural features such as floating-point units, hardware multipliers, scratchpad memories, or cache size can be changed at any time on an FPGA through reconfiguration. Also, since softcore processors are described in high-level languages, they are not created to target a specific device or FPGA technology. This means that softcore processors can be implemented on the latest FPGA technologies.

A softcore processor implemented in a commercial SRAM-based FPGA provides additional benefits for processors in space. The reconfigurability that SRAM-based FPGAs provide makes them extremely flexible. They can be reprogrammed while in orbit to adapt to changing mission needs or to correct design errors. Since softcore processors are not technology-specific,
they can be implemented in the latest FPGA technologies; thus, they are not based on older, slower, larger technologies like rad-hard processors are. Also, commercial SRAM-based FPGAs can be purchased for a fraction of the cost of a rad-hard processor [13,18]. Therefore, softcore processors are more flexible, less costly, faster, and smaller than rad-hard processors. If the reliability of a softcore processor can approach that of a rad-hard processor it becomes a very attractive alternative to rad-hard processors for space-based applications.

Unfortunately, the flexibility and reprogramability that FPGAs provide for softcore processors used in space-based applications comes at a price – SRAM-based FPGAs are inherently more sensitive to the effects of faults caused by high-energy particles than ASICs are. Although all electronics are sensitive to these single-event upsets (SEUs), SRAM-based FPGAs are particularly sensitive, since SEUs can occur not only in FPGA user memory bits but also in the FPGA configuration bits that define the logic and routing of the softcore processor.

A significant amount of research and testing has provided effective ways to protect the logic and routing of SRAM-based FPGA designs. One technique that is quickly becoming very attractive is a radiation-hardened SRAM-based FPGA [19–21]. The configuration memory cells in this kind of FPGA are hardened by design, providing the protection of radiation hardening to the logic and routing of a softcore processor. This option provides a processor for space-based applications whose reliability approaches that of an ASIC-based rad-hard processor, but also has the flexibility that an SRAM-based FPGA provides. However, like rad-hard ASIC-based processors, this option is very expensive, and is not implemented on the latest technologies. Thus, it has a higher area, performance, and monetary cost than a softcore processor implemented on the latest commercial SRAM-based FPGA.

The most popular way to protect designs implemented on a commercial SRAM-based FPGA is with triple modular redundancy (TMR) [22–28]. TMR works by triplicating a design and voting on the outputs of the three modules. TMR voters detect and mask a faulty module. Although this technique can be implemented on the latest commercial FPGA technologies, and its reliability can approach that of processor protected with radiation hardening, it is expensive in terms of area and power. Thus, this protection technique performs well on two of the three processor design-space axes: reliability and performance cost, but it has a very large area cost.
A less costly protection strategy for softcore processors implemented on commercial SRAM-based processors uses terrestrial, ASIC-based processor protection techniques. Software fault-tolerance is a popular low-cost protection technique for terrestrial, ASIC-based processors [29–34]. Software techniques are designed to protect user memory elements from SEUs through the use of time instead of area. These techniques work well for protecting the user memory elements of ASIC-based processors, but they must do much more in order to protect a softcore processor. They must also protect against faults in the logic and routing of the processor caused by upsets in the configuration memory. Software fault-tolerance greatly reduces the area costs, but causes an increase the performance costs. If software fault-tolerance can protect the logic and routing of a softcore processor implemented in an FPGA, it will provide an ideal option for processors used in spaceflight.

1.3 Dissertation Contributions

This work shows that a software fault-tolerant softcore processor can have a reliability that is comparable to a popular rad-hard processor, but with more flexibility, and at a much lower cost. In accomplishing this goal, this work provides four main contributions:

- A softcore processor protected with low-cost software fault-tolerance techniques.
- Updated metrics for evaluating the reliability of softcore processors.
- An updated hardware fault-injection methodology specifically created for softcore processors.
- Probability models for finding the optimal checkpoint interval and consistency check interval for a softcore processor protected with these software techniques.

The primary contribution of this work is a low-cost fault-tolerant LEON3 [35] softcore processor designed for commercial SRAM-based FPGAs. The low-cost protection is provided by adapting software fault-tolerance techniques in a unique way to softcore processors. The software techniques applied to the LEON3 include a modified version of software implemented fault tolerance (SWIFT) [29, 36], consistency checks [31, 34], software indicators [37, 38], and checkpointing [32, 33]. These software techniques have not previously been used in this way to protect
the logic and routing of a softcore processor implemented on a commercial SRAM-based FPGA. A combination of software and hardware reliability techniques can protect over 99% of the configuration bits of a LEON3 processor for less than a third the cost of TMR, and for only a 1.7× performance cost.

The second contribution of this work is the creation of novel metrics to measure the reliability of softcore processors. The traditional way of measuring reliability for FPGA designs is to measure the number of sensitive bits in the configuration bitstream [39,40], or to measure the mean-time to failure (MTTF) of the design [28,41]. The traditional way of measuring reliability in ASIC-based processors is through MTTF or mean instructions to failure (MITF) [42]. This work applies these metrics to softcore processors and introduces two novel metrics: mean useful instruction to failure (MuITF) and reliability-area-performance (RAP). These metrics are used to demonstrate the benefits of the previously described softcore processor.

The third contribution of this work is the introduction of an updated hardware fault-injection methodology for softcore processors. Traditional fault-injection compares the design under test (DUT) with a golden design on a cycle-by-cycle basis. This model is insufficient for softcore processors since a fault could merely delay a correct processor output. Or, an upset may cause the softcore processor to temporarily misbehave before recovering. This work introduces a fault-injection model that is applicable to softcore processors. This fault-injection model is used to gather the raw data required for the unique reliability metrics previously mentioned.

The fourth contribution of this work is the creation of probability models to find the optimal checkpoint interval of a softcore processor, and to find how often to perform consistency checks on a softcore processor. Since checkpointing and consistency checks are among the software techniques used in this study, it is important to know how often they should be executed. The probability models introduced in this work are based on existing models for ASIC-based processors, but are adapted for softcore processors. This probability model is important for anyone applying checkpointing to a softcore processor.

In addition to applying novel metrics and models to a LEON3 softcore processor protected with low-cost software techniques, these novel metrics and models are also applied to a LEON3 processor protected with traditional hardware mitigation techniques. A common traditional ASIC-based processor protection technique combines duplication with compare (DWC) and rollback
The most popular commercial SRAM-based FPGA protection technique is TMR \cite{22, 25, 27, 28, 32, 45}. Part of this work implements a LEON3 processor protected with DWC and rollback checkpointing, and a LEON3 processor protected with TMR and roll-forward checkpointing \cite{44, 46}. The implementations of these traditional protection techniques act as baseline designs for the software fault-tolerant LEON3 to be compared against. They also provide a more thorough exploration of the processor design-space.

The metrics and models introduced in this dissertation show that a softcore processor protected with software fault-tolerance has a reliability that is comparable to a popular rad-hard processor. The software fault-tolerance protects over 99\% of the softcore processor at less than a third the cost of TMR, for only a $1.7\times$ performance cost. This dissertation also explores the processor design-space with different combinations of hardware and software mitigation techniques. Combinations that provide greater reliability come at the cost of area and/or performance.

1.4 Dissertation Organization

The presentation of this dissertation is organized into seven chapters:

- **Chapter 2** reviews how single-event upsets (SEUs) can affect the configuration memory and user memory of a softcore processor implemented in a commercial SRAM-based FPGA. Both types of FPGA memories have unique considerations that softcore processors must take into account, which make them unique compared to ASIC-based processors. Aeroflex Gaisler’s LEON3 softcore processor is also presented as the softcore processor used in this work.

- **Chapter 3** begins by discussing the issues surrounding softcore processor support of checkpointing. Since checkpointing is the only recovery technique used throughout this work, it is important first to establish how checkpointing is applied to softcore processors. Next, the traditional softcore processor mitigation techniques used in this study are introduced. Finally, the application and costs of TMR and DWC with checkpointing are presented.

- **Chapter 4** presents the software fault-tolerance techniques used to create a low-cost reliable softcore processor. This chapter discusses how each technique is adapted for use on a soft-
core processor, and presents each technique’s costs along the area and performance cost axes of the processor design-space.

• **Chapter 5** introduces novel metrics and models for evaluating the reliability of a soft-core processor. First, traditional reliability metrics are adapted to softcore processors: architectural vulnerability factor (AVF) and mean time to failure (MTTF). Next, two novel metrics are introduced: mean useful instructions to failure (MuITF), and reliability-area-performance (RAP). This chapter also introduces a hardware fault-injection model for softcore processors. The accuracy of this model is demonstrated with radiation testing (Appendix H). Finally, this chapter introduces the suite of testbench programs used to establish the reliability of the LEON3 softcore processor.

• **Chapter 6** applies the hardware fault-injection model and reliability metrics discussed in Chapter 5 to the unmitigated and traditionally protected LEON3 processor discussed in Chapter 3. Next, these metrics and models are applied to the software fault-tolerant LEON3 processor designs discussed in Chapter 4. These reliability results complete the exploration of the softcore processor design-space. Finally, the failure rates of popular rad-hard processors are compared with the failure rates of the softcore LEON3 processor designs presented throughout this work.

• **Chapter 7** concludes this work by summarizing its results. Next, future work is proposed which would build upon this work to provide novel protection techniques for softcore processors implemented on commercial SRAM-based FPGAs. Finally, the contributions of this work are summarized.
CHAPTER 2. PROTECTING SOFTCORE PROCESSORS IN SPACE

Softcore processors are an attractive option for space-based applications because of their flexibility, low application development costs, and reconfigurability when implemented on an SRAM-based FPGA. The FPGA holding the softcore processor can be reprogrammed while in orbit to adapt to changing mission needs or correct design errors. For example, the Mars rovers use FPGAs for their motor control and landing pyrotechnics [47]. Also, the Los Alamos National Laboratory CFESat satellite [15] and the Australian FedSat satellite [48] both use FPGAs in their high performance computing payloads. Although these examples do not include softcore processors as part of their computing payload, they demonstrate that FPGA designs similar to softcore processors can be successfully used in spaceflight.

Despite the advantages that softcore processors can bring to space-based applications, they are rarely used because they are much more sensitive to high-energy particles than ASIC-based processors. Upsets in the FPGA holding the softcore processor are caused by high-energy particles called single event effects (SEEs). They can occur not only in FPGA user memory bits but also in the FPGA configuration bits. Thus, softcore processors have unique fault modes compared to ASIC-based processors.

Before introducing softcore processor fault-tolerance, it is important first to discuss what software fault-tolerance must protect in a softcore processor. This chapter motivates the need for protection with a discussion on fault modes in SRAM-based FPGAs. It begins by discussing the effects of radiation on electronics in general, and then identifies the unique fault modes of softcore processors through a discussion on how SEEs affect an FPGA’s configuration and user memories. Finally, the softcore processor used throughout this work is introduced.
2.1 Effects of Radiation on Electronics

Objects in space are constantly bombarded by high-energy particles from cosmic rays and particles trapped in the Earth’s magnetic field [7,49,50]. Normally, the effect that this radiation has on electronics used in space can either temporarily change the behavior of a circuit (a soft error), or it can damage the circuit (a hard error). However, the effects of radiation on FPGAs designs can cause an error that has characteristics of both a hard and a soft error (a firm error).

A soft error results from a transient upset caused by radiation. Such an upset is called a single-event effect (SEE) and can be further classified as a single-event upset (SEU), single-event transient (SET), or single-event functional interrupt (SEFI) [51,52]. SEEs are one-time occurrences whose effects are temporary. Although soft errors do not permanently damage a circuit, they can cause faulty circuit behavior. In the worst case scenario, a device will have to be reset or reconfigured to overcome the effects of a soft error.

A hard error is a permanent effect that cannot be repaired. These errors are caused by electrical stress failures, intrinsic failure mechanisms, or external failure mechanisms. Electrical stress failures include electrical overstress, electrostatic discharge, latchup, gate oxide breakdown, ionic contamination, charge effects (slow trapping, hot carriers, plasma discharge), and electromigration [53, 54]. Intrinsic failures refer to the reliability of the semiconductor device and materials – problems related to the manufacturing process. Extrinsic failures refer to the reliability of the interconnects (metal and contacts), passivation, and packaging – problems related to the fabrication process. When a hard fault occurs, the circuit or unit containing the fault should no longer be used.

Transient upsets in SRAM-based FPGA configuration memories cause soft errors that temporarily act as hard errors. In this study, the name given to these kinds of error is firm errors. The configuration memory of an SRAM-based FPGA controls the routing and logic of a design on the device. Thus, upsetting an SRAM cell in the configuration memory can change the behavior of the design until the SRAM cell is repaired. Therefore, just like a hard error, a transient upset can cause lasting effects in the routing and logic of an FPGA design. However, unlike a hard error, these upsets can be repaired.

Single-event transients (SETS) are of particular concern for ASIC-based processors in a radiation environment. A single-event transient occurs when an upset causes a temporary voltage/current spike. If the pulse width of this spike is sufficiently large, and occurs at the right time,
it can be latched in a flip-flop and propagate through the circuit. The probability \( P_{\text{SET}} \) that a pulse of width \( t_{\text{SET}} \) is latched in given flip-flop is proportional to the ratio of \( t_{\text{SET}} \) to the clock period \( t_{\text{CLK}} \) [55]:

\[
P_{\text{SET}} \approx \frac{t_{\text{SET}}}{t_{\text{CLK}}}.
\]

Equation 2.1 shows that as the clock period decreases, the chance for a transient to be latched increases. Additionally, as transistor sizes decrease, device voltage levels also decrease. This means that less energy is required to cause a voltage spike. For these reasons, SETs are of increasing concern to ASIC-based processors, not only in space, but also on Earth [56, 57]. To protect ASIC-based processors, fault-mitigation techniques are used to detect and recover from transients [36].

Despite the rising concern of SETs in ASIC-based processors, firm errors are of much greater concern for SRAM-based FPGAs than single event transients. SETs are less significant for three reasons. First, the probability that a voltage/current spike will be sufficiently large to cause an SET is directly proportional to the capacitive loading of the signal path [58]. The SRAM-based FPGA structure, by its nature, is highly resistant to SETs due to its large capacitive loading of signal paths [59]. This loading is many times larger than that of an ASIC.

Secondly, there are hundreds of times more configuration memory bits than flip-flops on an FPGA. Thus, when a transient does occur, it is much more likely to occur in a configuration memory element. Transients at a configuration memory elements never become SETs since they are not enabled during design execution.

Finally, the clock speeds of FPGAs are typically lower than ASICs. Thus, in the rare event that capacitive loading does not prevent an SET from occurring, the likelihood for a temporary pulse to be latched is lower. The pulse widths of transients in FPGAs vary from a few picoseconds to 2ns [58, 60].

The focus of this work is on the detection and correction of the effects of SEUs. Since the detection of SETs is generally considered to be insignificant for SRAM-based FPGA fault-tolerance [59], transients are ignored. Although some of the fault-tolerance techniques used in this work will detect both SEUs and SETs, their intended use is for SEU detection and recovery.
The intended use of most software fault-tolerance techniques are traditionally meant to detect transients [36], but in this work, their intended use is changed to detect SEUs. The consequences of this change are discussed in Chapter 4.

2.2 Single Event Effects in FPGA Configuration Memory

An FPGA is an integrated circuit whose logic and routing matrices can be easily programmed \textit{in the field}. The circuitry of an FPGA includes a large array of reprogrammable logic resources and a rich interconnect system (Figure 2.1). The logic and interconnect of an FPGA can be programmed to create large, complex digital circuits such as a processor. Since designs are implemented on a programmable hardware fabric, FPGAs are extremely flexible and provide a low application development cost for designs. This flexibility and reprogrammability make FPGAs ideal for space-based applications [15, 47, 48]. They can be reprogrammed while in orbit to adapt to changing mission needs or to correct design errors.

Figure 2.1: SRAM bits in the configuration memory control logic and routing of a design in an SRAM-based FPGA.
The ability to reconfigure an SRAM-based FPGA is provided by the presence of internal SRAM cells, called the configuration memory. Figure 2.1 shows that routing matrices are controlled with SRAM cells. These SRAM cells are part of the configuration memory. Changing the routes through a routing matrix is accomplished by changing the values of these configuration memory bits (Appendix A). Figure 2.1 also shows that bits in the configuration memory control logic elements such as look-up tables (LUTs), multiplexers, flip-flop attributes, as well as other elements not shown in the figure. A large configuration memory is required to control all of the routing and logic for the entire FPGA. For the Xilinx FPGA used in this study (Xilinx Virtex4 FX60), nearly 21 million bits are included in the configuration memory.

### 2.2.1 Failure Modes in FPGA Designs

FPGAs have unique failure modes compared to application-specific integrated circuits (ASICs). An SEU in an FPGA’s configuration memory can result in a firm error occurring in either the logic or interconnect of a design. An upset is classified as a logic upset if it occurs in a part of the configuration memory that controls any logic element or any logic element attribute. An example of an SEU causing a logic failure is shown in Figure 2.2(b). In the figure, an SEU changes a bit in a look-up table (LUT) which changes the logic function that the circuit is supposed to perform.

![Figure 2.2: Single event upsets (SEUs) can cause firm errors. Firm errors are repaired only when the FPGA is reconfigured.](image)

(a) A commercial SRAM-based FPGA design operating correctly.  
(b) The logic SEU shown causes a bit flip in a part of the configuration memory that controls LUT bits.  
(c) The two SEUs shown cause bit flips in the configuration memory that control routing. The open SEU causes an open, and the short SEU causes two routes to drive a single input.
Another unique FPGA failure mode results from upsets in the parts of the configuration memory that control routing. Firm errors in routing can be classified as either an open or a short. Figure 2.2(c) shows an example of each of these types of configuration memory upsets. The figure shows that a firm error, classified as an open, causes the routing of the top input to the LUT to be disconnected, resulting in a floating input. A firm upset that is classified as a short causes multiple routes to drive a single input. Figure 2.2(c) shows a firm error in the routing that causes a second signal to drive the bottom input of the LUT. Appendix A discusses the details of what can happen when SEUs strike a routing matrix.

2.2.2 Traditional FPGA Design Protection

In order for softcore processors to be an attractive option for space-based applications, their FPGA configuration and user memories must be protected from SEUs. The term protect in this work describes the detection, correction, and recovery from upsets that occur within the configuration and user memories. This definition of protection is different than the definition assumed by rad-hard processors, which refers to the prevention of SEUs from occurring.

A significant amount of research has identified hardware techniques to protect FPGA configuration memory in the presence of SEUs. Triple modular redundancy (TMR) and configuration memory scrubbing [22–28] are among the most popular ways to protect FPGA designs. TMR works by triplicating a design and voting on the outputs of the three modules. A TMR voter detects and masks a faulty module (Figure B.3(a)). If a fault occurs in one of the triplicated versions of the design, its faulty output will be out-voted by the other two correct designs. However, since all logic and routing in an SRAM-based FPGA are susceptible to SEUs, the voter itself and all routing must also be triplicated in order to prevent errors due to a fault in the voter (Figure B.3(b)). Although voter triplication with full input and output triplication (Figure B.3(b)) provides better reliability than non-triplicated voters and inputs/outputs (Figure B.3(a)), it is not always possible to provide this triplication. This work considers both cases: TMR with a single voter as well as TMR with triplicated voters.

Another spatial redundancy-based protection technique used to protect the logic and routing of FPGA designs couples duplication with compare (DWC) with novel recovery techniques [61–63]. DWC detects when an upset in one of the duplicated modules has occurred, but by itself,
Figure 2.3: Triple modular redundancy (TMR) can be implemented with either a single, or triplicated voters.

cannot know which of the two modules are in error. Also, DWC cannot by itself correct the effects of an SEU. Recovery techniques must be coupled with DWC in order to know which module is in error, and/or to recover from the effects of upsets [25, 61]. In this study, when DWC detects an upset, checkpointing is used to recover (Figure 2.4).

Figure 2.4: DWC detects upsets, and checkpoint recovers.

FPGA configuration scrubbing (also called bitstream scrubbing) is an important counterpart to TMR or DWC protection. Bitstream scrubbing refers to the correction of upsets within the configuration memory. While a design runs on an SRAM-based FPGA, a scrubbing unit continually checks for upsets in the configuration memory. This is done by reading the contents of configuration memory and comparing them against a golden copy of the memory bitstream. Alternatively, CRC values of portions of the bitstream are calculated and compared to the known CRC
values. It is common for configuration scrubbers to be employed when SRAM-based FPGAs are used in radiation-filled environments such as space.

### 2.3 Single Event Effects in FPGA User Memory

The configuration memory is not the only FPGA memory that is sensitive to upsets. User memory is also sensitive to upsets. User memory refers to the SRAM cells used to create registers and memories within a design. For example, the user memories in a softcore processor include the SRAM cells that make up the main memory, register file, caches, and pipeline registers. FPGA user memory includes the SRAM cells of any flip-flop or block RAMs (BRAMs). Table 2.1 shows that for the FPGA device used in this study (Xilinx Virtex4 FX60), there are much fewer user memory bits than configuration memory bits. This section discusses the unique considerations for protecting user memories in the presence of upsets.

<table>
<thead>
<tr>
<th>Xilinx Virtex4 FX60 Memory Bits</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Configuration Bits</td>
<td>20,960,512</td>
</tr>
<tr>
<td>User BlockRAM Bits</td>
<td>4,870,144</td>
</tr>
<tr>
<td>User Flip-Flops</td>
<td>50,560</td>
</tr>
</tbody>
</table>

For softcore processors to be useful in spaceflight, upsets in the user memories must be detected and corrected. The traditional way of protecting user memories in ASIC-based processors is to use some sort of error correction code (ECC) [34, 64]. Appendix B considers how effective these techniques are for protecting user memories in SRAM-based FPGAs (validated in Appendix J). Since the ECC logic and routing is itself sensitive to upsets, measures must be taken to prevent faults caused by upsets in either user memory content, or upsets in the logic and routing protecting the memory.

To protect a user memory fully, memory scrubbing is required [25, 65]. Memory scrubbing refers to the deliberate repairing of memory words by a memory scrubbing module. Memory scrubbing is different than bitstream scrubbing. Bitstream scrubbing is a great tool to help protect against upsets in FPGA logic and routing, but it cannot protect FPGA memories. Since the contents
of memories change, there cannot be an a-priori known CRC value or golden content value that can be used for comparison. Thus, memory scrubbers such as the one shown in Figure 2.5 must be implemented independently of bitstream scrubbers.

Appendix B shows that full TMR and memory scrubbing can eliminate all errors due to SEUs in a BRAM or in the logic and routing leading to a BRAM. The study done in Appendix B shows that in general, at least one redundant memory is required for memory scrubbing to work effectively. In this dissertation, checkpointed memory contents act as a redundant memory, which are used for memory scrubbing purposes (Chapter 3).

2.4 LEON3 Softcore Processor

When used in a radiation environment such as space, a softcore processor is much more sensitive to upsets than an ASIC-based processor. Not only can upsets occur within the user memory elements of the softcore processor, but upsets can cause firm errors in its logic and routing. Upsets are much more likely to happen in the logic and routing than in the user memory (Table 2.1). This section introduces the softcore processor used throughout this work and shows how
vulnerable each of its architectural components are to upsets by identifying how many configuration bits are associated with each component. This softcore processor is used to implement the fault-tolerance techniques described in the following chapters.

### 2.4.1 LEON3 Processor Architecture

The softcore processor used in this study is Aeroflex Gaisler’s 32-bit LEON3 [66] processor – the processor used by the European space agency [35]. This processor is chosen because of its popularity in the space community. The LEON3 processor is open-source and comparable in performance and efficiency to other well-known softcore processors. In a study that compared three popular softcore processors, the LEON was shown to have the best performance and to be the most configurable [67]. Another study suggests that the LEON3 is the most efficient when compared with the Xilinx MicroBlaze softcore processor and hardcore on-chip PowerPC processor [68].

The LEON3 conforms to the SPARC V8 instruction set architecture (ISA). It contains a 7-stage integer pipeline (Figure 2.6) with separate instruction and data write-through caches, a hardware multiplier and divider, floating-point unit support, co-processor unit support, and multiprocessor capability. Both the data and instruction caches can be configured with up to 4 sets of 1 to 256 kbyte/set, and 16 or 32 bytes per line.

The register file uses windowed registers with up to 32 windows. Each register window consists of 32 registers: 8 global registers, 8 input registers, 8 output registers, and 8 local registers. The 8 global registers are shared by all register windows. Figure 2.7 shows how register windows are used when traps are used or function calls are made. When a function is called, the register window pointer decrements, pointing to a new window, and increments when function returns. The output registers of the caller become the input registers of the callee. Thus, parameters are passed from the caller to the callee by placing them in the caller output registers. These parameters become available to the function in the new window’s input registers. Register windows eliminate the uncertainty of whether the caller, or the callee is responsible for saving the state of the current scope.

Only the microarchitecture of the LEON3 processor is included in this study (Figure 2.8). The core units include the 7-stage integer pipeline, a 10 window register file, the hardware multiplier and divider, 1 Kbyte direct-mapped instruction and data caches, interrupt controller, and
Figure 2.6: LEON3 softcore processor pipeline architecture.
### Input Registers

%\texttt{i}0 \rightarrow% \%i7

### Local Registers

%\texttt{l}0 \rightarrow% \%l7

### Output Registers

%\texttt{o}0 \rightarrow% \%o7

### Input Registers

%\texttt{i}0 \rightarrow% \%i7

### Local Registers

%\texttt{l}0 \rightarrow% \%l7

### Output Registers

%\texttt{o}0 \rightarrow% \%o7

### Global Registers

%\texttt{g}0 \rightarrow% \%g7

#### Current window pointer (CWP)

<table>
<thead>
<tr>
<th>CWP + 1</th>
<th>CWP - 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input Registers (%i0 \rightarrow% %i7)</td>
<td>Input Registers (%i0 \rightarrow% %i7)</td>
</tr>
<tr>
<td>Local Registers (%i0 \rightarrow% %i7)</td>
<td>Local Registers (%i0 \rightarrow% %i7)</td>
</tr>
<tr>
<td>Output Registers (%o0 \rightarrow% %o7)</td>
<td>Output Registers (%o0 \rightarrow% %o7)</td>
</tr>
</tbody>
</table>

**Figure 2.7:** The current window pointer (CWP) in a windowed register file decrements when traps or function calls are made, and increments when returning from a trap or function call.

**Figure 2.8:** The LEON3 core units used in this study.

---

on-chip main memory implemented with 8 BRAMs. Since the focus of this work is on the reliability of the processor itself, peripheral units are removed. The units removed include the AMBA bus, AMBA bus controllers and interfaces, the default memory controller, the default interrupt controller, the debug support unit (DSU) and debug port, the trace buffer, and the translation lookaside buffer (TLB). Optional units that are not used include scratchpad memories (unnecessary when main memory is on-chip), co-processors, and a floating-point unit (not open source).
2.4.2 LEON3 Area Costs

Table 2.2 shows the resource usage of each of the LEON3 components in terms of slices, BRAMs, DSPs, and configuration memory bits (CFGbits). The configuration bits consumed by each processor component are obtained using an XDL-based tool developed at Brigham Young University called RapidSmith [69, 70]. The table shows that configuration memory bits can quantify components such as processor clocking, that are not quantified by the other metrics. Although the clock consumes no slices, BRAMs, or DSPs, the routing and buffers it consumes can be quantified in terms of configuration memory.

Table 2.2: Area costs for the LEON3 processor and each of its components.

<table>
<thead>
<tr>
<th>Component</th>
<th>Slices</th>
<th>BRAMs</th>
<th>DSPs</th>
<th>Config Bits (CFGbits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>8707</td>
</tr>
<tr>
<td>Reset</td>
<td>15</td>
<td>0</td>
<td>0</td>
<td>20,927</td>
</tr>
<tr>
<td>Top Outputs</td>
<td>55</td>
<td>0</td>
<td>0</td>
<td>21,692</td>
</tr>
<tr>
<td>IRQ Controller</td>
<td>12</td>
<td>0</td>
<td>0</td>
<td>1624</td>
</tr>
<tr>
<td>Mult/Div</td>
<td>206</td>
<td>4</td>
<td>0</td>
<td>45,816</td>
</tr>
<tr>
<td>Icache</td>
<td>180</td>
<td>1</td>
<td>0</td>
<td>32,004</td>
</tr>
<tr>
<td>Dcache</td>
<td>502</td>
<td>1</td>
<td>0</td>
<td>82,578</td>
</tr>
<tr>
<td>Main Memory</td>
<td>180</td>
<td>8</td>
<td>0</td>
<td>54,588</td>
</tr>
<tr>
<td>Register File</td>
<td>510</td>
<td>0</td>
<td>0</td>
<td>107,705</td>
</tr>
<tr>
<td>ALU</td>
<td>1398</td>
<td>0</td>
<td>0</td>
<td>293,984</td>
</tr>
<tr>
<td>Pipeline</td>
<td>3058</td>
<td>12</td>
<td>4</td>
<td>677,065</td>
</tr>
</tbody>
</table>

2.4.3 Modifications to the LEON3

One of the major advantages of using a softcore processor is that they are flexible enough to support changes. Minor modifications are made to the LEON3 processor to support fault-tolerance. First, the complete LEON3 processor includes a main memory unit that connects off-chip memory to an AMBA bus controller. Since the AMBA bus is not part of the core LEON3 components used
in this work, and since the main memory is built of on-chip BRAMs, a very simple main memory controller is created to replace the existing LEON3 memory controller. This simple memory controller acts as an arbitrator between main memory and the instruction and data caches.

Next, the interrupt controller that is included with the LEON3 processor is intended to be used with multiple LEON3 processor cores. Instead of making minor modifications to this controller, a smaller, simpler controller is created. This very small controller communicates with the LEON3 pipeline to help regulate the control-flow when interrupts occur. This unit is an essential part of synchronizing duplicated or triplicated processor cores when upsets occur in DWC and TMR LEON3 designs (Chapter 3).

Finally, since it is often important to monitor the output of the processor, support for a valid output signal is added to the LEON3 processor. To indicate when to drive this valid signal, a rarely used SPARCv8 instruction is hijacked. The processor output is externally examined only when this commandeered instruction is executed. The instruction chosen for this purpose is the trap never (tn) instruction. Since this instruction opcode is followed by an unsigned integer (normally representing a software trap number), this hijacked instruction can be used for multiple purposes.

When used to indicate that the processor output is valid, the unsigned integer value is set to zero. Listing 2.1 shows how to use this instruction in a normal C program. The location of the \texttt{asm} instruction statement should be directly below an assignment statement. The \texttt{asm} instructions statement at that location ensures that the value being examined on the output of the processor is the value assigned to the variable above the \texttt{tn 0} statement.

Listing 2.1: Example C code using instruction to indicate valid output
\begin{verbatim}
for (i = 1; i < len; i++) {
  if (str[i] > minchar && str[i] < maxchar) {
    newstr[i] = str[i] - 32;
    asm("tn 0");
  } else {
    newstr[i] = str[i];
    asm("tn 0");
  }
}
\end{verbatim}

To demonstrate how the valid output signal is created, consider the snippet of assembly code displayed in Listing 2.2. This is part of the assembly code created after compiling the C code in Listing 2.1. The code shows that the \texttt{tn 0} instruction on line 7 occurs right after a store instruc-
tion. The store instruction at line 6 stores the value in register %i2 at the memory location %o7 + %i5. This store instruction corresponds to the assignment to newstr[i] on line 3 in Listing 2.1. The value that is stored in memory is visible on the output of the processor while the instruction following a store is executed. Since the instruction following this store is the tn 0 instruction, the valid output signal is asserted at the same time as the processor outputs the value assigned to newstr[i].

Listing 2.2: Example assembly code using instruction to indicate valid output

```assembly
.LL14:
1    ld  [%o4+%lo(maxchar)], %g1
2    cmp %i3, %g1
3    bge .LL5
4    add %i4, -32, %i2
5    stb %i2, [%o7+%i5]
6    tn 0
7    add %i5, 1, %i5
8    ld  [%o5+%lo(len)], %i3
9    cmp %i5, %i3
10   bge .LL9
11   nop
```

There are other small modifications that are made to the LEON3 processor in order to support fault-tolerance. Throughout this work, as mitigation techniques are introduced any corresponding processor changes are also discussed. Despite the need to make minor changes to the processor, no changes are made to the SPARC V8 instruction set architecture (ISA).

2.5 Summary

For a softcore processor to be an attractive design choice for space-based applications, they must be protected. There are unique fault-modes for softcore processors compared to ASIC-based processors. The reliability of the logic and routing of an ASIC-based processor is rarely a concern [71]. But since the logic and routing of a softcore processor is implemented in the configuration memory of an SRAM-based FPGA, the logic and routing of a softcore processor are vulnerable to SEUs. Configuration upsets in a softcore processor can be classified as: a logic upset (Figure 2.2(b)), an upset causing a short in the routing (Figures A.2(c) and A.2(d)), or an upset causing an open in the routing (Figure A.2(a) and A.2(b)).
Although the focus of this work is on protecting the configuration memory of the LEON3 softcore processor, similar to ASIC-based processors, the user memory is also vulnerable to SEUs. When protecting the user memory of a softcore processor, it is important to consider the protection of the logic and routing protecting the user memory. To fully protect softcore processor user memory, memory scrubbing is usually required. The next chapter discusses traditional ways of protecting the configuration and user memories.
CHAPTER 3. TRADITIONAL SOFTCORE PROCESSOR FAULT-TOLERANCE

In a harsh radiation environment like space, mitigation techniques must be used to protect softcore processors from SEUs. A significant amount of research has identified hardware techniques to protect FPGA designs in the presence of SEUs. Two of these well-researched techniques include triple modular redundancy (TMR), as well as duplication with compare (DWC) and checkpointing. Although the focus of this work is on protecting a LEON3 softcore processor with software fault-tolerance, this work also considers a LEON3 processor protected with TMR and a LEON3 processor protected with DWC and checkpointing. These two traditional techniques are included to provide reference points for the software fault-tolerant LEON3, and to more fully explore the processor design-space. This chapter discusses how TMR and DWC with checkpointing are applied to the LEON3 softcore processor. Since checkpointing is the upset recovery technique used by all mitigated LEON3 processor designs, the chapter begins by first discussing the unique issues associated with adding checkpointing to softcore processors.

3.1 Fault Recovery Through Checkpointing

Regardless of whether traditional detection techniques (TMR and DWC) or low-cost software detection techniques are used to detect upsets, checkpointing is used to recover when an upset is detected. This section begins by discussing important considerations for applying checkpointing to softcore processors. These considerations include what state to save, how often to checkpoint, and how to protect checkpointed data. Afterward, the concept of detection loops (dloops) is introduced and it is shown how the checkpoint interval and average detection latency affect the frequency of dloops.
3.1.1 Applying Checkpointing to Softcore Processors

Saving State

Checkpointing is performed by periodically saving the state of the processor while a program executes. When a fault is detected, the most recent checkpoint state is used to recover. This is known as rollback checkpointing [46]. Rollback checkpointing reduces recovery time by allowing a program to resume from a saved state instead of having to reset the processor and restart the program from the beginning. Before a rollback is performed, processor execution is temporarily suspended in order for FPGA bitstream scrubbing [26–28] to repair any upsets that may exist in FPGA configuration memory.

When a checkpoint is performed in a softcore processor, the entire processor’s state should be saved. This state includes the processor state registers, register file contents, and other memory hierarchies such as cache and main memory. If the processor uses write-through caches, saving cache contents in a checkpoint is unnecessary. Instead the caches are simply invalidated.

Checkpointing can be used in place of explicit memory scrubbing. When an upset is detected in either the configuration memory or user memory of a softcore processor, a rollback is performed. Performing a rollback causes the checkpointed memory and state register contents to overwrite, or scrub, the processor’s memories and state registers. However, for memories implemented in BRAMs, it is still recommended to regularly check every memory location for upsets using a scrub counter (see Appendix B). If the scrub counter finds an upset, a checkpoint rollback scrubs the erroneous memory location.

Checkpointing Costs

Saving the contents of main memory can be expensive in terms of area and performance. Table 3.1 shows how expensive checkpointing is in terms of area. The table compares the area costs of an unprotected LEON3 processor to a processor with rollback checkpointing implemented. Rollback checkpointing requires 32% additional area compared to an unprotected processor.

The 32% area cost reflects minor additions to the LEON3 that are required to support checkpointing. When a checkpoint occurs, processor state is saved in two different ways. First, the contents of the register file and main memory are saved. Since the caches are write-through,
Table 3.1: Area cost comparison between the unmitigated LEON3 and a LEON3 with rollback checkpointing implemented designs.

<table>
<thead>
<tr>
<th>LEON3 Design</th>
<th>Slices</th>
<th>BRAMs</th>
<th>DSPs</th>
<th>Config Bits (CFGbits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unmitigated</td>
<td>3058 (1.00×)</td>
<td>12 (1.00×)</td>
<td>4 (1.00×)</td>
<td>677,065 (1.00×)</td>
</tr>
<tr>
<td>Checkpoint only</td>
<td>4386 (1.43×)</td>
<td>21 (1.75×)</td>
<td>4 (1.00×)</td>
<td>894,355 (1.32×)</td>
</tr>
</tbody>
</table>

they are invalidated instead of saved. To store this saved memory, a duplicate main memory and register file are needed. A hardware checkpoint controller unit is also added to control what data is sent to and from the main memory, register file, caches, and processor pipeline. For example, when performing a checkpoint, instead of sending data from the main memory to the caches, all of the memory contents are written to the duplicate checkpoint memory.

The second part of a checkpoint is done in software through the use of an interrupt service routine (ISR). This ISR saves all the pipeline state in a reserved register window in the register file. To save (and restore) to (from) this reserved register file window, very minor changes are made to the processor pipeline. When a checkpoint or restore is performed, the register file window pointer is set to the reserved window, and then restored when the ISR is complete. To ensure that the reserved register file window is not overwritten during normal program execution, the reserved window is located outside of the visible set of register windows. There is no hardware cost to add this extra register file window. All of the register file windows are located in half of a BRAM, so the reserved register file window can be located in the unused half of the BRAM.

The performance cost of saving processor state can be large if the main memory is saved in a serial fashion (as discussed in Appendix D), one memory location at a time. But in this work, memory is saved with a negligible performance penalty since main memory is implemented with on-chip block RAMs (BRAMs). Regardless of the size of the main memory, the time required to save or restore the contents of memory will be constant since all of the BRAMs that make up the main memory can be written-to simultaneously. The actual performance cost incurred from checkpointing depends on how frequently checkpoints are taken, but in this work there is a 1% performance cost observed. This cost is in terms of the additional clock cycles required to execute a program when compared to executing a program on an unprotected LEON3 processor.
If checkpointing is done frequently enough, the checkpointed memory itself is protected through implicit scrubbing. Upsets in checkpointed memory locations are repaired when they are overwritten with new checkpointed data. Thus, when upset rates are low, checkpointed data does not need to be explicitly protected. However, as is discussed in the next section, it is not possible to protect checkpointed memory when the data that is saved is itself faulty.

3.1.2 Endless Detection Loops: dloops

An undesirable situation occurs when upset recovery is unsuccessful, and the upset is again detected after each repair attempt. The term introduced in this work to refer to an endless detection loop scenario is a dloop (detection loop). Dloops can occur for many reasons, but the biggest cause of dloops is corruption of checkpoint data. If an upset is detected and corrupted checkpoint data is used to repair the processor, it is possible for the processor to not be adversely affected. But it is more likely that the processor will remain in error. If the processor remains in error, it is possible for detection techniques to again detect that the processor has been upset. This section shows how dloops occur, what influences how often they occur, and shows how they can be reduced.

How dloops Occur

Dloops are a function of the checkpoint interval \(T\) and the detection latency \(t_d\). The checkpoint interval refers to the length of time between two regularly occurring checkpoints. Detection latency refers to the time between the occurrence of an upset and the detection of that upset. Figure 3.1 shows the timeline of a program with \(n\) regularly occurring checkpoints. It also shows that some time before the \(j^{th}\) checkpoint an upset occurs, but that the upset is not detected till after the checkpoint. When the time between the upset occurrence and the next checkpoint is less than the detection latency of the upset then it is possible for the upset to cause the saved checkpoint data to be corrupted, and then for a dloop to occur.

3.1.3 Probability of dloop Occurrence

Since dloops are undesirable, it is important to know how likely they are to occur, and to know how to reduce or eliminate them. The likelihood of a dloop to occur is related to the check-
point interval ($T$), the detection latency ($t_d$), and the checkpoint corruption probability ($\tau$). The checkpoint corruption probability is the probability for an upset to corrupt checkpoint data given that a checkpoint is taken before the upset is detected. Just because the detection latency straddles a checkpoint, it does not mean the checkpoint data will be corrupted. Even if the checkpoint data is corrupted, it does not guarantee that a dloop will result. Equation 3.1 shows a probability model that describes how likely it is for a dloop to occur:

$$P(dloop) = \frac{\tau \cdot t_d}{T}.$$  \hfill (3.1)

This equation assumes that upsets are equally likely to occur anywhere in the checkpoint interval.

The probability that an upset corrupts checkpoint data, given that the upset occurs before the checkpoint is taken, and is detected after the checkpoint ($\tau$) is very low for softcore processors protected with spatial redundancy. In the case of triple modular redundancy (TMR), the data being checkpointed is voted on, thus even if upsets are not explicitly detected, they are masked or out-voted. It is more difficult for temporal techniques to reduce $\tau$, thus dloops are much more likely to occur in softcore processors protected with software fault-tolerance.

### 3.1.4 Preventing dloops

The probability of a dloop occurring can be reduced by reducing $\tau$, increasing $T$, and/or reducing $t_d$. When spacial redundancy mitigation techniques are used, reducing the detection la-
tency \( (t_d) \) is accomplished by strategically selecting which redundant processor signals to compare (Chapter 3). As Chapter 4 shows, it is more difficult to reduce the detection latency when temporal techniques are used. Reducing the detection latency is accomplished by applying a combination of software fault-tolerance techniques.

Protecting checkpoint memory with popular memory protection techniques such as ECC [34, 64] will not prevent dloops from happening. If there are upsets in the data, logic, or routing leading to the protected checkpoint memory, the mitigation technique cannot detect the upset. However, checkpoint memory can be scrubbed. When an upset is detected in the checkpoint memory, performing a rollback will obviously cause a dloop. So instead of performing a rollback, the corrupted checkpoint data is overwritten by signaling for an unscheduled checkpoint to occur. This protection technique will not prevent dloops, but it will reduce dloops caused by upsets in the checkpoint memory.

Dloops can be all but eliminated if checkpoints save data in alternating redundant memories [72]. The timeline in Figure 3.2 shows that as program execution progresses, a checkpoint first saves data at location A. When the next checkpoint is performed, state is saved at location B. As the program continues, checkpoints continue to alternate between saving state at location A and location B. When an upset occurs, the processor is rolled back to the most recent checkpointed state. If the upset occurs right before a checkpoint is taken, normally it would be possible for a dloop to occur. If the checkpointed data is corrupted in such a way that causes the upset to be continuously detected, to prevent a dloop, the processor is rolled back two checkpoints. This causes the uncorrupted checkpoint data to be restored.

Despite the potential for this theoretical approach to significantly reduce dloops, it is not used in this work. It is not used because it cannot guarantee to eliminate dloops, because its area costs are exorbitant, and because the number of dloops that occur in this work are low (Chapter 6). An upset with a detection latency greater than a checkpoint interval could corrupt both checkpoint locations. Also, the costs to implement this style of checkpointing are at least double those specified in Table 3.1. A 65% area cost is considered to be too large to reduce the small number of dloops observed in this work.

The other reason for not using this checkpointing strategy is that it cannot guarantee to eliminate dloops. There are two assumptions made by this theoretical checkpoint approach which
do not hold in this work. The first assumption is that detection latencies do not span more than one consecutive checkpoint. If the detection latency spans two checkpoints, then a dloop is still possible. To prevent this scenario, the checkpoint interval \((T)\) should be longer than the worst case detection latency. But as is shown in Chapter 6, some of the detection latencies (a very small percent) are longer than the checkpoint interval. The second assumption is that a single upset will not affect both sets of checkpoint data. However, it is possible for one SEU to have widespread effects \([24, 73]\).

### 3.1.5 Checkpoint Interval

When checkpointing is used, it is important to know how often to checkpoint. Appendix D introduces a checkpoint probability model for softcore processors. That probability model balances the overhead associated with checkpointing too often, with the time wasted re-executing instructions when a rollback is required when checkpointing is performed too infrequently. The consequences of checkpointing too often or not often enough are the same – a program takes longer to execute. When checkpointing is done too frequently, there is an unnecessary amount of overhead due to saving processor state. When not done frequently enough, there is a large amount of work to redo when a rollback is performed. Appendix D also shows that when checkpointing is not done at all, the chance of a program completing without error approaches zero as the error rate increases.
In the probability model introduced in Appendix D, there is one assumption that must be made in order to significantly reduce dloops. When bitstream scrubbing is used, the checkpoint interval must not be less than the bitstream scrub interval. The worst case detection latency of an upset in the bitstream is equal to the scrub interval. Thus, if the checkpoint interval is less than the scrub interval, multiple checkpoints are taken before the upset is repaired, which significantly increases the likelihood for checkpoint data corruption and dloops.

Throughout the rest of this work it is assumed that checkpointing is performed every 18,000 cycles. The runtime of the testbench programs used in this work (Chapter 5) is so small that the optimal number of checkpoints to run is one per program execution. A checkpoint interval of 18,000 cycles ensures that one checkpoint is taken in a single run of each of the programs in the testbench suite (Chapter 5). A consequence of using a small checkpoint interval is that the checkpointing overhead is larger than it would be in practice. The second consequence of a small checkpoint interval is that the number of dloops that occur will be larger than in practice. Since hardware fault-injection is used to insert faults, there is no danger of using a checkpoint interval that is smaller than the bitstream scrub interval (hardware fault-injection does not use a bitstream scrubber). However, when radiation testing is used to inject faults (Appendix H), or when a softcore processor is used in space it is important to use a more realistic checkpoint interval - one that is at least as long as the bitstream scrub interval.

3.2 Traditional Softcore Processor Mitigation Techniques

The two traditional mitigation techniques implemented in this work include triple modular redundancy (TMR) with roll-forward checkpointing, and duplication with compare (DWC) and rollback checkpointing. DWC with rollback checkpointing is used for comparison since it is a popular processor reliability technique [32, 33]. TMR with roll-forward checkpointing is included in the comparison because TMR is the most common SRAM-based FPGA design reliability technique [22, 25, 27, 28, 32, 45].
3.2.1 DWC with Checkpointing

The first mitigation technique used to protect the LEON3 processor uses duplication with compare (DWC) and rollback checkpointing (Figure 3.3). DWC is a reliability technique that uses duplication to detect upsets. Duplication is a popular processor reliability technique that can either be used in lockstep [46] or dual-core execution (DCE) [12]. This study uses duplicated processors in lockstep.

![LEON3 processor with DWC and checkpointing.](image)

Figure 3.3: LEON3 processor with DWC and checkpointing.

Lockstep refers to multiple processors running the same program with identical instruction sequences, memory load values, and interrupts. A *strict* lockstep requires the processors to run identically on a cycle-by-cycle basis. With a traditional hardcore processor, it can be difficult to detect if two processors are in strict lockstep since the program counters and instruction registers are not exposed on a cycle-by-cycle basis. Instead of strict lockstep, hardcore processors determine if two processors are in lockstep by comparing processor state at regular intervals [74].

Softcore processors have the luxury of using strict lockstep since *any* processor signal can be made visible on a cycle-by-cycle basis. The design of a softcore processor can be changed at any time to use any given signal for lockstep comparison. The lockstep processors in Figure 3.3 compare register file outputs, program counters, instruction registers, processor output, and main memory input in a strict lockstep fashion.
This kind of strict lockstep is simple and provides a very low detection latency since comparison is done on a cycle-by-cycle basis. When an SEU is detected quickly, its effects are prevented from spreading throughout the processor. But when left undetected for a long time, one upset can spread throughout the processor, acting like a multi-bit upset. For example, the effects of an upset in the register file can quickly spread to the cache and main memory, or spread throughout the pipeline. DWC in cycle-by-cycle lockstep detection provides a low detection latency which prevents the propagation of SEUs.

In addition to detecting upsets, it is important to also resynchronize duplicated processors when an upset occurs. Resynchronization restores both processors to the same cycle-accurate, lockstep state. For example, if an upset changes the program counter of one of the processors in lockstep, the synchronized, lockstep program execution is derailed. Detecting this upset can be done simply by comparing the duplicated program counters. However, there is more to recovery than simply restoring the latest checkpoint data - both processors also need to be resynchronized. Resynchronization is done by the interrupt controller which interrupts both processors when an upset is detected. The interrupt controller interrupts both processors regardless of what instruction is being executed, and forces them to both start executing the checkpoint recovery interrupt service routine. An important part of this rollback interrupt service routine is restoring program state data, which includes restoring the program counter of both processors.

Duplication cannot be used to detect upsets in the interrupt controller since it is at the heart of controlling the checkpoint recovery and resynchronization. If the interrupt controller is duplicated, and the duplicated controllers are out of sync, both controllers attempt to resynchronize both processors, resulting in a dloop. Although a single interrupt controller provides a single point of failure, it is important to have a single point of resynchronization to prevent dloops. For this reason, the DWC and checkpointing design does not duplicate the interrupt controller (as shown in Figure 3.3). A single interrupt controller eliminates dloops from the DWC LEON3 processor.

### 3.2.2 TMR with Checkpointing

The second reference design used for comparison is a LEON3 processor using TMR and roll-forward checkpointing [46] for protection (Figure 3.4). TMR is a very popular reliability tech-
nique for SRAM-based FPGA designs [22, 25, 27, 28, 32, 45]. Figure 3.4 shows that the processor core, the interrupt controller, checkpoint control, and voter are triplicated.

![Figure 3.4: LEON3 processor with TMR and checkpointing.](image)

Similar to the duplicated processor design, the TMR design keeps the triplicated processors in strict lockstep execution. Like the DWC design, the register file outputs, program counters, instruction registers, processor outputs, and main memory inputs are used to keep the processors in lockstep. But unlike DWC design, when one of the processors falls out of lockstep, a rollback is not required. Instead, the state of one of the two processors still in lockstep is used to correct and resynchronize the one that is out of lockstep. This acts like a roll-forward checkpoint [46].

Triplication is used to detect upsets, while voters and roll-forward checkpointing provide recovery and resynchronization. Triplicated interrupt controllers do not have the synchronization problems that duplicated interrupt controller have. Unlike the duplicated interrupt controllers, if one of the three interrupt controllers has been upset, the faulty controller can be identified and out-voted, preventing dloops. Thus, when applying TMR to the LEON3 softcore processor, it is important to place voters on the output of the triplicated interrupt controllers. Voters are also placed inside the checkpoint controllers.

Two versions of the TMR LEON3 design are used in this work. The first version triplicates all of the inputs and outputs of the LEON3 processor. Triplicated inputs remove single points of failure in the clock and reset lines, while triplicated outputs remove single points of failure on
the outputs signals. Reliable off-chip voting is required when triplicated outputs are used. But, there are cases where it is not possible to triplicate inputs and outputs. For example, the triplicated processor may be part of a larger untriplicated system. In this case the outputs will need to converge to a single domain on-chip, and there will only be one clock and reset signal. The second TMR design uses untriplicated inputs and outputs.

3.2.3 LEON3 Modifications

In addition to detecting upsets using hardware redundancy, the mitigated LEON3 softcore processors use memory scrubbing on the register file, caches, and main memory. Since each of these memories is implemented using BRAMs, deterministic scrubbing is used (every memory location is regularly checked for upsets). Appendix B shows that it is important for BRAMs to be scrubbed deterministically, especially when being used as a ROM. The lower addresses in main memory (where the program instructions are stored) are treated as a ROM, so they are especially important to scrub. It is common for some register file windows to remain static for a long period of time, thus these static parts of memory are also critical to scrub. Scrubbing the caches is also important since it is possible for parts of them to remain static for long periods of time.

Adding scrubbing to the LEON3 requires a very small modification to the memory controller and cache controllers. It also requires a small control unit, including scrub counters. As a scrub counter continually marches through BRAM addresses, the DWC and TMR LEON3 designs use redundancy to detect scrub errors, while the software fault-tolerance processor uses parity to detect if there is an upset at each memory location. Since the register file and main memory use both ports on a dual-ported BRAM, the scrub counters must share the ports with the processor. When the processor is not using them, the scrubber uses them.

Another modification makes control signals available on the output of the LEON3. Custom control signals are sent out of the LEON3 along with the processor output data, which indicate when upsets in mitigated LEON3 designs have been detected and repaired. An output module is created to pass these signals out of the processor design. The compare unit of the DWC LEON3 design, and the on-chip voters in the TMR design are also included in this output module. In the LEON3 processor protected with software fault-tolerance, the parity check unit is located in the output module.
3.3 Cost Comparisons for Traditional Softcore Processor Mitigation

These traditional mitigation techniques are often used because of the protection they provide, but they are expensive in terms of area. In order to establish their reliability-area-performance trade-off, this section identifies the area and performance costs of these softcore processor mitigation techniques. Reliability is considered in Chapter 6. The costs identified in this section can be combined with reliability to identify points in the processor design-space.

3.3.1 Area Costs

The area costs of the unmitigated LEON3 processor are compared with the mitigated LEON3 designs in Table 3.2. Area is compared in terms of slices, BRAMs, DSPs, and configuration memory bits (CFGbits). The values in braces show the area increase compared to the unmitigated processor. The table shows that the area cost of DWC is more than double and that the area cost of TMR is more than triple. The costs of adding checkpointing hardware account for most of the additional overhead.

Table 3.2: Area cost comparison for the unmitigated and traditionally mitigated LEON3 processor designs.

<table>
<thead>
<tr>
<th>LEON3 Design</th>
<th>Slices</th>
<th>BRAMs</th>
<th>DSPs</th>
<th>Config Bits (CFGbits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unmitigated</td>
<td>3058 (1.00x)</td>
<td>12 (1.00x)</td>
<td>4 (1.00x)</td>
<td>677,065 (1.00x)</td>
</tr>
<tr>
<td>DWC (1 clock)</td>
<td>8628 (2.82x)</td>
<td>33 (2.75x)</td>
<td>8 (2.00x)</td>
<td>1,834,002 (2.71x)</td>
</tr>
<tr>
<td>DWC (2 clocks)</td>
<td>8569 (2.80x)</td>
<td>33 (2.75x)</td>
<td>8 (2.00x)</td>
<td>1,837,719 (2.71x)</td>
</tr>
<tr>
<td>TMR (non-trip. in/out)</td>
<td>13,840 (4.53x)</td>
<td>36 (3.00x)</td>
<td>12 (3.00x)</td>
<td>2,907,842 (4.29x)</td>
</tr>
<tr>
<td>TMR (trip. in/out)</td>
<td>13,933 (4.56x)</td>
<td>36 (3.00x)</td>
<td>12 (3.00x)</td>
<td>2,933,719 (4.33x)</td>
</tr>
</tbody>
</table>

The area costs in Table 3.2 show that full DWC and TMR are very expensive. As a lower cost alternative, only the most sensitive processor components can be replicated. Alternatively, low-cost software fault-tolerance can be used to protect the softcore processor or some of its components (Chapter 4).
3.3.2 Performance Costs

The performance cost (\( \rho \)) incurred by the application of mitigation techniques is reported in terms of the additional number of clock cycles needed, on average, to run a program compared to when running on an unmitigated processor. The only performance penalties associated with TMR or DWC with checkpointing is related to the addition of checkpointing. The small cost introduced by checkpointing is proportional to the time it takes to perform a checkpoint and the frequency with which checkpoints are taken. As previously mentioned, in this work a checkpoint is taken every 18,000 clock cycles.

Table 3.3 shows that on average, programs on the DWC processors run 1.01\( \times \) longer than when run on an unmitigated processor. The time to record checkpoint information is what accounts for this performance penalty. The performance cost for the TMR processor is negligible since the only incurred performance cost comes when roll-forward checkpointing corrects and resynchronizes one of the three processors. Table 3.3 also shows the average code growth when checkpointing is added. The growth comes as a result of the addition of interrupt service routines which save and restore program checkpoint state.

Table 3.3: Average performance and code size costs for the unmitigated and traditionally mitigated LEON3 processor designs.

<table>
<thead>
<tr>
<th>LEON3 Design</th>
<th>Code Size</th>
<th>Run Time Cost (( \rho ))</th>
<th>Avg. Detection Latency (cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unmitigated</td>
<td>1.00( \times )</td>
<td>1.00( \times )</td>
<td>N/A</td>
</tr>
<tr>
<td>DWC (1 clock)</td>
<td>1.04( \times )</td>
<td>1.01( \times )</td>
<td>2392</td>
</tr>
<tr>
<td>DWC (2 clocks)</td>
<td>1.04( \times )</td>
<td>1.01( \times )</td>
<td>2488</td>
</tr>
<tr>
<td>TMR (non-triplicated in/out)</td>
<td>1.04( \times )</td>
<td>1.00( \times )</td>
<td>498</td>
</tr>
<tr>
<td>TMR (triplicated in/out)</td>
<td>1.04( \times )</td>
<td>1.00( \times )</td>
<td>734</td>
</tr>
</tbody>
</table>

Table 3.3 also shows the average detection latency for the TMR and DWC with checkpointing designs. Detection latency is an important consideration for the occurrence of dloops, and is also an important variable in determining the optimal checkpoint interval (Appendix D). The
table shows that a LEON3 processor protected with TMR has a detection latency that is almost $5\times$ smaller than a LEON3 protected with DWC and rollback checkpointing.

### 3.4 Summary

Checkpointing is the only upset recovery technique used throughout this work. When applying checkpointing to a softcore processor, it is important to consider how often to perform checkpoints. Checkpointing too often results in reduced performance due to excessive checkpoint overhead, and increases the potential for dloops. Checkpointing too infrequently causes a reduction in performance due to the large amount of work to redo when a rollback occurs. The checkpoint interval used in this work is 18,000 cycles.

Traditional mitigation techniques are considered as a baseline to compare with the software fault-tolerant techniques. DWC with rollback checkpointing, and TMR with roll-forward checkpointing provide design-space points that can be compared with the design-space points provided by a LEON3 softcore processor protected with software fault-tolerance. TMR and DWC with checkpointing are expensive in terms of area, but incur virtually no performance cost. These techniques will be compared against the techniques described in the next chapter.
CHAPTER 4. SOFTWARE DETECTION TECHNIQUES FOR SOFTCORE PROCESSORS

As an alternative to TMR or DWC, low-cost terrestrial-based software detection techniques can be used to detect upsets in the configuration memory of a softcore processor. Software techniques detect upsets in a softcore processor at the cost of time instead of area, by executing redundant instructions meant to detect SEUs. Once detected, checkpointing [32, 33] is used to recover from the upset. Softcore processors can take advantage of software detection techniques to provide mitigation at a much lower spatial cost than TMR and DWC with checkpointing.

This chapter introduces the software fault-tolerance techniques used to detect upsets in the configuration memory of the LEON3 softcore processor, and discusses the costs associated with each technique. The software techniques used to detect SEUs include a modified version of SWIFT [29, 36], consistency checks [31, 34], and software indicators [37, 38]. Parity and memory scrubbing are used to detect upsets in the user memory.

Each of these techniques will be applied individually as well as collectively to detect upsets in a LEON3 processor. Softcore processors become a very attractive option for space-based applications if the low-cost techniques introduced in this chapter can be used in conjunction with checkpointing to detect and recover from upsets in the processor’s configuration memory. The individual and collective reliability of the fault-tolerance techniques introduced in this chapter, are discussed in Chapter 6, where they are evaluated based on the detection coverage they provide to the individual components of the LEON3 softcore processor.

4.1 Modified SWIFT

Software implemented fault tolerance (SWIFT) [29] is a known software reliability technique for detecting transients in user memory. The memory elements intended for protection include the register file, pipeline control, and other processor state. SWIFT consists of two popular
software reliability techniques: code duplication [36, 75–77] to detect transients in the data, and control-flow monitoring [30, 36, 78, 79] to detect transients in the control-flow. This section discusses the changes that are made to SWIFT to protect a softcore processor.

Although SWIFT is well suited for detecting upsets in ASIC-based processors, there are two assumptions it makes that do not apply to softcore processors. The first assumption is that transients in memory elements are the predominant kind of upset to be detected. In softcore processors, transients are considered to be insignificant compared to SEUs in the configuration or user memories. So the goal of detection techniques for softcore processors changes from detecting transients in memory elements to detecting upsets in the logic and routing of softcore processor.

The second assumption is that SWIFT’s *sphere of replication* (SoR) does not include the memory hierarchy [36]. A SoR refers to the logical domain protected by code duplication. In ASIC-based processors, detection techniques such as ECC are sufficient for detecting upsets in the memory hierarchy. Thus, SWIFT relies on these detection techniques for memory hierarchy protection. But softcore processor reliability techniques must detect upsets in the logic leading to the memory hierarchy (Appendix B), thus the memory hierarchy cannot be excluded from the SoR. These differences between ASIC-based processors and softcore processors change how SWIFT can protect a softcore processor. Because of these differences, this work modifies how SWIFT is applied. Code duplication is replaced with selective hardware duplication.

### 4.1.1 Register File CDWC

The traditional SWIFT technique detects transient upsets in data with code duplication. Code duplication uses instruction redundancy to detect transients in the register file and ALU. The SWIFT form of code duplication, which is accomplished with a compiler, incurs a 1.41× performance cost. When a non-SWIFT form of code duplication is attempted at the assembly code stage (i.e. after a compiler has already performed register allocation), code duplication incurs a much larger performance penalty and requires changes to the instruction set architecture (Appendix F).

Since SWIFT’s purpose of code duplication in ASIC-based processors is not applicable for softcore processors, and since code duplication can incur a large performance cost, code duplication is replaced with a less costly upset detection technique. One of the most important roles
of code duplication is to protect the register file contents. Upsets in the register file can quickly spread throughout the processor, so they must be detected quickly [38]. A replacement upset detection technique for code duplication must be able to quickly detect upsets in the register file. A very low cost way to detect these upsets is to apply complement duplicate with compare (CDWC) to the register file (Figure 4.1). A study on FPGA memory reliability reveals that complement duplicate with compare is an effective protection technique for memory elements (Appendix B). Since the SoR for a softcore processor includes the memory hierarchy, CDWC is also applied to the cache tags. The cache contents and main memory are protected with parity (discussed later in this chapter).

CDWC is actually a hardware technique not a software technique, but it is only implemented on the register file and cache tags, so the area cost it incurs is small. Figure 4.1 shows that this detection technique adds a second register file is added to the processor pipeline. When a value is normally written to the register file in the write-back pipeline stage, the complement of that value is also written to the complement address location of the duplicate register file. When values are read from the register file, the complemented value is also read from the duplicated register file. If the value read from the register file is not the same as the complement of the value read from the duplicate register file, an upset is detected, and a checkpoint rollback is signaled.

![Diagram of CDWC](image)

Figure 4.1: Complement duplicate with compare (CDWC) is used only on the register file as an alternative to instruction duplication.
In contrast to assembly code duplication, using CDWC on the register file and cache tags has a much lower detection latency, requires no ISA changes, and has no performance penalty. However, it does have an area cost. The area cost of a LEON3 processor protected with this detection technique is compared in Table 4.1 with the area costs of an unmitigated LEON3 processor. The value in parentheses indicates how much more area is required compared to an unmitigated LEON3 softcore processor. The addition of the CDWC register file requires $1.10 \times$ more area (in terms of configuration bits). This additional area is consumed by error detection logic, as well as routing for the duplicated register file and cache tags.

<table>
<thead>
<tr>
<th>LEON3 Design</th>
<th>Slices</th>
<th>BRAMs</th>
<th>DSPs</th>
<th>Config Bits (CFGbits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unmitigated</td>
<td>3058 (1.00×)</td>
<td>12 (1.00×)</td>
<td>4 (1.00×)</td>
<td>677,065 (1.00×)</td>
</tr>
<tr>
<td>CDWC Register File</td>
<td>3332 (1.09×)</td>
<td>14 (1.17×)</td>
<td>4 (1.00×)</td>
<td>746,006 (1.10×)</td>
</tr>
</tbody>
</table>

Clearly, the detection coverage provided by register file and cache tag CDWC centers around the contents of the register file and cache tags, as well as their logic and routing. As will be shown in the next chapter, that coverage spills over to other parts of the processor. Some upsets in the logic and routing that drive the register file and cache tags are also detected.

### 4.1.2 Control-Flow Monitoring

The second part of SWIFT uses software control-flow monitoring [30, 36, 78] to protect against control errors. A true SWIFT implementation is based on control-flow checking by software signatures (CFCSS) [30]. The control-flow checking used in this work starts with CFCSS but adds improvements (ICFCSS) [78]. CFCSS is introduced first before discussing its improvements. To our knowledge, this work represents the first work to apply these control-flow monitoring algorithms to a softcore processor implemented on an SRAM-based FPGA to protect against firm errors, rather than transients.
CFCSS

This subsection introduces control-flow monitoring and shows how it is applied to assembly code. Before discussing the details of the algorithm, an overview is provided. The algorithm for applying CFCSS to a program is shown in Algorithm 1. CFCSS first breaks a program into basic blocks and assigns a static signature to each block. A basic block is defined as a maximal sequence of instructions that has only one entry point at the beginning of the block, and only one exit point at the end of the block. Figure 4.2 shows an overview of how CFCSS works. As a program executes, a dynamic signature ($d_3$ in Figure 4.2) indicates within which block the program is executing. Instructions are added at the beginning of each block (header3 in Figure 4.2) which compare the dynamic signature with the static signature ($s_3$ in Figure 4.2) of the block. If the dynamic signature ever differs from the static block signature, the block was entered erroneously and a control flow upset is detected.

The details of CFCSS are now discussed in more detail. CFCSS begins by creating a directed graph from the basic blocks of a program and then assigning each graph node ($v_i$), at compile-time, a unique runtime signature ($s_i$). While the program executes, a dedicated runtime
signature register (GSR) is used to hold the run-time signature (G) of the current node. When the program executes correctly, the value in GSR (G) is equal to current node’s signature (s_i).

In order for processors to support control-flow monitoring, one of the available registers must be dedicated as the GSR, or a new register must be created to act as the GSR. In the LEON3 implementation of CFCSS, one of the global registers is permanently assigned as the GSR (%G7). Since this register is commandeered for control-flow monitoring purposes, it is no longer available for normal program use.

When control transfers from one block to another (from the source block s, to the destination block d), a new runtime signature (G_d) is generated using a signature function (f). The signature function uses the signature of the previous node (s_s) and the signature of the current node (s_d). The signature function used in this work XORs these values together with the GSR (G_s) to produce the new GSR value (G_d):

\[
G_d = G_s \oplus s_s \oplus s_d = G_s \oplus d_d.
\]

The value d_d = s_s \oplus s_d is referred to as the dynamic signature of the destination node (v_d), and is calculated at compile-time. But since a node (v_d) can have multiple source nodes (i.e.: a fan-in greater than one), there needs to be a way to correctly generate the runtime signature (G_d) regardless of which source node enters the destination node. A second dedicated register called the runtime adjusting signature register (ASR) holding the dynamic adjustment value (D), allows the correct runtime signature (G) to always be calculated. In the LEON3 implementation of CFCSS, another of the global registers is assigned to be the ASR (%G6). At compile time, a node (v_d) that has a fan-in greater than one, arbitrarily chooses one of its source nodes (v_s) to calculate its dynamic signature (d_d). To account for each of the other predecessor nodes (v_p), before performing the signature function (f), the GSR is first XOR’ed with the ASR:

\[
G_d = D \oplus G_s \oplus d_d.
\]
The value of the ASR is set in each predecessor node \( (v_p) \) before entering the destination node \( (v_d) \):

\[
D = s_p \oplus s_s.
\]

**Algorithm 1**: Control-Flow Checking by Software Signatures (CFCSS)

```c
CFCSS begin
    Create a directed graph of basic blocks from the program;
    Assign a unique static signature \( s_i \) to each node \( v_i \);
    foreach \( v_i \) do
        if \( v_i \) has only one predecessor \( v_s \) then
            \( d_i = s_i \oplus s_s \);
            At the top of the code block add the instruction: \( G = G \oplus d_i \);
        else
            Arbitrarily choose a predecessor \( v_s \) of \( v_i \) and calculate \( d_i = s_i \oplus s_s \);
            At the top of the code block add the instruction: \( G = G \oplus d_i \);
            After the previously added instruction add: \( G = G \oplus D \);
        endif
    endforeach predecessor \( v_p \) of \( v_i \) do
        In predecessor code block, after the signature check, add the instruction: \( D = s_s \oplus s_p \);
    endforeach
    Compare the value of \( G \) with \( s_i \);
    Branch to an error routine if not equal;
end
```

Algorithm 1 shows how a compiler, or assembler applies CFCSS to a program. In this work, the SPARCV8 compiler compiles C code to assembly, and then a custom assembler applies control-flow monitoring to each of the code blocks.

To demonstrate how the CFCSS algorithm works, consider again the example program in Figure 4.2. After creating a directed graph from the basic blocks of the program, unique static signatures \( (s_i) \) are assigned to each node \( (v_i) \), and each node’s dynamic signature \( (d_i) \) is calculated. Figure 4.3(a) shows the directed graph, and the signature calculations for each node. Figure 4.3(b) shows the instructions that are added to code block (node) four. The added instructions are shown with a grey background. When the value in the runtime signature register \( (%G7) \) is equal to the value of the current block’s static signature \( (s_4 = 3) \), the branch instruction jumps to the instruction block, otherwise the error instruction \( (\text{tn} 2) \) is executed, which triggers a rollback. Since the
SPARCv8 architecture has a branch slot, the instruction in the branch slot of the branch that skips over the error instruction sets the new value of the runtime adjusting signature register (%G6). This instruction ensures that the value of %G6 is correct when block five is entered (since block five has a fan-in that is greater than one).

Now consider what happens when code block four is correctly entered from block three during normal program execution. The value in %G7 is block three’s static signature value ($s_3 = 9$). To check that block four has been entered legally, %G7 is XOR’ed with block four’s static signature and with %G6 (since block four has a fan-in greater than one). When entering block four from block three, the value in %G6 has been set by block three. The runtime signature check verifies that the new value of %G7 equals block four’s static signature:

\[
%G7 \quad = \quad %G7 \oplus %G6 \oplus d_4 \\
= \quad s_3 \oplus (s_3 \oplus s_2) \oplus (s_4 \oplus s_2) \\
= \quad s_4 \\
= \quad 3.
\]
Next, consider what happens if an upset causes block four to be entered erroneously from block five. Assume that block five was previously entered from block one. Thus, when block five is exited, the value in %G6 is $s_0 \oplus s_2$. Because code block four has a fan-in greater than one, %G6 is included in block four’s runtime signature check:

$$\%G7 = \%G7 \oplus \%G6 \oplus d_4$$

$$= s_5 \oplus (s_0 \oplus s_2) \oplus (s_4 \oplus s_2)$$

$$= s_5 \oplus s_0 \oplus s_4$$

$$\neq s_4.$$ 

The runtime signature check fails, the upset is detected, and a rollback is signaled.

Errors like this are common for processors in the presence of SEUs. An upset to any control instruction such as a branch, jump, trap, or function call can cause correct program flow to be derailed. Of course, direct upsets to the program counter can also cause the processor to begin executing instructions in the wrong part of the program. Control-flow monitoring detects these upsets as shown in the previous examples.

**ICFCSS**

There are two problems with the CFCSS algorithm previously discussed. First, the overhead associated with CFCSS is usually relatively large. Basic blocks are often small, thus the four to six overhead instructions that CFCSS adds can dwarf the instructions in the original code block (Figure 4.3(b)). The second problem is that it is possible for aliasing to occur with CFCSS. Aliasing refers to different node transitions (legal or illegal) in the directed program graph being indistinguishable from one another. The result of aliasing is that some illegal transitions are not detected, and some legal transitions are treated as illegal transitions. Aliasing occurs when $d_i = s_s \oplus s_i$, $d_j = s_s \oplus s_j$, but the predecessors of $v_i \neq$ predecessors of $v_j$. This aliasing problem is observed in each of the testbench programs used in this work.

To observe an aliasing problem that causes an illegal transition to go undetected, consider what would happen if an upset caused block four in Figure 4.3 to be entered erroneously from
block zero. In code block zero, the value of \( \%G6 \) is \( s_0 \oplus s_2 \). Since code block four has a fan-in greater than one, \( \%G6 \) is included in block four's runtime signature check:

\[
\%G7 = \%G7 \oplus \%G6 \oplus d_4 \\
= s_0 \oplus (s_0 \oplus s_2) \oplus (s_4 \oplus s_2) \\
= s_4 \\
= 3.
\]

The illegal transition produces a correct runtime signature check, thus the error goes undetected.

Both of these two problems are addressed with the improved CFCSS (ICFCSS) algorithm introduced by Yan et al. [78]. The first problem with CFCSS is addressed by using superblocks [80] instead of basic blocks. A super block is a maximal sequence of instructions that has only one entry point at the beginning of the block, but can have multiple exit points. Allowing for multiple exit points increases the size of the block, which reduces the overall overhead of control-flow checking. The exception to Yan’s code block definition is that function calls can only occur at the end of a block. Since function calls return, the return acts as an entrance to a new code block.

The second problem with CFCSS (aliasing) is resolved with the changes that are required in order to correctly support multiple exit points in a code block. Instead of setting the value of the runtime adjusting signature register (\( \%G6 \)) once at the top of the code block, it must be set right before leaving the code block. The value assigned to \( \%G6 \) depends on the destination code block.

**Modified ICFCSS**

An improvement to ICFCSS which is introduced in this work allows for larger code block sizes and less control-flow monitoring overhead. The improvement introduced in this work allows function calls to be included in the middle of a code block. When the function returns, it returns to the same point in the code block. Thus, the definition of code block used in this work is a maximal sequence of instruction that have a single point of entry, at the beginning of the block (with the exception of a function call return), but which can have multiple exit points (including function calls). To allow for function calls in a code block to return to the same point in the code block, the
%G7 register value must be restored to the code block’s static signature value upon completion of the function (since the function modifies %G7 when it performs a runtime signature check).

**Algorithm 2:** Modified, Improved Control-Flow Checking by Software Signatures (Modified ICFCSS)

**Modified ICFCSS**

```
begin
  Create a directed graph of basic blocks from the program;
  Assign a unique static signature si to each node vi;
  foreach vi do
    if vi has only one predecessor vs then
      di = si ⊕ ss;
      At the top of the code block add the instruction: G = G ⊕ di;
    else
      Arbitrarily choose a predecessor vs of vi and calculate di = si ⊕ ss;
      At the top of the code block add the instruction: G = G ⊕ di;
      After the previously added instruction add: G = G ⊕ D;
      foreach predecessor vp of vi do
        In predecessor code block, right before the instruction that jumps to node vi, add the instruction: D = ss ⊕ sp;
      end
    end
  Compare the value of G with si;
  Signal an error if not equal;
  foreach function call do
    Add an instruction to restore G right after the call returns
  end
```

The modified ICFCSS algorithm is shown in Algorithm 2. Figure 4.4 shows an example of the modified ICFCSS algorithm applied to a block in the Fhourcorners testbench program. The original code block has 14 instructions, and four exit points (including two function calls). The overhead added to the code block are the five instruction which perform the runtime signature check. These overhead instructions are grouped in their own code block immediately before the code block they protect. This couplet of code blocks is treated as a single code block in the modified ICFCSS algorithm. The next part of the algorithm sets the value of the runtime adjusting signature (%G6) right before each of the four exit points in the block. Finally, the value of the runtime signature register (%G7) is restored after each function call.
Figure 4.4: Modified ICFCSS added to a code block in the Fhourcorners testbench program.

Costs

The primary cost for implementing control-flow monitoring is the time required to execute checking instructions. In the code block shown in Figure 4.4, there are 11 added instructions to the original 14 instructions in the code block. Of all the software fault-tolerance techniques introduced in this chapter, control-flow monitoring has the largest performance cost. Table 4.2 shows the code growth for each of the testbench programs (Chapter 5), as well as the performance cost ($\rho$). Performance cost represents the increase of clock cycles relative to a LEON3 processor that does not have these additional instructions.

Table 4.2: Code increase and performance cost of control-flow monitoring.

<table>
<thead>
<tr>
<th>Control-Flow Monitoring Code Block Growth</th>
<th>Avg # Additional Insts./Block</th>
<th>Performance Cost ($\rho$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fhourcorners</td>
<td>5.8</td>
<td>1.31 $\times$</td>
</tr>
<tr>
<td>Hanoi</td>
<td>8.0</td>
<td>1.52 $\times$</td>
</tr>
<tr>
<td>MatrixMult</td>
<td>6.4</td>
<td>1.58 $\times$</td>
</tr>
<tr>
<td>ToUpper</td>
<td>6.0</td>
<td>1.60 $\times$</td>
</tr>
<tr>
<td>Average</td>
<td>6.6</td>
<td>1.50 $\times$</td>
</tr>
</tbody>
</table>
There is a small area cost for implementing control-flow monitoring. Table 4.3 compares the area cost of the unmitigated LEON3 processor with a LEON3 processor that detects upsets with control-flow monitoring. The table shows that control-flow monitoring incurs less than 1% area cost. This area cost is accounted for by control signals that are added to the LEON3 pipeline to monitor which type of software detection technique detected an SEU.

Table 4.3: Area costs of a LEON3 processor protected with control-flow monitoring and rollback checkpointing.

<table>
<thead>
<tr>
<th>Control-Flow Monitoring Area Costs</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEON3 Design</td>
</tr>
<tr>
<td>--------------</td>
</tr>
<tr>
<td>Unmitigated</td>
</tr>
<tr>
<td>Control-Flow</td>
</tr>
</tbody>
</table>

Undetected Upsets

The detection coverage provided by control-flow monitoring centers on the state registers of the processor pipeline which regulate control flow. These registers include the program counters (PCs), instruction registers (IRs), processor state register (PSR), window invalid mask (WIM), implementation-dependent ancillary state registers (ASRs), and trap base register (TBR). Protecting the logic and routing surrounding these state registers naturally provides upset detection coverage to other parts of the processor. For example, since the instruction cache feeds the instruction registers, control-flow monitoring provides significant protection to the logic and routing of the instruction cache.

Despite the large amount of upset detection coverage that control-flow monitoring provides to the control registers and their surrounding logic and routing, there are specific types of upsets that control-flow monitoring cannot detect. The first type of undetected upset is the direct result of using super blocks. An upset that prevents a branch from being taken, when it should be, will not be detected with control-flow monitoring. One cause for such an error is the modification of the integer condition codes (ICCs) in the PSR. Although control-flow monitoring may not catch this
type of upset, consistency checks (discussed in the next section) will catch this kind of upset. If the branch that is supposed to be taken is an unconditional branch at the end of the super block, control-flow monitoring can detect this by placing an error indicating instruction \((\text{tn} \ 2)\) after the unconditional branch at the end of the super block. If an upset prevents the branch from being taken, the error signaling instruction is executed.

Another type of upset that may go undetected by control-flow monitoring is an upset to the program counter that causes the processor to stride through the program instructions. This kind of upset will go undetected if the instruction that signals an error \((\text{tn} \ 2)\) is skipped over while the processor strides through the instruction memory. This kind upset can be detected by control-flow monitoring if the processor happens to stride to an error signaling instruction. This kind of upset is more likely to be detected by control-flow monitoring with the help of the compiler or assembler. Control-flow monitoring is more likely to catch this kind of upset if the error indicating instructions throughout the program are separated by an odd number of instructions. An odd spacing will make striding through memory more likely to execute one of the error indicating instructions. But even with odd spacing, it is possible for control-flow monitoring to stride past the program code and into the data section of memory. This kind of upset is detected by software indicators (discussed later in this chapter).

Finally, upsets to the current window pointer (CWP) within the PSR are rarely covered by control-flow monitoring. The current window pointer determines which register file window is currently active. Upsets to the CWP cause data from other register windows to be overwritten. It also can cause data to be stored in parts of the register file which may not be accessible in the future. This type of upset is difficult for any of the detection techniques discussed in this chapter to detect.

### 4.2 Consistency Checks

Performing consistency checks is a well known software reliability technique used to detect upsets in registers and functional units [34,81,82]. This technique works by periodically executing a set of instructions that verify the correct operation of functional units. For example, Figure 4.5 shows how an upset in an operand register leading to a functional unit is detected. An operation is performed with pre-determined operands, and the output is compared with the expected result.
An upset in operation or operand registers changes the value of what is expected to come out of the functional unit. Thus, the upset is detected when the output of the functional unit is compared with the expected result.

![Figure 4.5: Functional unit consistency check with an SEU in the operand register.](image)

In this work, consistency checks are used in two novel ways. First, they are used to not merely detect upsets in registers, but to detect upsets in the logic and routing going to or from functional units. Additionally, consistency checks are used in a novel way to detect upsets in the logic and routing leading to and from the main memory and caches. Upsets in the logic or routing leading to functional units are detected in the same way as depicted in Figure 4.5.

To detect upsets in the logic and routing leading to the main memory or caches, memory read and write tests are periodically performed. A memory read check reads a known value from a specific memory location and compares the value read to the expected value. For example, Figure 4.6 shows how an upset to an address signal is detected during a scheduled memory read test. The upset causes the wrong address to be read – causing the wrong value to be read. The upset is detected when the expected value is compared to the value read. A memory write test writes a known value to a known location. That location is then read, and the value that was read is compared to the value that was originally written. Whereas a memory read test tests a memory’s address and data output signals, a memory write test tests the memory’s data input, output and write enable signals.

Consistency checks are used in this work with the goal of detecting upsets in any and every ACE configuration memory bit corresponding to the user memory elements and functional units in the LEON3 processor. Consistency checks should detect upsets in routing leading to and from
the BRAMs corresponding to the main memory, instruction and data caches, and the register file. Upsets to BRAM attributes should also be detected. Consistency checks should also detect upsets to the hardware multiplier/divider unit and the ALU unit and control surrounding the unit. Upsets should be detected in the FPGA primitives that makeup these functional units (DSPs and slices), as well as upsets in the logic leading to and from the primitives.

To detect any ACE configuration memory bit upset, consistency checks must be able to detect both stuck-at "0" and stuck-at "1" values in the logic and routing. To detect both of these kinds of upsets, all memory and functional unit logic and routing must be exercised with both ones and zeros on every signal. For example, consider the main memory address line. To detect all stuck-at "0" and stuck-at "1" upsets, two memory reads are required. If the first memory read reads address zero (the first memory location), the second memory read test reads the last memory location (the complement of the first memory address). This ensures that every bit in the address bus has had both a zero and a one driving it. Similar to this example, every memory test and every functional unit test must be done at least twice to ensure that every signal has been exercised with both a one and a zero. This section discusses how this is done with memory tests, and functional unit tests.

### 4.2.1 Functional Unit Tests

The two functional units that are tested include the hardware multiplier/divider, and the LEON3 integer ALU. The integer ALU has four different categories of functions: step multiply (MULS), add operations (ADD, SUB), logic operations (OR, ORN, AND, ANDN, XOR, XNOR), and shift operations (SLL, SRL, SRA). The multiplier/divider also has four categories of opera-
...tions: unsigned multiply (UMUL), signed multiply (SMUL), unsigned divide (UDIV), and signed divide (SDIV). Thus, between the two functional units there are eight types of consistency checks that explicitly exercise these units. Most of these tests are performed twice, so that the inputs and outputs are exercised as much as possible.

Table 4.4: Functional unit tests run during a consistency check.

<table>
<thead>
<tr>
<th>Functional Unit</th>
<th>Instruction</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Expected Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>MUL/DIV</td>
<td>UMUL</td>
<td>0xFFFF FFFE</td>
<td>0x8000 0000</td>
<td>0x7FFF FFFF 0000 0000</td>
</tr>
<tr>
<td>MUL/DIV</td>
<td>UMUL</td>
<td>0x0000 0001</td>
<td>0xFFFF FFFF</td>
<td>0x0000 0000 FFFF FFFF</td>
</tr>
<tr>
<td>MUL/DIV</td>
<td>SMUL</td>
<td>0xFFFF FFFE</td>
<td>0x8000 0000</td>
<td>0x7FFF FFFF 0000 0000</td>
</tr>
<tr>
<td>MUL/DIV</td>
<td>SMUL</td>
<td>0x0000 0001</td>
<td>0xFFFF FFFF</td>
<td>0x0000 0001 FFFF FFFF</td>
</tr>
<tr>
<td>MUL/DIV</td>
<td>UDIV</td>
<td>0x0000 0000</td>
<td>0x0100 0000</td>
<td>0x0000 089A</td>
</tr>
<tr>
<td>MUL/DIV</td>
<td>SDIV</td>
<td>0xFFFF FFFF 6954 3210</td>
<td>0xFFFF FFFF</td>
<td>0x7FFF FFFE</td>
</tr>
<tr>
<td>ALU Mult</td>
<td>MULS</td>
<td>0x7FFF FFFF</td>
<td>0xFFFF FFFF</td>
<td>0x7FFF FFFE</td>
</tr>
<tr>
<td>ALU Mult</td>
<td>MULS</td>
<td>0x0000 0001</td>
<td>0xFFFF FFFE</td>
<td>0x0000 0001</td>
</tr>
<tr>
<td>ALU Logic</td>
<td>OR</td>
<td>0x0100 0000</td>
<td>0xFFFF FFFF</td>
<td>0xFFFF FFFF</td>
</tr>
<tr>
<td>ALU Logic</td>
<td>XNOR</td>
<td>0x0100 0000</td>
<td>0xFFFF FFFF</td>
<td>0xFFFF FFFF</td>
</tr>
<tr>
<td>ALU Add</td>
<td>ADD</td>
<td>0xFFFF FFFF</td>
<td>0xFFFF FFFF</td>
<td>0xFDFF FFFE</td>
</tr>
<tr>
<td>ALU Add</td>
<td>ADD</td>
<td>0x0100 0000</td>
<td>0x0100 0000</td>
<td>0x0200 0000</td>
</tr>
<tr>
<td>ALU Shift</td>
<td>SRA</td>
<td>0x8765 4321</td>
<td>0x0000 0009</td>
<td>0xFFC3 B2A1</td>
</tr>
<tr>
<td>ALU Shift</td>
<td>SLL</td>
<td>0x8765 4321</td>
<td>0x0000 0009</td>
<td>0x8765 4321</td>
</tr>
</tbody>
</table>

Table 4.4 shows the functional unit tests that are performed during a consistency check. The table shows that many of the functional tests are performed twice with complementary operand values. For example, the first two rows of the table show the unsigned multiply tests. The test is performed twice so that the multiplier inputs and outputs are tested with both ones and zeros in every bit position (except for the MSBs of operand 1, and the product). Even though not every ALU operation is shown in Table 4.4, the other ALU instructions are implicitly tested throughout the consistency checks. These instructions include: SUB (subtract), ORN (negated OR), AND, ANDN (negated AND), XOR, and SRL (shift right logic). Thus, every type of ALU and every type of multiply/divide instruction is tested.
4.2.2 Memory Tests

A unique feature of this work is its application of consistency checks to detect upsets in the logic and routing coming to and from memory elements. The memory elements included in this detection coverage are the main memory, both caches, the register file, and some pipeline control registers. Similar to functional unit tests, most of the memory consistency checks that detect upsets in these memory units are performed twice to detect both stuck-at "0" and stuck-at "1" errors.

Table 4.5 summarizes the explicit consistency checks that are performed to detect upsets in memories. Most of the table lists main memory and data cache read and write tests. A memory write test writes a known value (values in the Data column of Table 4.5) to a known location (addresses in the Address column of Table 4.5). That memory location is then read, and the value that was read is compared to the value that was originally written. Whereas a memory read test tests a memory’s address and data output signals, a memory write test also tests the memory’s data input and write enable signals.

<table>
<thead>
<tr>
<th>Memory Unit</th>
<th>Instructions</th>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCache</td>
<td>ST, LD</td>
<td>0x0000 0000</td>
<td>0x0100 0000</td>
</tr>
<tr>
<td>DCache</td>
<td>ST, LD</td>
<td>0xFFFF FFFF</td>
<td>0xFEFF FFFF</td>
</tr>
<tr>
<td>Main Mem</td>
<td>ST, LD</td>
<td>0x0000 0000</td>
<td>0x0100 0000</td>
</tr>
<tr>
<td>Main Mem</td>
<td>ST, LD</td>
<td>0xFFFF FFFF</td>
<td>0xFEFF FFFF</td>
</tr>
<tr>
<td>Main Mem &amp; DCache</td>
<td>STD, LDD</td>
<td>0x0000 0000 &amp;</td>
<td>0x0100 0000 8765 4321</td>
</tr>
<tr>
<td>Main Mem &amp; DCache</td>
<td>STD, LDD</td>
<td>0xFFFF FFFF</td>
<td>0xFFFF FFFF</td>
</tr>
<tr>
<td>Main Mem &amp; DCache</td>
<td>STB, LDUB, LDSTUB, SWAP</td>
<td>0xFFFF FFFD</td>
<td>0x21</td>
</tr>
<tr>
<td>Main Mem &amp; DCache</td>
<td>STB, LDUB, LDSTUB, SWAP</td>
<td>0xFFFF FFFE</td>
<td>0xEF</td>
</tr>
<tr>
<td>Main Mem &amp; DCache</td>
<td>STH, LDH</td>
<td>0xFFFF FFFA</td>
<td>0xCDEF</td>
</tr>
<tr>
<td>Main Mem &amp; DCache</td>
<td>STH, LDH</td>
<td>0xFFFF FFFC</td>
<td>0x3210</td>
</tr>
<tr>
<td>Instruction Register</td>
<td>SETHI, OR</td>
<td>N/A</td>
<td>0xFFFF FFFF</td>
</tr>
<tr>
<td>Instruction Register</td>
<td>SETHI, AND</td>
<td>N/A</td>
<td>0x0000 0000</td>
</tr>
<tr>
<td>ICC Register</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Forwarding Registers</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Table 4.5 shows that every signal in the data cache and main memory’s address bus against both stuck-at "0" and stuck-at "1" upsets. These upsets are detected by performing memory read
and write test at address 0x00000000 and at address 0xFFFFFFFF. Address 0x00000000 is the first entry in the processor’s trap vector table. The instruction at that address is a no operation (NOP) instruction (0x01000000). Rewriting a NOP at that address will not change the trap vector table. The complement of address 0x00000000 is address 0xFFFFFFFF, which is the last instruction in memory. To perform the second memory consistency check, the complement of a NOP instruction (0xFEFFFFFF) is written to, and then read from the last location in main memory. To ensure that these tests will not overwrite important data at address 0xFFFF FFFF, the program’s stack pointer is initialized to begin at address 0xFFFF FFC0 instead of address 0xFFFF FFFF. Modifying the stack pointer this way prevents the program from using the last 16 main memory locations. The same couplet of read and write tests are done in the data cache.

The last section of Table 4.5 lists control registers that are explicitly tested. These control registers include the instruction register, integer condition codes (ICC) register, and forwarding registers. The forwarding registers refer to the ALU result that gets pushed down the pipeline on the way to the write-back stage where it is written to the register file. If the result is needed before it reaches the register file, it can be accessed from a forwarding register in one of the execute, memory, exception, or write-back pipeline stages.

Table 4.5 shows that the integer condition codes (ICC) tested. Since the condition codes are important for conditional branches, detecting upsets in this register is important. The condition codes consist of four bits: N, Z, V, and C. When bit N is asserted the last signed instruction that modified the condition codes produced a negative result. When Z bit is asserted, the result of the last instruction that modified the condition codes was zero. The V bit is an overflow bit that indicates if the last condition code modifying instruction produced an overflow. Finally, when the C bit is set, it indicates that a carry-out occurred in the last instruction that modified the condition codes. Consistency checks detect upsets to the conditional codes by performing instructions that explicitly set them, and then they are read to test their correctness. Since there are four condition code bits, eight tests are performed to test each of them. Four tests test set each of them to zero, and four tests set each of them to one.

Table 4.5 shows that in the process of performing the memory-related consistency checks, all of the SPARC V8 load and store instructions are used. Many of them are tested twice with complementary address and data values. The load instructions tested include: LD (load normal
32-bit word), LDUB (load unsigned byte), LDSB (load signed byte), LDUH (load unsigned half word), LDSH (load signed half word), LDD (load double word), and LDSTUB (atomic load-store unsigned byte). The store instructions tested include: ST (store normal 32-bit word), STB (store byte), STH (store half word), STD (store double word), and SWAP (swap register value with main memory value).

4.2.3 Costs

To add consistency checks to a program, the assembler adds the consistency check interrupt service routine (ISR) to the program code. The ISR requires 435 cycles to execute the 221 instructions which perform 39 functional unit and memory consistency checks. The expected values of memory and functional unit consistency checks are pre-saved in a reserved register file window. When a program begins, before executing the program code, the consistency check expected values are first stored in a reserved register file window.

The main cost associated with consistency checks is performance. The actual performance cost of this software fault-tolerance technique is proportional to how frequently this ISR is executed. How often to run consistency checks is a complex issue. Appendices D and E address the issue of how often to run consistency checks from both probabilistic and empirical points of view. The study in Appendix E shows that an acceptable performance-reliability trade-off is reached when three consistency checks are performed per checkpoint interval. Table 4.6 shows the performance costs ($\rho$) for each of the testbench programs when three consistency check ISRs are run per checkpoint interval, with a checkpoint interval of 18,000 cycles (as decided in Chapter 3). The cost is expressed in terms of how many more cycles are required to run a program on a LEON3 processor protected with consistency checks and checkpointing compared to a program on an unmitigated LEON3 processor. On average only 14% more time is required to run a program on a LEON3 processor protected with consistency checks and rollback checkpointing.

Similar to control-flow monitoring, there is a very small area cost associated with consistency checks. The area costs shown in Table 4.7 compare the costs of an unmitigated LEON3 processor to a LEON3 processor with consistency check upset detection implemented. The value in parenthesis indicates the amount of additional area required compared to the unmitigated processor. The table shows that, like control-flow monitoring, consistency checks require 1% more
Table 4.6: Performance cost of consistency checks.

<table>
<thead>
<tr>
<th>Program</th>
<th>Performance Cost ($\rho$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fhourcorners</td>
<td>1.11×</td>
</tr>
<tr>
<td>Hanoi</td>
<td>1.13×</td>
</tr>
<tr>
<td>MatrixMult</td>
<td>1.14×</td>
</tr>
<tr>
<td>ToUpper</td>
<td>1.14×</td>
</tr>
<tr>
<td>Average</td>
<td>1.13×</td>
</tr>
</tbody>
</table>

area. This additional area is required by control signals in the pipeline which monitor which of the software detection techniques implemented in this study detects an SEU.

Table 4.7: Area costs of a LEON3 processor protected with consistency checks and rollback checkpointing.

<table>
<thead>
<tr>
<th>Consistency Checks Area Costs</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEON3 Design</td>
</tr>
<tr>
<td>---------------</td>
</tr>
<tr>
<td>Unmitigated</td>
</tr>
<tr>
<td>Consistency Checks</td>
</tr>
</tbody>
</table>

4.2.4 Undetected Upsets

The goal of consistency checks is to detect upsets in all the logic and routing surrounding memories and functional units. Due to the complexities of the processor pipeline routing, it can be difficult to detect upsets in all of these signals and logic elements. For example, careful planning is required to test the logic and routing of each of the forwarding registers. In general, an effective set of consistency checks is processor dependent, and complete coverage requires detailed knowledge of the pipeline architecture.

The set of consistency checks used in this work is not meant to guarantee complete coverage on the logic and routing leading to the memories and functional units of the LEON3 processor.
It is meant to show that the majority of the logic and routing can be covered without extensive architectural details. Greater protection also incurs a greater performance penalty.

4.3 Software Indicators

High-level observations of program behavior are often used to detect upsets [37, 38]. Instead of directly detecting a fault, symptoms of a fault are observed. These high-level observations are called software indicators. A common example of a software indicator is detection of an exception during program execution. Software indicators tend to have a high detection latency since it often takes a while for the effects of an upset to be observed in software. However they can be implemented for very little or no cost. Thus, they are useful for detecting upsets that other detection techniques do not catch. The two software indicators used in this work include program hangs, and illegal traps.

4.3.1 Program Hangs

The first indicator detects when a program appears to be hung. A common way to detect when a program is hung is for the processor to use a heartbeat-like signal. A watchdog timer is used to detect when the heartbeat signal has gone dead. Each time the heartbeat signal goes high, it resets the watchdog timer. When the heartbeat signal goes dead for a pre-determined amount of time, the watchdog timer elapses, a hang is detected, and a rollback is required.

The heartbeat signal used in the LEON3 processor protected with software indicators is the valid output signal. As shown in Chapter 2, this requires small modifications to the C code. When the valid signal goes high (i.e. when a \texttt{tn 0} instruction is executed), it resets the watchdog timer in the interrupt controller. Since this valid output signal is not necessarily regular, the maximum watchdog timer value is programmable. If the watchdog timer reaches its maximum value, a hang is detected and a rollback is performed.

4.3.2 Illegal Traps

An easily observed fault symptom is an illegal trap. The SPARCV8 architecture includes a trap table for 256 traps. Under normal program execution, none of the unused traps should ever
be executed. If an upset causes one of these traps to be executed, a rollback should be signaled. A call to the rollback ISR can also be used as a part of the trap routine for the traps that are used.

### 4.3.3 Costs

Software indicators come with almost no cost. There is no performance cost, but there is a very small area cost for the watchdog timer, and the control signals associated with the timer. Table 4.8 compares the area cost of an unmitigated LEON3 processor to a LEON3 processor with the two software indicators mentioned. The value in parentheses shows the amount of additional area required compared to an unmitigated LEON3. The table shows that software indicators can be implemented for about 1% additional area. This area includes the watchdog timer as well as the control logic which monitors which software detection techniques detect an SEU.

<table>
<thead>
<tr>
<th>Software Indicator Area Costs</th>
<th>Slices</th>
<th>BRAMs</th>
<th>DSPs</th>
<th>Config Bits (CFGbits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEON3 Design</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unmitigated</td>
<td>3058 (1.00×)</td>
<td>12 (1.00×)</td>
<td>4 (1.00×)</td>
<td>677,065 (1.00×)</td>
</tr>
<tr>
<td>Software Indicators</td>
<td>3098 (1.01×)</td>
<td>12 (1.00×)</td>
<td>4 (1.00×)</td>
<td>682,964 (1.01×)</td>
</tr>
</tbody>
</table>

### 4.3.4 Undetected Upsets

Software indicators are the only detection technique discussed in this chapter that explicitly protect the interrupt controller. But other than that processor unit, it is difficult to identify specific processor units that this detection technique protects. This detection technique is meant to support other detection techniques rather than act as a stand-alone technique. It detects upsets that fall through the coverage nets cast by the other techniques. For example, it detects control-flow upsets which cause a processor to stride through instruction memory past the end of a program and into data memory.
4.4 BRAM Parity and Memory Scrubbing

Although this study focuses on protecting softcore processors by protecting the configuration memory, the user memory must also be protected. The software mitigation techniques discussed in this section protect smaller user memory elements such as registers, LUT-based RAMs, and SRLs. The larger memory elements are protected with simple 4-bit odd parity and memory scrubbing. Parity can detect any single-bit upset, many multi-bit upsets, as well as an all-ones or all-zeros upset.

4.4.1 BRAM Parity

Parity detects upsets in the contents of any memory implemented using a BRAM. These memory units include the main memory, register file, and caches, and the cache tags. Parity is checked on the data inputs of all of these memories. Checking the inputs is important in order to prevent corrupt data from being written to memory. In addition to checking the inputs to these memories, the parity of the main pipeline result is checked, and the parity of the data inputs and outputs of the cache controllers are checked. The memory outputs are checked using memory scrubbing. When a parity error is found, a rollback is signaled.

4.4.2 Memory Scrubbing

In addition to protecting data before it is written to memory, it is also important to protect against SEUs that can happen while data sits in the memory. Appendix B shows that BRAM scrubbing is an essential part of full BRAM protection. That study shows that a fully redundant memory is required in order to scrub out upsets when they are found. The checkpointed memory serves as the duplicate memory for the main memory and register file. A scrubbing counter walks through every memory location checking for upsets by checking the parity of each data word. When an upset is detected, checkpointing takes over to correct the upset with a rollback. Since the LEON3 processor uses write-through caches, it is unnecessary to have a duplicate cache. Instead, when upsets in the cache are found, the cache is invalidated and a rollback is performed.
4.4.3 Costs

Table 4.9 reports the area usage for parity and memory scrubbing. The area of the unmitigated LEON3 and the LEON3 with rollback checkpointing are also reported in order to show the relative resource usage of the LEON3 protected with parity, memory scrubbing, and rollback checkpointing. The table shows that parity and scrubbing require an additional 8% of hardware.

<table>
<thead>
<tr>
<th>LEON3 Design</th>
<th>Slices</th>
<th>BRAMs</th>
<th>DSPs</th>
<th>Config Bits (CFGbits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unmitigated</td>
<td>3058 (1.00×)</td>
<td>12 (1.00×)</td>
<td>4 (1.00×)</td>
<td>677,065 (1.00×)</td>
</tr>
<tr>
<td>Parity, Scrub</td>
<td>3327 (1.09×)</td>
<td>12 (1.00×)</td>
<td>4 (1.00×)</td>
<td>730,403 (1.08×)</td>
</tr>
</tbody>
</table>

4.4.4 Undetected Upsets

The purpose of the previously mentioned techniques is to detect upsets in the logic and routing of the processor. The primary purpose of this detection technique is to protect the user memory bits. Under a single event upset model, parity and memory scrubbing protect against all single-bit upsets in the user memory. The coverage provided by these techniques also spills over to provide protection to some of the logic and routing of the processor.

4.5 Undetected Upsets

Each of the individual detection techniques have a specific fault coverage, but overall when combined together, there are two types of SEUs that are undetected by the collection of detection techniques. The first and largest class of undetected upsets comes from single points of failure on the processor inputs and outputs. The processor inputs include the clock and reset lines. The processor output is the output coming from the processor pipeline. Between the inputs and outputs, single points failure on the output are much more numerous. More than 50% of undetected SEUs
occur on the processor output. One method for detecting upsets on the output would be to encode the processor output and perform off-chip decoding.

The second class of undetected upsets are upsets to the cache controllers. The logic and routing of the cache controller state machines have no explicit detection technique protecting them. Although a large percentage of cache controller upsets are detected, upsets to the cache controllers make up about 15% of undetected SEUs.

4.6 Summary

The software fault-tolerance techniques used in this work to detect upsets the LEON3 processor include a modified version of SWIFT (which includes code duplication and control-flow monitoring), consistency checks, and software indicators. Parity and memory scrubbing are also used to detect upsets in the LEON3 user memory. Checkpointing is the software fault-tolerance technique used to recover when upsets are detected. Software fault-tolerance can be used to detect upsets in the configuration memory of a softcore processor for a much lower area cost than traditional techniques. Instead of spatial costs, software fault-tolerance usually comes at the cost of time.

The area costs of a LEON3 softcore processor protected with each of the upset detection techniques introduced in this chapter are shown in Table 4.10. The area costs of an unmitigated LEON3 processor and a LEON3 processor with checkpointing implemented are also included in the table. When any of the detection techniques discussed in this chapter are used to detect upsets in a softcore processor, checkpointing must also be implemented to recover from the detected upset. The table shows that the largest area cost for software fault-tolerance comes from the addition of checkpointing.

Only two of the detection techniques introduced in this chapter have performance costs associated with them. The performance costs of control-flow monitoring and consistency checks are shown in Table 4.11. The table shows that control-flow monitoring incurs the largest performance cost. The performance cost of consistency checks is proportional to how frequently they are run. How often to perform consistency checks is the subject of Appendices D and E.

Combining individual upset detection techniques with checkpointing is the subject of Chapter 6. In addition to investigating the reliability of individual detection techniques, different combi-
Table 4.10: Area costs of a LEON3 softcore processor protected with various fault-tolerance techniques.

<table>
<thead>
<tr>
<th>LEON3 Design</th>
<th>Slices</th>
<th>BRAMs</th>
<th>DSPs</th>
<th>Config Bits (CFGbits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unmitigated</td>
<td>3058 (1.00×)</td>
<td>12 (1.00×)</td>
<td>4 (1.00×)</td>
<td>677,065 (1.00×)</td>
</tr>
<tr>
<td>Checkpoint only</td>
<td>4386 (1.43×)</td>
<td>21 (1.75×)</td>
<td>4 (1.00×)</td>
<td>894,355 (1.32×)</td>
</tr>
<tr>
<td>CDWC Register File</td>
<td>3332 (1.09×)</td>
<td>14 (1.17×)</td>
<td>4 (1.00×)</td>
<td>746,006 (1.10×)</td>
</tr>
<tr>
<td>Control-Flow</td>
<td>3062 (1.00×)</td>
<td>12 (1.00×)</td>
<td>4 (1.00×)</td>
<td>681,940 (1.01×)</td>
</tr>
<tr>
<td>Consistency Checks</td>
<td>3062 (1.00×)</td>
<td>12 (1.00×)</td>
<td>4 (1.00×)</td>
<td>681,940 (1.01×)</td>
</tr>
<tr>
<td>Software Indicators</td>
<td>3098 (1.01×)</td>
<td>12 (1.00×)</td>
<td>4 (1.00×)</td>
<td>682,964 (1.01×)</td>
</tr>
<tr>
<td>Parity, Scrub</td>
<td>3327 (1.09×)</td>
<td>12 (1.00×)</td>
<td>4 (1.00×)</td>
<td>730,403 (1.08×)</td>
</tr>
</tbody>
</table>

Table 4.11: Performance cost of software fault-tolerance.

<table>
<thead>
<tr>
<th>Program</th>
<th>Control-Flow Monitoring</th>
<th>Consistency Checks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fhourcorners</td>
<td>1.31×</td>
<td>1.11×</td>
</tr>
<tr>
<td>Hanoi</td>
<td>1.52×</td>
<td>1.13×</td>
</tr>
<tr>
<td>MatrixMult</td>
<td>1.58×</td>
<td>1.14×</td>
</tr>
<tr>
<td>ToUpper</td>
<td>1.60×</td>
<td>1.14×</td>
</tr>
<tr>
<td>Average</td>
<td>1.50×</td>
<td>1.13×</td>
</tr>
</tbody>
</table>

Different combinations of detection techniques have different associated costs. Weighing reliability with cost is also the subject of Chapter 6. The metrics and hardware fault-injection methodology used to evaluate this trade-off are described in the next chapter.
CHAPTER 5. RELIABILITY METRICS AND MODELS FOR SOFTCORE PROCESSORS

Metrics are needed to measure the benefits of the fault-tolerance techniques established in the previous chapters, and accurate fault-injection models are required to accurately use these metrics. The traditional way of measuring reliability for FPGA designs is to measure the number of sensitive bits in the configuration bitstream [22, 39, 83], while the traditional way of measuring processor reliability is with the mean instruction to failure (MITF) [42]. Neither of these metrics provides enough information for the reliability of softcore processors or to explore the softcore processor design-space. This chapter introduces two new reliability metrics for softcore processors: mean useful instruction to failure (MuITF), and reliability-area-performance (RAP).

The traditional way of performing hardware fault-injection in an FPGA design compares the design under test (DUT) with a golden version of the design, on a cycle-by-cycle basis [25, 40, 83]. The traditional way of performing fault-injection in an ASIC-based processors is to upset only the bits in register file and pipeline registers [84–86]. Neither of these hardware fault-injection models is accurate for softcore processors. This chapter introduces a more effective hardware fault-injection model for softcore processors.

5.1 Reliability Metrics for Softcore Processors

Traditionally, FPGA design reliability is measured by classifying each of the bits in the configuration memory as being either sensitive or insensitive [22, 39, 83]. Traditional processor reliability is measured in terms of mean time to failure (MTTF) or, alternatively, in terms of mean instructions to failure (MITF) [42]. This section combines both methods to produce a new way of calculating MITF, and introduces a novel metric called reliability-area-performance (RAP). The metrics and equations developed in this section are used to characterize the reliability of softcore
processors and softcore processor components. These metrics are used to evaluate the reliability-cost trade-off of mitigated softcore processors, and to explore the processor design-space.

### 5.1.1 Architectural Vulnerability Factor (AVF)

The traditional metric for expressing how likely it is for an upset to lead to a processor error is called the architectural vulnerability factor (AVF) [87]. For ASIC-based processors, the AVF of a single-bit storage cell is equal to the percentage of time that that cell holds a bit that is required for architecturally correct execution (ACE). In softcore processors, however, an upset in the configuration memory cell can cause lasting effects to routing and logic. Configuration upsets remain on the device for many clock cycles until repaired with configuration scrubbing. Because of the relatively long upset time of a configuration bit, each configuration bit is given an AVF of either 100% (an ACE bit), or 0% (upsetting the configuration bit does not affect processor output). This “all-or-nothing” assumption is conservative, but it is used in this work to provide worst-case measurements.

Each bit in the configuration memory\(^1\) is classified as being either required for architecturally correct execution (ACE) or unnecessary for architecturally correct execution (unACE) [42, 87]. Upsetting an ACE bit causes a program running on the softcore processor to produce an incorrect output, while upsetting an unACE bit does not hinder correct program execution. The terms ACE bits and sensitive bits are used synonymously. Likewise, unACE is a synonym for insensitive.

Upsets to ACE bits are classified as being either detected recoverable upsets (DRU), detected unrecoverable errors (DUE), or silent data corruption (SDC) bits. DRUs are upsets that are detectable and which can be prevented from causing erroneous output. DUE upsets are those which are detected by the processor, but which cause errors even after recovery attempts. SDC upsets are upsets which are never detected, but which cause erroneous output. In order to have a reliable processor, there should be as few SDCs and DUEs as possible, but SDCs are especially bad since they are not even detected. DRU bits are only considered when a mitigation technique is applied to a processor. Detection techniques attempt to reduce the number DUEs and SDCs and increase the number of DRUs.

\(^1\)There are 21 million configuration bits in the Xilinx Virtex4 FX60 FPGA used in this study.
A special class of DUEs include **dloops**, a detected upset that is unsuccessfully repaired and continually re-detected as discussed in Chapter 3. Although a softcore processor cannot detect when an SDC occurs, nor can it distinguish a DUE from a DRU, a softcore processor protected with upset detection techniques can detect when a dloop occurs. Often, the only way to repair a softcore processor when a dloop occurs is to reconfigure the FPGA holding the processor. When a reconfiguration is required, the program must restart from the beginning.

A processor’s sensitivity to SDCs and DUEs is measured using AVF. The AVF of a processor (or processor component) is the percentage of the processor area or processor component area that, when upset, causes DUEs or SDCs [87]. Equation 5.1 shows how AVF is computed for softcore processors with the ”all-or-nothing” assumption. AVF is the fraction of the total number of configuration bits (CFGbits) used by the softcore processor that are SDC or DUE bits.

\[
\text{AVF} = \frac{\# \text{SDCs} + \# \text{DUEs}}{\text{CFGbits}}. \tag{5.1}
\]

AVF measurements are sensitive to the timing of an upset. If a piece of data has a very short lifetime, or if a processor element is rarely used during the execution of the program, then an upset to that data or processor unit may be likely to be classified as unACE. For example, if a program uses the hardware multiplier once at the beginning of the program, upsets that occur in the multiplier may not be detected if they occur after the multiplier is used. For this reason, some studies that use AVF assign a percentage value to a bit indicating how likely it is for the bit to be classified as an ACE bit [42]. This kind of modeling is especially important when AVF measures transients rather than SEUs. This work assumes worst case behavior by classifying a configuration bit as an ACE bit 100% of the time if it can ever be classified as an ACE bit. To detect this worst case behavior, a program is executed from start to finish after an upset is inserted (see the next section).

AVF as shown in Equation 5.1 is a worst-case design sensitivity measurement. Some applications might be more interested in detection than both detection and recovery. When upsets are detected, the application resets and restarts. In those cases, a more accurate AVF includes only the SDC bits in the configuration memory (AVF_{SDC}). AVF_{SDC} (Equation 5.2) is an indication of how
sensitive a processor or processor component is to SDCs:

\[
AVF_{SDC} = \frac{\# \text{ SDCs}}{\text{CFGbits}}.
\]  

(5.2)

In order to tell how successful a given recovery technique is, the vulnerability of a softcore processor should be reported only in terms of DUE bits (AVF\textsubscript{DUE}). AVF\textsubscript{DUE} (Equation 5.3) is an indication of how sensitive a processor or component is to DUEs:

\[
AVF_{DUE} = \frac{\# \text{ DUEs}}{\text{CFGbits}}.
\]  

(5.3)

As reliability techniques are applied to a softcore processor, the AVF of the processor goes down. The AVF goes down since SDC and DUE bits turn into DRU bits. But, as reliability techniques are applied, the total number of configuration bits (CFGbits) also goes up. A larger denominator in Equations 5.1 - 5.3 produces a smaller AVF. This means that reliability techniques that incur a larger area cost (such as TMR) produce a smaller AVF. A more accurate reliability metric should take area cost into account.

5.1.2 MTTF

The mean time to failure (MTTF) of a softcore processor is a metric that measures the average time until the processor produces erroneous output. MTTF is equal to the inverse of the error rate (MTTF = 1/\(\lambda_{err}\)). The error rate (\(\lambda_{err}\)) of the processor refers to the rate at which a softcore processor produces erroneous output. The error rate is different than the upset rate (\(\lambda_{upset}\)). The upset rate refers to the rate at which SEUs strike the softcore processor. A fraction of those SEUs will cause errors (a fraction of the upsets are SDCs or DUEs). The error rate is equal to the upset rate multiplied by the softcore processor’s AVF (\(\lambda_{err} = \lambda_{upset} \cdot \text{AVF}\)), thus MTTF is calculated as follows:

\[
\text{MTTF} = \frac{1}{\lambda_{upset}} \cdot \frac{1}{\text{AVF}}.
\]  

(5.4)

The bit upset rate (\(\lambda_{bit}\)) refers to the rate at which any given configuration bit is upset. The upset rate (\(\lambda_{upset}\)) of a softcore processor is equal to the bit upset rate multiplied by the number of
configuration bits used by the component: $\lambda_{\text{upset}} = \lambda_{\text{bit}} \cdot \text{CFGbits}$. The value of $\lambda_{\text{bit}}$ depends upon the spacecraft orbit, space environment, and device properties. For this work, $\lambda_{\text{bit}}$ is estimated using a geosynchronous Earth orbit (GEO) at solar minimum. The estimated upset rate for a Xilinx Virtex4 FX60 device in this environment is 4.03 upsets/device-day [88,89] or 2.78E-12 upsets/bits [90]. When using $\lambda_{\text{bit}}$ to calculate MTTF, this work assumes that only one upset occurs at a time. It is assumed that multi-bit upsets do not occur [73]. With these assumptions, the MTTF of a softcore processor is:

$$
\text{MTTF} = \frac{1}{\lambda_{\text{bit}}} \cdot \frac{1}{\text{CFGbits}} \cdot \frac{1}{\text{AVF}} = \frac{1}{\lambda_{\text{bit}}} \cdot \frac{1}{\# \text{SDCs} + \# \text{DUEs}}.
$$

(5.5)

MTTF$\text{SDC}$ (Equation 5.6) is the mean time until an undetectable upset occurs:

$$
\text{MTTF}_{\text{SDC}} = \frac{1}{\lambda_{\text{bit}}} \cdot \frac{1}{\# \text{SDCs}}.
$$

(5.6)

MTTF$\text{DUE}$ (Equation 5.7) is the mean time until a detectable unrecoverable error occurs.

$$
\text{MTTF}_{\text{DUE}} = \frac{1}{\lambda_{\text{bit}}} \cdot \frac{1}{\# \text{DUEs}}.
$$

(5.7)

These equations provide different meanings of failure in MTTF. For example, there may be cases where detecting upsets is all that is required. In this case, failure means that an upset causes an SDC, and MTTF$\text{SDC}$ should be used instead of MTTF. When evaluating the effectiveness of an upset recovery technique, failure is defined as the number of detected upsets which cause a processor to produce erroneous output. This can be measured by the number of DUEs that occur, thus MTTF$\text{SDC}$ should be used instead of MTTF.

5.1.3 MuITF

The mean number of instructions to failure (MITF) is a reliability measure that is specific to processors [42]. MITF measures the number of instructions that are executed between failures. The definition of failure for MITF is the same as it is for MTTF – the processor produces incorrect
output. MITF is more meaningful for processors than MTTF since it focuses on how much work is performed rather than on how much time passes between failures. Traditional MITF is calculated as shown in Equation 5.8. MITF is equal to MTTF multiplied by the frequency of the processor clock (f) and the number of instructions per cycle (IPC) that are committed.

\[
\text{MITF} = \frac{f}{\lambda_{\text{upset}}} \cdot \frac{\text{IPC}}{\text{AVF}}. \tag{5.8}
\]

This work introduces a new reliability metric for softcore processors which measures the mean number of useful instructions to failure (MuITF). MITF is a useful reliability metric for processors, however, a more accurate metric is required when mitigation techniques are applied to a softcore processor. When mitigation techniques are used to protect a processor, Equation 5.8 must be adjusted to account for the performance cost ($\rho$) of the techniques. The performance cost represents the execution of additional unproductive instructions. Thus, in order to calculate the mean useful instructions to failure (MuITF), MITF must be divided by the performance cost of the mitigated processors.

\[
\text{MuITF} = \frac{f}{\lambda_{\text{bit}}} \cdot \frac{1}{\text{CFGbits}} \cdot \frac{\text{IPC}}{\text{AVF} \cdot \rho} \cdot \frac{\text{IPC}}{\rho} = \frac{f}{\lambda_{\text{bit}}} \cdot \frac{\text{IPC}}{\text{# SDCs + # DUEs}}. \tag{5.9}
\]

When the detection of upsets is important but recovery is not needed, MuITF is expressed in terms of SDCs. In this case, a failure happens when an upset hits an SDC configuration bit. MuITF\text{SDC} is shown in Equation 5.10:

\[
\text{MuITF}_{\text{SDC}} = \frac{\text{frequency}}{\lambda_{\text{bit}}} \cdot \frac{\text{IPC}}{\rho} \cdot \frac{\text{IPC}}{\text{# SDCs}}. \tag{5.10}
\]

MuITF can also be expressed in terms of DUE bits. In this case, failure occurs when an SEU strikes a DUE configuration bit. MuITF\text{DUE} is shown in Equation 5.11:

\[
\text{MuITF}_{\text{DUE}} = \frac{\text{frequency}}{\lambda_{\text{bit}}} \cdot \frac{\text{IPC}}{\rho} \cdot \frac{\text{IPC}}{\text{# DUEs}}. \tag{5.11}
\]
MuITF is an good metric for evaluating the trade-off between reliability and performance for processors protected with mitigation techniques. Some mitigation techniques such as check-pointing, control-flow monitoring, code duplication, or consistency checks, execute redundant instructions or occasionally suspend program execution. Although these diversions from program execution are important, they extend program execution times. MuITF is a metric that can account for this time spent away from the execution of the program.

Although MuITF does a good job of evaluating the reliability-performance trade-off, it does not take area costs into account. Thus, MuITF favors mitigation techniques such as TMR or DWC that come at the cost of area instead of time, even if the area costs are very large. A better reliability metric for softcore processors should include area costs as well as performance costs.

5.1.4 RAP

This work introduces a new reliability metric for softcore processors protected with fault-tolerance techniques. This metric, called reliability-area-performance (RAP), includes both area and performance costs in its evaluation of the reliability-cost trade-off. Equation 5.12 shows how RAP is calculated for a fault-tolerant softcore processor. It is calculated by dividing the relative MTTF improvement over an unprotected processor by the product of the performance cost ($\rho$) and area cost. These costs are also ratios, expressed with respect to an unprotected processor. Although RAP balances reliability, area, and performance, it does not necessarily identify the best mitigation technique. Rather, it is an indication of how well a mitigation technique balances all three trade-offs. Each of the three metrics in Equation 5.12 are given equal weight.

$$\text{RAP} = \frac{\text{MTTF improvement}}{\text{Area cost} \cdot \rho}. \quad (5.12)$$

RAP is similar to the area-time (AT) product metric [91]. AT is used to evaluate attempts to reduce the area and/or runtime of a digital application. Usually a technique that reduces area increases runtime, or vice-versa. The AT metric is an indicator of whether attempts to lower area can be done without an equal or greater increase in cost to latency (or vice-versa). The AT product after an improvement should be greater than or equal to the AT product before the improvement. RAP is similar to AT, but uses performance instead of runtime, and adds reliability to the product.
5.2 A Hardware Fault-Injection Model for Softcore Processors

In order to evaluate the reliability of a softcore processor, hardware fault-injection is used to identify ACE bits in the configuration memory. Traditionally for ASIC-based processors, ACE (and unACE bits) bits only occur in user memories and registers. ACE bits are normally identified by tracking them through the pipeline or with the use of software simulation models [92].

Instead of using software models, or only upsetting registers and user memories, full hardware fault-injection is performed to identify the ACE bits in the configuration memory of a softcore processor. Hardware fault-injection is a well-known way of evaluating the impact of SEUs in the configuration memory of commercial SRAM-based FPGA devices [83]. Fault-injection is performed by upsetting each bit in the FPGA configuration memory, one at a time, while a design runs on the FPGA. For the Xilinx Virtex4 FX60 FPGA used in this study, Table 2.1 shows that almost 21 million upsets and program executions are required to test every bit in the entire configuration memory.

Traditionally when fault-injection is done on FPGA designs, the design under test (DUT) whose configuration memory bits are upset, is compared with a golden copy of the design on a cycle-by-cycle basis (Figure 5.1). If the DUT and golden designs differ on any clock cycle, an error is detected. Although this cycle-by-cycle fault-injection model works well for many FPGA designs, it produces less accurate results for some FPGA designs such as a softcore processor.

![Figure 5.1: Traditional cycle-by-cycle hardware fault-injection methodology.](image)

Cycle-by-cycle comparison is an inaccurate way to verify the output of a softcore processor. As shown in Figure 5.2, an upset in the softcore processor acting as the design under test (DUT) may cause the processor to temporarily misbehave before resuming correct execution. A cycle-by-cycle comparison for softcore processors would lead to a large number of false positives. Softcore
processors need an updated fault-injection model that is more flexible than the traditional cycle-by-cycle comparison model. This section introduces a more accurate methodology for performing hardware fault-injection on a softcore processor. The accuracy of this novel fault-injection model was verified to be within 7% of results obtained through radiation testing (Appendix H).

![Fault-injection model for softcore processors](image)

Figure 5.2: A fault-injection model for softcore processors needs to allow the DUT to temporarily misbehave.

### 5.2.1 Hardware Fault-Injection for Unmitigated Softcore Processors

The novel fault-injection model introduced in this section uses the Xilinx Radiation Test Consortium (XRTC) board (Figure 5.3). The XRTC board contains 3 FPGAs: the configuration monitor (configmon), the functional monitor (funcmon), and the design under test (DUT). The DUT holds the fault-tolerant LEON3 design that is injected with faults. The DUT is a Xilinx Virtex4 FX60 FPGA. The funcmon holds a golden copy of the processor and compares the outputs of the golden and DUT processors. This comparison is not done on a cycle-by-cycle basis, since an upset in the DUT could simply delay a correct value. The configmon is responsible for injecting...
faults into the DUT bitstream, and communicating the results of the fault-injection to a host through a USB bus.

![Block diagram of the XRTC board which is used for hardware fault-injection.](image1.png)

![SEAKR’s XRTC board.](image2.png)

(a) Block diagram of the XRTC board which is used for hardware fault-injection.  
(b) SEAKR’s XRTC board.

Figure 5.3: SEAKR’s XRTC board is used to perform hardware fault-injection.

The first fault-injection issue to address is how to compare the DUT and golden processor outputs. Since the outputs are not compared on a cycle-by-cycle basis, it is important to know when to compare the outputs. Chapter 2 describes modifications that are made to the LEON3 processor which allow for a valid signal to be output from the LEON3 processor, indicating that its output is valid. This signal indicates when to compare processor outputs. Chapter 2 emphasizes that it is the programmer’s responsibility to control when this valid signal is asserted. In the programs run in this work, the valid signal is asserted every 10-20 cycles on average.

The assertion of a valid signal allows the DUT and golden processor to be compared without having to be compared on a cycle-by-cycle basis, but the valid signal alone is insufficient. Figure 5.2 suggests that upsets to the processor on the DUT could cause it to lag behind the golden processor. To allow the processor on the DUT to lag, golden processor valid outputs could be collected in a buffer. However, the size of this buffer might need to be extremely large if the lag becomes too large.
If the program outputs are known a priori (which is the case when hardware fault-injection is done), then the golden processor can be replaced with one or more BRAMs whose contents are initialized with the series of expected program outputs. Thus, as fault-injection proceeds, the output of the processor on the DUT is compared with the test-vector BRAM when the processor outputs a valid signal. If the processor output does not match the test-vector BRAM’s data, the funcmon indicates to the configmon that the configuration bit that was upset is an ACE bit. After performing a comparison, the golden BRAM’s address pointer is incremented in anticipation of the next valid signal from the DUT. Between valid signals, it does not matter what the processor outputs. However, if the valid signal is not sent by the DUT for a predetermined period of time, the funcmon communicates to the configmon that the processor on the DUT is hung.

Having established how a softcore processor’s output is verified, the hardware fault-injection methodology is now described for a processor without fault-tolerance protection. For each of the configuration bits in the DUT FPGA, the fault-injection procedure outlined in Figure 5.4 is followed. During the procedure, a program runs twice to completion for each configuration bit. In the first program execution, the program runs for a random number of clock cycles before the configuration bit is upset. The program is run for a random number of cycles to ensure that the upset isn’t inserted at the same point in the program for every configuration bit. After the upset is inserted, this first program run is allowed to finish. The program is then run a second time, providing ample observation time to determine if the current configuration bit is an ACE or unACE bit. If during this time, the processor output ever differs from the test-vector BRAM output, the current configuration bit is determined to be an ACE bit. But if, by the end of the second program run, the processor output still matches the test-vector BRAM output the configuration bit is determined to be an unACE bit. At the end of the second run, the current configuration bit is repaired and the LEON3 processor is reset. The processor is reset to provide the same initial processor state when each configuration bit is upset.

5.2.2 Hardware Fault-Injection for Fault-Tolerant Softcore Processors

The fault-injection procedure for testing the mitigated designs is different than the procedure followed by the unmitigated design (as shown in Figure 5.4). Figure 5.5 shows the procedure followed when performing hardware fault-injection on a mitigated softcore processor. For each of
The DUT FPGA configuration bits, the program running in the DUT processor runs for the amount of time it takes to run the program four times. The program runs once to ensure that at least one checkpoint is taken. In the next program iteration, the program runs for a random number of clock cycles before the configuration bit is upset. This ensures that the upset isn’t inserted at the same point in the program for every configuration bit. After the upset is inserted, the second program execution finishes. The program then runs a third time, giving the LEON3 ample time to detect the upset. This third run is also what allows for the conservative “all-or-nothing” AVF measurements. If and when the upset is detected, the upset is immediately repaired by the fault-injector. This configuration bit repairing happens immediately when the upset is detected, before the end of the third program execution. If the upset is detected, the DUT attempts to recover from this upset with a checkpoint rollback. This rollback is performed immediately after the configuration bit is repaired. But if the upset goes undetected by the end of the third program execution, it is then repaired. Finally, the program runs for a final time to ensure that any checkpoint recovery attempted by the DUT is successful.

The fault injection methodology introduced in this section works well for softcore processors. Appendix H shows that its results are accurate to within 7% of results obtained with radiation.
testing. This hardware fault-injection model is used in the next chapter to establish the reliability of the unmitigated and fault-tolerant LEON3 softcore designs.

5.2.3 Benchmark Programs

To establish the reliability of the unmitigated and fault-tolerant LEON3 softcore processors, the fault-injection procedures shown in Figures 5.4 and 5.5 are run for every FPGA configuration bit for each of the micro-benchmarks used in this study. This subsection introduces the programs in the benchmark suite and their characterizations in terms of program size, processor utilization, and instruction mix. This characterization is presented for the programs that run on the unprotected LEON3 processor. The fault-tolerance techniques presented in this work make changes to the programs, and the effect that has on the characterization of the program is discussed in Chapters 4 and 6.

The size and length of the benchmarks that can be used are limited by the constraints of the fault injector and by the number of BRAMs available on the FPGA. Since on-chip BRAMs are used to hold the program, the size of the programs are limited, and the programs are run on the processor without an OS. Since running the fault-injection procedure on a softcore processor with a single micro-benchmark program takes up to 60 hours to complete, the length of the programs are limited. The benchmark programs chosen for this study test worst-case behavior of the processor structures and instruction set architecture. Since these benchmarks are meant to be stress tests, it is expected that they will be pessimistic compared to a typical workload.

The benchmark suite includes five programs:

- **Fhourcorners**: a modified version of the Connect Four game.

- **Hanoi**: the classic towers of Hanoi problem with seven disks.

- **MatrixMult**: a kernel that performs a 20x20 matrix multiplication.

- **ToUpper**: a kernel that capitalizes the letters of a string of words.

Each of these benchmarks are characterized in Tables 5.1 – 5.3. Table 5.1 reports the number of register windows each program uses, as well as the percentage of instruction cache, data cache,
and main memory used by each program. The number of register windows used by each benchmark is out of 10, since there are 10 register windows on the processor. The main memory includes both the program code as well as the data. The table shows that collectively, the benchmarks use all of the register windows, all of the data and instruction cache, and over 50% of the main memory. When software fault-tolerance is used to protect a softcore processor, the characterizations shown in Table 5.1 changes slightly. On average, an additional register window is used, the minimal amount of instruction cache used is 75%, the data cache usage goes up by 3% on average, and the amount of main memory used increases by an average of 15%.

Table 5.1: Memory usage report for each of the testbench programs

<table>
<thead>
<tr>
<th>Program</th>
<th># Reg Windows</th>
<th>Icache</th>
<th>Dcache</th>
<th>Main Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fhourcorners</td>
<td>3 (30%)</td>
<td>100%</td>
<td>15%</td>
<td>50%</td>
</tr>
<tr>
<td>Hanoi</td>
<td>8 (80%)</td>
<td>31%</td>
<td>4%</td>
<td>32%</td>
</tr>
<tr>
<td>MatrixMult</td>
<td>2 (20%)</td>
<td>55%</td>
<td>100%</td>
<td>37%</td>
</tr>
<tr>
<td>ToUpper</td>
<td>2 (20%)</td>
<td>25%</td>
<td>4%</td>
<td>28%</td>
</tr>
</tbody>
</table>

Table 5.2 shows the average size of the benchmark programs. The code size column reports the number of memory addresses required for the program code, including the trap table, actual program code blocks, and data. The table also reports the number of code blocks for each program along with the average number of instructions per code block, and the average number of exit points per code block. The last column is significant when control-flow monitoring is applied.

Table 5.3 shows the SPARCv8 instruction mix used by each benchmark program before any fault-tolerance is applied. The instruction mix is simplified and divided into five different categories:

- **load/store**: LDSB, LDSH, LDUB, LDUH, LD, LDD, STB, STH, ST, STD, LDSTUB, SWAP
- **ALU**: AND, ANDN, OR, ORN, XOR, XNOR, SLL, SRL, SRA, ADD, SUB
- **mult/div**: UMUL, SMUL, UDIV, SDIV
Table 5.2: Program code usage report for the benchmark programs.

<table>
<thead>
<tr>
<th>Program</th>
<th>Code Size (# addresses)</th>
<th># Code Blocks</th>
<th>Avg # Inst./Block</th>
<th>Avg # Exit Points / Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fhourcorners</td>
<td>1960</td>
<td>140</td>
<td>11.2</td>
<td>2.14</td>
</tr>
<tr>
<td>Hanoi</td>
<td>1107</td>
<td>4</td>
<td>14.3</td>
<td>1.83</td>
</tr>
<tr>
<td>MatrixMult</td>
<td>1464</td>
<td>14</td>
<td>8.4</td>
<td>1.93</td>
</tr>
<tr>
<td>ToUpper</td>
<td>1097</td>
<td>7</td>
<td>6.0</td>
<td>1.85</td>
</tr>
<tr>
<td>Average</td>
<td>1407</td>
<td>41.3</td>
<td>10.0</td>
<td>1.85</td>
</tr>
</tbody>
</table>

- **branch**: BA, BN, BNE, BE, BG, BLE, BGE, BL, BGU, BLEU, BCC, BCS, BPOS, BNEG, BVC, BVS, CALL, JMPL, RETT

- **other**: SETHI, NOP, SAVE, RESTORE, RD, WR, FLUSH

Table 5.3 shows that collectively, most of the instructions are covered. The ones not covered include some of the minor variations on branching, some of the more specialized loads and stores, and a signed divide. Some software fault-tolerance techniques such as consistency checks change the results of Table 5.3. When consistency checks are applied, every program has a 100% complete instruction mix, since the consistency checks test every instruction in the mix.

Table 5.3: Instruction mix report for each of the testbench programs.

<table>
<thead>
<tr>
<th>Program</th>
<th>load/store (out of 12)</th>
<th>ALU (out of 11)</th>
<th>mul/div (out of 4)</th>
<th>branch (out of 19)</th>
<th>other (out of 7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fhourcorners</td>
<td>6 (50%)</td>
<td>10 (91%)</td>
<td>2 (50%)</td>
<td>10 (53%)</td>
<td>7 (100%)</td>
</tr>
<tr>
<td>Hanoi</td>
<td>2 (17%)</td>
<td>4 (36%)</td>
<td>0 (0%)</td>
<td>4 (21%)</td>
<td>5 (71%)</td>
</tr>
<tr>
<td>MatrixMult</td>
<td>2 (17%)</td>
<td>4 (36%)</td>
<td>1 (25%)</td>
<td>6 (32%)</td>
<td>5 (71%)</td>
</tr>
<tr>
<td>ToUpper</td>
<td>4 (33%)</td>
<td>4 (36%)</td>
<td>0 (0%)</td>
<td>7 (37%)</td>
<td>6 (86%)</td>
</tr>
<tr>
<td>Combined</td>
<td>6 (50%)</td>
<td>11 (100%)</td>
<td>3 (75%)</td>
<td>11 (58%)</td>
<td>7 (100%)</td>
</tr>
</tbody>
</table>

Hardware fault-injection is performed for the unmitigated and protected softcore LEON3 processors with each of the benchmark programs. When performing hardware fault-injection, programs run on the processor without an OS. Each program is written in C and compiled to
assembly code using the SPARC V8 gcc compiler. An assembler then creates the VHDL memory files required to synthesize the FPGA design, and applies the software fault-tolerance, when it is used. The hardware fault-injection results and reliability metrics for each of the LEON3 softcore processor designs are discussed in Chapter 6.

5.3 Summary

Existing reliability metrics for ASIC-based processors and FPGA designs are inaccurate for softcore processors. Mean useful instruction to failure (MuITF) and reliability, area, performance (RAP) are introduced as metrics to evaluate the reliability of a softcore processor. RAP evaluates the reliability-area-performance trade-off for mitigated softcore processor techniques.

Traditional fault-injection models are insufficient for softcore processors. The fault-injection model introduced in this chapter builds on the traditional cycle-by-cycle accurate model, but accounts for the fact that upsets in a softcore processor can cause a temporary departure from being cycle-accurate, while still producing correct results. This chapter also introduces the suite of test-bench programs used in fault-injection throughout this work. The hardware fault-injection methodology and reliability metrics introduced in this chapter are used in the next chapter to establish the reliability of the LEON3 softcore processor.
CHAPTER 6. RELIABILITY OF THE LEON3 SOFTCORE PROCESSOR

Having established the hardware fault-injection model for softcore processors, the traditional and low-cost softcore processor mitigation techniques can now be compared. This chapter uses the hardware fault-injection model previously introduced to measure the number of ACE bits in the unprotected and fault-tolerant LEON3 softcore processors. The ACE bit information gathered from hardware fault-injection is then used to measure reliability using the metrics introduced in Chapter 5. The reliability results show that full TMR provides the best reliability, but software fault-tolerance provides significant reliability at a lower cost.

This chapter begins by establishing the baseline reliability of an unmitigated LEON3 processor, and its processor components. Next, the reliability of the traditional fault-tolerance techniques are compared with the reliability of an unprotected processor. Finally, the reliability of the software fault-tolerant LEON3 is presented. The reliability of the individual software mitigation techniques are discussed before considering combinations of fault-tolerance techniques. Appendix I shows additional insights and observations to the results presented in this chapter.

6.1 Unmitigated LEON3

In this section, the reliability of an unmitigated LEON3 softcore processor is established. Reliability results are presented for the processor as a whole, as well as for each processor component. To measure reliability, the raw number of ACE and unACE bits are found using the hardware fault-injection procedure discussed in Chapter 5. These results are used to calculate the AVF, MTTF, MuITF, and RAP of the LEON3 processor and each of its components.

A decisive set of ACE and unACE configuration bits for the unprotected LEON3 softcore processor is found by performing hardware fault-injection with each of the testbench programs individually. The individual fault-injection results for each of these programs is collected into a single set of ACE and unACE configuration bits by taking the union of the individual results. To
perform the union of the individual tests, a pessimistic approach is taken. If a given configuration bit is labeled as an SDC in any of the individual fault-injection tests, it is counted as an SDC bit in the collected result. In other words, if a configuration could ever be considered as an SDC bit, it is counted as an SDC bit in the collected results. Next, if any configuration bit is labeled as a DUE bit in any of the individual tests (and not labeled as an SDC bit in any of the tests), then that configuration bit is labeled as a DUE bit in the collected results. The remaining configuration bits in the unmitigated tests are considered to be unACE bits.

The collected results of the fault-injection tests with each of the testbench programs is shown in Table 6.1. The results are reported on each of the processor components as well as for the entire LEON3 processor. The processor components are considered individually in order to observe which components are the most sensitive, and which are the most difficult to protect. The mitigated LEON3 processor results discussed in the next few sections consider how well each of the components (and the overall processor) can be protected. Table 6.1 establishes the baseline sensitivity of each processor component.

The unACE column in Table 6.1 reports the number of bits that, when upset, do not affect correct program execution. The number in parenthesis indicates the percentage of configuration
bits for the processor component that are unACE bits. The register file has the lowest percentage of unACE bits. Of the configuration bits corresponding to the register file, 38% are unACE, thus 62% are sensitive to upsets. The values in the parentheses of the DUE and SDC columns report the percentage of ACE bits that are DUE and SDC bits. For example, 98% of the sensitive register file configuration bits are SDCs, and the remaining 2% are DUEs. There is no DRU column in the table since, with an unmitigated LEON3 softcore processor, there is no way to recover from a detected upset, thus the number of DRUs will always be zero.

Table 6.1 shows that some of the upsets are detected by the LEON3 processor (as shown in the DUE column). Although no upset detection techniques have been explicitly applied to the LEON3 at this point, the LEON3 processor pipeline has some built-in error detection. The processor throws an error signal when interrupts occur in an unexpected way. Since illegal instructions are often detected in this way, the table shows that the largest percentage of detected upsets occur in the interrupt controller unit, instruction cache, and main memory.

The ACE bits collected with hardware fault-injection are used to determine the reliability of the LEON3 processor and its components. The metrics introduced in the previous chapter are used to calculate reliability. Table 6.2 lists the AVF, MTTF, and MuITF for the processor and each of its components. To calculate MTTF (Equation 5.4) and MuITF equation (Equation 5.9) the upset rate \( \lambda_{bit} \) must be declared. The upset rate used in this work is the upset rate exhibited by a Xilinx Virtex4 FX60 FPGA in a geosynchronous orbit (GEO) at solar minimum: \( \lambda_{bit} = 2.78E-12 \) upsets/bit-s [88]. The value of two other variables in the MuITF equation must also be defined. The average number of instructions per clock cycle (IPC), required by MuITF (Equation 5.9), is observed to be 0.62 for the programs that run on the LEON3. Also, since the XRTC board clock runs at 33 MHz, the frequency value \( f \) used in Equation 5.9 is 33 MHz.

Table 6.2 shows that over 20% of the processor is sensitive to SEUs, and that less than 2% of upsets to that 20% are detectable. It also shows that without any mitigation, in the GEO orbit at solar minimum, the processor will fail an average of about 14 times a year, or every 26 days. The table shows that the register file has the highest AVF, which means that the register file is the most sensitive LEON3 processor component. Upsets to the register file can quickly propagate to the cache and main memory, or to the processor pipeline and ALU [38]. The MTTF and MuITF columns in Table 6.2 show that the processor pipeline and ALU are the components with the lowest
Table 6.2: Architectural vulnerability factors (AVFs) and relative mean instruction to failure (rMITF) for each LEON3 processor component.

<table>
<thead>
<tr>
<th>Component</th>
<th>AVF Values</th>
<th>MTTF Values (years)</th>
<th>MuITF (\times 10^{14}) (instructions)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AVF_{SDC}</td>
<td>AVF_{DUE}</td>
<td>AVF</td>
</tr>
<tr>
<td>Clock</td>
<td>22.39%</td>
<td>1.55%</td>
<td>23.94%</td>
</tr>
<tr>
<td>Reset</td>
<td>11.70%</td>
<td>0.77%</td>
<td>12.47%</td>
</tr>
<tr>
<td>Top Outputs</td>
<td>3.24%</td>
<td>0.02%</td>
<td>3.26%</td>
</tr>
<tr>
<td>IRQ Controller</td>
<td>6.71%</td>
<td>0.55%</td>
<td>7.27%</td>
</tr>
<tr>
<td>Multi/Div</td>
<td>4.26%</td>
<td>0.14%</td>
<td>4.40%</td>
</tr>
<tr>
<td>Icache</td>
<td>15.08%</td>
<td>3.18%</td>
<td>18.26%</td>
</tr>
<tr>
<td>Dcache</td>
<td>16.40%</td>
<td>0.39%</td>
<td>16.79%</td>
</tr>
<tr>
<td>Main Memory</td>
<td>10.65%</td>
<td>2.63%</td>
<td>13.28%</td>
</tr>
<tr>
<td>Register File</td>
<td>60.77%</td>
<td>1.04%</td>
<td>61.82%</td>
</tr>
<tr>
<td>ALU</td>
<td>30.55%</td>
<td>0.77%</td>
<td>31.32%</td>
</tr>
<tr>
<td>Pipeline</td>
<td>19.83%</td>
<td>1.71%</td>
<td>21.54%</td>
</tr>
<tr>
<td>Full LEON3</td>
<td>18.85%</td>
<td>1.72%</td>
<td>20.57%</td>
</tr>
</tbody>
</table>

overall reliability. A large MuITF often indicates that the processor component has a small area relative to the processor (such as the interrupt controller). The register file is the most vulnerable component (largest AVF), but due to its small relative size, it does not have the smallest MuITF.

### 6.2 Reliability of Hardware-Based Softcore Processor Mitigation

To reduce the vulnerability of the unmitigated LEON3 softcore processor, traditional mitigation techniques are applied. In this section, hardware fault-injection results are presented for the traditional mitigation techniques introduced in Chapter 3. These fault-injection results are used to measure the reliability of the LEON3 processors protected with TMR with roll-forward checkpointing and DWC with rollback checkpointing. The reliability results show that a LEON3 processor protected with full TMR has almost a 6000× improvement in MTTF.

As is done for the unprotected LEON3 processor, a decisive set of ACE and unACE configuration bits are collected for each of the mitigated LEON3 processor designs. This set of configuration bits is obtained by taking the union of individual fault-injection tests performed with each of the testbench programs discussed in Chapter 5. Similar to the unprotected LEON3 processor,
to perform the union of the individual tests, a pessimistic approach is taken. First, if any configuration bit in any of the tests is counted as an SDC bit, it is counted as an SDC bit in the collected results. Next, if any configuration bit in any of the tests is labeled as DUE bit, and is not already counted as an SDC bit in the collected results, then that configuration bit is counted as a DUE bit in the collected results. Next, if a configuration bit in any of the individual results is considered to be a DRU bit, and is not already labeled as an SDC or DUE bit in the collected results, then that bit is counted as a DRU bit in the collected results. Finally, all remaining configuration bits are considered to be unACE bits. The collected set of ACE and unACE bits is pessimistic since the reliability of the union is worse than the reliability of any of the individual tests.

Table 6.3 compares the number of unACE and ACE bits for the unmitigated LEON3 processor against the traditionally mitigated LEON3 processors. The percentages reported in the DRE, DUE, and SDC columns are with respect to only the ACE bits for the given processor. The dloop column reports the number of dloops (Chapter 3) that occur. The table shows that the number of SDCs and DUEs are significantly reduced with the application of these traditional mitigation techniques. DWC and checkpointing reduces the number of SDCs by 36×, reduces the number of DUEs by 30×. Full TMR reduces the number of SDCs by over 12,700× and reduces the number of DUEs by almost 900×. The table also shows that dloops are extremely rare. The rarity of dloops comes from the small average detection latency of TMR and DWC and checkpointing (Table 3.3).

Table 6.3: Fault-injection results for the unmitigated and hardware mitigated LEON3 processors.

<table>
<thead>
<tr>
<th>LEON3 Design</th>
<th>unACE</th>
<th>ACE</th>
<th>dloop</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DRU</td>
<td>DUE</td>
<td>SDC</td>
</tr>
<tr>
<td>Unmitigated</td>
<td>537,825 (79.4%)</td>
<td>0 (0.0%)</td>
<td>11,637 (8.36%)</td>
</tr>
<tr>
<td>DWC &amp; Check (1 clock)</td>
<td>1,378,813 (75.2%)</td>
<td>450,527 (98.98%)</td>
<td>762 (0.16%)</td>
</tr>
<tr>
<td>DWC &amp; Check (2 clocks)</td>
<td>1,369,864 (74.5%)</td>
<td>463,957 (99.17%)</td>
<td>376 (0.08%)</td>
</tr>
<tr>
<td>TMR (no triplicated in/out)</td>
<td>2,458,337 (79.2%)</td>
<td>642,523 (99.68%)</td>
<td>73 (0.01%)</td>
</tr>
<tr>
<td>TMR (triplicated in/out)</td>
<td>2,421,376 (78.7%)</td>
<td>654,013 (99.996%)</td>
<td>13 (0.002%)</td>
</tr>
</tbody>
</table>

The ACE bits in Table 6.3 are used to calculate the reliability of the traditionally mitigated LEON3 processor designs. The reliability of each of the LEON3 processors is shown in Table 6.4.
The AVF values in the table show that the unmitigated LEON3 is almost $100\times$ more vulnerable than the LEON3 protected with DWC and checkpointing, and over $27,000\times$ more vulnerable than the LEON3 protected with full TMR and roll-forward checkpointing. When compared to the unmitigated processor, the MTTF and MuITF of the LEON3 processor with full TMR implemented are almost $6000\times$ greater. The RAP metric in Table 6.4 shows that gains in reliability for each of the mitigated LEON3 processor designs outweigh their increase in cost.

Table 6.4: Comparison of AVF, MTTF, and MuITF of an unmitigated LEON3 against the traditionally mitigated LEON3 processors.

<table>
<thead>
<tr>
<th>LEON3 Design</th>
<th>AVF</th>
<th>MTTF (years)</th>
<th>MuITF $\times 10^{14}$ (instructions)</th>
<th>RAP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unmitigated</td>
<td>20.57% (1.00×)</td>
<td>0.07 (1.00×)</td>
<td>0.46 (1.00×)</td>
<td>1.00×</td>
</tr>
<tr>
<td>DWC (1 clock)</td>
<td>0.22% (94×)</td>
<td>2.44 (33.7×)</td>
<td>13.83 (30.1×)</td>
<td>12.31×</td>
</tr>
<tr>
<td>DWC (2 clocks)</td>
<td>0.22% (94×)</td>
<td>2.53 (34.9×)</td>
<td>14.31 (35.4×)</td>
<td>13.26×</td>
</tr>
<tr>
<td>TMR (non-trip. in/out)</td>
<td>0.07% (294×)</td>
<td>4.70 (64.7×)</td>
<td>30.01 (64.7×)</td>
<td>14.13×</td>
</tr>
<tr>
<td>TMR (trip. in/out)</td>
<td>0.00075% (27,466×)</td>
<td>428.5 (5906×)</td>
<td>2737 (5906×)</td>
<td>1300×</td>
</tr>
</tbody>
</table>

The individual LEON3 processor component protection provided by TMR with roll-forward checkpointing and by DWC with rollback checkpointing is shown in Table 6.5. The table shows that for all but the full TMR design, the most sensitive processor component is the processor output. Since full TMR triplicates the output, and performs votes off-chip voting, these sensitivities on the output are completely removed. For the DWC with checkpointing design it is not surprising that the interrupt controller is one of the more sensitive units. As discussed in Chapter 3, the interrupt controller is not duplicated in the DWC designs.

6.3 Reliability of Individual Software Fault-Tolerance Techniques

In this section the hardware fault-injection results of the individual software detection techniques are presented. Each detection technique is applied alone to the LEON3 processor along with rollback checkpointing (to recover when upsets are detected). The individual techniques are
Table 6.5: The protection that TMR and DWC with checkpointing provide to each LEON3 processor component.

<table>
<thead>
<tr>
<th>LEON3 Component</th>
<th>DWC (1 clock)</th>
<th>DWC (2 clocks)</th>
<th>TMR (non-trip. in/out)</th>
<th>TMR (trip. in/out)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock</td>
<td>96.7%</td>
<td>96.6%</td>
<td>96.5%</td>
<td>100%</td>
</tr>
<tr>
<td>Reset</td>
<td>91.6%</td>
<td>92.3%</td>
<td>96.5%</td>
<td>99.99%</td>
</tr>
<tr>
<td>Top Outputs</td>
<td>88.2%</td>
<td>88.2%</td>
<td>71.5%</td>
<td>100%</td>
</tr>
<tr>
<td>IRQ Controller</td>
<td>98.7%</td>
<td>96.8%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Mult/Div</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Icache</td>
<td>99.99%</td>
<td>99.9%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Dcache</td>
<td>99.94%</td>
<td>99.97%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Main Memory</td>
<td>99.95%</td>
<td>99.97%</td>
<td>100%</td>
<td>99.99%</td>
</tr>
<tr>
<td>Register File</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>99.99%</td>
</tr>
<tr>
<td>ALU</td>
<td>99.99%</td>
<td>99.99%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Pipeline</td>
<td>99.5%</td>
<td>99.5%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Checkpoint Control</td>
<td>97.8%</td>
<td>99.9%</td>
<td>99.99%</td>
<td>99.99%</td>
</tr>
<tr>
<td>Output Unit</td>
<td>98.8%</td>
<td>99.0%</td>
<td>99.3%</td>
<td>99.97%</td>
</tr>
<tr>
<td>Full LEON3</td>
<td>99.0%</td>
<td>99.2%</td>
<td>99.7%</td>
<td>99.99%</td>
</tr>
</tbody>
</table>

compared in terms of the protection they provide to the various processor components. Individually, the software detection techniques do not provide a large reliability improvement over an unprotected LEON3 processor. But the reliability results in this section provide important information to create effective combinations of software fault-tolerance techniques which do provide significant reliability improvements.

Before comparing the individual software detection techniques in terms of the protection they provide to the LEON3 processor components, their hardware fault-injection results are considered. The raw hardware fault-injection results for each individual detection technique is shown in Table 6.6 when only the ToUpper testbench program is run. The fault-injection results for the unmitigated LEON3 softcore processor are also for only the ToUpper benchmark program. The results show that consistency checks eliminate the most SDC and DUE bits ($3 \times$ fewer SDCs),
and that applying CDWC to the register file provides the least amount of protection (only $1.2 \times$ fewer SDCs). The results also show that the hardware-based detection techniques (CDWC register file, parity, and memory scrubbing) allow the least number of dloop bits. Control-flow monitoring produces the largest number of dloops.

Table 6.6: Fault-injection comparisons for the unmitigated LEON3 processor and LEON3 processor protected with each of the individual detection techniques and rollback checkpointing.

<table>
<thead>
<tr>
<th>LEON3 Design</th>
<th>unACE</th>
<th>ACE</th>
<th>dloop</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DUE</td>
<td>SDC</td>
<td></td>
</tr>
<tr>
<td>Unmitigated</td>
<td>0 (0.0%)</td>
<td>6,240 (10.0%)</td>
<td>N/A</td>
</tr>
<tr>
<td>CDWC Register File</td>
<td>31,404 (39.7%)</td>
<td>120 (0.15%)</td>
<td>2</td>
</tr>
<tr>
<td>Control-Flow Monitoring</td>
<td>33,585 (52.2%)</td>
<td>5885 (9.2%)</td>
<td>4962</td>
</tr>
<tr>
<td>Consistency Checks</td>
<td>102,663 (84.5%)</td>
<td>1212 (1.0%)</td>
<td>842</td>
</tr>
<tr>
<td>Software Indicators</td>
<td>41,304 (51.4%)</td>
<td>4396 (5.5%)</td>
<td>2592</td>
</tr>
<tr>
<td>Parity &amp; Scrubbing</td>
<td>87,779 (74.2%)</td>
<td>730 (0.62%)</td>
<td>45</td>
</tr>
</tbody>
</table>

The ACE configuration bit results in Table 6.6 are used to measure the reliability of the individual fault-tolerance techniques. Table 6.7 shows that individually, none of the reliability techniques provide sufficient protection. In fact, two of the detection techniques (CDWC register file, and control-flow monitoring) have a RAP that is less than one, suggesting that the reliability they provide alone (individually not collectively) is not likely worth the costs they incur. Ironically, these two detection techniques are the two most expensive detection techniques. Control-flow monitoring incurs the largest performance penalty ($1.51 \times$), and register file CDWC has the largest area cost ($1.10 \times$). The results in the table also reaffirm that consistency checks are the single most effective individual software detection technique. Alone, none of these software detection techniques provides significant reliability, but combining them provides much more reliability.

Although consistency checks provide the best overall individual protection, other techniques may provide better protection to the individual LEON3 processor components. Table 6.8 shows how much protection each of the individual detection techniques provides to the LEON3 processor components. The percentages shown represent the ratio of DRU bits to ACE bits for each processor component. The table shows that although consistency checks provide the best
Table 6.7: Comparison of the architectural vulnerability factors (AVFs), relative mean time to failure (MTTF), relative mean useful instruction to failure (MuITF), and RAP of an unmitigated LEON3 against the LEON3 processor protected with each of the software fault-tolerance techniques individually.

<table>
<thead>
<tr>
<th>LEON3 Design</th>
<th>AVF (×)</th>
<th>MTTF (years) (×)</th>
<th>MuITF (×10^14 instructions) (×)</th>
<th>RAP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unmitigated</td>
<td>9.31%</td>
<td>0.16 (1.00×)</td>
<td>1.02 (1.00×)</td>
<td>1.00</td>
</tr>
<tr>
<td>CDWC Register File</td>
<td>4.96%</td>
<td>0.21 (1.28×)</td>
<td>1.33 (1.30×)</td>
<td>0.90</td>
</tr>
<tr>
<td>Control-Flow Monitoring</td>
<td>3.42%</td>
<td>0.32 (1.99×)</td>
<td>1.29 (1.26×)</td>
<td>0.99</td>
</tr>
<tr>
<td>Consistency Checks</td>
<td>1.93%</td>
<td>0.52 (3.23×)</td>
<td>2.96 (2.90×)</td>
<td>2.15</td>
</tr>
<tr>
<td>Software Indicators</td>
<td>4.27%</td>
<td>0.25 (1.56×)</td>
<td>1.63 (1.60×)</td>
<td>1.18</td>
</tr>
<tr>
<td>Parity &amp; Scrubbing</td>
<td>3.37%</td>
<td>0.31 (1.90×)</td>
<td>1.98 (1.94×)</td>
<td>1.36</td>
</tr>
</tbody>
</table>

overall protection to the LEON3 and to some of its components (clock, reset, mult/div, register file, ALU, pipeline, and checkpoint control), other detection techniques provide the best protection to some of the components. For example, parity with memory scrubbing is the only detection technique that provides any protection to the top-level outputs, and also provides the most protection to the instruction cache, data cache, main memory, and output unit. Software indicators are the only detection technique to provide any protection to the interrupt controller.

### 6.4 Reliability of Full Software Fault-Tolerance

In this section the individual software fault-tolerance techniques are combined to provide greater protection to the LEON3 softcore processor. This section shows that although the best protection is provided when all techniques are combined, there are interesting processor protection-space points provided when only some the detection techniques are applied. Although the best individual detection technique only provides a 3× improvement in MTTF and MuITF, combining all the detection techniques provides a 12× MTTF improvement and 8× MuITF improvement.

When considering which software detection techniques combine to provide the best protection at the lowest cost, it is important to first determine if all of the detection techniques are required for optimal reliability. Table 6.8 provides indicators to initially determine which software detection techniques must be included in an optimal combination of techniques. Consistency checks provide
Table 6.8: The protection that each individual fault-tolerance technique provides to each LEON3 processor component.

<table>
<thead>
<tr>
<th>LEON3 Component</th>
<th>CDWC Register File</th>
<th>Control-Flow Monitoring</th>
<th>Consistency Checks</th>
<th>Software Indicators</th>
<th>Parity &amp; Scrubbing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock</td>
<td>24.3%</td>
<td>50.7%</td>
<td>76.8%</td>
<td>46.9%</td>
<td>62.3%</td>
</tr>
<tr>
<td>Reset</td>
<td>6.4%</td>
<td>42.8%</td>
<td>75.8%</td>
<td>41.3%</td>
<td>59.5%</td>
</tr>
<tr>
<td>Top Outputs</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>5.4%</td>
<td>50.4%</td>
</tr>
<tr>
<td>IRQ Controller</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>100.0%</td>
<td>0%</td>
</tr>
<tr>
<td>Mult/Div</td>
<td>0%</td>
<td>30.4%</td>
<td>98.1%</td>
<td>5.6%</td>
<td>65.5%</td>
</tr>
<tr>
<td>Icache</td>
<td>83.1%</td>
<td>74.0%</td>
<td>77.4%</td>
<td>78.5%</td>
<td>83.7%</td>
</tr>
<tr>
<td>Dcache</td>
<td>77.3%</td>
<td>4.6%</td>
<td>90.9%</td>
<td>5.9%</td>
<td>94.0%</td>
</tr>
<tr>
<td>Main Memory</td>
<td>1.4%</td>
<td>64.0%</td>
<td>90.4%</td>
<td>84.7%</td>
<td>99.0%</td>
</tr>
<tr>
<td>Register File</td>
<td>92.7%</td>
<td>93.5%</td>
<td>98.1%</td>
<td>5.1%</td>
<td>77.9%</td>
</tr>
<tr>
<td>ALU</td>
<td>1.4%</td>
<td>63.6%</td>
<td>91.4%</td>
<td>23.3%</td>
<td>78.2%</td>
</tr>
<tr>
<td>Pipeline</td>
<td>2.7%</td>
<td>48.1%</td>
<td>80.1%</td>
<td>41.0%</td>
<td>57.9%</td>
</tr>
<tr>
<td>Checkpoint Control</td>
<td>75.3%</td>
<td>61.3%</td>
<td>82.8%</td>
<td>43.5%</td>
<td>78.9%</td>
</tr>
<tr>
<td>Output Unit</td>
<td>82.8%</td>
<td>28.3%</td>
<td>20.3%</td>
<td>16.7%</td>
<td>98.7%</td>
</tr>
<tr>
<td>Full LEON3</td>
<td>39.7%</td>
<td>53.6%</td>
<td>84.5%</td>
<td>51.4%</td>
<td>75.1%</td>
</tr>
</tbody>
</table>

the best detection coverage to all but five of the LEON3 processor components listed in Table 6.8, so this detection technique should be included in an optimal combination. Parity with memory scrubbing, and software indicators provide the best detection coverage to the remaining five techniques, thus these two detection techniques should also be included in an optimal combination of detection techniques.

Although control-flow monitoring or a CDWC register file do not provide the best coverage to any of the processor components, it cannot be assumed that they should not be added to an optimal combination of detection techniques. Based on the results in Table 6.8, it is unclear whether or not these two techniques are required.
The four different combinations of detection techniques considered in this work are shown in Table 6.9. Rollback checkpointing is added to each combination of techniques to recover when an upset is detected. The first combination combines all five of the software detection techniques previously discussed (referred to as Full SW). The second combination reduces performance cost by not including control-flow monitoring (referred to as no Control-Flow). The third combination reduces area cost by including all detection techniques except for the CDWC register file (referred to as no CDWC). Finally, the last combination reduces both area and performance by only combining consistency checks, software indicators, and parity with scrubbing (referred to as Min SW).

Table 6.9: The four combinations of detection techniques used in this work. Each combination includes rollback checkpointing for upset recovery.

<table>
<thead>
<tr>
<th>Combination</th>
<th>Detection Techniques</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full SW</td>
<td>Consistency Checks&lt;br&gt;Parity &amp; Mem. Scrubbing&lt;br&gt;Software Indicators&lt;br&gt;Control-Flow Monitoring&lt;br&gt;CDWC Register File</td>
</tr>
<tr>
<td>no Control-Flow</td>
<td>Consistency Checks&lt;br&gt;Parity &amp; Mem. Scrubbing&lt;br&gt;Software Indicators&lt;br&gt;CDWC Register File</td>
</tr>
<tr>
<td>no CDWC</td>
<td>Consistency Checks&lt;br&gt;Parity &amp; Mem. Scrubbing&lt;br&gt;Software Indicators&lt;br&gt;Control-Flow Monitoring</td>
</tr>
<tr>
<td>Min SW</td>
<td>Consistency Checks&lt;br&gt;Parity &amp; Mem. Scrubbing&lt;br&gt;Software Indicators</td>
</tr>
</tbody>
</table>

6.4.1 Reliability Comparisons

The details of the collected results for each combination of software detection techniques is shown in Table 6.10. There are three important results shown in this table. First, the best protection comes from combining all of the software fault-tolerance techniques. Compared to the unprotected
LEON3 processor, the full combination of detection techniques with checkpointing reduces SDCs by over $8 \times$ and reduces the number of DUEs by $2 \times$. In terms of configuration bit detection coverage, the full combination detects and recovers from upsets to 99% of the configuration bits (including the unACE bits). The second observation is that control-flow monitoring adds very little in terms of performance (comparing Min SW with no CDWC), but incurs a large performance cost. The third result is that register file CDWC adds significant reliability (comparing Min SW with no Control-Flow). These results are made much clearer when reliability metrics are compared.

Table 6.10: Comparison of fault-injection results for the unmitigated LEON3 processor and LEON3 processor protected with combinations of software detection techniques and rollback checkpointing.

<table>
<thead>
<tr>
<th>LEON3 Design</th>
<th>unACE</th>
<th>ACE</th>
<th>DRU</th>
<th>DUE</th>
<th>SDC</th>
<th>dloop</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unmitigated</td>
<td>537,825 (79.4%)</td>
<td>0 (0.0%)</td>
<td>11,637 (8.4%)</td>
<td>105,704 (91.6%)</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>Min SW &amp; Checkpoint</td>
<td>677,385 (76.3%)</td>
<td>192,235 (91.2%)</td>
<td>6027 (2.9%)</td>
<td>12,451 (5.9%)</td>
<td>2159</td>
<td></td>
</tr>
<tr>
<td>no CDWC &amp; Checkpoint</td>
<td>705,499 (77.8%)</td>
<td>186,168 (92.4%)</td>
<td>4028 (2.0%)</td>
<td>11,316 (5.6%)</td>
<td>3270</td>
<td></td>
</tr>
<tr>
<td>no Control-Flow &amp; Checkpoint</td>
<td>733,423 (74.2%)</td>
<td>244,218 (95.6%)</td>
<td>4685 (1.8%)</td>
<td>6497 (2.5%)</td>
<td>2530</td>
<td></td>
</tr>
<tr>
<td>Full SW &amp; Checkpoint</td>
<td>723,054 (73.0%)</td>
<td>257,014 (95.9%)</td>
<td>4634 (1.73%)</td>
<td>6426 (2.40%)</td>
<td>2524</td>
<td></td>
</tr>
</tbody>
</table>

The reliability metrics reported in Table 6.11 reemphasize the observations shown by the fault-injection results of Table 6.10 and adds some new observations. First, the full combination of detection techniques (Full SW) provides the best reliability in terms of MTTF, but the combination that does not include control-flow monitoring (no Control-Flow) provides the best reliability in terms of MuITF. Comparing the Full SW combination results with the no Control-Flow results shows that control-flow monitoring adds very little additional reliability to the other detection techniques. Comparing the MuITF and RAP of these two detection combinations shows that the large performance cost incurred by control-flow monitoring outweighs the reliability it adds. Judging by the RAP metric, the optimal combination of detection techniques includes all but control-flow monitoring.

Table 6.12 shows the percent DRU coverage the each of the detection combinations provide to each LEON3 processor component (compare with Table 6.8). The percentages reported are the
Table 6.11: Comparison of the architectural vulnerability factors (AVFs), relative mean time to failure (MTTF), relative mean useful instruction to failure (MuITF), and RAP of an unmitigated LEON3 against the LEON3 processor protected with combinations of software fault-tolerance techniques.

<table>
<thead>
<tr>
<th>LEON3 Design</th>
<th>AVF</th>
<th>MTTF (years)</th>
<th>MuITF ×10¹⁸ (instructions)</th>
<th>RAP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unmitigated</td>
<td>20.57% (1.00×)</td>
<td>0.07 (1.00×)</td>
<td>0.46 (1.00×)</td>
<td>1.00</td>
</tr>
<tr>
<td>Min SW &amp; Checkpoint</td>
<td>2.08% (9.89×)</td>
<td>0.53 (7.35×)</td>
<td>3.05 (6.67×)</td>
<td>4.96</td>
</tr>
<tr>
<td>no CDWC &amp; Checkpoint</td>
<td>1.69% (12.17×)</td>
<td>0.64 (8.85×)</td>
<td>2.54 (5.57×)</td>
<td>4.05</td>
</tr>
<tr>
<td>no Control-Flow &amp; Checkpoint</td>
<td>1.13% (18.20×)</td>
<td>0.88 (12.15×)</td>
<td>5.03 (11.03×)</td>
<td>7.38</td>
</tr>
<tr>
<td>Full SW &amp; Checkpoint</td>
<td>1.12% (18.37×)</td>
<td>0.89 (12.28×)</td>
<td>3.53 (7.73×)</td>
<td>5.15</td>
</tr>
</tbody>
</table>

The number of DRU bits out of the total number of ACE bits for each processor component. The table shows that combining detection techniques significantly increases the protection for each processor component. As expected, the most sensitive components are the top-level outputs and the top-level inputs (clock and reset). The coverage rows for the instruction and data caches show that the main contribution of control-flow monitoring is its protection of the caches.

6.4.2 Cost Comparisons

The area costs of each combination of detection techniques is shown in Table 6.13. The table shows that the area costs of every combination are very close to the sum of the individual components (Tables 4.1, 4.3, 4.7, 4.8, and 4.9).

Performance costs of each combination of software detection techniques are shown in Table 6.14. The table reports the performance cost ($\rho$) in terms of how many more clock cycles are required to execute a program compared to an unprotected processor. The table also reports each combination’s average detection latency, and average detection latency for the dloops that occur. The performance costs are just the sum of each of the individual detection technique performance costs.

Software fault-tolerance allows the processor design-space to be more fully explored. Different software detection technique combinations trade-off reliability for area and performance. The most reliable combination of software detection techniques has a MTTF that approaches that
Table 6.12: The protection that different combinations of software fault-tolerance techniques provide to each LEON3 processor component.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock</td>
<td>87.5%</td>
<td>91.5%</td>
<td>95.3%</td>
<td>95.5%</td>
</tr>
<tr>
<td>Reset</td>
<td>90.8%</td>
<td>91.9%</td>
<td>92.4%</td>
<td>91.9%</td>
</tr>
<tr>
<td>Top Outputs</td>
<td>58.4%</td>
<td>49.3%</td>
<td>56.7%</td>
<td>50.8%</td>
</tr>
<tr>
<td>IRQ Controller</td>
<td>98.5%</td>
<td>98.4%</td>
<td>97.7%</td>
<td>100%</td>
</tr>
<tr>
<td>Mult/Div</td>
<td>98.0%</td>
<td>98.2%</td>
<td>98.2%</td>
<td>98.5%</td>
</tr>
<tr>
<td>Icache</td>
<td>83.9%</td>
<td>94.8%</td>
<td>98.2%</td>
<td>98.6%</td>
</tr>
<tr>
<td>Dcache</td>
<td>88.8%</td>
<td>82.8%</td>
<td>94.0%</td>
<td>95.0%</td>
</tr>
<tr>
<td>Main Memory</td>
<td>99.4%</td>
<td>99.3%</td>
<td>99.5%</td>
<td>99.5%</td>
</tr>
<tr>
<td>Register File</td>
<td>99.1%</td>
<td>99.3%</td>
<td>99.6%</td>
<td>99.6%</td>
</tr>
<tr>
<td>ALU</td>
<td>94.3%</td>
<td>95.4%</td>
<td>94.9%</td>
<td>95.6%</td>
</tr>
<tr>
<td>Pipeline</td>
<td>90.7%</td>
<td>93.6%</td>
<td>93.7%</td>
<td>94.2%</td>
</tr>
<tr>
<td>Checkpoint Control</td>
<td>89.6%</td>
<td>90.2%</td>
<td>96.6%</td>
<td>97.0%</td>
</tr>
<tr>
<td>Output Unit</td>
<td>97.9%</td>
<td>98.7%</td>
<td>98.2%</td>
<td>98.6%</td>
</tr>
<tr>
<td>Full LEON3</td>
<td>91.2%</td>
<td>92.4%</td>
<td>96.8%</td>
<td>98.6%</td>
</tr>
</tbody>
</table>

of DWC with rollback checkpointing, but for half the area cost, and a minor performance cost. The area cost of this combination is well below a third the cost of TMR with roll-forward checkpointing.

6.5 Processor Design-Space Comparisons

The softcore processor detection and recovery techniques discussed throughout this work provide varying amounts of reliability at the cost of area or performance or both. Although full TMR provides the best protection for the LEON3 softcore processor it comes with the largest area cost. A combination of software-based detection techniques and checkpointing may be an
Table 6.13: Area costs of a LEON3 processor protected with different combinations of software detection techniques and rollback checkpointing.

<table>
<thead>
<tr>
<th>Software Detection Area Costs</th>
<th>LEON3 Design</th>
<th>Slices</th>
<th>BRAMs</th>
<th>DSPs</th>
<th>Config Bits (CFGbits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unmitigated</td>
<td>3058 (1.00×)</td>
<td>12 (1.00×)</td>
<td>4 (1.00×)</td>
<td>677,065 (1.00×)</td>
<td></td>
</tr>
<tr>
<td>Checkpoint only</td>
<td>4386 (1.43×)</td>
<td>21 (1.75×)</td>
<td>4 (1.00×)</td>
<td>894,355 (1.32×)</td>
<td></td>
</tr>
<tr>
<td>Min SW &amp; Chkpt.</td>
<td>4592 (1.50×)</td>
<td>21 (1.75×)</td>
<td>4 (1.00×)</td>
<td>888,098 (1.31×)</td>
<td></td>
</tr>
<tr>
<td>no CDWC &amp; Chkpt.</td>
<td>4596 (1.50×)</td>
<td>21 (1.75×)</td>
<td>4 (1.00×)</td>
<td>907,011 (1.34×)</td>
<td></td>
</tr>
<tr>
<td>no Control-Flow &amp; Chkpt.</td>
<td>4907 (1.60×)</td>
<td>23 (1.92×)</td>
<td>4 (1.00×)</td>
<td>986,624 (1.46×)</td>
<td></td>
</tr>
<tr>
<td>Full SW &amp; Chkpt.</td>
<td>4864 (1.59×)</td>
<td>23 (1.92×)</td>
<td>4 (1.00×)</td>
<td>991,128 (1.46×)</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.14: Performance costs of a LEON3 processor protected with different combinations of software detection techniques and rollback checkpointing.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Unmitigated</td>
<td>1.00×</td>
<td>1.00×</td>
<td>N/A</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>Checkpoint only</td>
<td>1.04×</td>
<td>1.01×</td>
<td>N/A</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>Min SW &amp; Chkpt.</td>
<td>1.27×</td>
<td>1.13×</td>
<td>2450</td>
<td>15,233</td>
<td></td>
</tr>
<tr>
<td>no CDWC &amp; Chkpt.</td>
<td>1.35×</td>
<td>1.63×</td>
<td>2446</td>
<td>15,178</td>
<td></td>
</tr>
<tr>
<td>no Control-Flow &amp; Chkpt.</td>
<td>1.31×</td>
<td>1.13×</td>
<td>1989</td>
<td>14,736</td>
<td></td>
</tr>
<tr>
<td>Full SW &amp; Chkpt.</td>
<td>1.35×</td>
<td>1.63×</td>
<td>3329</td>
<td>17,882</td>
<td></td>
</tr>
</tbody>
</table>

acceptable lower-cost alternative. This section explores the processor design-space created by the reliability-cost trade-off.

The processor design-space created by the reliability-cost trade-off can be defined in one of two ways. Reliability-cost trade-off points can be plotted on either an MTTF/area graph, or an MuITF/area graph. Figure 6.1 shows the MTTF/area trade-off curve. Area (in terms of configuration bits) is plotted along the x-axis, and reliability (in terms of MTTF) is plotted along a logarithmic y-axis. The graph shows that for a relatively small area cost software fault-tolerance provides a significant increase in reliability. All four combinations of detection techniques are clustered around the same area. The next cluster of points on the graph belong to the DWC designs. DWC with checkpointing provides an improvement in reliability for significant increase in
area. Finally, the two TMR points show up on the extreme right of the graph. TMR with triplicated inputs/outputs dwarfs TMR without triplicated inputs/outputs. Both have about the same area cost, but the reliability of TMR with triplicated inputs/outputs is significantly larger.

![MTTF/Area Graph](image)

**Figure 6.1:** Processor design-space created by a MTTF-area trade-off.

The graph shown in Figure 6.2 shows the MuITF improvement relative to the unmitigated LEON3 softcore processor, versus the relative area cost, with respect to the unmitigated LEON3. This graph adds performance costs to the design-space. The plotted points in Figure 6.2 are very similar in arrangement to the plotted points of the MTTF/area graph. Although software fault-tolerance comes at the cost of performance, Figure 6.2 shows that the relative cost is low. The cluster of points corresponding to the software fault-tolerant designs is still in the same position relative to the other points. On a linear y-axis, the slope of the line drawn from the origin to any of the points in the graph is equal to the RAP of that mitigation technique. Both the MTTF/area graph and the MuITF area graphs show that software fault-tolerance provides a significant increase in reliability for a small cost.
### 6.6 Radiation Hardened Processor Comparisons

The fault-injection results presented in this chapter show that softcore processors can be made reliable with software fault-tolerance for a low area cost. But an important question to ask is: Is the reliability that software fault-tolerance provides for softcore processors sufficient for space-flight? Although a definitive answer to this question is beyond the scope of this work, comparisons between the reliability of software fault-tolerant softcore processors and rad-hard ASIC-based processors suggest an answer. This section compares the failure rates of two popular rad-hard processors with the failure rates of the LEON3 softcore processor designs previously discussed. The two rad-hard processors used for this comparison are the BAE Systems RAD750 [4, 5], and the BAE Systems RAD6000 [93]. Although this is not entirely an apples-to-apples comparison, it suggests that software fault-tolerance, and the other mitigation techniques previously discussed, provide sufficient reliability for softcore processors to be used in some space-based applications.

In order to compare the reliability of a softcore processor with a rad-hard processor, accurate upset rates ($\lambda_{\text{upset}}$) must be established for both processors operating in the same environment. The upset rate used throughout this work for the LEON3 softcore processor (2.78E-12 upsets/bit-s) is the upset rate for a Xilinx Virtex 4 FX60 FPGA observed in a geosynchronous Earth orbit (GEO).
at solar minimum [88–90]. The rad-hard processors used for this comparison operate in a GEO orbit at solar minimum with an upset rate of $\lambda_{upset} = 1.16 \times 10^{-15}$ upsets/bit-s [93, 94].

The failure rates (the inverse of MTTF) of the two rad-hard processors in a GEO orbit at solar minimum are shown in Table 6.15. These failure rates were obtained through extensive radiation testing [93, 94]. The failure rates of the LEON3 softcore processors in a GEO orbit at solar minimum are shown in Table 6.16. These failure rate are obtained through hardware fault-injection, however, radiation testing validates the accuracy of these failure rates to within 7% (Appendix H).

Table 6.15: Failure rates of two popular rad-hard processors using an upset rate of a GEO orbit at solar minimum.

<table>
<thead>
<tr>
<th>Rad-Hard Failure Rates</th>
<th>Failure Rate (fails/year)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BAE RAD750</td>
<td>2.2</td>
</tr>
<tr>
<td>BAE RAD6000</td>
<td>0.2</td>
</tr>
</tbody>
</table>

Table 6.16: Failure rates of LEON3 softcore processors using an upset rate of a GEO orbit at solar minimum.

<table>
<thead>
<tr>
<th>Softcore Failure Rates</th>
<th>Failure Rate (fails/year)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEON3 unmitigated</td>
<td>14.3</td>
</tr>
<tr>
<td>LEON3 Min SW &amp; Checkpointing</td>
<td>1.8</td>
</tr>
<tr>
<td>LEON3 Full SW &amp; Checkpointing</td>
<td>1.1</td>
</tr>
<tr>
<td>LEON3 DWC &amp; Checkpointing (2 clocks)</td>
<td>0.4</td>
</tr>
<tr>
<td>LEON3 TMR (trip. in/out)</td>
<td>0.002</td>
</tr>
</tbody>
</table>

Although a comparison of the failure rates in Tables 6.15 and 6.16 shows that fault-tolerant softcore processors match-up very favorably to rad-hard processors, there is more to consider when comparing rad-hard processors to softcore processors. The total ionizing dose (TID) [6, 7] of a
commercial SRAM-based FPGA is significantly lower than that of the rad-hard processors. Also, this work has not considered SEFIs or SETs in commercial SRAM-based FPGAs [95]. But comparing the failure rates in Tables 6.15 and 6.16 suggests that fault-tolerant softcore processors could be used in spaceflight. The comparison also suggests that software fault-tolerance provides enough reliability for processors used in some space-based applications.

Softcore processors are more attractive than rad-hard processors for spaceflight for reasons other than reliability. First, softcore processors are infinitely more flexible and cost orders of magnitude less expensive than rad-hard processors [4, 11–13]. Additionally, softcore processors are less expensive in terms of area and performance than rad-hard processors. Since, unlike rad-hard processors, softcore processors can be implemented in the latest technologies, they have the potential for smaller sizes and faster clock speeds.

6.7 Summary

Hardware fault-injection results presented in this section allow the processor design-space to be explored with different fault-tolerance techniques. In a GEO orbit at solar minimum, an unmitigated LEON3 processor fails about once every 26 days. A LEON3 softcore processor protected with TMR has a MTTF that is almost 6000× greater than the unmitigated processor, but incurs an area cost that is more than 4× greater. The area costs of DWC and rollback checkpointing are much lower than TMR (only 2.7× greater than an unmitigated softcore processor), but its reliability is not as good (only 35× greater than an unmitigated softcore processor). The software fault-tolerant LEON3 has a low area cost (only 1.4× greater than the unprotected LEON3), and a moderate performance cost (1.6× greater) and reliability (about 12× better than the unprotected LEON3). Failure rate comparisons with rad-hard processors show that the reliability improvement provided by software fault-tolerance may be sufficient for spaceflight.
CHAPTER 7. CONCLUSION

Software fault-tolerance can detect and recover from 99% of upsets to the configuration memory of a softcore processor. This protection comes for only a 1.4× area cost and 1.6× performance cost compared to an unprotected LEON3 softcore processor. In terms of area, this reliability is over 3× less costly than TMR, and about half as expensive as DWC with checkpointing. Although TMR and DWC with checkpointing provide greater reliability than software fault-tolerance, failure rate comparisons show that software fault-tolerance may be sufficient to protect a softcore processor used in space-based applications.

The most reliable combination of software detection techniques with checkpointing provides reliability that is over 12× greater than the reliability of an unprotected LEON3 processor. An ultra low-cost software fault-tolerant softcore processor can be created with a minimal combination of software detection techniques with checkpointing. This ultra low-cost fault-tolerant LEON3 softcore processor provides a failure rate that is still comparable to a rad-hard processor but which only incurs a 1.3× area cost and 1.1× performance cost.

At the other end of the processor design-space, TMR with triplicated inputs and outputs provides large reliability improvements, but requires a large amount of additional area. If processor inputs and outputs cannot be triplicated, then TMR is hardly worth the excessive area costs it incurs. DWC with checkpointing provides a comparable amount of protection for a significantly reduced area cost. Future work might identify additional points in the processor design-space which provide increases in reliability for a lower cost.

7.1 Future Work

There is room for more processor design-space exploration. New hybrid combinations of hardware and software detection techniques can improve performance and reliability, and potentially reduce area costs. New low-cost detection techniques might also improve reliability and
reduce area costs. Two novel combination of detection techniques are proposed in this section for future work.

7.1.1 Bitstream Scrubbing Detection

Bitstream scrubbing is an essential part of all SRAM-based FPGA designs used in space-flight. A bitstream scrubber continually reads the static parts of an FPGA design’s configuration memory, and compares it to a golden version of the configuration memory. When upsets are detected in the configuration memory, the bitstream scrubber repairs them through partial runtime reconfiguration [96]. Normally, when the bitstream scrubber repairs the configuration memory, the design running on the FPGA receives no notification that a repair has been made.

A novel softcore processor mitigation scheme has the bitstream scrubber send an upset detection signal to the softcore processor, which triggers a rollback. Thus, rollback checkpointing is still used in this scheme. In theory, the bitstream scrubber should detect any and every upset to the logic and routing of a softcore processor, thus this detection technique should be sufficient to protect all of the logic and routing of the processor. Parity and memory scrubbing can be used to protect the user memories in the processor. If SEUs are detected by parity, or the bitstream or memory scrubber, a rollback is performed. The advantage of this protection scheme is that it is very simple, has the potential to consume less area, and could have a better reliability than the full software combination reported in Chapter 4.

The problem with using the bitstream scrubber for upset detection is that it can have an extremely large detection latency. For example, it takes 118ms to scrub the configuration memory of the Xilinx Virtex4 FX60. With a 33 MHz clock, the worst-case detection latency will be 3,894,000 cycles, which is about 3 orders of magnitude worse than the average detection latency in Chapter 4. Such a large detection latency may cause upsets to propagate through the processor in such a way that will lead to more dloops. At the very least, the larger detection latency will require a larger checkpoint interval.

The other potential disadvantage of this technique is that it will produce a significant amount of false positives. Even if the softcore processor consumes only a tenth of the configuration memory of the FPGA, upsets to any part of the configuration memory will produce a rollback. This means that 90% of the rollbacks caused by this detection technique would be unnecessary.
An improved version of this technique could have the bitstream scrubber know whether a given configuration bitstream upset should signal a rollback.

7.1.2 DWC with Parity and Software Indicators

Simple low-cost techniques can be added to the DWC and rollback checkpointing design which would improve performance and reliability. Chapter 4 shows that software indicators provide additional reliability with negligible area and performance costs. Chapter 4 also shows that parity detects a large percentage of upsets with a low detection latency, and a small area cost.

The advantage of adding these two techniques to DWC is that the individual duplicated processors can often self-detect when they have been upset. Normally a compare unit compares the signals of the duplicated processors to detect when one of the two processors has been upset. Since the comparator has no way of determining which of the two processors has been upset, roll-forward checkpointing is not an option. But if the processors can self-detect if they have been upset, roll-forward checkpointing can be used. The state of the good processor is used to correct the faulty processor. Not only does this improve the performance of the processor (since costly rollbacks are less likely required), the additional detection techniques add to the overall reliability of the processor. The RAP of this proposed design can be used to determine if the improvements in reliability and performance outweigh the added area costs.

7.2 Concluding Remarks

Microprocessors are an important part of most space-based applications. Protecting processors from the effects of upsets caused by radiation is important, but traditional methods for protecting processors is extremely expensive. Radiation-hardening is the traditional way of protecting processors. These processors are big, slow, based on old technology and very expensive. Other protection methods can provide comparable reliability for reduced costs.

Softcore processors implemented in SRAM-based FPGAs offer an attractive alternative to rad-hard processors. Softcore processors can be implemented in the latest technologies, they are fast, flexible and significantly less expensive than rad-hard processors. The flexibility provided by FPGAs allow softcore processors to be modified or corrected even while in orbit.
But due to the fact SRAM-based FPGAs are more sensitive to the effects of SEUs than ASIC-based processors, extra care must be taken to detect and correct the effects of SEUs in their configuration and user memories. This work has shown that detection and correction techniques can be applied softcore processors implemented on SRAM-based FPGAs which provide reliability that is comparable to the reliability of rad-hard processors. The hardware fault-injection methodology, reliability metrics, and probability models introduced in this work make this comparison possible. The unique application of software fault-tolerance to softcore processors demonstrated in this work provide a low-cost softcore processor whose reliability is comparable to that of a rad-hard processor.

One of the key contributions of this work is the application of software fault-tolerance to softcore processors implemented in SRAM-based FPGAs. To our knowledge, this represents the first work to apply software fault-tolerance for the purpose of protecting against the effects of upsets in the configuration memory of a softcore processor. This contribution can have a significant impact in the application of reliable softcore processors.

One of the most useful contributions of this dissertation may be its thorough considerations of how to effectively apply checkpointing to softcore processors. This research articulates the unique considerations for applying checkpointing to softcore processors. It provides a thorough discussion on these issues, and provides a probability model for identifying the optimal checkpoint interval for programs running on a softcore processor. This contribution may especially be useful for softcore processors implemented on radiation-hardened FPGAs [68].

Another contribution that is useful for future work is the application of consistency checks for protection against the effects of upsets in the configuration memory of a softcore processor. This research has shown that consistency checks can effectively detect upsets to the functional units of a softcore processor, including a hardware multiplier/divider. In this work, the LEON3 hardware multiplier/divider is implemented with Xilinx DSP functional blocks. This research shows that consistency checks are a viable alternative to traditional spatial redundancy techniques for DSP block reliability.

The flexibility, reconfigurability, and low cost of softcore processors make them an attractive solution for space-based applications. Despite the inherent susceptibility of softcore processors to radiation-induced upsets, softcore processors can be made reliable. This research shows
that there is a reliability-cost design space to be explored when considering softcore processor reliability. This research also shows that the reliability of softcore processors can be comparable to the reliability of rad-hard processors. Low-cost software fault-tolerance techniques may provide sufficient reliability for some space-based applications.
REFERENCES


107


[84] H. Guzman-Miranda, M. Aguirre, and J. Tombs, “Noninvasive fault classification, robustness and recovery time measurement in microprocessor-type architectures subjected
to radiation-induced errors,” *Instrumentation and Measurement, IEEE Transactions on*, vol. 58, no. 5, pp. 1514 –1524, may 2009. 66


113


APPENDIX A. FPGA CONFIGURATION MEMORY FAILURE MODES IN ROUTING

A more detailed look at an SRAM-based FPGA routing matrix shows how SEUs can cause an open or a short. FPGA routing matrices are implemented with programmable interconnect points (PIPs). A simplified PIP (Figure A.1) [70] shows an example of a softcore processor register file write enable signal (WE) correctly routed through the PIP to the output buffer. Figure A.2 shows that upsets to the configuration memory bits that control the PIP cause either an open or a short.

![Figure A.1: A 2x2 programmable interconnect point (PIP) which passes WE to the output buffer.](image)

When an SEU disconnects the WE signal, causing an open (Figures A.2(a) and A.2(b)), a level-restoring buffer pulls the undriven output high [97]. When the expected signal is a register write enable, driving a constant "1" will cause disastrous results (Appendix B). When an SEU instead causes a short, the output buffer is driven by either an AND’ing or OR’ing of the driving signals. For example Figure A.2(c) shows that an upset to Bit B causes an OR’ing of WE with buf1. If buf1 is driven low, then the result of the OR’ing is that WE is still driven correctly to the output buffer. But if buf1 is driven high, then the result of the OR’ing is that the output buffer...
is driven high, which again causes disastrous results. Figure A.2(d) shows that an upset to Bit D causes an AND’ing of \textit{WE} with \textit{buf2}. If \textit{buf2} is driven high, then the result of the AND’ing is that \textit{WE} is correctly driven to the output buffer. But if \textit{buf2} is driven low, then the output buffer is driven low, and the register file is never updated with new results.

![Diagram showing SEU effects on a PIP](image)

(a) An open is created when an SEU hits Bit A.
(b) An open is created when an SEU hits Bit C.
(c) A short is created when an SEU hits Bit B.
(d) A short is created when an SEU hits Bit D.

Figure A.2: An SEU in a PIP can cause an open or a short.
APPENDIX B. A COMPARISON OF FAULT-TOLERANT MEMORIES IN SRAM-BASED FPGAS

SRAM-based field programmable gate arrays (FPGAs) are a popular option for space-based applications because of their flexibility, reprogrammability, and low application development costs. They can be reprogrammed while in orbit to adapt to changing mission needs or correct design errors. However SRAM-based FPGAs are inherently sensitive to the effects of faults caused by high-energy particles. These upsets, also called single event upsets (SEUs), can occur in FPGA user memory bits or in FPGA configuration bits (which control user logic and routing). A lot of research and experimentation has identified techniques to make SRAM-based FPGA logic and routing reliable in the presence of SEUs [22–24, 98]. But there has been limited research into the reliability of FPGA user memories.

Memory elements are an essential part of FPGA designs. FPGA memory elements include block RAMs (BRAMs), LUTRAMs, and SRLs. BRAMs provide single or dual-ported, wide, deep memories. LUTRAMs are smaller, faster memories built of FPGA LUTs that provides better access to memory contents. SRLs are shift registers built efficiently out of FPGA LUTs.

Each of these memories are useful in FPGA designs used for space-based applications, and thus require protection from SEUs. Previous memory reliability studies have focused only on BRAM protection [99] using TMR [45] and scrubbing [65]. But in addition to protecting BRAMs, fault-tolerant techniques must be used to protect LUTRAMs and SRLs.

There are a variety of techniques that could provide fault-tolerance for each of the FPGA memory elements. Triple modular redundancy (TMR) is a popular technique for FPGA logic [22–24, 98] and BRAMs [99]. TMR requires a triplication of all resources and the addition of voting logic. Thus the area and power costs of TMR are at least 3x [100]. Other reliability techniques such as duplication with an error detection code (EDC) which have been applied to FPGA logic [24, 64, 98] may provide adequate protection at a reduced cost. Application specific integrated
circuit (ASIC) memory protection techniques such as error control coding (ECC) [64,66] may also provide less expensive protection.

This Appendix compares the effectiveness and cost of different fault-tolerant techniques for SRAM-based FPGA memories (BRAMs, LUTRAMs, and SRLs). The following fault-tolerant techniques are compared: TMR, interlaced parity [34] with duplication, complement duplicate (CD) [34] with duplication, single error correction/double error detection (SEC/DED) [34], and SEC/DED with duplication [64]. Additionally, each of these reliability techniques is augmented with the application of scrubbing [65]. The reliability of each of these techniques is measured in terms of single-bit upset sensitivity and critical failures. The cost is measured in terms of area. Despite the reduced significance of area as a cost in ASIC design, the cost of area is still significant for designs implemented in FPGAs. Performance is not considered as a cost since the focus of this paper is on FPGA reliability and not on high-speed fault-tolerant technique implementations.

Figure B.1: Upsetting a BRAM write enable can cause entire contents to be overwritten - resulting in a critical failure.

This Appendix begins by identifying how upsets can affect FPGA memories. Next the fault-tolerant techniques used in this study are introduced. A discussion on scrubbing in each of the different FPGA memory structures follows. Finally, the study results show that although there are some cheaper alternatives to TMR, the most reliable way to protect SRAM-based FPGA memories is with TMR and scrubbing.

B.1 Fault Mechanisms in Memories

To motivate the need for memory protection techniques, this section considers the types of problems that SEUs can cause in FPGA memories. These problems are measured in terms of
*sensitive bits* and *critical failures*. Problems are reduced with the application of memory reliability techniques, however they are rarely eliminated. This section also discusses how sensitivity and critical failures affect each type of FPGA memory (BRAMs, LUTRAMs, and SRLs) so that the effectiveness of each fault-tolerant technique can be evaluated.

Both ASICs and FPGAs are susceptible to SEUs in their user memories, but unlike ASICs, SRAM-based FPGAs are also vulnerable to faults within their configuration memory. FPGA configuration memory controls user logic and routing. This means that, unlike ASICs [71], FPGAs are also subject to faults in the logic and routing protecting the memory bits. So the goal of FPGA memory protection techniques is to prevent faults caused by upsets in either user memory content, or upsets in the logic and routing protecting the memory.

The effectiveness of different fault-tolerant techniques is measured, first in terms of sensitivity to single-bit upsets. A *sensitive* bit refers to a user memory content bit or an FPGA configuration memory bit that, when upset, causes erroneous memory output. Clearly, in the absence of any memory protection technique, every memory bit is sensitive. Every fault-tolerant technique used in this study *eliminates* sensitive bits caused by single-bit upsets in user memory. But to protect memory bits, added logic is required, which is itself sensitive. Thus effective fault-tolerant techniques must remove sensitivity in both the memory content and memory-protecting logic and routing.

In addition to sensitivity, the effectiveness of the memory reliability techniques is measured in terms of the number of *critical failures* caused by upsets in the memory bits and FPGA configuration memory bits. A critical failure is an upset whose effects are lasting, and can only be repaired through device reconfiguration. An upset that causes a critical failure is also considered sensitive, but not all sensitive bits cause a critical failure.

Critical failures are very similar to *persistent* errors [101] in FPGA logic. Persistent errors are SEU-induced errors that cannot be repaired without a global system reset. But unlike FPGA logic, FPGA memories are unaffected by reset signals. Thus critical failures refer to upsets whose effects cannot be repaired without FPGA reconfiguration.

It is possible for the effects of a critical failure to be overcome in user memories when the memory acts as a RAM. So to declare that an SEU-induced fault has caused a critical failure, a window of time must be defined to observe that the effects of the SEU have not been repaired.
Corrupted memory locations can be naturally overwritten with new data, repairing upsets caused by an upset. The likelihood of this happening depends on the probability of writing to a given memory location, as well as the size of the critical failure window of observation. Thus it is more likely to observe critical failures in larger memories like BRAMs, which have a smaller probability of writing to any given memory location than smaller memories like LUTRAMs or SRLs.

B.1.1 Upsets in BRAMs

A critical failure will usually occur in an unprotected memory when the memory’s input port signals are upset. For example, Figure B.1 shows how upsetting the write enable port on a BRAM can have disastrous consequences. Consider a BRAM that is being used as an instruction memory for a softcore processor [102]. When acting as an instruction memory, the BRAM behaves like a ROM. In other words, the write enable port should never be active. Figure B.1 shows that if an SEU causes the write enable port to become active, the memory contents pointed to by the address port will be overwritten with the value on the data input port. If the address port steps through memory (as is common for an instruction memory), the contents of the entire memory can be overwritten, destroying the entire program. When no reliability techniques are present, the only way to recover from this kind of upset is by taking the FPGA off-line and reconfiguring it.

B.1.2 Upsets in LUTRAMs

Like BRAMs, critical failures can be caused in LUTRAMs when any of the input port signals are upset. However the consequences of upsetting the LUTRAM write enable port might not be as catastrophic as upsetting a BRAM write enable port. Figure B.2 shows a 16 entry, 16-bit wide LUTRAM. If each of the 16 LUTs that make up the 16-bit wide LUTRAM has its own write enable signal, upsetting one of the write enable ports causes only one bit in one of the 16 words to be upset. Even if the LUTRAM address signal increments, only one bit in each of the 16-bit words is upset. Memory reliability techniques can detect and sometimes correct a single-bit error in a memory word, so this scenario is less severe than upsetting the BRAM write enable port.

However it is possible that the consequences of upsetting the write enable port on a LUTRAM are just as severe as upsetting the write enable on a BRAM. If the write enable ports of all
16 LUTRAMs are tied to the same write enable signal, upsetting that signal will cause every bit in the 16-bit word to be upset. In this case, as the address signal increments, the entire memory contents are wiped out. Thus it is usually desirable to provide (if possible) separate write enable signals to each LUTRAM bit within a LUTRAM memory. However providing individual write enable signals will often require additional design area.

B.1.3 Upsets in SRLs

Critical failures in SRLs are different than in BRAMs and LUTRAMs. SRLs have no true ROM behavior. Even if they continually cycle through the same memory contents, they have to write their output value back to their input. Thus an upset to any input signal (data input, address,
or clock enable), will cause erroneous output in an unprotected SRL. If the SRL feeds its output back to its input, upsetting input signals is likely to cause a critical error. But if the SRL does not feed its output back to its input, a critical error can still occur by upsetting the clock enable signal. In this study both feedback and non-feedback scenarios are studied for each reliability technique.

In the absence of any memory protection, every FPGA memory bit is sensitive, and every upset to a memory content bit can cause a critical failure. But every memory fault-tolerant technique used in this study removes all critical failures and all sensitive bits due to upsets in the memory content. However the logic and routing required to protect FPGA memories is vulnerable to upsets. Thus, after protecting memory bits, the goal of FPGA memory reliability techniques is to reduce the sensitivity and critical failures in their own logic and routing.

B.2 Memory Fault-Tolerant Techniques

The fault-tolerant techniques that are compared in this study fall under one of three general categories: TMR, ECC, or duplication with EDC/ECC. TMR is a tested and proven reliability technique for FPGA logic [40, 45], and has even been proposed for FPGA memory reliability [103]. Duplication with EDC/ECC has been proposed for FPGA logic [104] as well as for memories [98], but its effectiveness has not been proven, nor have its practical costs been evaluated. ECC techniques work well for ASICs [64, 66], but they may not be practical for FPGAs.

B.2.1 TMR

TMR is the most commonly used reliability technique used to protect SRAM-based FPGAs [45]. This technique simply triplicates the design and votes on the outputs of each triplicated version of the design (Figure B.3(a)). If a fault occurs in one of the triplicated versions of the design, its faulty output will be out-voted by the other two correct designs. However since all logic and routing in an SRAM-based FPGA are susceptible to SEUs, the voter itself and all routing must also be triplicated in order to prevent errors due to a fault in the voter (Figure B.3(b)).

Although triplicated voters and inputs provide better reliability, there are times when only single inputs and voters can be used. For example, a triplicated memory might be part of a non-triplicated system. In that case, the inputs and the outputs cannot be triplicated (Figure B.3(a)).
Figure B.3: Triple modular redundancy (TMR) can be implemented with either a single, or triplicated voters.

But even in a triplicated system there must be a reliable way to bring the triplicated design back to a single design domain. This can be done on a reliable ASIC or radiation hardened device [105], or by using minority voters [45]. Also in a triplicated system, there must be enough I/O for triplicated signals to come in and out of the FPGA. When one or both of these criteria cannot be met, a single voter must be used on the FPGA (Figure B.3(a)). This study investigates the reliability of both TMR with a single voter as well as TMR with triplicated voters.

**B.2.2 Duplication with EDC**

Another proposed FPGA-based reliability technique (Figure B.4) combines duplication with an error detection code (EDC) [34]. Error detection codes encode memory words with redundant bits. The redundancy allows faults to be detected but not corrected (error correction codes can correct and detect errors).

Figure B.4: A memory protected by duplication and with an error detection code (EDC).
This study investigates two different error detection code techniques. First, a 2-bit interleaved parity detection scheme is used [34]. 2-bit interleaved, odd parity has the advantage of detecting any single-bit upset, a large percentage of adjacent-bit upsets, an all-zeros upset, an all-ones upset, and all upsets with an odd number of upsets in at least one of the two parity groups. For a 16-bit memory word, 2-bit parity requires only 12.5% more memory. Even after adding duplication, parity with duplication has a 112.5% memory increase - significantly less than the 200% required by TMR.

The second detection method uses a complement duplicate (CD) scheme [34]. CD detects any single-bit upset, 66% of double-bit upsets, and any multiple adjacent unidirectional upset. CD detects more upsets than 2-bit interleaved odd parity, but requires more code bits. Since CD adds an entirely duplicated memory, it has a 100% increase in memory, which when duplicated turns into a 200% increase. However CD has the advantage that its encoder is simply an inverter, which usually takes up no FPGA area. But its decoder requires more than just an inverter; it also requires a comparator to compare the original memory contents with the inverted contents of the CD memory.

At first glance, it appears that duplication with EDC might provide fault-tolerance at a lower cost than TMR since duplication is used instead of triplication. However, additional logic is required for this technique which is not required for TMR. Added logic is required for memory encoders, decoders, and comparators. This added additional logic is comparable in size to the logic required by TMR voters.

The memory encoder, decoder, comparator, and selection logic of duplication with EDC acts as a single point of failure just as the single voter with TMR (Figure B.3(a)) acts as a single point of failure. Thus in order to eliminate this single point of failure, this added logic must be triplicated.

Even with triplicated logic, all the input signals still act as single points of failure with this technique (Figure B.4). Also since there is only one output signal, any upsets on that signal will lead to a fault. But if the memory feeds in to a triplicated domain, errors on the output can be removed by triplicating the outputs and applying TMR voters to the outputs. Errors on the input cannot be outvoted this way - even if the inputs come from a triplicated domain. The triplicated
domain will have to merge to a duplicated or single domain before leading to a memory protected by duplication with EDC.

There is no advantage to using duplicated input signals over a single set of input signals. Suppose for example, a duplicated address signal feed into the duplicated memories. If an upset changes the value of one of the two address signals, the memory word that is written into that memory will still be correctly encoded (even though it is written to the wrong address), or the word that is read from memory will still be correctly decoded (even though the wrong word is read) without any indication of an error. It is possible to detect differences between the two address signals by comparing their values, but it is not possible to tell which of the two is incorrect. Thus, in this study, when duplication with EDC is used to protect memories, input signals are not duplicated.

One study uses bitstream scrubbing to correct faults when duplication detects them [106]. When a fault is detected by duplication with compare, design execution pauses until a bitstream scrubber [26] corrects the fault. But pausing design execution can be costly and difficult, therefore this study does not consider the use of duplication with bitstream scrubbing to protect the logic and routing of memory encoders/decoders.

B.2.3 ECC

Error correcting codes (ECCs) are an effective way to protect ASIC memories [64,66] and may also be effective for protecting FPGA memories (Figure B.5). ECCs protect memories by encoding memory words with redundant bits. The redundancy allows faults to be both detected and corrected. If there are k data bits in the original memory word, and c redundant code bits are added to make the encoded word n bits wide \((n = c + k)\), then an \((n, k)\) code is used. The new n-bit encoded memory word is called a code word. This study uses a SEC/DEC encoding to protect a 16-bit memory word. To protect a 16-bit memory word, a \((22, 6)\) SEC/DED code is used, creating a 22-bit code word.

A single-error correction / double-error detection (SEC/DEC) code is an efficient way to correct the effects of a single SEU in a memory word, and detect when there are two errors in the word. When there are more than two errors in a memory word, one of three things happen. Either the erroneous word is an incorrect but valid code word (thus no correction or detection occurs and the output is incorrect), or a single error is falsely corrected (and the output is incorrect), or
a double error is detected. If a double error is reported when there are more than two upsets, the upsets will be caught, otherwise SEC/DED fails.

An ASIC memory reliability study determined that SEC/DED with duplication (i.e. duplication with ECC) is more effective than SEC/DEC by itself, TMR, or duplication with EDC [64]. To see if this holds true for SRAM-based FPGAs, this paper investigates both SEC/DED (Figure B.5) and SEC/DED with duplication (Figure B.4).

Like duplication with EDC, single points of failure can be removed by triplicating the outputs and logic of ECC encoders and decoders. Also like duplication with EDC, single points of failure on the input signals (leading to critical failures) cannot be removed. This study investigates both triplicated and non-triplicated versions of the SEC/DED and duplication with SEC/DED designs.

### B.3 Memory Scrubbing

Memory scrubbing [65] is technique that can be used in conjunction with a fault-tolerant technique to protect a memory from critical failures. Scrubbing refers to the correction of upsets within memories. Without scrubbing, upsets within memories can only be masked with effective fault-tolerant techniques, but the upsets remain in the memory. As upsets within the memory accumulate, it is less and less likely that a fault-tolerant technique will mask the error. Therefore scrubbing is essential to protect memories against SEUs over time. Scrubbing is also the only way to repair the effects of critical failures in memories.

Memory scrubbing is different than bitstream scrubbing [26]. Bitstream scrubbing is a reliability technique that uses readback [107] and partial reconfiguration to protect the logic and
routing (but not the user memory contents) of FPGA designs. Bitstream scrubbing uses readback to compare portions of the FPGA bitstream to a *golden* copy of the bitstream. If there is ever a difference, that portion is reconfigured using partial reconfiguration. Alternatively CRC values of a portion of the bitstream are calculated and compared to a known CRC value for that portion. When the CRC values differ, that portion is reconfigured. Bitstream scrubbing is a great tool to help protect against upsets in FPGA logic and routing, but it cannot protect FPGA memories. Since the contents of memories change, there cannot be an a-priori known CRC value or golden content value that can be used for comparison. Thus memory scrubbers must be implemented independently of bitstream scrubbers.

Memory scrubbing refers to the deliberate repairing of memory words by a memory scrubbing module. A scrubber can work either *deterministically* or *non-deterministically*. A non-deterministic scrubber checks memory words only when they are read. This method is only practical for small memories such as LUTRAMs or SRLs, and requires less area than a deterministic scrubber. But non-deterministic scrubbing does not work well with BRAMs, since it would allow a large number of upsets to potentially accumulate if all memory locations are not read on a regular basis. BRAMs use deterministic scrubbing which *regularly* checks all memory locations for upsets. In this study BRAMs are deterministically scrubbed.

### B.3.1 Scrubbing BRAMs

Adding scrubbing to TMR is a common fault-tolerant technique used to protect FPGA BRAMs [65]. Figure B.6 shows the TMR BRAM scrubber implemented in this study. Although this subsection discusses how a TMR scrubber protects BRAMs, most of the concepts also apply to ECC BRAM scrubbers and duplication with EDC/ECC BRAM scrubbers. This study adds scrubbing to all three of these memory fault-tolerant techniques.

In order to apply scrubbing to BRAMs, an additional memory port is required. Thus if a design uses single-ported memories, the addition of scrubbing turns them into dual-ported memories. FPGA BRAMs can only be used as single or dual-ported memories [107], thus if a memory is already being used as a dual-ported memory, scrubbing may not be viable. This study assumes the use of single-ported BRAMs. Thus scrubbing is easily provided with the use of the second port.
To properly apply scrubbing to FPGA memories, there are implementation details that must be considered. First in order for every BRAM address to be scrubbed in a deterministic manner, a triplicated counter cycles through the entire address space of all three BRAMs, continually scrubbing their contents. It is essential that this counter be triplicated in a reliable way [22] so that the memory location being scrubbed receives the correct data. BRAM contents are only scrubbed when there is an upset detected in one of the BRAM words.

For ECC BRAM scrubbers and duplication with EDC/ECC scrubbers, when an error is detected at a memory address pointed to by the triplicated scrub address counter, instead of only scrubbing that memory location, the entire BRAM contents are scrubbed. Unlike the TMR BRAM scrubber, the entire BRAM contents are scrubbed because not all upsets are detectable by these other fault-tolerant techniques. Although these fault-tolerant techniques are guaranteed to detect all single-bit errors, they are not guaranteed to detect all multi-bit errors (except for duplication with CD). Thus if an all zeros, or all ones error occurs at a memory location (which can occur if the write enable signal is upset) it may not be caught. But if an upset is detected at some other
memory address, the all zeros or all ones error will be scrubbed out since the entire BRAM contents are scrubbed when any error is detected.

The next implementation detail concerns the tripling of BRAM write enable signals. The logic controlling the assertion of each of the triplicated (or duplicated) BRAM write enable signals must be completely separate from each other. This detail provides protection against SEUs causing critical failures (Figure B.1). If the same logic controls the write enable signals of two BRAMs, then a single SEU could overwrite the contents of both BRAMs. Scrubbing would then overwrite the third BRAM with the invalid contents in the other two BRAMs. On the other hand, when the write enable logic of each BRAM is kept separate, if an SEU causes a critical failure in one of the BRAMs, the effects of the critical failure will be corrected with scrubbing, and the effects of the critical failure are never felt.

Finally, BRAM Scrubbers must also consider address conflicts. An address conflict arises when the BRAM scrubber attempts to scrub the memory contents at an address that is already being accessed by the other BRAM port. Some FPGA architectures will allow the scrubber to scrub the memory contents if the other port is simply performing a read at that address [107]. But an address conflict always occurs if the other port is performing a write at the same address as the scrubber. When this happens, the write takes precedence since it will repair that memory location.

B.3.2 Scrubbing LUTRAMs

In this study LUTRAMs are non-deterministically scrubbed. Figure B.7 shows an example of a duplication with EDC/ECC LUTRAM scrubber. Although only the duplication with EDC/ECC scrubber is shown, the design details of this non-deterministic scrubber also apply to TMR LUTRAM scrubbers and ECC LUTRAM scrubbers.

LUTRAM memory locations are only scrubbed when they are read during normal design execution (non-deterministic scrubbing). Unlike BRAM deterministic scrubbers, all LUTRAM locations are not regularly scrubbed. This means that the scrubbing logic is smaller, but it also means that upsets could accumulate in memory. But non-deterministic scrubbing is realistic for LUTRAMs since there are only a small number of memory locations. In other words there is little chance that upsets will collect in memory locations before they are either read and scrubbed, or repaired by a memory write.
Figure B.7: A LUTRAM protected by duplication with EDC/ECC and scrubbing.

Just like BRAM scrubbers, in order for LUTRAMs to be scrubbed an additional memory port is required. In order to simplify this study, only single-ported LUTRAMs are considered. Thus the second port is available for a scrubbing module. Although scrubbing may be possible without the need of the second port, it facilitates the logic required for scrubbing.

Also like BRAM scrubbers, the logic controlling the LUTRAM write enable signal must be properly designed in order to prevent critical failures. The write enable signals leading to each duplicated domain (or triplicated domain in the case of TMR LUTRAM scrubbers) must be completely independent. Further, within a domain it is possible to have either a single write enable signal for the entire LUTRAM memory, or a write enable signal for each single-bit LUT of the memory (Figure B.2). Some fault-tolerant techniques such as TMR and CD with duplication do not need more than just a single write enable signal. Other techniques such as parity with duplication have no way of determining which write enable signals to assert, and therefore use a single write enable signal. Finally some techniques such as SEC/DED or SED/DED with duplication can take full advantage of multiple write enable signals. In this study all LUTRAM scrubbers use a single write enable signal except for the SEC/DED scrubbers. SEC/DED with duplication works well enough with a single write enable signal that it is not worth the extra area to have multiple signals.

Unlike BRAM scrubbers, there is no need to check for address conflicts. Although LUTRAMs are dual-ported, one port is used for reading and one is used for writing. Thus there can
never be a write conflict. Also, LUTRAMs do not suffer from the same problem that BRAMs do in some FPGA architectures when a read and a write are requested from the same port. This means that there is less logic required to build a LUTRAM scrubber.

### B.3.3 Scrubbing SRLs

At first glance, scrubbing SRLs seems difficult since, unlike LUTRAMs and BRAMs, there are no dual-ported SRLs. This means that deterministic, scrubbing cannot be performed the way it can be for BRAMs or LUTRAMs. Also, to access any SRL value other than the one available on the output, the SRL contents must be cycled through. Despite these difficulties, it is possible to build a relatively simple non-deterministic scrubber (Figure B.8).

Figure B.8 shows an example of an EDC with duplication SRL scrubber. When the EDC in one of the two SRLs detects an error, the contents of the other SRL are used to scrub the faulty SRL using a clock that is $n + 1$ times faster than the normal clock (when $n$ is the length of the SRL). The first $n$ cycles scrub the SRL contents, and the extra cycle is used to perform a write when the clock enable signal is high. The duplicated SRL also feeds its own data back into itself in order to preserve its contents. A TMR or ECC SRL scrubber can be similarly implemented.

![An SRL protected by duplication with EDC and scrubbing.](image)

Although the ability of the scrubber shown in Figure B.8 (as well as a TMR and ECC version) have been verified, this study finds that there is no need to scrub SRLs. The natural
operation of SRLs which causes new values to be written to the SRL is good enough to protect SRLs from critical errors. Even when not written to on every clock cycle, SRL scrubbing is unnecessary.

### B.4 Memory Fault-Tolerance Comparisons

This section compares the fault-tolerance of FPGA memories protected with different reliability techniques, and evaluates each of their costs. The fault-tolerant techniques applied include TMR, duplication with EDC (parity and CD), ECC (SEC/DED) and ECC with duplication. Fault-tolerance is measured in terms of the number sensitive bits, and the number of critical failures. The cost of each fault-tolerant technique is measured in terms of FPGA slices, and BRAMs (where applicable).

Fault-injection tests are run on SEAKR’s XRTC board. The XRTC board contains 3 FPGAs: the configmon, the funcmon, and the DUT (Figure B.9). The DUT (design under test) holds the mitigated FPGA memory design that will injected with faults. The DUT is a Xilinx Virtex4 FPGA. The funcmon (functional monitor) holds a golden copy of the memory design and compares the outputs of the golden and DUT memories. The configmon (configuration monitor) is responsible for injecting the faults into the DUT bitstream, and communicating with a host through a USB bus. The configmon configures both the funcmon and DUT FPGAs and reports sensitive bits and critical failures back to the host.

![Figure B.9: Fault-injection is performed using SEAKR’s XRTC board.](image-url)
In order to determine the sensitivity of each design, the configmon design upsets each bit of the DUT bitstream. After a bit is upset, the configmon allows the DUT to cycle \( x \) times (where \( x \) is a variable). During the \( x \) cycles, the memory on the DUT and the golden memory on the funcmon are fed pseudo-random inputs. If the output of the memory on the DUT ever differs from the output of the golden memory, an error signal is sent back to the host, indicating which bitstream bit is sensitive. Every memory address is compared in order to ensure that all errors are caught. After \( x \) cycles, the bitstream bit is repaired, and the next bit is upset. For all BRAM reliability tests, \( x = 5000 \). For all LUTRAM and SRL tests, \( x = 200 \).

Critical failures are observed by allowing the DUT to cycle an extra \( x \) cycles after repairing a bit, and before upsetting the next bit. Before these second \( x \) cycles, the funcmon allows the DUT to first cycle \( y \) times (where \( y \) is a fraction of \( x \)) before comparing the DUT and golden memories. These \( y \) cycles provide a window of opportunity for upsets in the memory to be repaired. If, after the \( y \) cycles, any differences between the DUT and golden memories are observed, a critical failure is indicated to the host. After a critical failure is observed, the DUT is reconfigured, and the golden memory is reinitialized. For all BRAM tests \( y = 2000 \), and for all LUTRAM and SRL tests \( y = 50 \).

When comparing reliability techniques it is important to have a baseline to compare effectiveness and cost. For each FPGA memory type (BRAM, LUTRAM, and SRL) the sensitivity, number of critical failures, and area cost of an unprotected memory is provided to compare the values of each fault-tolerant technique against.

To test the effectiveness of each of the fault-tolerant techniques on each of the FPGA memory types, different sets of tests are defined. First, all of the reliability techniques are applied to BRAMs and LUTRAMs while acting as a ROM, and then again while acting as a RAM. For SRLs, all of the fault-tolerant techniques are applied, first when the SRL outputs are fed to its inputs, and then when random inputs are fed to the SRL. For all three memory types, comparisons are done first among non-triplicated, non-scrubbing techniques, and then among triplicated, non-scrubbing techniques. For BRAMs and LUTRAMs, comparisons are also done among triplicated scrubbing techniques.
B.4.1 BRAM Fault-Tolerant Technique Comparisons

For each BRAM reliability design, 16-bit wide memory words are used. With 16-bit memory words, a Xilinx Virtex4 BRAM can hold 1024 words. The Virtex4 BRAM will actually hold 1024 18-bit words, but only 16 bits of the 18 are used in the baseline design. The extra two bits per word allow parity to be added for no additional area cost. However other error coding techniques such as CD and SEC/DED will require additional BRAM area.

The cost for each BRAM protection technique (with and without scrubbing) is shown in Table B.1 in terms of BRAM area. The table shows the number of memory bits that each technique requires, and compares that cost to the original baseline design. Unfortunately, since BRAMs contain a fixed number of memory words, very few reliability techniques can make efficient use of the BRAM area. For example, the SEC/DED encoding technique causes a 16-bit memory word to grow to 22-bits. Since there is no natural way to store 22-bit words in a Xilinx BRAM, that 22 bits is broken into two 11-bits halves, and each half is stored in a separate BRAM. Since only 11-bits are being used in each word of the BRAM, the other 7 bits per word are wasted. Only the parity, CD, and TMR techniques use BRAMs efficiently.

Table B.1: The cost of fault-tolerance in terms of BRAM bits.

<table>
<thead>
<tr>
<th>Design</th>
<th>No Scrubbing</th>
<th></th>
<th>Scrubbing</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BRAM Bits</td>
<td>BRAMs</td>
<td>BRAM Bits</td>
<td>BRAMs</td>
</tr>
<tr>
<td>Original</td>
<td>16384</td>
<td>1</td>
<td>16384</td>
<td>1</td>
</tr>
<tr>
<td>Parity Dup</td>
<td>34816</td>
<td>2.1x</td>
<td>36864</td>
<td>2.3x</td>
</tr>
<tr>
<td>CD Dup</td>
<td>49152</td>
<td>3x</td>
<td>65536</td>
<td>4x</td>
</tr>
<tr>
<td>SEC/DED</td>
<td>22528</td>
<td>1.4x</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>SEC/DED Dup</td>
<td>38912</td>
<td>2.4x</td>
<td>45056</td>
<td>2.8x</td>
</tr>
<tr>
<td>TMR</td>
<td>49152</td>
<td>3x</td>
<td>49152</td>
<td>3x</td>
</tr>
<tr>
<td>Xilinx ECC</td>
<td>23552</td>
<td>1.4x</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Table B.2 compares the fault-tolerant techniques for the case when no triplication is performed in the ECC or duplication with EDC/ECC techniques, and when the TMR design has only a single voter (Figure B.3(a)). In addition to these techniques, Xilinx’s ECC-protected BRAM block [107] is included for comparison. None of the designs in Table B.2 implement scrubbing.
The table includes the area cost in slices, as well as the number of sensitive bits and critical failures for each technique.

Table B.2: Non-triplicated, non-scrubbing reliability and cost comparisons for BRAMs.

<table>
<thead>
<tr>
<th>Design</th>
<th>ROM</th>
<th>RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Slices</td>
<td>Sensitive Bits</td>
</tr>
<tr>
<td>Original</td>
<td>0</td>
<td>16725</td>
</tr>
<tr>
<td>Parity Dup</td>
<td>12</td>
<td>674</td>
</tr>
<tr>
<td>CD Dup</td>
<td>12</td>
<td>329</td>
</tr>
<tr>
<td>SEC/DED</td>
<td>21</td>
<td>949</td>
</tr>
<tr>
<td>SEC/DED Dup</td>
<td>34</td>
<td>1073</td>
</tr>
<tr>
<td>TMR (1 Voter)</td>
<td>16</td>
<td>484</td>
</tr>
<tr>
<td>*Xilinx ECC</td>
<td>0</td>
<td>340</td>
</tr>
</tbody>
</table>

*Virtex4 ECC BRAM contents cannot be initialized and is therefore not useful as a ROM.

Since each technique prevents all single-bit upsets within the memory content, all of the techniques have significantly fewer sensitive bits than the original unprotected BRAM. When treated as a ROM, most of the critical failures are due to upsets in the signals leading to the write enable signals of the BRAMs. When treated as a RAM, most of the critical failures are due to upsets in logic and signals leading to input ports. Table B.2 shows that when treated as a ROM, the parity with duplication, and CD with duplication techniques provide the best protection for the lowest cost. Although the Xilinx ECC BRAM provides the lowest cost (no added slices needed), the Virtex4 version of the primitive cannot be initialized, which makes it useless as a ROM. When a BRAM is treated as a RAM, TMR with a single voter provides the best protection at the lowest cost.

In general, triplicating the logic in the duplication with EDC/ECC and ECC techniques, and triplicating the TMR voters (Figure B.3(b)) improves the reliability of memory protection. Table B.3 compares the techniques when triplication is added to the designs from Table B.2. This table shows that when no scrubbing is used TMR provides the best protection at the lowest cost. The sensitivity in all of these designs is due, mainly, to upsets in the input logic and signals.
Table B.3: Triplicated, non-scrubbing reliability and cost comparisons for BRAMs.

<table>
<thead>
<tr>
<th>Design</th>
<th>ROM Slices</th>
<th>Sensitive Bits</th>
<th>Critical Fail</th>
<th>RAM Slices</th>
<th>Sensitive Bits</th>
<th>Critical Fail</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>0</td>
<td>16725</td>
<td>17</td>
<td>0</td>
<td>17103</td>
<td>306</td>
</tr>
<tr>
<td>Parity Dup</td>
<td>37</td>
<td>611</td>
<td>9</td>
<td>41</td>
<td>655</td>
<td>495</td>
</tr>
<tr>
<td>CD Dup</td>
<td>40</td>
<td>503</td>
<td>5</td>
<td>56</td>
<td>1050</td>
<td>824</td>
</tr>
<tr>
<td>SEC/DED</td>
<td>66</td>
<td>548</td>
<td>34</td>
<td>100</td>
<td>845</td>
<td>736</td>
</tr>
<tr>
<td>SEC/DED Dup</td>
<td>102</td>
<td>544</td>
<td>23</td>
<td>136</td>
<td>1001</td>
<td>900</td>
</tr>
<tr>
<td>TMR (3 Voters)</td>
<td>34</td>
<td>135</td>
<td>10</td>
<td>34</td>
<td>494</td>
<td>110</td>
</tr>
</tbody>
</table>

Table B.3 shows that even with TMR and triplicated voters and inputs, not every sensitive bit is eliminated. Some of the sensitivities in each fault-tolerant technique design in all of the tables correspond to single FPGA configuration bits that affect multiple routes [24]. Although this study does not focus on eliminating these kinds of sensitivities, in order to reduce them the memory elements in each design are hand placed. Even though hand placing the memories reduces sensitive bits which affect multiple routes, the sensitivities reported in every table will include these kind of sensitive bits.

In order to reduce, or even eliminate critical errors, scrubbing is required. Table B.4 compares different reliability techniques that implement scrubbing. The logic and voters of all of these designs are also triplicated. There is no SEC/DED scrubber in this table since SEC/DED on its own cannot overcome critical failures due to upsets on the write enable port (Figure B.1). This table leaves not doubt that TMR with scrubbing provides the best protection for BRAMs.

**B.4.2 LUTRAM Fault-Tolerant Technique Comparisons**

Like BRAMs, each fault-tolerant LUTRAM technique design uses 16-bit memory words. But instead of 1024 memory locations, each design uses a 16 entry LUTRAM. Unlike BRAMs, LUTRAMs do not suffer from inefficient memory use, and all area costs are determined by slice count.
Table B.4: Triplicated, scrubbing reliability and cost comparisons for BRAMs.

<table>
<thead>
<tr>
<th>Design</th>
<th>ROM</th>
<th>RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Slices</td>
<td>Sensitive</td>
</tr>
<tr>
<td>Original</td>
<td>0</td>
<td>16725</td>
</tr>
<tr>
<td>Parity Dup</td>
<td>106</td>
<td>568</td>
</tr>
<tr>
<td>CD Dup</td>
<td>112</td>
<td>393</td>
</tr>
<tr>
<td>SEC/DED Dup</td>
<td>181</td>
<td>525</td>
</tr>
<tr>
<td>TMR (3 Voters)</td>
<td>103</td>
<td>1</td>
</tr>
</tbody>
</table>

The reliability and cost for the non-triplicated, non-scrubbing fault-tolerant LUTRAM techniques are shown in Table B.5. The TMR technique used in these comparisons does not triplicate voters (Figure B.3(a)). When treated as a RAM, all critical failures are repaired through natural memory writes. But these critical failures cannot be repaired when treated as a ROM. When treated as a ROM or a RAM, CD with duplication provides the best reliability at a reasonable cost.

Table B.5: Non-triplicated, non-scrubbing reliability and cost comparisons for LUTRAMs.

<table>
<thead>
<tr>
<th>Design</th>
<th>ROM</th>
<th>RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Slices</td>
<td>Sensitive</td>
</tr>
<tr>
<td>Original</td>
<td>8</td>
<td>1113</td>
</tr>
<tr>
<td>Parity Dup</td>
<td>28</td>
<td>341</td>
</tr>
<tr>
<td>CD Dup</td>
<td>36</td>
<td>308</td>
</tr>
<tr>
<td>SEC/DED</td>
<td>32</td>
<td>617</td>
</tr>
<tr>
<td>SEC/DED Dup</td>
<td>53</td>
<td>553</td>
</tr>
<tr>
<td>TMR (1 Voter)</td>
<td>40</td>
<td>448</td>
</tr>
</tbody>
</table>

Triplicating logic and voters (Figure B.3(b)) generally improves the sensitivity of these designs. Table B.6 compares the fault-tolerant techniques when triplication is added to the designs in Table B.5. Again, CD with duplication provides the best reliability at a reasonable cost.
Table B.6: Triplicated, non-scrubbing reliability and cost comparisons for LUTRAMs.

<table>
<thead>
<tr>
<th>Design</th>
<th>ROM</th>
<th>RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Slices</td>
<td>Sensitive</td>
</tr>
<tr>
<td>Original</td>
<td>8</td>
<td>1113</td>
</tr>
<tr>
<td>Parity Dup</td>
<td>64</td>
<td>472</td>
</tr>
<tr>
<td>CD Dup</td>
<td>76</td>
<td>323</td>
</tr>
<tr>
<td>SEC/DED</td>
<td>102</td>
<td>267</td>
</tr>
<tr>
<td>SEC/DED Dup</td>
<td>119</td>
<td>495</td>
</tr>
<tr>
<td>TMR (3 Voters)</td>
<td>56</td>
<td>409</td>
</tr>
</tbody>
</table>

In order to eliminate critical failures, scrubbing and triplication are combined. Triplicated scrubbing designs are compared in Table B.7. Like BRAMs, this table shows that combining TMR and scrubbing provides the best protection with a relatively low cost.

Table B.7: Triplicated, scrubbing reliability and cost comparisons for LUTRAMs.

<table>
<thead>
<tr>
<th>Design</th>
<th>ROM</th>
<th>RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Slices</td>
<td>Sensitive</td>
</tr>
<tr>
<td>Original</td>
<td>8</td>
<td>1113</td>
</tr>
<tr>
<td>Parity Dup</td>
<td>106</td>
<td>126</td>
</tr>
<tr>
<td>CD Dup</td>
<td>136</td>
<td>278</td>
</tr>
<tr>
<td>SEC/DED</td>
<td>110</td>
<td>95</td>
</tr>
<tr>
<td>SEC/DED Dup</td>
<td>255</td>
<td>42</td>
</tr>
<tr>
<td>TMR (3 Voters)</td>
<td>124</td>
<td>2</td>
</tr>
</tbody>
</table>

When treated as a ROM, Table B.7 shows that sometimes critical failures are not eliminated. In the case of the TMR scrubber, the one critical failure occurs at a location where two of the triplicated clock signals run very close to each other. It is likely that upsetting this bit affects both of these clock routes [24]. The other scrubbing designs that have only one or two critical failures might also be caused by a single bit affecting multiple domains. These failures can be removed by constraining the placement and routing of the designs more carefully.
B.4.3 SRL Fault-Tolerant Technique Comparisons

Unlike BRAMs and LUTRAMs, SRLs cannot operate as a ROM. Thus instead of including ROM and RAM modes for each fault-tolerant test, SRLs operate in feedback mode and non-feedback mode. In feedback mode, the SRL outputs are fed back to its inputs. In non-feedback mode random inputs feed into to the SRL. Also unlike BRAMs and LUTRAMs, the address of an SRL determines the length of the SRL. In this study only static addressing is considered. Like the LUTRAM reliability designs, the SRL designs use a 16 entry, 16-bit wide SRL. Thus the address value is fixed for a length of 16 bits.

The non-triplicated, non-scrubbing SRL memory protection comparisons are shown in Table B.8. In feedback mode, all of the designs are susceptible to critical failures caused by upsets to either the input or output logic and routing. In this mode, Table B.8 shows that some techniques actually reduce reliability. In this mode TMR provides the best protection. In non-feedback mode, TMR provides the best protection, but parity with duplications provides relatively good protection for the lowest cost.

Table B.8: Non-triplicated, non-scrubbing reliability and cost comparisons for SRLs.

<table>
<thead>
<tr>
<th>Design</th>
<th>Slices</th>
<th>Sensitive Bits</th>
<th>Critical Fail</th>
<th>Slices</th>
<th>Sensitive Bits</th>
<th>Critical Fail</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>16</td>
<td>1002</td>
<td>584</td>
<td>16</td>
<td>617</td>
<td>0</td>
</tr>
<tr>
<td>Parity Dup</td>
<td>48</td>
<td>1023</td>
<td>578</td>
<td>49</td>
<td>319</td>
<td>0</td>
</tr>
<tr>
<td>CD Dup</td>
<td>68</td>
<td>1080</td>
<td>626</td>
<td>76</td>
<td>336</td>
<td>0</td>
</tr>
<tr>
<td>SEC/DED</td>
<td>51</td>
<td>677</td>
<td>412</td>
<td>51</td>
<td>483</td>
<td>0</td>
</tr>
<tr>
<td>SEC/DED Dup</td>
<td>76</td>
<td>637</td>
<td>458</td>
<td>79</td>
<td>493</td>
<td>0</td>
</tr>
<tr>
<td>TMR (1 Voter)</td>
<td>64</td>
<td>325</td>
<td>113</td>
<td>64</td>
<td>207</td>
<td>0</td>
</tr>
</tbody>
</table>

When triplication is added to the non-scrubbing designs, Table B.9 shows that all critical failures are eliminated when the SRL is in non-feedback mode, regardless of which fault-tolerant technique is used. In feedback mode, all critical failures are eliminated when TMR is used. This is true even though the clock enable port is not always asserted (the clock enable port value is pseudo-randomly determined). Since all critical failures are eliminated, no SRL scrubbing designs
are provided in this study. With triplication in feedback and non-feedback modes, TMR provides the cheapest protection.

Table B.9: Triplicated, non-scrubbing reliability and cost comparisons for SRLs.

<table>
<thead>
<tr>
<th>Design</th>
<th>Feedback Slices</th>
<th>Feedback Sensitive Bits</th>
<th>Feedback Critical Fail</th>
<th>Non-Feedback Slices</th>
<th>Non-Feedback Sensitive Bits</th>
<th>Non-Feedback Critical Fail</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>16</td>
<td>1002</td>
<td>584</td>
<td>16</td>
<td>617</td>
<td>0</td>
</tr>
<tr>
<td>Parity Dup</td>
<td>89</td>
<td>963</td>
<td>796</td>
<td>87</td>
<td>307</td>
<td>0</td>
</tr>
<tr>
<td>CD Dup</td>
<td>102</td>
<td>887</td>
<td>708</td>
<td>116</td>
<td>245</td>
<td>0</td>
</tr>
<tr>
<td>SEC/DED</td>
<td>144</td>
<td>196</td>
<td>195</td>
<td>144</td>
<td>74</td>
<td>0</td>
</tr>
<tr>
<td>SEC/DED Dup</td>
<td>178</td>
<td>199</td>
<td>188</td>
<td>159</td>
<td>159</td>
<td>0</td>
</tr>
<tr>
<td>TMR (3 Voters)</td>
<td>80</td>
<td>0</td>
<td>0</td>
<td>80</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

B.5 Summary

This Appendix compares the reliability and cost of different FPGA memory fault-tolerant techniques. TMR, parity with duplication, CD with duplication, SEC/DED, and SEC/DED with duplication are the techniques used to protect BRAMs, LUTRAMs, and SRLs. Memory scrubbing is also added to each reliability technique. Reliability is measured in terms of the number of sensitive bits in a design, and the number of critical failures. This study defines a critical failure as an upset whose effects can only be repaired by reconfiguring the FPGA. The cost of a reliability technique is measured in terms of area (slices and BRAMs).

This study finds that the best way to protect BRAMs and LUTRAMs is with a TMR scrubber. Despite the potential cost advantages that ECC and duplication with EDC/ECC sometimes have, they have a major weakness that TMR does not suffer from - they all have single points of failure on their input and output port signals. Also this study finds that scrubbing is unnecessary to protect SRLs. When an SRL is used in feedback or non-feedback mode, TMR provides complete protection.
Although TMR or TMR with scrubbing seems to provide the best protection, there are often cheaper alternatives whose reliability is almost as good. This study reveals these reliability-area trade-offs points provided by the different memory protection techniques. For example, a LU-TRAM protected with CD and duplication, whose logic is not triplicated, provides good protection at a low cost. A SEC/DED scrubber with triplicated logic provides better protection at a greater cost, while a TMR scrubber provides the best protection, but at an even greater cost. The trade-off points provided by this study will be valuable when designing for space-based applications, where area and reliability are of concern.
APPENDIX C. IDEAL CHECKPOINTING INTERVALS FOR ASIC-BASED PROCESSORS

Checkpointing is an important technique for processor recovery. When checkpointing is used, it is important to determine how often to perform a checkpoint. The period of time between checkpoints is called the checkpoint interval \( T \). This parameter balances two competing interests. First, checkpointing too frequently creates a large amount of overhead with limited returns on reliability. More frequent checkpointing causes a larger amount of time spent saving processor state instead of executing a program. On the other hand, if checkpointing is done too infrequently, there is a large amount of work to redo when a rollback is performed. Also, the checkpoint data could be corrupted if it is left static for too long. If there was no overhead associated with saving processor state, checkpointing would be done on every clock cycle in order to eliminate the amount of work to redo when a rollback is performed. But since there is a non-zero cost to perform a checkpoint, a balance must be found between added checkpointing overhead and the amount of work to redo when a rollback is required. This appendix discusses two ways of determining an optimal checkpoint interval for ASIC-based processors.

The general concept shown in Figure D.1 for all of the probability models discussed in this section is the same. A program with an execution time of \( S \) is divided into \( n \) equally spaced intervals \( T \), with a checkpoint occurring at the end of each interval. When an upset is detected, the processor rolls back to the most recent checkpoint. \( t_s \) is the time it takes to run a checkpoint, and \( t_r \) is the time it takes to restore checkpoint state. All of the probability models used in this study assume a constant error rate \( \lambda \).

For each of the probability models presented in this study, the following variables are used:

- \( S \): the original execution time of a program without faults or checkpoints.
- \( n \): the number of equally distributed CSCP checkpoints to insert into a program.
- \( T \): the time to complete an interval of the program between CSCP checkpoints \( (T = S/n) \).
• $t_s$: the time to store processor state.

• $t_r$: the time to roll back to the previous checkpoint.

• $\lambda$: constant error rate assuming a Poisson distribution.

When numerical examples are provided, it is assumed (unless otherwise stated) that $S = 10,000$ s, $t_s = 10.73\mu$s, $t_r = 9.33\mu$s, and $\lambda = 1$ error/year. All checkpointing times used in this chapter, including the values for $t_s$ and $t_r$, are actual measured values taken from a softcore processor running on an FPGA with a 33 MHz clock.

### C.1 Simplest 1st Order Approximation of an Optimal Checkpoint Interval

In 1974, a simple first-order probability model was introduced by John Young for approximating the optimal checkpoint interval for a processor [108]. This model provides a very simple way to select a checkpoint interval, but it makes some simplifying assumptions. Although it makes no assumptions about what kind of fault-detection technique is used, it assumes that upsets are always detected with zero detection latency. It also assumes that there is no overhead associated with performing a rollback. Finally, it does not consider that upsets may occur during a retry. Other probability models account for this using a recurrence relation.
Figure C.2 shows a timeline that represents the execution time of a processor. Every $T$ seconds, a checkpoint is taken, which takes $T_c$ seconds to perform. The time interval between upsets is assumed to be $t_f$. When a fault does occur, the amount of time spent re-executing the program after rolling back to the previous checkpoint is assumed to be $t_{re}$. The total amount of overhead due to checkpoint recovery is $t_l$, and the total time lost due to reruns caused by failures and checkpointing is $T_l$. The constant error rate is $\lambda$ and the time between failures (MTBF) is $T_f = 1/\lambda$.

![Timeline showing the terminology for the first-order probability model to select an optimal checkpointing interval for a processor](image_url)

- $L$: the time interval between checkpoints ($L = T + t_s$).
- $i$: a number, representing the current checkpoint interval $iL$. 
- $t_{re}$: the amount of time spent re-executing work after a rollback is performed.
- $t_f$: the time interval between failures ($t_f = t_{re} + iL$).
- $t_l$: the overhead due to checkpointing and rollback ($t_l = i \cdot t_s + t_{re} = t_f - i \cdot T$).
- $T_l$: the total time lost due to reruns caused checkpointing and rollback.
- $T_f$: mean time between failures ($T_f = MTBF = 1/\lambda$). 

145
The probability density function \( P(x) \) for the time interval of length \( x \) between failures is
\[ P(x) = \lambda e^{-\lambda x}. \]
Then the probability that the interval between failures is of length \( t_f \) is
\[ \lambda e^{\lambda t_f} \Delta t, \]
where \( t < t_f < t + \Delta t \). This is also the probability the that the lost time due to failures is of time \( t_f \).
Thus, the total time lost to reruns and checkpointing is:
\[
T_l = \sum_{i=0}^{\infty} E[t_i]
= \sum_{i=0}^{\infty} \int_{iL}^{(i+1)L} t_f P(t_f) dt
= \sum_{i=0}^{\infty} \int_{iL}^{(i+1)L} (t - iT) (\lambda e^{-\lambda t}) dt
= \frac{1}{\lambda} + T \sum_{i=1}^{\infty} \left[ e^{-\lambda (i+1) (T + t_s)} - e^{-\lambda iT} \right]
= \frac{1}{\lambda} + T \left( 1 - e^{-\lambda (T + t_s)} \right) \cdot e^{-\lambda (T + t_s)} \sum_{i=1}^{\infty} i e^{-\lambda (i-1) (T + t_s)}
= \frac{1}{\lambda} \cdot \frac{T \left( 1 - e^{-\lambda (T + t_s)} \right) \cdot e^{-\lambda (T + t_s)}}{(1 - e^{-\lambda (T + t_s)})^2}
= \frac{1}{\lambda} + \frac{T}{(1 - e^{\lambda (T + t_s)})}. \quad \text{(C.1)}
\]

The graph in Figure C.3 shows that there is an optimal value \( T^* \) that minimizes \( T_l \) (the graph uses the timing assumptions previously stated and an error rate of one error per year). To find this value of \( T \), differentiate Equation C.1 with respect to \( T \), equate the result to zero, and solve for \( T \):
\[
\frac{dT_l}{dT} = \frac{1 - e^{\lambda (T + t_s)} - T (-e^{\lambda (T + t_s)})}{(1 - e^{\lambda (T + t_s)})^2}. \quad \text{(C.2)}
\]
Thus,
\[
e^{\lambda (T + t_s)} (1 - \lambda T) - 1 = 0,
e^{\lambda T} e^{\lambda t_s} (1 - \lambda T) - 1 = 0. \quad \text{(C.3)}
\]

146
Figure C.3: Time lost due to checkpointing and reruns as a function of the checkpoint interval length ($T$) in seconds.

Equation C.3 cannot be solved for directly since $T$ occurs inside and outside of the exponent. Thus numerical methods are used find an approximate value $\tilde{T}$. To solve for $T$, $e^{\lambda T}$ is expanded to the second degree, and Equation C.3 becomes:

$$\frac{1}{2} \lambda^2 \tilde{T}^2 = 1 - e^{-\lambda t_s}. \tag{C.4}$$

A simplifying assumption is that since $t_s << T_f$, the second degree approximation of $e^{\lambda t_s}$ can be accurately used. Equation C.4 then becomes:

$$\tilde{T}^2 = 2t_s T_f - t_s^2. \tag{C.5}$$
Since $t_s^2$ is second order compared to $2t_sT_f$ (IE: $t_s^2 << 2t_sT_f$), this term is neglected, and the approximated ideal interval $\bar{T}$ becomes:

$$\bar{T} = \sqrt{2t_sT_f}.$$  \hfill (C.6)

This result is very concise and is often used in practice [109], but because it makes a number of simplifying assumptions, its accuracy can be improved. Table C.1 compares the accuracy of the approximated and actual optimal checkpoint interval (for this probability model) using different error rates. The table shows that despite the simplifying assumptions made, the approximated value is very close to the actual value. The actual optimal value is the minimum value in the curve of Figure C.3, and represents the actual optimal value for this probability model, but does not necessarily represent the true optimal checkpointing interval. The true optimal checkpoint interval requires a more accurate probability model.

Table C.1: Comparison of optimal checkpoint interval ($T^*$) with approximated optimal checkpoint interval ($\bar{T}$).

<table>
<thead>
<tr>
<th>$\lambda$ (errors/year)</th>
<th>$\bar{T}$ (s)</th>
<th>$T^*$ (s)</th>
<th>%Err</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.01</td>
<td>260.1140</td>
<td>255.2614</td>
<td>1.90</td>
</tr>
<tr>
<td>0.1</td>
<td>82.2551</td>
<td>82.0348</td>
<td>0.27</td>
</tr>
<tr>
<td>1</td>
<td>26.0114</td>
<td>26.0367</td>
<td>0.097</td>
</tr>
<tr>
<td>10</td>
<td>8.2255</td>
<td>8.2212</td>
<td>0.052</td>
</tr>
<tr>
<td>100</td>
<td>2.6011</td>
<td>2.6001</td>
<td>0.040</td>
</tr>
<tr>
<td>1000</td>
<td>0.82255</td>
<td>0.8227</td>
<td>0.018</td>
</tr>
<tr>
<td>10,000</td>
<td>0.2601</td>
<td>0.2601</td>
<td>0</td>
</tr>
<tr>
<td>100,000</td>
<td>0.0823</td>
<td>0.0822</td>
<td>0.067</td>
</tr>
<tr>
<td>1,000,000</td>
<td>0.0260</td>
<td>0.0260</td>
<td>0</td>
</tr>
</tbody>
</table>

C.2 Nakagawa’s Ideal Checkpointing Interval Probability Model

Nakagawa presents an improved probability model for finding the optimal checkpointing interval for two processors in a duplication with compare (DWC) detection scheme [43]. This DWC probability model assumes that the processors are not compared on a cycle-by-cycle basis,
but are compared only at distinct points of during program execution. The model improves upon the simple first order probability model in three ways. It does not assume a zero fault detection latency, it accounts for the overhead required to recover a checkpoint, and finally, it uses a recurrence relation to account for the fact that upsets may occur during a retry.

Nakagawa’s probability model uses three well known types of checkpoints [110] for duplicated processors: compare-and-store-checkpoint (CSCP), store-checkpoint (SCP), and compare-checkpoint (CCP). The CSCP checkpoint compares the states of the duplicated processors, and if they match, a checkpoint is taken. If the states do not match, a fault is detected and both processors are rolled back to the previous checkpoint. A SCP checkpoint performs a checkpoint without comparing the states of the processors - there is no potential for rollback. Finally, a CCP checkpoint compares the states of the processors, but does not save any state.

Nakagawa’s probability model has two parts. The first part finds the optimal number of CSCP intervals \((n)\), and the second part finds the optimal number of CCP or SCP intervals \((m)\). Although the number of optimal checkpoint intervals \(n\) and \(m\) are not mutually exclusive, for simplicity, Nakagawa considers them independently. First the ideal CSCP interval is determined, and that result is used to determine the optimal CCP or SCP interval.

C.2.1 Finding the Optimal CSCP Interval

First, consider the model to find the optimal CSCP interval. Figure C.4 shows a time-line that represents the execution of duplicated processors. It is assumed that \(S\) is the amount of time required to execute a program without checkpointing, and without faults. This time is divided equally into \(n\) intervals, and a CSCP checkpoint is placed at each interval. The amount of time between checkpoints spent executing a task is \(T\). The amount of time required to compare the state of the two processors is \(t_{cp}\) and the time required to save processor state is \(t_s\). The constant error rate is \(\lambda\) and the probability that the two processors execute without errors during the interval \([((j - 1)T, jT])\) is \(e^{-2\lambda T}\). In this subsection, it is assumed that \(t_{cp} = 13.21\mu s\).

- \(n\): the number of equally distributed CSCP checkpoints to insert into a program.
- \(T\): the time to complete an interval of the program between CSCP checkpoints \((T = S/n)\).
- \(t_{cp}\): the time to compare processor state.
Figure C.4: Timeline showing the terminology for Nakagawa’s probability model to find the optimal number of CSCP checkpoints.

- $e^{-2\lambda T}$: the probability that two processors execute correctly during the interval $((j-1)T, jT]$.

A task is completed when all $n$ CSCP intervals have successfully completed. The mean execution time $L_1(1)$ of a single interval $((j-1)T, jT]$ is given by the renewal equation:

$$L_1(1) = (T + t_s + t_{cp}) e^{-2\lambda T} + [T + t_s + t_{cp} + t_r + L_1(1)] \left(1 - e^{-2\lambda T}\right). \quad (C.7)$$

This equation considers the time to execute an interval $T$ with two different assumptions: the assumption that no faults occur (with probability $e^{-2\lambda T}$, and the assumption that a fault does occur during the interval (with probability $(1 - e^{-2\lambda T})$). If a fault does occur during the interval $((j-1)T, jT]$, the interval will have to be re-executed, and it is possible that a fault could again occur during the interval. The possibility of an upset on re-execution is accounted for by including $L_1(1)$ in the right side of Equation C.7 (a renewal equation).

Equation C.7 shows that to find an optimal value for $T$, there is a balance that must be established. The probability terms of the equation show that as $T$ goes down, the probability that a failure occurs goes down. But the execution time terms in the equation show that as $T$ goes down, the relative amount of overhead goes up ($T$ in relation to $t_s$ and $t_{cp}$). The ideal value of $T$ will minimize execution time by balancing the checkpointing overhead with the probability that a fault will occur (which causes the interval to have to be rerun).
Equation C.7 is solved to find the mean execution time for a CSCP interval:

\[ L_1(1) = (T + t_s + t_{cp}) e^{2\lambda T} + t_r \left( e^{2\lambda T} - 1 \right). \]  
\[(C.8)\]

The mean execution time \( L_1(n) \) to complete an entire task is:

\[
L_1(n) = nL(1) \\
= n[(T + t_s + t_{cp}) e^{2\lambda T} + t_r \left( e^{2\lambda T} - 1 \right)].
\]  
\[(C.9)\]

But since \( T = S/n \):

\[
L_1(n) = [S + n(t_s + t_{cp} + t_r)] e^{2\lambda S/n} - nt_r.
\]  
\[(C.10)\]

The goal is to find an optimal value of \( n \) that minimizes \( L_1(n) \), and clearly:

\[
L_1(1) = (S + t_s + t_{cp}) e^{2\lambda S} + t_r (e^{2\lambda S} - 1),
\]

\[
L_1(\infty) = \infty.
\]

Thus there exists a number \( n^* \) \((1 \leq n^* < \infty)\) that minimizes \( L_1(n) \).

Also, since \( n = S/T \):

\[
L_1(T) = S\left[ 1 + \frac{t_s + t_{cp} + t_r}{T} \right] e^{2\lambda T} - \frac{t_r}{T}.
\]  
\[(C.11)\]

It can be shown that:

\[
L(0) = \infty,
\]

\[
L(S) = (S + t_s + t_{cp} + t_r) e^{2\lambda S} - t_r.
\]

Thus there exists an optimal \( T^* \) \((0 < T^* \leq S)\) which minimizes \( L_1(T) \).

Figure C.5 shows that there is an optimal checkpoint interval \( T^* \) that will minimize the total execution time for a program (the graph assumes the timing previously stated and assumes an error rate of one error per year). To find the value of \( T^* \) that will minimize Equation C.11, Equation
C.11 is differentiated with respect to $T$ and equated to zero, producing:

$$2\lambda T(T + ts + tcp + tr) - tr(1 - e^{-2\lambda T}) = ts + tcp.$$  \hfill (C.12)

Figure C.5: Total execution time of a program as a function of the checkpoint interval length ($T$) in seconds.

Similar to Equation C.3, Equation C.12 cannot be solved for directly in terms of $T$, since $T$ occurs in inside and outside of the exponential. One way to solve this equation is to approximate the exponential using a Taylor series. Nakagawa instead finds a lower bound by assuming that $tr = 0$. It can be shown that Equation C.12 is an increasing function of $tr$. Thus, when $tr = 0$:

$$\hat{T} = \frac{ts + tcp}{2} \left[ \sqrt{1 + \frac{2}{\lambda(ts + tcp)}} - 1 \right].$$  \hfill (C.13)

Regardless of what method is used to find a value $\hat{T}$, the optimal number of intervals $n^\ast$ can be found as follows:
- If $\tilde{T} < S$, then $n = \lfloor S/\tilde{T} \rfloor$. But, if $L_1(n) > L_1(n+1)$ (using Equation C.10), then $n^* = n + 1$, else $n^* = n$.

- If $\tilde{T} \geq S$, then $n^* = 1$.

Table C.2 compares the optimal and estimated CSCP checkpoint intervals using Nakagawa’s probability model, for a number of different error rates. The table shows that to four decimal places, there is almost no error between the optimal checkpoint interval ($T^*$) and the approximated interval ($\tilde{T}$) for this probability model.

<table>
<thead>
<tr>
<th>$\lambda$ (errors/year)</th>
<th>$n^*$</th>
<th>$\tilde{T}$ (s)</th>
<th>$T^*$ (s)</th>
<th>%Err</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.01</td>
<td>51</td>
<td>194.2870</td>
<td>194.2866</td>
<td>0.000462</td>
</tr>
<tr>
<td>0.1</td>
<td>163</td>
<td>61.4391</td>
<td>61.4390</td>
<td>0.000152</td>
</tr>
<tr>
<td>1</td>
<td>515</td>
<td>19.4287</td>
<td>19.4285</td>
<td>0.00123</td>
</tr>
<tr>
<td>10</td>
<td>1628</td>
<td>6.1439</td>
<td>6.1439</td>
<td>0</td>
</tr>
<tr>
<td>100</td>
<td>5147</td>
<td>1.9429</td>
<td>1.9429</td>
<td>0</td>
</tr>
<tr>
<td>1000</td>
<td>16,277</td>
<td>0.6144</td>
<td>0.6144</td>
<td>0</td>
</tr>
<tr>
<td>10,000</td>
<td>51,473</td>
<td>0.1943</td>
<td>0.1943</td>
<td>0</td>
</tr>
<tr>
<td>100,000</td>
<td>162,795</td>
<td>0.0614</td>
<td>0.0614</td>
<td>0</td>
</tr>
<tr>
<td>1,000,000</td>
<td>515,018</td>
<td>0.0194</td>
<td>0.0194</td>
<td>0</td>
</tr>
</tbody>
</table>

Compared to the first-order model, Nakagawa’s probability model produces shorter checkpoint intervals, thus more checkpoints are taken as a program executes. More frequent checkpoints are taken for two reasons. First, in Nakagawa’s model, upsets are only detected at checkpoints, compared to the first-order model which assumes they are immediately detected. Thus Nakagawa’s model has more work to redo when a rollback is required. To reduce the amount of added work to redo, the checkpoint intervals are smaller. The second reason why Nakagawa’s checkpoint intervals are smaller is that, unlike the first-order model, it assumes that upsets can occur during a retry. An upset on a retry causes even more additional work to redo. A smaller checkpoint interval reduces the amount of work to redo when an upset occurs on a retry.
Table C.3 shows the percentage of additional overhead required for checkpointing, with different error rates. To calculate the overhead percentage, the total program runtime with checkpointing \( L_1(n^*) \) is compared to the runtime without checkpointing \( S \). The table shows that performance overhead associated with checkpointing almost imperceptible. Of course, without checkpointing, the actual runtime would be much longer due to program restarts, caused by upsets. To emphasize how much time checkpointing saves, Table C.3 shows the percentage of time saved by choosing an optimal checkpoint interval rather than having no checkpointing at all \( L_1(1) \). The table shows that as the error rate increases, the mean program execution time without checkpointing approaches infinity. Thus checkpointing is essential.

Table C.3: Percentage overhead due to checkpointing, and percentage of execution time saved with checkpointing compared to without.

<table>
<thead>
<tr>
<th>( \lambda ) (errors/year)</th>
<th>( L_1(n^*) )</th>
<th>% Overhead</th>
<th>( L_1(1) ) (s)</th>
<th>% Time Saved</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.01</td>
<td>10,000.0025</td>
<td>0.000025</td>
<td>10,000.0634</td>
<td>0.0061</td>
</tr>
<tr>
<td>0.1</td>
<td>10,000.0078</td>
<td>0.000078</td>
<td>10,000.6342</td>
<td>0.0063</td>
</tr>
<tr>
<td>1</td>
<td>10,000.0246</td>
<td>0.00025</td>
<td>10,006.3440</td>
<td>0.063</td>
</tr>
<tr>
<td>10</td>
<td>10,000.0779</td>
<td>0.00080</td>
<td>10,063.6211</td>
<td>0.63</td>
</tr>
<tr>
<td>100</td>
<td>10,000.2464</td>
<td>0.0025</td>
<td>10,654.7380</td>
<td>6.14</td>
</tr>
<tr>
<td>1000</td>
<td>10,000.7793</td>
<td>0.0078</td>
<td>18,855.0529</td>
<td>46.96</td>
</tr>
<tr>
<td>10,000</td>
<td>10,002.4646</td>
<td>0.025</td>
<td>5.6791 x 10^6</td>
<td>99.82</td>
</tr>
<tr>
<td>100,000</td>
<td>10,007.7958</td>
<td>0.078</td>
<td>3.4896 x 10^{31}</td>
<td>100</td>
</tr>
<tr>
<td>1,000,000</td>
<td>10,024.6720</td>
<td>0.25</td>
<td>2.6778 x 10^{279}</td>
<td>100</td>
</tr>
</tbody>
</table>

Despite the improvements this probability model makes over the first-order probability model, its assumptions are not accurate for a softcore processor. But before developing a probability model for softcore processors, Nakagawa’s probability model for ideal CCP checkpoint intervals is considered.

C.2.2 Finding the Optimal CCP Interval

The optimal checkpoint interval \( T \) found in the previous subsection is used in this subsection to determine the optimal CCP interval. Figure C.6 presents a time-line depicting the terminology used to find the ideal number of CCP intervals \( m \). The previously determined CSCP interval
$T$ is used to calculate the ideal CCP interval $T_m$. $T$ is divided equally into $m$ intervals and the state of the two processors are compared at the end of each interval. If the state of the processors do not agree, they are both rolled back to the last CSCP checkpoint. The same terminology used for CSCP checkpoints is used for finding the optimal CCP checkpoint.

![Timeline showing the terminology for Nakagawa’s probability model to find the optimal number of CCP checkpoints.](image)

- $T$: the optimal checkpoint interval previously determined.
- $m$: the number of equally distributed CCPs to insert into a CSCP interval.
- $T_m$: the time to complete an interval of the program between CCP checkpoints ($T_m = T / m$).

The mean execution time $L_2(1)$ for a CSCP interval $((j - 1)T, jT]$ is given by the renewal equation:

$$L_2(m) = e^{-2\lambda m T_m} (m T_m + m t_{cp} + t_s)$$

$$+ \sum_{i=1}^{m-1} \left( \int_{(i-1)T_m}^{iT_m} 2\lambda e^{-2\lambda t} dt \right) [iT_m + iT_{cp} + t_r + L_2(m)]$$

$$+ \int_{(m-1)T_m}^{mT_m} 2\lambda e^{-2\lambda t} dt [mT_m + mt_{cp} + t_s + t_r + L_2(m)].$$
This equation has three terms. The first term represents the time to complete a CSCP interval when no failures occur. When no failures occur, \( m \) CCP intervals \( T_m \) are executed, \( m \) state comparisons \( (t_{cp}) \) are performed, and the state of the processors is saved \( (t_s) \). The second and third terms of Equation C.14 are used to calculate the execution time of a CSCP interval when a fault does occur. When a fault occurs within the interval \( ((i-1)T_m, iT_m] \), \( i \) CCP intervals \( T_m \) are executed, \( i \) state comparisons \( (t_{cp}) \) are performed, a rollback is required \( (t_r) \), and the CSCP interval must be restarted \( (L_2(m)) \). The integral inside the second and third term of Equation C.14 represents the probability that a failure occurs within the \( i^{th} \) CCP interval. Note that integrating the exponential probability density function over the interval \( ((i-1)T_m, iT_m] \) produces the exponential probability distribution function for that interval. The summation in the second term sums probabilities from the first to the \( m-1^{th} \) CCP interval. The final \( m^{th} \) interval is considered in the third term of Equation C.14. This third term is not part of the second term summation since the \( m^{th} \) interval is longer than the previous intervals – it also includes the time to store the state of the processors \( (t_s) \).

Solving Equation C.14 produces a non-recurrence relation form solution for the mean execution time \( L_2(m) \) for a CSCP interval \( ((j-1)T, jT] \):

\[
L_2(m) = t_s e^{2\lambda T_m} + (e^{2\lambda mT_m} - 1)(\frac{T_m + t_{cp}}{1 - e^{-2\lambda T_m}} + t_r). 
\]  
(C.15)

To write Equation C.15 with respect to \( m \), \( T_m \) is substituted with \( T_m = T/m \):

\[
L_2(m) = t_s e^{2\lambda T/m} + (e^{2\lambda T} - 1)(\frac{T/m + t_{cp}}{1 - e^{-2\lambda T/m}} + t_r), 
\]  
(C.16)

which is equal to Equation C.8 for \( m = 1 \). The goal is to find an optimal value of \( m \) that minimizes \( L_2(m) \). It can be shown that:

\[
L_2(1) = (T + t_s + t_{cp})e^{2\lambda T} + t_r(e^{2\lambda T} - 1),
\]

\[
L_2(\infty) = \infty.
\]

Thus there exists a number \( m^* \) \((1 \leq m^* < \infty)\) that minimizes \( L_2(m) \).
In order to find an equation that minimizes a CSCP interval with respect to \( T_m \), the value of \( m \) in Equation C.15 is substituted with \( m = T / T_m \):

\[
L_2(T_m) = t_s e^{2\lambda T_m} + (e^{2\lambda T} - 1)\left(\frac{T_m + t_{cp}}{1 - e^{-2\lambda T_m}} + t_f\right).
\]  
(C.17)

It can be shown that:

\[
L(0) = \infty, \\
L(T) = (T + t_s + t_{cp}) e^{2\lambda T} + t_r e^{2\lambda T - 1}.
\]

Thus there exists an optimal \( T_m^* \) \( (0 < T_m^* \leq T) \) which minimizes \( L_2(T_m) \).

To find the value of \( T_m \) that will minimize Equation C.17, Equation C.17 is differentiated and equated to zero, which produces:

\[
2\lambda t_s (e^{2\lambda T_m} - 1)^2 + (e^{2\lambda T} - 1)[e^{2\lambda T_m} - (1 + 2\lambda T_m)] = 2\lambda t_{cp} (e^{2\lambda T} - 1).
\]  
(C.18)

Although \( T_m \) cannot be solved for directly, it can be approximated using numerical methods. Nakagawa does not endorse a specific approximation technique, but a simple approximation uses a Taylor series expansion. When such an approximation is performed, and the results of the previous subsection are used to provide a value for \( T \), the optimal value for \( m \) is found to always be 1 (for the timing values used in this study). When an approximated \( \tilde{T}_m \) is found, the optimal number of CCP intervals \( m^* \) can be found as follows:

- If \( \tilde{T}_m < T \), then \( m = \lfloor T / \tilde{T}_m \rfloor \). But, if \( L_2(m) > L_2(m + 1) \) (using Equation C.16), then \( m^* = m + 1 \), else \( m^* = m \).

- If \( \tilde{T}_m \geq T \), then \( m^* = 1 \).
APPENDIX D. IDEAL CHECKPOINTING INTERVALS FOR SOFTCORE PROCES- 
SORS

The novel probability models introduced in this chapter build on Nakagawa’s probability model for ideal checkpoints for duplicated processors (Appendix C). Nakagawa’s model uses three well known types of checkpoints [110] for duplicated processors: compare-and-store-checkpoint (CSCP), store-checkpoint (SCP), and compare-checkpoint (CCP). The CSCP checkpoint compares the states of the duplicated processors, and if they match, a checkpoint is taken. If the states do not match, a fault is detected and both processors are rolled back to the previous checkpoint. A SCP checkpoint performs a checkpoint without comparing the states of the processors - there is no potential for rollback. Finally, a CCP checkpoint compares the states of the processors, but does not save any state.

Nakagawa’s probability model has two parts. The first part finds the optimal number of CSCP intervals \( n \), and the second part finds the optimal number of CCP or SCP intervals \( m \). Although the number of optimal checkpoint intervals \( n \) and \( m \) are not mutually exclusive, for simplicity, Nakagawa considers them independently. First the ideal CSCP interval is determined, and that result is used to determine the optimal CCP or SCP interval. These same ideas are used to first determine the optimal number of checkpoints to perform on a duplicated softcore processor. They are also used to determine the optimal checkpoint interval and optimal number of consistency checks to perform on a softcore processor protected with software fault-tolerance.

When checkpointing is used, it is important to determine how often to perform a checkpoint. The period of time between checkpoints is called the checkpoint interval \( T \). This parameter balances two competing interests. First, checkpointing too frequently creates a large amount of overhead with limited returns on reliability. More frequent checkpointing causes a larger amount of time spent saving processor state instead of executing a program. On the other hand, if checkpointing is done too infrequently, there is a large amount of work to redo when a rollback is performed. Also, the checkpoint data could be corrupted if it is left static for too long. If there was no overhead
associated with saving processor state, checkpointing would be done on every clock cycle in order to eliminate the amount of work to redo when a rollback is performed. But since there is a non-zero cost to perform a checkpoint, a balance must be found between added checkpointing overhead and the amount of work to redo when a rollback is required.

The general concept shown in Figure D.1 for all of the probability models discussed in this chapter is the same. A program with an execution time of $S$ is divided into $n$ equally spaced intervals ($T$), with a checkpoint occurring at the end of each interval. When an upset is detected, the processor rolls back to the most recent checkpoint. $t_s$ is the time it takes to run a checkpoint, and $t_r$ is the time it takes to restore checkpoint state. All of the probability models used in this study assume a constant error rate $\lambda$.

![Figure D.1: Program execution timeline with $n$ equally spaced checkpoint intervals.](image)

For each of the probability models presented in this study, the following variables are used:

- $S$: the original execution time of a program without faults or checkpoints.
- $n$: the number of equally distributed CSCP checkpoints to insert into a program.
- $T$: the time to complete an interval of the program between CSCP checkpoints ($T = S/n$).
- $t_s$: the time to store processor state.
- $t_r$: the time to roll back to the previous checkpoint.
• \( \lambda \): constant error rate.

When numerical examples are provided, it is assumed (unless otherwise stated) that \( S = 10,000s \), \( t_s = 10.73\mu s \), \( t_r = 9.33\mu s \), and \( \lambda = 1 \) error/year. All checkpointing times used in this chapter, including the values for \( t_s \) and \( t_r \), are actual measured values taken from a softcore processor running on an FPGA with a 33 MHz clock.

Some of the assumptions that are made in the ASIC-based probability models (Appendix C) do not hold for softcore processors. For example, softcore processors are not restricted in their fault detection like Nakagawa’s model is. Softcore processors are flexible enough to compare processor state on a cycle-by-cycle basis, and therefore do not need an explicit CCS checkpoint to check for faults. This has three implications: first, it means that CCS checkpoints are completely unnecessary, secondly, it means that \( t_{cp} = 0 \), and lastly it means that faults are detected with a latency \( t_d \) after an upset has occurred. This section presents probability models for softcore processors protected using DWC and checkpointing, as well as for softcore processors protected with software fault-tolerance techniques. A probability model for a softcore processor using TMR is not presented since TMR uses roll-forward checkpointing and does not save processor state during program execution.

D.1 Ideal Checkpoint Interval for a Duplicated Softcore Processor

First, consider the optimal checkpoint interval for a softcore processor protected using DWC and checkpointing. Figure D.2 shows an updated time-line compared to Figure C.4. Similar terminology used to describe Nakagawa’s probability model is used to describe the model for duplicated softcore processors. It is assumed that \( S \) is the amount of time required to execute a task when no upset occur, and when no checkpointing is performed. This time is divided equally into \( n \) intervals \( T \), with an SCP checkpoint placed at the end of each interval. A CSCP checkpoint is unnecessary since state comparison is done implicitly on a cycle-by-cycle basis. The time required to perform the SCP checkpoint is \( t_s \). The constant error rate is \( \lambda \), and the probability that the two processors execute without errors during the interval \( ((j-1)T, jT] \) is \( e^{-2\lambda T} \). When an upset does occur in the interval \( ((j-1)T, jT] \), \( T_u \) after the last checkpoint, it is assumed that the detection
latency is $t_d$, and that the time required to perform a rollback is $t_r$. In this subsection, it is assumed that $t_d = 21.21 \mu s$.

Figure D.2: Timeline showing the terminology to find the optimal number of SCP checkpoints for a duplicated softcore processor.

- $n$: the number of equally distributed SCPs to insert into a program.
- $T$: the time to complete an interval of the program between SCP checkpoints ($T = S/n$).
- $T_u$: the time from the last checkpoint to point the upset happens.
- $t_d$: the detection latency of the upset.
- $T_d$: the time from the last checkpoint to the failure detection ($T_d = T_u + t_d$).
- $e^{-2\lambda T}$: the probability that two processors execute correctly during the interval $((j-1)T, jT]$.

Before considering the mean execution time $L_3(1)$ for an SCP interval $((j-1)T, jT]$, a probability model is considered for where within the interval an upset occurs $T_u$. Assuming that an upset does occur within an SCP interval $((j-1)T, jT]$ at a time $T_u$ after the checkpoint at time $(j-1)T$, $T_u$ can be found using a uniform probability distribution. A uniform distribution is used since the upset is equally likely to occur anywhere within the interval. Using a uniform
distribution, the value of \( T_u \) is:

\[
T_d = \int_{-\infty}^{\infty} xf(x)dx
= \int_{0}^{T} \frac{x}{T}dx
= \frac{T}{2}.
\]  

(A.1)

A task finishes when all \( n \) SCP checkpoint intervals are complete. The mean execution time \( L_3(1) \) for a single SCP interval \([(j-1)T, jT]\) is given by the renewal equation:

\[
L_3(1) = (T + t_s)e^{-2\lambda T} + \left[ \frac{T}{2} + t_d + t_r + L_3(1) \right] (1 - e^{-2\lambda T}).
\]  

(A.2)

Similar to Equation C.7, Equation D.2 considers the time required to execute a checkpoint interval when an upset occurs and when an upset does not occur. Equation D.2 differs from Equation C.7 in three ways. First, the value of \( t_{cp} \) is equal to zero. Next, \( T/2 \) is used in the second term instead of \( T \). Finally, Equation D.2 has the additional term \( t_d \).

Also similarly to Equation C.7, Equation D.2 shows that a balance must be found for the ideal value of \( T \). As \( T \) goes up, the probability of a failure goes up. But as the value of \( T \) goes up, the relative amount of checkpointing overhead (\( T \) in relation to \( t_s \)) goes down. The optimal value of \( T \) will balance the added checkpointing overhead with rerun overhead needed when an upset occurs. It is difficult to compare Nakagawa’s model (Equation C.7), to the ideal checkpoint interval for a softcore processor. On one hand, the softcore processor might have a smaller checkpoint interval \( T \) since it has less checkpoint overhead \( (t_{cp} = 0) \). But on the other hand, the cost of an upset is a lot lower. The softcore processor does not have to execute the entire interval before the upset is detected. Overall it is observed that checkpoint interval of a softcore processor is longer (fewer checkpoints) than for a processor under Nakagawa’s model.

Solving Equation D.2 produces:

\[
L_3(1) = \left( \frac{T}{2} + t_d + t_r \right)e^{2\lambda T} - t_d - t_r + t_s + \frac{T}{2}.
\]  

(D.3)
The mean time to complete an entire task is:

\[ L_3(n) = nL_3(1) = n \left[ \frac{T}{2} + t_d + t_r \right] e^{2\lambda T} - t_d - t_r + t_s + \frac{T}{2} \]  \hspace{1cm} (D.4)

But since \( T = S/n \):

\[ L_3(n) = n \left[ \frac{S}{2n} + t_d + t_r \right] e^{2\lambda S/n} + t_s - (t_d + t_r) + \frac{S}{2} \]  \hspace{1cm} (D.5)

To find an equation that minimizes \( L_3(n) \) with respect to \( T \), the value of \( n \) in Equation D.5 is substituted with \( n = S/T \):

\[ L_3(T) = \frac{S}{T} \left[ \frac{T}{2} + t_d + t_r \right] e^{2\lambda T} + t_s - (t_d + t_r) + \frac{S}{2} \]  \hspace{1cm} (D.6)

Plotting the total execution time of a program running on duplicated softcore processors as a function of \( T \) (Figure D.3) shows that there is an optimal checkpointing interval \( T^* \) to minimize the total execution time (the graph assumes the timing values previously stated and uses an error rate of one error per year). To find this optimal value of \( T \), Equation D.6 is differentiated with respect to \( T \) and equated to zero, producing:

\[ \lambda T^2 + 2\lambda T(t_r + t_d) + (t_d + t_r - t_s) e^{-2\lambda T} - t_d - t_r = 0. \]  \hspace{1cm} (D.7)

Since Equation D.7 cannot be solved for directly, a second order Taylor series approximation is used to expand \( e^{-2\lambda T} \approx 1 - 2\lambda T + 2\lambda^2 T^2 \). Applying this approximation to Equation D.7 and simplifying produces:

\[ \left[ \lambda + 2\lambda^2(t_d + t_r - t_s) \right] T^2 + 2\lambda T t_s - t_s = 0. \]  \hspace{1cm} (D.8)

Using the quadratic theorem, there is only valid solution for \( T \):

\[ \tilde{T} = \frac{-\lambda t_s + \sqrt{\alpha + \lambda t_s}}{\beta + \lambda}, \]  \hspace{1cm} (D.9)
where
\[ \alpha = \lambda^2 t_s (2t_d + 2t_r - t_s), \] and
\[ \beta = 2\lambda^2 (t_d + t_r - t_s). \]

With an approximation for the optimal checkpoint interval length \( \tilde{T} \), the ideal number of checkpoint intervals \( n^* \) is found as follows:

- If \( \tilde{T} < S \), then \( n = \lfloor S/\tilde{T} \rfloor \). But, if \( L_3(n) > L_3(n+1) \) (using Equation D.5), then \( n^* = n + 1 \), else \( n^* = n \).
- If \( \tilde{T} \geq S \), then \( n^* = 1 \).

Table D.1 shows the optimal number of checkpoints \( n^* \) for a duplicated softcore processor for a number of different error rates. The table also shows the error in approximating the optimal checkpoint interval. Similar to Nakagawa’s probability model, Table D.1 shows that to four decimal places the approximated optimal checkpoint interval (\( \tilde{T} \)) and the actual optimal checkpoint
interval \((T^*)\) are equal, except at low error rates \((\lambda)\). But even at low error rates the difference between the approximated and optimal checkpoint interval is almost imperceptible.

Table D.1: Comparison of optimal checkpoint interval \((T^*)\) with approximated optimal checkpoint interval \((\tilde{T})\) for a duplicated softcore processor.

<table>
<thead>
<tr>
<th>(\lambda) (errors/year)</th>
<th>(n^*)</th>
<th>(T) (s)</th>
<th>(T^*) (s)</th>
<th>%Err</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.01</td>
<td>54</td>
<td>183.934</td>
<td>183.920</td>
<td>0.0040</td>
</tr>
<tr>
<td>0.1</td>
<td>172</td>
<td>58.163</td>
<td>58.163</td>
<td>0.0033</td>
</tr>
<tr>
<td>1</td>
<td>544</td>
<td>18.383</td>
<td>18.393</td>
<td>0.00051</td>
</tr>
<tr>
<td>10</td>
<td>1719</td>
<td>5.8163</td>
<td>5.8163</td>
<td>0</td>
</tr>
<tr>
<td>100</td>
<td>5437</td>
<td>1.8393</td>
<td>1.8393</td>
<td>0</td>
</tr>
<tr>
<td>1000</td>
<td>17,193</td>
<td>0.5816</td>
<td>0.5816</td>
<td>0</td>
</tr>
<tr>
<td>10,000</td>
<td>54,372</td>
<td>0.1839</td>
<td>0.1839</td>
<td>0</td>
</tr>
<tr>
<td>100,000</td>
<td>171,962</td>
<td>0.0582</td>
<td>0.0582</td>
<td>0</td>
</tr>
<tr>
<td>1,000,000</td>
<td>544,008</td>
<td>0.0184</td>
<td>0.0184</td>
<td>0</td>
</tr>
</tbody>
</table>

Compared to Nakagawa’s probability model, the probability model for duplicated softcore processors has slightly smaller checkpoint intervals. There are two differences between the two probability models which cause the optimal checkpoint interval to be pulled in opposite directions. First, since duplicated softcore processors do not have any overhead associated with comparing processor states \((t_{cp} = 0)\), there is a lower penalty for having more checkpoints. A smaller checkpoint overhead means more checkpoints can be taken while a program executes. The second pull on the optimal checkpoint interval comes from the fact that duplicated softcore processors have a lower average detection latency than hardcore processors. Duplicated softcore processors are not restricted to detecting upsets only at CSCP or CCP checkpoints like hardcore processors are. Softcore processors compare processor state on a cycle-by-cycle basis, producing a smaller average detection latency than hardcore processors. A smaller detection latency translates to less work to redo when a rollback is performed (compare Figures C.4 and D.2). Less work to do when a rollback is performed means that longer checkpoint intervals are tolerated. Thus compared to Nakagawa’s probability model, the model for duplicated softcore processors has a smaller average detection latency, leading to a longer optimal checkpoint interval. But the duplicated softcore processor model also has a smaller checkpoint overhead, which allows for more checkpoints to be
done. The overall result is that the optimal checkpoint interval for duplicated softcore processors is slightly shorter than Nakagawa’s probability model allows.

Table D.2 shows the percentage of overhead associated with checkpointing, for different error rates. The amount of overhead is determined by comparing the amount of time to run the program without checkpointing ($S$) with the amount of time to run the program with the optimal amount of checkpointing ($L_4(n^*)$). The table shows that the checkpointing overhead is almost negligible. The second thing Table D.2 shows is the amount of time that is saved by using checkpointing. The amount of time to run a program with optimal checkpointing ($L_4(n^*)$) is compared to the amount of time to run a program without any checkpointing ($L_4(1)$). When no checkpointing is used, a program has to restart from the beginning when an error occurs. Thus as the error rate increases, the amount of time saved by performing checkpointing approaches infinity.

Table D.2: Percentage overhead due to checkpointing in a duplicated softcore processor, and percentage of execution time saved with checkpointing compared to without.

<table>
<thead>
<tr>
<th>$\lambda$ (errors/year)</th>
<th>$L_3(n^*)$</th>
<th>% Overhead</th>
<th>$L_3(1) (s)$</th>
<th>% Time Saved</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.01</td>
<td>10,000.0012</td>
<td>0.000012</td>
<td>10,000.0317</td>
<td>0.00031</td>
</tr>
<tr>
<td>0.1</td>
<td>10,000.0037</td>
<td>0.000037</td>
<td>10,000.3171</td>
<td>0.0031</td>
</tr>
<tr>
<td>1</td>
<td>10,000.0117</td>
<td>0.00012</td>
<td>10,003.1720</td>
<td>0.032</td>
</tr>
<tr>
<td>10</td>
<td>10,000.0369</td>
<td>0.00037</td>
<td>10,031.8106</td>
<td>0.32</td>
</tr>
<tr>
<td>100</td>
<td>10,000.1167</td>
<td>0.0012</td>
<td>10,327.3690</td>
<td>3.17</td>
</tr>
<tr>
<td>1000</td>
<td>10,000.3689</td>
<td>0.0037</td>
<td>14,427.5265</td>
<td>30.69</td>
</tr>
<tr>
<td>10,000</td>
<td>10,001.1667</td>
<td>0.012</td>
<td>2.8445 x 10^6</td>
<td>99.65</td>
</tr>
<tr>
<td>100,000</td>
<td>10,003.6910</td>
<td>0.037</td>
<td>1.7448 x 10^4</td>
<td>100</td>
</tr>
<tr>
<td>1,000,000</td>
<td>10,011.6874</td>
<td>0.12</td>
<td>1.3389 x 10^2</td>
<td>100</td>
</tr>
</tbody>
</table>

The duplicated softcore processor probability model has a smaller average total program runtime $L_4(n^*)$ compared to Nakagawa’s probability model (compare Tables C.2 and D.2). Even with 6% more checkpoints taken while executing a program, the checkpointing overhead on a duplicated softcore processor is 53% smaller than for a hardcore processor using Nakagawa’s probability model. The lower overhead comes from having less work to redo on a rollback and from having no overhead associated with comparing processor states.
D.2 Ideal Checkpoint Interval for a Softcore Processor Using Software Fault-Tolerance Techniques

When software fault-tolerance techniques are used to protect a softcore processor, there are additional considerations when finding an optimal checkpoint interval. First, the assumption that the processor can recover from all detected faults no longer holds. There are some upsets from which the processor cannot recover, which cause the processor to continually attempt a rollback recovery. When this happens, the FPGA holding the softcore processor design must be reconfigured, and the task must be restarted, not from the last checkpoint, but from the beginning of the task. This type of unrecoverable error is different than silent data corruption (an undetected upset), since the processor can detect when it happens, and thus knows that it must be reconfigured. The probability model that determines the optimal checkpoint interval for a softcore processor protected with software techniques must account for the need for reconfigurations, since performing a reconfiguration is extremely expensive in terms of performance.

The next special case to consider occurs when consistency checks are used as one of the software fault-tolerance techniques protecting the softcore processor. When consistency checks are used, it is important to determine how often to run the consistency checks. Although it was assumed that CCS checkpoints are unnecessary for softcore processors, when consistency checks are used, they can be considered to be a CCS checkpoint, and similar probability models are used to find the ideal consistency check interval.

The probability model used to find ideal checkpoint intervals for a softcore processor has two parts. First, the optimal number of CSCP checkpoint intervals \( n \) is found, taking into account the fact the sometimes the processor has to be reconfigured. Next, this information is used to determine the ideal number of consistency checks \( m \) per CSCP checkpoint interval. Although these two intervals are not mutually exclusive, they are treated independently to simplify the model.

D.2.1 Finding the Optimal CSCP Checkpoint Interval

First, the optimal CSCP checkpointing interval is considered for a softcore processor protected with software fault-tolerance techniques. CSCP checkpoints are used instead of SCP checkpoints since consistency checks act like CCP checkpoints, and a consistency check is performed right before processor state is saved. Figure D.4 shows a time-line similar to that of Figure D.2, but
with reconfigurations taken into account. The same terms used to find the optimal SCP checkpoint interval of a duplicated softcore processor are used when determining the optimal CSCP checkpoint interval for a softcore processor protected with software fault-tolerance techniques. $S$ is the amount of time to execute a task without faults and without checkpoints. This time is divided equally into $n$ intervals of length $T$, with a CSCP checkpoint at the end of each interval. The time required to perform a consistency check is $t_{cc}$, and the time required to save the state of the processor is $t_s$. The constant error rate is $\lambda$, and the probability that the processor executes without an upset during the interval $((j - 1)T, jT]$ is $e^{-\lambda T}$. When an upset does occur in the interval $((j - 1)T, jT]$, at time $T_u$, the detection latency is $t_d$ and the required to perform a rollback is $t_r$. When an upset occurs, a reconfiguration is required with probability $\delta$, and $T_r$ is the time required to reconfigure the FPGA with the softcore processor design. In this subsection it is assumed that $t_d = 212.1\mu s$, $t_{cc} = 13.21\mu s$, $T_r = 5s$, and $\delta = 0.0016$.

![Timeline showing the terminology used to find the optimal number of CSCP checkpoints for a softcore processor protected with software fault-tolerance.](image)

- $n$: the number of equally distributed CSCP points to insert into a program.
- $T$: the time to complete an interval of the program between CSCP checkpoints ($T = S/n$).
• $T_u$: the time from the last checkpoint to point an upset occurs.

• $t_d$: the detection latency.

• $T_d$: the time from the last checkpoint to the failure detection ($T_d = T_u + t_d$).

• $t_{cc}$: the time to perform a consistency check.

• $T_r$: the time to reconfigure the FPGA with the softcore processor design.

• $\delta$: the probability that a reconfiguration is required when a fault occurs.

• $e^{-\lambda T}$: the probability the processor executes correctly during the interval $((j - 1)T, jT]$.

Similar to the duplicated softcore processors, it is assumed that a uniform probability distribution is used to determine when during a CSCP interval an upset occurs ($T_u$). A uniform probability distribution is appropriate since a fault is equally likely to occur anywhere in the CSCP interval. With this assumption, $T_u = T/2$.

Unlike previous optimal checkpoint probability models, the optimal checkpoint interval of a softcore processor protected with software reliability techniques cannot be considered, one CSCP interval at a time. Since reconfigurations are sometimes required, requiring the re-execution of multiple checkpoint intervals, the mean execution time of all of the checkpoint intervals must considered at once. The mean execution time of an entire task $L_4(n)$ is determined by the recurrence relation:

\[
L_4(n) = e^{-\lambda nT} (nT + nt_{cc} + nt_s) \\
+ \sum_{j=1}^{n} \left( \int_{(j-1)T}^{jT} \lambda e^{-\lambda t} dt \right) \left\{ \delta \left[ (j-1)(T + t_{cc} + t_s) + T/2 + t_d + t_r + T_r + L_4(n) \right] \\
+ (1 - \delta) \left[ (j-1)(T + t_{cc} + t_s) + T/2 + t_d + t_r + (n-i+1)L_4(n)/n \right] \right\}.
\] (D.10)

Equation D.10 consists of two main parts. The first part considers the execution time when no upset occurs during the task, and the second term considers the execution time when an upset occurs. When no upsets occur, $n$ intervals ($T$) of the task are executed, $n$ consistency checks ($t_{cc}$) are performed, and state of the processor ($t_s$) is saved $n$ times. When an upset does
occur in the interval \((j-1)T, jT]\), \((j-1)\) program intervals \((T)\) have been executed, and \((j-1)\) CSCP checkpoints have been taken \((t_{cc} + t_s)\). Also, \(T_u + t_d\) of the current CSCP interval has been executed, a rollback is required \((t_r)\), and either the entire program must be re-executed \((L_4(n))\) or only the program starting from the previous checkpoint interval must be executed \(((n-i+1)L_4(n)/n)\), depending on whether a reconfiguration is required. The integral inside the second term of Equation D.10 represents the probability that a failure occurs within the \(i^{th}\) checkpoint interval. The summation in this second term sums the probabilities that an upset occurs over each of the \(n\) intervals.

The overall goal is to find a value of \(T\) that will minimize the total execution time of a task. If the value of \(T\) is too small, excessive amounts of time are spent in CSCP checkpoints, and there is a greater chance that an upset will occur during task execution. The largest cost comes when a reconfiguration is required. The time required to perform a reconfiguration can be up to seven orders of magnitude larger than the time required to perform a CSCP checkpoint. Thus, on one hand a large value of \(T\) will reduce the probability of an upset occurring during the task, which reduces the chance of an expensive reconfiguration. However, on the other hand, a large value of \(T\) means that a larger amount of work has to be rerun when an upset occurs and a reconfiguration is not required. Solving Equation D.10 and substituting \(T = S/n\) produces:

\[
L_4(n) = \frac{(e^{\lambda S} - 1)}{2^n \delta^{(e^{\lambda S}/n - 1)} - (1 - \delta)(e^{\lambda S} - 1)} \left[ S(1 + e^{\lambda S/n}) + 2n(t_{cc} + t_s) + 2n(e^{\lambda S/n} - 1)(t_d + t_r + \delta T_r) \right].
\]  

To find an equation the minimizes the time to complete the task with respect to \(T\), the value of \(n\) in Equation D.11 is substituted with \(n = S/T\):

\[
L_4(T) = \frac{S(e^{\lambda S} - 1)}{2^n \delta^{(e^{\lambda S}/T - 1)} - T(1 - \delta)(e^{\lambda S} - 1)} \left[ S(1 + e^{\lambda S/T}) + 2(t_{cc} + t_s) + 2(e^{\lambda S/T} - 1)(t_d + t_r + \delta T_r) \right].
\]  

Figure D.5 shows a graph of the total program execution time \((L_4(T))\) as a function of the CSCP checkpoint interval \((T)\). The lowest point on the graph represents the optimal value \(T^*\) which minimizes the total program execution time. To find the value of \(T\) that minimizes \(L_4(T)\), Equation
D.12 is differentiated with respect to $T$ and equated to zero. Simplifying the result produces:

\[
(\delta \alpha S - \mu \beta T) \left[ 1 + (1 + \lambda T) e^{\lambda T} + 2\lambda e^{\lambda T} t_o \right] = \left[ \delta (\lambda S + e^{\lambda T}) - \mu e^{\lambda S} - 1 \right] \left[ T(1 + e^{\lambda T}) + 2t_{chk} + 2\alpha t_o \right], \tag{D.13}
\]

where

\[
\begin{align*}
    t_{chk} & = t_{cc} + t_s, \\
    t_o & = t_d + t_r + \delta T_r, \\
    \mu & = (\delta - 1), \\
    \alpha & = (e^{\lambda T} - 1), \text{ and} \\
    \beta & = (e^{\lambda S} - 1).
\end{align*}
\]

Figure D.5: Total execution time of a program on a softcore processor as a function of the checkpoint interval length $T$ (in seconds).
Solving Equation D.13 with respect to $T$ cannot be done without making approximations. Using a Taylor series approximation to expand $e^{-\lambda T} \approx 1 - \lambda T + \lambda^2 T^2 / 2$ produces a very large quadratic equation with respect to $T$. Using the quadratic theorem to solve the equation produces only one valid approximation for $T$:

$$
\tilde{T} = \frac{\phi + \sqrt{\phi^2 - 2\lambda t_{chk}(\rho + S\lambda\delta)[\rho(1 - \lambda t_o) + \phi]}}{\lambda [\rho(\lambda t_o - 1) - \phi]},
$$

(D.14)

where

- $t_{chk} = t_{cc} + t_s$,
- $t_o = t_d + t_r + \delta T_r$,
- $\mu = (\delta - 1)$,
- $\rho = \mu(1 - e^{\lambda S})$, and
- $\phi = S\delta\lambda^2 t_{chk}$.

Having an approximation for the optimal checkpoint interval length $\tilde{T}$, an approximation for the ideal number of checkpoint intervals $n^*$ is found as follows:

- If $\tilde{T} < S$, then $n = \lfloor S / \tilde{T} \rfloor$. But, if $L_4(n) > L_4(n + 1)$ (using Equation D.11), then $n^* = n + 1$, else $n^* = n$.

- If $\tilde{T} \geq S$, then $n^* = 1$.

The optimal number of CSCP checkpoints ($n^*$) for a softcore processor protected with software fault-tolerance techniques is shown in Table D.3 for different error rates. The error between the optimal checkpoint interval ($T^*$) and the approximated optimal checkpoint interval ($\tilde{T}$) is also shown. Similar to the other probability models, The table shows that the error between the approximated and optimal checkpoint interval is negligible.

Table D.3 shows that the checkpoint intervals of a softcore processor with software fault-tolerance are less than half as long as the checkpoint intervals of a duplicated softcore processor (compare with Table D.1). The large difference is due to the enormous reconfiguration cost in the probability model for the softcore processor protected with fault-tolerance techniques. Since the cost of reconfiguration is so high (at least 250,000× more costly than a CSCP checkpoint) it
Table D.3: Comparison of optimal checkpoint interval ($T^*$) with approximated optimal checkpoint interval ($\tilde{T}$) for a softcore processor.

<table>
<thead>
<tr>
<th>$\lambda$ (errors/year)</th>
<th>$n^*$</th>
<th>$T$ (s)</th>
<th>$T^*$ (s)</th>
<th>%Err</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.01</td>
<td>26</td>
<td>388.8863</td>
<td>388.9465</td>
<td>0.015</td>
</tr>
<tr>
<td>0.1</td>
<td>81</td>
<td>122.9766</td>
<td>122.9787</td>
<td>0.0017</td>
</tr>
<tr>
<td>1</td>
<td>257</td>
<td>38.8886</td>
<td>38.8913</td>
<td>0.0071</td>
</tr>
<tr>
<td>10</td>
<td>813</td>
<td>12.2976</td>
<td>12.2976</td>
<td>0</td>
</tr>
<tr>
<td>100</td>
<td>2571</td>
<td>3.8888</td>
<td>3.8888</td>
<td>0</td>
</tr>
<tr>
<td>1000</td>
<td>8133</td>
<td>1.2296</td>
<td>1.2296</td>
<td>0</td>
</tr>
<tr>
<td>10,000</td>
<td>25,734</td>
<td>0.3886</td>
<td>0.3886</td>
<td>0</td>
</tr>
<tr>
<td>100,000</td>
<td>81,383</td>
<td>0.1229</td>
<td>0.1229</td>
<td>0</td>
</tr>
<tr>
<td>1,000,000</td>
<td>257,284</td>
<td>0.0389</td>
<td>0.0389</td>
<td>0</td>
</tr>
</tbody>
</table>

is important to detect as soon as possible if a reconfiguration is required. This leads to shorter checkpoint intervals.

Checkpointing overhead costs, and the cost of not checkpointing are both shown in Table D.4. The overhead associated with checkpointing pales in comparison the cost of not checkpointing. The cost of not checkpointing increases exponentially with the error rate. Thus checkpointing should always be used to protect a softcore processor.

Table D.4: Percentage overhead due to checkpointing, and percentage of execution time saved with checkpointing compared to without.

<table>
<thead>
<tr>
<th>$\lambda$ (errors/year)</th>
<th>$L_4(n^*)$</th>
<th>% Overhead</th>
<th>$L_4(1)$ (s)</th>
<th>% Time Saved</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.01</td>
<td>10,000.0013</td>
<td>0.000013</td>
<td>10,000.0159</td>
<td>0.0015</td>
</tr>
<tr>
<td>0.1</td>
<td>10,000.0041</td>
<td>0.000041</td>
<td>10,000.1586</td>
<td>0.0015</td>
</tr>
<tr>
<td>1</td>
<td>10,000.0149</td>
<td>0.00015</td>
<td>10,001.5858</td>
<td>0.016</td>
</tr>
<tr>
<td>10</td>
<td>10,000.0643</td>
<td>0.00064</td>
<td>10,015.8801</td>
<td>0.16</td>
</tr>
<tr>
<td>100</td>
<td>10,000.3757</td>
<td>0.0038</td>
<td>10,161.0898</td>
<td>1.58</td>
</tr>
<tr>
<td>1000</td>
<td>10,002.7955</td>
<td>0.028</td>
<td>11,865.6881</td>
<td>15.70</td>
</tr>
<tr>
<td>10,000</td>
<td>10,015.0560</td>
<td>0.15</td>
<td>124,154.2235</td>
<td>91.93</td>
</tr>
<tr>
<td>100,000</td>
<td>10,020.1838</td>
<td>0.20</td>
<td>2.9537 x 10^{17}</td>
<td>100</td>
</tr>
<tr>
<td>1,000,000</td>
<td>10,030.9229</td>
<td>0.31</td>
<td>2.5873 x 10^{41}</td>
<td>100</td>
</tr>
</tbody>
</table>
Compared to the duplicated softcore processor, average total program runtime \( L_4(n^*) \) of the processor protected with software fault-tolerance increases at a greater rate as the error rate increases (compare Tables D.2 and D.4). The accelerated increase in total program execution time occurs because as the error rate goes up, the probability of reconfiguring the software protected processor also increases. Compared to Nakagawa’s duplicated processor model, the average program execution times of a software fault-tolerant softcore processor are smaller at lower error rates, but are bigger at larger error rates (compare Tables C.2 and D.4).

D.2.2 Finding the Optimal Consistency Check Interval

The value for the optimal checkpoint interval \( T \) found in the previous subsection is used in this subsection to determine the optimal number of consistency checks \( m \). Figure D.6 shows the time-line depicting the terminology used to find an optimal consistency check interval \( T_{cc} \). \( T \) is divided into \( m \) equally spaced intervals \( (T_{cc}) \) and a consistency check \( (t_{cc}) \) is performed after each interval. When a fault occurs in the interval \(((i-1)T_{cc},iT_{cc}]\), it is detected at consistency check \( iT_{cc} \) with probability \( \epsilon \), and rolled back to the previous CSCP checkpoint. If it is not detected by a consistency check (probability \( 1 - \epsilon \)), it is detected by a different software fault-tolerance technique with a detection latency \( t_d \), and the processor is rolled back to the last CSCP checkpoint. The same terminology used to find the optimal CSCP checkpoint for a softcore processor protected with software techniques, is used to find the optimal consistency check interval. In this subsection it is assumed that \( t_{cc} = 13.21\mu s \), and \( \epsilon = 0.28 \).

- \( T \): the optimal checkpoint interval previously determined.
- \( m \): the number of equally distributed consistency checks to insert into a CSCP interval.
- \( T_{cc} \): the time to complete an interval of the program between consistency checks \( (T_{cc} = T/m) \).
- \( t_{cc} \): the time to perform a consistency check.
- \( \epsilon \): the probability a consistency check detects a fault when an upset occurs.
This study finds that for the value of $\varepsilon$ measured ($\varepsilon = 0.28$), unless the overhead associated with checkpointing is at least $8 \times$ greater than the time required to perform a consistency check, the optimal value of $m$ is always $m^* = 1$. But, in practice the optimal number of consistency checks may be much greater. The value of $t_s$ used in this study ($10.73\mu s$) is the measured time to perform a checkpoint for the LEON3 softcore processor used in this study. The bottleneck for performing a checkpoint is determined by the time it takes to save the main memory contents. If the main memory of a softcore processor is implemented on the FPGA using BRAMs, then that bottleneck is significantly reduced. Regardless of how many BRAMs are used to implement the main memory, the time it takes to save (or restore) memory contents is equal to the time it takes to write (or read) to half of a BRAM, since all of the main memory’s dual-ported BRAMs can be written to (or read from) simultaneously. On the other hand, if the softcore processor’s main memory is located off-chip, then the time it takes to perform a checkpoint is directly proportional to the size of the main memory, since every memory location has to be written to (read from) serially. For this subsection only, it is assumed that the main memory is located off-chip and the time it takes to perform a checkpoint is $t_s = 643.6\mu s$, and the time it takes to perform a rollback is $t_r = 560.0\mu s$. 

Figure D.6: Timeline showing the terminology used to find the optimal number of consistency checks for a softcore processor protected with software fault-tolerance.
The mean execution time $L_5(m)$ of a CSCP checkpoint interval $((j-1)T, jT]$ with $m$ consistency checks is given by the following renewal equation:

$$L_5(m) = e^{-\lambda mT_{cc}}(mT_{cc} + mt_{cc} + t_s) + \sum_{i=1}^{m} \left( \int_{(i-1)T_{cc}}^{iT_{cc}} \lambda e^{-\lambda t} dt \right) \{ \varepsilon [iT_{cc} + (i-1)t_{cc} + t_{cc}/2 + t_r + L_5(1)] \\
+ (1 - \varepsilon) [(i-1)T_{cc} + (i-1)t_{cc} + T_{cc}/2 + t_d + t_r + L_5(1)] \}.$$  \hspace{1cm} (D.15)

Equation D.15 has two main terms: the term representing the execution time of a CSCP checkpoint interval $T$ when no upset occurs, and the execution time of the interval when an upset does occur. When there are no upsets, $m$ task intervals ($T_{cc}$) and $m$ consistency checks ($t_{cc}$) are executed, and the state of the processor is saved ($t_s$). When an upset does occur in the consistency check interval $((i-1)T_{cc}, iT_{cc}]$, it is either detected with a consistency check (with probability $\varepsilon$) or it is detected by some other detection technique (with probability $1 - \varepsilon$). When the fault is detected with a consistency check, it is detected some time between the start and the end of the $i^{th}$ check. Thus with an equal probability that the fault is detected at any point during the check, on average it is detected in time $t_{cc}/2$. Thus $(i-1) + 1/2$ consistency checks have executed when the fault is detected. Also, $i$ consistency check intervals ($T_{cc}$) have executed, a rollback is required ($t_r$), and the CSCP interval is restarted ($L_5(m)$). When a consistency check does not detect the upset in interval $((i-1)T_{cc}, iT_{cc}]$, which occurs at time $T_{cc}/2$ (can occur anywhere during the interval with equal probability), it is detected by some other detection technique with detection latency $t_d$. Once the fault is detected, $(i-1)$ consistency check intervals and consistency checks have executed ($T_{cc} + t_{cc}$), a rollback is required ($t_r$) and the CSCP interval is restarted ($L_5(m)$). The summation in the second term of Equation D.15 sums the probability that an upset occurs during one of the $m$ consistency check intervals.

Similar to the optimal checkpoint interval $T$, the optimal consistency check interval $T_{cc}$ minimizes the mean execution time of a CSCP interval (Equation D.15). If $T_{cc}$ is too small, there is an excessive amount of consistency check overhead, and it is more likely for a fault to occur during the CSCP interval. If $T_{cc}$ is too large, there is a larger amount of the program to rerun when a rollback occurs. The probability $\varepsilon$ that a consistency check detects an upset also affects the value
of \( T_{cc} \). If consistency checks are less likely to detect an upset (a small value of \( \varepsilon \)), then fewer consistency checks are performed (larger value of \( T_{cc} \)).

Solving Equation D.15 and substituting \( T_{cc} = T / m \) produces:

\[
L_5(m) = \frac{\alpha T \left[ 1 - \varepsilon + e^{\lambda T/m}(1 + \varepsilon) \right] + m[\alpha(2 + \beta \varepsilon)T_{cc} - 2\beta (\alpha - 1)t_d - \alpha t_r - t_s]}{2m\beta},
\]

(D.16)

where

\[
\alpha = (e^{\lambda T} - 1), \text{ and }
\]
\[
\beta = (e^{\lambda T/m} - 1).
\]

To find the value of \( T_{cc} \) that minimizes the time to complete a CSCP checkpoint interval, \( m = T / T_{cc} \) is substituted into Equation D.16 to produce:

\[
L_5(T_{cc}) = \frac{\alpha T_{cc} \left[ 1 - \varepsilon + e^{\lambda T_{cc}}(1 + \varepsilon) \right] + \alpha(2 + \beta \varepsilon)T_{cc} - 2\beta [\alpha(\varepsilon - 1)t_d - \alpha t_r - t_s]}{2\beta},
\]

(D.17)

where

\[
\alpha = (e^{\lambda T} - 1), \text{ and }
\]
\[
\beta = (e^{\lambda T_{cc}} - 1).
\]

Figure D.7 shows the optimal value \( T_{cc}^* \) as the value of \( T_{cc} \) that minimizes the execution time of a CSCP checkpoint interval. The optimal value of \( T_{cc} \) is found by differentiating Equation D.17 with respect to \( T_{cc} \), equating it to zero and solving for \( T_{cc} \). Differentiating Equation D.17, equating it to zero, and simplifying produces:

\[
\varepsilon - 1 + e^{2\lambda T_{cc}}(1 + \varepsilon) - 2e^{\lambda T_{cc}}(\varepsilon + \lambda T_{cc} + \lambda t_{cc}) == 0.
\]

(D.18)

Since \( T_{cc} \) in Equation D.18 cannot be solved for directly, a second degree Taylor series expansion is used to approximate the exponential terms. After performing the approximation and simplifying, the following equation is produced:

\[
2(\lambda t_{cc}) + 2(\lambda^2 t_{cc}) T_{cc} - (\varepsilon \lambda^2 - \lambda^3 t_{cc}) T_{cc}^2 == 0.
\]

(D.19)
The only valid solution to Equation D.19 produces an approximation for the optimal consistency check interval:

\[ T_{cc}^* = \frac{\lambda t_{cc} + \sqrt{\lambda t_{cc} (2\epsilon - \lambda t_{cc})}}{\lambda (\epsilon - \lambda t_{cc})} \]  

(D.20)

With an approximation for the optimal consistency check interval \( T_{cc}^* \), the optimal number of consistency check intervals \( m^* \) is found as follows:

- If \( T_{cc}^* < T \), then \( m = \lfloor T / T_{cc}^* \rfloor \). But, if \( L_5(m) > L_5(m + 1) \) (using Equation D.16), then \( m^* = m + 1 \), else \( m^* = m \).

- If \( T_{cc}^* \geq T \), then \( m^* = 1 \).

The optimal number of consistency checks \( (m^*) \) per CSCP checkpoint interval for a softcore processor are shown in Table D.5 for a series of error rates. The table also shows the error between
the optimal consistency check interval \( (T_{cc}^*) \) and the approximated ideal consistency check interval \( (\tilde{T}_{cc}) \), as well as the amount of overhead per checkpoint interval due to consistency checks and checkpointing. Although the consistency check interval approximation error is small, it is two orders of magnitude larger than the approximation error in the optimal checkpoint interval.

Table D.5: Comparison of optimal consistency check interval \( (T_{cc}^*) \) with approximated ideal consistency check interval \( (\tilde{T}_{cc}) \) for a softcore processor.

<table>
<thead>
<tr>
<th>( \lambda ) (errors/year)</th>
<th>( m^* )</th>
<th>( T_{cc} ) (s)</th>
<th>( T_{cc}^* ) (s)</th>
<th>%Err</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.01</td>
<td>4</td>
<td>545.5387</td>
<td>539.2289</td>
<td>1.17</td>
</tr>
<tr>
<td>0.1</td>
<td>4</td>
<td>172.5145</td>
<td>173.1913</td>
<td>0.39</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>54.5539</td>
<td>54.4269</td>
<td>0.23</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
<td>17.2515</td>
<td>17.2676</td>
<td>0.093</td>
</tr>
<tr>
<td>100</td>
<td>4</td>
<td>5.4554</td>
<td>5.4581</td>
<td>0.049</td>
</tr>
<tr>
<td>1000</td>
<td>4</td>
<td>1.7252</td>
<td>1.7247</td>
<td>0.029</td>
</tr>
<tr>
<td>10,000</td>
<td>4</td>
<td>0.5456</td>
<td>0.5455</td>
<td>0.018</td>
</tr>
<tr>
<td>100,000</td>
<td>4</td>
<td>0.1725</td>
<td>0.1725</td>
<td>0</td>
</tr>
<tr>
<td>1,000,000</td>
<td>4</td>
<td>0.0546</td>
<td>0.0546</td>
<td>0</td>
</tr>
</tbody>
</table>
APPENDIX E.  OPTIMAL NUMBER OF CONSISTENCY CHECKS

The probability model in Appendix D indicates that in order to minimize performance penalties, one consistency check per checkpoint interval should be run. That probability model does a great job of optimizing in terms of performance, but it makes some simplifying assumptions. This appendix finds the optimal consistency check interval empirically.

The probability model in Appendix D makes two simplifying assumptions about the optimal number of consistency checks. The assumptions are that software detection techniques detect every upset, and that the optimal checkpoint interval is independent from the optimal consistency check interval. About the first assumption, Chapter 3 discusses that not only do software detection techniques not detect every upset, but that large detection latencies lead to undetected upsets, and endless detection loops (dloops). Dloops are accounted for in the optimal softcore processor checkpoint interval model (Equation D.10) with the $\delta$ constant. But dloops are not directly accounted for in the optimal consistency check interval probability model (Equation D.15).

To decide whether the simplifying assumptions made in the probability model can be trusted, the reliability of a LEON3 softcore processor protected only with consistency checks and rollback checkpointing is compared when different checkpoint intervals are selected. Hardware fault-injection is performed on the protected LEON3 processor designs while the ToUpper testbench program is run. The graphs in Figure E.1 shows the reliability results when one, two, three, and four consistency checks are performed per checkpoint interval. The graphs in the figure plot the MTTF in years, $\text{MuITF} \times 10^{14}$, and RAP.

Selecting the optimal checkpoint interval based on the graphs in Figure E.1 is done by selecting the number of consistency checks that produces the maximum reliability. For all three metrics (MTTF, MuITF, and RAP) the optimal number of consistency checks per checkpoint interval is three. Interestingly, the three metrics do not agree on the second-best number of optimal consistency checks per checkpoint interval. Using MTTF, the second best number of consistency
Figure E.1: Decide the number of consistency checks per checkpoint interval based on empirical fault-injection results (MTTF, MuITF, and RAP) of a LEON3 softcore processor protected with consistency checks and rollback checkpointing.

The number of consistency checks to perform is four. Using MuITF the second-best value is two, and using RAP the second-best number to use is one. Throughout this work it is assumed that three consistency checks are performed per checkpoint interval.
APPENDIX F. CODE DUPLICATION OVERHEAD IN LEON3

Code duplication [36, 75–77] is one of two software fault-tolerant techniques used in the software implemented fault tolerance (SWIFT) [29] processor reliability technique. When performed by a compiler, code duplication incurs about a 1.4× performance penalty. But if code duplication is instead applied at the assembly code level, the costs are much greater. This appendix shows how code duplication can be applied at the assembly code level (i.e. after a compiler has allocated registers).

Assembly code duplication causes redundant instructions to be executed whenever a data instruction (load, store, or ALU instruction) is normally run. To support assembly code duplication, a second register file is required, and instruction set architecture (ISA) changes are required. This appendix shows how costly code duplication is for a LEON3 softcore processor. In addition to requiring ISA changes, program runtimes grow by almost 4×.

F.1 Architectural Changes Required by Assembly Code Duplication

First, assembly code duplication causes duplicate instructions to be executed on duplicate registers. For example, consider an add instruction: \texttt{add \%i2, \%i1, \%i3}. This instruction adds source registers \%i1 and \%i2 and stores the sum in register \%i3. To detect upsets in the instruction itself, the instruction is executed a second time. But to also detect upsets in the registers, alternate registers are used in the second execution: \texttt{add \%i2', \%i1', \%i3'}. The apostrophe indicates a duplicated register.

Second, code duplication causes most instructions to grow to at least three instructions, and in some cases, much more. In the case of an ALU instruction such as an \texttt{add}, the instruction is executed twice, and then the result of both instructions is checked for equality. If the two results are not the same, the program branches to a fault handling function \texttt{(faultFunction)} which executes additional instructions to let the processor know that an upset is detected.
Listing F.1 contains pseudo-code examples of how code duplication works for data instructions (loads, stores, and ALU instructions). The pseudo-code shows every ALU instruction is executed twice (lines 1 and 2) using a duplicated set of registers, and the results of the two computations are compared (line 3) to ensure correctness. If there is an upset in one of the duplicated registers, or instruction words, the comparison will fail and the faultFunction is executed (line 4). For loads and stores, the memory address exists in duplicated registers. Before loading or storing data in memory, the duplicated address and duplicated data to write to memory (in the case of a store) are first checked for correctness (lines 1 and 7). This protects memory from reading from the wrong location, writing to the wrong location, or writing the wrong data.

Listing F.1: Pseudo-assembly code showing code growth due to code duplication

```
add %i2, %i1, %i3  →  add  %i2, %i1, %i3
add %i2’, %i1’, %i3’
cmp  %i3, %i3’
bne  faultFunction
5
ld  [%i4], %i2  →  cmp  %i4, %i4’
bne  faultFunction
6
ld  [%i4], %i2
7
mov  %i2, %i2’
8
st  %l2, [%o2]  →  cmp  %o2, %o2’
bne  faultFunction
9
st  %l2, [%o2]
10
```

F.2 Costs Of Assembly Code Duplication

Implementing instruction duplication at the assembly code level incurs exorbitant costs in terms of performance and in terms of processor modifications. First, doubling the number of registers requires a second register file, and requires changes to the instruction set architecture (ISA). SPARCv8 instructions that use registers require five bits per register (since there are 32 visible registers per register window), and there are, at most, three registers accessed in one instruction. For example, the ADD instruction shown in Figure F.1 shows two source register references (rs1 and rs2), and a destination register reference (rd). If the number of registers is doubled, then six bits are required to uniquely address any given register, necessitating the growth of each register field in SPARCv8 instructions.
F.2.1 ISA Changes

There are different ways to accommodate this doubling of registers. One way to accommodate this change is to add more bits to each instruction, causing a non-standard word bitwidth. This change would require at least three more bits to be added to each instruction, causing 32-bit instructions to grow to be at least 35-bits wide. Non-standard word widths are undesirable, and are more difficult to support, making this ISA change is unattractive.

Another ISA change that would support larger a larger register file is to hijack some of the bits in existing instructions. As many as three bits would have to be stolen from other fields in SPARCV8 instruction to support larger register fields. In some instructions, such as the one in Figure F.1, there are unused bits available, thus this change would merely require the realignment of fields within the instruction. But most instructions do not have unused bits, thus expanding register fields would encroach upon other fields in the instruction. This option causes fundamental changes to the SPARCV8 ISA, thus it is also unattractive.

### ADD Instruction

![ADD Instruction](image)

Figure F.1: SPARCV8 add instruction.

It is possible to double the number of registers without making changes to existing SPARCV8 instructions. The addition of three new instructions allow support for code duplication with duplicated registers, without changing the existing SPARCV8 instructions. First, the **DUP** instruction is used to rerun instructions but with duplicated registers. The instruction is placed after ALU instructions. The other two instructions that are added include the **DMOV** and **DCMP** instructions. The **DMOV** instruction moves a value from a non-duplicate register to a duplicate register. The **DCMP** instruction compares the values of a non-duplicate and duplicate register.
Listing F.2 shows a code duplication example which uses these less intrusive ISA changes. The `dup` instruction on line 2 indicates that the previous instruction (the `add` instruction) should be rerun, but using the duplicate register file. The `dcmp` instruction on line 3 (also used on lines 5 and 9) indicates that register `%i4` should be compared with register `%i4'`. The result of the comparison changes control codes in the same way a normal compare instruction would. Finally, the `dmov` instruction on line 8 moves the value in register `%i2` to the register `%i2'`. By adding these three instructions, code duplication is supported without the need to make intrusive ISA changes.

Listing F.2: Pseudo-assembly code showing code growth due to code duplication

```
add  %i2, %i1, %i3 → add  %i2, %i1, %i3
  dup
  dcmp %i3, %i3
  bne  faultFunction
  dcmp %i4, %i4
  bne  faultFunction
  ld  [%i4], %i2
  dmov %i2, %i2
  st  %l2, [%o2]
  → st  %l2, [%o2]
```

F.2.2 Performance Costs

Software fault-tolerance techniques provide protection with redundant instruction execution, so performance costs are their primary cost. This subsection shows that the performance cost of implementing code duplication is exorbitant. A program protected with code duplication runs $4 \times$ longer than an unprotected program. Chapter 4 shows that there are other ways to protect the register file and ALU which are much less costly.

Table F.1 shows the performance cost of three of the testbench programs used in this work. The cost represents the number of additional clock cycles required to run a program with code duplication compared to an unprotected program. On average, it takes $4 \times$ longer to run a program with code duplication. Since less costly methods can provide the same protection, code duplication is not used in this work.
Table F.1: Performance cost of code duplication.

<table>
<thead>
<tr>
<th>Program</th>
<th>Performance Cost ((\rho))</th>
</tr>
</thead>
<tbody>
<tr>
<td>ToUpper</td>
<td>4.29\times</td>
</tr>
<tr>
<td>Fhourcorners</td>
<td>3.73\times</td>
</tr>
<tr>
<td>MatrixMult</td>
<td>3.72\times</td>
</tr>
<tr>
<td>Average</td>
<td>3.91\times</td>
</tr>
</tbody>
</table>
APPENDIX G. SYNTHESIS-SENSITIVE RELIABILITY OBSERVATIONS

Previous studies have shown that different design placements directly affect the power consumption of a design running on an SRAM-based FPGA [111]. This work finds that different placements can also have a significant impact on reliability. This appendix compares the reliability of a software-fault tolerant LEON3 softcore processor with and without placement constraints. The MTTF of an unconstrained placement is 45% worse than a constrained placement.

Chapter 6 shows that the most sensitive part of the LEON3 softcore processor protected with software fault-tolerance is its output (Table 6.12). This is not surprising since the output acts as a single point of failure. To minimize the effects of this single point of failure on reliability, the logic and routing leading to the outputs should be placed as closely as possible to the IOBs of the FPGA. Spreading this logic and routing across the chip causes more SDC and DUE configuration bits. This principle is true, not only for the output logic and routing, but for any sensitive logic and routing. To achieve the best reliability, sensitive routes should be as short as possible, and sensitive logic should be mapped and placed in such a way as to minimize the number of sensitive configuration bits.

This work does not provide mapping, placement, and routing strategies to minimize sensitivity, but simply observes reliability variation due to different placements. Figure G.1(a) shows the placement and routing of an unconstrained software fault-tolerant LEON3 softcore processor, as it appears in Xilinx’s FPGA Editor tool. In an attempt to reduce the sensitivities in the logic and routing leading to the output IOBs, Figure G.1(b) shows the same LEON3 processor design when it is constrained to be placed only in the FPGA rows that contain the output IOBs. This placement constraint is applied in the top-level LEON3 VHDL entity.

Hardware fault-injection is performed on both LEON3 processors shown in Figure G.1, with the ToUpper testbench program running (Chapter 5). Table G.1 shows the number of SDC, DUE, and dloop configuration bits for each placement, as well as the MTTF. The values reported
(a) FPGA Editor view of a software fault-tolerant LEON3 softcore processor with unconstrained placement.

(b) FPGA Editor view of a software fault-tolerant LEON3 softcore processor with constrained placement.

Figure G.1: FPGA Editor view of an unconstrained and constrained software fault-tolerant LEON3 softcore processor.

in parentheses indicate how much better the constrained LEON3 processor performs compared to the unconstrained processor. Overall, the MTTF of the LEON3 with constrained placement is 45% better than the LEON3 with unconstrained placement.

Table G.1: Comparison of fault-injection results for a software fault-tolerant LEON3 processor with and without constrained placement.

<table>
<thead>
<tr>
<th>LEON3 Design</th>
<th>SDC</th>
<th>DUE</th>
<th>dloop</th>
<th>MTTF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unconstrained</td>
<td>4497 (1.00×)</td>
<td>3767 (1.00×)</td>
<td>2841 (1.00×)</td>
<td>1.19 (1.00×)</td>
</tr>
<tr>
<td>Constrained</td>
<td>3666 (1.23×)</td>
<td>2046 (1.84×)</td>
<td>1301 (2.18×)</td>
<td>1.73 (1.45×)</td>
</tr>
</tbody>
</table>
The impact of mapping, placement, and routing on reliability is surprisingly large. The 45% MTTF improvement was obtained without attempting to place specific processor units, or routes. The placement directive was on the entire processor. With greater care, additional reliability improvements may be possible by performing fine-grain placement. As a result of observing a 45% reliability improvement, every LEON3 processor design used in this work has constrained placement. The results of Table G.1 also suggests the need of a reliability-based synthesis tool.
This appendix validates the accuracy of the hardware fault-injection model presented in Chapter 5 with radiation test results. Heavy ion beam experiments were performed at the Cyclotron Institute at Texas A&M University in July 2011, which tested the unmitigated LEON3 softcore processor as well as the LEON3 protected with all of the software fault-tolerance techniques described in Chapter 4. The radiation test results show that hardware fault-injection is accurate to within 7%.

The version of the LEON3 processor protected with software fault-tolerance that was tested at the cyclotron was an older, less effective version. Thus before presenting the radiation MTTF results, the hardware fault-injection results of this older version are first presented. The version of the software-protected LEON3 that was tested in the cyclotron performed only a subset of the consistency checks described in Chapter 4 and did not perform memory scrubbing.

### H.1 Hardware Fault-Injection Results

The model used for performing hardware fault-injection is the same model described in Chapter 5. Fault-injection tests were performed on the unmitigated LEON3 processor and the deprecated software fault-tolerant LEON3 with only three of the benchmark programs described in Chapter 5. The programs used include the ToUpper, Fhourcorners, and MatrixMult programs. The raw fault-injection results are shown in Table H.1. The percentages reported in the unACE column of Table H.1 refer to the percentage of processor bits out of all the configuration bits in the processor that are unACE. For example, out of 598,647 configuration bits used by the unmitigated LEON3 processor, 525,628 are unACE bits. The remaining 14% are ACE bits. Of the upsets to those ACE bits, the unmitigated LEON3 processor detects 16% of them, and the remaining 84% are undetected. The table shows that using software mitigation techniques, 93% of the ACE upsets
Table H.1: Average fault-injection results for the protected and unprotected LEON3 processors.

<table>
<thead>
<tr>
<th>LEON3 Design</th>
<th>unACE</th>
<th></th>
<th>ACE</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>DRU</td>
<td>DUE</td>
<td>SDC</td>
</tr>
<tr>
<td>Unmitigated</td>
<td>525,628 (86.4%)</td>
<td>0 (0.0%)</td>
<td>11,680 (16.2%)</td>
<td>63,855 (83.8%)</td>
</tr>
<tr>
<td>SW Techniques</td>
<td>740,334 (81.7%)</td>
<td>153,582 (92.5%)</td>
<td>2557 (1.5%)</td>
<td>9874 (6.0%)</td>
</tr>
</tbody>
</table>

are detected and successfully recovered. It also indicates that the processor protected with software techniques can detect and recover from 99% of all upsets to the processor (including unACE bits).

The architectural vulnerability factor (AVF) and mean time to failure (MTTF) resulting from the hardware fault-injection are computed as described in Chapter 5, except that the relative mean time to failure rMTTF is reported instead of MTTF. If the upset rate in the MTTF equation (Equation 5.5) is considered to be a constant, then the MTTF is proportional to the second term in the equation for MTTF. This second term is referred to as the relative mean time to failure (rMTTF):

\[
rMTTF = \frac{1}{\text{CFGbits}} \cdot \frac{1}{\text{AVF}} = \frac{1}{\# \text{SDCs} + \# \text{DUEs}}. \tag{H.1}
\]

The AVF and rMTTF of both LEON3 processor designs is shown in Table H.2. The first half of the table reports three different types of AVF. The second half of the table reports three different types of rMTTF for the processors. \(rMTTF_{\text{SDC}}\) and \(rMTTF_{\text{DUE}}\) are each proportional to the average time before an SDC or DUE (respectively) happens. Table H.2 shows that the processor protected with software techniques is about \(6 \times\) more reliable than an unmitigated processor.

### H.2 Radiation Test Results

In addition to performing hardware fault-injection, radiation testing is used to measure the reliability of the unprotected LEON3 processor and the LEON3 processor protected with software fault-tolerant techniques. Heavy ion beam experiments were performed at the Cyclotron Institute at
Table H.2: Comparison of the architectural vulnerability factors (AVFs) and relative mean useful instruction to failure (rMuITF) of an unmitigated LEON3 against the mitigated LEON3 processor.

<table>
<thead>
<tr>
<th>LEON3 Design</th>
<th>AVF Values</th>
<th>rMTTF Values $\times 10^{-5}$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AVF&lt;sub&gt;SDC&lt;/sub&gt;</td>
<td>AVF&lt;sub&gt;DUE&lt;/sub&gt;</td>
</tr>
<tr>
<td>Unmitigated</td>
<td>10.62%</td>
<td>1.94%</td>
</tr>
<tr>
<td>SW Techniques</td>
<td>1.09%</td>
<td>0.28%</td>
</tr>
</tbody>
</table>

Texas A&M University in July 2011. This section presents the radiation test results which validate the hardware fault-injection results.

The radiation test setup is shown in Figure H.1. The test procedure for the heavy-ion beam is similar to the hardware fault-injection procedure (Chapter 5), but instead of artificially inserting upsets, upsets are caused by the heavy ions hitting the device. Both the mitigated and unmitigated softcore processor designs were irradiated for a total of about two hours.

In the tests performed in this study the heavy ions used included helium, nitrogen, argon, and copper. The linear energy transfers (LETs) ranged from 0.11 to 30.8 MeV $\cdot cm^2/mg$, and the average fluxes ranged from $9.02 \times 10^1$ to $3.69 \times 10^5 p/cm^2/s$.

Unfortunately, the checkpoint interval that was used in the software fault-tolerant LEON3 design was much lower than it should have been. A checkpoint interval of 76.3ms was used, but the bitstream scrub interval of the Xilinx Virtex4 FX60 (the device used in the beam), is 118ms. Thus, as explained in Chapter 3, the checkpoint interval should not have been less than 118ms. The result of having using a checkpoint interval that is too small was that dloops occurred more often than they should have, causing the need to reconfigure the DUT more often. Thus the radiation test results are pessimistic.

The accelerator test results in Table H.3 report the various categories of ACE bits for the unmitigated LEON3 and the LEON3 protected with software reliability techniques. The results in this table are very similar to the hardware fault-injection ACE results shown in Table H.1 for both the mitigated and unmitigated LEON3 processor. The close match between radiation test results and hardware fault-injection results shows that hardware fault-injection is an accurate way to measure softcore processor reliability.
Figure H.1: The accelerator test setup.

Table H.3: Average fault-injection results for the protected and unprotected LEON3 processors.

<table>
<thead>
<tr>
<th></th>
<th>DRU</th>
<th>DUE</th>
<th>SDC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unmitigated</td>
<td>0 (0.0%)</td>
<td>6 (15.4%)</td>
<td>33 (84.6%)</td>
</tr>
<tr>
<td>SW Techniques</td>
<td>481 (95.4%)</td>
<td>3 (0.6%)</td>
<td>20 (4.0%)</td>
</tr>
</tbody>
</table>

The slight improvement in the mitigated LEON3’s DRUs shown in Table H.3 is most likely due to upsets detected in BRAMs. In this study hardware fault-injection only upsets FPGA configuration memory. But in radiation tests, user memories can also be upset. Protected with parity, the mitigated processor should detect and recover from any upset to a BRAM.

Table H.4 compares the measured and modeled MTTF for the unprotected and protected LEON3 processor. The measured MTTF values are found by dividing the number of recorded
Table H.4: Average error between modeled and measured MTTF.

<table>
<thead>
<tr>
<th>LEON3 Design</th>
<th>Modeled</th>
<th>Measured</th>
<th>%Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unmitigated</td>
<td>42.2 sec</td>
<td>40.4 sec</td>
<td>4.4%</td>
</tr>
<tr>
<td>SW Techniques</td>
<td>264.6 sec (6.0×)</td>
<td>246.8 sec (5.8×)</td>
<td>7.2%</td>
</tr>
</tbody>
</table>

events by the radiation test run time. The modeled results are found by using Equations 5.4 and 5.5. The AVF value in these equations is measured using hardware fault-injection (Table H.2), and the raw error rate is measured from radiation test results.

The results in Table H.4 show that the measured and modeled MTTF for the unmitigated LEON3 processor differ by only 4%. For the LEON3 processor protected with software fault-tolerant techniques, the modeled MTTF is about 7% larger than the measured MTTF. The differences between modeled and measured MTTF might be reduced with longer radiation testing, and if a larger checkpoint interval is used.

The measured and modeled MTTF for both the protected and unprotected designs in Table H.4 have a raw upset rate of 0.18 upsets per second. Since both designs have the same upset rate, the unmitigated and mitigated MTTF values in Table H.4 can be directly compared. The values in parentheses show how much better the MTTF of the LEON3 processor protected with software techniques is compared to the unmitigated processor. The modeled MTTF estimates that the mitigated LEON3 is $6 \times$ more reliable than the unmitigated LEON3, but the the measured MTTF reports that it is $5.8 \times$ more reliable. The difference between measured and modeled MTTF could be reduced with longer radiation testing and with a larger checkpoint interval used in the radiation tests.

The radiation test provides two important results. First, hardware fault-injection is consistent with radiation testing. This means that hardware fault-injection can be confidently used to model softcore processor reliability. The second important result is that the estimated and measured MTTF match relatively well. Thus the AVF and MTTF metric for softcore processors introduced in Chapter 5 is accurate.
APPENDIX I. ADDITIONAL FAULT-INJECTION RESULTS

This appendix acts as an extension to Chapter 6. Additional details and results are provided for each of the first four sections in the chapter. These additional details offer insights and details that did not fit in the chapter.

I.1 Unmitigated LEON3 Reliability

The charts in Figure I.1 show the relative number of unACE, DUE, and SDC configuration bits for the unmitigated LEON3 processor when hardware fault-injection is performed on each testbench program. The figure also shows the collective results of all of the programs. The collective results represent a union of all of the individual program results, so the collective results show the largest percentage of SDCs and DUEs compared to the individual program results. The collected results in Figure I.1 also show that there is a lot of overlap in the SDCs and DUEs among the different benchmark programs.

The pie chart in Figure I.2 categorizes the ACE bits of Table 6.1 in terms of FPGA configuration bit failure mode (Figures 2.2 and A.2). The chart shows that over 80% of the upsets occur in routing, which are almost evenly split between open and short errors. Interestingly, the ACE bit failure mode ratios of all of the mitigated LEON3 processors are almost identical to the ratios shown in Figure I.2.

I.2 Reliability of Hardware-Based Softcore Processor Mitigation

The charts in Figure I.3 show the percentages of DRE, DUE, and SDC bits for each of the testbench programs when DWC with rollback checkpointing is used. Figure I.3(b) shows these ACE bit categories for the DWC design with a duplicated clock and reset signal, and Figure I.3(a) shows the ACE bit percentages for the DWC design without duplicated inputs. Both charts also include the collected fault-injection results for all of the testbench designs.
Figure I.1: Relative number of unACE, SDC, and DUE bits for an unmitigated LEON3 processor when each of the testbench programs are run.

Figure I.2: Fault mode ratios of the ACE bits in the unmitigated LEON3 softcore processor.

The charts of Figure I.4 show the same results as Figure I.3 but for the LEON3 processor protected with TMR and roll-forward checkpointing. The right-most bar in each of the charts collects all of the fault-injection information into a unified result. This bar represents the union of all of the benchmarks, not an average. The collection all of the results into a unified result is done pessimistically. If a configuration bit in any result shows up as an SDC, it is recorded in the unified result as an SDC, even if that configuration bit is reported as a DRE bit in all of the
Figure I.3: Hardware fault-injection results for the LEON3 processor protected with DWC and rollback checkpointing.

other benchmark programs. The details of the collected results for the unmitigated and hardware mitigated LEON3 softcore processors is shown in Table 6.3.

Figure I.4: Hardware fault-injection results for the LEON3 processor protected with TMR and roll-forward checkpointing.

The pie charts in Figure I.5 shows the locations of the SDC and DUE configuration upsets within the DWC LEON3 processors with duplicated and non-duplicated clock domains, as well as within the TMR LEON3 processors with triplicated and non-triplicated inputs and outputs. The pie charts show that there is very little difference between the two DWC designs in terms of where the
upsets occur. On the other hand, there is significant difference between the two TMR designs (as also reported by Table 6.3). In the processor protected with non-triplicated inputs and outputs, 99% of the SDCs and DUEs occur in the logic and routing coming into, or going out of the processor. But in the LEON3 processor protected with TMR and triplicated inputs and outputs, there is only a single DUE in the processor inputs and outputs. The handful of SDCs and DUEs that do occur are most likely the result of single upsets that affect more than one FPGA resource [24].

Figure I.5: Locations of the SDC and DUE configuration upsets in the traditionally mitigated LEON3 processors.
I.3 Reliability of Individual Software Fault-Tolerance Techniques

As discussed in Chapter 3 dloops are more likely to occur when a detection technique has a large detection latency. The average detection latency for each fault-tolerance technique is shown in Table I.1. The average detection latency for the dloops allowed by each detection technique is also shown. The table shows that, as expected, the hardware-based detection techniques have a smaller detection latency. Thus it is no coincidence that the hardware-based detection techniques also have the fewest dloops. Although software indicators have virtually no area or performance cost, Table I.1 shows that they have a very large detection latency. The table also shows that software indicators’ average detection latency is greater than their average dloop detection latency. This reinforces the idea discussed in Chapter 3 that a large detection latency alone does not guarantee that a dloop will occur.

Table I.1: Average detection latencies for each detection technique when upsets are detected and successfully repaired (DRU) and when upsets result in a dloop.

<table>
<thead>
<tr>
<th></th>
<th>Average Detection Latency</th>
<th>Avg. DRE Detection Latency (cycles)</th>
<th>Avg. dloop Detection Latency (cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEON3 Design</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CDWC Register File</td>
<td>3724</td>
<td>13,597</td>
<td></td>
</tr>
<tr>
<td>Control-Flow Monitoring</td>
<td>5885</td>
<td>20,222</td>
<td></td>
</tr>
<tr>
<td>Consistency Checks</td>
<td>3487</td>
<td>52,336</td>
<td></td>
</tr>
<tr>
<td>Software Indicators</td>
<td>33,126</td>
<td>19,753</td>
<td></td>
</tr>
<tr>
<td>Parity &amp; Scrubbing</td>
<td>1516</td>
<td>16,284</td>
<td></td>
</tr>
</tbody>
</table>

I.4 Reliability of Full Software Fault-Tolerance

The hardware fault-injection results for each combination of detection techniques are shown shown in the charts of Figure I.6. The charts show the percentage of ACE bits that are DRE, DUE, and SDC for each of the testbench programs. Additionally, the collected fault-injection results are shown when all of the program test results are combined. As previously done, this combination is a union of all the fault-injection results rather than an averaging. Thus worst-case behavior is reported in the collected results.
(a) Percentage of DRE, DUE, and SDC bits for the LEON3 processor with the Min SW detection combination and rollback checkpointing.

(b) Percentage of DRE, DUE, and SDC bits for the LEON3 processor with the no CDWC detection combination and rollback checkpointing.

(c) Percentage of DRE, DUE, and SDC bits for the LEON3 processor with the no Control-Flow detection combination and rollback checkpointing.

(d) Percentage of DRE, DUE, and SDC bits for the LEON3 processor with the Full SW detection combination and rollback checkpointing.

Figure 1.6: Hardware fault-injection results for the LEON3 processor protected with different combinations of software detection techniques and rollback checkpointing.
APPENDIX J.  ON-ORBIT EXPERIMENTS WITH MISSE-8

This dissertation shows that softcore processors implemented in commercial SRAM-based FPGAs are ideal for space-based applications. The processor design-space is explored to find low-cost or ultra-reliable mitigation techniques to protect the softcore processor. The conclusion of this work shows that softcore processors can be made even more reliable than costly rad-hard processors. Some of this work was validated through radiation testing (Appendix H), but some of this work is also validated with on-orbit experiments.

Through collaborations with Sandia National Laboratory, some of this work is validated on an experimental payload placed on the International Space Station (ISS) [112, 113]. The experimental payload, called the Materials International Space Station Experiment (MISSE), was deployed on 20 May 2011 as a part of the space shuttle Endeavour’s final voyage to the ISS (Figure J.1).

The Materials International Space Station Experiment (MISSE) is a series of experiments focused on testing the effects of a space environment on materials and computing elements [114, 115]. The materials and computing elements are tested for the effects of atomic oxygen, ultraviolet rays, direct sunlight, radiation, and the extremes of head and cold. Each MISSE payload is attached to the outside of the ISS. The 8th MISSE experiment delivered on 20 May 2011 contains the second SEU Xilinx-Sandia Experiment (SEUXSE II) [116]. Part of the SEUXSE II payload contains a space-qualified Xilinx Virtex 4 and space-qualified Virtex 5 FPGA. Some of the experiments running on that FPGA include softcore processor tests.

In collaboration with Sandia National Laboratory, we have developed two experiments to run on the Virtex 5 FPGA. The first experiment validates the results of the reliable FPGA user memory study presented in Appendix B. Figure J.2 shows a block diagram of the experiment. BRAMs protected with TMR and memory scrubbing self-detect when an upset occurs. A tri-
A second similar experiment is performed on Xilinx’s 8-bit Picoblaze processor [102]. Figure J.3 shows that this second experiment contains a triplicated Picoblaze processor with a triplicated, scrubbing main memory. The program running on the triplicated Picoblaze processor continuously generates a Fibonacci sequence. Similar to the memory experiment, self-detected upsets are counted in a triplicated counter whose value is available upon request.

The results of these two experiments are summarized in Table J.1. Although the experiments have been running since 20 May 2011 (255 days as of this writing), they are not continually run. These experiments are swapped out with a series of other experiments. Only two detected, repaired upsets have occurred in the Picoblaze test, and only one in the memory test. No undetected errors have occurred in either experiment.
Figure J.2: Memory experiment performed on the MISSE-8.

Figure J.3: Picoblaze processor experiment performed on the MISSE-8.

Table J.1: MISSE-8 SEUXSE II test results

<table>
<thead>
<tr>
<th>Experiment</th>
<th>SEUXSE II Operational Days</th>
<th>Detected Events</th>
<th>Undetected Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>255</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Picoblaze</td>
<td>255</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>