

High-level synthesis of triple modular redundant FPGA circuits with energy efficient error recovery mechanisms

Dimitrios Agiakatsikas

A thesis in fulfillment of the requirements for the degree of

Doctor of Philosophy



School of Computer Science and Engineering

Faculty of Engineering

The University of New South Wales

June 2019

THE UNIVERSITY OF NEW SOUTH WALES
Thesis/Dissertation Sheet

Surname or Family name: **Agiakatsikas**

First name: **Dimitrios** Other name/s:

Abbreviation for degree as given in the University calendar: **PhD**

School: **School of Computer Science and Engineering**

Faculty: **Faculty of Engineering**

Title: High-level synthesis of triple modular redundant FPGA circuits with energy efficient error recovery mechanisms

Abstract

There is a growing interest in deploying commercial SRAM-based Field Programmable Gate Array (FPGA) circuits in space due to their low cost, reconfigurability, high logic capacity and rich I/O interfaces. However, their configuration memory (CM) is vulnerable to ionising radiation which raises the need for effective fault-tolerant design techniques. This thesis provides the following contributions to mitigate the negative effects of soft errors in SRAM FPGA circuits.

Triple Modular Redundancy (TMR) with periodic CM scrubbing or Module-based CM error recovery (MER) are popular techniques for mitigating soft errors in FPGA circuits. However, this thesis shows that MER does not recover CM soft errors in logic instantiated outside the reconfigurable regions of TMR modules. To address this limitation, a hybrid error recovery mechanism, namely FMER, is proposed. FMER uses selective periodic scrubbing and MER to recover CM soft errors inside and outside the reconfigurable regions of TMR modules, respectively. Experimental results indicate that TMR circuits with FMER achieve higher dependability with less energy consumption than those using periodic scrubbing or MER alone.

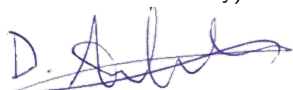
An imperative component of MER and FMER is the reconfiguration control network (RCN) that transfers the minority reports of TMR components, i.e., which, if any, TMR module needs recovery, to the FPGA's reconfiguration controller (RC). Although several reliable RCs have been proposed, a study of reliable RCNs has not been previously reported. This thesis fills this research gap, by proposing a technique that transfers the circuit's minority reports to the RC via the configuration-layer of the FPGA. This reduces the resource utilisation of the RCN and therefore its failure rate. Results show that the proposed RCN achieves higher reliability than alternative RCN architectures reported in the literature.

The last contribution of this thesis is a high-level synthesis (HLS) tool, namely TLegUp, developed within the LegUp HLS framework. TLegUp triplicates Xilinx 7-series FPGA circuits during HLS rather than during the register-transfer level pre- or post-synthesis flow stage, as existing computer-aided design tools do. Results show that TLegUp can generate non-partitioned TMR circuits with 500x less soft error sensitivity than non-triplicated functional equivalent baseline circuits, while utilising 3-4x more resources and having 11% lower frequency.

Declaration relating to disposition of project thesis/dissertation

I hereby grant to the University of New South Wales or its agents the right to archive and to make available my thesis or dissertation in whole or in part in the University libraries in all forms of media, now or here after known, subject to the provisions of the Copyright Act 1968. I retain all property rights, such as patent rights. I also retain the right to use in future works (such as articles or books) all or part of this thesis or dissertation.

I also authorise University Microfilms to use the 350 word abstract of my thesis in Dissertation Abstracts International (this is applicable to doctoral theses only).



Signature **Dimitrios Agiakatsikas**



Witness **Oliver Diessel**

Date **01 April, 2019**

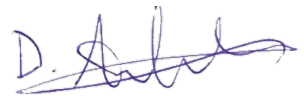
The University recognises that there may be exceptional circumstances requiring restrictions on copying or conditions on use. Requests for restriction for a period of up to 2 years must be made in writing. Requests for a longer period of restriction may be considered in exceptional circumstances and require the approval of the Dean of Graduate Research.

FOR OFFICE USE ONLY

Date of completion of requirements for Award

Originality Statement

I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, or substantial proportions of material which have been accepted for the award of any other degree or diploma at UNSW or any other educational institution, except where due acknowledgement is made in the thesis. Any contribution made to the research by others, with whom I have worked at UNSW or elsewhere, is explicitly acknowledged in the thesis. I also declare that the intellectual content of this thesis is the product of my own work, except to the extent that assistance from others in the project's design and conception or in style, presentation and linguistic expression is acknowledged.



Dimitrios Agiakatsikas

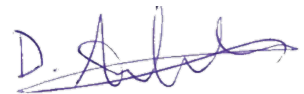
01 April, 2019

Copyright Statement

I hereby grant the University of New South Wales or its agents the right to archive and to make available my thesis or dissertation in whole or part in the University libraries in all forms of media, now or here after known, subject to the provisions of the Copyright Act 1968. I retain all proprietary rights, such as patent rights. I also retain the right to use in future works (such as articles or books) all or part of this thesis or dissertation.

I also authorise University Microfilms to use the 350 word abstract of my thesis in Dissertation Abstract International (this is applicable to doctoral theses only).

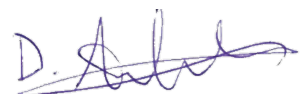
I have either used no substantial portions of copyright material in my thesis or I have obtained permission to use copyright material; where permission has not been granted I have applied/will apply for a partial restriction of the digital copy of my thesis or dissertation.



Dimitrios Agiakatsikas
01 April, 2019

Authenticity Statement

I certify that the Library deposit digital copy is a direct equivalent of the final officially approved version of my thesis. No emendation of content has occurred and if there are any minor variations in formatting, they are the result of the conversion to digital format.



Dimitrios Agiakatsikas
01 April, 2019

INCLUSION OF PUBLICATIONS STATEMENT

UNSW is supportive of candidates publishing their research results during their candidature as detailed in the UNSW Thesis Examination Procedure.

Publications can be used in their thesis in lieu of a Chapter if:

- The student contributed greater than 50% of the content in the publication and is the “primary author”, i.e., the student was responsible primarily for the planning, execution and preparation of the work for publication
- The student has approval to include the publication in their thesis in lieu of a Chapter from their supervisor and Postgraduate Coordinator.
- The publication is not subject to any obligations or contractual agreements with a third party that would constrain its inclusion in the thesis

Please indicate whether this thesis contains published material or not.

☐

This thesis contains no publications, either published or submitted for publication

☒

Some of the work described in this thesis has been published and it has been documented in the relevant Chapters with acknowledgement


☐

This thesis has publications (either published or submitted for publication) incorporated into it in lieu of a chapter and the details are presented below

CANDIDATE'S DECLARATION

I declare that:

- I have complied with the Thesis Examination Procedure
- where I have used a publication in lieu of a Chapter, the listed publication(s) below meet(s) the requirements to be included in the thesis.

Name	Signature	Date (dd/mm/yy)
Dimitrios Agiakatsikas		27/03/19

Postgraduate Coordinator's Declaration (to be filled in where publications are used in lieu of Chapters)

I declare that:

- the information below is accurate
- where listed publication(s) have been used in lieu of Chapter(s), their use complies with the Thesis Examination Procedure
- the minimum requirements for the format of the thesis have been met.

PGC's Name	PGC's Signature	Date (dd/mm/yy)

Abstract

There is a growing interest in deploying commercial SRAM-based Field Programmable Gate Array (FPGA) circuits in space due to their low cost, reconfigurability, high logic capacity and rich I/O interfaces. However, their configuration memory (CM) is vulnerable to ionising radiation which raises the need for effective fault-tolerant design techniques. This thesis provides the following contributions to mitigate the negative effects of soft errors in SRAM FPGA circuits.

Triple Modular Redundancy (TMR) with periodic CM scrubbing or Module-based CM error recovery (MER) are popular techniques for mitigating soft errors in FPGA circuits. However, this thesis shows that MER does not recover CM soft errors in logic instantiated outside the reconfigurable regions of TMR modules. To address this limitation, a hybrid error recovery mechanism, namely FMER, is proposed. FMER uses selective periodic scrubbing and MER to recover CM soft errors inside and outside the reconfigurable regions of TMR modules, respectively. Experimental results indicate that TMR circuits with FMER achieve higher dependability with less energy consumption than those using periodic scrubbing or MER alone.

An imperative component of MER and FMER is the reconfiguration control network (RCN) that transfers the minority reports of TMR components, i.e., which, if any, TMR module needs recovery, to the FPGA's reconfiguration controller (RC). Although several reliable RCs have been proposed, a study of reliable RCNs has not been previously reported. This thesis fills this research gap, by proposing a technique that transfers the circuit's minority reports to the RC via the configuration-layer of the FPGA. This reduces the resource utilisation of the RCN and therefore its failure rate. Results show that the proposed RCN achieves higher reliability than alternative RCN architectures reported in the literature.

The last contribution of this thesis is a high-level synthesis (HLS) tool, namely TLegUp, developed within the LegUp HLS framework. TLegUp triplicates Xilinx 7-series FPGA circuits during HLS rather than during the register-transfer level pre- or post-synthesis flow stage, as existing computer-aided design tools do. Results show that TLegUp can generate non-partitioned TMR circuits with 500x less soft error sensitivity than non-triplicated functional equivalent baseline circuits, while utilising 3-4x more resources and having 11% lower frequency.

Acknowledgements

“Ideal teachers are those who use themselves as bridges over which they invite their students to cross, then having facilitated their crossing, joyfully collapse, encouraging them to create bridges of their own.”

Nikos Kazantzakis

I am thankful to my PhD advisor, Dr. Oliver Diessel. His advice, encouragement and support played crucial roles in completing my PhD degree. Oliver taught me how to conduct research, how to write scientific manuscripts, and how to organise my thoughts. Oliver was always available for me, during weekends, during holidays, even when he was not feeling well. I appreciate his passion for teaching me about how research should be done. Without Oliver’s mentorship, guidance and careful reviews I would not have written this PhD thesis. Most chapters would have been presented in a bottom-up style and the reader would have been lost in unnecessary details. Chapter 3 would have had less analytical results, and I would not have appreciated mathematics as much as I do now. During my PhD studies, Oliver gave me the opportunity to experience as much as possible the life of an academic. Next to Oliver, I learned how to review scientific papers and how to teach the course, Digital Circuits and Systems. All this experience helped me mature and come a step closer to becoming an academic.

I am also thankful to my PhD co-supervisor, Dr. Ediz Cetin, which involved me in the development of the RUSH CubeSat payload. The RUSH project motivated many outcomes of this thesis and I am very happy to have been part of it.

I am pleased to have collaborated with my colleagues at UNSW, Alexander Kroh, Dr. Ganghee Lee, Dr. Lingkan Gong, Dr. Nguyen Tran Huu Nguyen, Thomas Mitchell, Tong Wu, and Zhuoran Zhao. I learned a lot from them, and the outcome of this thesis would not have been possible without their help and our fruitful discussions and debates.

I gratefully acknowledge my Annual Progress Review (APR) panel members, Dr. Sri Parameswaran (panel chair), Dr. Hui Wu, and Dr. Eric Martin for keeping an eye on my

PhD candidature.

My studies would have been more difficult if I had not received an Australian Postgraduate Award (APA) scholarship that financially supported me for the first 3.5 years of my studies. I am also thankful to my friend and PhD colleague, Amir Antonir, which offered me a job at his IT company. With Amir's generous salary, I was able to self-sponsor my PhD studies for 1.5 years.

I would also like to express my gratitude to my MSc supervisor, Dr. Michalis Psarakis and my friend Aitzan Sari. Through their mentorship and guidance I gained invaluable knowledge and skills, such as how to estimate the vulnerability of SRAM FPGA circuits with analytical and fault-injection techniques, which I used in this thesis.

Many thanks go to Dr. Antonio Miele who provided me with a floorplanning tool in order to investigate how floorplanning affects the soft error sensitivity of Triple Modular Redundant (TMR) FPGA circuits.

I am glad that I made meaningful and lasting friends at UNSW. These are Amir Antonir, Dr. Hongyan Rong, Dr. Mubashir Hussain, Hassaan Saadat and Dr. Arash Shaghghi. I am lucky that I met them during my PhD studies.

I am thankful to my friend Hassaan Saadat which referred me to Xilinx Research Labs at Singapore. It was a great opportunity for me to gain industry experience at Xilinx during the last 6 months of my PhD studies.

Many thanks go to my friends, Dr. Vangelis Tasoulas and Tasos Vereses who helped set up several Linux servers when conducting the experiments for this thesis.

Last, I would like to express my gratitude to my parents, my sister and my niece for always giving me the emotional and financial support that I so much needed during my PhD studies.

Contents

Abstract	v
Acknowledgements	vii
Contents	ix
List of Figures	xv
List of Tables	xix
1 Introduction	1
1.1 Thesis Aims	5
1.2 Research Gaps and Thesis Contributions	5
1.3 List of Publications	11
1.4 Thesis Organisation	15
2 Background and Literature Review	16
2.1 Radiation Sources	16
2.2 Radiation Effects	17
2.2.1 Total ionizing dose effects	17
2.2.2 Single event effects	19
2.3 Fault-tolerant SRAM FPGA Circuits	20

2.3.1	Increasing reliability through circuit partitioning	22
2.3.2	Configuration memory error recovery	22
2.3.3	User memory error recovery	29
2.3.4	Mitigating common mode errors in routing resources	33
2.3.5	Computer-aided automation for fault-tolerance	34
2.4	Summary	36
3	Fast and Energy Efficient Configuration Memory Recovery	38
3.1	Introduction	38
3.2	Problem Statement	39
3.3	Dependability – Energy Consumption Models	42
3.3.1	Configuration memory architecture	42
3.3.2	Error susceptibility of modern FPGAs	44
3.3.3	Mean-time-to-recover models	46
3.3.4	Hierarchical dependability models of the SoC	47
3.3.5	Reliability and availability of SoC components	48
3.3.6	SoC design formulation	52
3.3.7	Recovery technique: Impact on SoC reliability and availability . . .	54
3.3.8	Error Recovery: Impact on SoC energy consumption	56
3.3.9	Assumptions	58
3.4	Analytical Results	60
3.4.1	Reliability results	61
3.4.2	Availability results	64
3.4.3	Energy consumption results	65
3.5	Implementation of FMER	65
3.5.1	Generating frame address lists	66

3.5.2	Obtaining data for each configuration frame address of the FPGA	67
3.6	Practicality and Applicability of FMER	68
3.6.1	Benchmarks and implementation of the SoCs	68
3.6.2	Utilised configuration frames, essential bits and resources	69
3.6.3	Dependability and energy consumption	71
3.6.4	Experimental results	74
3.7	Related Work	76
3.8	Chapter Summary	77
4	Reconfiguration Control Networks	78
4.1	Introduction	78
4.2	Overview of Reconfiguration Control Networks	80
4.3	RCN Architecture	82
4.3.1	Star RCN	82
4.3.2	Bus RCN	83
4.3.3	Token-ring RCN	83
4.3.4	ICAP RCN	84
4.4	Latency of each RCN type	86
4.5	Reliability Analysis	86
4.6	Fault Emulation System	88
4.7	Experiments and Results	90
4.7.1	Experimental Methodology	91
4.7.2	Implementation results of the synthetic layout designs	92
4.7.3	Fault-injection results	94
4.7.4	RUSH case study results	94
4.8	Related Work	96

4.8.1	Star-based RCNs	96
4.8.2	Bus-based RCNs	97
4.8.3	Token-ring RCNs	97
4.8.4	Configuration-layer RCNs	98
4.9	Chapter Summary	98
5	TLegUp: High-Level Synthesis of TMR FPGA circuits	100
5.1	Introduction	100
5.2	Background	105
5.2.1	LegUp high-level synthesis flow	105
5.2.2	LegUp design methodology	107
5.2.3	Architecture and RTL hierarchy of pure hardware LegUp generated designs	108
5.3	Challenges of Partitioning LegUp Generated RTL Designs	110
5.4	Architecture of the TLegUp Toolflow	117
5.4.1	Front-end	119
5.4.2	Back-end	124
5.5	Experimental Methodology	126
5.5.1	Resource utilisation, performance, resource balance and implementation time	126
5.5.2	Estimating the soft error sensitivity	128
5.5.3	Configuration of the tool flow	128
5.6	Experimental Results	129
5.6.1	Simplex circuits	129
5.6.2	TMR circuits	129
5.6.3	Resource balancing	131
5.6.4	Utilization and performance	132

5.6.5	Soft error sensitivity	134
5.7	Chapter Summary	134
6	Conclusions	137
6.1	Thesis Summary and Concluding Remarks	138
6.2	Future Research Directions	141
	References	142

List of Figures

1.1	Simplified model of an FPGA architecture [9].	2
2.1	Occurrence of SEUs in the TAOS mission [60, 88].	18
2.2	Triple modular redundancy.	20
2.3	A fully triplicated SRAM FPGA circuit.	21
2.4	Partitioning a TMR FPGA circuit.	22
2.5	$R(t)$ for $t \in [0, 3000]$ seconds, $\lambda = 0.001, \mu = 0.1$	23
2.6	A conceptual model of a Xilinx FPGA [48].	24
2.7	Conceptual model of a TMR-based FPGA system with MER.	28
2.8	Voter with both fault masking and fault localisation capabilities.	28
2.9	Simplex circuit with a registered loop.	30
2.10	TMR circuit without synchronisation voters.	31
2.11	TMR circuit with synchronisation voters.	32
2.12	Scrubbing BRAMs in a TMR circuit [101].	33
2.13	Possible CMFs caused by a single bit upset [118].	34
3.1	(a) Possible sub-systems in an FPGA-based SoC, (b) FPGA-based SoC design formulation.	40
3.2	Simplified layout of a Xilinx FPGA.	43

3.3	Markov-chain dependability models for the <i>types</i> of components encountered in sub-systems $a_1 - a_4$	48
3.4	Expressing the availability by two terms.	50
3.5	Comparison of $R(t)$ for $t \in [0,3000]$ seconds between Eq. (3.6) and the Markov model of [73].	59
3.6	Reliability, availability and energy consumption results.	62
4.1	Architectural layout of TMR FPGA SoCs with MER or FMER.	80
4.2	Architecture of a star-based RCN.	82
4.3	Architecture of a bus-based RCN.	83
4.4	Architecture of a token-ring based RCN.	84
4.5	Extract of a Xilinx logic allocation file.	85
4.6	Fault-injection flowchart.	89
4.7	Input stimulus of each network terminal in the SoCs.	90
4.8	a) Synthetic layout of a 31-voter design and b) RUSH floorplan.	91
4.9	a) Unprotected RCN b) TMR triplicated c) TMR triplicated with recovery.	95
4.10	Conceptual models of RCN topologies.	97
5.1	Typical design flow for generating a TMR design version of an HLS program.	103
5.2	Implementation of a 2-bit ripple adder on a Xilinx Virtex FPGA [54].	104
5.3	LegUp high-level synthesis flow.	106
5.4	LegUp design methodology: (a) Pure hardware implementation; (b) Hardware/software implementation.	108
5.5	Architecture of a Verilog module in LegUp generated designs.	109
5.6	(a) Call graph of an HLS C program. (b) Architecture and RTL hierarchy of a pure hardware LegUp generated design.	110
5.7	(a) Architecture of a TLegUp generated design with all C function calls inlined, (b) Architecture of a TLegUp generated design that is partitioned with ILP.	112

5.8	(a) RTL hierarchy of a random design, (b) Modifying the RTL hierarchy, (c) FPGA floorplanning layout.	113
5.9	(a) Function call graph of an HLS application, (b) TLegUp RTL hierarchy output when the HLS application is partitioned with ILP for $k = 2$	114
5.10	Floorplanning an ILP partitioned design: (a) Conceptual floorplanning layout, (b) FPGA floorplanning layout	115
5.11	An FLP partitioned design with non-flattened RTL hierarchy.	116
5.12	An FLP partitioned design with flattened RTL hierarchy.	116
5.13	Floorplanning an FLP partitioned design.	117
5.14	Memory hierarchy of the LegUp generated design shown in Fig. 5.6(b).	118
5.15	The TLegUp toolflow.	118
5.16	TLegUp generated TMR code and voter locations.	119
5.17	The architecture of the TLegUp ILP front-end.	120
5.18	(a) A sequence of IR instructions forming a registered loop, (b) Scheduling table of the IR instructions.	122
5.19	(a) Datapath of the IR instructions, (b) Graph for capturing the clock slack of instructions between registers in a cycle.	122
5.20	The FLP design flow of TLegUp.	124
5.21	The architecture of the floorplanning flow.	125
5.22	Number of inserted voters, resource utilization and maximum frequency of TMR circuits relative to their corresponding simplex circuits.	133

List of Tables

1.1	Configuration and user memory bits within the Kintex-7 325T device [136]	4
1.2	Correlation between contributions, publications and chapters	14
2.1	Truth table of the voter shown in Fig. 2.8.	29
3.1	Upset rate per configuration bit per second	44
3.2	Reliability functions of SoC/FMER, SoC/Scrub, SoC/MER and SoC/NR. .	56
3.3	Availability functions of SoC/FMER, SoC/Scrub, SoC/MER and SoC/NR.	56
3.4	Number of CFs, essential bits (Ebit) and resource utilization of the Pblocks and SRs for each SoC	70
3.5	R(t), A(t) and energy consumption for a 2-year LEO mission	75
4.1	Results of implementing four type of RCNs on a Xilinx Artix-7 XC7A200 FPGA	93
4.2	Average number of functional errors (FEs)	94
4.3	Results of implementing 9 TMR components on a Xilinx Artix-7 XC7A200 FPGA	95
5.1	Naming convention used to describe the designs that were implemented on the Artix-7 FPGA.	127
5.2	Resource utilisation, performance, SES and implementation time of the simplex circuits.	130
5.3	Implementation time for the TMR circuits.	130

5.4	Coefficients of variation (standard deviation/average) of resource balance (LUTs and CFs) between partitions.	132
5.5	Normalized soft error sensitivity (SES) of the TMR circuits.	135

List of Abbreviations

ASIC	Application Specific Integrated Circuit
ANSI	American National Standards Institute
AVF	Architectural Vulnerability Factor
BLTMR	BYU-LANL partial Triple Modular Redundancy
BRAM	Block Random Access Memory
CAD	Computer-Aided Design
CED	Concurrent Error Detection
CF	Configuration Frame
CLB	Configuration Logic Block
CFDATA	Configuration Frame Data
CFG	Control Flow Graph
CM	Configuration Memory
CMF	Common Mode Failure
CMOS	Complementary Metal Oxide Semiconductor
COTS	Commercial Off The Shelf
CRC	Cyclic Redundancy Check
CV	Coefficient of Variation
DCE	Domain Crossing Error
DFG	Data Flow Graph
DMR	Dual Modular Redundancy
DFS	Depth First Search

DPR Dynamic Partial Reconfiguration

DSP Digital Signal Processor

DUT Design Under Test

ECC Error Correction Code

EDIF Electronic Design Interchange Format

ER Error Recovery

FAD Frame Address

FAR Frame Address Register

FLP Function Level Partitioning

FDIR Fault Detection, Isolation and Recovery

FMER Frame- and Module-based Configuration Memory Error Recovery

FPGA Filed-Programmable Gate Array

FSM Finite State Machine

GEO GEosynchronous Orbit

GPS Global Positioning System

GRM General Routing Matrix

HDL Hardware Description Language

HLS High Level Synthesis

IC Integrated Circuit

ICAP Internal Configuration Access Port

IDF Isolation Design Flow

ILP Instruction Level Partitioning

IO Input Output

IOB Input/Output Block

IP Intellectual Property

IR Intermediate Representation

ISS International Space Station

IT Implementation Time

ITAR International Traffic in Arms Regulations

LEO Low Earth Orbit

LLVM Low Level Virtual Machine

LUT Look-Up table

MB MircoBlaze

MBU Multi-Bit Upset

MER Module-based Configuration Memory Error Recovery

MIPS Million Instructions Per Second

MTTD Mean Time To Detect

MTTR Mean Time To Recover

NASA National Aeronautics and Space Administration

NC Network Controller

NT Network Terminal

PIP Programmable Interconnect Point

PL Programmable Logic

PT Programmable Tile

PV Partitioning Voter

RAM Random Access Memory

RC Reconfiguration Controller

RCN Reconfiguration Control Network

RR Reconfiguration Request

RTL Register Transfer Level

RUSH Rapid recovery from SEUs in Reconfigurable Hardware

RV Reducing Voter

SBU Single-Bit Upset

SECDDED Single-Error Correction, Double-Error Detection

SEFI Single-Event Functional Interrupt

SEL Single Event Latch-up

SEM Soft Error Mitigation
SES Soft Error Sensitivity
SET Single Event Transient
SEU Single Event Upset
SoC System-on-Chip
SPF Single Point Of Failure
SRAM Static Random Access Memory
SV Synchronisation Voter
TID Total Ionizing Dose
TMR Triple Modular Redundancy
UM User Memory
XTMR Xilinx TMRtool

Chapter 1

Introduction

In contrast to most terrestrial computing systems, spacecraft and satellite electronics need to operate in a harsh environment and adhere to strict requirements, such as high dependability and performance, subject to limited available resources, e.g., power, mass and size [40, 85, 123]. This significantly increases the time and costs associated with the development and maintenance of space-grade computing systems. Modern satellite instruments can generate more than 3 Gbit of raw data per minute at a peak data rate of 300 Mbit per second, therefore requiring some form of on-board processing to reduce the transmitted data volume to base stations on Earth and save communications bandwidth [62, 77, 83]. Unfortunately, the low power-performance efficiency of space-qualified general-purpose processors [11] limits their use in such high-throughput space applications [70]. An alternative solution is to identify which processing tasks are computationally intensive and optimise their data-path with custom hardware in order to perform operations faster and more energy efficiently than using processors exclusively [136]. The most efficient solution, in terms of power and performance, to realise custom hardware are Application-Specific Integrated Circuits (ASICs). However, the high development cost of ASICs and the inability to modify their architecture after fabrication mitigates their economic viability for deployment in many missions, especially those designed for low-cost nano-/microsatellites, such as CubeSats.

An attractive alternative solution to ASICs is the use of Field-Programmable Gate Arrays (FPGAs) [77, 135]. FPGAs are programmable integrated circuits that, within resource limitations, can implement any digital circuit after their fabrication [9, 103]. Fig. 1.1 depicts a simplified model of an FPGA, which consists of an array of Programmable Tiles (PTs), such as Configurable Logic Blocks (CLBs), Block RAMs (BRAMs), Digital

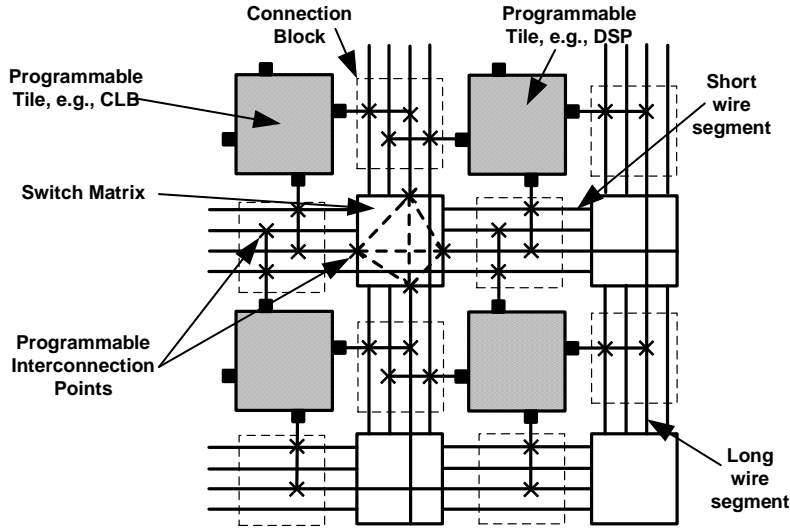


Figure 1.1: Simplified model of an FPGA architecture [9].

Signal Processing blocks (DSPs) and Input/Output Blocks (IOBs), that are embedded in a very dense matrix of programmable routing resources called the General Routing Matrix (GRM). The GRM consists of switch matrices, connection blocks, and horizontal and vertical wire segments of various lengths. Both switch matrices and connection blocks contain several programmable interconnection points (PIPs), whereby each PIP is simply a programmable switch that, when enabled, connects the source with the sink of two wire segments. Signals between the PTs are routed by enabling and disabling PIPs in the switch matrices and connection blocks of the FPGA.

In a nutshell, the PTs of the FPGA implement the logic functions of a design, which are thereafter interconnected with the device's GRM to form the final digital circuit [9, 103]. Computer-aided design (CAD) tools synthesise, place and route a design – that is typically specified in a Hardware Description Language (HDL) or a High-Level Synthesis (HLS) language – on a target FPGA architecture, and finally translate the post-routed netlist of the design into a collection of configuration bits, namely the configuration bitstream. The bitstream is thereafter loaded in-situ into the FPGA to realise the digital circuit, e.g., by appropriately configuring the device's PTs and GRM. The development cost of an FPGA system is significantly lower than for a corresponding ASIC, because there is no need to produce expensive photolithographic masks when implementing or modifying a circuit [124] – a new circuit can be simply realised by loading a new bitstream into the FPGA.

Mainstream FPGAs can be classified into one-time-programmable and reprogrammable

devices, whereby the former commonly use antifuse-based memory to store the configuration of their programmable resources, while the latter use flash- or static RAM-based memory [77]. Static RAM-based FPGAs, or in short SRAM FPGAs, have two distinct differences when compared with their flash-based counterparts. Firstly, they can be re-configured hundreds of thousands of times, and secondly, many of these devices support Dynamic Partial Reconfiguration (DPR), that is, a portion of the FPGA’s programmable resources can be selectively reprogrammed during circuit operation. Both virtually unlimited reconfiguration and DPR are very useful when implementing space computing architectures. For example, hardware modules that do not need to operate simultaneously during a mission can be realised with the same programmable resources of the FPGA at runtime in a time-division-multiplexed manner in order to save energy, mass and size [47, 62, 68, 69].

Although, all three types of the anformentioned FPGAs are used in space, this thesis focuses on SRAM FPGAs, and to be more specific on the state-of-the-art Xilinx 7-series FPGAs. SRAM FPGAs give scientists and engineers the freedom to upgrade satellite computing systems as many times as needed in order to support new mission requirements [136] as well as to avoid any permanent failure in the FPGA by migrating/re-mapping logic from faulty programmable resources to unutilised ones [129]. Nevertheless, along with the benefits of implementing space computing systems with SRAM FPGAs also come some challenges. Space lacks the Earth’s magnetosphere and atmosphere to protect electronics from radiation, in other words, from energy transmitted to them, predominantly from the Sun, in the form of waves or particles.

Integrated circuits, such as SRAM FPGAs, which embed a large amount of SRAM in a relatively small area of silicon, are particularly sensitive to high radiation [88]. One or more SRAM cells in the chip may become corrupted if a charged particle (e.g., a heavy ion) strikes the device’s memory cells with sufficient energy to revert their state, i.e., to change the stored value of one (or more) SRAM cells from 0 to 1 or vice-versa [55]. Such a radiation effect is called a Single Event Upset (SEU) or a soft-error because the corrupted memory can be recovered simply by overwriting (reprogramming) the memory with its initial value [55]. The occurrence of SEUs in electronics is not a recent observation. In fact, the possibility of SEUs occurring in electronic systems operating in a high radiation environment was shown theoretically in 1962 [131], while SEUs were first observed in an operational satellite in 1975 [10]. Since 1975, SEUs have become a major concern when designing electronic systems for use in high radiation environments.

SRAM FPGAs use SRAM cells to store: 1) the configuration of their programmable

resources, referred to as configuration memory (CM), 2) the application’s state and data, referred to as user memory (UM), and 3) the state of various resources essential for the vital functioning of the FPGA itself, referred to as Internal Proprietary State¹ (IPS) [97, 115, 136]. For example, Table 1.1 shows how 92 million user accessible bits of a state-of-the-art Xilinx Kintex-7 325T FPGA are shared across its most important programmable resources [136]. An SEU occurring in any of these memory types is equivalent to a *fault* which in many cases may manifest as a circuit malfunction or otherwise an *error*. This means that all errors are caused by faults but there are faults that do not cause any errors. For example, a UM upset (i.e., a fault) in an unutilised BRAM of an FPGA circuit will not cause an error since it is not used, but a CM upset that disconnects the clock from the circuit will certainly cause an error. *Fault-tolerant* design techniques aim to mask those faults in the circuit that will cause an error if they occur, that is, to prevent faults from becoming errors [6, 59, 79]. The number of successfully detected faults divided by the number of potential faults in a circuit is referred to as *fault-coverage*.

Table 1.1: Configuration and user memory bits within the Kintex-7 325T device [136]

Memory Type		# Bits	%
Configuration Memory*		72,868,672	79.3%
User Memory	Block RAM	18,661,568	20.3%
	User Flip-Flops	407,600	0.4%
Total		91,937,840	100.0%

*5.6% (i.e, 4,096K) of the FPGA’s configuration bits can be used as distributed RAM, i.e., as user memory.

Although, space-grade SRAM FPGAs [144] exist and have higher immunity to radiation-effects, these devices have lower logic capacity and performance than commercial off-the-shelf (COTS) SRAM FPGAs, are more expensive [115], and are restricted through export control regulations, such as the US International Traffic in Arms Regulations (ITAR). Traditionally, the space industry has implemented electronic systems with space-grade components that can withstand the harmful radiation of space. However, the advancement of fault-tolerant design techniques has enabled the implementation of much cheaper, equally effective electronic systems using COTS components.

A noticeable example of building a space application using exclusively COTS components is the COTS Ultra-high frequency Communication Unit (CUCU) avionics system, which

¹The ISP is not visible to the user and it accounts only for a very small amount of the total memory bits of the FPGA [136].

was built at the Space exploration Technologies (SpaceX) corporation [110]. CUCU is installed inside the International Space Station (ISS) in order to assist ISS crew members in monitoring and commanding the approach and departure of the SpaceX Dragon spacecraft during cargo delivery missions to the ISS [109]. Interestingly, the CUCU passed NASA's test protocol at the first try, and as stated in [128], it cost approximately US\$10K. If, however, CUCU had been implemented with space-grade components, it could have cost as much as US\$10M [128]. The US\$10M figure may seem an overstatement, but it is worth noting that only a single space-graded processor in a space computing system can cost a few hundred thousand US dollars. For example, the cost of a mainstream space-graded processor, the RAD750, which operates at 166MHz and achieves 300 Million Instructions Per Second (MIPS), is approximately US\$200K [11].

1.1 Thesis Aims

This thesis aims to develop fault-tolerant design techniques and CAD tools that will help researchers and practitioners implement low cost dependable circuits with COTS Xilinx FPGAs.

In more detail, this thesis aims to:

1. Develop techniques that will recover soft-errors in Triple Modular Redundant (TMR) FPGA circuits [26] in less time and with less energy than current state-of-the-art error recovery mechanisms;
2. Develop a CAD toolflow that will triplicate FPGA designs for Xilinx 7-series FPGAs with high-level synthesis techniques;
3. Thoroughly evaluate the performance of the proposed error recovery techniques and CAD tools by implementing and testing a rich set of TMR FPGA circuits.

1.2 Research Gaps and Thesis Contributions

In the following we present the contributions of this thesis. We provide necessary background so that the reader can understand which research gaps we have attempted to address in this thesis. Readers that are new to the field of fault-tolerant FPGA circuits are however advised to first read Chapter 2 before proceeding to our contributions.

As mentioned earlier, SRAM FPGAs are well suited for accelerating computational tasks in space applications without the non-recurring engineering costs of manufacturing ASICs. However, their inherent sensitivity to user and configuration memory upsets reduces their usefulness when employed in high-radiation environments. Fortunately, several fault-tolerant design techniques, such as state-machine encoding, quadded logic, and temporal redundancy, can mitigate radiation-induced faults in FPGA circuits [78]. However, none of these techniques provide greater reliability benefits than TMR and in many cases have higher area overhead than TMR [78]. TMR design techniques implement a circuit three times and the result of each replicated circuit (also called *TMR domain* or *module* of the TMR scheme) is processed by a majority voter circuit [26]. The majority voter, or in short voter, simply masks any erroneous result from a faulty TMR domain by outputting the result corresponding to at least two of its inputs.

A TMR circuit is often referred to as a 2-out-of-3 redundant system, which means that the output of the circuit becomes erroneous when two or all three of its TMR domains fail. Such a situation can arise for various reasons, e.g., Common Mode Failures (CMFs) that occur from a design bug in all three TMR domains of the circuit [61], or from a CM upset in a routing resource that inadvertently connects or disconnects wires between multiple TMR domains [25, 118]. Nevertheless, even when no CMFs exist in the TMR circuit, the possibility still exists that a succession of soft-errors causes a second and a third TMR domain to fail. This can be alleviated in many ways, of which the following are common:

- *Circuit partitioning*: Instead of triplicating a circuit as a whole and voting on the output of its three TMR domains, the circuit is first partitioned into a logical series network of $k \in \mathbb{Z}_{\geq 1}$ smaller components and thereafter each of these k components is triplicated and voted upon [73]. In this way, the partitioned TMR circuit can function correctly even when every one of its k partitions (i.e., TMR components) are operating with one faulty TMR domain. In other words, the more partitions a TMR circuit has the more errors it can withstand and operate with correctly, assuming that these errors do not affect more than one TMR domain per partition.
- *Error recovery mechanisms*: By incorporating rapid UM and CM error recovery mechanisms in order to repair a faulty TMR domain before the second or the third domain of the TMR scheme fail.

SEUs in the UM and the CM of the FPGA are commonly recovered by periodically refreshing their contents, e.g., by applying UM scrubbing [26, 66] and CM scrubbing [21, 48], respectively. However, TMR FPGA circuits that require lower error detection latency and

energy consumption than periodic CM scrubbing affords often resolve to Module-based CM Error Recovery (MER) [17,30,86]. With MER, each TMR domain is implemented as a reconfigurable region [100], also known as a reconfigurable Pblock, according to Xilinx terminology, and is partially reconfigured when it experiences a functional error. To be more specific, MER determines that UM or CM upsets have occurred in the device by executing lock-step comparison between the outputs of TMR domains or, in other words, by implementing Concurrent Error Detection (CED) mechanisms [76]. CED mechanisms can easily be realised in TMR circuits by enhancing the voters with additional logic, such as comparators, that reports which module, in the minority, is experiencing ongoing errors [34]. Therefore, the voters in TMR circuits with MER have two outputs; one output for the *majority result* and one output for the *minority report*, i.e., a report to indicate which of the three modules of the TMR circuit, if any, needs reconfiguration.

Research Gap 1 (RG1):

A closer look at TMR FPGA circuits that utilise either device periodic scrubbing or MER to recover from faults reveals the following limitations that have not been previously studied. On the one hand, periodic CM scrubbing does not take advantage of the rapid error detection and localisation capabilities provided by CED mechanisms. Therefore, periodic scrubbing spends energy and time scanning for CM upsets in the TMR modules of the circuit although this information can be inferred with CED mechanisms. On the other hand, MER utilises the circuit's voters to rapidly detect erroneous modules, but is unable to detect any radiation-induced errors in logic that is instantiated outside module boundaries, i.e., outside Pblocks. FPGA circuits that employ MER therefore become unavailable in the long run due to an accumulation of SEUs in this unprotected programmable logic of the device. Additionally, there are situations where only a portion of the circuit can be triplicated, either because programmable resources are not sufficient in the device, e.g., limited Input/Output Block (IOB) pins, or because it is simply difficult, or sometimes even impossible, to triplicate complex resources, such as the high-speed transceivers of an FPGA [22]. Therefore, FPGA circuits that employ MER cannot use CED mechanisms to detect errors in the simplex logic of the design.

Contribution 1 (C1):

The *first contribution* of this thesis addresses RG1 by proposing a hybrid CM error recovery mechanism, which we have called Frame- and Module-based CM Error Recovery (FMER). FMER achieves the fault-coverage of device-periodic CM scrubbing alone, but with less error recovery latency and energy consumption.

In particular, FMER uses MER to rapidly and efficiently recover the CM of the Pblocks hosting the modules of the TMR circuit, and also uses selective periodic scrubbing to recover the CM of logic located outside the Pblocks, e.g., the interconnection nets between the Pblocks. We derive models for estimating the reliability, availability and energy consumption of partitioned TMR FPGA circuits that incorporate either device periodic scrubbing, MER, FMER, or no recovery at all. By exploring our models under several parameters, such as the length and the orbit of the mission, the number of circuit partitions, and the incorporated CM error recovery mechanism, we show that FMER is beneficial for missions with high reliability or availability requirements subject to a low energy budget. It should be noted that the CM error recovery models we derive in order to quantify the energy consumption of MER and periodic scrubbing in partitioned TMR circuits have not been reported in the literature. We therefore believe these to be a contribution in their own right.

Research Gap 2 (RG2):

Two very important components in TMR FPGA circuits that incorporate MER or FMER are the Reconfiguration Control Network (RCN) [15, 29, 120] and the Reconfiguration Controller (RC) [41, 48]. The RCN simply transfers the minority reports from the voters of the TMR circuit to the RC of the FPGA in order to inform which modules, if any, need reconfiguration. As might be expected, the overall reliability of a TMR circuit with MER, and therefore with FMER, depends significantly on the reliability of both the RCN and the RC. If the RCN fails, the minority report of a faulty module may never be conveyed to the RC. Similarly, if the RC fails, a reported faulty module may never be reconfigured. Either way, the overall reliability of the FPGA circuit will be compromised. The research community has proposed reliable RCs [41, 48], but has neither considered the design of reliable RCNs, nor has investigated how the latency and reliability of an RCN affects the overall reliability of a TMR circuit with MER.

Contribution (C2):

The *second contribution* of this thesis aims to address RG2 by implementing and comparing several feasible RCN topologies, e.g, a star, a bus and a token-ring RCN topology, in terms of reliability, scalability and performance, while also investigating the benefits of implementing an RCN with the *configuration layer* [52] of the FPGA.

In more detail, we have implemented a configuration layer RCN that stores the 2-bit minority report from each voter in the circuit into a 2-bit register (i.e., two flip-flops). In turn, the RC takes advantage of the *Readback Capture* feature of Xilinx 4–7 series

FPGAs [139, 142] to capture the state of all registered minority reports in the CM of the FPGA and then it selectively reads back the CM containing these reports. The probability of having radiation-induced faults in the configuration layer RCN is thus significantly reduced since almost no SRAM configurable resources are used in the implementation of the RCN; the state of each minority report in the TMR circuit is simply read back throughout the configuration layer of the FPGA. Our experimental results show that, of the RCNs studied, the configuration layer RCN is the most reliable despite having a higher, yet practically acceptable latency.

Research Gap 3 (RG3):

Achieving high reliability in FPGA circuits, however, requires in addition to state-of-the-art error recovery mechanisms, experienced engineers to produce high quality TMR designs. Although, the concept of partitioning and triplicating a circuit may feel simple, in practice it is a relatively complex procedure. Most real-life circuits contain several Finite State Machines (FSMs) that are formed with registered loop structures, that is, the output of state registers are fed back to logic instantiated before these registers. To guard against UM errors becoming trapped in these register loops and to allow for self-synchronisation between TMR domains, so-called *synchronization* voters need to be inserted into optimal locations of the circuit in order to “cut” all feedback paths that form registered loops while conserving FPGA resources and circuit performance [54]. In fact, synchronization voters play two important roles. Firstly, they overwrite any incorrect state of a TMR domain with the majority state result of the TMR scheme, and secondly, since UM errors are not trapped in TMR domains, repeated erroneous output results from a TMR domain are indicative of a CM error, and therefore can be used to trigger CM recovery in FPGA circuits [31].

It quickly becomes evident that manually partitioning and triplicating an FPGA design as well as finding optimal locations to insert voters is not a trivial task. Fortunately, CAD tools, such as the BYU-LANL TMR (BL-TMR) [20], R4R [14] and XTMR [141], exist that take as an input an RTL or post-synthesised FPGA design and automatically produce a TMR version of the design. Triplicating a design during RTL pre- or post synthesis, though, has the following limitations that have not been previously investigated.

- The flexibility of pipelining the TMR design in order to mitigate the performance overheads of voter insertion and hardware redundancy is limited. Inserting registers into critical paths requires the CAD tools to modify the design’s FSMs (i.e., the timing), which in turn changes the specifications of the design.

- Partitioning and triplicating a design at the post-synthesis level is complex because the CAD tools need to be fully aware of all details of the targeted FPGA architecture, which limits the flexibility of conducting fast design exploration and of supporting a broad range of FPGAs. For example, although inserting synchronization voters between the nets of high-speed carry chains in Xilinx Virtex slices may cut the feedback paths in a TMR ripple carry adder, it will also negate the performance gains of utilising the high-speed carry chains [54].

Contribution 3 (C3):

The *third and last contribution* of this thesis aims to address RG3 by triplicating an FPGA design during HLS rather than RTL pre- or post-synthesis.

To be more concrete, we have developed a toolflow, namely TLegUp, that automatically compiles an ANSI C program to a partitioned TMR RTL design and thereafter implements and floorplans the design on a Xilinx Artix-7 FPGA. In fact, TLegUp leverages a front-end part for HLS and a back-end part for synthesising and implementing the generated RTL on a target FPGA. The front-end part has been developed on top of the open-source LegUp HLS research framework [23] and it performs partitioning and triplication within the Low-Level Virtual Machine [63] compiler Intermediate Representation (IR) of the design, before allocation, binding and scheduling take place. On the other hand, the back-end incorporates the Vivado design suite and an academic tool [100] in order to implement and floorplan the RTL, respectively. Floorplanning a TMR design firstly minimises resource sharing between TMR domains, which in turn minimises the probability of CMFs, and secondly, it aids designers to rapidly realise MER or FMER mechanisms in the FPGA circuit.

By conducting design triplication during HLS we believe that the limitations presented in RG3 are addressed for the following reasons:

- The control- and data-path of the final TMR RTL design are synthesised rather than modified as done when triplication is performed at the RTL pre- or post-synthesis level. TLegUp extends the delay of LLVM IR instructions that are to be voted upon before performing HLS binding and scheduling. In this way, the HLS tools account for the delay of voter insertion during binding and scheduling, and automatically pipeline and retime the design in order to meet the targeted timing requirements.
- Partitioning and triplicating a design in the LLVM IR level is simpler than doing so in the more detailed RTL pre- or post synthesis netlist level. This gives the

opportunity to broadly explore various design parameters, e.g., targeted operating frequency or number of partitions, in less time than doing so in RTL. Alternative versions of a TMR design are rapidly synthesised and explored in HLS, while the low-level architectural details of the targeted FPGA are taken care of with the Vivado tools during RTL synthesis and optimisation.

We implemented a fine- and a coarse-grain approach to partitioning the design into k TMR components. The fine-grained approach uses a network flow algorithm to partition the application's Data Flow Graph (DFG) at the instruction level, which as our experiments indicated produced TMR circuits with up to 14.6x area overhead. Needless to say, the fine-grained approach was a research dead-end, which motivated us to find a more practical solution – the Function Level Partitioning (FLP) approach. As its name suggests, FLP partitions the design at the C function level rather than at the instruction level and is therefore able to generate TMR circuits with much less area overhead and higher performance.

We synthesised and implemented with TLegUp, TMR design versions of 17 HLS benchmarks on a Xilinx Artix-7 200T FPGA in order to compare their characteristics with non-triplicated counterpart designs which were produced with the official HLS tools of the LegUp framework. When comparing the TLegUp generated TMR circuits with their simplex counterparts, the triplicated circuits that were partitioned at the C function level for $k = 1$ and $k = 2$ had on average:

- 500x reduction in soft error sensitivity, which was further reduced by a factor of 1.3x when the designs were floorplanned;
- 3–4x more resources; and
- 11% and 13% frequency drop off for the non-floorplanned and floorplanned circuits, respectively.

1.3 List of Publications

Research for this thesis was conducted in the period August 2014 - March 2018 at the School of Computer Science and Engineering, UNSW Sydney. The Contributions C1 - C3 of this thesis are presented in Chapters 3 - 5, respectively, and are based on the following publications:

- **Publication 1 (P1):** D. Agiakatsikas, E. Cetin, and O. Diessel, “FMER: A hybrid configuration memory error recovery scheme for highly reliable FPGA SoCs”, in *International Conference on Field Programmable Logic and Applications (FPL)*, Lausanne, 2016, pp. 1–4. [Online]. Available: <https://doi.org/10.1109/FPL.2016.7577339>
- **Publication 2 (P2):** D. Agiakatsikas, E. Cetin, O. Diessel, “FMER: an energy efficient error recovery methodology for SRAM-based FPGA designs”, *IEEE Transactions on Aerospace and Electronic Systems (TAES)*, 2018, vol. 54, no. 6, pp. 2695–2712. [Online]. Available: <https://doi.org/10.1109/taes.2018.2828201>
- **Publication 3 (P3):** D. Agiakatsikas, N. T. H. Nguyen, Z. Zhao, T.Wu, E. Cetin, O. Diessel, and L. Gong, “Reconfiguration control networks for TMR systems with module-based recovery,” in *IEEE International Symposium on Field Programmable Custom Computing Machines (FCCM)*, Washington DC, 2016, pp. 88–91. [Online]. Available: <https://doi.org/10.1109/fccm.2016.30>
- **Publication 4 (P4):** N. T. H. Nguyen, D. Agiakatsikas, Z. Zhao, T.Wu, E. Cetin, O. Diessel, and L. Gong, “Reconfiguration control networks for FPGA-based TMR systems with modular error recovery,” in *Microprocessors and Microsystems*, 2018, pp. 86–95. [Online]. Available: <https://doi.org/10.1016/j.micpro.2018.04.006>
- **Publication 5 (P5):** G. Lee, D. Agiakatsikas, T. Wu, E. Cetin and O. Diessel, “TLegUp: A TMR code generation tool for SRAM-based FPGA applications using HLS”, in *IEEE International Symposium on Field Programmable Custom Computing Machines (FCCM)*, Napa, CA, 2017, pp. 129–132. [Online]. Available: <https://doi.org/10.1109/fccm.2017.57>
- **Publication 6 (P6):** D. Agiakatsikas, G. Lee, T. Mitchell, E. Cetin and O. Diessel, “From C to fault-tolerant FPGA-based systems”, in *IEEE International Symposium on Field Programmable Custom Computing Machines (FCCM)*, Boulder, CA, 2018, pp. 1–1 [Online]. Available: <https://doi.org/10.1109/FCCM.2018.00046> (**poster**).

Table 1.2 shows the relevant publications and chapters where C1 – C3 contributions are presented. Details about the publications P1 – P6 as well as the contributions of this author in each of these publications are provided below:

- P1, P2: Preliminary results of FMER were presented in the conference paper P1, while the journal P2 extended P1 by providing the following additional information: 1) The derivation of reliability, availability and energy consumption models for TMR FPGA circuits, 2) a more thorough discussion of the assumptions made in the derivation of the dependability models, 3) a discussion of the CM upset rate in Xilinx Artix-7 FPGAs, 4) the implementation of FMER was detailed, and 5) the practicality and applicability of FMER was demonstrated through the implementation of several triplicated HLS applications on an Artix-7 200T FPGA. Finally, a related work section was added to P2. This author wrote the manuscripts of P1 and P2, which were reviewed by his supervisor Dr. Oliver Diessel and his co-supervisor Dr. Ediz Cetin. The limitation that MER is unable to recover any logic located outside the TMR modules of a TMR FPGA circuit was pointed out by this author. The idea of combining periodic selective scrubbing with MER in order to overcome the limitation of classic MER was proposed by this author. The idea of providing dependability and energy consumption models for TMR FPGA circuits that incorporate either FMER, MER or device periodic scrubbing so that their benefits and limitations can be better understood was proposed by Dr. Oliver Diessel. The dependability and energy consumption models were derived by this author. The experimental methodology of P1 and P2 was planned and executed by this author. The tools required to analyse the essential bits of the HLS case study designs in P2 were developed by this author.
- P3, P4: Initial results of the configuration-layer RCN were presented in the conference paper P3, while the journal P4 provided the following additional information: 1) A background on common CM error recovery techniques such as periodic scrubbing and MER, 2) an overview of TMR FPGA circuits with MER as well as discussion about the advantages and disadvantages of MER compared with periodic CM scrubbing, 3) a literature survey of RCN topologies, and 4) fault injection experiments. The idea of implementing a configuration-layer RCN in order to reduce the RCN's resource utilisation and therefore its failure rate was proposed by Dr. Oliver Diessel. This author wrote approximately 50% of P1, namely, the abstract, 50% of the "introduction" section, 100% of the "reliability evaluation" section as well as 50% of the "experiments and results" section. This author contributed to the derivation of the reliability models in P3 and P4, most of which were based on models derived in P1 and P2. The CM failure rate of the Xilinx Artix-7 FPGA was estimated by this author and was used in P1 – P4. This author planned most of the experimental methodology in P3. The tools used to analyse the essential bits and the number of CM frames in the modules of the TMR components in the RUSH payload were

developed by this author. In P4, this author mainly contributed to the experimental methodology, the development of fault-injection tools, as well as, reviewing the manuscript.

- P5, P6: The initial version of TLegUp was presented in P5. The main contributions of this author to P5 were: 1) The planning of the experimental methodology, 2) the development of tools for conducting fault-injection experiments, 3) the implementation of TMR designs on a Xilinx Virtex-6 FPGA, 4) the collection and analysis of results, and 5) writing approximately 70% the of the manuscript. The initial idea of TLegUp was proposed by Dr. Oliver Diessel. The front-end of TLegUp and many underlying algorithms were developed by Dr. Ganghee Lee. Mr. Tong Wu, Mr. Thomas Mitchell, and Dr. Ganghee Lee assisted with the experiments of P5 and P6 as well. The manuscript of P5 and P6 were mainly reviewed by Dr. Oliver Diessel and Dr. Ediz Cetin. The idea of partitioning the TMR designs at the C function level was proposed by this author as was the idea of investigating how floorplanning improves the reliability of the TMR circuits. The main contributions of this author to P6 were the following: 1) wrote the manuscript, 2) planned the experimental methodology, 3) implemented several TLegUp generated TMR designs on an Artix-7 FPGA and collected/analysed the results, 4) developed with Mr. Thomas Mitchell the back-end of TLegUp, and 5) developed tools to inject-faults into the essential bits of the designs in order to speed up the experiments.

Table 1.2: Correlation between contributions, publications and chapters

Contribution	Publications	Chapter
C1	P1, P2	3
C2	P3, P4	4
C3	P5, P6	5

Other co-authored publications during my PhD candidature that are not included as contributions in this thesis are listed below:

- **Publication 7 (P7):** Z. Zhao, **D. Agiakatsikas**, N. T. H. Nguyen, E. Cetin and O. Diessel, “Fine-grained module-based error recovery in FPGA-based TMR systems”, *International Conference on Field-Programmable Technology (FPT)*, Xi’an, 2016, pp. 101-108. [Online]. Available: <https://doi.org/10.1109/fpt.2016.7929433>
- **Publication 8 (P8):** Z. Zhao, N. T. H. Nguyen, **D. Agiakatsikas**, G. Lee, E. Cetin and O. Diessel, “Fine-grained module-based error recovery in FPGA-based

TMR Systems”, *ACM Reconfigurable Technology and Systems (TRETS)*, vol. 11, no. 1, March 2018, pp. 4:1–4:23. [Online]. Available: <https://doi.org/10.1145/3173549>

- **Publication 9 (P9):** N. T. H. Nguyen, **D. Agiakatsikas**, E. Cetin and O. Diessel, “Dynamic scheduling of voter checks in FPGA-based TMR systems”, *International Conference on Field-Programmable Technology (FPT)*, Xi’an, 2016, pp. 169-172. [Online]. Available: <https://doi.org/10.1109/fpt.2016.7929525>
- **Publication 10 (P10):** L. Gong, T. Wu, N.T.H. Nguyen, **D. Agiakatsikas**, Z. Zhao, E. Cetin and O. Diessel, “A programmable configuration controller for fault-tolerant applications”, *International Conference on Field-Programmable Technology (FPT)*, Xi’an, 2016, pp. 117-124. [Online]. Available: <https://doi.org/10.1109/fpt.2016.7929515>
- **Publication 11 (P11):** L. Gong, A. Kroh, **D. Agiakatsikas**, N. T. H. Nguyen, E. Cetin and O. Diessel, “Reliable SEU monitoring and recovery using a programmable configuration controller”, *International Conference on Field Programmable Logic and Applications (FPL)*, Ghent, 2017, pp. 1-6. [Online]. Available: <https://doi.org/10.23919/fpl.2017.8056798>

1.4 Thesis Organisation

The background and literature review for this thesis is provided in Chapter 2, while Chapters 3 – 5 present the FMER, the RCN, and the TLegUp HLS flow studies, respectively. The last chapter concludes this thesis.

Chapter 2

Background and Literature Review

High energy particles, such as protons, electrons, and heavy ions in space cause several negative effects in the electronics of satellites and spacecraft.

This chapter firstly provides a background of space radiation and its effects on SRAM FPGAs and secondly presents in detail mainstream fault-tolerant design techniques and CAD tools used to mitigate soft-errors in SRAM FPGA circuits.

2.1 Radiation Sources

The energy of charged particles in space commonly ranges from keV to GeV and causes a number of problems when they interact with spacecraft electronics [115]. The primary sources of radiation in space are the following [88].

- **Sun:** Protons, electrons and heavy ions emitted from the sun in a solar energetic particle event during solar flares. Solar flares can last from a few hours to several days.
- **Radiation belts:** Trapped particles in radiation belts around planets, which consist mainly from protons and electrons. Satellites orbiting over the South Atlantic Anomaly (SAA) – the red area of the map shown in Fig. 2.1 (a) where the Earth's inner Van Allen radiation belt comes closest to the Earth's surface – are exposed

to high radiation, which commonly consists of energetic protons at 10 MeV. For example, Fig. 2.1 (b) shows the locations of 1300 detected SEUs within the on-board computer of the Autonomous Operational Survivability (TAOS) satellite. Nearly 50% of the total SEUs occurred over the SAA region, although only 5% of orbital time was spend there [60, 88]. Similarly, results from the Mission Response Module (MRM) satellite payload that hosted four Xilinx Virtex-4 FPGAs and was deployed in Low Earth Orbit (LEO) showed that the FPGAs experienced most SEUs when orbiting over the SAA [97].

- **Outer space:** Galactic cosmic rays consisting of high-energized protons, electrons, and fully ionized nuclei coming from outer space toward the Earth.

It is worth mentioning that direct ionization from protons usually does not upset an SRAM cell. Only approximately one in 10^5 protons causes nuclear reactions in silicon which in turn produces heavy ions capable of upsetting the cell [88].

2.2 Radiation Effects

Radiation effects on electronics are classified as either Total Ionizing Dose effects (TID) or Single Event Effects (SEEs). TID considers the long-term effects of radiation on an electronic device, while SEEs describe the instantaneous response of the device to a highly energised particle strike. The following provides an overview of radiation effects on SRAM FPGAs.

2.2.1 Total ionizing dose effects

TID is the amount of ionizing radiation a device can accumulate before failing to meet its published specification and is commonly measured in krad. As ionizing radiation accumulates in an FPGA, the electrical characteristics of its transistors degrade due to increasing leakage currents and other effects [115]. TID effects slow down the transistors of the FPGA, which in turn cause several problems, such as operating frequency drop off and power consumption increase [127]. In more detail, as radiation accumulates in a CMOS transistor, the required current to switch “on” or “off” the transistor increases until the point at which it fails to switch in any way [96]. Similarly, the required voltage to switch on an N-channel MOSFET decreases as TID increases. When TID exceeds a certain point the MOSFET remains permanently switched on [84].

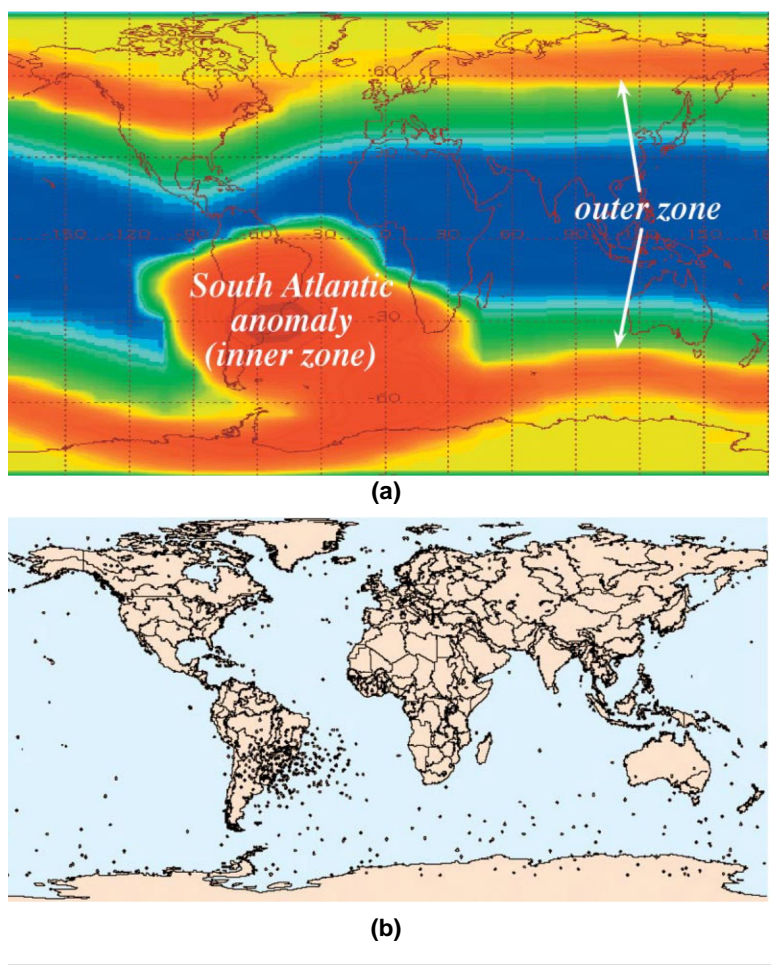


Figure 2.1: Occurrence of SEUs in the TAOS mission [60, 88].

The amount of ionizing radiation an FPGA absorbs during a mission depends on many factors, such as the length of the mission, the location of the FPGA in the satellite, the thickness of the satellite's shielding, as well as the satellite's location in space [96]. Different FPGAs can withstand different levels of TID and therefore engineers have to choose the most suitable device for a given mission. Space-graded SRAM FPGAs can withstand a large amount of TID and therefore can be used in high-radiation environments for extended time periods. For example, the Xilinx Virtex-5QV FPGA can accumulate up to 1 Mrad TID and still operate reliably in space. However, space computing systems that are expected to operate in orbits with lower radiation levels, say in LEO, or for shorter time periods, say 2 years, can use COTS FPGAs. In fact, COTS SRAM FPGAs can withstand more than 100 krad TID, while many satellites, especially those operating close to earth are exposed to only 1 – 5 krad per year [96]. For example, the Netherlands-China Low-

frequency Explorer (NCLE) mission incorporates a payload that embeds a state-of-the-art Xilinx Kintex-7 325T FPGA. Radiation experiments revealed that the Kintex-7 FPGA can withstand a total dose of 340 krad when irradiated with 180 MeV protons [137], while the expected total dose of the NCLE mission is estimated to be less than 10 krad [127].

2.2.2 Single event effects

Of the many types of SEEs that occur in SRAM FPGA circuits, the most common are:

- **Single Event Upset (SEU)** is a change of the logic state of one memory cell, called Single Bit Upset (SBU), or multiple memory cells, called Multi-Bit Upsets (MBU). An SEU occurs when a charged particle, such as a heavy ion or a proton, strikes the semiconductor device with sufficient energy to create a charge capable of reverting the state of one or more of its memory cells [55, 133].
- **Single Event Transient (SET)** is a temporary voltage variation or glitch in the FPGA's programmable logic that propagates through the FPGA circuit and can either disappear after some time or become an SEU if it is latched by a memory element [96, 115].
- **Single Event Latchup (SEL)** is a radiation-induced latchup, where the parasitic thyristor (PNPN structure) in CMOS is turned on from a particle strike, which in turn shorts the power rails of the CMOS. If an FPGA experiencing a SEL is not powered off promptly, its current consumption can increase beyond device specifications and cause a permanent failure [96]. Radiation experiments show that space-grade Xilinx Virtex-5QV FPGAs as well as some COTS Xilinx FPGAs, such as the Kintex-7 325T, are immune to SELs at linear energy transfers of more than 100 MeV [65, 144].
- **Single Event Functional Interrupt (SEFI)** is an SEU or SET in the Internal Proprietary State (IPS) of the FPGA, which can be recovered by resetting or reconfiguring the device. The effects of SEFIs in SRAM FPGAs include reset or shutdown of the device, malfunctions in the configuration circuitry of the FPGA, and other failures that put in danger a mission when not mitigated [96]. Fortunately, the cross section of the ISP logic in an SRAM FPGA is small and therefore the probability of the device experiencing SEFIs during most space missions is low [99].

2.3 Fault-tolerant SRAM FPGA Circuits

Over the last two decades, novel techniques have been proposed to mitigate the negative effects of radiation in both space-grade and COTS SRAM FPGA circuits [26, 57, 98, 115, 136]. Additionally, several CAD tools have been developed to simplify the application of these techniques [16, 39, 91, 111, 141]. Most fault-tolerant techniques insert some form of temporal or spatial redundancy into the design in order to mask functional errors from the user. Of these, spatial TMR [26] is the most popular technique used to mask Single Point of Failure (SPF) in FPGA circuits. TMR FPGA circuits are commonly combined with error recovery mechanisms since they drastically improve their reliability [78].

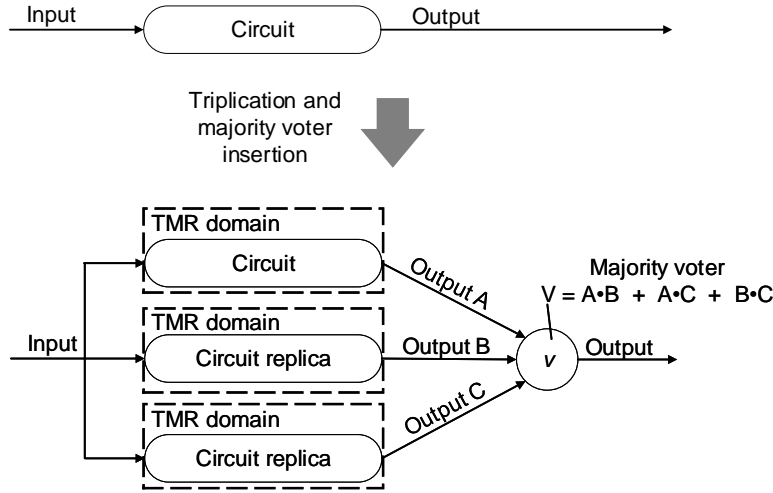


Figure 2.2: Triple modular redundancy.

The basic concept of TMR is to replicate a circuit three times, provide the three identical circuits with the same input stimulus, and perform bit-wise majority voting on the outputs of each circuit replica [59] as shown in Fig. 2.2. The majority voter (V) simply masks any erroneous result from a faulty TMR by outputting the result corresponding to at least two of its inputs. In other words, if A , B and C are the outputs of the three circuit replicas, respectively, then the result of the voter is determined by the logic function $V = A \cdot B + B \cdot C + A \cdot C$. As mentioned in Chapter 1, each circuit replica is referred to as *TMR domain* or module of the TMR scheme.

However, the TMR scheme of Fig. 2.2 is more suited to ASICs, where only the UM of the circuit can be affected by SEUs since logic such as the voter, the routing wires and the input/output (I/O) ports are hardwired and therefore immune to SEUs. In contrast,

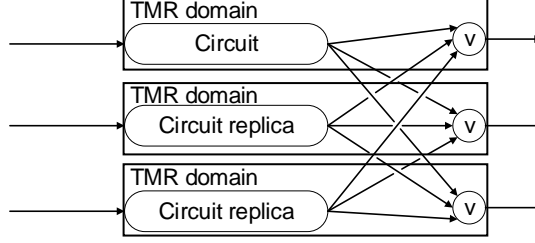


Figure 2.3: A fully triplicated SRAM FPGA circuit.

the functionality of most logic in an SRAM FPGA circuit is specified via the device’s CM, which means that the entire design needs to be triplicated, including the voters, the routing wires and the I/O ports as shown in Fig. 2.3.

A fully triplicated scheme guards the FPGA circuit against Single Point(s) of Failure (SPF), or in other words, errors originating from one TMR domain at a time, but there is a possibility that failures in multiple TMR domains will overcome the TMR scheme [94,118]. Multiple failures in multiple TMR domains are referred to as Domain Crossing Errors (DCEs). Common causes of DCEs in FPGA circuits include:

- SEUs accumulate in the device over time causing multiple TMR domains to fail;
- Multi-bit upsets (MBUs) cause two or more TMR domains to fail simultaneously [94];
- CMFs in routing resources of the FPGA circuit cause DCEs [25,118]; and
- When mapping circuits onto an FPGA, many inputs of the programmable resources need to be tied with fixed logic one or logic zero constants. SRAM FPGAs commonly use half-latches to provide these logic constants to the circuit, which are susceptible to SEUs. In contrast to other PL, half-latches cannot be directly programmed from the FPGA’s CM and therefore require a device power cycle to re-initialise their state when corrupted by SEUs [43]. Fortunately, from Xilinx Virtex-II Pro FPGAs on, SEUs in half-latches have only transient effects because their state recovers naturally due to current leakage [141].

In the following sections we discuss how DCEs can be mitigated in SRAM FPGA circuits.

2.3.1 Increasing reliability through circuit partitioning

TMR is a *2-out-of-3* redundancy scheme, which means that a TMR circuit can withstand faults in only one TMR domain at a time. However, if the same circuit is partitioned into k smaller TMR components, as shown in Fig. 2.4, it can then mask faults in k TMR domains, assuming that each partition (i.e., TMR component) has no more than one faulty TMR domain at a time. The more partitions a TMR FPGA has, the less the likelihood of soft errors affecting the one TMR component and the higher the total reliability of the circuit. However, when k becomes very large the benefits of circuit partitioning are overwhelmed by the area and performance overheads of the added voters and additional routing resources used between the TMR components.

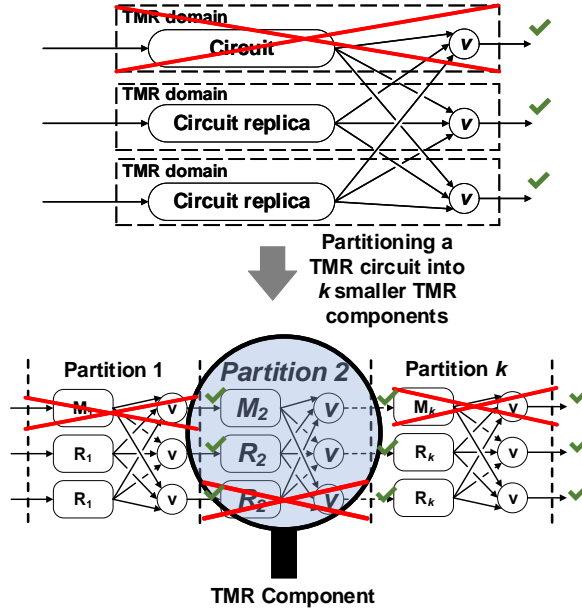


Figure 2.4: Partitioning a TMR FPGA circuit.

2.3.2 Configuration memory error recovery

The reliability of a TMR FPGA circuit increases considerably when it is combined with robust error recovery mechanisms that effectively detect and correct soft errors in both the CM [17, 31, 48, 72, 87, 105] and the UM [26, 66, 101] of the FPGA. In fact, when soft errors are not repaired in a TMR FPGA circuit, its reliability is only higher than its simplex (i.e., non-triplicated) functionally equivalent circuit in its early operation [59, 112, 132].

The reason for this counter-intuitive fact is that once one module of the TMR circuit fails, the remaining two healthy modules have a higher probability of failing than the simplex circuit itself because the two modules of the TMR scheme expose a higher number of utilised programmable resources to radiation. However, when a TMR circuit is combined with error recovery mechanisms, the only time the circuit operates on two healthy modules alone and therefore has an increased risk of failing is whilst the faulty TMR domain is recovering.

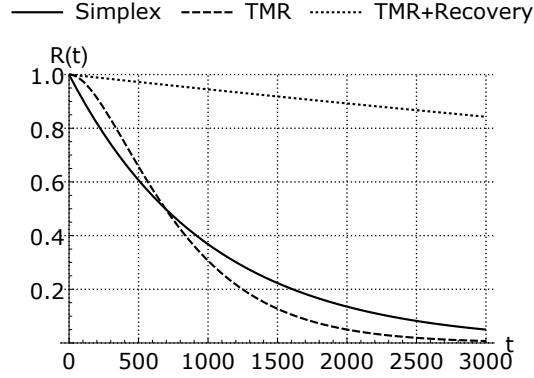


Figure 2.5: $R(t)$ for $t \in [0, 3000]$ seconds, $\lambda = 0.001$, $\mu = 0.1$

The benefits of error recovery in TMR FPGA circuits can be seen in Fig. 2.5, which compares the reliability¹ of three circuit versions for $t \in [0, 3000]$ seconds: 1) Simplex (i.e., a single circuit copy), 2) TMR with no error recovery (referred to as *TMR* in the figure) and 3) TMR with error recovery (referred to as *TMR+Recovery* in the figure). The failure rate of each module in the TMR circuits, as well as that of the simplex circuit is $\lambda = 0.001$, while the recovery rate in the TMR+recovery circuit is $\mu = 0.1$.

Most SRAM cells of an SRAM FPGA are devoted to its CM. For example, as shown in Table 1.1, 79% of the total user-accessible memory bits in the Xilinx Kintex-7 325T FPGA are devoted to its CM [136]. All these CM bits are vulnerable to SEUs and can cause functional errors when corrupted, such as by changing the functionality of a CLB or disconnecting an input from a circuit. Fortunately, although the CM of the device is large and sensitive to SEUs, most CM upsets do not result in an error. Typical FPGA circuits utilise only a fraction of their CM bits – especially those devoted to their GRM. Therefore, most upsets will occur in unutilised CM bits of the device, which commonly do not affect the circuit, while many upsets that actually corrupt utilised CM bits may cause faults that are logically masked by the circuit [79]. According to Xilinx terminology, those

¹The reliability functions for the simplex, TMR and TMR+recovery circuits are given in Eqns. (3.8), (3.11), and (3.12) of Chapter 3, respectively.

CM bits that have the potential to cause a failure when corrupted are called *essential bits*, while those that actually cause a failure are referred to as *critical bits*. Xilinx reports that on average only 5% of the CM bits in their devices are critical for a typical circuit, while in the worst case the critical bits never exceed 10% of the total CM bits [143].

Two mechanisms are mentioned in the literature for recovering SEUs in the CM of the FPGA: 1) periodic scrubbing [21, 48] and 2) Fault Detection, Isolation and Recovery (FDIR) [49, 114]. Both mechanisms correct CM errors, either by completely reconfiguring the CM of the device or by selectively reconfiguring only the portion of CM that is affected from SEUs. Either way, reconfiguration is always done by writing into the device one or more Configuration Frames (CFs), whereby a CF is the atomic unit of configuration in modern SRAM FPGAs. What differentiates one approach from the other, however, is the way CM errors are detected.

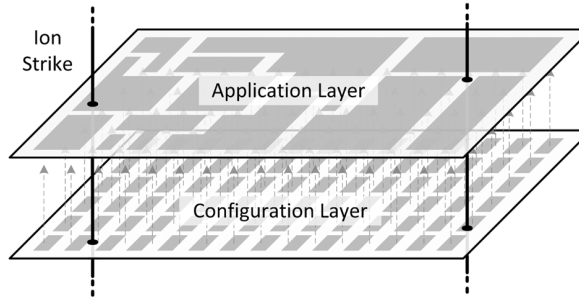


Figure 2.6: A conceptual model of a Xilinx FPGA [48].

To better understand how each mechanism detects SEUs in the CM of the FPGA we depict in Fig. 2.6 a conceptual model of an SRAM FPGA that is helpful for illustrating various radiation-induced failure modes in such devices [48, 115]. The model separates the chip into two layers, namely the CM layer that holds the configuration of the FPGA and the application layer that implements the actual circuit. A failure in the application layer may occur 1) either because highly energised particles, such as heavy ions, strike and corrupt the CM of resources responsible for the implementation of the circuit, or 2) because particles strike and corrupt UM elements in the application layer, such as utilised flip-flops and BRAMs of the circuit.

The first error recovery mechanism, referred to as CM scrubbing, operates exclusively on the CM layer in order to detect and correct CM errors [21, 48]. Scrubbing consists of reading and/or writing the CM in order to detect and/or correct bit errors. There are many ways to detect CM errors with scrubbing, from which the most fundamental and widely used are the following:

- *Cyclic Redundancy Check (CRC) readback scrubbing* periodically reads back the CM of the device and checks its CRC against that of the initial (i.e., valid) CM setting in order to detect any number of SEUs during a readback scrub cycle. Note that a scrub cycle is the completion of reading or writing a set or all CFs of the device. CRC readback scrubbing is a robust mechanism that can detect MBUs in the CM, but it takes a relatively long time to calculate the CRC for the CM, since all CFs need first to be read back from the device. For example, the maximum error detection latency of performing a CRC readback cycle on the Xilinx Kintex-7 325T is 47 milliseconds [145]. Nevertheless, a CRC mismatch at the end of a readback scrub cycle indicates that the CM is corrupted, and a complete reconfiguration of the device is performed to correct the memory.
- *Single Error Correction Double Error Detection (SECDED) readback scrubbing* checks for upsets on a frame-by-frame basis and identifies single or double adjacent bit upsets within each CF. The Mean Time to Detect (MTTD) upsets with readback SECDED scrubbing is also relatively long, since on average half of the devices CFs need to be checked before detecting a corrupted CF. However, once an upset is localised within a CF, error correction occurs promptly since only the corrupted CF needs reconfiguration.
- *SECDEC/CRC readback scrubbing* combines both SECDEC and CRC mechanisms. SECDEC localises and corrects single and double adjacent bit upsets within each CF, while a CRC check at the end of each readback scrub cycle detects any missed upsets from the SECDEC mechanism. A device reconfiguration is required to recover any MBUs detected from the CRC checking mechanism.

In addition, a technique referred to as *preventative or blind scrubbing* [48] prevents SEUs from accumulating in the device by periodically reconfiguring the CM of the FPGA, without, however, carrying out any error detection strategy whatsoever [27]. In contrast to readback scrubbing, blind scrubbing is more sensitive to SEFIs because it operates the internal configuration interface of the FPGA always in write mode. Although, SEFIs occur extremely rarely in the configuration interface of FPGAs, the whole CM of an FPGA system incorporating blind scrubbing can become jeopardised when configuration SEFIs occur [48]. On the other hand, with readback scrubbing, a SEFI in the FPGA's configuration interface will result in the corruption of only one CF, assuming that appropriate guidelines have been followed, such as to check the Frame Address Register (FAR), the status register and the control register of the FPGA before writing an CF [28, 115].

In general, periodic scrubbing is a well-established CM error recovery mechanism and

many FPGA manufactures provide a variety of CM scrubbing solutions. For example, Xilinx has embedded Error Correction Codes (ECC) in each CF from the Virtex-5 FPGAs onwards and also include hardwired logic that, when enabled, continuously performs SECDEC/CRC readback scrubbing [142] in the device. Xilinx, however, suggests to integrate their Intellectual Property (IP) Soft-Error Mitigation (SEM) scrubbing controller in circuits with high reliability and availability needs. The SEM controller uses the internal SECDEC/CRC readback scrubbing circuitry of modern Xilinx FPGAs to detect CM upsets, but it corrects the upsets with several more advanced mechanisms than the built-in logic of the FPGA, from which, the most robust is “CF replacement”. When the SEM controller operates in CF replacement mode, it replaces corrupted CFs with *golden* CF copies stored in an external radiation-hardened memory. Therefore, the latency of recovering MBUs with the SEM controller operating in CF replacement mode is lower than the built-in SECDED/CRC readback scrubbing mechanism of the FPGA, which always resorts to a device reconfiguration when MBUs are detected [142, 145].

Although, it is easy to incorporate CM scrubbing in an FPGA circuit (e.g., using the SEM controller), the error detection latency of such a solution can be prohibitive for demanding real-time applications, e.g., an automatic landing system [61]. Furthermore, continuously using a controller to read or write the CFs of the FPGA in order to implement periodic CM scrubbing increases the energy consumption in the system. Unfortunately, power availability is limited in many space applications, especially those implemented within nano- or micro-satellites, and therefore periodic scrubbing may not be possible.

Demand for faster and more energy-efficient CM error recovery mechanisms has motivated many researchers and practitioners to utilise FDIR in TMR FPGA circuits, which as we discuss in the following are more responsive to CM upsets and consume less energy than periodic scrubbing. In contrast to periodic CM scrubbing, FDIR operates on both the application and the configuration layers of the FPGA in order to detect and correct CM errors [114]. In more detail, FDIR mechanisms speculate that CM or UM upsets may have occurred in the device when the user circuit experiences ongoing functional errors. Fault detection with FDIR is typically implemented by executing lock-step comparison between the result of redundant modules or in other words with Concurrent Error Detection (CED) mechanisms [76]. Therefore, faults are detected at circuit speed, which considerably reduces the MTTF in the system. In fact, the average time to detect failures with FDIR is commonly much lower than periodic CM scrubbing, since radiation-induced faults within a module require typically only a few hundred clock cycles to propagate to its outputs [31]. This can be beneficial in TMR-based FPGA systems, since their reliability and availability depend on the time a faulty module remains unrepaired. Furthermore,

the controller for FDIR does not spend energy searching for CM upsets as the controller for periodic scrubbing does.

Only when an ongoing failure is detected in the application layer, a device- or module-level reconfiguration [17, 18, 31, 53, 86, 115] is performed to correct any CM upsets. CM error recovery occurring at *device-level* is referred to as *device FDIR*, while CM error recovery occurring at the circuit *module-level* is referred to as *MER*. MER can be advantageous over periodic scrubbing or device FDIR for the following reasons:

- The time and energy expended to detect CM upsets is less than for periodic scrubbing [18];
- CM upsets require less time and energy to recover than device FDIR, since only the CFs of faulty modules are reconfigured rather than all CFs of the FPGA; and
- The required energy and time to correct CM upsets with MER can be further reduced by making modules in TMR circuits smaller through design partitioning.

However, FPGA circuits with device FDIR or MER that do not incorporate UM error recovery mechanisms can become troublesome. Unrecovered UM errors will cause permanent circuit failures, which in turn will initiate a sequence of repeated *false* CM reconfigurations. Commonly, TMR FPGA circuits implement mechanisms in the application layer to detect and correct UM errors, e.g., utilise mechanisms that synchronise the state between TMR domains [54] or repair upsets in BRAMs [26, 75, 101]. In such systems, UM errors typically have a transient effect on the circuit, since these errors are promptly repaired through UM error recovery mechanisms, while errors due to CM upsets become permanent until reconfiguration is performed [31].

Normally, the implementation of a TMR system with MER follows a DPR design methodology [51, 147]. Fig. 2.7 depicts a typical model of a TMR FPGA system with MER, whereby the three TMR domains are implemented as dynamically reconfigurable modules (see grey coloured boxes) [17, 31, 115] or Pblocks in Xilinx terminology [147]. The interconnection nets between the Pblocks and other component in a system with MER are hosted in the static area of the FPGA circuit.

In contrast to the voters depicted in Fig. 2.3, the voters in a TMR FPGA circuit with MER are enhanced with additional logic, e.g., comparators, that identify which module in the minority is experiencing ongoing errors [34]. Such a voter is depicted in Fig. 2.8, whereby a 1-bit output provides the majority voting result (V) and a 2-bit output provides the

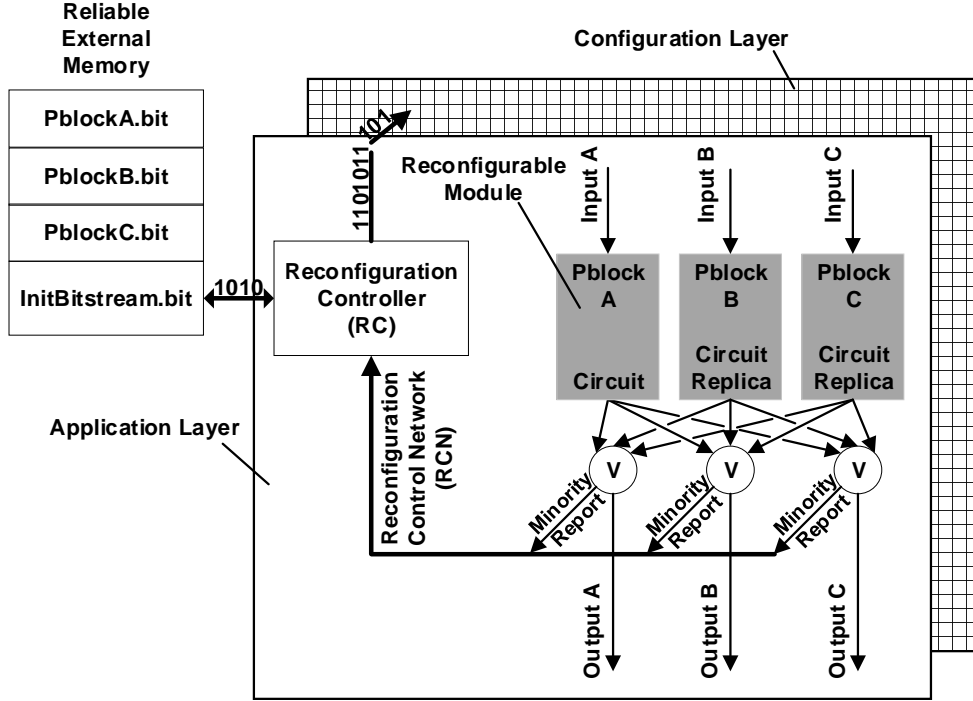


Figure 2.7: Conceptual model of a TMR-based FPGA system with MER.

minority report (E), i.e., which module, if any, is experiencing a failure. Table 2.1 shows the truth table of the voter.

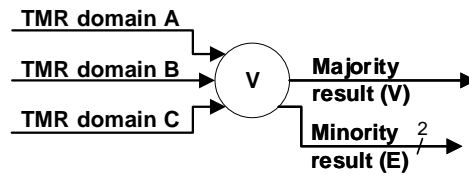


Figure 2.8: Voter with both fault masking and fault localisation capabilities.

The three input binary variables of the voter are denoted with A , B and C in the first three columns of the table, while the logic functions of V and E are shown in the fourth through sixth columns of the table, respectively. Note that the cases where A , B or C are in the minority are encoded using $E=“01”$, $E=“10”$ and $E=“11”$, respectively, while the case where all inputs agree is encoded using $E=“00”$.

Table 2.1: Truth table of the voter shown in Fig. 2.8.

A	B	C	V	E
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	1
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	0

Finally, as shown in Fig. 2.7 the minority result of the three voters is transmitted to a reconfiguration controller (RC) via a reconfiguration control network (RCN) that commonly follows a star network topology [15, 120], although more sophisticated topologies, such as a token-ring network, have been suggested [29]. As might be expected, the duty of the RC is to fetch from a reliable external memory the partial bitstream of any corrupted module (i.e., Pblock) and to write it to the device [42]. It is very important that both the RCN and the RC are reliable. A faulty module may never be repaired if either the RCN or the RC becomes corrupted. Even worse, a failure of the RC can completely jeopardise the configuration of the device. Accelerating radiation tests have shown that the more the CM becomes corrupted, the more current is drawn from the internal supply voltage (i.e., VCC_{int}) of the FPGA, which may damage the device [28]. In order to reduce the probability of such a failure, the RC itself is commonly triplicated and combined with self-recovery mechanisms [8, 41, 48]. Alternatively, the RC can be implemented on an external radiation-tolerant device for higher reliability [8, 48].

2.3.3 User memory error recovery

As mentioned earlier, in addition to CM error recovery, mechanisms are also implemented in the application layer to repair upsets in the UM of the FPGA, or in other words, upsets in flip-flops, BRAMs, and distributed RAMs. Although, flip-flops occupy the smallest portion of the total SRAM cells in the UM (see Table 1.1), soft errors occurring in these memory elements almost always result in failures since they hold important information of a circuit, such as the state of a finite-state machine (FSM) or the data of a register file. An upset within the flip-flops of an FSM, for instance, may send the FSM to a state from which it cannot return, in other words, the FSM may enter a deadlock state. Upsets in flip-flops either cause transient failures, which naturally *flush-out* of the circuit after

some clock cycles of circuit operation or cause permanent failures when they get trapped, as mentioned, in *registered feedback path loops* or in short *registered loops* [26, 54, 90, 91]. Registered loops are formed when the outputs of registers are fed back to logic instantiated prior to these registers. Such logic structures are common in sequential circuits, whose transition to a new state partly depends on their current state (e.g., in FSMs).

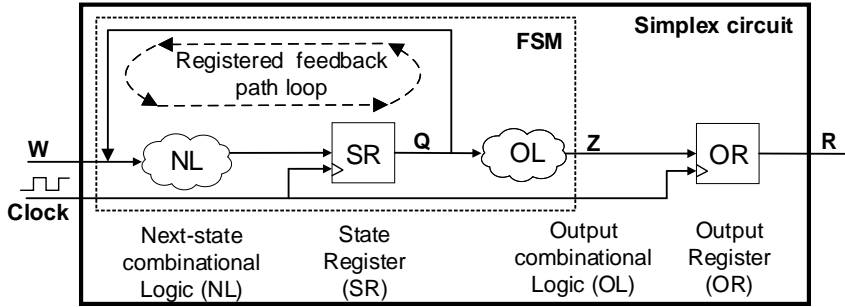


Figure 2.9: Simplex circuit with a registered loop.

To better understand how upsets within components of registered loops cause permanent failures in an FPGA circuit and how these can be recovered, it is worth looking closer at the design example of Fig. 2.9. In this design, an FSM has been formed by interconnecting three components, namely the Next-state combinational Logic (NL), the State Register (SR) and the Output combinational Logic (OL). The FSM receives a set of inputs W and produces a set of outputs Z . The current state Q of the FSM is held in the SR, which on every clock cycle is updated with the output value of the NL. The NL determines which will be the next state of the FSM according to the current state Q and the inputs W of the FSM. Finally, the OL sets the FSM's outputs Z according to the current state Q . When CM upsets corrupt the functionality of the OL, erroneous results will start propagating from this component to the outputs R of the FPGA circuit. If the FPGA circuit incorporates CM error recovery (e.g., scrubbing), the OL component will be repaired and the FSM circuit will become functional again when the Output Register (OR) is instantiated just after the OL gets updated with new *healthy* values. Similarly, the FPGA circuit will experience a failure when upsets directly corrupt the state of the OR, and will naturally recover when the state of the OR gets updated with new values.

Reasoning in a similar way, let us evaluate how the FPGA circuit behaves under radiation-induced failures in the NL or the SR components within the registered loop of the FSM. Assume that the FSM circuit transitions in a cyclic way between states A, B, C and D, and that the FSM is currently in state $Q = A$. If upsets directly corrupt the state of

the SR, say change Q from state A to state C, then on the next clock cycle the NL will send the FSM to state D rather than to the expected state B. From the time the FSM enters state B onwards, the FSM will enter states A, B, C and D at a different time than specified, namely, one clock cycle earlier. This will cause the outputs R of the circuit to be permanently erroneous. Similarly, CM upsets in the NL of the FSM may send the FSM to an unexpected state which may cause similar permanent failures in the circuit. In simplex circuits, trapped errors in registered loops can be cleared by simply resetting all registers (e.g., flip-flops) in the circuit.

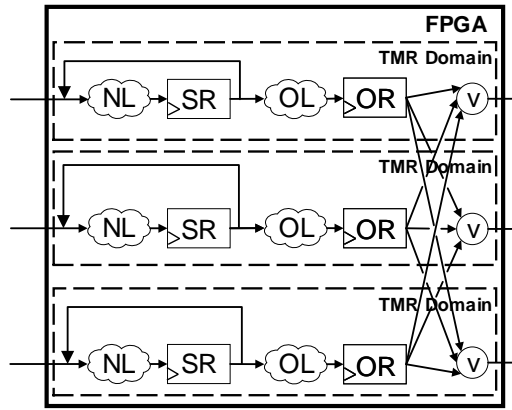


Figure 2.10: TMR circuit without synchronisation voters.

Suppose that a TMR version of the simplex design depicted in Fig. 2.9 is implemented as shown in Fig. 2.10. In this TMR design, a reset to all registers of the circuit can be applied only when the TMR scheme fails and all three modules (i.e., TMR domains) need to recover and start operation in a synchronous state. In cases where an error gets trapped in the registered loop of only one module, the module's state will be permanently desynchronised with the remaining two healthy modules of the TMR scheme. Resetting the registers of all modules will synchronise their state but will also disturb the operation of the circuit. Remember that if one of the three modules of a TMR circuit fails, the output of the circuit will be provided by the remaining two healthy modules. Therefore, the UM or CM error recovery mechanisms should correct the one faulty module without affecting the operation of the remaining two healthy modules of the component. In order to avoid such problems and allow self-synchronisation between TMR domains, voters are typically inserted within all registered loops of the circuit – commonly after flip-flops with the highest fan-in or fan-out [54].

For example, the TMR circuit of Fig. 2.11 supports state self-synchronisation since voters

have been inserted after the SRs of all modules. These so-called *synchronisation voters* update the state of desynchronised registered loops in a corrupted module with the majority voting result from the remaining two healthy modules of the TMR scheme.

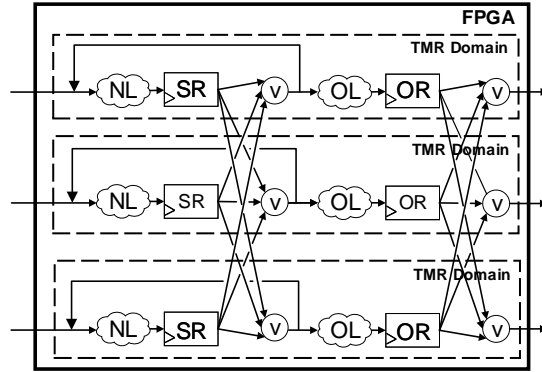


Figure 2.11: TMR circuit with synchronisation voters.

Although flip-flops are useful for storing small amounts of data in sequential circuits, such as in small buffers, synchronisers and delay lines – BRAMs and distributed RAMs are more commonly used for storing larger amounts of data, such as in FIFOs and shift-registers. Like for CM upsets, upsets in BRAMs and distributed RAMs do not always result in errors. For example, upsets occurring in unused address spaces of BRAMs will not affect the functionality of the circuit. Various mechanisms have been proposed for recovering SEUs in BRAM and distributed RAM components [66, 101]. However, in TMR circuits the most effective solution, in terms of reliability, can be achieved by periodically scrubbing the BRAMs and distributed RAMs used by the circuit.

Fig. 2.12 illustrates how BRAM scrubbing is realised within an TMR FPGA circuit implemented on a Xilinx 7-series FPGA. Note that the same scrubbing technique can be applied for distributed RAM. Xilinx 7-series devices embed true dual port BRAMs, from which, the first port is devoted to storing data for the application, while the second port is preserved for periodically scrubbing this data. In more detail, a voter is inserted after the second read port of each BRAM and the majority result of the TMR scheme is fed back into their second write ports as shown in Fig. 2.12. A scrubbing mechanism is thereafter implemented to periodically cycle through all memory addresses and repair any accumulated SEUs [26, 101], i.e., by reading data from each BRAM address and writing back the majority result from the TMR scheme. In the circuit of Fig. 2.12, scrubbing is implemented with the TMR counter and the three FSMs. More details for UM error recovery techniques can be found in Reference [101].

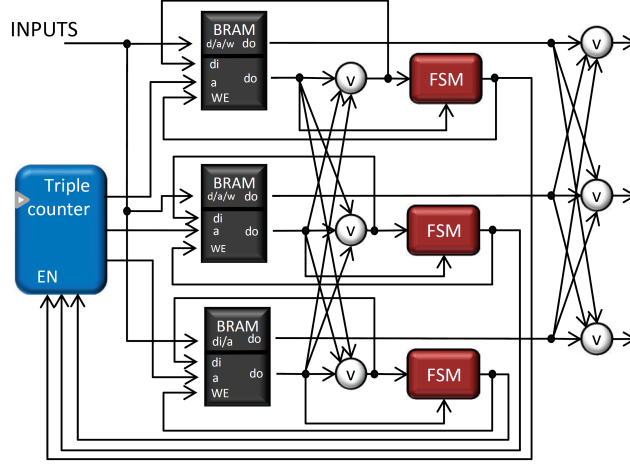


Figure 2.12: Scrubbing BRAMs in a TMR circuit [101].

2.3.4 Mitigating common mode errors in routing resources

Modern FPGA architectures have a very dense and complex GRM. In many cases, a single CM upset can enable or disable multiple PIPs in the GRM, which in turn may short or disconnect nets belonging to multiple TMR domains and compromise the TMR scheme. Although, such Common Mode Failures (CMFs) occur very rarely, their impact on the reliability of the TMR FPGA circuit is immense [25]. Sterpone et al [118] studied the effects of CMFs in the GRM and identified three possibilities: Given two pairs of connections, (A_1, A_2) and (B_1, B_2) , as shown in Fig. 2.13 (a), a single CM upset can create a short between the two pairs of connections (b), an open (disconnect) in both pairs of connections (c), or a short and an open between the pairs (d) [115, 118]. As a result, domain crossing errors (DCEs) can occur in the TMR circuit if this pair of connections belongs to two different TMR domains of the same TMR component. In a more recent work [25], Cannon et al. observed through fault-injection experiments that in Xilinx 7-series FPGA only CMFs in the routing muxes of CLB connection blocks can cause DCEs in TMR circuits when clock nets belonging to two TMR domains are corrupted. In other words, DCEs occur only due to CMFs in routing resources connecting clock nets to the slices of CLBs.

Fortunately, one or more of the following techniques can be applied during the development phase of the design in order to mitigate DCEs caused by CMFs in the routing resources of the FPGA circuit:

- By placing and routing the design in a way that avoids the use of those PIPs that

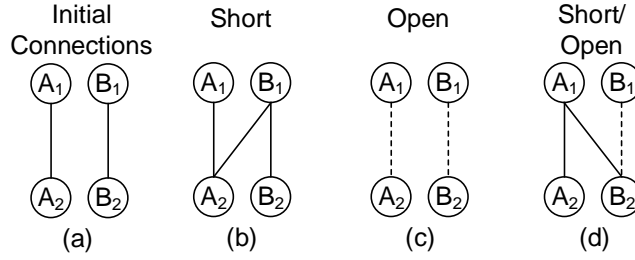


Figure 2.13: Possible CMFs caused by a single bit upset [118].

when corrupted will bridge or disconnect wire segments from multiple TMR domains [25, 118];

- By reducing as much as possible the interconnections – and therefore the utilised routing resources – between the modules [7]; and
- By physically separating the TMR domains into non-overlapping areas of the device, for example, by floorplanning the circuit, so that less programmable resources are shared between the modules [16, 45, 126]. Floorplanning also mitigates DCEs due to MBUs in the circuit, since MBUs normally affect programmable resources that are close in proximity [126].

2.3.5 Computer-aided automation for fault-tolerance

Implementing a TMR circuit on an SRAM FPGA is a difficult task, and this is further exacerbated when the application is relatively complex. Engineers may require numerous design revisions in order to properly triplicate the circuit, lengthy soft-error vulnerability analysis procedures [5, 107, 117], fault-injection campaigns [4, 12, 56, 82, 108] as well as radiation experiments [2, 22, 58] to prove that triplication has been correctly performed. Any designer trying to manually triplicate an SRAM FPGA circuit will potentially confront one or more of the following issues:

- Complex circuits, especially those that are control-oriented, contain hundreds of registered loops. Manually determining where to insert the minimum number of synchronisation voters within a design in order to cut all feedback paths is a demanding and error prone (NP-hard) task [54].
- During the synthesis and implementation phase of a design, CAD tools *fight against you* by removing necessary redundant logic of the TMR circuit in order to optimise

for area and speed. A designer needs to insert constraints, such as the Xilinx-specific *don't touch* constraints, into the design's HDL in order to prohibit CAD tools from optimising redundant logic. However, manually inserting such constraints into HDL can become troublesome. Some redundant parts of the design (e.g., signals, logic) may be missed, while other parts of the design may be over-constrained, thereby preventing CAD tools from removing any unnecessary logic;

- CAD tools tend to place and route logic from different TMR domains into nearby programmable resources in order to reduce the critical path of the circuit. This increases resource sharing between TMR domains, which, as earlier discussed, makes the circuit more vulnerable to MBUs and to CMFs in routing resources [96, 126];
- MER can be incorporated only within floorplanned TMR circuits. Floorplanning a TMR design with three modules is normally a simple task. However, floorplanning a circuit with several TMR components (i.e., partitions) requires considerably more effort and time to be done properly – especially when floorplanning needs to comply with design rule checkings for a DPR flow [100];
- Meeting mission requirements, such as reliability, recovery time, area, power and performance, in a TMR FPGA circuit can be time consuming. Various versions of the TMR circuit may have to be designed, implemented and verified, from which the most optimal will be chosen to satisfy these requirements [16].

It quickly becomes clear that some form of automation is needed to simplify the process of triplicating an FPGA design. Fortunately, several CAD tools that automatically synthesise a TMR version of a simplex design have been developed, both by academia and industry. Examples are the commercial Synopsis Simplify Premier [121], Mentor Graphics Precision Hi-Rel [74] and Xilinx TMRtool [141] products, and the academic BL-TMR [20, 91] and Reconfiguration for Reliability (R4R) [16] tools. Most of these tools analyse, modify and apply triplication of a design during the synthesis or post-synthesis phase of the CAD flow.

TMRtool, or in short, XTMR, is the most popular among the aforementioned commercial tools and has sparked the interest of many academics and practitioners. XTMR reads a post-synthesised netlist of a simplex design and generates a post-synthesised TMR netlist for the design. XTMR provides many helpful features, such as half-latches removal from Virtex and Virtex-II FPGA circuits and synchronisation voter insertion. However, it does not support the Spartan, Virtex-6 or 7-series Xilinx FPGA families.

On the other hand, BL-TMR supports Xilinx’s latest devices and is open-source [19]. BL-TMR provides interesting features, such as design partitioning and sophisticated algorithms that *cut* all feedback edges in the design by inserting less synchronisation voters than XTMR [54]. In fact, XTMR inserts a synchronisation voter after the output of every utilised flip-flop in the design in order to intersect all registered loops. However, this considerably increases the area overhead of TMR [54, 141].

XTMR and BL-TMR tools are useful for generating post-synthesised TMR designs, but the designer may need to experiment with several alternative circuit implementations in order to meet mission requirements. This motivated the development of the design space exploration tool R4R [13, 16]. R4R explores various TMR or Double Modular Redundant (DMR) schemes which: 1) have been replicated at different levels of granularity; 2) have been partitioned into a different number of TMR components (see Fig. 2.4); and 3) have been floorplanned in various ways.

Nevertheless, once a TMR design is synthesised, it needs to be mapped, placed and routed into the FPGA fabric, and in many cases, it needs to be floorplanned before placement and routing take place. It is during this phase of the CAD flow, where DCEs due to MBUs or CMFs in routing resources of the TMR design can be mitigated. For example, the authors in [16] triplicate and floorplan RTL designs with the R4R tool, localise CMFs with STAR, and guide V-Place and RoRa [116] to place and route the design in a way that allows each TMR domain to use only a set of wire segments that do not cause DCEs when corrupted. Similarly, Cannon et. al. places and routes Xilinx 7-series FPGA designs with the Vivado design suite and then go through several incremental placement modifications in a way that forces clock nets connecting the TMR domains to be routed through paths that do not cause DCEs when corrupted [25].

2.4 Summary

This chapter presented the main sources of radiation in space and explained how radiation affects electronic devices, particularly SRAM FPGAs. It then provided details about the most common fault-tolerant design techniques used for implementing high-reliability FPGA circuits as well as about various CAD tools that automate the application of these techniques.

Specifically, the reader will be able to answer the following questions after reading this chapter:

- What are the main sources of space radiation and what problems does radiation cause in SRAM FPGA circuits;
- How is TMR implemented and how it improves the reliability of the circuit;
- What is design partitioning and why it is beneficial to partition a TMR circuit into a logical series network of smaller TMR components;
- What are the most common user and configuration memory error recovery mechanisms in TMR FPGA circuits and what are their operating principles;
- What are the benefits of using MER in an TMR FPGA circuit and what is the overall architecture of such a system;
- What causes common mode errors in TMR circuits and how does floorplanning mitigate such errors;
- What are the most popular CAD tools for implementing robust TMR FPGA circuit implementations and what are their benefits and limitations.

Chapter 3

Fast and Energy Efficient Configuration Memory Recovery

3.1 Introduction

This chapter presents FMER (Frame- and Module-based configuration memory Error Recovery), a combination of selective periodic CM scrubbing and MER that achieves the fault-coverage of the traditional device-periodic CM scrubbing alone, but with lower error recovery latency and energy consumption. To demonstrate the efficacy of FMER, we derive and compare the reliability, availability and energy consumption of partitioned TMR circuits that incorporate either FMER, MER, device-periodic CM scrubbing and no recovery at all. We provide both analytical and experimental results that show the benefits of recovering CM upsets in TMR circuits with FMER.

More importantly, the derived dependability¹ and energy consumption models are helpful for quantifying the benefits of the fault-tolerant techniques presented in Chapter 2 as well as to evaluate the performance of the proposed RCN in Chapter 4. For example, in Chapter 2 we stated that MER consumes less energy than periodic CM scrubbing to recover CM upsets in an TMR circuit, but in this chapter, we quantify the energy savings of MER. In more detail, we explore various architectural parameters of the design, such as the number of partitions, its failure and recovery rate, and the type of CM error recovery mechanism used, so that the trade-off between these parameters can be studied. The

¹We use the term *dependability* to describe both the reliability and the availability of an FPGA circuit.

results of this chapter show that partitioning an TMR circuit into several smaller TMR components improves system dependability and also reduces the energy consumption when FMER or MER is used. These results motivated as to develop the TLegUp CAD tool in Chapter 5 in order to automate the procedure of implementing partitioned TMR circuits that utilize MER or FMER when recovering from CM upsets.

This chapter is organised as follows. Section 3.2 lists several limitations of recovering CM upsets in FPGA-based TMR systems with periodic CM scrubbing or MER and explains how FMER addresses these limitations. Section 3.3 derives dependability and energy consumption models for the aforementioned systems. It also presents the CM architecture of Xilinx 7-series FPGAs. Additionally, Section 3.3 estimates the expected CM upset rate of Xilinx 7-series FPGAs in several orbits, including Low Earth Orbit (LEO) and Geosynchronous Earth Orbit (GEO). In Section 3.4 we explore our models under various design parameters and provide analytical results. Section 3.5 provides the implementation details of FMER, while Section 3.6 compares the dependability and energy consumption of eleven triplicated HLS CHStone benchmarks that are implemented on an Artix-7 200T FPGA and incorporate either FMER, MER or periodic CM scrubbing. Section 3.7 provides related work while the summary of the chapter is given in the final section.

3.2 Problem Statement

Mainstream SRAM FPGAs are perfect candidate ICs for implementing System on Chip (SoC) space applications, not least because of their role in reducing the overall system mass, weight and power consumption. FPGA-based SoC applications are typically implemented by integrating several independently developed sub-systems, of which, depending upon the dependability requirements of the mission, some sub-systems are triplicated to enhance their reliability. The following paragraphs present such SoCs in more detail and point out the advantages and disadvantages of using either periodic CM scrubbing or MER for repairing CM upsets in them. Finally, the benefits of combining these two methods, as investigated in FMER, are outlined.

Note that we assume that the SoCs presented in this chapter utilise the same UM ER mechanisms as described in Subsection 2.3.3. Specifically, we assume that the TMR components of the SoCs incorporate: 1) TMR BRAMs and TMR distributed RAMs, which are combined with periodic UM scrubbing, and 2) synchronisation voters throughout all feedback paths of the triplicated components. We also assume that the simplex (non-triplicated) components of the SoCs initialise their BRAMs, distributed RAMs and flip-flops whenever

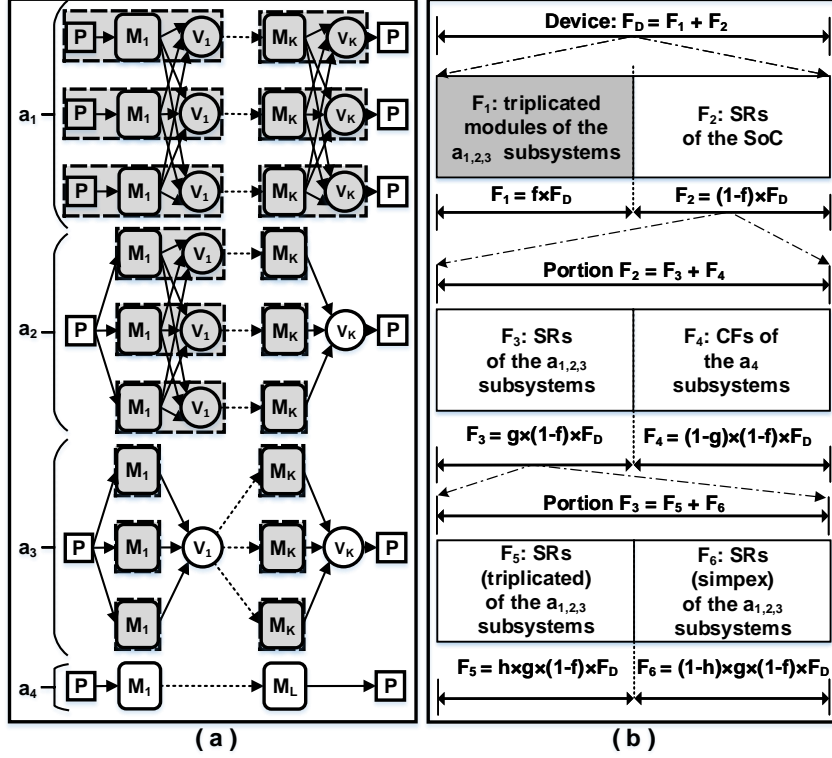


Figure 3.1: (a) Possible sub-systems in an FPGA-based SoC, (b) FPGA-based SoC design formulation.

these components recover from radiation-induced failures.

Consider an FPGA-based SoC design that is composed of a combination of some or all of the a_1 , a_2 , a_3 and a_4 sub-systems depicted in Fig. 3.1(a), where each sub-system is composed of some different number of K TMR components or L simplex components. Of the sub-systems depicted in Fig. 3.1(a) the most reliable structure is that of sub-system a_1 , whose logic is completely triplicated including the voters V and the IO pins P [26]. Although fully triplicated schemes provide better reliability than those with simplex (non-triplicated) IO pins and voters, there are situations in which this triplication is not possible, such as when the number of available pins in the device do not suffice [102]. Such a case is represented by sub-system a_2 , in which all the logic of the sub-system is triplicated except for the input pins, and the output pins with their associated voters and their interconnections. Moreover, as illustrated in sub-system a_3 , there are situations in which even the intermediate voters of a sub-system's components are not triplicated in order to decrease the cost of the fully triplicated scheme. Even worse from a reliability perspective, the SoC may include sub-systems, shown as a_4 , that are not triplicated at all, due to

performance issues, resource unavailability and inflexibility. An example of an a_4 sub-system is a hardwired high-speed transceiver, found on modern SRAM FPGAs, that is difficult to triplicate [22].

When periodic CM scrubbing, say blind scrubbing, is incorporated in the SoC model of Fig. 3.1(a), any logic of the circuit affected by CM upsets will recover after a scrub cycle. On the other hand, if MER is utilised in the SoC, any permanent functional error caused by CM upsets in any single TMR domain will be detected by its voters/comparators (see Fig. 2.8 in Sec. 2.3.2), and corrected by the RC of the SoC. However, functional errors in the following resources of the SoC will not be detected or corrected with MER: 1) the triplicated output pins, 2) the non-triplicated voters, 3) the non-triplicated IO pins, 4) the simplex components of a_4 sub-systems, and 5) all interconnections not contained inside the Pblocks, i.e., interconnections that extend into and pass through the non-shaded area of Fig. 3.1(a). We refer to those resources that are not included in the Pblocks, but that are present in the static area of the design, that is, in the non-shaded area of Fig. 3.1(a), as Support Resources (SRs).

It is evident that periodic CM scrubbing has higher CM fault-coverage than MER since CM upsets in both the modules of the TMR components and their SRs are corrected by periodically reconfiguring the CM of the device. However, as discussed in Chapter 2 periodic scrubbing has considerably higher CM error detection latency than MER. Moreover, periodic scrubbing wastes energy scanning for CM upsets in TMR circuits, although this information can be inferred by exploiting Concurrent Error Detection (CED) mechanisms with the replicated modules of the TMR scheme.

Our contribution is a hybrid CM ER mechanism, namely FMER, that combines the advantages of both scrubbing and MER; FMER periodically scrubs the SRs of the SoC until it is interrupted by a reconfiguration request from a faulty module, whereupon it recovers the module by MER before resuming its periodic scrubbing of the SRs. Therefore, FMER achieves the CM fault-coverage of scrubbing, but provides lower Mean Time To Recover (MTTR) in the SoC since errors within modules (i.e., Pblocks) are recovered immediately after they are detected. Moreover, the energy expended recovering the SRs via scrubbing is less than that used were the entire device scrubbed. We model and compare the dependability and energy consumption of four identical SoCs that are composed of the sub-systems depicted in Fig. 3.1(a), but incorporate either (a) FMER, (b) blind scrubbing, (c) MER or (d) NR (no error recovery). Firstly, this requires the derivation of the reliability and availability functions for the components of the a_1 , a_2 , a_3 and a_4 sub-systems when each component is repaired either with blind scrubbing or MER, or is

left unrecovered depending on the adopted ER mechanism in the SoC.

We explore the proposed SoC models at various radiation levels and design parameters and show that FMER affords higher reliability and availability to SoCs with lower energy consumption than obtained with classic CM scrubbing or MER.

3.3 Dependability – Energy Consumption Models

This section provides background on the CM architecture of Xilinx FPGAs and their soft-error vulnerabilities in various operating orbits. It then provides models for the MTTR of CM blind scrubbing and MER, while it formulates the reliability, availability and power consumption models of the SoC when it incorporates either blind scrubbing, MER, FMER or no recovery (NR) at all. Last, the assumptions made in the derivation of the dependability models are provided. Note that this work presents a new CM ER technique and compares the dependability of four equivalent SoCs that utilise the same user memory ER mechanisms. Therefore, our dependability analysis does not account for soft-errors in the user memory of the designs as their effect on the dependability of each SoC is the same.

3.3.1 Configuration memory architecture

This subsection introduces the CM architecture of Xilinx FPGAs [142], but similar properties and models also apply to Intel SRAM-based FPGAs. As illustrated in Fig. 3.2, the programmable logic of Xilinx FPGAs are organised in a grid of columns, with each column containing fixed numbers of specific types of resources, such as CLBs, BRAMs, DSPs, IOBs, clock management tiles (CMTs) and configuration tiles (CFGs) like the ICAP. Vertical clock routing resources (GCLKs) in the middle of the device split the FPGA resource columns into two halves, while each half is further sub-divided by equally distributed horizontal clock routing resources (HCLKs). This clock tree, consisting of GCLKs and HCLKs, divides the programmable resources of the FPGA into several rows with HCLK tiles passing through the middle of the rows, while the GCLK tiles of the FPGA divide the rows into clock regions.

The CM of the FPGA is accessed by reading or writing one or more CFs into the FPGA, with each CF spanning the height of a row. A CF consists of a unique frame address (FAD) that specifies the FPGA resources it configures, as well as the configuration data

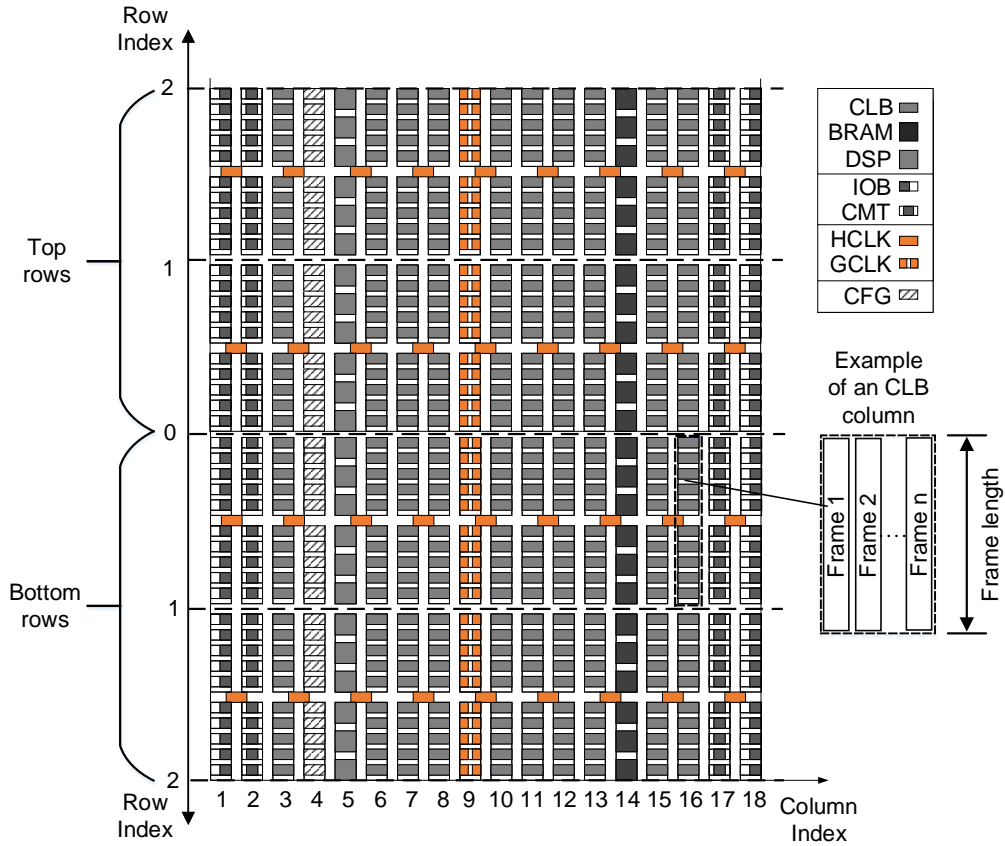


Figure 3.2: Simplified layout of a Xilinx FPGA.

(CFDATA) for these resources. The FAD in Xilinx 7-series FPGAs is 32-bits wide and it consists of 5 segments:

- **[25:23] bits** determine the block type of the CF, e.g., CLB, CLK or BRAM content;
- **[22] bit** selects the top or bottom rows of the device;
- **[21:17] bits** select the index of the top or the bottom rows of the device;
- **[16:7] bits** select the column of the row; and
- **[6:0] bits** select the CF within a column, which is referred to as its minor address [142]. The minor CFs configure the interconnection resources adjacent to the column, e.g., the General Routing Matrix (GRM) as well as the functionality of the column resources, e.g., the LUT contents of an CLB column. In addition, the number of CFs for each column depends on the column type. For example, CLB and DSP columns are configured with 36 and 28 CFs, respectively.

In a nutshell, the reconfiguration of a CF occurs by initialising the configuration process (i.e., by writing a sequence of commands to specific configuration registers of the device), setting the Frame Address Register (FAR) with the FAD, and writing the CFDATA to the frame data input register. In order to speed-up the configuration of a contiguous sequence of CFs, only the first FAD of the CF sequence is written to the FAR, as the FAR can be configured to auto-increment whenever the CFDATA of a CF is written (or read) to the frame data input register [119, 142].

The number of CFs, F_D , in an FPGA (from the same family) depends on the amount of programmable logic integrated into the device. The more programmable logic the FPGA embeds, the more CFs are used for storing their configuration, and the more time it takes to completely reconfigure the device. Each of the F_D CFs of the FPGA can be accessed via an I_B (32-, 16- or 8-bit) wide Internal Configuration Access Port (ICAP) bus when an internal RC is implemented in the FPGA SoC. Other types of configuration access ports, such as the SelectMap, are commonly used when accessing the CM of the device via an external RC. A CF is composed of B_F bits and the ideal time, t_F , needed to read or write a CF into the device depends upon the operating frequency, f_{ICAP} , of the ICAP primitive, the I_B bus width of the ICAP, and the number of bits a CF embeds:

$$t_F = \frac{1}{f_{ICAP}} \times \frac{B_F}{I_B} \quad (3.1)$$

3.3.2 Error susceptibility of modern FPGAs

The upset rate of a CM bit, λ_b , in Xilinx 7-series, specifically the Kintex-7 family, is given in Table 3.1. The results were calculated using SPENVIS [1] for Geosynchronous Equatorial Orbit (GEO), Global Positioning System (GPS) orbit and Low Earth Orbit (LEO) in order to be used as CM upset rate references in this thesis.

Table 3.1: Upset rate per configuration bit per second

Orbit Alt./Incl.	Worst Week	Worst Day	Peak 5-Min.
GEO 35,768 km / 0°	2.16E-11	7.34E-11	2.66E-10
GPS 20,200 km / 0°	1.43E-11	4,84E-11	1.75E-10
LEO (ISS) 400 km 51.60°	3.76E-14	1.10E-13	3.86E-13

In deriving the figures of Table 3.1, we used the Worst Week, Worst Day and Peak 5-minute CREME96 models [125] and assumed a typical presence of 2.54 millimetres (0.1 inches) of aluminium shielding, while the cross section of the device was obtained from [65]. We used the cross section of a Kintex-7 FPGA to find the CM upset rates in Table 3.1 because we did not have the cross section of the Artix-7 FPGA, which is used in this thesis. However, we believe that both FPGAs (i.e., the Kintex-7 and the Artix-7) have a similar cross section since both share a unified architecture and are manufactured using the TSMC’s 28 nm High-K Metal Gate (HKMG) process [92].

To put our discussion into context and get a feeling for the expected failure rate of a simplex FPGA circuit deployed in a relatively high radiation environment, we provide the following scenario. Assume, that a simplex circuit deployed in GEO is implemented on an Artix-7 200T FPGA. According to the worst week GEO model of Table 3.1, the expected CM upset rate of the Artix-7 FPGA is:

$$\begin{aligned}\lambda_{device} &= \lambda_b \times F_D \times B_F \\ &= 2.16\text{E-}11 \times 18,300 \times 3,232 \\ &\approx 0.0013 \text{ SEUs/Device/second},\end{aligned}\tag{3.2}$$

i.e., one CM upset per 13 minutes. Although the estimated CM upset rate of the FPGA device may look relatively high, the failure rate of the circuit is much lower since as mentioned in Sec. 2.3.2, only a portion of the device’s CM cells lead to circuit failures when corrupted. For example, assuming that the circuit utilises $U = 80\%$ of the device’s programmable resources, and that it has an Architectural Vulnerability Factor (AVF) = 15% [36, 79], then the expected failure rate, $\lambda_{circuit}$, of the circuit is the following:

$$\begin{aligned}\lambda_{circuit} &= \lambda_{device} \times U \times AVF \\ &= 0.0013 \times 0.8 \times 0.15 \\ &\approx 0.00015 \text{ failures per second},\end{aligned}\tag{3.3}$$

or otherwise one failure per 1.8 hours. Note that the AVF of the FPGA circuit denotes the portion of UM and CM cells in the device that lead to errors when corrupted. As already mentioned in Sec. 2.3.2, Xilinx claims that AVF is approximately 15% for an average circuit [36], i.e., results averaged from a number of circuits that utilise more than 70% of the FPGA’s available programmable resources. However, accurate AVF estimations for an FPGA circuit can only be acquired through fault-injection testing and radiation experiments. The AVF of an FPGA circuit depends on many factors, such as the architecture of the circuit, how the circuit is placed and routed onto the FPGA, as well as the architecture of the FPGA itself.

3.3.3 Mean-time-to-recover models

This subsection provides the MTTR models of blind scrubbing and MER that are then used to derive our proposed hybrid ER technique.

Blind scrubbing

The simplest way to recover CM upsets in an SRAM FPGA circuit is to periodically reconfigure the CFs of the FPGA device with golden CF data, which are typically stored externally in a radiation-tolerant memory, e.g., in a radiation-hardened flash or an SRAM when high memory throughput is required [93]. The maximum time to repair a CM upset with blind scrubbing occurs when the n^{th} CF of the FPGA gets corrupted and the scrubber has just started to reconfigure the $n^{th} + 1$ CF of the device. Thus, the CM upset will not be corrected until after a whole scrub cycle. Hence, on average the FPGA requires half a scrub cycle to recover from an CM upset, in addition to any waiting time w that is inserted between the scrub cycles to adjust the scrubbing period:

$$MTTR = \left(\frac{F_D}{2} t_F \right) + w \quad (3.4)$$

Note that the reciprocal of Eq. (3.4) gives the rate, μ_s , at which CM upsets recover with periodic CM scrubbing.

MER

TMR FPGA SoCs inherently incorporate redundant logic into the circuit and therefore can steadily implement CED mechanisms to benefit from their low error detection latency. The time it takes to recover a faulty module in a TMR component with MER depends on the error manifestation delay, t_P , to the SoC's RC [31], and on the time required to reconfigure the F_M CFs of the Pblock hosting the faulty module. However, t_P is neglected from the calculation since typically ($t_P \ll F_M t_F$):

$$MTTR = t_P + F_M t_F \approx F_M t_F \quad (3.5)$$

Similarly, the reciprocal of Eq. (3.5) gives the rate, μ_m , at which the CM of replicated modules recover when MER is incorporated in the FPGA.

In this chapter, we demonstrate FMER by combining blind scrubbing and MER. However, FMER can be implemented by combining any type of CM scrubbing (e.g., SECCED readback scrubbing) with MER. The provided dependability and energy consumption models of FMER can support any type of CM scrubbing by simply modifying Eq. (3.4) with the MTTR model of the scrubbing method used.

3.3.4 Hierarchical dependability models of the SoC

By definition, the reliability function $R(t)$ represents the probability that a system has operated according to its specifications over the interval $[0, t]$, where $t \in \mathbb{R} \geq 0$ denotes the mission duration. In contrast, the availability function $A(t)$ is the probability of the system operating correctly at time t . When the system does not incorporate any ER mechanism, then $R(t) = A(t)$. Additionally, the steady state availability $A = \lim_{t \rightarrow \infty} A(t)$ is a useful dependability metric which estimates the probability of the system operating in the long term.

There are many ways to model the dependability of a system. Models range from simple combinatorial models that hold only under specific assumptions, such as when components in the system fail and are repaired independently, to more complex models, like Markov-chain models [104], which are able to capture these dependencies. On the other hand, one can selectively apply Markov-chain models to those components for which the accuracy of the dependability modelling would be negatively impacted if the failure or repair dependencies were not modelled. This work uses simple combinatorial models to capture the dependability of the SoC as a whole, while Markov-chain reliability and availability models are used to capture failure and repair dependencies in each sub-system's components.

In more detail, an SoC that is composed of a number of the a_1 - a_4 sub-systems depicted in Fig. 3.1(a), in which each sub-system is composed of a different number of components can be viewed as a series logical structure of components with the following combinatorial system reliability and availability functions [104]:

$$R(t) = \prod_{i=1}^K R_i^{type}(t) \times \prod_{j=1}^L R_j^{type}(t) \quad (3.6)$$

$$A(t) = \prod_{i=1}^K A_i^{type}(t) \times \prod_{j=1}^L A_j^{type}(t), \quad (3.7)$$

where variables K and L denote the total number of components that realise the $a_{1,2,3}$ (TMR-based) and a_4 (simplex-based) sub-systems respectively. The sub-products of the K and L components denote the total reliability and availability of the $a_{1,2,3}$ and the a_4 sub-systems respectively. Eqs. (3.6) & (3.7) simply state that the total reliability and availability of the SoC depends on the reliability and availability of each individual component in the $a_{1,2,3,4}$ sub-systems of the SoC.

Depending on the properties of a sub-system's component, i.e., whether or not it is trip-

licated and whether or not it recovers with MER or periodic CM scrubbing (when ER is applicable in the SoC), then the reliability and availability of each type of component in the SoC is given by one of the following *type* of functions:

- (a) **Simplex & NR:** Simplex component that does not recover from CM upsets;
- (b) **Simplex & Scrub:** Simplex component that recovers with periodic CM scrubbing;
- (c) **TMR & NR:** TMR component that does not recover from CM upsets;
- (d) **TMR & Scrub:** TMR component that recovers with periodic CM scrubbing; and
- (e) **TMR & MER:** TMR component that recovers with MER.

The following subsections provide the reliability and availability functions for the above types of components, which are derived with Markov-chain models. Moreover, Sec. 3.3.9 specifies the assumptions made for the derivation of these models and also discusses their accuracy and flexibility.

3.3.5 Reliability and availability of SoC components

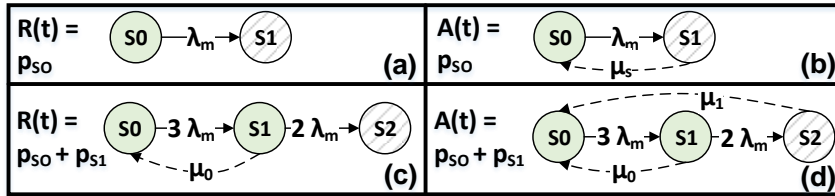


Figure 3.3: Markov-chain dependability models for the *types* of components encountered in sub-systems $a_1 - a_4$.

In this section we use continuous-time discrete-state Markov chains to derive dependability functions for the *types* of components encountered in the $a_{1,2,3,4}$ sub-systems of Fig. 3.1(a). The derived functions are thereafter substituted into Eqs. (3.6) and (3.7) to calculate the overall reliability and availability of the SoCs, respectively.

Markov chains generate a set of differential equations that are solved in order to derive the probability distribution of the chain's states, where each state represents a distinct behaviour of the modelled component. We use Mathematica [71] to apply the methodologies presented in [73], [112] and [104] in order to find the $R(t)$ and $A(t)$ functions for each

type of component. We assume that all components start in state S0 (error-free state). In other words, we assume that the initial probability distribution of the Markov chains depicted in Fig. 3.3 is $p_{S0} = 1$ and 0 elsewhere.

(a) Simplex & NR

The reliability of a simplex (i.e., non-redundant) component that has no means of recovering from errors is presented with the two states (S0: functional, S1: failed) of the Markov chain depicted in Fig. 3.3(a), where λ_m represents the failure rate with which the component (simplex module) transitions to the failed S1 state. The probability distribution of the functional state p_{S0} gives the reliability function of the Simplex & NR component:

$$R^a(t) = p_{S0} = e^{-\lambda_m t}, \quad (3.8)$$

which corresponds to the well-known reliability function of a non-redundant component [104]. The availability function of the component is also given by Eq. (3.8), i.e., $A^a(t) = R^a(t)$, since the component is never repaired.

(b) Simplex & Scrub

The reliability function of a simplex component that incorporates periodic CM scrubbing is the same as for a simplex component with no recovery at all; it can therefore be modelled with the Markov chain depicted in Fig. 3.3(a). This underscores the fact that CM scrubbing will simply recover the component when it fails, but will not avoid the occurrence of a failure. The reliability of the Simplex & Scrub component is therefore:

$$R^b(t) = R^a(t) = e^{-\lambda_m t}. \quad (3.9)$$

On the other hand, the availability of a simplex component that is periodically scrubbed is modelled with the Markov chain depicted in Fig. 3.3(b). This figure augments the reliability Markov chain of the component shown in Fig. 3.3(a) with a transition from S1 to S0 with rate μ_s , which represents the ER rate of the component. The Simplex & Scrub component is available when it is in state S0 and therefore the chain is solved for p_{S0} [112]:

$$A^b(t) = p_{S0} = \frac{\mu_s}{\lambda_m + \mu_s} + \frac{\lambda_m e^{-t(\lambda_m + \mu_s)}}{\lambda_m + \mu_s}. \quad (3.10)$$

The availability function of Eq. (3.10) is given by two terms. The first term expresses the steady-state availability of the component, while the second term expresses the transition to the component's steady state as shown in Fig. 3.4.

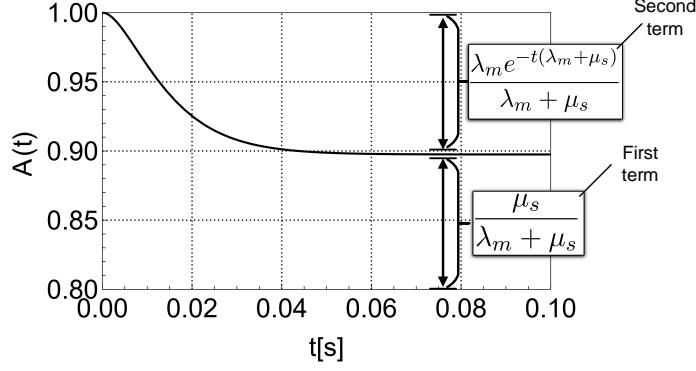


Figure 3.4: Expressing the availability by two terms.

In this thesis, all $A(t)$ functions are presented in the above form, i.e., a first term, that captures the steady state expression for the component, and a second term, that captures the transitory behaviour as it approaches steady state.

(c) TMR & NR

The reliability model of a TMR component without any form of ER is illustrated in Fig. 3.3(c). States S0 (no faulty modules) and S1 (one faulty module) represent the functional states of the component, while state S2 (two or more faulty modules) represents its failed state. If the three modules of the TMR component are identical, i.e., all modules have on average the same failure rate λ_m , then the chain transitions from S0 to S1 with rate $3\lambda_m$ since in S0 all modules are functional. Similarly, the chain transitions from S1 to S2 with rate $2\lambda_m$ since in S1 one module has already failed. The reliability function is given by summing the probability distribution of states S0 and S1, i.e., the functional states of the TMR & NR component, which yields:

$$R^c(t) = p_{S0} + p_{S1} = 3e^{-2\lambda_m t} - 2e^{-3\lambda_m t} \quad (3.11)$$

Additionally, since the component does not incorporate any recovery mechanism, then the availability of the component is also given by Eq. (3.11), i.e., $A^c(t) = R^c(t)$.

(d) TMR & Scrub or (e) TMR & MER

The reliability model of a TMR component that recovers from errors with periodic CM scrubbing or with MER is given in Fig. 3.3(c). The rate at which a module fails, λ_m , in both TMR & Scrub or TMR & MER components is the same, while the rate at which a module recovers, μ_0 , depends on the adopted ER method. As mentioned in Sec. 3.3.3, the average time to recover an error in a module with scrubbing, μ_s^{-1} , is a function of

the device's size, given in F_D CFs, and the performance of the RC, i.e., the required time, t_f , for reconfiguring a CF. Similarly, the average time to recover one module of a TMR component with MER, μ_m^{-1} , depends on the RC's performance, i.e., t_f , and on the size of the Pblock hosting the module, which is given in F_M CFs. For this reason, the same reliability Markov chain can be applied to both cases by just substituting the corresponding recovery rate for periodic CM scrubbing, $\mu_0 = \mu_s$, or for MER, $\mu_0 = \mu_m$, respectively. The probability distribution of S0 and S1 ($p_{S0} + p_{S1}$) gives the reliability of both the TMR & Scrub and the TMR & MER components:

$$R^d(t) = R^e(t) = \frac{e^{-\frac{1}{2}(at)} \left(a \sinh\left(\frac{bt}{2}\right) + b \cosh\left(\frac{bt}{2}\right) \right)}{b}, \quad (3.12)$$

where $a = 5\lambda_m + \mu_0$ and $b = \sqrt{\lambda_m^2 + 10\lambda_m\mu_0 + \mu_0^2}$, while $\mu_0 = \mu_s$ in the case of periodic CM scrubbing and $\mu_0 = \mu_m$ in the case of MER.

The availability model for a TMR component with periodic CM scrubbing is given in Fig. 3.3(d), where $\mu_0 = \mu_1 = \mu_s$ since one, two or all three modules of the component recover on average after half a scrub cycle. The probability distribution of S0 and S1 gives the availability function of the TMR & Scrub component:

$$A^d(t) = p_{S0} + p_{S1} = \frac{\mu_s(5\lambda_m + \mu_s)}{ab} + \frac{6\lambda_me^{-bt}(-2\lambda_m - \mu_s + \mu_se^{\lambda_mt} + 3\lambda_me^{\lambda_mt})}{ab}, \quad (3.13)$$

where $a = 2\lambda_m + \mu_s$ and $b = 3\lambda_m + \mu_s$.

Similarly, the availability model for a TMR component that recovers with MER is given by the Markov chain of Fig. 3.3(d), where $\mu_0 = \mu_m$ and $\mu_1 = \frac{\mu_m}{3}$, since either one or three modules need to be recovered when the component is in state S1 or S2, respectively. The availability function of the TMR & MER component is:

$$A^e(t) = p_{S0} + p_{S1} = \frac{\mu_m(5\lambda_m + \mu_m)}{b} + \frac{18\lambda_m^2 e^{-\frac{1}{6}(ct)} \left(c \sinh\left(\frac{\sqrt{a}t}{6}\right) + \sqrt{a} \cosh\left(\frac{\sqrt{a}t}{6}\right) \right)}{\sqrt{ab}} \quad (3.14)$$

where $a = 9\lambda_m^2 + 60\lambda_m\mu_m + 4\mu_m^2$, $b = 18\lambda_m^2 + 5\lambda_m\mu_m + \mu_m^2$ and $c = 15\lambda_m + 4\mu_m$.

Taking the limit of Eqs. (3.8), (3.10), (3.11), (3.13) and (3.14), as $t \rightarrow \infty$, yields the steady state availability of; (a) simplex & NR, (b) simplex & Scrub, (c) TMR & NR, (d)

TMR & Scrub, and (e) TMR & MER components respectively:

$$A^a = \lim_{t \rightarrow \infty} R^a(t) = 0 \quad (3.15)$$

$$A^b = \lim_{t \rightarrow \infty} A^b(t) = \frac{\mu_s}{\lambda_m + \mu_s} \quad (3.16)$$

$$A^c = \lim_{t \rightarrow \infty} R^c(t) = 0 \quad (3.17)$$

$$A^d = \lim_{t \rightarrow \infty} A^d(t) = \frac{\mu_s(5\lambda_m + \mu_s)}{6\lambda_m^2 + 5\lambda_m\mu_s + \mu_s^2} \quad (3.18)$$

$$A^e = \lim_{t \rightarrow \infty} A^e(t) = \frac{\mu_m(5\lambda_m + \mu_m)}{18\lambda_m^2 + 5\lambda_m\mu_m + \mu_m^2} \quad (3.19)$$

3.3.6 SoC design formulation

Fig. 3.1(b) models the distribution of CFs across the components of a TMR FPGA SoC composed with the illustrated $a_1 - a_4$ sub-systems of Fig. 3.1(a). As shown at the top of Fig. 3.1(b), the FPGA's CFs have been divided into two subsets, $F_D = \{F_1, F_2\}$ where:

- $F_1 = F_D - F_2$ CFs are devoted to mapping (implementing) the logic of the $3K$ modules (shaded dashed boxes in Fig. 3.1(a)) of the K TMR components in $a_1 - a_3$ sub-systems that can either be recovered by MER, FMER or periodic scrubbing.
- $F_2 = F_D - F_1$ CFs are devoted to mapping the SRs of the SoC, that is, the CFs of the non-shaded area in Fig. 3.1(a) that recover from CM upsets when either FMER or periodic CM scrubbing is utilised in the SoC. Note that upsets in the F_2 CFs of the SoC do not recover with MER.

Moreover, Fig. 3.1(b) shows F_2 being further subdivided into two subsets, $F_2 = \{F_3, F_4\}$ in the middle of the figure where:

- $F_3 = F_2 - F_4$ CFs are devoted to mapping the SRs for the $a_{1,2,3}$ sub-systems, e.g., their simplex IO pins, their triplicated output pins, their simplex voters and any interconnections associated with these resources.
- $F_4 = F_2 - F_3$ CFs are devoted to mapping the logic for the simplex components in a_4 sub-systems, e.g., high-speed transceivers with their IO pins and interconnection resources.

Last, F_3 is further subdivided at the bottom of the figure into two subsets, $F_3 = \{F_5, F_6\}$ where:

- $F_5 = F_3 - F_6$ CFs are devoted to mapping the triplicated SRs of the $a_{1,2,3}$ sub-systems, e.g., triplicated output pins and any triplicated routing resources between the modules of the TMR components in $a_{1,2,3}$ sub-systems.
- $F_6 = F_3 - F_5$ CFs are devoted to mapping the simplex SRs of the $a_{1,2,3}$ sub-systems, e.g., simplex IO pins, simplex voters, or simplex interconnections.

The proposed model includes three parameters, $f, g, h \in [0, 1]$ in order to distinguish between F_1 and F_2 , F_3 and F_4 , and F_5 and F_6 , respectively:

$$\begin{aligned} F_D &= F_1 + F_2 = [f \times F_D] + [(1 - f) \times F_D] \\ F_2 &= F_3 + F_4 = [g \times (1 - f) \times F_D] + [(1 - g) \times (1 - f) \times F_D] \\ F_3 &= F_5 + F_6 = [h \times g \times (1 - f) \times F_D] + [(1 - h) \times g \times (1 - f) \times F_D] \end{aligned}$$

The average number of CFs of one module in a TMR component is given by:

$$\overline{F_M} = \frac{F_1}{3 \times K} = \frac{f \times F_D}{3 \times K}, \quad (3.20)$$

i.e., the modules of each TMR component in the SoC occupy F_1/K CFs on average and therefore each individual module occupies $(F_1/K)/3$ CFs.

Additionally, it is assumed that each TMR component in the $a_{1,2,3}$ sub-systems require

$$\overline{F_{SS}} = \frac{F_6}{K} = \frac{(1 - h) \times g \times (1 - f) \times F_D}{K} \quad (3.21)$$

CFs for the implementation of their simplex SRs, while each module in the TMR components require

$$\overline{F_{TS}} = \frac{F_5}{3 \times K} = \frac{h \times g \times (1 - f) \times F_D}{3 \times K} \quad (3.22)$$

CFs for the implementation of their triplicated SRs.

The remaining CFs of the device, i.e., the F_4 CFs, implement the simplex components of the a_4 sub-systems, whereby each component with its interconnections and IO pins occupies:

$$\overline{F_{SysSRs}} = \frac{F_4}{L} = \frac{(1 - g) \times (1 - f) \times F_D}{L} \quad (3.23)$$

CFs on average.

Moreover, the total CM upset rate of the device is modelled as follows:

$$\lambda_D = F_D \times B_F \times \lambda_b, \quad (3.24)$$

where as we mentioned earlier, F_D denotes the total number of CFs in the FPGA device, B_F denotes the bits per CF, while λ_b denotes the upset rate per CM bit.

Consequently, the average failure rates of logic implemented with the $\overline{F_M}$, $\overline{F_{SS}}$, $\overline{F_{TS}}$ and $\overline{F_{SysSRs}}$ CFs are:

$$\overline{\lambda_m} = \frac{f \times \lambda_D}{3 \times K} \times U_M \times AVF, \quad (3.25)$$

$$\overline{\lambda_{SS}} = \frac{(1-h) \times g \times (1-f) \times \lambda_D}{K} \times U_S \times AVF, \quad (3.26)$$

$$\overline{\lambda_{TS}} = \frac{h \times g \times (1-f) \times \lambda_D}{3 \times K} \times U_S \times AVF, \quad (3.27)$$

$$\overline{\lambda_{SysSRs}} = \frac{(1-g) \times (1-f) \times \lambda_D}{L} \times U_C \times AVF, \quad (3.28)$$

respectively. Note that the U_M and U_S variables denote the resource utilization of each triplicated module and their SRs in the TMR components of the $a_{1,2,3}$ sub-systems respectively, while U_C denotes the resource utilization in the simplex components of the a_4 sub-systems.

3.3.7 Recovery technique: Impact on SoC reliability and availability

This section formulates the reliability and availability of four SoC implementations: 1) SoC/FMER, 2) SoC/Scrub, 3) SoC/MER, and 4) SoC/NR (No Recovery), which follow the model of Fig. 3.1(b).

- 1) SoC/FMER: CM upsets in the F_1 CFs (modules of the TMR components) recover with MER, while in the F_2 CFs (SRs) with selective periodic scrubbing;
- 2) SoC/Scrub: CM upsets in both the F_1 and the F_2 CFs recover with device periodic scrubbing;
- 3) SoC/MER: CM upsets in the F_1 CFs recover with MER. CM upsets in the F_2 CFs (i.e., those CFs that are not included in Pblocks) are not recovered; and
- 4) SoC/NR: The SoC does not incorporate any CM ER mechanism.

The difference between the above SoC implementations is the incorporated CM ER mechanisms and therefore the rate at which their components recover. The average rate at which the modules and the associated SRs fail is the same irrespective of the repair mode. In the following paragraphs of this subsection we derive the total reliability and availability of the SoC/FMER.

The rate, μ_m , at which the average module replica recovers with MER in F_1 can be found by substituting Eq. (3.20) into the reciprocal of Eq. (3.5):

$$\begin{aligned}\mu_m &= (\overline{F_M} \times t_F)^{-1} \\ &= \left(\frac{F_1}{3K} \times t_F\right)^{-1} \\ &= \left(\frac{f \times F_D}{3K} \times t_F\right)^{-1}\end{aligned}\tag{3.29}$$

Moreover, the rate, μ_s , at which the F_2 CFs of the SRs recover with scrubbing can be found by replacing F_D with F_2 in the reciprocal of Eq. (3.4):

$$\begin{aligned}\mu_s &= \left(\frac{F_2}{2} \times t_F + w\right)^{-1} \\ &= \left(\frac{(1-f) \times F_D}{2} \times t_F + w\right)^{-1}\end{aligned}\tag{3.30}$$

Therefore, the reliability of the SoC follows from Eq. (3.6) and is given by the product of two sub-products:

$$R(t) = \prod_{i=1}^K R_i^e(t) R_i^d(t) R_i^b(t) \times \prod_{j=1}^L R_j^b(t)\tag{3.31}$$

The first sub-product of Eq. (3.31) denotes the reliability of the $a_{1,2,3}$ sub-systems, while the reliability function of each of their K components depends on their *type* and is given as follows:

- The reliability function of the modules of the i^{th} TMR component is $R_i^e(t)|_{(3.25) \Rightarrow \lambda_m}^{(3.29) \Rightarrow \mu_m}$ since they are recovered with MER, while
- the reliability of the i^{th} triplicated SRs associated with the i^{th} TMR component is given by $R_i^d(t)|_{(3.27) \Rightarrow \lambda_m}^{(3.30) \Rightarrow \mu_s}$ since they are recovered by selective CM scrubbing,
- and that of the i^{th} simplex SRs associated with the i^{th} TMR component is given by $R_i^b(t)|_{(3.26) \Rightarrow \lambda_m}$

The second sub-product of Eq. (3.31) denotes the reliability of the a_4 sub-systems where the reliability of its j^{th} simplex module with its associated pins and interconnection is $R_j^b(t)|_{(3.28) \Rightarrow \lambda_m}$.

Similarly, the availability of the SoC/FMER is calculated as

$$A(t) = \prod_{i=1}^K A_i^e(t) A_i^d(t) A_i^b(t) \times \prod_{j=1}^L A_j^b(t),\tag{3.32}$$

CHAPTER 3. FAST AND ENERGY EFFICIENT CONFIGURATION MEMORY RECOVERY

whereby $A_i^e(t)|_{(3.25) \Rightarrow \lambda_m}^{(3.29) \Rightarrow \mu_m}$, $A_i^d(t)|_{(3.27) \Rightarrow \lambda_m}^{(3.30) \Rightarrow \mu_s}$ and $A_i^b(t)|_{(3.26) \Rightarrow \lambda_m}^{(3.30) \Rightarrow \mu_s}$ are substituted in the first sub-product of the equation, and $A_j^b(t)|_{(3.28) \Rightarrow \lambda_m}^{(3.30) \Rightarrow \mu_s}$ is substituted in the second sub-product of the equation.

The reliability and availability functions of the other implementations, namely SoC/Scrub, SoC/MER and SoC/NR are similarly derived and are summarised in Tables 3.2 and 3.3, respectively.

Table 3.2: Reliability functions of SoC/FMER, SoC/Scrub, SoC/MER and SoC/NR.

SoC	R(t)
SoC/FMER	$\prod_{i=1}^K R_i^e(t) _{(3.25) \Rightarrow \lambda_m}^{(3.29) \Rightarrow \mu_m} \times R_i^d(t) _{(3.27) \Rightarrow \lambda_m}^{(3.30) \Rightarrow \mu_s} \times R_i^b(t) _{(3.26) \Rightarrow \lambda_m} \times \prod_{j=1}^L R_j^b(t) _{(3.28) \Rightarrow \lambda_m}$
SoC/Scrub	$\prod_{i=1}^K R_i^d(t) _{(3.27) \Rightarrow \lambda_m}^{(3.30) \Rightarrow \mu_s} \times R_i^d(t) _{(3.27) \Rightarrow \lambda_m}^{(3.30) \Rightarrow \mu_s} \times R_i^b(t) _{(3.26) \Rightarrow \lambda_m} \times \prod_{j=1}^L R_j^b(t) _{(3.28) \Rightarrow \lambda_m}$
SoC/MER	$\prod_{i=1}^K R_i^e(t) _{(3.25) \Rightarrow \lambda_m}^{(3.29) \Rightarrow \mu_m} \times R_i^c(t) _{(3.27) \Rightarrow \lambda_m} \times R_i^a(t) _{(3.26) \Rightarrow \lambda_m} \times \prod_{j=1}^L R_j^a(t) _{(3.28) \Rightarrow \lambda_m}$
SoC/NR	$\prod_{i=1}^K R_i^c(t) _{(3.25) \Rightarrow \lambda_m} \times R_i^c(t) _{(3.27) \Rightarrow \lambda_m} \times R_i^a(t) _{(3.26) \Rightarrow \lambda_m} \times \prod_{j=1}^L R_j^a(t) _{(3.28) \Rightarrow \lambda_m}$

Table 3.3: Availability functions of SoC/FMER, SoC/Scrub, SoC/MER and SoC/NR.

SoC	A(t)
SoC/FMER	$\prod_{i=1}^K A_i^e(t) _{(3.25) \Rightarrow \lambda_m}^{(3.29) \Rightarrow \mu_m} \times A_i^d(t) _{(3.27) \Rightarrow \lambda_m}^{(3.30) \Rightarrow \mu_s} \times A_i^b(t) _{(3.26) \Rightarrow \lambda_m}^{(3.30) \Rightarrow \mu_s} \times \prod_{j=1}^L A_j^b(t) _{(3.28) \Rightarrow \lambda_m}^{(3.30) \Rightarrow \mu_s}$
SoC/Scrub	$\prod_{i=1}^K A_i^d(t) _{(3.25) \Rightarrow \lambda_m}^{(3.30) \Rightarrow \mu_s} \times A_i^d(t) _{(3.27) \Rightarrow \lambda_m}^{(3.30) \Rightarrow \mu_s} \times A_i^b(t) _{(3.26) \Rightarrow \lambda_m}^{(3.30) \Rightarrow \mu_s} \times \prod_{j=1}^L A_j^b(t) _{(3.28) \Rightarrow \lambda_m}^{(3.30) \Rightarrow \mu_s}$
SoC/MER	$\prod_{i=1}^K A_i^e(t) _{(3.25) \Rightarrow \lambda_m}^{(3.29) \Rightarrow \mu_m} \times A_i^c(t) _{(3.27) \Rightarrow \lambda_m} \times A_i^a(t) _{(3.26) \Rightarrow \lambda_m} \times \prod_{j=1}^L A_j^a(t) _{(3.28) \Rightarrow \lambda_m}$
SoC/NR	$\prod_{i=1}^K A_i^c(t) _{(3.25) \Rightarrow \lambda_m} \times A_i^c(t) _{(3.27) \Rightarrow \lambda_m} \times A_i^a(t) _{(3.26) \Rightarrow \lambda_m} \times \prod_{j=1}^L A_j^a(t) _{(3.28) \Rightarrow \lambda_m}$

3.3.8 Error Recovery: Impact on SoC energy consumption

This section estimates the energy consumption of each SoC implementation, depending upon which CM ER method is used and the mission's length (T).

As mentioned in Sec. 2.3.2, a CF is the atomic unit of reconfiguration in modern SRAM FPGAs. Irrespective of which CM ER technique is used in the SoC, CM upsets always recover by reconfiguring one or more CFs of the FPGA. One can calculate how much energy will be consumed recovering CM upsets with periodic scrubbing, MER or FMER during a mission when the following two parameters are known or can be estimated:

1. The required energy to write or read a CF, denoted E_F .
2. The average number of CF reads and writes during the mission.

The energy consumed reading or writing a CF depends on the architecture of the RC and on the utilised FPGA. For instance, a resource-hungry Microblaze-based RC is not as energy efficient as a highly optimised low-resource RC [41, 145]. Similarly, a Xilinx Virtex-7 FPGA consumes more power than an Artix-7 FPGA. A methodology to estimate E_F is presented in [122].

Energy consumption of SoC/Scrub

The time it takes to reconfigure all F_D CFs of SoC/Scrub is $(F_D t_F)$, where, as previously stated, t_F is the time it takes to reconfigure one CF. Therefore, the CM of the SoC with periodic blind scrubbing will be reconfigured $(\frac{T}{F_D t_F + w})$ times during a mission of length T . Recall that w denotes the waiting time between scrub cycles. The energy required to reconfigure all F_D CFs of the FPGA is $(F_D E_F)$. Thus, the energy consumption of SoC/Scrub is:

$$E_{Scrub} = \frac{T}{F_D t_F + w} F_D E_F \quad (3.33)$$

The first term, $(\frac{T}{F_D t_F + w})$, denotes the number of executed scrub cycles during the mission and the second term, $(F_D E_F)$, denotes the energy consumption of a scrub cycle.

Energy consumption of SoC/MER

Recall that the failure rate of an average sized TMR module in SoC/MER is $\overline{\lambda_m}$, which is given in failures per units of time.

One can estimate how many times, on average, TMR modules will fail during a mission by multiplying $3\overline{\lambda_m}$ with the length T of the mission. As shown in Fig. 3.3(c), a module is reconfigured whenever any of the three modules of the TMR component fails. For example, when $3\overline{\lambda_m} = \frac{90 \text{ failures}}{\text{month}}$ and $T = 100$ months, then the expected failures are $\frac{90 \text{ failures}}{\text{months}} \times 100 \text{ months} = 9,000$ failures, which triggers 9,000 module reconfigurations.

Since the average-sized TMR module is configured with $\overline{F_M}$ frames, the energy consumed by MER at the end of the mission will be:

$$E_{MER} = 3\overline{\lambda_m}T \overline{F_M}E_f \quad (3.34)$$

The first term, $(3\overline{\lambda_m}T)$, represents the expected number of failures in the modules of the TMR components during the mission, while the energy needed to recover a faulty module is given by the second term of the equation, $(\overline{F_M}E_f)$.

Energy consumption of SoC/FMER

The energy consumption of recovering CM upsets in SoC/FMER can be calculated as follows. The RC of the SoC recovers the F_1 CFs (that implement the modules of the TMR components) with MER for:

$$T_{MER} = 3\overline{\lambda_m}T \overline{F_M}t_F \quad (3.35)$$

time of the mission. Thus, during the remaining time of the mission:

$$T_{Scrub} = T - T_{MER} = T(1 - 3\overline{\lambda_m} \overline{F_M}t_F) \quad (3.36)$$

the RC is either scrubbing the F_2 CFs of the SRs or is waiting between scrub cycles. Thus, the energy consumption of SoC/FMER for a mission time T is:

$$\begin{aligned} E_{FMER} &= E_{MER} + E_{Scrub} \\ &= (3\overline{\lambda_m}T \overline{F_M}E_f) + \left(\frac{T(1 - 3\overline{\lambda_m} \overline{F_M}t_F)}{(1 - f)F_Dt_F + w} (1 - f)F_DE_f \right), \end{aligned} \quad (3.37)$$

where $(1 - f)F_D$ denotes the F_2 portion of the device's CFs, which is scrubbed periodically.

3.3.9 Assumptions

The reliability and availability functions in this section were derived given the following assumptions.

Failures in the SoC follow a Poisson distribution, while Eqs. (3.6) and (3.7) hold if all $K+L$ components fail and recover independently in the SoC. The assumption of independent failures between the SoC's components hold when appropriate design techniques are followed during the implementation of the FPGA circuit. CAD tools that isolate the modules of a TMR circuit have been proposed. These include the Xilinx Isolation Design Flow (IDF) [44], the academic IPRDF tool that enhances the IDF so that it supports DPR

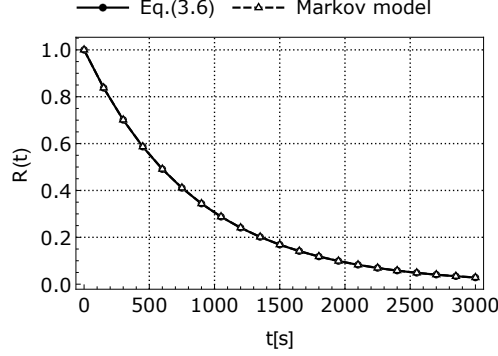


Figure 3.5: Comparison of $R(t)$ for $t \in [0,3000]$ seconds between Eq. (3.6) and the Markov model of [73].

FPGA circuits [89], as well as the Intel/Altera design separation flow [50]. Additionally, the assumption of independent ER between the SoC’s components holds when the system is completely scrubbed, since a complete reconfiguration of the FPGA is equivalent to having a dedicated repair facility for each component in the SoC, which requires on average $t = \text{Eq. (3.4)}$ time to recover the component from a fault.

As mentioned earlier, our reliability analysis models use simple combinatorial functions, i.e., Eq. (3.31), for calculating the dependability of the SoC as a whole, while the reliability of each individual component in $a_1 - a_4$ sub-systems is calculated with the simple Markov-chain models shown in Fig. 3.3. An alternative way to calculate the reliability of the SoCs would have been to exclusively exploit Markov-chain models as has been done in [73]. We checked if our assumptions hold by comparing our combinatorial reliability model (see Eq. (3.31)) with that of the model presented in [73], which uses Markov chains exclusively. The comparison between the two models is shown in Fig. 3.5, where we plot the reliability of a TMR circuit with 50 partitions ($K = 50$) using Eq. (3.6) and the Markov model (52 states) of [73]. We assume that each TMR module fails with rate, $\lambda_m = \lambda_{\text{device}}/3K$, where $\lambda_{\text{device}} = 0.1$ SEU/dev/s. Moreover, the rate at which faulty modules recover with periodic CM scrubbing is $\mu_s = 10\lambda_{\text{device}}$. Fig. 3.5 shows that Eq. (3.6) yields the same results as the Markov model given in [73] despite being considerably more straightforward to evaluate.

In addition, when the SoC incorporates MER an independent reconfiguration process (recovery) for every faulty module holds as long as the repair rate of the module is much

larger than its failure rate, $\mu_m \gg \lambda_m$, i.e., when the probability of executing a recovery process in a faulty TMR component while another TMR component is waiting to be repaired, is negligible [112]. In practise, this is usually the case, since the rate at which modules fail and recover in a typical TMR-based SoC that incorporates scrubbing or MER is on the order of *hours* and *milliseconds* or *microseconds*, respectively [25, 31].

The energy consumption models of MER assume that the RC does not consume energy during fault detection. This assumption holds when the RC consumes negligible energy while waiting for a module reconfiguration request. For example, during the mission the RC may always be in sleep mode in order to save energy and wake up only for short time periods to reconfigure faulty modules. Additionally, the energy consumption models for MER and FMER do not account for the energy spent to recover a TMR component from two or more simultaneous module failures. That is to say, energy consumption is estimated only for when transitioning from state $S1$ to state $S0$ in the Markov-chain models of Fig. 3.3(c) and (d). This assumption holds because the probability of finding a TMR component in state $S2$ is much lower than that of finding it in state $S1$.

3.4 Analytical Results

This section explores and compares the reliability, availability and energy consumption of the four SoC implementations presented in Sec. 3.3, namely the SoC/FMER, SoC/Scrub, SoC/MER and SoC/NR. We recall and summarize the most relevant parameters that are frequently used in this section:

- $K \in \mathbb{N}^+$: number of TMR components (i.e., partitions) in the $a_1 - a_3$ sub-systems of the SoC.
- $f \in [0, 1]$: the fraction of the FPGA's CFs devoted to the $3K$ modules of the TMR components in the $a_1 - a_3$ sub-systems. All CFs are devoted to the SRs of the SoC when $f = 0$, while all CFs are devoted to the $3K$ modules when $f=1$.
- $h \in [0, 1]$: the fraction of CFs devoted to the triplicated SRs of the $a_1 - a_3$ sub-systems of the SoC. All CFs are devoted to simplex SRs of the $a_1 - a_3$ sub-systems when $h = 0$.
- $AVF \in [0, 1]$: denotes the portion of UM and CM cells in the device that lead to errors when corrupted.

- $w \in \mathbb{R}^+ \cup \{0\}$: waiting time between scrub cycles given in seconds (s).
- $\lambda_b \in \mathbb{R}^+$: upset rate of a configuration bit given in SEUs/bit/s.
- $T \in \mathbb{R}^+$: denotes the mission duration given in hours (hrs) or years (yrs).
- $U_M \in \mathbb{R}^+ \cup \{0\}$: resource utilisation of the 3K modules in the $a_1 - a_3$ sub-systems.
- $U_S \in \mathbb{R}^+ \cup \{0\}$: resource utilisation of the SRs for the 3K modules in the $a_1 - a_3$ sub-systems.
- $U_C \in \mathbb{R}^+ \cup \{0\}$: resource utilisation of the simplex components in the a_4 sub-systems.

We assume that each SoC is implemented with a Xilinx Artix-7 200T FPGA, which has the following specifications; $F_D = 18,300$ CFs, $B_F = 3,232$ bits and $t_F = 1.01$ microseconds (considering the maximum configuration speed of the FPGA). We feel that the following base parameters for our models capture a fully triplicated FPGA SoC that operates in a relatively high radiation environment; $\lambda_b = 2.66\text{E-}11$ SEU/bit/s, $w = 0$ s, $f = 0.6$, $g = 1.0$, $h = 1.0$, $AVF = 0.15$, $U_S = 0.1$, $U_M = 0.8$, $U_C = 0.8$, $K = 5$ and $T = 5$ yrs. However, we explore all possible values of our model's parameters, i.e., we vary λ_b, w, K , etc. Similarly, we explore the proposed models from LEO up to GEO radiation levels, i.e., $\lambda_b \in [3.76\text{E-}14, 2.66\text{E-}10]$, which were estimated in Sec. 3.3.2.

The reliability, availability and energy consumption results are captured on the y-axis of 2D and 3D plots in the sub-figures of Fig. 3.6, while the other dimensions are devoted to the model's parameters. All plots use the aforementioned base parameter values unless otherwise stated. We report energy consumption in Joules (J) and assume that the RC requires on average $E_f = 535\text{E-}9$ J to reconfigure a CF [122]. Note that E_f is not measured on an actual Artix-7 FPGA. Instead, E_f is referenced from [122]. Please note that the authors in [122] conducted experiments on a Xilinx Virtex-5 FPGA (rather than the Artix-7 FPGA used in this thesis) in order to find the energy consumption of reconfiguring a CF of the device. Although, E_F , may be different for the Artix-7 FPGA, this will not change the shape of the provided energy results, since E_f is a constant.

3.4.1 Reliability results

Fig. 3.6(a) shows the reliability of all SoCs for a 24-hour mission ($T = 24$ hrs). As expected, SoC/MER is considerably less reliable than both SoC/FMER and SoC/Scrub since SoC/MER leaves CM upsets in its SRs (F_2 CFs in Fig. 3.1(b)) unrecovered. On

CHAPTER 3. FAST AND ENERGY EFFICIENT CONFIGURATION MEMORY RECOVERY

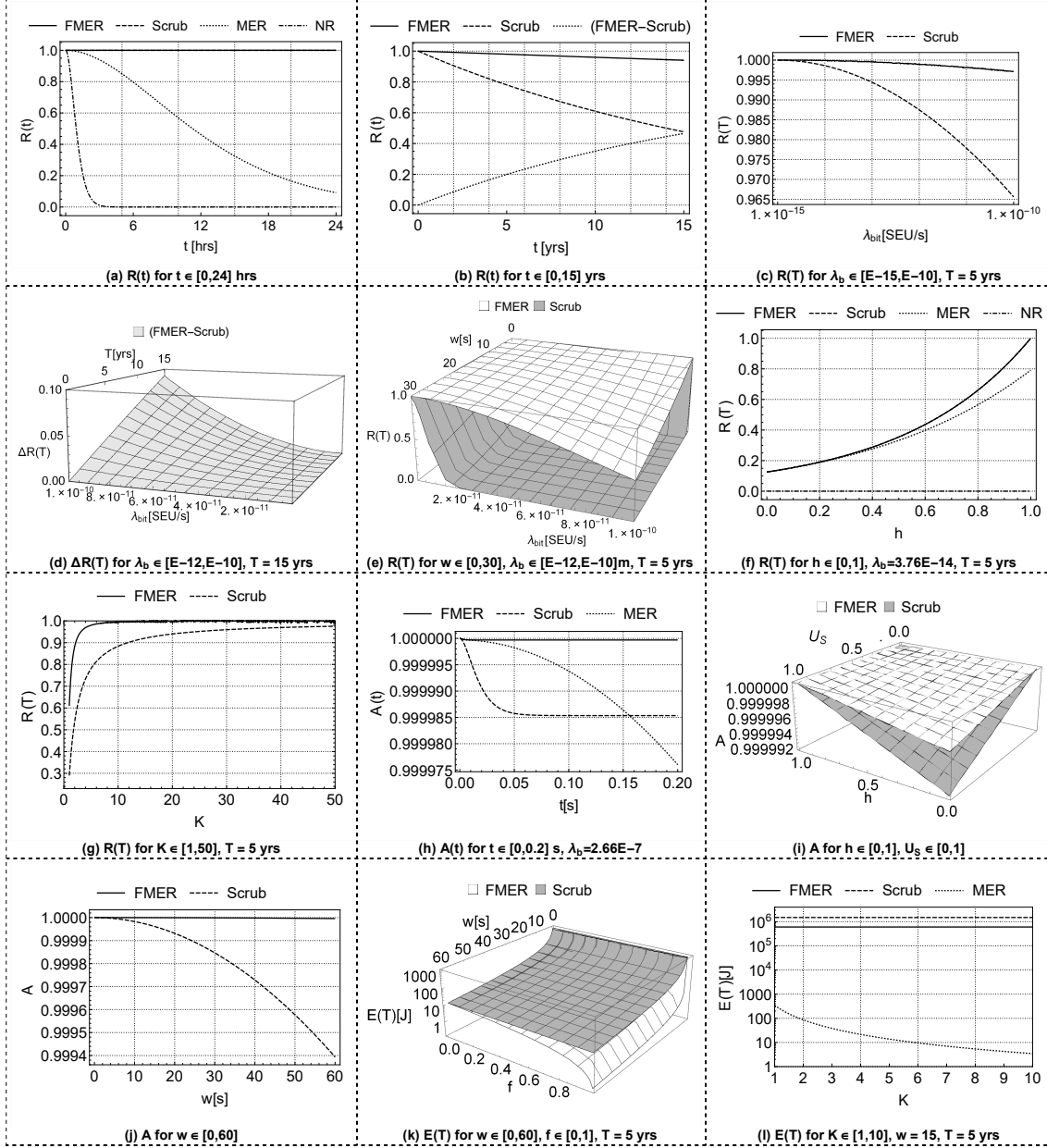


Figure 3.6: Reliability, availability and energy consumption results.

the other hand, the SoC/NR has the lowest reliability of all implementations since no ER technique is incorporated into the SoC whatsoever. Therefore, SoC/FMER and SoC/Scrub have the lowest probability of experiencing a failure during their first 24 hrs of operation, and their difference in reliability at $T = 24$ hrs is negligible, i.e., $\Delta R(24 \text{ hrs}) \approx 43E-6$. However, for a longer mission the reliability of SoC/FMER is significantly higher than that of SoC/Scrub. For example, Fig. 3.6(b) shows the reliability of SoC/FMER and

SoC/Scrub, and their difference in reliability (FMER-Scrub) for a 15-year mission, at the end of which SoC/FMER has $R(15 \text{ yrs}) \approx 0.94$, much higher than SoC/Scrub, which has $R(15 \text{ yrs}) \approx 0.47$, i.e., $\Delta R(15 \text{ yrs}) = 0.94 - 0.47 \approx 0.47$.

Nevertheless, when the CM upset rate is low, both SoC/FMER and SoC/Scrub achieve high reliability. This can be seen in Fig. 3.6(c), where we plot the reliability of SoC/Scrub and SoC/FMER at $T = 5 \text{ yrs}$ for $\lambda_b \in [\text{E-15}, \text{E-10}]$. At the lower end of this range, $\lambda_b \in [\text{E-15}, \text{E-13}]$ (e.g., the expected CM upset rates at the ISS orbit), the SoC/Scrub achieves high reliability since periodic CM scrubbing achieves a much higher recovery rate than the failure rate when $\lambda_b \in [\text{E-15}, \text{E-13}]$. In other words, μ of periodic CM scrubbing is adequate to reduce the probability of having more than one faulty module per TMR component to a negligible level when $\lambda_b \in [\text{E-15}, \text{E-13}]$.

Additionally, the 3D plot of Fig. 3.6(d) shows the difference in reliability $\Delta R(T)$ between SoC/FMER and SoC/Scrub for $T \in [1, 15] \text{ yrs}$ and $\lambda_b \in [\text{E-12}, \text{E-10}]$. The figure shows that $\Delta R(T) \rightarrow 0$ as $\lambda_b, T \rightarrow 0$. The results of Fig. 3.6(d) indicate that SoC/FMER achieves substantially better reliability than SoC/Scrub, particularly in higher radiation environments or on longer missions. Moreover, FMER should be considered in missions with a tight energy budget, i.e., the waiting time between scrub cycles, w , in SoC/FMER can be increased to the level of achieving the same reliability as SoC/Scrub does, but by consuming less energy. For example, Fig. 3.6(e) shows the reliability of a mission with $\lambda_b \in [\text{E-12}, \text{E-10}]$ and $w \in [0, 30] \text{ s}$. The figure reveals that the reliability of SoC/Scrub is affected more than the reliability of SoC/FMER as w increases. The reason behind this observation, is that SoC/FMER scrubs only the SRs, i.e., only the F_2 CFs shown in Fig. 3.1(b), while SoC/Scrub scrubs all CFs of the FPGA device, i.e., both the F_1 and F_2 CFs. Therefore, SoC/FMER can have a longer w than SoC/Scrub and still recover CM upsets with the same rate as SoC/Scrub does. In other words, SoC/FMER achieves the reliability of SoC/Scrub but with less frequent scrub cycles, which results in saved energy. For instance, in Fig. 3.6(e) we observed that both SoC/FMER and SoC/Scrub have $R(5 \text{ yrs}) \approx 0.992$ when $\lambda_b = \text{E-11}$, however, with $w = 30 \text{ s}$ in the case of SoC/FMER and $w = 0.198 \text{ s}$ in the case of SoC/Scrub. Using Eqs. (3.33) and (3.37) we calculated that SoC/Scrub consumes 347 times more energy than SoC/FMER ($E_{\text{FMER}} \approx 20,297 \text{ J}$, $E_{\text{Scrub}} \approx 7.03\text{E}6 \text{ J}$) during the mission, in order to achieve the same reliability as SoC/FMER does.

However, the above results only hold when the SoC is fully triplicated, i.e., when $g, h = 1$. As shown in Fig. 3.6(f), the reliability of all SoCs is dramatically reduced, even in low radiation orbits ($\lambda_b = 3.76\text{E-14}$), when the proportion of simplex components increases

in the $a_{1,2,3}$ sub-systems, i.e., when $h \rightarrow 0$. Note that similar results are observed when simplex a_4 sub-systems are included in the SoC, i.e., when $g \rightarrow 0$. In more detail, the reliability of the SoCs is reduced because any included simplex resources become SPFs. In other words, the reliability of a TMR SoC is mostly determined by the reliability of any simplex component in the SoC. For example, let us assume a fully triplicated SoC that has $R_{\text{SoC}}(1 \text{ yr}) = 0.9999$. If an additional simplex component with $R_{\text{Simplex}}(1 \text{ yr}) = 0.5$ is included in the design, then the overall reliability of the SoC will be $R_{\text{SoC}}(1 \text{ yr}) = 0.9999 \times 0.5 \approx 0.5 \approx R_{\text{Simplex}}(1 \text{ yr})$.

Nevertheless, when the SoC is fully triplicated, i.e., when $g, h = 1$, then the reliability of the system can be further increased by partitioning the design at a finer granularity ($K \rightarrow \infty$), e.g., by triplicating every stage of a processor rather than the processor as a whole. Firstly, as K increases, the number of TMR modules in the system increase and the likelihood of multiple errors affecting the one TMR component is reduced. Moreover, the average number of CFs per TMR module decreases, and thus the overall MTTR in SoC/FMER (not in SoC/Scrub) decreases since less CFs have to be reconfigured per faulty TMR module with MER according to Eq. (3.5). The improvement in reliability as K increases is captured in Fig. 3.6(g), which depicts the reliability of SoC/FMER and SoC/Scrub for $K \in [1, 50]$. As can be observed, the reliability of SoC/FMER improves faster than for SoC/Scrub as K increases.

3.4.2 Availability results

Achieving high availability in an FPGA-based SoC is easier than achieving high reliability. The ratio between the CM upset rate and the CM ER rate in modern SRAM FPGAs makes them attractive SoC candidates for high-availability space missions [113].

For example, Fig. 3.6(h) depicts the transition to steady state availability of SoC/FMER, SoC/Scrub and SoC/MER for an extremely high CM upset rate, $\lambda_b = 1,000 \times 2.66\text{E-}10$ (1,000x the Peak-5-Min GEO λ_b), which is more likely to be encountered in high-energy physics experiments [137] than in space. As can be observed, even with this extremely high CM upset rate the steady state availability of SoC/Scrub is 4 nines, while the steady state availability for SoC/FMER is even higher. In contrast, the availability of SoC/MER does not reach a steady state since the SRs never recover when MER alone is applied to the SoC.

Nevertheless, SoC/FMER and SoC/Scrub achieve high availability even when $h \rightarrow 0$ and $U_S = 1$, i.e., when the SRs are not fully triplicated and are highly utilised. This is shown

in Fig. 3.6(i) where in the worst case ($h = 0, U_S = 1$), the steady state availability of SoC/FMER and SoC/Scrub is 0.999997 and 0.999991 respectively. However, FMER can be used in an FPGA SoC to achieve high availability with less energy consumption than when periodic CM scrubbing alone is used. Fig. 3.6(j) shows the steady state availability of SoC/FMER and SoC/Scrub as w is varied. SoC/FMER achieves 5 nines availability when $w = 60$ s, while SoC/Scrub achieves approximately 3 nines for the same w . However, $E_{FMER} \approx 10,163$ J, while $E_{Scrub} \approx 25,369$ J, i.e., SoC/FMER achieves higher availability than SoC/Scrub with 2.5 times less energy. Note that similar results are obtained when the SoC incorporates simplex a_4 sub-systems, i.e., when $g \rightarrow 0$.

3.4.3 Energy consumption results

We found that SoC/FMER and SoC/Scrub energy consumption decreases geometrically as w increases. Fig. 3.6(k) illustrates the energy consumption for both systems in logarithmic scale for $w \in [0,60]$ and $f \in [0,1]$. We observe that E_{FMER} is always less than E_{Scrub} for equal values of w and for $f > 0$. When $f = 0$ in SoC/FMER then the system is completely scrubbed, thus $E_{FMER} = E_{Scrub}$. Additionally, as K increases, the energy consumption of SoC/MER decreases. This is because the system is partitioned at finer granularity, which means faulty TMR modules can be localised and corrected more precisely as K increases, thus the RC reconfigures less CFs per fault. This is shown in Fig. 3.6(l), in which we plot the energy consumption of SoC/FMER, SoC/Scrub and SoC/MER against a logarithmic energy scale for $K \in [1,10]$ and $w = 1$ s. We observe that E_{MER} decreases as K in SoC/MER increases. In the case of SoC/FMER the energy consumption that is expended in repairing TMR modules is negligible compared to the energy consumed periodically scrubbing the SRs. Therefore, K does not significantly affect the overall energy consumption of SoC/FMER. Furthermore, the energy consumption of SoC/Scrub is not affected at all as K varies, since μ depends on F_D and not on $\overline{F_M}$. Last, we observe that E_{MER} is less than E_{FMER} and E_{Scrub} since SoC/MER does not involve periodic CM scrubbing as it only reconfigures the CFs of faulty TMR modules when errors are detected.

3.5 Implementation of FMER

FMER can be implemented with any internal or external RC that is able to reliably configure the CM of the FPGA by fetching CFs located in an external radiation-hardened memory. In principle, FMER can be realised by: (i) constraining the implementation of

modules to specific regions of the FPGA, i.e., Pblocks, and (ii) generating lists of frame addresses (FADs) for each Pblock and for the SRs of the design so that the CFs of the SR FAD list are periodically scrubbed and the CFs of any corrupted Pblock are reconfigured on-demand. In order to generate the lists of FADs for the Pblocks and SRs, we execute the following steps: (i) generate a FAD list for the whole device, (ii) generate a FAD list for each Pblock, and (iii) create a FAD list for the SRs by subtracting the Pblock FAD lists from the device's FAD list.

3.5.1 Generating frame address lists

To the best of our knowledge, two methods have been described in the literature to create a FAD list for a Xilinx device. The first method extracts the FAD from the bitstream with custom bitstream manipulation tools [105, 107], that can be implemented using academic CAD frameworks such as RapidSmith [64]. However, to use this method, the bitstream has to have a format in which the FAD for each CF in the bitstream is associated with its configuration data (CFDATA). This bitstream format can be obtained from the Xilinx Vivado design suite by enabling the CRC-per-frame flag or the debug flag during bitstream generation. The second method to generate the FAD list for the device is to readback the CM of the FPGA and capture the Frame Address Register (FAR) as it auto-increments [119]. Reading back the CM of the FPGA is the best method for creating a FAD list for a Xilinx FPGA device. As pointed out in [119], Xilinx FPGAs contain several CFs that are not included in the bitstream; bitstream manipulation tools do not therefore cover these CFs. We currently use tools that have developed within the RapidSmith framework to extract the FAD list for the device from its bitstream and the FAD list for each Pblock from its partial bitstream. However, we intend to develop tools that create FAD lists by reading back the CM of the device.

In addition, we have developed tools that create the FAD lists of a TMR-based FPGA circuit that does not follow a DPR flow and therefore only one bitstream for the circuit is generated. In fact, the DPR flow is useful for developing FPGA circuits that share the resources of a Pblock among different hardware modules in order to save area and power. However, the CAD tools interconnect the static area with the dynamic area (i.e., Pblocks) of the DPR-based FPGA circuit using proxy pins, that unfortunately introduce area and timing overheads. In a TMR circuit with FMER or MER, only a single hardware module is assigned per Pblock and therefore proxy pins are not required. Therefore, we avoid the DPR flow and generate FAD lists for the device and the Pblocks using the following method.

The TMR modules of the SoC are assigned to Pblocks that are always aligned to the configuration rows (see Fig. 3.2) of the FPGA, i.e., each Pblock has its height set to one or more rows. Algorithm 1 is then used to generate the FAD list for the device. The

Algorithm 1 Generation of the FAD list of the FPGA device

Initialisation

BlockType $b \leftarrow$ "000"

FAD list $L \leftarrow$ new list

```

for topBottom  $t \leftarrow 0$  to 1 do
  for row  $r \leftarrow 0$  to getMaxRows( $t$ )-1 do
    forall column  $c$  in DeviceColumns do
      for minor  $m \leftarrow 0$  to getMaxMinors( $c$ )-1 do
        FAD  $\leftarrow$  createFAD(  $b, t, r, c, m$  )
        Insert FAD into  $L$ 
      end
    end
  end
end
return  $L$ 

```

algorithm consists of 4 nested loops that create values for the top/bottom, row, column and minor address of each FAD in the list. The three outer loops iterate through the coordinates of all resource columns in the device, i.e., the top/bottom, row and column address, while the innermost loop iterates over the minor addresses of each column. The body of the innermost loop creates a FAD with block type (b), top/bottom (t), row (r), column (c) and minor (m) addresses and adds the FAD into the FAD list (L). Note that the block type is always set to "000" in order to exclude CFs that configure BRAM contents. In a similar way, we have implemented an algorithm to iterate through the resource columns of each Pblock in order to generate FAD lists for the Pblocks.

3.5.2 Obtaining data for each configuration frame address of the FPGA

The CFDATA for each FAD is obtained from Vivado's *.ebc* file. This file is generated by Vivado when the essential bits flag is enabled during bitstream generation. The *.ebc* file contains the CFDATA, in ASCII representation, as obtained from reading back the CM of the device [119, 142]. The *.ebc* file is converted into binary form and is stored in the external memory of the FPGA SoC, together with the FAD lists and the bitstream of the design. The device is initially configured with a complete bitstream, and while it is in

operation, CM upsets recover with the CFDATA from the *.ebc* file. The same *.ebc* file can also be used to recover CFs using blind scrubbing or MER. For example, the Xilinx Soft Error Mitigation (SEM) controller uses the *.ebc* file to implement readback CM scrubbing with frame replacement [145].

3.6 Practicality and Applicability of FMER

In order to demonstrate the practicality and applicability of FMER in real-world, fault-tolerant SoCs, we implemented TMR versions of various HLS applications on a Nexys Video board, which hosts a Xilinx Artix-7 XC7A200T FPGA. We tested each design with either blind scrubbing, MER or FMER, which were implemented using the internal TMR reconfiguration controller presented in [41]. The reliability, availability and power consumption of all SoCs were compared for a 2-year LEO mission.

The following subsections provide a description of the SoC implementations and outline how we derived their dependability and their CM ER consumption during the 2-year LEO mission. Experimental results are presented at the end of the section.

3.6.1 Benchmarks and implementation of the SoCs

We implemented a range of different SoCs with applications from the CHStone, DWARV and Bambu HLS benchmark suites [46, 80] that fit onto the XC7A200T device when triplicated. These benchmarks come from various application domains, such as communications, encryption, compression, arithmetic, compute and media.

We used our TLegUp HLS tool, which will be presented in detail in Chapter 5, to generate TMR register-transfer level Verilog code for each HLS application. Each TMR design consisted of 3 modules (i.e., the 3 replicas of the TMR design), while each module incorporated a 2-bit health status port to report which modules of the TMR circuit, if any, were corrupted by soft errors. Additionally, each HLS application included three 1-bit input ports: clock, reset and start, as well as three output ports: finish (1-bit) and result (32-bit). Test vectors for each application were provided by the benchmark suite and were stored on chip to perform functional verification during their operation. The architecture of TLegUp generated TMR designs is detailed in Chapter 5.

We used the RC of [41] to implement each CM ER technique and to control the ports of

the application in order to verify its operation. The three modules of the TMR RC were placed into one Pblock that was located in one of the ten available clock regions of the Artix-7 200T FPGA. In more detail, the three modules of the RC were placed in a single Pblock because the authors in [41] found that the RC achieves higher performance with this layout. The resources of the remaining nine clock regions of the FPGA were used to create three additional Pblocks, with each one hosting a module of the TMR design.

All designs were synthesised, floorplanned [100] and implemented with the 2017.2 Vivado design suite.

3.6.2 Utilised configuration frames, essential bits and resources

With the essential bits flag enabled, Vivado generates a “mask” for the *.ebc* file described in Sec. 3.5.2 that is called an *.ebd* file or essential bits (Ebits) file. This mask indicates which bits in the CM may produce a functional error when upset [105, 107, 145]. We analysed the *.ebd* file with custom tools [107] that were implemented using the RapidSmith CAD framework [64] in order to count the Ebits in each Pblock and the SRs of each SoC. The number of Ebits and CFs of the Pblocks and SRs were used to estimate the dependability and energy consumption of each SoC, depending on the utilised error recovery mechanism. Note that the number of CFs of each Pblock and the SRs is equal to their FAD list size.

The CFs, Ebits and resource utilization of each SoC are listed in Table 3.4. The sub-columns f and $1 - f$ under the F_D column denote the fractions fF_D and $(1 - f)F_D$ of the device’s CM devoted to the Pblocks (including the RC Pblock) and the SRs respectively. Columns 2-4 of the table report the number of CFs and Ebits of the 1st, 2nd, and 3rd HLS application Pblocks (PBs), respectively, while column 4 shows the same information for the SRs of each SoC. The Ebits of the SRs are further divided into bits located in the configuration (Cfg), IO Block (IOB) and clock (Clk) resource frames of the device. Note that the CM bits for the ICAP primitive are included under the Cfg column. The bits allocated to the remaining resource frames of the SRs are shown in the “Rem” field of the table. These configuration bits realise the routing of the triplicated nets between the Pblocks of the SoC. Column 6 provides the post-routing resource utilization of each HLS design in terms of number of slices, BRAM16 (BM) and DSP48 (DSP), in order to show the relationship between utilised CFs and resources, respectively. The bottom row of the table provides the geometric mean (GMean) of the results. Information for the RC is excluded from the table, since the same RC is instantiated in all SoCs; the RC is realised with 1062 CFs and 1476.51K Ebits, as averaged over all SoC implementations.

Table 3.4: Number of CFs, essential bits (Ebit) and resource utilization of the Pblocks and SRs for each SoC

SoC	F _D		1 st HLS PB		2 nd HLS PB		3 rd HLS PB		Support Resources (SRs)						Resource Util.		
	f	1-f	CFs	Ebit(K)	CFs	Ebit(K)	CFs	Ebit(K)	CFs	Cfg	IOB	Ebit(K)			Slice (K)	BM	DSP
aes	0.23	0.77	1,052	942.80	1,052	1094.72	1,052	949.49	14,122	1.52	0.35	38.77	15.19	55.83	3.88	12	0
aesdec	0.24	0.76	1,124	913.08	1,124	1079.09	1,124	910.81	13,906	1.55	0.35	20.00	19.93	41.83	3.49	12	0
bell	0.14	0.86	344	332.81	600	332.78	600	390.31	15,734	1.70	0.34	4.12	32.07	38.23	0.94	6	0
dfadd	0.23	0.77	1,032	728.43	1,032	730.14	1,032	751.34	14,182	1.69	0.34	17.92	9.39	29.35	5.94	18	0
dfmul	0.19	0.81	924	550.72	924	522.96	580	507.91	14,850	1.44	0.34	16.51	7.37	25.65	4.14	18	48
gsm	0.31	0.69	1,920	1446.63	1,200	1438.75	1,560	1377.18	12,598	1.50	0.34	33.18	223.81	258.83	5.93	21	177
mips	0.12	0.88	380	440.28	408	431.20	380	478.11	16,110	3.58	0.36	12.35	2.13	18.42	1.89	12	12
mmult	0.09	0.91	236	157.88	236	181.71	236	143.88	16,570	1.44	0.34	10.59	11.33	23.71	0.54	12	15
motion	0.43	0.57	2,280	2900.22	2,316	2868.96	2,280	3252.83	10,402	1.70	1.59	55.13	58.02	116.43	7.64	23	0
satl	0.16	0.84	416	354.50	708	384.65	708	364.26	15,446	1.62	0.34	4.10	12.52	18.58	1.73	12	0
sha	0.30	0.70	1,484	1803.28	1,484	1802.68	1,484	2228.30	12,826	1.63	2.06	10.55	56.67	70.92	6.79	30	0
GMean	0.20	0.77	807	705.13	860	734.69	838	732.07	14,138	1.78	0.64	18.44	19.44	43.89	2.99	15	35

Note that the number of Ebits for the RC changes slightly between each case study due to the heuristic nature of the algorithms used in the CAD tools.

The SoC with the largest fraction of CM devoted to Pblocks is `motion`, with $f = 0.43$, while the smallest is `mmult`, with $f = 0.09$. `Motion` utilises 14 times more slices than `mmult`, which is also reflected in the considerably higher number of Ebits for this design. All SoCs have on average (geometric) $f = 0.20$.

It is worth mentioning that the analytical results of Sec. 3.4 show that FMER achieves higher dependability with less energy consumption as $f \rightarrow 1$. Therefore, the benefits of FMER in our experimental results would have been more pronounced if the implemented SoCs had greater f .

3.6.3 Dependability and energy consumption

In the following, we outline the derivation of the dependability and energy consumption for each SoC depending on which CM ER mechanism it utilises.

The SoCs include three simplex sub-systems, namely the ICAP, the clock (e.g., clock buffers, clock manager etc.) and IO (e.g., clock input pin) sub-systems, that are implemented with the *Cfg*, *IOB* and *Clk* Ebits of Table 3.4, respectively. The SoCs also include three TMR sub-systems: (i) the HLS application, that is implemented with the 1st, 2nd and 3rd HLS PB Ebits of Table 3.4, (ii) the RC, that is implemented with 1476.51K Ebits, and (iii) the interconnection nets between the modules of the application and the RC, that are implemented with the *Rem.* Ebits of Table 3.4.

Reliability of the SoCs

The reliability of the simplex SR sub-systems in each SoC is:

$$R_{\text{simplex}}(t) = R_{\text{cfg}}(t) \times R_{\text{IOB}}(t) \times R_{\text{clk}}(t), \quad (3.38)$$

where $R_{\text{cfg}}(t)$, $R_{\text{IOB}}(t)$, and $R_{\text{clk}}(t)$ are the reliabilities of each of the simplex sub-systems, and the reliability of each of these sub-systems is given by Eq. (3.9). In more detail, the Ebits of each sub-system (i.e., *Cfg*, *IOB* and *Clk* in Table 3.4) multiplied by the failure rate of a configuration bit, λ_b , gives its failure rate, λ_m , which is then used in Eq. (3.9).

The reliability of the TMR sub-systems in each SoC is:

$$R_{\text{TMR}}(t) = R_{\text{app}}(t) \times R_{\text{RC}}(t) \times R_{\text{inet}}(t), \quad (3.39)$$

where $R_{app}(t)$, $R_{RC}(t)$ and $R_{inet}(t)$ is the reliability of the TMR HLS application, the TMR RC, and the triplicated interconnection nets between them, respectively, and the reliability of each sub-system is given by Eq. (3.12), except for SoC/MER, where $R_{inet}(t)$ is given by Eq. (3.11).

The average Ebits of the 1st, 2nd and 3rd Pblock of Table 3.4 are used to calculate the average failure rate, λ_m , of each of the three modules in the HLS application. On the other hand, all three modules of the TMR RC are placed in one Pblock, which is implemented with 1476.51K Ebits. Therefore, the average failure rate per RC module is $\frac{1476.51K}{3}\lambda_b$ upsets/s. Similarly, the average failure rate of 1/3 of the triplicated interconnection nets, that are modelled as belonging to one module in our dependability analysis, is $\frac{Rem.}{3}\lambda_b$ upsets/s.

The average recovery rate, μ_s , for all TMR sub-systems in SoC/Scrub is given by the reciprocal of Eq. (3.4). In SoC/MER and SoC/FMER, the average number of CFs of the three Pblocks of the HLS application is used in the reciprocal of Eq. (3.5) to calculate the recovery rate, μ_m , of each module of the HLS application.

In contrast, all three modules of the RC are reconfigured whenever any module fails, since all RC modules have been placed into a single Pblock. The recovery rate for the RC is calculated as the reciprocal of Eq. (3.5), with $F_M = 1062$ CFs, i.e., the total number of CFs for the RC Pblock.

According to Eq. (3.31), the total reliability of SoC/Scrub, SoC/MER and SoC/FMER is:

$$R(t) = R_{simplex}(t) \times R_{TMR}(t), \quad (3.40)$$

where $R_{simplex}(t)$ and $R_{TMR}(t)$ are the corresponding reliabilities of the simplex and TMR sub-systems of each SoC.

Availability of the SoCs

Similar to the Xilinx SEM controller [145], the RC outputs a heartbeat that stops when a fatal failure occurs in the SoC. Fatal failures may occur in the RC when one or more simplex SR sub-systems fail or when two or more modules of the RC fail. For example, the heartbeat of the RC may stop when a clock manager, the ICAP or the clock input pin of the SoC fails. Fatal failures are recovered by a complete reconfiguration of the FPGA. On average it takes half the period of the heartbeat (T_{HB}), plus the latency of a complete reconfiguration of the FPGA ($T_{reconfig.}$) to detect a heartbeat stop and to recover the SoC:

$$T_{\text{fatal-recovery}} = \frac{T_{\text{HB}}}{2} + T_{\text{reconfig}}. \quad (3.41)$$

Therefore, the availability of the simplex sub-systems of all SoCs is:

$$A_{\text{simplex}}(t) = A_{\text{cfg}}(t) \times A_{\text{IOB}}(t) \times A_{\text{clk}}(t), \quad (3.42)$$

where $A_{\text{cfg}}(t)$, $A_{\text{IOB}}(t)$, and $A_{\text{clk}}(t)$ are given by Eq. (3.10) and $\mu_m = (T_{\text{fatal-recovery}})^{-1}$.

Similarly, the availability of the triplicated sub-systems in each SoCs is:

$$A_{\text{TMR}}(t) = A_{\text{app}}(t) \times A_{\text{RC}}(t) \times A_{\text{inet}}(t), \quad (3.43)$$

where $A_{\text{app}}(t)$ and $A_{\text{inet}}(t)$ are given by Eq. (3.13) for SoC/Scrub, and Eq. (3.14) for SoC/FMER. In SoC/MER, $A_{\text{app}}(t)$ and $A_{\text{inet}}(t)$ are given by Eqs. (3.14) and (3.11), respectively. Lastly, $A_{\text{RC}}(t)$ in SoC/Scrub is calculated by adding the probability distributions p_{S0} and p_{S1} in the Markov model of Fig. 3.3(d), with $\mu_1 = \mu_s$, i.e., the scrub rate of the device, and $\mu_2 = (T_{\text{fatal-recovery}})^{-1}$, i.e., the rate at which the device recovers from a fatal failure. Similarly, $A_{\text{RC}}(t)$ in SoC/MER and SoC/FMER is calculated by adding p_{S0} and p_{S1} of the same Markov model, where $\mu_1 = (1062 \times t_F)^{-1}$, i.e., the recovery rate of the RC Pblock, and $\mu_2 = (T_{\text{fatal-recovery}})^{-1}$.

The total availability of each SoC is the product of their simplex and TMR sub-system availabilities:

$$A(t) = A_{\text{simplex}}(t) \times A_{\text{TMR}}(t) \quad (3.44)$$

Energy consumption of recovering CM upsets in the SoCs

The CM ER energy consumption of SoC/Scrub is given by Eq. (3.33), while for SoC/MER it is:

$$E = E_{\text{HLS-app}} + E_{\text{RC}}, \quad (3.45)$$

where both $E_{\text{HLS-app}}$ and E_{RC} are given by Eq. (3.34). However, $\overline{F_M}$ in the E_{RC} part is equal to 1062 CFs since all three modules of RC are placed in one Pblock and all three modules are reconfigured when one or more modules of the RC fail, i.e., $\overline{F_M}$ is equal to the number of CFs contained in the RC's Pblock.

The energy consumption of SoC/FMER is given by Eq. (3.37), except for T_{MER} , which is derived as follows. The RC and the modules of the HLS application in the SoCs/FMER recover with MER for $T_{\text{MER}} = T_{\text{HLSapp}} + T_{\text{RC}}$ time of the mission. Both T_{HLSapp} and T_{RC}

are equal to $3\overline{\lambda_m}T\overline{F_M}t_F$, however in T_{RC} , $3\overline{\lambda_m}$ and $\overline{F_M}$ are substituted with 1,476.51K and 1062, respectively.

Calculation of “ w ” in SoC/Scrub and SoC/FMER

Xilinx suggests scrubbing at a rate at least 10 times faster than the expected CM upset rate [138]. Therefore, the scrub rate, μ_s , of SoC/Scrub should be:

$$\mu_s = k\lambda_D, k > 10, \quad (3.46)$$

where λ_D is as in Eq. (3.24) and k determines how many times faster to scrub than the expected CM upset rate. By setting μ_s equal to the reciprocal of Eq. (3.4) in Eq. (3.46) we get:

$$\left(\frac{F_D}{2} \times t_F + w\right)^{-1} = k\lambda_D \quad (3.47)$$

Solving for w in Eq. (3.47) sets:

$$w = \frac{1}{k\lambda_D} - \frac{F_D t_F}{2} \quad (3.48)$$

Similarly, values for w in SoC/FMER are also calculated with Eqs. (3.24) and (3.48), but by substituting F_D with the number of CFs for the SRs.

3.6.4 Experimental results

The reliability, availability and energy consumption of all SoCs were calculated with the following parameters: (i) $T = 2$ years, i.e., the mission’s length, (ii) $\lambda_b = 1.10\text{E-}13$, i.e., the configuration bit upset rate of the worst day value for LEO from Table 3.1, (iii) $t_F = 16.56$ microseconds, i.e., the time required for the RC to read a CF from the external SPI flash of the Nexys Video board and to write it to the CM of the FPGA, (iv) $T_{HB} = 100$ milliseconds, i.e., the heartbeat period of the RC, (v) $T_{\text{reconfig.}} = 400$ milliseconds, i.e., the time the device takes to completely reconfigure the FPGA, and (vi) $k = 100$, i.e., the scrub rate was set to 100 times the expected CM upset rate of the mission.

Table 3.5 provides the reliability, $R(T)$, availability, $A(T)$, in number of nines [59] and energy consumption, E , in Joules for all SoCs.

A noticeable observation is that SoC/Scrub, SoC/MER and SoC/FMER have equal $R(T)$ in Table 3.5. Our experimental and analytical results show that $R_{\text{simplex}}(T)$ in Eq. (3.40) determines the total $R(T)$ of the SoC. For example, the $R_{\text{simplex}}(T)$ and $R_{\text{TMR}}(T)$ of the aes SoC/Scrub in Table 3.5 is 0.75 and 0.99 respectively. By substituting these values

3.6. PRACTICALITY AND APPLICABILITY OF FMER

Table 3.5: $R(t)$, $A(t)$ and energy consumption for a 2-year LEO mission

SoC	$R(T)$	$A(T)$ [# of 9s]			E [Joules]		
	Any	Scrub	MER	FMER	Scrub	MER	FMER
aes	0.76	8.67	2.47	8.70	396	0.017	236
aesdec	0.86	8.92	2.24	8.96	396	0.018	229
bell	0.96	9.49	1.85	9.52	396	0.008	293
dfadd	0.87	8.98	2.88	9.01	396	0.014	238
dfmul	0.88	9.02	3.08	9.04	396	0.010	261
gsm	0.79	8.71	0.45	8.76	396	0.030	188
mips	0.91	9.13	3.59	9.15	396	0.008	307
mmult	0.92	9.21	2.72	9.21	396	0.006	325
motion	0.67	8.41	1.37	8.54	396	0.081	128
satd	0.96	9.49	2.63	9.52	396	0.008	282
sha	0.91	8.94	1.39	9.15	396	0.037	195
Gmean	0.86	8.99	2.00	9.05	396	0.015	237

into Eq. (3.40) we obtain the total $R(T)$ of the SoC, i.e., $R(T) = 0.75 \times 0.97 = 0.74 \approx R_{\text{simplex}}(T)$. The `satd` and `motion` applications use the lowest and highest number of Ebbs respectively for the implementation of their simplex SRs (*Total-Rem.* in Table 3.4); this is reflected in their reliability, where $R(T) = 0.96$ for `satd` and $R(T) = 0.67$ for `motion`. All SoCs achieve on average $R(T) = 0.86$.

Further, all SoC/Scrub and SoC/FMER designs achieve more than 8 nines $A(T)$. SoC/FMER has on average a slightly higher $A(T)$ than SoC/Scrub, because the TMR sub-systems of the SoC recover faster with FMER. In fact, the $A(T)$ of SoC/FMER would have been much greater than SoC/Scrub if the designs were fully triplicated; the availability of the SoCs is mostly determined by the availability of their simplex sub-systems, not by their triplicated sub-systems, whereby their reliability depends significantly on the CM ER mechanism used. On average, SoC/MER has approximately 7 nines less $A(T)$ than SoC/Scrub and SoC/FMER, since the triplicated interconnection nets between the Pblocks are not recovered from SEUs. However, in the long term the availability of SoC/MER becomes zero when $T \rightarrow \infty$, except if a heartbeat is included in the application to trigger a reconfiguration of the device when the interconnection nets between the Pblocks of the SoC fail.

The energy consumption of SoC/Scrub depends only on the FPGA used, i.e., it depends on F_D and λ_D , which determines μ_s . Therefore, SoC/Scrub requires 396 Joules to recover CM upsets for the 2-year LEO mission. In contrast, the energy used to recover with SoC/MER and SoC/FMER depends on the device and on the FPGA circuit, since the Ebbs of each Pblock determines how often a fault in the Pblock will trigger its reconfiguration. SoC/MER and SoC/FMER will have consumed, respectively, 19,812 and 1.68 times less

energy than SoC/Scrub during the 2-year mission. Of all SoCs, `mmult` and `motion` SoCs have the lowest and highest f . This is reflected in the energy consumed recovering from CM upsets, where `mmult` and `motion` SoC/MER have the lowest and highest consumption, respectively. On the other hand, `motion` SoC/FMER consumes the lowest amount of energy to recover from CM upsets of those applications studied, since f is large and not many CFs of the SRs have to be recovered with periodic scrubbing.

3.7 Related Work

Demand for fast and energy-efficient CM ER techniques has led to a number of interesting proposals, some of which are described in the following paragraph [48, 115].

Sari et al. [105] reduces both the time and energy expended to recover CM errors by placing the design in a highly utilised Pblock, in order to gather the design's essential bits into a small chip area, and scrub only the CFs that contain at least one essential bit, also called essential CFs. The authors in [87] use a deadline-aware scrubbing scheme which dynamically chooses the frame to commence scrubbing with in real-time FPGA systems so as to reduce the number of missed deadlines. Tonfat et al. [122] introduced a customised design flow that places and routes the three modules of a TMR design in a way that all modules have identical CFDATA and MBUs can be corrected by using information from the TMR scheme. A novel technique that uses a lightweight error detection code and erasure codes to detect and correct MBUs in CFs, respectively, is presented in [38]. Bolchini et al. [17] investigated how the number of partitions and the location of inserted voters affect the size and recovery time of modules in a TMR design that incorporates MER. Cetin et al. [31] proposed a scalable token-ring network to transfer to an RC the health status from the modules of TMR designs that incorporate MER. A follow up paper [3] showed that the most reliable and scalable solution for the implementation of such a network is to use the ICAP to readback CFs containing the health status of these modules.

Our work considers the advantages of scrubbing and MER and proposes FMER as a way to reliably and efficiently recover CM errors in SRAM FPGA SoCs of current and future space missions.

3.8 Chapter Summary

In summary, this chapter proposes FMER, an energy-efficient ER technique that targets TMR-based FPGA SoC designs. To demonstrate the efficacy of this ER technique the reliability, availability and energy consumption of various SoC implementations that incorporate either FMER, blind scrubbing, MER or no recovery were modelled and compared. It was shown that MER was the most energy-efficient CM ER mechanism. However, since MER does not recover CM upsets in the SRs of the SoCs it has the lowest CM fault coverage compared with FMER or periodic CM scrubbing alone. Moreover, it was shown that in SoC/Scrub unnecessary energy is consumed refreshing the contents of the FPGA's CM when no upsets are present in this memory. The results demonstrate that SoC/FMER consumes less energy than SoC/Scrub while it always achieves higher reliability and availability than SoC/Scrub and SoC/MER, especially in high radiation environments or on long missions.

As mentioned in Chapter 2, the reliability of a TMR FPGA SoC with FMER depends heavily on the reliability of the RC and the RCN. Although several studies have proposed reliable RCs, the design of reliable RCN architectures has not been studied. We investigate this question next.

Chapter 4

Reconfiguration Control Networks

4.1 Introduction

Two very important infrastructural components in TMR-based FPGA SoCs with MER or FMER are the Reconfiguration Control Network (RCN) and the Reconfiguration Controller (RC). If the RCN fails, the minority report, i.e., which, if any, module is faulty, from the voters of TMR components may never be conveyed to the RC. Similarly, if the RC fails, a reported faulty module may never be reconfigured. Either way, the overall dependability of the SoC will be reduced. Although, several reliable RC architectures have been reported in the literature, an evaluation of RCN architectures and topologies has not previously been conducted.

In this chapter we compare the latency, reliability, scalability and power consumption of several feasible RCN topologies in order to understand how these properties affect the performance of MER mechanisms. Additionally, we explore the idea of using the configuration layer of the FPGA to transfer the minority reports from the voters of TMR components to the centralised RC of the SoC [52]. That is to say, an ICAP-based RCN is implemented, whereby the RC selectively reads back the CFs of the FPGA containing the minority report of each voter in the SoC. The resulting area and failure rate of the ICAP-based RCN is negligible, since almost no programmable routing resources or logic is used between the RC and the minority reports of voters. We refer to those networks implemented on the FPGA's application layer, i.e., with programmable logic, as *application-layer networks*, and those RCNs that are implemented on the FPGA's configuration layer as *configuration-layer networks*, or in short *fabric networks*.

In order to compare the properties, such as the reliability and scalability, of various RCN topologies we conducted the following experiments:

- We deployed star, bus and token-ring application-layer RCNs, as well as ICAP-based configuration-layer RCNs in several synthetic designs that incorporated either 7, 15 or 31 voters.
- The aforementioned RCNs were also deployed in the RUSH CubeSat payload which incorporates 9 TMR components [32]. Please note that both the synthetic and the RUSH designs utilise TMR components with simplex voters. In other words, the TMR components of the designs have the structure of the a_3 sub-systems shown in Fig. 3.1 (a) of Sec. 3.2.

An analysis with respect to the reliability, latency and scalability of the synthetic and the RUSH designs showed that the designs that incorporated the ICAP-based RCN had the best reliability and scalability results, but the worst, yet practically acceptable latency.

Finally, we implemented a version of the RUSH payload that incorporated blind scrubbing instead of MER, in order to determine which of the two CM error recovery mechanisms provides the highest reliability for a GEO mission. Our results show that the RUSH/blind scrubbing payload achieves the highest reliability for the assumed GEO mission unless the simplex RCN in the RUSH/MER payload is itself triplicated and repaired when it is corrupted. These findings come as no surprise. The RUSH/MER payload utilises a simplex RCN, while the RUSH/blind scrubbing design does not utilise an RCN at all. As we showed in Sec. 3.4.1 and Sec. 3.6.4, the reliability of a TMR FPGA SoC is most significantly determined by any simplex component it incorporates.

This chapter is organised as follows: Sections 4.2 and 4.3 present the overall architecture of a TMR FPGA SoC that incorporates MER or FMER, while also providing more details in regards to the RCN of the SoC which is the focus of this chapter. Sections 4.4 and 4.5 present the models we used to evaluate the reliability of our RCN implementations. Section 4.6 presents the fault-injection methodology we followed in order to estimate the soft-error sensitivity of all RCN implementations. Section 4.7 describes our experimental methodology and reports our findings while Section 4.8 reviews the literature available on RCN topologies. A summary of the chapter is given in Section 4.9.

4.2 Overview of Reconfiguration Control Networks

Both the synthetic designs and the RUSH payload that were implemented in this chapter follow the architecture of the design example shown in Fig. 4.1. In more detail, Fig. 4.1 illustrates the overall architecture of a TMR-based FPGA SoC that incorporates MER or FMER for recovering CM upsets. The components of the SoC are divided into three groups:

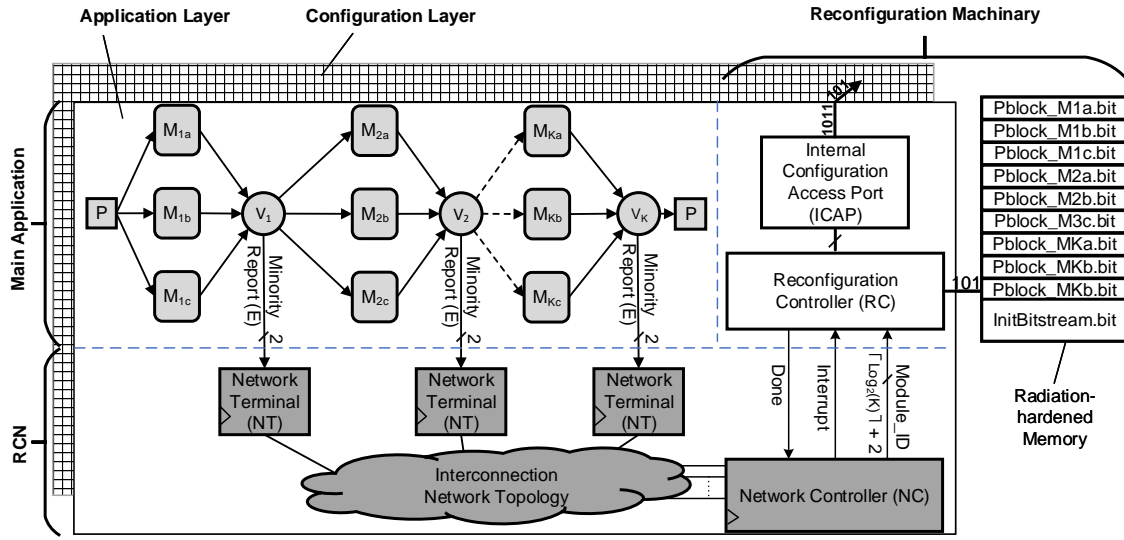


Figure 4.1: Architectural layout of TMR FPGA SoCs with MER or FMER.

- The first group is the *main application*, which has the structure of the a_3 sub-system shown in Fig. 3.1 (a) of Sec. 3.2, and consists of the lightly-shaded K TMR components at the top-left of Fig. 4.1. Please note that depending on the requirements of a mission, the main application can have the structure of any of the $a_1 - a_3$ sub-systems shown in Fig. 3.1 (a) of Sec. 3.2.
- The second group is referred to as the *RCN* and consists of the darkly-shaded components at the bottom of Fig. 4.1, i.e., the K *Network Terminals* (NTs), a centralised *Network Controller* (NC) and an *interconnection network* between these components. Each NT incorporates a 2-bit register for buffering the 2-bit minority report (E)¹ from each of the K voters in the main application, as well as a network interface

¹As mentioned in Chapter 2, the 2-bit minority report of the voters in Fig. 2.7, sets $E = "01"$, $E = "10"$ and $E = "11"$ when the 1st, the 2nd or the 3rd module of a TMR

for establishing a connection with the centralised NC. Similarly, the NC incorporates one or more network interfaces for communicating with the NTs, as well as three ports for communicating with the RC: 1) an *Interrupt* output port that notifies the RC that a module has failed, 2) a $\lceil \log_2(K) \rceil + 2$ bit wide *Module ID* output port that indicates which module has failed, and 3) an input *Done* port that is controlled from the RC to notify the NC that a faulty module has been repaired.

- The third group, referred to as the “*reconfiguration machinery*”, consists of the non-shaded components at the top-right of Fig. 4.1, i.e., the internal RC that uses the ICAP primitive to gain access to the FPGA’s CM and an off-chip radiation-hardened memory for storing the partial bitstreams of the $3K$ modules of the main application.

In normal operation the NC of the SoC continuously collects the minority reports from the K voters in the main application in a round-robin manner until any of these reports indicates that a module is corrupted and therefore needs reconfiguration. In such an event, the NC stops collecting the minority reports and immediately asserts the *Interrupt* signal in order to notify the RC that a module needs reconfiguration. In turn, the RC downloads the partial bitstream corresponding to the faulty module, which is reported on the *Module_ID* port of the NC. Once the faulty module has been reconfigured, the RC waits for t_{sync} time and then asserts the *Done* signal, which in turn notifies the NC to continue collecting the minority reports of the SoC.

As mentioned in Chapter 2, TMR circuits include synchronisation voters throughout all registered feedback paths of the circuit in order to resynchronise the state of any corrupted single TMR domain with the state of the TMR scheme. The time, t_{sync} , it takes to synchronise the state of a module corresponds to the module’s latency, or in other words, the time it takes for new input values to propagate to the module’s outputs. Naively checking for errors as soon as a faulty module is reconfigured otherwise results in an endless loop of reconfigurations, since not enough time has elapsed for the reconfigured module to resynchronise its state with the state of the TMR scheme.

component needs reconfiguration, respectively. The case where no module needs reconfiguration is encoded with $E=“00”$.

4.3 RCN Architecture

In this section, we provide more details about the architecture of the star, bus, token-ring and ICAP-based RCNs that have been implemented and compared in this chapter.

4.3.1 Star RCN

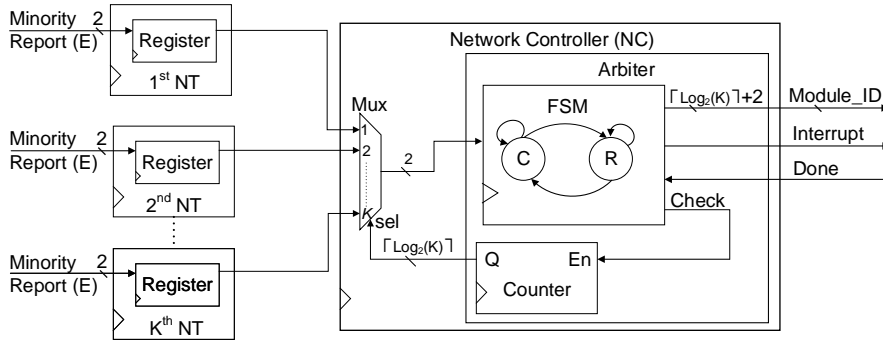


Figure 4.2: Architecture of a star-based RCN.

In the star RCN, shown in Fig. 4.2, each NT is simply a 2-bit register that buffers the minority report of each voter. The NC consists of an arbiter (Counter & FSM) and a multiplexer (Mux). The output Q of the counter drives the sel port of the multiplexer so that the minority report from each voter is collected in round-robin order. The counter is controlled through the FSM, which has two states, namely the *Collect* (C) state and the *Repair* (R) state. The default state of the FSM is C, in which the counter is enabled, i.e., $Check = 1$, and each minority report of the SoC is sequentially collected through the multiplexer of the NC. The FSM transitions to state R in the event of a Reconfiguration Request (RR) or in other words whenever a minority report indicates that a module needs reconfiguration. In state R the FSM asserts the *Interrupt* signal and sets the *Module_ID* signal with the ID of the reported faulty module. The RC then reconfigures the faulty module, waits for t_{sync} time and finally asserts the *Done* signal, which triggers the FSM to transition to its default state C. Please note that the *Module_ID* port is simply a concatenation of the counter (Q) value and the minority report (E) value.

4.3.2 Bus RCN

Fig. 4.3 illustrates the bus RCN, which has the same NC as the star RCN, but without a multiplexer. In the star-based RCN, the output Q of the counter is driving the multiplexer, whereas in the bus-based RCN it is driving the *Address-bus* of the RCN. The 2-bit registered minority report from each NT is connected on a shared *Data-bus* through a 2-bit OR gate. In order to guarantee that only the result of one NT is transmitted on the data bus at a time, the 2-bit registers in the NTs are always held in reset mode except when they are selected for communication by the NC of the RCN. The reset control signal of the 2-bit register in each NT is controlled through an address decoder as shown in Fig. 4.3.

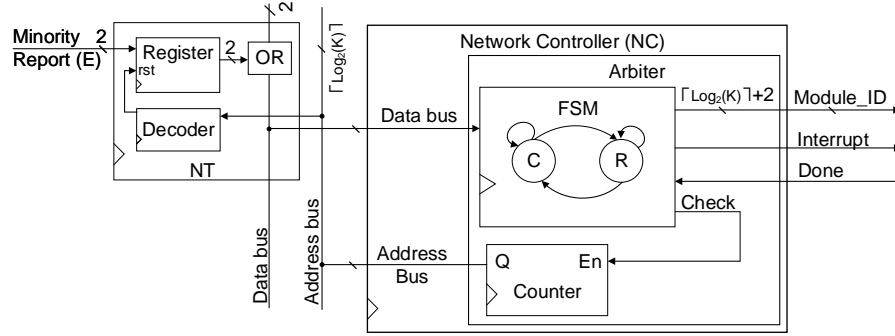


Figure 4.3: Architecture of a bus-based RCN.

4.3.3 Token-ring RCN

Fig. 4.4 shows an abstract schematic of the token-ring RCN. The NTs and the NC are connected in a daisy chain manner and a token continuously traverses the nodes of the ring clockwise. The token acts as permission for an NT to communicate an RR to the NC, which in case of the token-ring RCN, is a message containing the ID of a module in error. In other words, only one NT at a time can transmit an RR message to the NC – the NT that has the token.

When the minority report indicates that all three modules of a TMR component are healthy, the token is simply re-transmitted to its neighbouring node. Otherwise, when the minority report of a voter indicates that a module needs reconfiguration, the NT waits until it receives the token, keeps the received token, and passes a *RR* message with the ID of the corrupted module to its neighbouring node. The *RR* message is passed around the

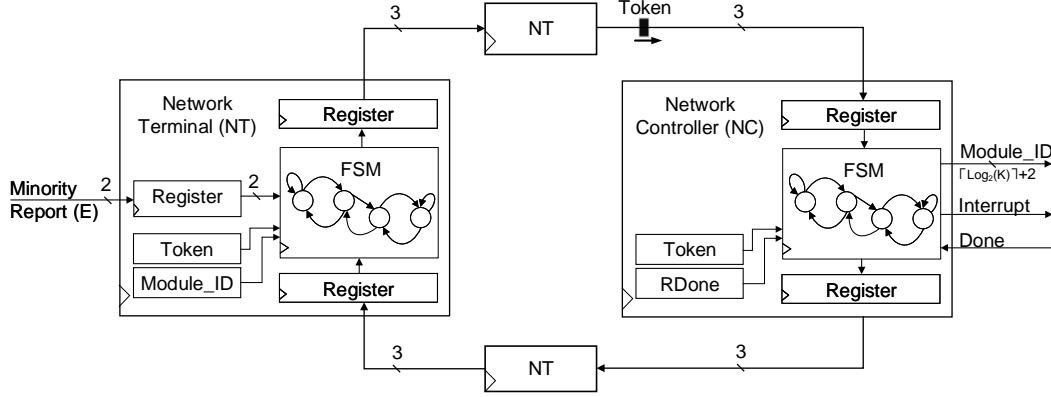


Figure 4.4: Architecture of a token-ring based RCN.

ring until it reaches the NC. Once the NC receives the *RR* message, it notifies the RC that a module in the SoC needs reconfiguration, i.e., by setting the *Interrupt* and *Module_ID* ports with appropriate values. The RC reconfigures the module, waits for t_{sync} time and then raises the *Done* signal, which in turn triggers the NC to send an *RDone* message to the NT of the recovered module. Finally, the token is released when the NT (that transmitted the *RR* message) receives the *RDone* message. Details of the architecture of the token-ring RCN can be found in Chapter 3 of the master thesis [148].

4.3.4 ICAP RCN

The ICAP RCN does not utilise any programmable logic for its implementation except for a 2-bit register at each NT to store the 2-bit minority report of each voter in the SoC, i.e., it uses the same NTs as shown in Fig. 4.2. The RC in the SoC continuously reads back the CFs containing the registered 2-bit value of each minority report until a report indicates that a module requires reconfiguration. In such an event, the RC reconfigures the faulty module, waits for t_{sync} time, and finally resumes reading back the minority reports of the SoC.

The RC takes advantage of the *Readback Capture* feature of Xilinx 4–7 series FPGAs [139, 142] to capture and readback the minority report of each NT in the SoC. In more detail, the RC executes the following three step procedure whenever it reads back the minority report from a NT:

1. Capture the state of the FPGA's UM, i.e., the state of flip-flops, LUTRAM, BRAM

etc., by issuing the “GCAPTURE” command to the ICAP primitive.

2. Selectively, read back the CF containing the captured state of the two flip-flops (mapped to a CLB slice) that store the minority report of the currently checked NT.
3. Check the state of the corresponding captured minority report bits within the read CF.

The exact CF address and bit offset within the CF that corresponds to the captured state of the two flip-flops in each NT of the SoC is given in the logic allocation (*.ll) file that can be generated with the Xilinx Vivado Design Suite [139]. The logic allocation file includes four fields as illustrated in Fig. 4.5:

- *Bit offset*: The bit position within the complete read back CM data of the FPGA;
- *Frame address*: The address of the frame containing the state of the minority report;
- *Frame offset*: The exact position of a captured UM bit within the specified frame address;
- *Resources*: Information about the captured UM resources in the design, e.g., the slices containing the two flip-flops that store the minority report from each NT.

```

<Bit offset> <Frame address> <Frame offset> <Resources>
Bit 7487688 0x0001839f 104 Block=SLICE_X10Y141 Latch=CQ Net=NT[1]/minority_report[0]
Bit 7487710 0x0001839f 126 Block=SLICE_X10Y141 Latch=DQ Net=NT[1]/minority_report[1]
Bit 7487944 0x0001839f 360 Block=SLICE_X10Y145 Latch=CQ Net=NT[2]/minority_report[0]
Bit 7487966 0x0001839f 382 Block=SLICE_X10Y145 Latch=DQ Net=NT[2]/minority_report[1]
...
...
Bit 7676999 0x0001859f 487 Block=SLICE_X19Y147 Latch=CQ Net=NT[K]/minority_report[0]
Bit 7677021 0x0001859f 509 Block=SLICE_X19Y147 Latch=DQ Net=NT[K]/minority_report[1]

```

Figure 4.5: Extract of a Xilinx logic allocation file.

As mentioned in Chapter 3, Xilinx devices expect a specific sequence of commands to be issued to the ICAP in order to read data from a specific CF address [142]. A CF read request necessitates the read of a dummy word and a pad frame before the desired data of the CF can be read. The time to read a frame in Xilinx 7-Series FPGAs is approximately 230 clock cycles. This includes 20 clock cycles for initialising the configuration process, 203 clock cycles to read the CF data (i.e., to read the dummy word, the pad frame and actual CF data), and 10 clock cycles to terminate the configuration process [142].

We calculated that the RC requires approximately 2.3 microseconds to read back a CF when it operates the ICAP tile at its highest frequency (100 MHz) and uses the 32-bit data bus of the ICAP tile. However, the 2.3 microseconds readback latency can be reduced to approximately 1 microsecond by placing the minority report registers in slices located at the bottom of clock regions. When this is done, the readback of a CF can be aborted after the slices containing the minority reports have been read back.

4.4 Latency of each RCN type

RCN latency is defined as the average period of time needed for the NC to receive a reconfiguration request from a voter. As described in Sec. 4.2, all four networks that we have implemented check voters in round-robin order.

Thus, assuming a system with K TMR components, K NTs and one NC, the average latency of the token ring network is given by

$$latency = (K + 1) \times c_{hop} \times \frac{1}{F_{network}}, \quad (4.1)$$

where c_{hop} denotes the number of clock cycles per node hop, and $F_{network}$ denotes the maximum clock frequency of the RCN. Eq. (4.1) corresponds to the average time needed for the token to arrive (half the ring) and the time for the request to make it back to the NC (also half the ring).

The RCN latency for all other topologies is given by

$$latency = \frac{K}{2} \times c_{hop} \times \frac{1}{F_{network}}, \quad (4.2)$$

which corresponds to the time it takes to check half the voters in the system before the one that wishes to raise a reconfiguration request is checked.

4.5 Reliability Analysis

In this section, we analyse the reliability of the synthetic and the RUSH SoC designs by using the reliability models that were derived in Chapter 3. In more detail, we outline how we model the reliability of a simplex component, the reliability of a TMR component and the reliability of a complete FPGA SoC composed of both simplex (the RCN) and TMR components (the main application). Our analysis is based on the number of critical

bits per component for which we use the number of essential bits reported by the vendor's tools as a worst case estimate.

In the following, we assume that the flip of an essential bit leads to a module failure. With this assumption, the module failure rate λ_m is given by the product of the bit error rate, λ_{bit} , and the number of essential bits (*Ebits*) in module m . We also assume that the three modules of an TMR component have the same failure rate λ_m .

We use the *Peak 5-minute GEO* CM upset rate model of Table 3.1, i.e., we assume that the upset rate of each CM bit in the Artix-7 FPGA (used in our experiments) is $\lambda_{bit} = 2.7 \times 10^{-10}$ upsets/bit/s.

We assume that module reliability decreases exponentially over time t as expressed by the function:

$$R_m(t) = e^{-\lambda_m t}, \quad (4.3)$$

whereby the reliability, $R_m(t)$, of a module at time t denotes the probability that the module operates without any failure in the interval $[0, t]$.

When module m is triplicated, its reliability function becomes:

$$R_m^{TMR}(t) = 3R_m^2(t) - 2R_m^3(t). \quad (4.4)$$

In order to achieve higher reliability, for a given SEU rate, we employ TMR with MER. The reliability function is then given by

$$R_m^{TMR+Recovery}(t) = \frac{e^{-\frac{1}{2}(at)} \left(a \sinh\left(\frac{bt}{2}\right) + b \cosh\left(\frac{bt}{2}\right) \right)}{b}, \quad (4.5)$$

where $a = 5\lambda_m + \mu_m$, $b = \sqrt{\lambda_m^2 + 10\lambda_m\mu_m + \mu_m^2}$.

The term μ_m denotes the repair rate of a module, which is the reciprocal of the time needed to recover the faulty module:

$$\mu_m = \frac{1}{t_{repair}} = \frac{1}{t_d + t_c + t_{sync}} \approx \frac{1}{t_d + t_c}, \quad (4.6)$$

where t_d denotes the average error detection time, t_c denotes the error correction time and t_{sync} denotes the synchronisation time, which we omit in our case study because it normally only accounts for a small fraction of the recovery time.

Note that t_d depends on the method used to detect errors and corresponds in our case to the average latencies that were derived in Eqs. (4.1) and (4.2), whereas t_c , which depends

on parameters of the target system and the size of the module, is given by the number of 32-bit words per frame, the number of frames in the given TMR module and the ICAP write throughput.

The reliability of an FPGA-based system composed of K TMR components that use MER to recover from CM errors and an RCN for aggregating reconfiguration requests can be derived as follows. We model the reliability of the RCN $R_{RCN}(t)$ using Eqs. (4.3, 4.4 or 4.5). Respectively, the reliability of each TMR component $R_{i,r}^{TMR}(t)$ in the system is modelled using Eq. (4.5). Finally, the reliability of the system is given by the product of the reliability of each individual component, namely the RCN and the K TMR components [112]:

$$R_s^{TMR}(t) = R_{RCN}(t) \prod_{i=1}^K R_{i,r}^{TMR}(t). \quad (4.7)$$

In this derivation, it is assumed that failures follow a Poisson distribution and the occurrence of errors in modules or components are statistically independent and uncorrelated. Note that Eq. (4.7) holds true only if $\mu \gg \lambda$, which ensures repairs are completed independently [112]. Moreover, since the main objective of this paper is to evaluate the impact of various RCN architectures on the total reliability of FPGA-based designs that incorporate MER, we omit inclusion of the reconfiguration controller and the voters in our reliability analysis.

4.6 Fault Emulation System

In this section, we outline the fault emulation system we implemented to assess the soft error sensitivity (SES) of the RCNs we studied. A typical fault emulation system requires mechanisms to emulate CM upsets in the FPGA by flipping CM bits (i.e., injecting faults), as well as mechanisms to stimulate and test the application circuit after a fault has been injected [95]. We incorporated a Xilinx MicroBlaze (MB) processing system into our experimental platforms in order to inject faults into the CM of the RCNs and to check their functionality. The complete fault-injection procedure is controlled by a program running on a PC. The MB processing system incorporates the Xilinx UART, HWICAP [140] and GPIO IP cores for communicating with the PC, injecting CM faults and testing the RCN circuits, respectively. Please note that we inject faults into random CM bits of the RCNs. Of the 18,300 configuration frames in the Artix-7 XC7A200T targeted in our study, 14,250 frames are contained in the clock regions we used to implement the RCNs.

Fig. 4.6 outlines the fault-injection procedure we followed in order to evaluate the SES

of each RCN. Once the FPGA is configured and initialised, the MicroBlaze informs the PC that it is ready to inject a CM fault. In turn, the PC sends to the MB the address of a random CM bit that is to be tested, i.e, the address of a random CM bit in the 14,250 frames used to implement the RCNs. The MB reads the corresponding CF, flips the corresponding bit within the CF and writes the CF back into the FPGA's CM to emulate a CM upset.

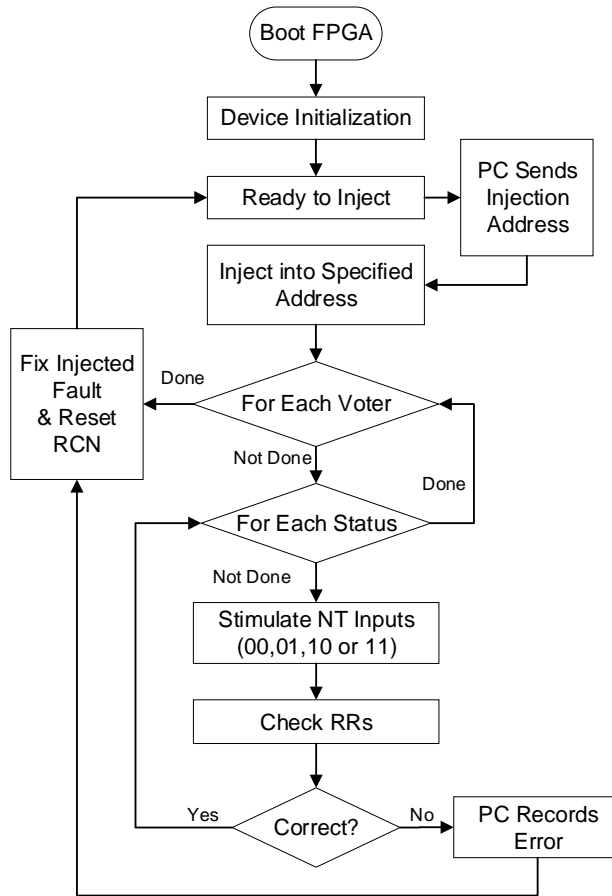


Figure 4.6: Fault-injection flowchart.

Once a fault is inserted into the CM of the FPGA, the input of each NT in the RCN is driven with all four possible minority report (E) values that can occur during a mission. The input stimuli for each NT is provided through LUTRAM as shown in Fig. 4.7. In more detail, the LUTRAM is composed from two SLICEM LUT (LUTM) primitives that are configured as distributed RAM or in short LUTRAM. These LUTMs are reconfigured at runtime via the ICAP to stimulate the inputs of the NTs. Given the site number and

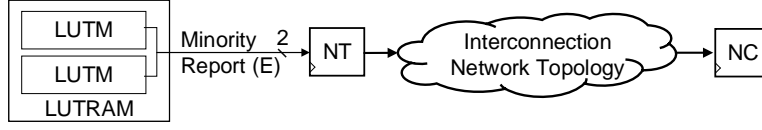


Figure 4.7: Input stimulus of each network terminal in the SoCs.

logic locations, the positions of the LUTM bits can be obtained from the `*.11` file as described in Sec. 4.3.4.

The MB processing system, injects a fault into the RCN, i.e., the NTs, the NC and their interconnections, and then checks the integrity of the design by observing the output of the NC, while stimulating the inputs of the NTs. As mentioned above, for each NT we iterate through every possible combination of the minority report values ($E = "01"$, $E = "10"$, $E = "11"$) while holding the inputs to every other NT constant at $E = "00"$, which signifies the “no error” condition. Whenever a new minority report value is written to a NT, the MB checks that the correct RR is received. In the case of the application-layer RCNs, we always wait for the maximum number of clock cycles required for the NC to receive an RR, before reading the *Interrupt* and *Module_ID* signals of the NC. In the case of the ICAP-based RCN, the MicroBlaze processor utilises the ICAP to read back the values of each NT’s 2-bit register in order to determine the RR. If the RR is as expected, we change the minority report signal to the next value. When we have cycled through every possible status and there is no unexpected RR, we move on to the next NT. If an unexpected RR is received, an error report is sent to the PC.

The fault emulation tool must also remove the injected fault and return the circuit to a known functioning state before injecting the next fault. In our system, the injected fault is fixed by writing back the frame as it was before injection, all NT inputs are set to “00”, the RCN is reset to its initial state (i.e., the registers and the FSMs of the NTs and NC are reset to their initial default state) and the MB returns to wait for a new fault-injection address from the PC.

4.7 Experiments and Results

In this section we evaluate the performance of the RCNs presented in Section 4.3, in terms of post-routing resource utilisation, latency, operating frequency, power consumption and soft-error sensitivity. All networks have been implemented on a Xilinx Artix-7 XC7A200T

FPGA using the vendor’s Vivado design suite 2014.4 with default settings.

4.7.1 Experimental Methodology

Synthetic layout designs

In a first experiment we studied “synthetic” layouts in which the TMR components, their voters, and thus the NTs were distributed in a checkerboard pattern across the majority of the device area. Moreover, the NTs and the NC were always located in partitions that utilised the same FPGA resources irrespective of the RCN topology under test. To obtain resource utilisation and performance results, we initially implemented designs that only contained the components of the RCNs being tested and constrained the implementation tools to prevent optimisations across the port interfaces of the NTs and the NC. To perform the fault-injection experiments, we added a MicroBlaze-based RC for injecting faults and connected a distributed RAM-based test vector (Fig. 4.7) to the input of each NT in the RCN under test. We tested each RCN type for K equal to 7, 15 and 31 NTs. The synthetic layout of a 31-voter design (in this case for testing the star network topology) is shown in Fig. 4.8(a), in which the RCN into which faults were injected is depicted as the shaded region to the right of the RC.

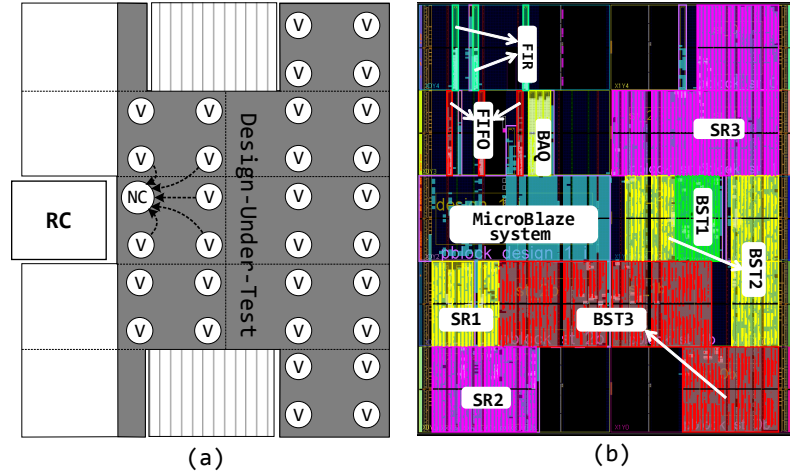


Figure 4.8: a) Synthetic layout of a 31-voter design and b) RUSH floorplan.

RUSH payload designs

In a second experiment, we investigated the utilisation and performance of each RCN when used to collect reconfiguration requests for the RUSH CubeSat payload board [32]. For this case study, we implemented the four RCN types (i.e., star, bus, token-ring and

ICAP) with the 9 TMR components comprising the RUSH payload. These components include a single MAC-based 21-tap *Finite Impulse Response* (FIR) filter with 16-bit signal width, an 8-to-3-bit *Block Adaptive Quantizer* (BAQ), an 8,096-word deep 32-bit *FIFO*, three 32-bit *Shift Registers* (SRs) having different lengths and a range of combinational logic between the stages and three 32-bit *Binary Search Trees* (BSTs) of different heights and a range of combinational logic at each node. A MicroBlaze soft-processor with the HWICAP IP [140] was used to implement the RC. The layout of this system is depicted in Fig. 4.8(b).

4.7.2 Implementation results of the synthetic layout designs

Table 4.1 presents information extracted from the the vendor’s implementation tools. The results are listed according to the resource utilisation of each design; the dynamic power consumption and the number of essential bits follow the same pattern. An exception to this trend is the static power consumption which was the same for all RCN designs. Given that the designs utilised less than 0.2% of the total FPGA resources on average, we believe that the contribution of the RCN to the total static power consumption of the FPGA is negligible, and due to this we have obtained the same result for all designs.

As expected, the ICAP-based RCN was realised with the fewest resources compared to the other RCN architectures. This is primarily because the ICAP NTs are implemented with just two FFs and a small amount of support logic being mapped to LUTs. As expected, the number of PIPs and Switch Matrices (SMs) used by the ICAP approach is significantly lower than for the other approaches. As a consequence, the ICAP-based RCN has on average 2.7, 3.6 and 6.0 times fewer essential bits than the synthetic layouts of the star, bus and ring networks respectively. However, the ICAP-based RCN suffers from high network latency. It requires two to three orders of magnitude more time than the other RCNs to transfer reconfiguration requests to the NC. In contrast, the ring has the lowest latency, since it can achieve a higher operating frequency and only needs 1 clock cycle per node hop. We used Eqs. (4.1) and (4.2) to calculate the latency for each RCN. The latency of the ICAP approach is on average over 175 times that of the ring and the latency of the star and bus networks was about 1.4 times that of the ring for the synthetic layouts.

We investigated an optimisation of the ICAP RCN that entails constraining the registers of those groups of NTs that are located within each clock region. These registers are forced to be placed into a single configuration frame so that they can be accessed in a single frame

Table 4.1: Results of implementing four type of RCNs on a Xilinx Artix-7 XC7A200 FPGA

Type	ICAP				STAR				BUS				RING			
	L1		L1*		L1		L2		L1		L2		L1		L2	
# NTs	7	15	31	31	7	15	31	9	7	15	31	9	7	15	31	9
Slices	7	15	31	31	12	29	50	18	21	33	60	21	30	50	141	35
LUTs	0	0	0	0	14	27	30	16	28	50	108	33	54	130	279	87
FFs	14	30	62	62	26	44	77	32	35	61	110	43	61	134	295	87
PIPs	440	889	1,770	1,858	1,101	1,996	3,513	1,243	1,341	2,553	4,625	1,729	2,057	3,894	7,986	2,724
SMs	38	62	102	181	277	453	792	274	351	616	1074	466	426	496	861	426
Freq. (MHz)	100				112	109	107	126	109	107	104	114	132	203	186	145
Clocks / Hop	230				2				2				1			
# hops	7	15	31	8	9	7	15	31	9	7	15	31	9	8	16	32
Latency (us)	8.05	17.25	35.65	9.20	300	0.06	0.14	0.29	0.5	0.06	0.14	0.30	0.5	0.07	0.08	0.18
Static (mW)	138	138	138	138	138	138	138	138	138	138	138	138	138	138	138	138
Dynamic (mW)	3	4	5	5	3	4	7	4	6	7	9	5	4	5	8	4
Ess. bits (Kb)	3.42	5.6	10.4	13.7	4.1	9.4	15.8	26.0	10.1	11.9	20.1	38.6	14.9	18.3	33.7	69.4

L1: Synthetic layout L2: RUSH layout L1*: optimised ICAP layout

read. With reference to Fig. 4.8(a), which depicts 4 voters per clock region (the 10 grey rectangles), this optimisation resulted in the creation of horizontal wires leading from each voter to a frame that was centrally located in each clock region. Instead of requiring 31 separate frame reads to check all voters, this approach reduced the number of frame reads needed to 8 in total — one for each clock region used by the design. The results of this implementation are reported in Table 4.1 in the ICAP column headed L1*. As can be seen, this optimisation reduced the latency of the ICAP approach by a factor of 4 while increasing the number of essential bits used over the unoptimised 31-voter ICAP design by 32%.

4.7.3 Fault-injection results

Table 4.2 tabulates the average number of functional errors (FEs) we found after five trials of one million fault-injections as described in Sec. 4.6. These results demonstrate that the ICAP-based RCN is more reliable than the other approaches. Additionally, the number of errors that occur in each RCN is directly proportional to the number of voters and thus the number of essential bits per design.

Table 4.2: Average number of functional errors (FEs)

Type	ICAP		STAR		BUS		RING	
# voters	FEs	SD	FEs	SD	FEs	SD	FEs	SD
7	7.0	1.5	8.2	2.3	16.8	2.1	51.0	4.7
15	8.2	3.5	17.0	3.0	36.6	5.0	122.1	16.7
31	20.7	1.4	38.6	4.6	78.6	7.9	213.4	27.3

SD: Standard deviation

4.7.4 RUSH case study results

Table 4.3 presents the resource utilisation and essential bits of each TMR component in the RUSH payload. Additionally, the 6th column of the table presents the average failure rate, λ_m , per module in each TMR component, while the last two columns of the table present their average number of CFs, F_M , and recovery time, t_c , respectively. Please note that of all TMR components in the RUSH payload, only the FIFO component utilises BRAMs, and to be more specific, 15 BRAM16 (2.05%). Moreover, we calculated the failure rate per module with $\lambda_{bit} = 2.7 \times 10^{-10}$ upsets/bit/s, and as mentioned, we assumed that all CM upsets are critical. Last in our design, since we were using the AXI HWICAP,

Table 4.3: Results of implementing 9 TMR components on a Xilinx Artix-7 XC7A200 FPGA

Cmpnt	Utilisation			Essential Bits	λ_m upsets/s	F_M	t_c ms
	LUTs	FFs	DSP	Ebits			
FIR	33 (0.02%)	16 (0.01%)	1 (0.13%)	12,0K (0.02%)	3.25×10^{-6}	65	1.2
FIFO	72 (0.05%)	111 (0.04%)	–	41,8K (0.07%)	1.13×10^{-5}	192	3.5
BAQ	305 (0.22%)	197 (0.07%)	–	49,0K (0.08%)	1.32×10^{-5}	73	1.3
BST1	1,4K (1.04%)	2,5K (0.94%)	–	281,6K (0.46%)	7.60×10^{-5}	145	2.6
SR1	1,6K (1.20%)	3,3K (1.22%)	–	285,9K (0.46%)	7.72×10^{-5}	378	6.8
SR2	2,6K (1.96%)	5,5K (2.05%)	20 (2.70%)	515,9K (0.84%)	1.39×10^{-5}	474	8.5
BST2	3,8K (2.84%)	6,2K (2.32%)	31 (4.18%)	793,5K (1.30%)	2.14×10^{-4}	610	11.0
SR3	7,0K (5.24%)	14,6K (5.44%)	40 (5.40%)	1,403,6K (2.30%)	3.79×10^{-4}	1,1K	19.6
BST3	9,1K (6.82%)	12,2K (4.56%)	31 (4.18%)	1,833,2K (3.00%)	4.95×10^{-4}	1,5K	26.7

the ICAP throughput using the MicroBlaze was limited to 10 MB/s, considerably less than the maximum possible throughput of 400 MB/s. The reduction in ICAP bandwidth also affected the latency for checking a voter using the ICAP to 60 us, and we therefore observed a much higher network latency.

Fig. 4.9 plots the system reliability for each RCN type and the 9 RUSH application circuits using Eq. (4.7) against the reliability of a blind scrub implemented on the same system. The MicroBlaze RC and off-chip flash configuration storage used by the RUSH system supports a random FPGA configuration frame read latency of 60 us and a sustained frame write period of 18 us per frame. Blind scrubbing, which entails rewriting each configuration frame of the device, therefore takes 330 ms on the Artix-7 200T used, and errors are recovered by scrubbing after 165 ms on average. Please note that in Fig. 4.9, the scrub plots only account for the 9 application components; they specifically exclude an RCN component, which is not needed for blind scrubbing. Fig. 4.9(a) assumes the four

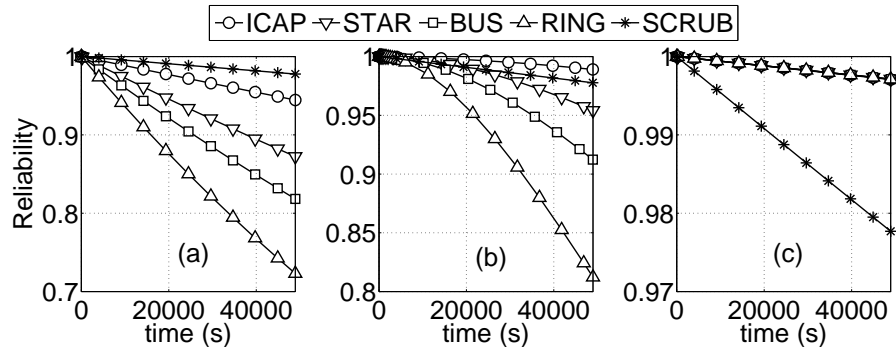


Figure 4.9: a) Unprotected RCN b) TMR triplicated c) TMR triplicated with recovery.

RCNs are implemented as simplex components. While the ICAP RCN results in the best

reliability for MER, all four RCNs weigh down the reliability of the system because they are single points of failure.

Fig. 4.9(b) assumes the RCNs are implemented as triplicated components, but that errors that occur in these components are not repaired. The RCN is implemented outside the $3K$ PBlocks hosting the $3K$ modules, and therefore do not recover from CM upsets when MER is used. Only some limited error mitigation is therefore in place. Only the ICAP outperforms scrubbing over the time period shown. However, eventually (when $t > 120,000$ s) even this approach succumbs to errors that remain unrepaired and scrubbing once again dominates.

In Fig. 4.9(c) we assume that the device is partially reconfigured in its entirety when an error in the triplicated RCN component is detected. This error recovery period is longer than desired, but the approach ensures any error in the network is corrected. Despite the long recovery time (equivalent to reconfiguring the complete device), the reliability is not significantly affected because errors occur infrequently in the relatively small RCN components.

4.8 Related Work

Several network topologies and architectures have been proposed for Network-on-Chip (NoC) communications in TMR-based SRAM FPGA circuits, from which, some, were explicitly proposed for RCNs, [15, 29, 120], while, others, for more general-purpose NoCs [81, 130].

This section provides a literature survey of star, bus and token-ring application-layer networks, as well as fabric (i.e., configuration-layer) networks. Fig 4.10 illustrates the conceptual models of star, bus, ring and fabric RCNs.

4.8.1 Star-based RCNs

Star-based RCNs use simple network interfaces to connect the minority reports from each voter in the SoC to a central NC [15, 120]. However, the interconnecting nets between the NTs and the NC typically span a large area across the FPGA, as shown in Fig. 4.10(a), therefore passing through numerous programmable interconnection resources which increases the failure rate as well as the critical path in the design. Most star-based RCNs

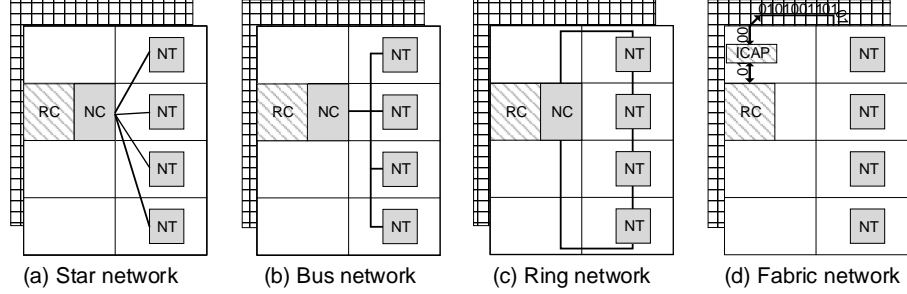


Figure 4.10: Conceptual models of RCN topologies.

described in the literature utilise an NC that polls the minority reports in a round-robin manner, but it is also feasible to consider an interrupt-driven approach, whereby the voters in the SoC interrupt the NC or the RC when a module requests reconfiguration.

4.8.2 Bus-based RCNs

Several works have utilised bus networks in TMR circuits, but none of these explicitly targeted RCN communications. For example the authors in [81] used the Advanced eXtensible Interface (AXI) bus to transfer the outputs of individual TMR modules to a centralised voter, thereby reducing the area overheads of the TMR scheme.

Bus networks can be advantageous over star networks since their shared data and address bus wires allow new modules to be readily integrated into the system. On the other hand, the network interfaces of nodes in bus networks are more complex than those in star networks, which increases their resource utilisation and soft-error sensitivity.

4.8.3 Token-ring RCNs

A token-ring RCN that supported several interesting features, such as transferring the minority report between TMR components with different operating frequencies, as well as forward error correction techniques to mitigate soft-errors during data transmission was proposed in [29]. In this chapter we implemented a simpler version of the token-ring RCN proposed that utilises less resources in order to reduce its failure rate.

An important aspect of token-ring RCNs is that they scale better than star or bus networks as the number of NTs increase in the SoC. Token ring RCNs typically link nearby NTs

and therefore utilise more local than global wire segments of FPGA to interconnect these NTs. In contrast, star and bus topologies interconnect NTs at a range of distances, thereby utilising both local and global wire segments of the FPGA. However, SRAM-based FPGAs integrate more local than global wires and therefore token-ring RCNs, which tend to utilise more local wires, are considered to be more scalable than star and bus RCNs when implemented in such devices.

4.8.4 Configuration-layer RCNs

Like bus RCNs, configuration-layer RCNs have not been reported in the literature. However, the idea to utilise the configuration layer of the FPGA in order to transfer data between the components of FPGA circuits was first introduced in [52], and it was first applied in TMR circuits in [130]. In more detail, the authors in [130] used the ICAP primitive in a Xilinx FPGA to transfer the results from triplicated modules to a centralised processor, which in turn applied software-based majority voting on the read back results. The centralised majority voting technique was used as a secondary backup mechanism in case the hardware voters of the TMR circuit failed due to soft-errors [130].

Needless to say, an FPGA SoC without an RC cannot implement a configuration-layer network. However, TMR FPGA SoCs that employ MER or FMER integrate an RC anyway, and therefore can take advantage of the DPR features of modern Xilinx FPGA to implement a lightweight configuration-layer RCN. The most important benefit of fabric network implementations is that they utilise a negligible amount of programmable routing resources. This overcomes problems such as increased routing congestion or increased area (and therefore failure rate) when the FPGA SoC embeds a large number of TMR components that need to connect to the RCN.

4.9 Chapter Summary

In this chapter, we compared four RCN types in terms of reliability, scalability, resource utilisation, power consumption and soft error sensitivity. The utilisation and performance of these RCNs were assessed for synthetic designs with 7, 15 and 31 voters. The results demonstrate that the ICAP-based readback approach, which uses the configuration layer of the FPGA to transfer reconfiguration requests, utilises the least resources of those networks studied.

The experimental results of a case study that was implemented on the RUSH payload reveal that the ICAP-based readback approach has the highest system reliability despite having a relatively high latency. This higher latency may not be too problematic except when radiation levels become much higher than the high rate assumed in our work. We have shown that the latency of the ICAP approach can be reduced by clustering the registers that are to be read from one clock region into a single frame. This optimisation does not have a significant impact on the resource utilisation. We have also determined that for the reliability of an FPGA SoC with MER to be competitive with a system with periodic device scrubbing, the RCN must be triplicated and repaired when SEUs affect it. Therefore, our TLegUp toolflow, that is presented in the next chapter, incorporates a triplicated RCN when FMER is incorporated in the FPGA SoCs.

Chapter 5

TLegUp: High-Level Synthesis of TMR FPGA circuits

5.1 Introduction

State-of-the-art SRAM FPGA SoCs embed numerous scalar processing units (e.g., CPUs), vector processing units (e.g., DSPs), as well as a large amount of programmable logic (PL) [146], which makes them key components for implementing complex computing systems on a single chip. This trend is enhanced by the development of sophisticated high-level synthesis (HLS) tools that enable researchers and practitioners to increase their productivity and rapidly realise high-performance and energy efficient FPGA SoCs [33,80]. Although HLS is not a new concept, it has only recently started gaining traction in the FPGA design industry, selectively, or completely replacing HDL, such as Verilog, in which a digital circuit is described at the RTL rather than at a more abstract algorithmic level.

Typically, hardware/software FPGA SoCs are developed by coding the hardware part of the design with RTL HDL and the software part of the design using a programming language such as C or C++. However, the escalating complexity of these systems has motivated engineers to specify the functionality of the entire design using a high-level language, avoiding RTL coding altogether. This development methodology also affords the opportunity to conduct rapid design space exploration, such as to experiment with and verify various algorithms, hardware/software partitioning styles and to explore trade-offs between area, power and performance. Moreover, it is common these days to use a high-level modelling language, such as the SystemC transaction-level modelling (TLM)

language, to explore and verify design architectures at a very early development stage of the SoC. Once a suitable architecture is found and verified, it is much simpler to turn the HLS program into an RTL specification using HLS tools rather than having to proceed to manual re-coding in RTL. Finally, the design cycle of an SoC is significantly shorter when it is developed in a high-level language. For example, describing the functionality of a 1M-gate design with RTL typically requires 300K lines of code, while with a high-level programming language, e.g., with ANSI C, the same 1M-gate design can be described with 30K – 40K lines [33].

Given the significance of HLS tools for developing SoC designs, this chapter explores the benefits of triplicating a design at the HLS stage, rather than at the RTL or post-synthesis stages, which most mainstream tools, such as the XTMR or BLTMR tools do. In more detail, we propose an HLS toolflow, which we have called TLegUp, in order to compile HLS programs specified in ANSI C to TMR RTL designs that can be implemented on Xilinx 7-series FPGAs. TLegUp is an extension of the LegUp HLS research framework [24] and the triplication of the HLS designs is performed within the Low-Level Virtual Machine (LLVM) [63] compiler Intermediate Representation (IR)¹ of the framework, before allocation, scheduling, and binding take place. TLegUp triplicates a design, partitions it into k TMR components, and optionally floorplans the modules of the TMR components using the academic tool [100]. As mentioned in Sec. 2.3.4, floorplanning the design reduces resource sharing between TMR domains and therefore mitigates Domain Crossing Errors (DCEs). Automatic floorplanning also enables designers to readily employ MER or FMER mechanisms in their SoCs.

We have investigated a fine- and a coarse-grained approach to partitioning the TMR generated designs into a network of k TMR components, as shown in Fig. 2.4 of Sec.2.3.1. The fine-grained approach, which we refer to as Instruction-Level Partitioning (ILP), uses a max-flow, min-cut algorithm [67] to partition the Data Flow Graph (DFG) of the entire computation of the C program at the LLVM IR instruction level, after all C function calls of the program are inlined and translated into LLVM IR. In contrast, the coarse-grained approach, which we refer to as Functional Level Partitioning (FLP), prohibits the C functions from being inlined during C compilation and uses the same max-flow, min-cut algorithm as used in the ILP approach to cluster the application’s C functions into partitions. Both, the ILP and the FLP approaches aim to balance resource utilisation between partitions while minimising the total bit-width of signals interconnecting them.

As shall be further explained, we examined the FLP approach with a view to reducing

¹LLVM IR is a machine-independent RISC-like instruction set.

the number of wires and voters used to interconnect partitions, in order to overcome routing congestion problems when implementing the designs, especially when they are floorplanned. As we show in Sec. 5.2.3, LegUp generates a hardware module for each C function call in the `main` C function, and each of these modules has its own local FSM for controlling its datapath. Therefore, a TMR design that is partitioned with the ILP approach has only one centralised FSM to control the datapath of the entire circuit since, as mentioned, ILP inlines all C function calls into the application's `main` function in order to generate a DFG of the entire computation and partition the design. Therefore, more partitions result in more wires and as a consequence in more voters to interconnect the centralised FSM with the partitions (TMR components) of the circuit. This has negative effects on the resource utilisation, operating frequency and the resource balance between the partitions of the ILP circuits. In contrast, the TMR circuits that are partitioned with the FLP approach do not suffer from any of the aforementioned negative effects since each partition consists of several Verilog modules (which correspond to C function calls) and each of these Verilog modules contains its own private FSM. Please note that hereinafter we refer to Verilog hardware modules as Vmodules in order to distinguish them from the triplicated modules (TMR domains) of a TMR scheme. Also note that a partition is equivalent to a TMR component as shown in Fig. 2.4 of Sec.2.3.1, and that TLegUp specifies in RTL each TMR domain with one Vmodule in the case of ILP designs, and with multiple Vmodules in the case of FLP designs.

Currently, there are three ways to compile an HLS design description to a TMR RTL design. The first way is to code TMR schemes directly into the HLS design (e.g., directly in C), which, as investigated in [37], results in a partially triplicated circuit. The reason for this partial triplication of the circuit is the inflexibility of inserting synchronisation voters throughout all feedback paths of the design as well as triplicating the design's FSMs when a TMR scheme is coded directly into the HLS program [37]. The second way is to first compile the C HLS program to an RTL design and then to use a tool like R4R [14], BLTMR [20] or XTMR [141] to triplicate the design at the pre-synthesis or post-synthesis level. The third way is to triplicate a design during HLS as proposed in [111,134] and in this chapter. In contrast to TLegUp, the tool of [111,134] does not insert synchronisation voters into the TMR design and also has not been validated with fault-injection experiments. As we show in our experimental results, inserting synchronisation voters is not a straightforward procedure. Voter insertion can have considerable side-effects on the resource utilisation and operating frequency of the design. Additionally, during the development of TLegUp, we had to go through a lengthy cycle of implementation and fault-injection experiments in order to be sure that redundant logic was not optimised during RTL synthesis.

Fig. 5.1 illustrates how an HLS C program can be synthesised to RTL and then triplicated with BLTMR. The HLS program is first compiled to RTL source with an HLS tool, say LegUp, and thereafter synthesised to a proprietary post-synthesis netlist. The netlist is thereafter converted to an Electronic Design Interchange Format (EDIF) netlist before being provided as input to the BLTMR tool. The BLTMR tool triplicates the design and the EDIF netlist (TMR design) is converted back to a proprietary netlist, in order to be technology mapped, placed and routed with vendor CAD tools like the Vivado design suite.

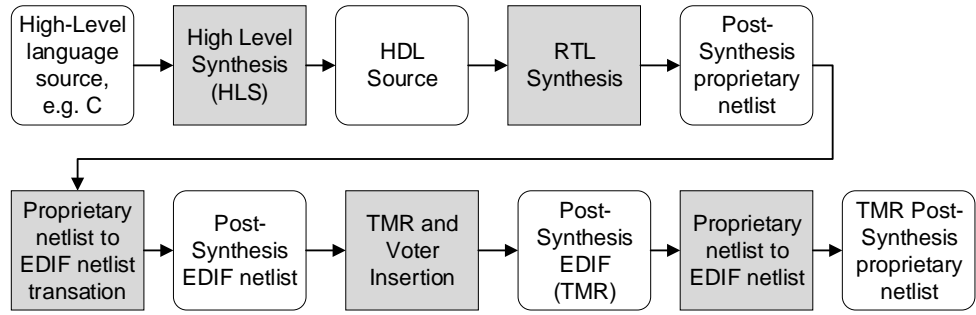


Figure 5.1: Typical design flow for generating a TMR design version of an HLS program.

In contrast to the work of dos Santos et al. [37], the toolflow illustrated in Fig. 5.1 is able to generate a fully triplicated design with TMR FSMs and synchronisation voters inserted throughout all feedback paths of the design. However, we believe that the opportunity for rapidly modelling, exploring and verifying TMR designs using this flow is diminished for the following reasons.

Firstly, it is difficult to quickly explore trade-offs between reliability, performance, area, and power of various TMR design versions because there is a lengthy design cycle from compilation to synthesis and profiling, which makes it difficult to conduct rapid design space exploration.

Secondly, triplicating a design at the RTL pre- or post-synthesis stages has the considerable drawback that the circuit’s schedule needs to be preserved while voters are inserted into the data- and control-paths; the design’s critical path length is consequently increased and the flexibility to pipeline and retime the design are hampered. In contrast, triplicating a design at the HLS level, before scheduling and binding occur, provides the opportunity to pipeline and retime the design in order to mitigate the negative timing effects of design triplication and voter insertion. This is possible because the FSM and the datapath of the TMR design are synthesised (i.e., not modified during the pre- or post-synthesis stages) when triplication is conducted during HLS.

Finally, tools that triplicate the design and insert voters at the post-synthesis stage need to be fully aware of all low-level architectural details of the targeted FPGA. For example, Fig. 5.2 depicts a 2-bit ripple-carry adder that is mapped to two slices (i.e., one CLB) of a Xilinx Virtex FPGA [54]. Let us assume that the adder illustrated in Fig. 5.2 is triplicated and used in a TMR design and that nets A and B are chosen as optimal locations for inserting synchronisation voters in order to cut feedback paths in the design with the least cost (e.g., area or performance). Although net A may be an optimal location for inserting a synchronisation voter in the TMR design, doing so, will corrupt the post-synthesis netlist of the design because there is no PL between the output of MULTAND and the input of MUXCY primitives of the FPGA's slice to implement the voting logic. On the other hand, inserting a voter on net B may be possible but doing so will negate the benefits of using the FPGA's high-speed carry chain and the ripple-carry adder will experience a considerable performance penalty.

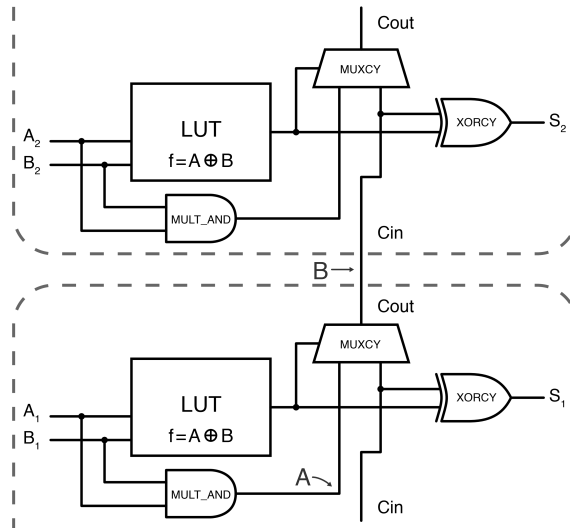


Figure 5.2: Implementation of a 2-bit ripple adder on a Xilinx Virtex FPGA [54].

TLegUp triplicates a design at the HLS stage and the output RTL is synthesised with the Vivado design suite tools. This reduces the complexity of triplication since Vivado accounts for the low-level architectural details of the FPGA when synthesising and optimising the TMR RTL design. However, as discussed in Sec. 2.3.5, TMR RTL designs need to be carefully constrained in order to prevent the Vivado synthesizer from optimising the redundant logic of the design. In contrast, such problems do not exist when triplication is performed at the post-synthesis stage, e.g., like XTMR and BLTMR tools do, since redundancy is inserted into an already optimised netlist of the design.

We have triplicated, implemented, analysed and tested numerous TLegUp generated TMR

RTL designs in order to assure that the Vivado tools do not optimise redundant logic and that the generated designs are of high-quality. In more detail, we used the TLegUp toolflow to implement non-floorplanned and floorplanned TMR designs of 17 CHstone, DWARV and Bambu HLS benchmarks [80] on a Xilinx Artix-7 200T FPGA and compared them against simplex (non-triplicated) design versions. The TMR designs were automatically partitioned with ILP and FLP into $k = 1, 2, 4$ and 8 TMR components. The quality of these designs was quantified in terms of resource utilisation, maximum frequency, latency, execution time and soft error sensitivity (SES).

Our experimental results show that both the FLP and ILP circuits utilise approximately $3 - 4\times$ more resources than the simplex circuits when $k = 1$. However, the ILP circuits suffer an exponential utilisation increase as k increases. On the other hand, results of FLP circuits are consistently more balanced than the results of the ILP circuits across all metrics we considered as k increases. Finally, fault-injection experiments show that both the ILP and the FLP circuits with $k = 1$ and 2 are approximately $500\times$ less sensitive to CM upsets, which can be further improved by a factor of $1.3\times - 3.4\times$, on average, when these circuits are floorplanned.

This chapter is organised as follows. Sec. 5.2 provides the background information needed to understand the development of TLegUp described in this chapter. Sec. 5.3 discusses the challenges we had to address in order to partition the TLegUp generated designs. Sec 5.4 presents the architecture of the TLegUp toolflow. Sec 5.5 describes the experimental methodology of this work, while Sec. 5.6 presents our results. Finally, the chapter is summarised in Sec. 5.7.

5.2 Background

In this section, we briefly present the architecture of the official LegUp flow, the design methodology for implementing FPGA systems with LegUp, as well as the architecture and RTL hierarchy of LegUp generated designs. This information will help the reader to better understand the architectural details of our TLegUp toolflow.

5.2.1 LegUp high-level synthesis flow

HLS is the procedure of turning a timed (e.g., SystemC) or untimed (e.g., C or C++) high-level algorithm into a cycle-accurate RTL design [33]. Fig. 5.3 illustrates how LegUp

compiles a program that is specified in ANSI C to synthesisable Verilog [24].

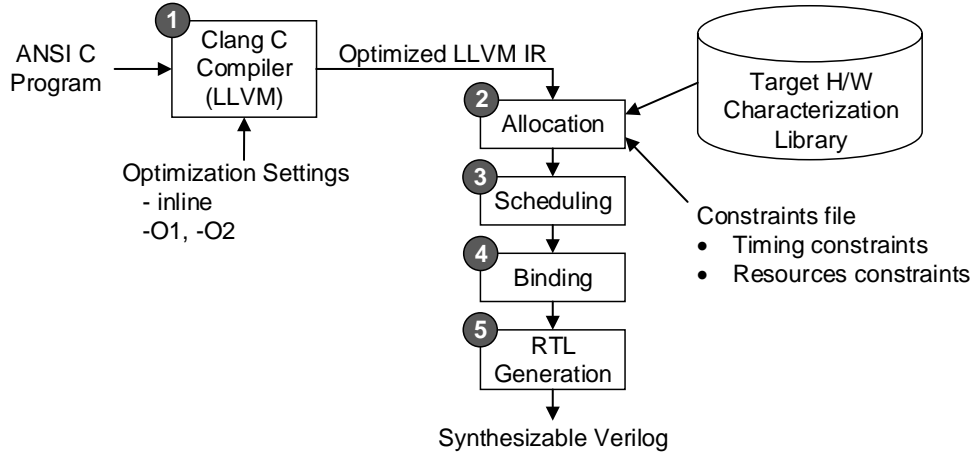


Figure 5.3: LegUp high-level synthesis flow.

At step ① (C compilation), the LLVM Clang C compiler reads an ANSI C program as well as any optimisation settings (e.g., function inlining cost) and produces an optimised LLVM IR.

At step ② (allocation), the optimised IR code, the user constraints file, as well as the characterisation library for the target hardware are analysed in order to define the constraints of the HLS problem. In LegUp the hardware units corresponding to LLVM IR instructions are pre-synthesised for the target FPGA in all supported bit-widths so that the FPGA resources needed to implement a given instruction as well as their associated delay can be determined. This information is stored in the characterisation library shown on the right side of Fig. 5.3 which is used to annotate each LLVM instruction in order to make early circuit speed and area predictions. The constraints file contains several allocation parameters, such as which functional units (e.g., multipliers, dividers etc.) can be shared across multiple IR instructions and what is the intended post-route clock period of the target FPGA circuit. The user can perform design space exploration by modifying the default allocation settings in LegUp. For example, a shorter target clock period will result in a more deeply pipelined circuit.

At step ③ (scheduling), each IR instruction is assigned to a specific clock cycle and this information is used in step ④ (binding) to synthesise an FSM per Vmodule. Depending on data dependencies and on the delay of each instruction, several instructions can be either chained or executed in parallel in the same clock cycle.

At step ④ (binding), IR instructions and IR variables are assigned to hardware units and registers (flip-flops and BRAMs), respectively. Several instructions and variables can be implemented on shared hardware units or registers, which, however, utilise additional multiplexers to realise resource sharing that can be expensive in terms of area and delay. For example, both a 32-bit adder and a 32-bit 2-to-1 multiplexer require 32 4-input LUTs when implemented on an FPGA [24]. Therefore, LegUp commonly avoids sharing hardware units among multiple instructions since the potential area savings are lost in implementing the multiplexers needed to share these resources.

Finally, the IR instructions and the scheduling and binding results are analysed at step ⑤ (RTL generation) in order to generate a suitable FSM and datapath for the output RTL design that meet the user constraints that were specified during allocation at step ②. In short, step ⑤ generates the corresponding RTL description of the HLS C program according to the user constraints and the scheduling and binding results.

5.2.2 LegUp design methodology

LegUp can synthesise an ANSI C HLS program entirely into hardware, or into a hardware/software co-design. Implementing the HLS program entirely into hardware is simple. It only requires compiling the C program to RTL and then implementing the circuit on the FPGA as shown in Fig. 5.4(a).

The hardware/software co-design methodology, as shown in Fig. 5.4(b), is more complex. The user typically profiles the input C HLS program in order to determine which regions of the program will be implemented with custom hardware and which regions of the program will run on a Tiger MIPS soft-processor [24]. In LegUp, profiling is performed at the granularity of C functions by compiling and executing the C program directly on the MIPS processor. Any C function that is chosen for implementation in hardware, as well as any nested C functions within this chosen hardware-based function, are synthesised into separate Vmodules. Finally, LegUp re-compiles and runs the C program on the MIPS processor, while replacing the body of the C functions that are implemented in hardware with simple code that sends and receives data between the hardware accelerators and the MIPS processor.

The current version of TLegUp does not support hardware/software co-design SoC implementations.

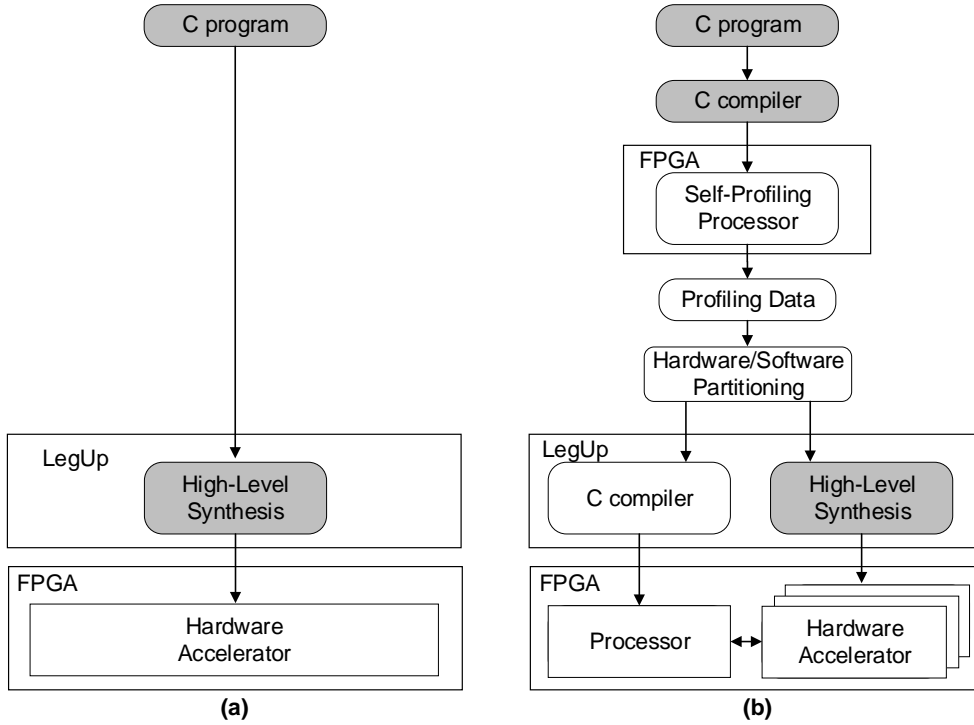


Figure 5.4: LegUp design methodology: (a) Pure hardware implementation; (b) Hardware/software implementation.

5.2.3 Architecture and RTL hierarchy of pure hardware LegUp generated designs

This section presents the architecture and RTL hierarchy of an official LegUp generated design.

LegUp generates a separate Vmodule (Verilog module) for each C function call in the HLS program, except for very small functions that are typically inlined into the root caller function [24]. The threshold for inlining functions is configured by the `-inline-threshold=N` setting of the Clang C compiler, which is set to `N=100` by default.

Nevertheless, a Vmodule consists of a local memory as well as an FSM that controls the Vmodule's datapath, as shown in Fig. 5.5, and has the following IO ports:

- Data ports
 - Input port for providing the stimulus data to the Vmodule.

- `Result` port for receiving the result data from the Vmodule.
- Control ports
 - `Start` port to start the execution of the computation.
 - `Reset` port for resetting the Vmodule if required.
- Status port
 - `Finish` port that indicates that the Vmodule has processed all input data and the result data is ready for collection.
- Memory ports that exist only when a Vmodule needs to connect to a global memory controller of the design
 - `Mem_data` ports that connects the Vmodule’s datapath to the design’s global memory controller.
 - `Mem_address` port that connects the Vmodule’s FSM to the address bus of the global memory controller.
 - `Mem_control` ports that connect the Vmodule’s FSM to the control ports of the global memory controller, e.g., `write_enable` control port etc.

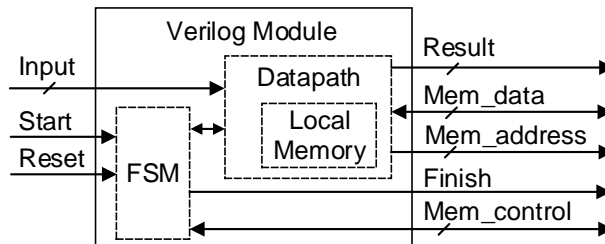


Figure 5.5: Architecture of a Verilog module in LegUp generated designs.

By default, LegUp generates RTL code that has the same hierarchy as given by the compiled call graph of the C application, whereby C functions deeper in the call graph correspond to Vmodules instantiated deeper in the RTL hierarchy [24]. An exception to this rule is the main function that is always instantiated under a `top` Vmodule as shown in Fig. 5.6(b).

LegUp uses the `top` Vmodule in order to instantiate and interconnect the mentioned global memory controller with the main Vmodule so that global variables can be accessed

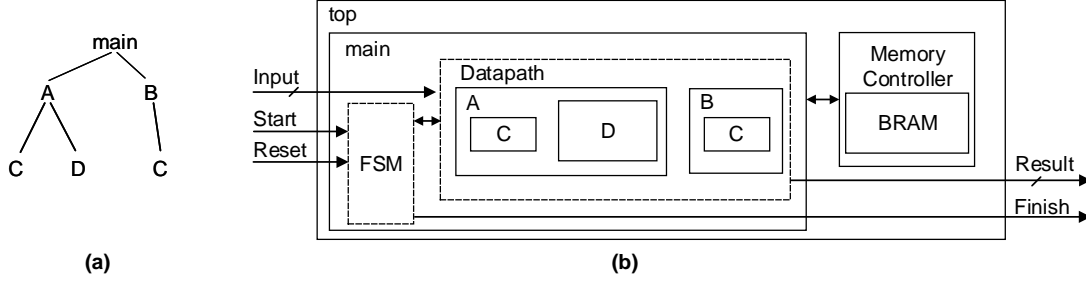


Figure 5.6: (a) Call graph of an HLS C program. (b) Architecture and RTL hierarchy of a pure hardware LegUp generated design.

across multiple Vmodules in the design. For example, if a C program has the program call graph of Fig. 5.6(a), LegUp will generate an RTL design with the hierarchy of Fig. 5.6(b).

Please note that each Vmodule that is instantiated in the main Vmodule of Fig. 5.6(b) has the architecture depicted in Fig. 5.5, in other words, each Vmodule has both its own local FSM and its own local memory.

5.3 Challenges of Partitioning LegUp Generated RTL Designs

As mentioned in Sec. 5.1, we explored ILP and FLP approaches to partitioning the TLegUp generated TMR designs. In the following, we discuss the challenges we encountered when partitioning the designs with ILP and how these challenges were eventually addressed with FLP.

Our goal with the ILP approach was to balance the utilisation of each partition while minimising the interconnections, and thus the number of partitioning voters, required between partitions. Balancing the resource utilisation between the partitions of a TMR circuit firstly increases its reliability and secondly reduces the worst case recovery time of the circuit when MER or FMER is used. Nevertheless, to achieve this goal, we reasoned we should within LegUp create a flattened DFG representing the complete computation of the application before applying a min-cut, max-flow algorithm [67] to form the individual partitions based on the LUT area consumed by the instructions and the data-widths of their operands and results. Building the DFG is straightforward within the LegUp framework once the C code has been translated into the LLVM IR using Clang [24].

However, since LegUp synthesises each C function as a Vmodule with its own data- and control-flow, we decided to inline all functions into the main as an efficient means of building a single DFG representing the entire computation.

A consequence of LegUp’s HLS strategy and our decision to inline all C function calls into main is that LegUp synthesises a single FSM to control the datapath of the entire design as shown in Fig. 5.7(a), which has flow-on effects for the efficacy and scalability of the ILP approach. Namely, the status and control signals for each partition are tightly-coupled with this single, central FSM controller. As the number of partitions used increases, the central FSM becomes relatively larger and more unbalanced, and the distribution of signals between the FSM and the k partitions becomes increasingly complex, congested and slow.

For example, Fig. 5.7(b) illustrates how the design of Fig. 5.7(a) is partitioned with ILP into k partitions. The more partitions the ILP design contains the more interconnection wires and voters are required to connect these partitions to the central FSM. The impact of the central FSM becomes more prominent when the ILP designs are floorplanned. Each of the k datapath partitions is constrained for implementation into a PBlock, which implies that the FPGA CAD tools cannot place the logic of the design’s FSM and the datapath in close proximity, thereby increasing the critical path of the circuit.

Please note that we do not depict the Vmodules of the remaining two modules of the TMR scheme in Figs. 5.7 for clarity reasons. Specifically, the TMR design of Fig. 5.7(a) contains three instances of the main Vmodule and three instances of the memory controller, but we do not depict these in the figure.

A further problem we found with the ILP approach was that although we had calculated resource usage based on implementing instructions with LUTs, many were eventually mapped to DSPs and BRAMs, as they should be for enhanced performance. Our estimates of resource usage based on LUT requirements did not, therefore, assist in balancing the partition sizes.

In order to overcome the signal distribution problem resulting from having a centralised FSM in the ILP design, we explored the FLP approach. Since LegUp synthesises a Vmodule with its own local FSM controller for each C function, we explored forming partitions based on clustering functions together to balance the partition sizes, thereby distributing the single FSM produced for ILP and reducing the number of interconnections between the partitions.

Secondly, to more accurately balance the partition sizes, we use estimates of the number

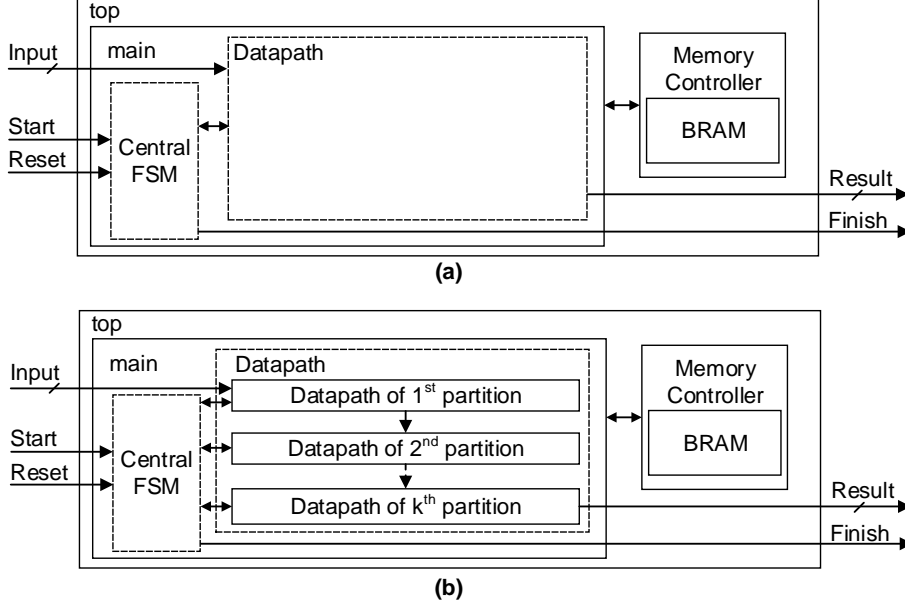


Figure 5.7: (a) Architecture of a TLegUp generated design with all C function calls inlined, (b) Architecture of a TLegUp generated design that is partitioned with ILP.

of CFs needed to implement the functions within each partition since these are more representative of the partition's soft error sensitivity and recovery time. These estimates are obtained by synthesising and mapping the HLS designs on the FPGA after a first pass through the front-end. We estimate the number of CFs needed per function by scaling the number of resources (i.e., number of LUTs, slices, DSPs and BRAMs) used to implement the function by the number of CFs used to implement a column of the corresponding resource type. These estimates are then back-annotated into the hardware library used by TLegUp. On a second pass through TLegUp, the max-flow, min-cut algorithm [67] uses these estimates to balance the resource utilisation of the partitions based on the functions they contain.

Although the FLP approach solved the routing congestion problems that were caused by the central FSM of the ILP designs, the RTL hierarchy of the FLP designs however was not suited for implementing each TMR domain of the designs in a separate Pblock. To be more specific, when floorplanning either an ILP design or an FLP design, each TMR domain needs to be implemented into a separate Pblock in order to 1) avoid resource sharing between the TMR domains, and 2) selectively reconfigure the corresponding Pblock of a faulty TMR domain as described in Sec. 2.3.

In order to understand why the RTL hierarchy of the FLP designs did not assist in properly floorplanning them, it is important to understand the following restriction: *Vivado prohibits any children of a Vmodule to be assigned to a Pblock other than the Pblock of the parent.*

For example, Vivado prevents the user from assigning the Vmodules `A_inst` and `B_inst` of the RTL hierarchy shown in Fig. 5.8(a) to `Pblock_1` and `Pblock_2` of Fig. 5.8(c), respectively, because `B_inst` is a child of `A_inst`.

One can overcome this limitation by modifying the RTL hierarchy of the design to that of Fig. 5.8(b), in which `B_inst` is not a child of `A_inst`. This necessitates:

- Converting signals connecting `A_inst` and `B_inst` in Fig. 5.8(a) to external ports of `A_inst` in Fig. 5.8(b); and
- Connecting `A_inst` and `B_inst` with signals in the `main_inst` of Fig. 5.8(b).

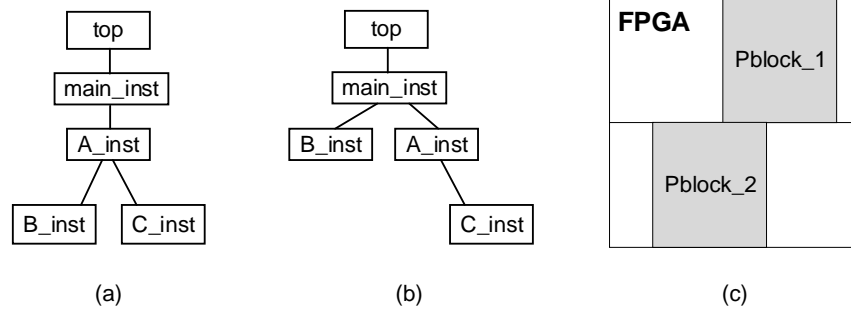


Figure 5.8: (a) RTL hierarchy of a random design, (b) Modifying the RTL hierarchy, (c) FPGA floorplanning layout.

Due to the floorplanning limitation mentioned above, only the ILP designs can be floorplanned without modifying their RTL hierarchy because their TMR domains are implemented with Vmodules at the same level of the hierarchy, for instance, like `A_inst` and `B_inst` in Fig. 5.8(b). In contrast, floorplanning an FLP design may require placing sub-Vmodules in different Pblocks to their parents, which, as mentioned, is prohibited in Vivado. In order to be able to place the Vmodules (corresponding to C functions) of a TMR domain into a single Pblock when floorplanning an FLP design, we flatten their RTL hierarchy. In other words, irrespective of depth, nested functions are instantiated at the same RTL hierarchy level as function main.

The following two examples show how ILP and FLP designs are floorplanned.

Floorplanning an ILP design

An ANSI C HLS program with the C function call graph of Fig. 5.9(a) has the RTL hierarchy of Fig. 5.9(b) when triplicated and partitioned using ILP with $k=2$. The three modules (i.e., TMR domains) of partition 1 are the Vmodules `partition_1A`, `partition_1B` and `partition_1C` shown in Fig. 5.9(b) and Fig. 5.10(a), respectively. Similarly, the second partition consists of Vmodules `partition_2A`, `partition_2B` and `partition_2C`.

ILP inlines all C functions of the main program depicted in Fig. 5.9(a) in order to create a single DFG for the whole computation and then partitions it. This results in the TMR design of Fig. 5.9(b), whereby the Vmodules corresponding to its TMR domains are all instantiated at the same RTL hierarchy level and can thus be floorplanned without modification. The six TMR domains of the ILP design are simply assigned to six separate Pblocks, as shown in Fig. 5.10(b).

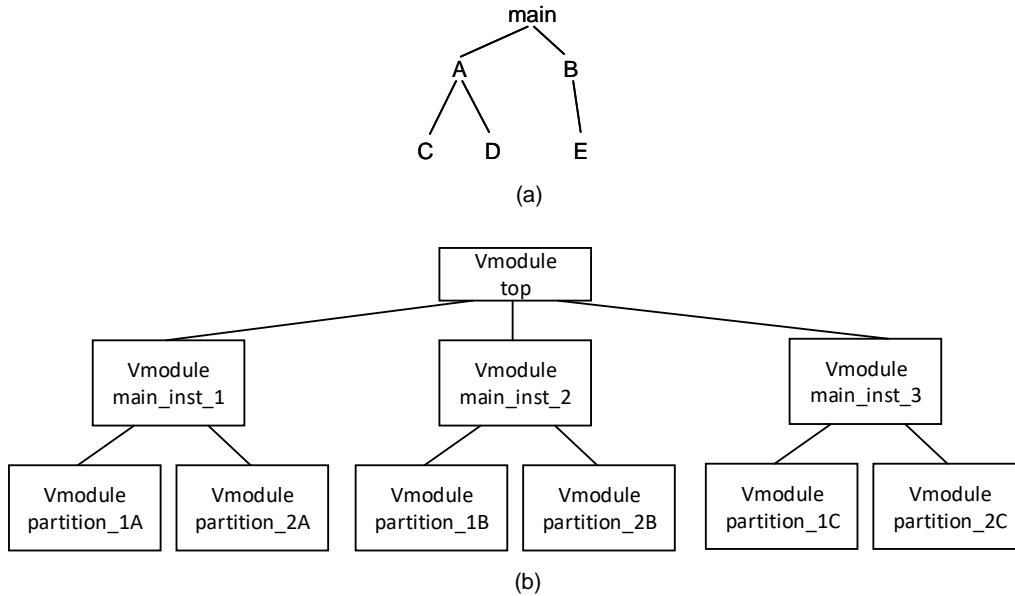
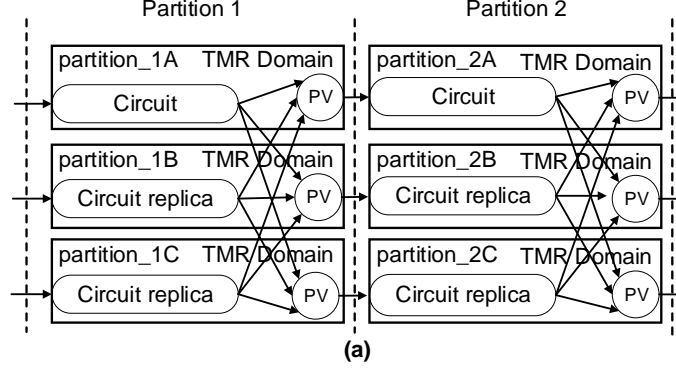


Figure 5.9: (a) Function call graph of an HLS application, (b) TLegUp RTL hierarchy output when the HLS application is partitioned with ILP for $k = 2$.



FPGA	Vmodules of Pblock_1A:	Vmodules of Pblock_2A:
	<ul style="list-style-type: none"> partition_1A 	<ul style="list-style-type: none"> partition_2A
	Vmodules of Pblock_1B:	Vmodules of Pblock_2B:
	<ul style="list-style-type: none"> partition_1B 	<ul style="list-style-type: none"> partition_2B
	Vmodules of Pblock_1C:	Vmodules of Pblock_2C:
	<ul style="list-style-type: none"> partition_1C 	<ul style="list-style-type: none"> partition_2C

(b)

Figure 5.10: Floorplanning an ILP partitioned design: (a) Conceptual floorplanning layout, (b) FPGA floorplanning layout

Floorplanning an FLP design

Let us assume that the C program of Fig. 5.9(a) is partitioned using FLP with the TMR design with the RTL hierarchy of Fig. 5.11. TLegUp creates three copies of `main_inst`, `A_inst` and `C_inst`, to implement the three TMR domains of the first partition (i.e., TMR component) and three copies of `B_inst`, `D_inst` and `E_inst` to implement the three TMR domains of the second partition. Please note that in Fig. 5.11 we do not depict the child Vmodules of `main_inst_2` (TMR domain B) and `main_inst_3` (TMR domain C) for the sake of clarity.

TLegUp firstly flattens the RTL hierarchy of the design as shown in Fig. 5.12 and then floorplans the design as shown in Fig. 5.13. As Fig. 5.13 depicts, the first, the second and the third TMR domains of Partition 1 are implemented in `PBlock_A1`, `PBlock_B1`, and `PBlock_C1`, respectively, while the first, second and third TMR do-

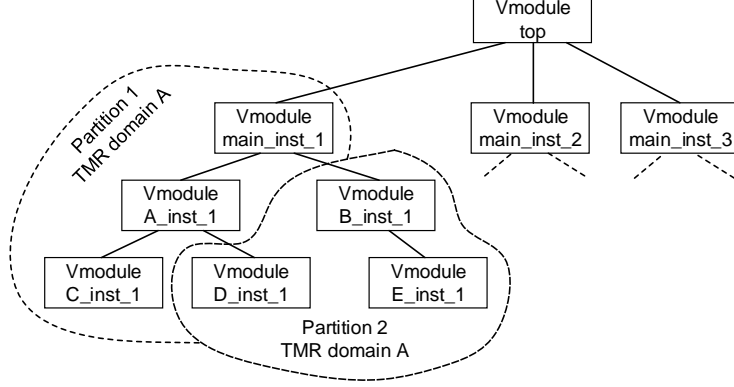


Figure 5.11: An FLP partitioned design with non-flattened RTL hierarchy.

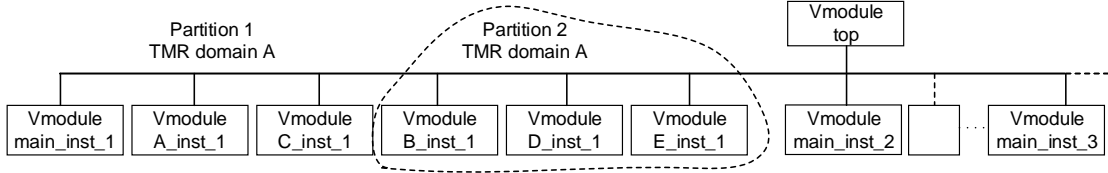


Figure 5.12: An FLP partitioned design with flattened RTL hierarchy.

mains of Partition 2 are implemented into PBlock_A2, PBlock_B2, and PBlock_C2, respectively.

In TLegUp, Vmodule flattening is performed in the LLVM IR before RTL code generation by identifying those signals that cross function boundaries and adding ports to the function’s interface if these are not already present.

Needless to say, the relative sizes of the specified functions impact the balancing of the partition sizes. TLegUp provides feedback to the user regarding the number of resources and the estimated number of frames used to implement each function. Together with feedback on the partitioning of the functions derived by the system, the total bitwidth of the signals crossing the partitions, and the number of partitioning and synchronisation voters added to each partition, we believe the designer has the necessary information to re-factor the C specification to achieve a better resource balance if desired.

Finally, both the ILP and FLP designs experience a further drawback similar to that caused by the central FSM in the ILP designs. All partitions that contain code that uses the shared global memory need to connect with it. This feature of the framework can,

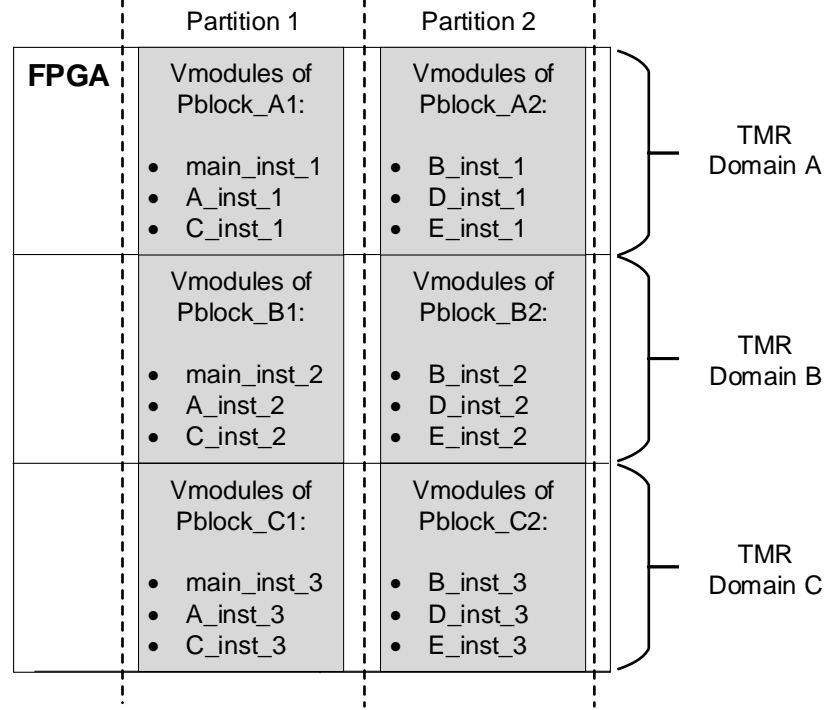


Figure 5.13: Floorplanning an FLP partitioned design.

therefore, lead to a proliferation of wires between the partitions and the shared memory block when we try to adapt it to automatically produce TMR designs. This is primarily a drawback of the approach taken by LegUp to provide a global memory to the Vmodules of the design. But it becomes more prominent when the designs are triplicated and even worse when triplicated and floorplanned. For example, Fig. 5.14 illustrates the required multiplexing that is instantiated in the LegUp generated design of Fig. 5.6(b) in order to share the global memory between all Vmodules of the design. The deeper the instantiation of a Vmodule in the RTL hierarchy the more multiplexing required to reach the memory controller which can considerably reduce the maximum operating frequency of the circuit.

5.4 Architecture of the TLegUp Toolflow

Fig. 5.15 illustrates the architecture of the TLegUp toolflow, which is divided into two parts, a front- and a back-end part. The front-end of TLegUp involves high-level synthesis of the C program as done in the official LegUp flow, as well as three additional

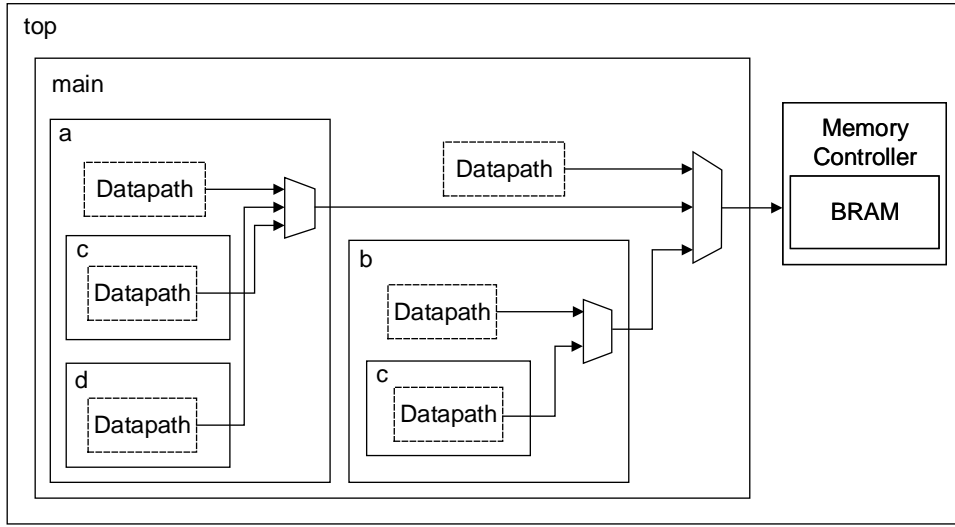


Figure 5.14: Memory hierarchy of the LegUp generated design shown in Fig. 5.6(b).

tasks: 1) design triplication, 2) design partitioning and 3) voter insertion. The RTL is then implemented on an Artix-7 200T FPGA with the back-end, which involves netlist synthesis, technology mapping, optional floorplanning, placement, routing, and bitstream generation. The following sections provide more detail about the architecture and implementation of both parts in TLegUp.

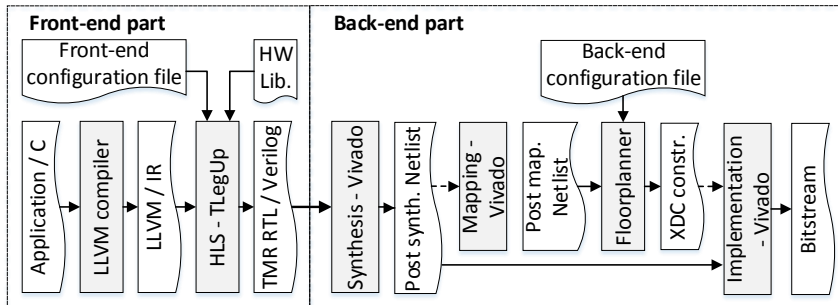


Figure 5.15: The TLegUp toolflow.

However, before providing the details of our proposed TLegUp flow, it is helpful to illustrate the structure of the TegUp generated designs as well as the locations in which TLegUp inserts voters. At the top level, as depicted in part 1) of Fig. 5.16, three copies of the partitioned design are implemented. Reducing voters (RV) are inserted to convert each triplicated output of the circuit to a single signal. Each circuit replica consists of a partitioned circuit as illustrated in part 2). Each partition corresponds to a TMR compo-

nent that has three functionally identical modules (TMR domains). Partition voters (PV) are inserted on the partition boundaries for each exiting signal. In order to re-synchronise the state between TMR domains and to prevent error proliferation, synchronisation voters (SV) are inserted into every datapath cycle in part 3). These are of course also needed within the next state logic of FSMs, as illustrated in part 4). Finally, TLegUp inserts PVs on the output of the global memory controller as depicted in part 5).

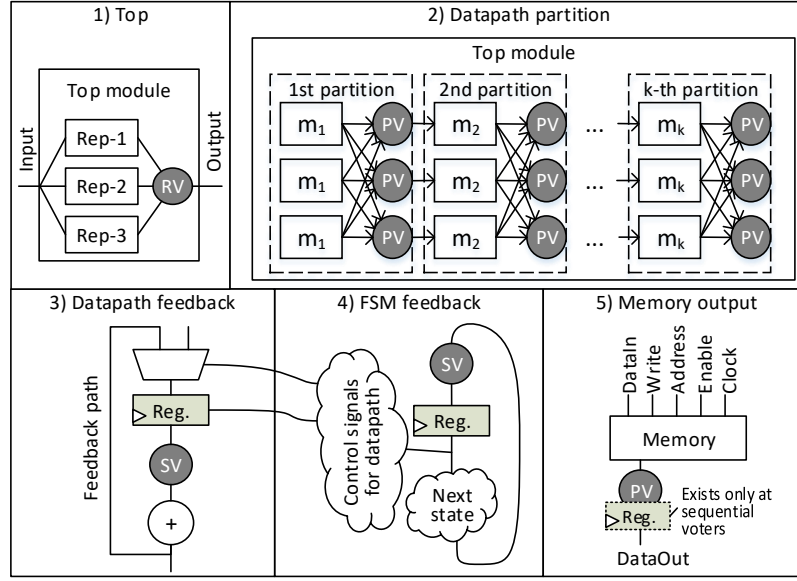


Figure 5.16: TLegUp generated TMR code and voter locations.

TLegUp supports the generation of TMR circuits that can be either repaired with periodic CM scrubbing, MER or FMER. In case of TMR circuits that incorporate MER or FMER, TLegUp automatically instantiates voters with a 2-bit minority report output as shown in Fig 2.8 of Sec. 2.3.2. Additionally, the user can instruct TLegUp to insert registers to the PVs of the global memory controller in part 5) in order to decrease the critical path between the memory controller and the Vmodules, especially for those Vmodule that are instantiated deeper in the RTL hierarchy (e.g., as shown in Fig. 5.14). Finally, it should be noted that TLegUp does not triplicate the clock and reset signals in the generated TMR RTL designs.

5.4.1 Front-end

TLegUp uses two different front-end versions to implement the ILP and the FLP approach, which have a similar architecture. In this section, we present the architecture of the ILP front-end and show the extensions we made in order to realise the FLP front-end.

5.4.1.1 Instruction-Level Partitioning

Fig. 5.17 illustrates the design flow of the ILP front-end in TLegUp.

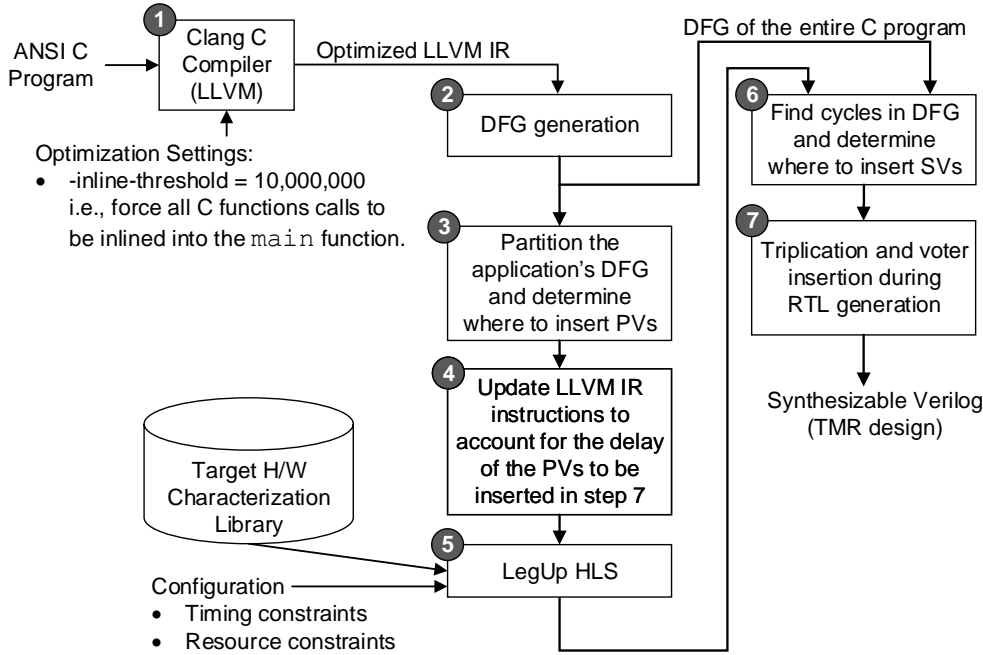


Figure 5.17: The architecture of the TLegUp ILP front-end.

At step ①, the LLVM Clang compiler reads an ANSI C program and inlines all C function calls of the program in order to produce an optimised LLVM Intermediate Representation (IR) of the whole computation. The LLVM Clang compiler inlines all C function calls when the `-inline-threshold` cost of functions is set to an extremely large value (e.g., `-inline-threshold=10,000,000`). In other words, it is almost impossible for any of the HLS application's C functions to have a cost larger than 10,000,000, which therefore forces all C function calls to be inlined during compilation.

At step ②, the LLVM IR is converted into a DFG. Nodes in the DFG represent IR instructions, edges represent data dependencies between nodes, while edge weights represent the bit-widths of the LLVM instruction operands and results. In more detail, each node in the DFG is a data structure that stores an IR instruction as well as a Boolean flag to indicate that the specific IR instruction should be voted on at step ⑦.

At step ③, a modified version of the max-flow, min-cut algorithm presented in [67] is used

to partition the DFG of the C program into k sub-graphs, while minimising both the total number of interconnection nets between partition boundaries and the standard deviation between the average utilised LUTs of all k partitions. The outputs of this step are: 1) a list of LLVM instructions per partition that do not overlap, and 2) a list of instructions per partition that would be voted on in the final RTL design, i.e., at step ⑦.

At step ④, the delay of each LLVM IR that will be voted on at step ⑦ is modified so that these instructions account also for the delay of the voters. This is very useful when scheduling the LLVM IR instructions at step ⑤. It enables the HLS tools to minimise the negative timing effects of inserting partitioning voters (PVs) into the design during scheduling and binding. In more detail, the LegUp HLS tools try to achieve the targeted clock period (given in the configuration file as shown in Fig. 5.17) by pipelining the design, i.e., by inserting registers into critical paths.

At step ⑤, allocation, scheduling, and binding are performed on the updated list of LLVM IR instructions using LegUp HLS as described in Sec. 5.2.1 for steps 2 – 4 of Fig. 5.3.

At step ⑥, a Depth First Search (DFS) algorithm [35] is used to find all cycles (feedback paths) in the application’s DFG. The output of the DFS algorithm is a list of cycles, where each cycle is represented as a sequence of LLVM IR instructions. The list of cycles, as well as the scheduling table (created at step ⑤) is analysed, and synchronisation voters (SVs) are inserted into each cycle of the application’s DFG. In order to minimise the performance penalty caused by inserting SVs in the design, we first find the delay of chained instructions between all pairs of registers in the LLVM IR and then insert the SVs into those paths with the most slack. For example, Fig. 5.18(a) illustrates three LLVM IR instructions in the DFG that form a registered loop. The scheduling table of these three instructions is shown in Fig. 5.18(b), while the datapath of the RTL design that would have been generated if we used the unmodified LegUp flow is shown in Fig. 5.19(a). Assuming that each instruction shown in Fig. 5.19(a) has a delay of 5 ns, the logic “add” between the output of R1 and the input of R2 registers will have a 5 ns delay, while the logic “sub+phi” between the output of R2 and the input of R1 will have $5\text{ ns} + 5\text{ ns} = 10\text{ ns}$ delay. Assuming that the delay of an SV is 3 ns, the maximum operating frequency of the circuit will not be affected when an SV is inserted at the output of R1, since the overall delay of the “add” logic and the “SV” logic is 7 ns. This is less than the delay of the “sub” logic and the “phi” logic that sums to 10 ns. TLegUp creates a graph for each cycle in the DFG, where nodes represent registers and edges the delay between registers as shown in Fig. 5.19(b). It then iterates through all nodes in the graph in order to find and insert an SV on the edge with the least delay.

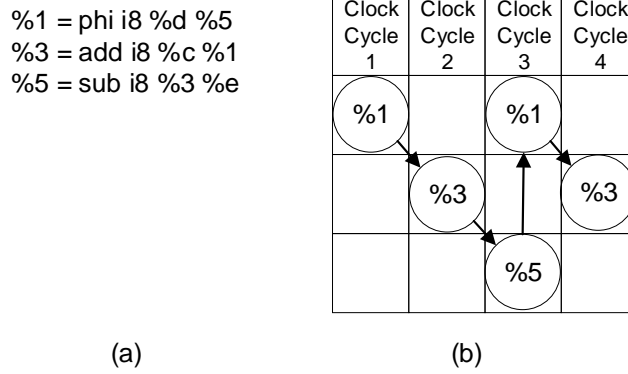


Figure 5.18: (a) A sequence of IR instructions forming a registered loop, (b) Scheduling table of the IR instructions.

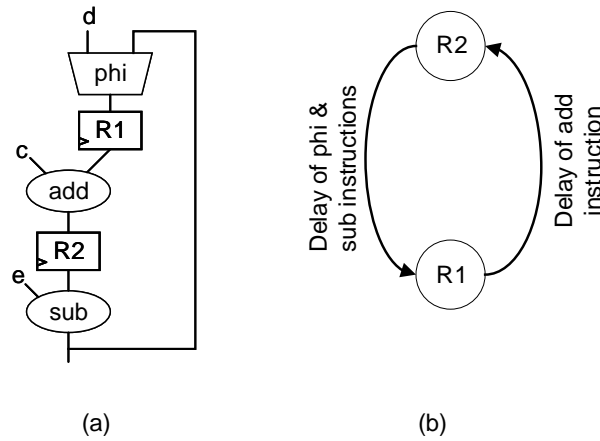


Figure 5.19: (a) Datapath of the IR instructions, (b) Graph for capturing the clock slack of instructions between registers in a cycle.

Finally, at step ⑦, Verilog code for the TMR design is generated. The RTL generation step is seamlessly coupled with triplicating the logic of the RTL design and inserting the voter logic. Because the RTL generation is a simple translation step that converts LLVM IR instructions into Verilog RTL code, we do not triplicate the LLVM IR instructions in previous steps. Instead, we translate each LLVM instruction into RTL code, replicate this RTL code two more times and insert RTL code for voters and interconnection signals if the output of the LLVM instructions was identified as a voter insertion location at steps ③ or ⑥. Triplication is postponed until step ⑦ because all three TMR modules must be

identical in their datapaths, FSMs and voter locations.

5.4.1.2 Function-Level Partitioning

This section presents how the TLegUp ILP front-end has been extended in order to enable FLP, which, as discussed earlier forms partitions by clustering the C function calls (Vmodules in RTL) of the HLS application.

Fig. 5.20 illustrates the FLP design flow, which in contrast to the ILP flow shown in Fig. 5.17 incorporates the following two additional steps:

- Step ⑧: A preliminary RTL synthesis of the design is executed in order to obtain an accurate estimation of the resource utilisation per Vmodule.
- Step ⑨: The RTL hierarchy of the final FLP partitioned design is flattened in order to be properly floorplanned as described in Sec. 5.3.

In more detail, the FLP design flow can be coarsely divided into two parts as shown in Fig. 5.20. In the first part of the flow, steps ①, ②, ⑤, ⑥, ⑦, and ⑧, TLegUp compiles the HLS C application into a non-partitioned TMR RTL design and synthesises it with the Vivado design suite in order to extract the resource utilisation per Vmodule (i.e., C function call in the HLS application) as well as the total interconnection signals between each pair of Vmodules. This information is stored in the characterisation library shown in the bottom-left of Fig. 5.17 and is used in the second part of the flow, namely, steps ③, ⑦, and ⑨, in order to partition the design with the FLP approach. In particular, the second part of the FLP flow uses the resource utilisation per VModule and the total number of interconnections between each pair of Vmodules in the design in order to apply function-level partitioning at step ③, RTL hierarchy flattening at step ⑨, and finally triplicate and generate the RTL of the final design during the second invocation of step ⑦.

Steps ② and ③ in the FLP flow are also modified as follows:

- At step ② of the FLP flow two different kinds of DFGs are generated: 1) an instruction-level DFG, as used in the ILP design flow, in order to find and insert SVs into the design, and 2) a function-level DFG that is used for partitioning the design. In the function-level DFG, nodes represent the corresponding Vmodules of C function calls in the HLS application, while edges represent the data dependencies

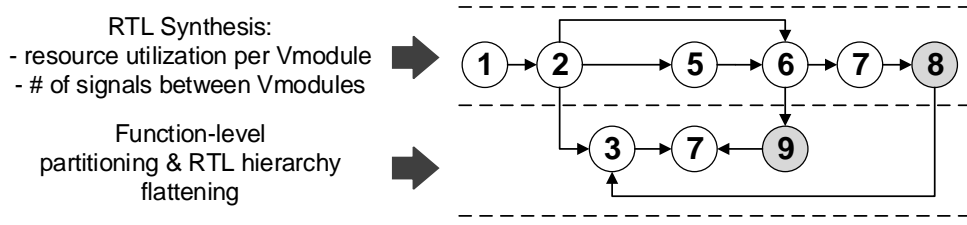


Figure 5.20: The FLP design flow of TLegUp.

between nodes. Specifically, each edge is simplified with a directed edge, with the edge weight being the sum of the bit-widths of interconnections between Vmodules. Please note that at step ② of the ILP flow, only the instruction-level DFG of the HLS application is generated and used for both SV and PV insertion.

- Like the ILP design flow, the same network-flow algorithm is used for the FLP design partitioning at step ③, but the algorithm is modified in a way that balances the number of configuration frames (CFs) between Vmodules instead of the number of LUTs. By using the number of CFs instead of the number of LUTs, we can balance the area between the partitions more accurately, and also estimate the reconfiguration time of each TMR domain.

5.4.2 Back-end

The back-end part of our TLegUp flow incorporates the Xilinx Vivado 2017.2 design suite in order to synthesise and implement the generated TMR RTL designs on a Xilinx Artix-7 200T FPGA as well as an academic floorplanner [100] to optionally floorplan the designs. A flag to enable the floorplanning of a design is included in the back-end configuration file shown in Figs. 5.15 and 5.21.

As mentioned, each TMR domain in the ILP design is described with a single Vmodule, while a TMR domain in the FLP design is described though a list of Vmodules. The names of Vmodules corresponding to each TMR domain of the design, as well as the total bitwidth of interconnections between each pair of TMR domains, are reported by the front-end and stored in the back-end configuration file.

Fig. 5.21 illustrates how the back-end part of the TLegUp toolflow automatically floorplans a TMR design. At step ①, the floorplanner analyses the post-mapping netlist of the design

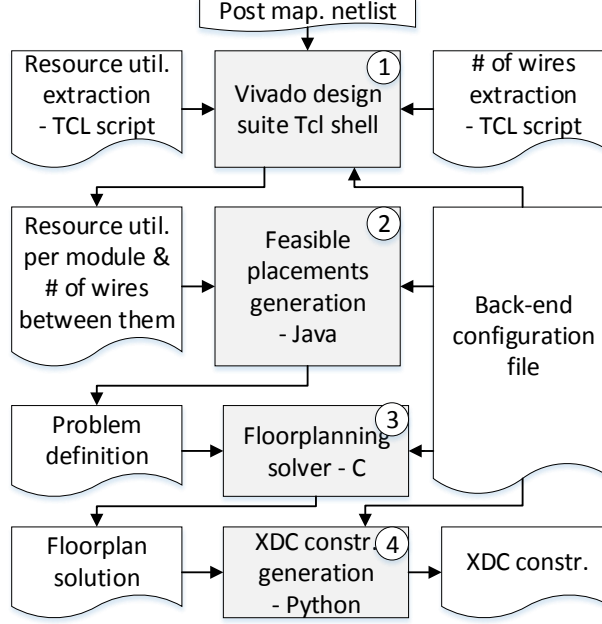


Figure 5.21: The architecture of the floorplanning flow.

and extracts the required number of slices, DSPs and BRAMs for each TMR domain of the design as reported in the back-end configuration file.

At step ②, the floorplanner generates a list of all possible Pblocks that can be created on the fabric of an Artix-7 200T FPGA. Note that the Pblocks span one or more complete configuration rows and are aligned to slice, DSP and BRAM columns so that partial reconfiguration can easily and effectively be applied [100]. Subsequently, a set of feasible Pblocks is assigned to each TMR domain of the design. The selection of Pblocks for the TMR domains is determined by the availability of resources a Pblock has for each TMR domain. For example, in our experiments, we only assign a Pblock to a TMR domain if its utilisation ranges between 40% and 87% (i.e., similar to the default values of Vivado’s 2017.4 automatic floorplanner), on average, for all resource types (CLBs, DSPs, and BRAMs) we consider.

At step ③, the floorplanning solver [100] is used to find a near-optimal floorplanning solution for the design. The solver uses a genetic algorithm to select a suitable Pblock from the set of Pblocks that were assigned to each TMR domain in step ② in a way that reduces the overall wire-length and resource utilisation of the final circuit.

Finally, at step ④ the solution from the solver is translated into XDC constraints so that all TMR domains are placed and routed into specific Pblocks of the FPGA during implementation.

5.5 Experimental Methodology

5.5.1 Resource utilisation, performance, resource balance and implementation time

We used LegUp to produce simplex (non-triplicated) designs and TLegUp to produce triplicated designs of the 17 CHstone, DWARV and Bambu HLS benchmark applications [80]. For each benchmark, we generated the following designs:

- Two simplex versions: we created one simplex version in which we prevented LegUp from inlining any functions (Simplex/NIF) and a second version in which all functions were inlined (Simplex/IF); and
- Eight TMR versions: we created four TMR designs using FLP with $k = 1, 2, 4$ and 8 partitions, and four TMR designs using ILP, also with $k = 1, 2, 4$ and 8 partitions. We refer to these sets of designs as TMR/FLP and TMR/ILP, respectively. Both the ILP and the FLP designs were implemented on the FPGA with and without floorplanning them, which we denote with the acronyms FL and NFL, respectively.

The above naming convention is summarised in Table 5.1.

In our results, we compare the utilisation, performance, soft error sensitivity (SES) and implementation time of the TMR/ILP circuits against the Simplex/IF circuits as the degree of partitioning, k , is varied. Similarly, we compare the results for the TMR/FLP circuits with those for the Simplex/NIF circuits as k is varied. Last, we compare the resource balance achieved by TMR/FLP with that achieved by TMR/ILP as k is varied.

We used the following metrics in order to evaluate and compare the quality of all circuits:

- number of utilised Slices (SLI), BRAM16 (BM) and DSPs;
- maximum operating frequency (FM) given in MHz;

Table 5.1: Naming convention used to describe the designs that were implemented on the Artix-7 FPGA.

Name	Description
Simplex/NIF	Simplex designs with none of the application's C functions inlined.
Simplex/IF	Simplex designs with all of the application's C functions inlined.
TMR/FLP	TMR designs with none of the application's C functions inlined that were partitioned with the FLP approach.
TMR/ILP	TMR designs with all of the application's C functions inlined that were partitioned with the ILP approach.

- latency (LT) given in number of required clock cycles to obtain the result from the application; and
- execution time (ET), that is $ET = \frac{LT}{FM}$, given in μs .

Note that LT was measured via functional simulation of the designs, while resource utilisation and FM were extracted from post-routed circuits. The FM for all designs was derived from the critical path slack for an FM target of 66.7 MHz.

Additionally, for the TMR circuits we report:

- The total number of instantiated synchronisation (SV) and partitioning (PV) voters in the designs;
- The coefficient of variation (CV), i.e., standard deviation / arithmetic mean, of the utilised LUTs and the number of CFs for the circuit's partitions to assess how well FLP and ILP algorithms balance resources between partitions. Smaller values of CV imply more resource-balanced partitions.
- The implementation time (IT) of each design which is given in minutes. The IT consists of the time it takes to generate the RTL TMR designs with our front-end plus the time it takes to implement the designs on the FPGA with our back-end.

5.5.2 Estimating the soft error sensitivity

In order to evaluate the SES of all circuits, we re-implemented the designs, but this time incorporating a MicroBlaze (MB) soft-processor to perform fault injection experiments. Fault-injection testing involves randomly flipping an essential bit (EB) of the Design Under Test's (DUT's) CM and subsequently checking whether the DUT still functions correctly. Please note that each DUT contains test vectors stored in BRAM.

The MB was programmed to flip a random EB using the ICAP, to test the functionality of the circuit by setting the start bit and comparing the result with a golden reference value stored in the MB. After a test, the injected fault was reversed and the circuit was reset prior to the next test. We injected faults into a random 15% of the EBs of each DUT in order to speed up the fault-injection campaign [106], and counted as functional errors those tests for which the circuit's output did not match the golden result or for which the finish signal was not asserted after the expected ET had elapsed. We report the SES of each DUT, which is calculated as the number of functional errors observed divided by the number of injected bits. Needless to say, we did not inject faults into the CM of the MB, since it forms part of the test harness rather than the DUT.

5.5.3 Configuration of the tool flow

In order to prevent the inlining of C function calls for Simplex/NIF and TMR/FLP, we set the inlining cost to 10,000,000, while it was set to 0 to force inlining of C functions for Simplex/IF and TMR/ILP. We synthesised the RTL designs by setting Vivado's synthesizer to target "design performance" as this is the default configuration for LegUp. We also disabled all resource sharing options in Vivado to prevent the tools from optimising redundant logic. We used the default settings for the the genetic algorithm of the floor-planner, which was run for 30 minutes to floorplan the design, while the mutation and crossover probabilities were set to 0.15 and 0.85 respectively. We used a server with two Intel Xeon X5660 CPUs (48 cores running at 2.80GHz) and 64GB of RAM to run our flow on Ubuntu 64-bit 16.04 LTS, but we assigned only one core for the implementation of each circuit in order to be able to compile 48 designs in parallel at a time.

5.6 Experimental Results

5.6.1 Simplex circuits

In Table 5.2, we report the resource utilisation, performance, implementation time (IT), essential bits (EB) and SES of simplex circuits with non-inlined functions (NIF) and with inlined functions (IF), while at the bottom of the table we report the geometric mean (Gmean) of the results for all circuits excluding *jpeg* for which the C Clang compiler failed to inline the application’s C function calls.

When comparing the NIF results with those for IF, there is a noticeable difference in FM (31% higher), LT (69% higher) and ET (28% higher than for IF). The shorter mean execution time for the IF circuits is primarily due to a significantly lower latency, despite a lower circuit speed. We believe this reduction in latency is primarily due to the increased parallelism present when all functions are inlined in the DFG.

While there are some significant individual differences between the NIF and IF designs, apart from the difference in performance between the two design types, there is no significant difference in mean utilisation, soft error sensitivity or implementation time.

All circuits required on average 6 minutes to implement, i.e., from C to bitstream.

5.6.2 TMR circuits

Table 5.3 reports the implementation time (IT) – from C to bitstream – of the TMR/FLP and TMR/ILP circuits that were partitioned into $k = 1, 2, 4$ and 8 TMR components and were either not floorplanned (NFL) prior to implementation, or were floorplanned (FL) before being implemented. Designs that could not be partitioned into $k \geq 4$ with FLP, due to an insufficient number of C functions, are reported with the letter *C* in the table. Similarly, circuits that failed implementation, due to routing congestion or unavailability of resources are reported with the letters *R* and *L*, respectively. As mentioned, *jpeg* was not implemented because the Clang C compiler fails to inline all of its C functions, which is reported with the letter *I*.

We found that although ILP was able to synthesise designs for any k , many of these designs failed routing when floorplanned due to routing congestion. On the other hand, not all values of k could be explored with FLP due to a lack of C functions in their HLS

Table 5.2: Resource utilisation, performance, SES and implementation time of the simplex circuits.

DUT	Simplex / NIF: Simplex circuits with non-inlined functions										Simplex / IF: Simplex circuits with inlined functions									
	SLI	BM	DSP	FM (MHz)	LT	ET (us)	IT (min.)	EB (Mbit)	SES	SLI	BM	DSP	FM (MHz)	LT	ET (us)	IT (min.)	EB (Mbit)	SES		
adpcm	2,446	28	54	77	14,575	189	6	1.61	0.099	2,840	4	81	56	5,267	94	11	1.78	0.089		
aes	1,564	4	-	82	1,740	21	6	0.82	0.062	1,488	8	-	13	1,109	82	6	0.87	0.059		
aesdec	1,145	4	-	94	2,889	31	5	0.68	0.060	1,562	8	3	11	2,207	207	6	0.95	0.056		
bell	238	2	-	98	671	7	4	0.14	0.051	564	2	-	95	701	7	4	0.35	0.047		
blowf	2,574	21	-	54	177,164	3,264	7	1.35	0.154	9,422	17	-	68	121,915	1,791	14	4.95	0.134		
dfadd	1,983	6	-	86	3,346	39	6	1.19	0.029	1,070	-	-	98	502	5	4	0.62	0.028		
dfdiv	6,367	6	32	81	3,573	44	12	3.58	0.018	5,252	-	24	72	1,895	26	9	2.79	0.016		
dfmul	1,371	6	16	100	1,329	13	5	0.85	0.025	591	-	16	98	198	2	4	0.38	0.026		
dfsfn	11,892	10	51	73	127,622	1,748	15	6.53	0.057	8,351	4	59	71	47,172	668	14	4.77	0.055		
gsm	2,244	7	59	80	8,364	105	6	1.30	0.036	1,559	6	51	82	4,764	58	6	0.95	0.039		
jpeg	6,233	48	51	72	1,152,167	16,013	13	NA	0.045	537	4	4	79	5,036	63	4	0.33	0.045		
mips	539	4	4	80	5,101	64	5	0.35	0.060	131	4	4	132	10,044	76	3	0.08	0.060		
mmult	153	4	5	134	11,644	87	4	0.09	0.016	3,092	2	1	85	6,173	72	7	1.74	0.012		
nmult	2,151	8	-	82	6,514	80	6	1.21	0.042	455	1	-	71	137	2	4	0.28	0.042		
said	535	4	-	70	147	2	4	0.32	0.149	2,661	10	-	85	206,321	2,423	7	1.57	0.151		
sha	1,707	10	-	98	207,411	2,125	5	1.02	0.053	271	128	2	73	2,861,104	39,409	4	0.28	0.054		
sobel	322	128	2	75	2,861,109	38,104	4	0.29	0.049	1,340	6	10	64	5,681	89	6	0.82	0.047		
Gmean*	1,298	8	15	84	9,574	114	6	0.78	0.049	1,340	6	10	64	5,681	89	6	0.82	0.047		

*The results of jpeg are excluded from the Geometric mean (Gmean)

Table 5.3: Implementation time for the TMR circuits.

	<i>k</i>	adpcm	aes	aesdec	bell	blowf	dfadd	dfdiv	dfmul	dfsfn	gsm	jpeg	mips	mmult	motion	said	sha	sobel	Implementation time (min.)
FLP	NFL	1	21	10	5	15	19	78	14	L	14	54	7	5	20	8	15	7	
	FL	2	21	11	5	18	20	39	14	L	18	64	7	5	21	8	16	6	
ILP	NFL	4	20	11	C	19	21	55	16	L	18	77	C	C	12	C	15	C	
	FL	8	C	C	C	C	23	C	C	L	C	61	C	C	14	C	C	C	
Implementation time (min.)	NFL	1	57	48	38	58	62	76	52	L	53	745	38	35	58	41	59	41	
	FL	2	72	46	38	59	61	125	118	L	62	785	37	37	98	40	57	38	
Implementation time (min.)	NFL	4	203	46	C	63	82	130	91	L	214	1,812	C	C	181	C	59	C	
	FL	8	C	C	C	583	8	C	C	L	C	2,815	C	C	91	C	C	C	
Implementation time (min.)	NFL	1	39	14	6	L	8	24	6	L	16	-	12	4	19	6	60	8	
	FL	2	52	15	8	L	11	27	8	L	30	-	11	5	48	8	57	8	
Implementation time (min.)	NFL	4	37	17	8	L	12	34	8	L	31	-	15	5	87	11	85	9	
	FL	8	105	135	10	L	14	48	19	L	156	-	16	5	86	19	74	9	
Implementation time (min.)	NFL	1	78	117	41	L	38	63	39	L	57	-	41	36	106	40	582	41	
	FL	2	1,545	137	48	L	62	114	44	L	80	-	47	37	3,679	44	R	42	
Implementation time (min.)	NFL	4	R	101	87	L	64	R	49	L	273	-	91	47	R	67	R	56	
	FL	8	R	R	126	L	149	R	R	L	R	-	123	51	R	R	R	R	

* L: LegUp failed to inline functions
* R: Implementation failed due to Routing congestion
* L: Insufficient # of LUTs to implement this design on the Artix-7 200T
* C: Insufficient # of C functions to create the required *k* partitions

specification, but all feasible FLP designs were successfully routed since they were less congested than the ILP designs.

We found our flow was able to implement 90% of completed TMR designs in 123 minutes or less, which, while it represents a significant increase from the time to implement the simplex designs, we believe to be relatively low when compared to the time needed to handcraft implementations. We found that the times to implement the non-floorplanned results were fairly constant and similar between the FLP and ILP designs, although the ILP results for *adpcm*, *gsm*, *motion* and *sha* are noticeably higher than for FLP. Floorplanning appeared to lead to more variability in the results, particularly as we pushed up against resource constraints. Only *jpeg* required excessive time to finish because we targeted $FM = 66.7$ MHz, an unrealistic target for Vivado’s router. For example, *jpeg* finishes implementation in 1 hour, when the FM target is set to 10 MHz.

Note that we do not report averaged implementation times since the lack of results as k increases provides insufficient data for meaningful comparison.

5.6.3 Resource balancing

Table 5.4 reports on the coefficient of variation (smaller is better) in the LUTs and CFs utilisation for the $k = 1, 2, 4$ and 8 partitions generated with FLP and ILP. The Gmeans are calculated over those entries for which we have a result for both FLP and ILP, e.g., for $k = 8$, we have only included the *dfadd* and *motion* results in our calculations. The results show that the FLP designs balance the number of LUTs better than the ILP designs across all values of k . This happens for the following two reasons: 1) although many LLVM instructions in the ILP designs are expected to be implemented with LUTs in the final circuit, they are eventually implemented using different resources, such as DSPs and BRAMs, which leads to poor resource estimates, and 2) the single FSM of an ILP design is generated only after the design is partitioned, which depending on the HLS design, such as whether or not it is control oriented, may be much smaller or larger than the partitions of the design.

Contrary to our goal, the FLP designs are worse in terms of CF balance than the ILP designs. We believe that these unexpected results are observed for the following reason. The FLP approach triplicates and implements a design before partitioning in order to obtain from post-routing results the resource utilisation per Vmodule and uses the result to estimate the CFs per Vmodule. However, after partitioning the design, Vivado packs and maps differently because the characteristics of the design have been altered. For example,

the RTL hierarchy of the design is modified, many Vmodules incorporate additional ports, and partitioning voters have been inserted.

Table 5.4: Coefficients of variation (standard deviation/average) of resource balance (LUTs and CFs) between partitions.

DUT	FLP						ILP					
	$k=2$		$k=4$		$k=8$		$k=2$		$k=4$		$k=8$	
	LUT	CF	LUT	CF	LUT	CF	LUT	CF	LUT	CF	LUT	CF
adpcm	0.13	0.44	0.70	0.66	-	-	0.54	0.36	1.20	0.26	1.88	0.32
aes	0.53	0.04	0.85	0.28	-	-	0.49	0.00	1.22	0.12	2.05	0.38
aesdec	0.59	0.34	0.96	0.36	-	-	0.67	0.20	1.00	0.95	2.06	0.76
bell	1.01	0.61	-	-	-	-	0.69	0.02	1.19	0.28	2.18	0.59
blowf.	0.11	0.32	0.79	0.40	-	-	-	-	-	-	-	-
dfadd	0.37	0.02	1.20	0.24	2.05	0.20	0.82	0.28	1.27	0.39	2.05	0.31
dfdiv	0.41	0.58	0.73	0.96	-	-	0.07	0.97	1.31	1.21	2.13	1.73
dfmul	0.65	0.05	1.16	0.37	-	-	0.73	0.30	1.27	0.44	2.11	0.66
gsm	0.22	0.09	0.50	0.50	-	-	0.32	0.10	0.83	0.19	1.89	0.44
jpeg	0.33	0.17	0.70	0.11	1.04	0.15	-	-	-	-	-	-
mips	0.02	1.03	-	-	-	-	0.87	0.86	1.56	0.59	2.05	0.70
mmult	0.72	0.08	-	-	-	-	1.03	0.42	1.42	0.29	1.91	0.44
motion	0.51	0.02	1.09	0.22	1.63	0.46	0.56	0.13	1.11	0.27	1.72	0.33
satd	0.54	0.53	-	-	-	-	0.63	0.38	1.25	0.59	2.20	0.78
sha	0.49	0.15	0.86	0.35	-	-	0.80	0.32	1.18	0.39	1.89	0.60
sobel	1.11	1.29	-	-	-	-	0.84	0.95	1.29	1.08	1.50	1.37
Gmean	0.39	0.18	0.87	0.39	1.83	0.30	0.57	0.11	1.14	0.37	1.88	0.32

5.6.4 Utilization and performance

In Fig. 5.22 (a) we report on the number of inserted synchronisation and partitioning voters in the TMR designs. In Fig. 5.22 (b) we show how these voters affect the slice utilisation. In Fig. 5.22 (c) we show how triplication affects the FM of the circuits when the designs are not floorplanned, while in Fig. 5.22 (d) we show how FM is affected when they are floorplanned. We found that the slice utilisation of the floorplanned implementations is similar to that of the non-floorplanned implementations. In more detail, Fig. 5.22 (a) reports the actual number of inserted voters, while Figs. 5.22 (b), (c), and (d) report results that are normalised to the corresponding simplex results (Simplex/NIF for TMR/FLP, and Simplex/IF for TMR/ILP).

The left and right sides of the figure illustrate the results for the FLP and ILP circuits respectively. As expected, the number of voters in the ILP circuits increased much more than for the FLP circuits as k increases from $1 \rightarrow 8$, due to the increasing number of nets

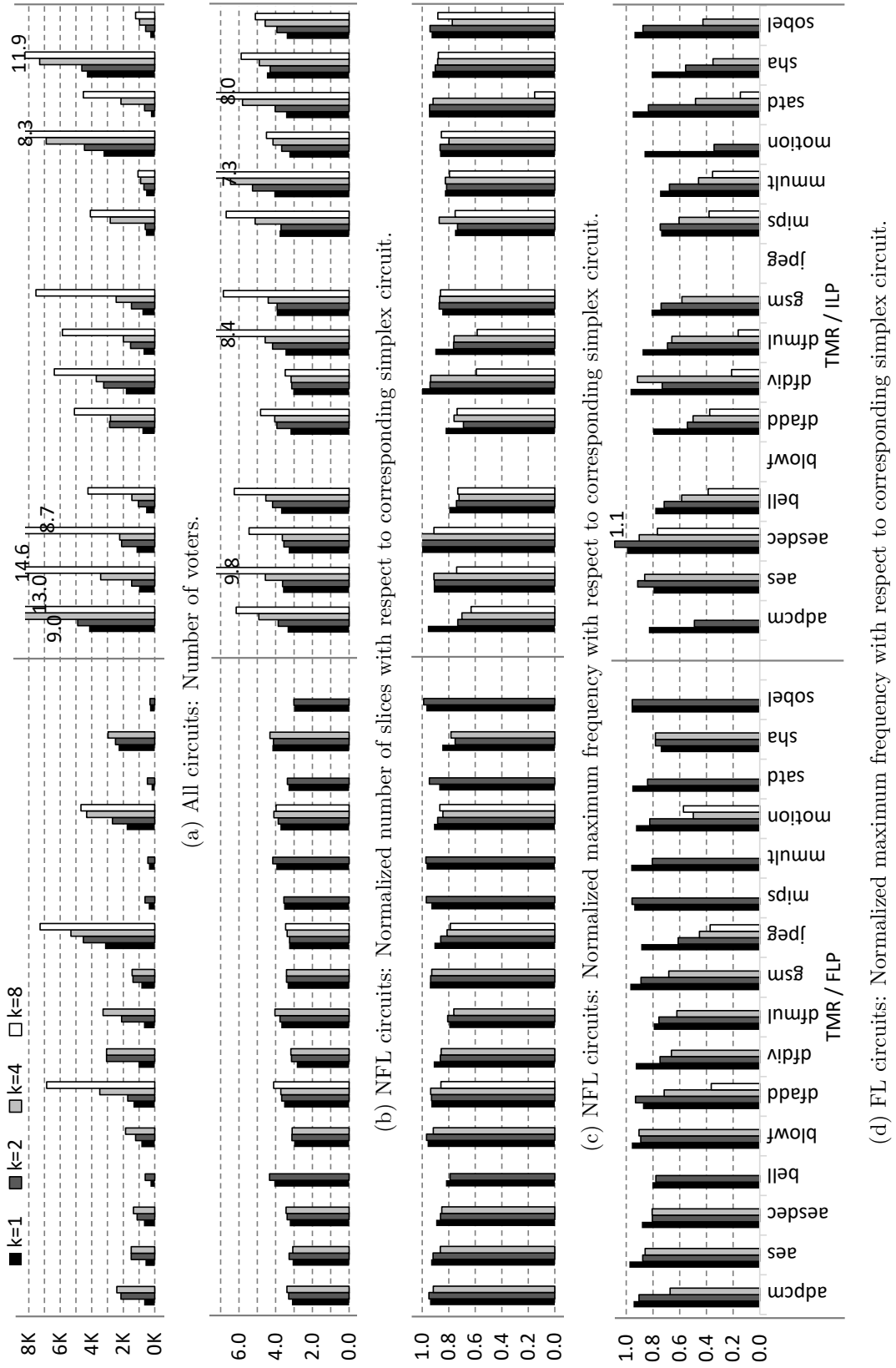


Figure 5.22: Number of inserted voters, resource utilization and maximum frequency of TMR circuits relative to their corresponding simplex circuits.

and therefore utilised voters between the central FSM and the partitions.

We found the slice utilization of the FLP designs to be more consistently around $3\text{--}4\times$ that of the Simplex/NIF designs and that it did not suffer the exponential utilization increase ILP designs exhibit as k increases.

For the non-floorplanned circuits, the performance penalty suffered by FLP is generally less than that suffered by the ILP circuits. The FLP results also appear to be more stable as k increases, and there is less variability in the results across the circuits. With floorplanning, the performance hit is greater and there is a more pronounced deterioration in performance for ILP with increasing k . When comparing the FLP TMR circuit with their simplex counterparts for $k=1$ and $k=2$, the triplicated circuits had on average 11% frequency drop when they were not floorplanned and 13% when they were. We do not compare the ILP TMR circuits with their simplex counterparts since most triplicated circuits that were floorplanned failed to finish routing.

5.6.5 Soft error sensitivity

Table 5.5 reports on the SES of the TMR/FLP and the TMR/ILP designs relative to the Simplex/NIF and the Simplex/IF designs, respectively, for both NFL and FL implementations. Since comparable data is lacking, we computed Gmeans across all designs excluding *blowfish*, *dfsin*, *jpeg* and *sha* for $k = 1$ and 2 , and found that TMR affords approximately $500\times$ improvement in SES over the corresponding simplex circuits. This improvement is enhanced by a factor of $1.3\times - 3.4\times$, on average, when the circuits are floorplanned. For $k \geq 4$, we feel that there is insufficient data to produce meaningful results.

We generally observe a decrease of SES in both the NFL and FL designs across all values of k . Exceptions to this observation, such as *mmult*, *satd*, *sha* and *sobel*, may simply be due to the different AVF of each circuit.

5.7 Chapter Summary

In this chapter we presented an automated flow that produces fault-tolerant FPGA implementations of TMR circuits from C specifications using a high-level synthesis tool based on LegUp, and an academic floorplanner to isolate the modules of each component and to facilitate recovery from configuration memory errors. We presented and compared two

Table 5.5: Normalized soft error sensitivity (SES) of the TMR circuits.

	k	Normalized SES (Simplex / TMR)														sha	sobel	Gmean
		adpcm	aes	aesdec	bell	blowf	dfadd	dfdiv	dfmul	dfsin	gsm	jpeg	mips	mmult	motion	satd		
FLP	1	3,719	733	893	496	3	540	1,619	643	-	220	-	2,150	26	692	3,416	12	499
	2	2,678	1,587	633	344	29	575	984	1,092	-	811	-	774	10	503	1,756	22	402
	4	2,656	921	911	-	23	482	775	652	-	722	-	-	-	484	-	348	NA
	8	-	-	-	-	-	587	-	-	-	-	-	-	-	748	-	-	NA
FL	1	4,954	1,525	4,401	959	2	1,102	1,005	1,315	-	79	-	2,935	39	2,047	1,709	15	638
	2	20,573	2,735	2,839	932	9	903	1,140	4,762	-	1,879	-	9,325	523	1,894	2,412	32	1,380
	4	11,945	2,979	2,601	-	77	1,239	4,482	1,018	-	1,083	-	-	-	710	-	100	NA
	8	-	-	-	-	-	2,256	-	-	-	-	-	-	-	1,022	-	-	NA
NLP	1	1,483	1,895	1,033	1,684	-	515	696	300	-	985	-	1,301	129	438	911	86	493
	2	1,135	1,499	2,356	845	-	431	1,084	519	-	694	-	1,021	777	639	821	151	593
	4	1,643	1,384	2,040	871	-	1,031	638	589	-	555	-	2,083	370	954	1,610	393	NA
	8	1,485	3,710	3,151	993	-	611	1,271	440	-	-	-	2,052	114	655	1,187	1,512	NA
FL	1	5,057	7,247	5,646	2,080	-	1,139	1,104	1,323	-	6,084	-	2,499	65	946	911	160	1,035
	2	19,354	3,307	3,330	1,583	-	1,545	647	862	-	3,029	-	2,637	286	347	956	-	1,010
	4	-	2,259	-	2,762	-	1,212	-	670	-	3,569	-	2,876	8,531	-	1,351	-	NA
	8	-	-	-	8,612	-	419	-	-	-	-	-	3,210	348	-	-	-	NA

automated approaches to partitioning the circuits into a number of equally-sized TMR components for the sake of enhancing the reliability of the circuit and reducing the error recovery time when MER or FMER is used.

Our experimental analysis, using standard HLS benchmark designs, found that partitioning the designs at the C function level (FLP) afforded a number of advantages over partitioning the designs at the more fine-grained instruction level (ILP). The advantages of triplicating with FLP over ILP, and relative to corresponding circuits that are not triplicated, include: lower resource utilization, less performance impact, and better resource balancing. We found little difference between FLP and ILP in the improvement in SEU sensitivity (SES) gained from triplication, which was found to be equally good for both methods. We also found that floorplanning resulted in a marked further improvement in SES compared with not floorplanning the circuit implementations. Overall, we observed that most circuits could be implemented in under 123 minutes after the C specification was available. This is a significant improvement over manually implementing partitioned, TMR circuits. Our automatic flow effectively relieves the designer of a good deal of complex and tedious labour.

Chapter 6

Conclusions

Traditionally, researchers and practitioners have used space-grade Field Programmable Gate Arrays (FPGAs) in space applications. These devices are more immune to SEUs than commercial SRAM FPGAs and can withstand a higher level of total ionising dose (TID). However, this thesis argues that space-grade FPGAs are not practical for developing low-cost, high-performance space applications, for instance, computing systems used in CubeSats, due to their high cost, restricted availability, and outdated characteristics.

This thesis proposed fault-tolerant techniques and computer-aided design (CAD) tools to enable commercial SRAM FPGA to operate reliably in space. In summary, a mechanism for rapidly and efficiently recovering configuration memory (CM) upsets in Triple Modular Redundancy (TMR) FPGA was proposed. Dependability and energy consumption models were derived to better understand how TMR FPGA circuits perform in space, depending on several design parameters, such as the utilised CM error recovery mechanism, and the number of TMR partitions. A technique to reliably transfer the minority reports of TMR circuits, i.e., which TMR domains require recovery, to the reconfiguration controller (RC) of the FPGA circuits was proposed. A computer-aided design (CAD) tool, namely TLegUp, that automatically triplicates and partitions Xilinx 7-series FPGA designs with high-level synthesis (HLS) techniques was proposed. An investigation of how floorplanning affects the soft error sensitivity (SES) in TMR FPGA circuits was conducted. Finally, a rich set of analytical and experimental results were provided to demonstrate the effectiveness and practicality of the thesis contributions.

This chapter summarises the contents of this thesis, draws conclusions, and finally provides future research directions.

6.1 Thesis Summary and Concluding Remarks

As Chapter 1 reported, instruments and sensors in space applications can generate a large amount of raw data in short time periods. This raises the need for onboard high-performance computing systems to perform computationally intensive tasks, such as data compression and real-time data mining, in order to reduce bandwidth saturation and enable data transmission to ground stations. Commercial SRAM-based FPGAs provide attractive specifications for developing high-performance computing systems but cannot reliably operate in the space radiation environment. As mentioned, SRAM FPGAs store their user memory (UM), internal proprietary state (ISP) and CM in SRAM cells which may become corrupted when struck by high-energy particles in space. Such events are called Single Event Upsets (SEUs) and often corrupt the functionality of the FPGA circuit. To overcome these limitations, engineers incorporate fault-tolerant techniques, such as hardware redundancy and error recovery mechanisms when developing SRAM FPGA circuits for space applications.

To better understand how ionising radiation affects the functionality of commercial SRAM FPGA circuits and how these have been handled by previously reported fault-tolerant techniques, Chapter 2 provided a thorough literature survey. In more detail, information about the sources of radiation in space and the classification of radiation effects in SRAM FPGAs was reported. Chapter 2 pointed out that state-of-the-art commercial SRAM FPGAs, such as the Xilinx 7 Series FPGAs, can withstand 100 krad TID and are immune to Single Event Latchups (SELs) at a linear energy transfer (LET) of more than 100 MeV. As related work revealed, many space missions operating in Low Earth Orbit (LEO) for short time periods, e.g., low-cost CubeSat applications, do not surpass these TID and LET limits. Therefore, it is clear that commercial SRAM FPGA circuits can be used in such space missions if the effects of upsets in their CM and UM are mitigated. Chapter 2 provided a survey of existing TMR design techniques for FPGA circuits. The survey revealed that the reliability of TMR circuits is significantly increased when combined with rapid error recovery mechanisms, design partitioning, as well as, circuit floorplanning.

Chapter 3 showed that Module-based CM error Recovery (MER), which is a faster and more energy efficient mechanism to recover CM upsets in TMR circuits than classic device periodic scrubbing, has a considerable limitation; Soft-errors in programmable resources that reside outside the reconfigurable regions of TMR modules (referred to as support resources) are not recovered. Therefore, both the reliability and availability of the circuit are decreased. To overcome this limitation a so-called Frame- and Module-based CM Error Recovery (FMER) technique was proposed. FMER uses selective periodic CM

scrubbing and MER to recover the support resources and the TMR modules of the circuit, respectively. The efficacy of this method was evaluated by deriving and comparing the reliability, availability and energy consumption of TMR FPGA circuits that incorporated either FMER, MER, blind scrubbing or no recovery at all. The results indicated that FMER is particularly beneficial in missions with high-reliability requirements as well as in missions where the mission’s energy budget prohibits periodically scrubbing the FPGA. For example, it was found that FMER is able to provide the same reliability in a TMR circuit as device periodic scrubbing does. However, the circuit with FMER consumed 347 times less energy. FMER was also demonstrated on 11 HLS designs which were triplicated with TLegUp. These circuits were implemented on an Artix-7 200T FPGA and incorporated FMER, MER or blind scrubbing. For a 2-year LEO mission, the energy consumption of the circuits utilising FMER was found to be on average 1.68x less than of those utilising device periodic scrubbing. However, the reliability and availability of all HLS circuits was found to be identical, independent of the utilised error recovery mechanism. Both, the analytical and experimental results of Chapter 3 revealed that the reliability and availability of a TMR FPGA circuit are dictated by the dependability of any simplex logic they have incorporated. It also showed that the reliability of simplex logic is independent of the utilised error recovery mechanism. The TMR HLS circuits used in Chapter 3 experiments included simplex components and therefore all had similar dependability.

Chapter 4 evaluated the impact of Reconfiguration Control Networks (RCNs) on the reliability and performance of TMR FPGA circuits with MER. The RCN can significantly compromise the reliability of TMR circuit with MER as it is used for conveying the minority reports from TMR components to a central RC. The RC will not recover a faulty TMR module if not reported from a faulty RCN. To reduce the possibility of this happening, a reliable RCN was proposed. Instead of using the programmable resources of the FPGA to transfer the minority reports to the RC, the proposed RCN used the configuration-layer of the FPGA. In this way, the failure rate of the RCN was reduced since a negligible amount of programmable resources were used for its implementation. Both classic application-layer RCNs that utilised programmable resources for their implementation as well as the proposed configuration-layer RCNs were implemented on a Xilinx Artix-7 FPGA. The post-routing utilization and performance of the designs as well as their SES were evaluated through analytical modelling and fault injection experiments. Results showed that of the RCN topologies studied, the configuration-layer RCN was the most reliable, despite having the highest network latency.

Finally, Chapter 5 presented the TLegUp CAD tool which is an extension of the open

source LegUp framework. TLegUp takes an algorithm expressed in the ANSI C programming language and produces partitioned TMR design descriptions in Verilog, which are implemented on Xilinx 7-series FPGAs with the Vivado design suite. TLegUp can also be configured to floorplan the design in order to reduce shared resources between redundant logic of the TMR scheme and decrease the single points of failure in the circuit. TLegUp improves the productivity of application designers for space; allows designers to experiment with alternative application partitioning; and supports the automatic insertion of the infrastructure needed to run a fault-tolerant system. As Chapter 5 reports, TLegUp is able to use voters with registered outputs in order to alleviate the frequency drop-off caused by voter insertion and circuit triplication. In contrast, current tools that triplicate the design during the register-level transfer (RTL) pre- or post-synthesis stages of the CAD flow are constrained to use only combinational voters so as to preserve the timing specification of the design; critical path lengths are consequently increased. Two circuit partitioning approaches were investigated in TLegUp: 1) an instruction level approach which inlined all C functions into a single Data Flow Graph (DFG) to partition the design at a fine level of granularity, and 2) a more coarse-grained approach that forms partitions by grouping C functions. TLegUp was evaluated by implementing non-floorplanned and floorplanned TMR designs of several HLS benchmarks. All designs were implemented on a Xilinx Artix-7 200T FPGA and compared against non-triplicated baseline designs. The TMR designs were partitioned at the instruction level (ILP) as well as at the C function level (FLP) for $k = 1, 2, 4$ and 8 partitions. The quality of these designs was evaluated in terms of post-routing resource utilisation, resource balancing between partitions, maximum frequency, latency and execution time, and SES. The experimental results of Chapter 5 illustrated that circuits that were partitioned with the FLP approach utilised approximately $3 - 4\times$ more resources than simplex baseline circuits, irrespective of the number of circuit partitions. In contrast, circuits that were partitioned with ILP suffered an exponential resource utilisation growth as k increased as well as a significant performance penalty. We, therefore, conclude that the ILP approach is not practical. When comparing the TMR FLP circuit with their simplex counterparts for $k=1$ and $k=2$, the triplicated circuits had on average 11% and 13% frequency reduction when they were not floorplanned, respectively. Finally, fault-injection experiments showed that the SES of the ILP and the FLP circuits for $k = 1$ and 2 was approximately $500\times$ less than the simplex baseline designs. This figure was further improved by a factor of $1.3\times - 3.4\times$, on average, when the circuits were floorplanned.

6.2 Future Research Directions

This thesis proposed several novel ideas to mitigate the effects of CM errors in commercial SRAM FPGA. These ideas can be further improved and complemented by the following:

- A thorough analysis of how the essential bits of TMR FPGA circuits are distributed across the configuration memory and resources of the device as proposed in [105] can be insightful to further improve FMER. The experimental results in Chapter 3 indicated that the programmable resources located outside the reconfigurable regions of TMR modules are not utilised as much as those located inside the regions. This suggests that the configuration frames for the non-utilised resources that reside outside the TMR modules may avoid scrubbing in order to reduce the recovery time or energy consumption of FMER.
- A hardened Network on Chip (NoC) was recently embedded on Xilinx’s next generation Versal FPGAs. It would be interesting to investigate if an RCN can be implemented with the NoC of these devices and what would be its performance and reliability.
- It would be useful to provide a means for the designer to manually specify which C functions to include in which partitions in TLegUp. Additionally, it would be interesting to compare the soft error sensitivity, operating frequency and resource utilisation of designs triplicated at the HLS level with those triplicated at the RTL pre- or post-synthesis level. For example, to compare TMR designs produced by TLegUp with those produced by BL-TMR.

References

- [1] J. H. Adams, A. F. Barghouty, M. H. Mendenhall, R. A. Reed, B. D. Sierawski, K. M. Warren, J. W. Watts, and R. A. Weller, “CREME: The 2011 revision of the cosmic ray effects on micro-electronics code,” *IEEE Transactions on Nuclear Science (TNS)*, vol. 59, no. 6, pp. 3141–3147, Dec 2012. [Online]. Available: <https://doi.org/10.2514/6.2000-371>
- [2] P. Adell and G. Allen, “Assessing and mitigating radiation effects in Xilinx FPGAs,” Pasadena, CA: Jet Propulsion Laboratory, California Institute of Technology, Tech. Rep., 2008.
- [3] D. Agiakatsikas, N. T. H. Nguyen, Z. Zhao, T. Wu, E. Cetin, O. Diessel, and L. Gong, “Reconfiguration control networks for TMR systems with module-based recovery,” in *IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2016, pp. 88–91. [Online]. Available: <https://doi.org/10.1109/fccm.2016.30>
- [4] M. Alderighi, S. D’Angelo, F. Casini, G. Sorrenti, D. M. Codinachs, and S. Davin, “The FLIPPER fault injection platform: experiences and knowledge from a ten-year project,” in *International Conference on Architecture of Computing Systems (ARCS)*, April 2017, pp. 1–8.
- [5] H. Asadi and M. B. Tahoori, “Analytical techniques for soft error rate modeling and mitigation of FPGA-based designs,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 15, no. 12, pp. 1320–1331, Dec. 2007. [Online]. Available: <https://doi.org/10.1109/TVLSI.2007.909795>
- [6] A. Avizienis, “Fault-tolerant systems,” *IEEE Transactions on Computers*, vol. C-25, no. 12, pp. 1304–1312, Dec 1976. [Online]. Available: <https://doi.org/10.1109/tc.1976.1674598>

- [7] J. R. Azambuja, F. Sousa, L. Rosa, and F. L. Kastensmidt, "Evaluating large grain TMR and selective partial reconfiguration for soft error mitigation in SRAM-based FPGAs," in *IEEE International On-Line Testing Symposium*, June 2009, pp. 101–106. [Online]. Available: <https://doi.org/10.1109/IOLTS.2009.5195990>
- [8] M. Berg, C. Poivey, D. Petrick, D. Espinosa, A. Lesea, K. A. LaBel, M. Friendlich, H. Kim, and A. Phan, "Effectiveness of internal versus external SEU scrubbing mitigation strategies in a Xilinx FPGA: design, test, and analysis," *IEEE Transactions on Nuclear Science (TNS)*, vol. 55, no. 4, pp. 2259–2266, Aug 2008. [Online]. Available: <https://doi.org/10.1109/TNS.2008.2001422>
- [9] V. Betz, J. Rose, and A. Marquardt, *Architecture and CAD for deep-submicron FPGAs*. Springer Science & Business Media, 2012, vol. 497. [Online]. Available: <https://doi.org/10.1007/978-1-4615-5145-4>
- [10] D. Binder, E. C. Smith, and A. B. Holman, "Satellite anomalies from galactic cosmic rays," *IEEE Transactions on Nuclear Science (TNS)*, vol. 22, no. 6, pp. 2675–2680, Dec 1975. [Online]. Available: <https://doi.org/10.1109/TNS.1975.4328188>
- [11] G. Blake, R. G. Dreslinski, and T. Mudge, "A survey of multicore processors," *IEEE Signal Processing Magazine*, vol. 26, no. 6, pp. 26–37, November 2009. [Online]. Available: <https://doi.org/10.1109/MSP.2009.934110>
- [12] C. Bolchini, F. Castro, and A. Miele, "A fault analysis and classifier framework for reliability-aware SRAM-based FPGA systems," in *IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT)*, Oct 2009, pp. 173–181. [Online]. Available: <https://doi.org/10.1109/DFT.2009.10>
- [13] C. Bolchini, P. L. Lanzi, and A. Miele, "A multi-objective genetic algorithm framework for design space exploration of reliable FPGA-based systems," in *IEEE Congress on Evolutionary Computation (CEC)*, July 2010, pp. 1–8. [Online]. Available: <https://doi.org/10.1109/CEC.2010.5586376>
- [14] C. Bolchini and A. Miele, "Design space exploration for the design of reliable SRAM-based FPGA systems," in *IEEE International Symposium on Defect and Fault Tolerance of VLSI Systems*, Oct 2008, pp. 332–340. [Online]. Available: <https://doi.org/10.1109/dft.2008.8>
- [15] C. Bolchini, A. Miele, and C. Sandionigi, "A novel design methodology for implementing reliability-aware systems on SRAM-based FPGAs," *IEEE Transactions on Computers*, vol. 60, no. 12, pp. 1744–1758, Dec 2011. [Online]. Available: <https://doi.org/10.1109/tc.2010.281>

- [16] C. Bolchini, A. Miele, C. Sandionigi, N. Battezzati, L. Sterpone, and M. Violante, “An integrated flow for the design of hardened circuits on SRAM-based FPGAs,” in *IEEE European Test Symposium (ETS)*, May 2010, pp. 214–219. [Online]. Available: <https://doi.org/10.1109/etsym.2010.5512757>
- [17] C. Bolchini, A. Miele, and M. D. Santambrogio, “TMR and partial dynamic reconfiguration to mitigate SEU faults in FPGAs,” in *IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT)*, Sept 2007, pp. 87–95. [Online]. Available: <https://doi.org/10.1109/dft.2007.25>
- [18] C. Bolchini, D. Quarta, and M. D. Santambrogio, “SEU mitigation for SRAM-based FPGAs through dynamic partial reconfiguration,” in *Proceedings of the 17th ACM Great Lakes Symposium on VLSI*, ser. GLSVLSI ’07. New York, NY, USA: ACM, 2007, pp. 55–60. [Online]. Available: <https://doi.org/10.1145/1228784.1228803>
- [19] Brigham Young University Configurable Computing Lab, “BYU-LANL Triple modular redundancy GitHub repository,” 2009. [Online]. Available: <https://sourceforge.net/projects/byuediftools/files/byuediftools/>
- [20] —, *BYU-LANL triple modular redundancy usage guide version 0.5.2*, 2009. [Online]. Available: <https://sourceforge.net/projects/byuediftools/files/byuediftools/proton%20%280.5.2%29/JEdifTools-0.5.2.pdf>
- [21] E. Brosser, F. Milh, V. Geijer, and P. Larsson-Edefors, “Assessing scrubbing techniques for Xilinx SRAM-based FPGAs in space applications,” in *International Conference on Field-Programmable Technology (FPT)*, Dec 2014, pp. 296–299. [Online]. Available: <https://doi.org/10.1109/fpt.2014.7082803>
- [22] M. Brusati, A. Camplani, M. Cannon, H. Chen, M. Citterio, M. Lazzaroni, H. Takai, and M. Wirthlin, “Mitigated FPGA design of multi-gigabit transceivers for application in high radiation environments of High Energy Physics experiments,” *Measurement*, vol. 108, pp. 171 – 192, 2017. [Online]. Available: <https://doi.org/10.1016/j.measurement.2017.02.025>
- [23] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson, “LegUp: an open-source high-level synthesis tool for FPGA-based processor/accelerator systems,” *ACM Transactions on Embedded Computing Systems*, vol. 13, no. 2, p. 24, 2013. [Online]. Available: <https://doi.org/10.1145/2514740>

- [24] A. C. Canis, “Legup: open-source high-level synthesis research framework,” Ph.D. dissertation, University of Toronto – Department of Electrical and Computer Engineering, 2015. [Online]. Available: <http://hdl.handle.net/1807/70811>
- [25] M. Cannon, A. Keller, and M. Wirthlin, “Improving the effectiveness of TMR designs on FPGAs with SEU-aware incremental placement,” in *IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2018, pp. 1–8. [Online]. Available: <https://doi.org/10.1109/fccm.2018.00031>
- [26] C. Carmichael, *Triple module redundancy design techniques for Virtex FPGAs application note v1.01 (XAPP197)*, Xilinx Inc., 2001.
- [27] C. Carmichael, M. Caffrey, and S. Anthony, *Correcting single-event upsets through Virtex partial configuration application note v1.0 (XAPP216)*, Xilinx Inc., 2000.
- [28] C. Carmichael and C. W. Tseng, *Correcting single-event upsets in Virtex-4 FPGA configuration memory application note (XAPP1088)*, Xilinx Inc., 2009.
- [29] E. Cetin, O. Diessel, L. Gong, and V. Lai, “Reconfiguration network design for SEU recovery in FPGAs,” in *IEEE International Symposium on Circuits and Systems (ISCAS)*, June 2014, pp. 1524–1527. [Online]. Available: <https://doi.org/10.1109/iscas.2014.6865437>
- [30] E. Cetin, O. Diessel, and L. Gong, “Improving Fmax of FPGA circuits employing DPR to recover from configuration memory upsets,” in *IEEE Inter. Symposium on Circuits and Systems (ISCAS)*, 2015.
- [31] E. Cetin, O. Diessel, L. Gong, and V. Lai, “Towards bounded error recovery time in FPGA-based TMR circuits using dynamic partial reconfiguration,” in *International Conference on Field Programmable Logic and Applications (FPL)*, 2013, pp. 1–4. [Online]. Available: <https://doi.org/10.1109/fpl.2013.6645571>
- [32] E. Cetin, O. Diessel, T. Li, J. A. Ambrose, T. Fisk, S. Parameswaran, and A. G. Dempster, “Overview and investigation of SEU detection and recovery approaches for FPGA-based heterogeneous systems,” in *FPGAs and Parallel Architectures for Aerospace Applications*. Springer, 2016, ch. 3, pp. 33–46. [Online]. Available: https://doi.org/10.1007/978-3-319-14352-1_3
- [33] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, “High-level synthesis for FPGAs: From prototyping to deployment,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 30, no. 4, pp. 473–491, April 2011.

- [34] S. D'Angelo, C. Metra, S. Pastore, A. Pogutz, and G. R. Sechi, "Fault-tolerant voting mechanism and recovery scheme for TMR FPGA-based systems," in *IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, Nov 1998, pp. 233–240. [Online]. Available: <https://doi.org/10.1109/DFTVS.1998.732171>
- [35] S. Dasgupta, C. H. Papadimitriou, and U. Vazirani, *Algorithms*, 1st ed. McGraw-Hill, Inc., 2008.
- [36] C. Derek and E. Crabill, *UltraScale devices maximize design integrity with industry-leading SEU resilience and mitigation v1.0 (WP462)*, Xilinx Inc., 2015.
- [37] A. F. dos Santos, L. A. Tambara, and F. L. Kastensmidt, "Evaluating the efficiency of using TMR in the high-level synthesis design flow of SRAM-based FPGA," in *IEEE Latin American Symposium on Circuits Systems (LASCAS)*, Feb 2017, pp. 1–4. [Online]. Available: <https://doi.org/10.1109/LASCAS.2017.7948064>
- [38] M. Ebrahimi, P. M. B. Rao, R. Seyyedi, and M. B. Tahoori, "Low-cost multiple bit upset correction in SRAM-based FPGA configuration Frames," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24, no. 3, pp. 932–943, March 2016.
- [39] S. T. Fleming and D. B. Thomas, "StitchUp: Automatic control flow protection for high level synthesis circuits," in *Proceedings of the 53rd Annual Design Automation Conference (DAC)*. New York, NY, USA: ACM, 2016, pp. 138:1–138:6. [Online]. Available: <http://doi.acm.org/10.1145/2897937.2898097>
- [40] A. D. George and C. M. Wilson, "Onboard processing with hybrid and reconfigurable computing on small satellites," *Proceedings of the IEEE*, vol. 106, no. 3, pp. 458–470, March 2018. [Online]. Available: <https://doi.org/10.1109/jproc.2018.2802438>
- [41] L. Gong, A. Kroh, D. Agiakatsikas, N. T. H. Nguyen, E. Cetin, and O. Diessel, "Reliable SEU monitoring and recovery using a programmable configuration controller," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, Sept 2017, pp. 1–6. [Online]. Available: <https://doi.org/10.23919/FPL.2017.8056798>
- [42] L. Gong, T. Wu, N. T. H. Nguyen, D. Agiakatsikas, Z. Zhao, E. Cetin, and O. Diessel, "A programmable configuration controller for fault-tolerant applications," in *2016 International Conference on Field-Programmable Technology (FPT)*, Dec 2016, pp. 117–124. [Online]. Available: <https://doi.org/10.1109/FPT.2016.7929515>

- [43] P. Graham, M. Caffrey, D. E. Johnson, N. Rollins, and M. Wirthlin, “SEU mitigation for half-latches in Xilinx Virtex FPGAs,” *IEEE Transactions on Nuclear Science (TNS)*, vol. 50, no. 6, pp. 2139–2146, Dec 2003. [Online]. Available: <https://doi.org/10.1109/TNS.2003.820744>
- [44] E. Hallett, *Developing secure and reliable single device designs with Xilinx 7 series FPGAs or Zynq-7000 AP SoCs using the isolation design flow Xilinx application note v1.3.1 (XAPP1086)*, Xilinx Inc., 2015.
- [45] —, *Isolation design flow for Xilinx 7 series FPGAs or Zynq-7000 AP SoCs (Vivado tools) application note v1.3 (XAPP1222)*, Xilinx Inc., 2016.
- [46] Y. Hara, H. Tomiyama, S. Honda, and H. Takada, “Proposal and quantitative analysis of the CHStone benchmark program suite for practical C-based high-level synthesis,” *Journal of Information Processing.*, vol. 17, pp. 242–254, 2009. [Online]. Available: <https://doi.org/10.2197/ipsjjip.17.242>
- [47] S. Hauck and A. DeHon, *Reconfigurable computing: The theory and practice of FPGA-based computation*. Elsevier, 2008. [Online]. Available: <https://doi.org/10.1016/b978-0-12-370522-8.x5001-8>
- [48] I. Herrera-Alzu and M. Lopez-Vallejo, “Design techniques for Xilinx Virtex FPGA configuration memory scrubbers,” *IEEE Transactions on Nuclear Science (TNS)*, vol. 60, no. 1, pp. 376–385, Feb 2013. [Online]. Available: <https://doi.org/10.1109/TNS.2012.2231881>
- [49] I. Hwang, S. Kim, Y. Kim, and C. E. Seah, “A survey of fault detection, isolation, and reconfiguration methods,” *IEEE Transactions on Control Systems Technology*, vol. 18, no. 3, pp. 636–653, May 2010. [Online]. Available: <https://doi.org/10.1109/tcst.2009.2026285>
- [50] Intel Corp., *Quartus II design separation flow v.1.0 (AN-567)*, June 2009.
- [51] —, *Partial reconfiguration user guide v2018.09.24 (UG-20136)*, 2018.
- [52] X. Iturbe, K. Benkrid, T. Arslan, R. Torrego, and I. Martinez, “Methods and mechanisms for hardware multitasking: Executing and synchronizing fully relocatable hardware tasks in Xilinx FPGAs,” in *International Conference on Field Programmable Logic and Applications (FPL)*, ser. FPL ’11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 295–300. [Online]. Available: <https://doi.org/10.1109/FPL.2011.60>

- [53] A. Jacobs, G. Cieslewski, A. D. George, A. Gordon-Ross, and H. Lam, “Reconfigurable fault tolerance: A comprehensive framework for reliable and adaptive FPGA-based space computing,” *ACM Transactions Reconf. Technol. Syst. (TRETS)*, vol. 5, no. 4, pp. 21:1–21:30, 2012. [Online]. Available: <https://doi.org/10.1145/2392616.2392619>
- [54] J. M. Johnson and M. J. Wirthlin, “Voter insertion algorithms for FPGA designs using triple modular redundancy,” in *Proceedings of ACM/SIGDA International Symposium on Field-programmable Gate Arrays (FPGA)*, 2010. [Online]. Available: <https://doi.org/10.1145/1723112.1723154>
- [55] T. Karnik and P. Hazucha, “Characterization of soft errors caused by single event upsets in CMOS processes,” *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 2, pp. 128–143, April 2004. [Online]. Available: <https://doi.org/10.1109/tdsc.2004.14>
- [56] F. L. Kastensmidt, C. Carmichael, J. Fabula, R. Padovani, and R. Reis, “A fault injection analysis of Virtex FPGA TMR design methodology,” in *6th European Conference on Radiation and Its Effects on Components and Systems (RADECS)*, Sept 2001, pp. 275–282. [Online]. Available: <https://doi.org/10.1109/RADECS.2001.1159293>
- [57] F. L. Kastensmidt and L. Carro, *Fault-tolerance techniques for SRAM-based FPGAs*. Springer, 2006. [Online]. Available: <https://doi.org/10.1007/978-0-387-31069-5>
- [58] A. M. Keller, T. A. Whiting, K. B. Sawyer, and M. J. Wirthlin, “Dynamic SEU sensitivity of designs on Two 28-nm SRAM-based FPGA architectures,” *IEEE Transactions on Nuclear Science (TNS)*, vol. 65, no. 1, pp. 280–287, Jan 2018. [Online]. Available: <https://doi.org/10.1109/TNS.2017.2772288>
- [59] I. Koren and C. M. Krishna, *Fault-tolerant systems*. Morgan Kaufmann, 2007. [Online]. Available: <https://doi.org/10.1016/b978-0-12-088525-1.x5000-7>
- [60] L. E. LaForge, J. R. Moreland, R. G. Bryan, and M. Sami Fadali, “Vertical cavity surface emitting lasers for spaceflight multi-processors,” in *IEEE Aerospace Conference*, March 2006, pp. 1–19.
- [61] J. H. Lala and R. E. Harper, “Architectural principles for safety-critical real-time applications,” *Proceedings of the IEEE*, vol. 82, no. 1, pp. 25–40, Jan 1994. [Online]. Available: <https://doi.org/10.1109/5.259424>

- [62] T. Lange, B. Fiethe, H. Michel, H. Michalik, K. Albert, and J. Hirzberger, “On-board processing using reconfigurable hardware on the solar orbiter PHI instrument,” in *NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, July 2017, pp. 186–191. [Online]. Available: <https://doi.org/10.1109/ahs.2017.8046377>
- [63] C. Lattner and V. Adve, “LLVM: A compilation framework for lifelong program analysis & transformation,” in *Code Generation and Optimization (CGO)*, 2004, p. 75. [Online]. Available: <https://doi.org/10.1109/cgo.2004.1281665>
- [64] C. Lavin, M. Padilla, J. Lamprecht, P. Lundrigan, B. Nelson, and B. Hutchings, “RapidSmith: Do-it-yourself CAD tools for Xilinx FPGAs,” in *International Conference on Field Programmable Logic and Applications (FPL)*, Sept 2011, pp. 349–355. [Online]. Available: <https://doi.org/10.1109/fpl.2011.69>
- [65] D. S. Lee, M. Wirthlin, G. Swift, and A. C. Le, “Single-event characterization of the 28 nm Xilinx Kintex-7 Field-Programmable Gate Array under heavy ion irradiation,” in *IEEE Radiation Effects Data Workshop (REDW)*, July 2014, pp. 1–5. [Online]. Available: <https://doi.org/10.1109/redw.2014.7004595>
- [66] Y. Li, B. Nelson, and M. Wirthlin, “Reliability models for SEC/DED memory with scrubbing in FPGA-based designs,” *IEEE Transactions on Nuclear Science (TNS)*, vol. 60, no. 4, pp. 2720–2727, Aug 2013. [Online]. Available: <https://doi.org/10.1109/tns.2013.2251902>
- [67] H. Liu and D. Wong, “Network-flow-based multiway partitioning with area and pin constraints,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 17, no. 1, pp. 50–59, 1998.
- [68] S. Liu, R. N. Pittman, A. Form, and J. Gaudiot, “On energy efficiency of reconfigurable systems with run-time partial reconfiguration,” in *IEEE Inter. Conference on Application-specific Systems, Architectures and Processors (ASAP)*, July 2010, pp. 265–272. [Online]. Available: <https://doi.org/10.1109/asap.2010.5540985>
- [69] S. Liu, R. N. Pittman, A. Forin, and J.-L. Gaudiot, “Achieving energy efficiency through runtime partial reconfiguration on reconfigurable systems,” *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 12, no. 3, p. 72, 2013. [Online]. Available: <https://doi.org/10.1145/2442116.2442122>
- [70] T. M. Lovelly and A. D. George, “Comparative analysis of present and future space-grade processors with device metrics,” *Journal of Aerospace*

- Information Systems*, vol. 14, no. 3, pp. 184–197, 2017. [Online]. Available: <https://doi.org/10.2514/1.I010472>
- [71] R. E. Maeder, *The Mathematica® programmer*. Academic Press, 2014.
- [72] Q. Martin and A. George, “Scrubbing optimization via availability prediction (SOAP) for reconfigurable space computing,” in *IEEE Conference on High Performance Extreme Computing (HPEC)*, Sept 2012, pp. 1–6. [Online]. Available: <https://doi.org/10.1109/hpec.2012.6408673>
- [73] D. McMurtrey, K. S. Morgan, B. Pratt, and M. J. Wirthlin, “Estimating TMR reliability on FPGAs using Markov models,” 2008, unpublished. [Online]. Available: <https://scholarsarchive.byu.edu/facpub/149/>
- [74] MentorGraphics Corp., *Precision Hi-Rel advanced FPGA synthesis*, 2018.
- [75] G. Miller, C. Carmichael, and G. Swift, *Single-event upset mitigation for Xilinx FPGA block memories application note v1.1 (XAPP962)*, Xilinx Inc., 2008.
- [76] S. Mitra and E. McCluskey, “Which concurrent error detection scheme to choose?” in *Proceedings International Test Conference (IEEE Cat. No.00CH37159)*, Oct 2000, pp. 985–994. [Online]. Available: <https://doi.org/10.1109/TEST.2000.894311>
- [77] N. Montealegre, D. Merodio, A. Fernández, and P. Armbruster, “In-flight reconfigurable FPGA-based space systems,” in *NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, June 2015, pp. 1–8. [Online]. Available: <https://doi.org/10.1109/ahs.2015.7231177>
- [78] K. S. Morgan, D. L. McMurtrey, B. H. Pratt, and M. J. Wirthlin, “A comparison of TMR with alternative fault-tolerant design techniques for FPGAs,” *IEEE Transactions on Nuclear Science (TNS)*, vol. 54, no. 6, pp. 2065–2072, Dec 2007. [Online]. Available: <https://doi.org/10.1109/tns.2007.910871>
- [79] S. Mukherjee, *Architecture Design for Soft Errors*. Morgan Kaufmann, 2011. [Online]. Available: <https://doi.org/10.1016/b978-0-12-369529-1.x5001-0>
- [80] R. Nane, V.-M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels, “A survey and evaluation of FPGA high-level synthesis tools,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 35, no. 10, pp. 1591–1604, 2016. [Online]. Available: <https://doi.org/10.1109/tcad.2015.2513673>

- [81] B. Navas, J. Öberg, and I. Sander, “The upset-fault-observer: A concept for self-healing adaptive fault tolerance,” in *NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, July 2014, pp. 89–96. [Online]. Available: <https://doi.org/10.1109/AHS.2014.6880163>
- [82] G. L. Nazar and L. Carro, “Fast single-FPGA fault injection platform,” in *IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, 10 2012, pp. 152–157. [Online]. Available: <https://doi.org/10.1109/DFT.2012.6378216>
- [83] C. D. Norton, T. A. Werne, P. J. Pingree, and S. Geier, “An evaluation of the Xilinx Virtex-4 FPGA for on-board processing in an advanced imaging system,” in *2009 IEEE Aerospace conference*, March 2009, pp. 1–9. [Online]. Available: <https://doi.org/10.1109/AERO.2009.4839460>
- [84] T. R. Oldham and F. B. McLean, “Total ionizing dose effects in MOS oxides and devices,” *IEEE Transactions on Nuclear Science (TNS)*, vol. 50, no. 3, pp. 483–499, June 2003. [Online]. Available: <https://doi.org/10.1109/tns.2003.812927>
- [85] A. Paschalis, H. Michalik, N. Kranitis, C. López-Ongil, and P. R. Vasallo, “Dependable reconfigurable space systems: Challenges, new trends and case studies,” in *IEEE International On-Line Testing Symposium (IOLTS)*, July 2014, pp. 222–227. [Online]. Available: <https://doi.org/10.1109/iolts.2014.6873703>
- [86] K. Paulsson, M. Hübner, and J. Becker, “Strategies to on-line failure recovery in self-adaptive systems based on dynamic and partial reconfiguration,” in *First NASA/ESA Conference on Adaptive Hardware and Systems (AHS’06)*, June 2006, pp. 288–291. [Online]. Available: <https://doi.org/10.1109/AHS.2006.67>
- [87] L. Pereira-Santos, G. L. Nazar, and L. Carro, “Repair of FPGA-based real-time systems with variable slacks,” *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 23, no. 2, pp. 19:1–19:20, January 2018. [Online]. Available: <https://doi.org/10.1145/3144533>
- [88] E. Petersen, *Single event effects in aerospace*. John Wiley & Sons, 2011. [Online]. Available: <https://doi.org/10.1002/9781118084328>
- [89] K. D. Pham, E. Horta, D. Koch, A. Vaishnav, and T. Kuhn, “IPRDF: An isolated partial reconfiguration design flow for Xilinx FPGAs,” in *IEEE International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoc)*, 2018, pp. 1–8.

- [90] B. Pratt, M. Caffrey, J. F. Carroll, P. Graham, K. Morgan, and M. Wirthlin, "Fine-grain SEU mitigation for FPGAs using partial TMR," *IEEE Transactions on Nuclear Science (TNS)*, vol. 55, no. 4, pp. 2274–2280, Aug 2008. [Online]. Available: <https://doi.org/10.1109/TNS.2008.2000852>
- [91] B. Pratt, M. Caffrey, P. Graham, K. Morgan, and M. Wirthlin, "Improving FPGA design robustness with partial TMR," in *IEEE International Reliability Physics Symposium Proceedings*, March 2006, pp. 226–232. [Online]. Available: <https://doi.org/10.1109/RELPHY.2006.251221>
- [92] B. Przybus, *Xilinx redefines power, performance, and design productivity with three new 28 nm FPGA families: Virtex-7, Kintex-7, and Artix-7 devices white paper v1.4 (WP373)*, Xilinx Inc., 2012.
- [93] H. Quinn, P. Graham, K. Morgan, Z. Baker, M. Caffrey, D. Smith, and R. Bell, "On-orbit results for the Xilinx Virtex-4 FPGA," in *IEEE Radiation Effects Data Workshop*, July 2012, pp. 1–8. [Online]. Available: <https://doi.org/10.1109/redw.2012.6353715>
- [94] H. Quinn, K. Morgan, P. Graham, J. Krone, M. Caffrey, and K. Lundgreen, "Domain crossing errors: Limitations on single device triple-modular redundancy circuits in Xilinx FPGAs," *IEEE Transactions on Nuclear Science (TNS)*, vol. 54, no. 6, pp. 2037–2043, Dec 2007. [Online]. Available: <https://doi.org/10.1109/TNS.2007.910870>
- [95] H. Quinn and M. Wirthlin, "Validation techniques for fault emulation of SRAM-based FPGAs," *IEEE Transactions on Nuclear Science (TNS)*, vol. 62, no. 4, pp. 1487–1500, Aug 2015. [Online]. Available: <https://doi.org/10.1109/TNS.2015.2456101>
- [96] H. Quinn, "Radiation effects in reconfigurable FPGAs," *Semiconductor Science and Technology*, vol. 32, no. 4, pp. 1–8, 2017. [Online]. Available: <https://doi.org/10.1088/1361-6641/aa57f6>
- [97] H. Quinn, P. Graham, K. Morgan, Z. Baker, M. Caffrey, D. Smith, M. Wirthlin, and R. Bell, "Flight experience of the Xilinx Virtex-4," *IEEE Transactions on Nuclear Science (TNS)*, vol. 60, no. 4, pp. 2682–2690, 2013. [Online]. Available: <https://doi.org/10.1109/tns.2013.2246581>
- [98] H. Quinn, K. Morgan, P. Graham, Z. Baker, M. Caffrey, D. Roussel-Dupree, W. Howes, E. Johnson, J. Johnson, J. Krone, D. Lee, K. Lundgreen, T. Nelson,

- B. Pratt, N. Rollins, A. Salazar, G. Swift, and M. Wirthlin, “Improving fault tolerance of SRAM-Based FPGAs in harsh radiation environments.” CRC Press, 2015, vol. 48, ch. 2, p. 31. [Online]. Available: <https://doi.org/10.1201/b19388>
- [99] H. Quinn, D. Roussel-Dupre, M. Caffrey, P. Graham, M. Wirthlin, K. Morgan, A. Salazar, T. Nelson, W. Howes, E. Johnson, J. Johnson, B. Pratt, N. Rollins, and J. Krone, “The Cibola flight experiment,” *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 8, no. 1, pp. 3:1–3:22, March 2015. [Online]. Available: <https://doi.org/10.1145/2629556>
- [100] M. Rabozzi, G. C. Durelli, A. Miele, J. Lillis, and M. D. Santambrogio, “Floorplanning automation for partial-reconfigurable FPGAs via feasible placements generation,” *IEEE Transactionss on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 1, pp. 151–164, Jan 2017. [Online]. Available: <https://doi.org/10.1109/TVLSI.2016.2562361>
- [101] N. Rollins, M. Fuller, and M. J. Wirthlin, “A comparison of fault-tolerant memories in SRAM-based FPGAs,” in *2010 IEEE Aerospace Conference*, March 2010, pp. 1–12. [Online]. Available: <https://doi.org/10.1109/aero.2010.5446661>
- [102] N. H. Rollins, “Hardware and software fault-tolerance of softcore processors implemented in SRAM-Based FPGAs,” Ph.D. dissertation, Brigham Young University – Department of Electrical and Computer Engineering, Provo, UT, USA, 2012, aAI3506158. [Online]. Available: <https://scholarsarchive.byu.edu/etd/2998/>
- [103] J. Rose, A. E. Gamal, and A. Sangiovanni-Vincentelli, “Architecture of field-programmable gate arrays,” *Proceedings of the IEEE*, vol. 81, no. 7, pp. 1013–1029, Jul 1993. [Online]. Available: <https://doi.org/10.1109/5.231340>
- [104] R. A. Sahner, K. Trivedi, and A. Puliafito, *Performance and reliability analysis of computer systems: an example-based approach using the SHARPE software package*. Springer Science & Business Media, 2012. [Online]. Available: <https://doi.org/10.1007/978-1-4615-2367-3>
- [105] A. Sari and M. Psarakis, “Scrubbing-based SEU mitigation approach for systems-on-programmable-chips,” in *International Conference on Field-Programmable Technology (FPT)*, Dec 2011, pp. 1–8. [Online]. Available: <https://doi.org/10.1109/fpt.2011.6132703>
- [106] —, “A fault injection platform for the analysis of soft error effects in FPGA soft processors,” in *IEEE International Symposium on Design and Diagnostics of*

- Electronic Circuits Systems (DDECS)*, April 2016, pp. 1–6. [Online]. Available: <https://doi.org/10.1109/ddecs.2016.7482459>
- [107] A. Sari, D. Agiakatsikas, and M. Psarakis, “A soft error vulnerability analysis framework for Xilinx FPGAs,” in *Proceedings of ACM/SIGDA International Symposium on Field-programmable Gate Arrays (FPGA)*. ACM, 2014, pp. 237–240. [Online]. Available: <https://doi.org/10.1145/2554688.2554767>
- [108] A. Sari and M. Psarakis, “A flexible fault injection platform for the analysis of the symptoms of soft errors in FPGA soft processors,” *Journal of Circuits, Systems and Computers*, vol. 26, no. 08, pp. 1740009:1–21, 2017. [Online]. Available: <https://doi.org/10.1142/S0218126617400096>
- [109] E. Seedhouse, *SpaceX: Making commercial spaceflight a reality*. Springer Science & Business Media, 2013. [Online]. Available: <https://doi.org/10.1007/978-1-4614-5514-1>
- [110] —, *Dragon at the International Space Station*. Cham: Springer International Publishing, 2016, pp. 45–62. [Online]. Available: https://doi.org/10.1007/978-3-319-21515-0_4
- [111] A. Shastri, G. Stitt, and E. Riccio, “A scheduling and binding heuristic for high-level synthesis of fault-tolerant FPGA applications,” in *IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2015, pp. 202–209. [Online]. Available: <https://doi.org/10.1109/asap.2015.7245735>
- [112] M. L. Shooman, *Reliability of computer systems and networks: Fault tolerance, analysis, and design*. John Wiley & Sons, 2003.
- [113] F. Siegle, T. Vladimirova, J. Ilstad, and O. Emam, “Availability analysis for satellite data processing systems based on SRAM FPGAs,” *IEEE Transactions on Aerospace and Electronic Systems (TAES)*, vol. 52, no. 3, pp. 977–989, June 2016. [Online]. Available: <https://doi.org/10.1109/taes.2016.140914>
- [114] F. Siegle, “Fault detection, isolation and recovery schemes for spaceborne reconfigurable FPGA-based systems,” Ph.D. dissertation, University of Leicester – Department of Engineering, 2016. [Online]. Available: <http://hdl.handle.net/2381/37521>
- [115] F. Siegle, T. Vladimirova, J. Ilstad, and O. Emam, “Mitigation of radiation effects in SRAM-based FPGAs for space applications,” *ACM Computing*

- Surveys*, vol. 47, no. 2, pp. 37:1–37:34, Jan. 2015. [Online]. Available: <https://doi.org/10.1145/2671181>
- [116] L. Sterpone, M. Aguirre, J. Tombs, and H. Guzmán-Miranda, “On the design of tunable fault tolerant circuits on SRAM-based FPGAs for safety critical applications,” in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '08. New York, NY, USA: ACM, 2008, pp. 336–341. [Online]. Available: <https://doi.org/10.1109/date.2008.4484702>
 - [117] L. Sterpone and M. Violante, “A new analytical approach to estimate the effects of SEUs in TMR architectures implemented through SRAM-based FPGAs,” *IEEE Transactions on Nuclear Science (TNS)*, vol. 52, no. 6, pp. 2217–2223, Dec 2005. [Online]. Available: <https://doi.org/10.1109/TNS.2005.860745>
 - [118] —, “A new reliability-oriented place and route algorithm for SRAM-based FPGAs,” *IEEE Transactions on Computers*, no. 6, pp. 732–744, 2006. [Online]. Available: <https://doi.org/10.1109/tc.2006.82>
 - [119] A. G. Stoddard, “Configuration scrubbing architectures for high-reliability FPGA systems,” Master’s thesis, Brigham Young University – Department of Electrical and Computer Engineering, Provo, UT, USA, 2015. [Online]. Available: <https://scholarsarchive.byu.edu/etd/5704>
 - [120] M. Straka, J. Kastil, Z. Kotasek, and L. Miculka, “Fault tolerant system design and SEU injection based testing,” *Microprocessors and Microsystems*, vol. 37, no. 2, pp. 155 – 173, 2013. [Online]. Available: <https://doi.org/10.1016/j.micpro.2012.09.006>
 - [121] Synopsis Inc., *FPGA design solution for high-reliability applications*, 2015.
 - [122] J. Tonfat, F. Kastensmidt, and R. Reis, “Energy efficient frame-level redundancy scrubbing technique for SRAM-based FPGAs,” in *NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, June 2015, pp. 1–8. [Online]. Available: <https://doi.org/10.1109/AHS.2015.7231160>
 - [123] R. Trautner, “ESA’s roadmap for next generation payload data processors,” in *Data Systems In Aerospace (DASIA) Conference*, 2011, pp. 1–5.
 - [124] S. M. S. Trimberger, “Three ages of FPGAs: A retrospective on the first thirty years of FPGA technology,” *IEEE Solid-State Circuits Magazine*, vol. 10, no. 2, pp. 16–29, 2018. [Online]. Available: <https://doi.org/10.1109/mssc.2018.2822862>

- [125] A. Tylka, J. Adams, P. Boberg, B. Brownstein, W. Dietrich, E. Flueckiger, E. Petersen, M. Shea, D. Smart, and E. Smith, “CREME96: A revision of the cosmic ray effects on micro-electronics code,” *IEEE Transactions on Nuclear Science (TNS)*, vol. 44, no. 6, pp. 2150–2160, Dec 1997. [Online]. Available: <https://doi.org/10.1109/tns.2012.2218831>
- [126] A. Ullah and L. Sterpone, “Recovery time and fault tolerance improvement for circuits mapped on SRAM-based FPGAs,” *Journal of Electronic Testing*, vol. 30, no. 4, pp. 425–442, Aug 2014. [Online]. Available: <https://doi.org/10.1007/s10836-014-5463-7>
- [127] L. van Harten, R. Jordans, and H. Pourshaghghi, “Necessity of fault tolerance techniques in Xilinx Kintex-7 FPGA devices for space missions: A case study,” in *Euromicro Conference on Digital System Design (DSD)*, Aug 2017, pp. 299–306. [Online]. Available: <https://doi.org/10.1109/dsd.2017.45>
- [128] A. Vance, *Elon Musk, Tesla, SpaceX, and the quest for a fantastic future*. Editions Eyrolles, 2017.
- [129] A. Vavousis, A. Apostolakis, and M. Psarakis, “A fault tolerant approach for FPGA embedded processors based on runtime partial reconfiguration,” *Journal of Electronic Testing*, vol. 29, no. 6, pp. 805–823, Dec 2013. [Online]. Available: <https://doi.org/10.1007/s10836-013-5420-x>
- [130] F. Veljković, T. Riesgo, and E. de la Torre, “Adaptive reconfigurable voting for enhanced reliability in medium-grained fault tolerant architectures,” in *NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, June 2015, pp. 1–8. [Online]. Available: <https://doi.org/10.1109/AHS.2015.7231165>
- [131] J. Wallmark and S. Marcus, “Minimum size and maximum packing density of nonredundant semiconductor devices,” *Proceedings of the Institute of Radio Engineers (IRE)*, vol. 50, no. 3, pp. 286–298, March 1962. [Online]. Available: <https://doi.org/10.1109/jrproc.1962.288321>
- [132] Z. Wang, L. Ding, Z. Yao, H. Guo, H. Zhou, and M. Lv, “The reliability and availability analysis of SEU mitigation techniques in SRAM-based FPGAs,” in *European Conference on Radiation and Its Effects on Components and Systems*, Sept 2009, pp. 497–503. [Online]. Available: <https://doi.org/10.1109/RADECS.2009.5994702>
- [133] D. White, *Considerations surrounding single event effects in FPGAs, ASICs, and processors white paper v1.0.1 (WP402)*, Xilinx Inc., 2012.

- [134] D. Wilson, A. Shastri, and G. Stitt, “A high-level synthesis scheduling and binding heuristic for FPGA fault tolerance,” *International journal of reconfigurable computing*, vol. 2017, pp. 1–17, 2017. [Online]. Available: <https://doi.org/10.1155/2017/5419767>
- [135] M. Wirthlin, “FPGAs operating in a radiation environment: Lessons learned from FPGAs in space,” *Journal of Instrumentation*, vol. 8, no. 02, pp. 1–8, 2013. [Online]. Available: <https://doi.org/10.1088/1748-0221/8/02/c02020>
- [136] —, “High-reliability FPGA-based systems: Space, high-energy physics, and beyond,” *Proceedings of the IEEE*, vol. 103, no. 3, pp. 379–389, 2015. [Online]. Available: <https://doi.org/10.1109/jproc.2015.2404212>
- [137] M. Wirthlin, H. Takai, and A. Harding, “Soft error rate estimations of the Kintex-7 FPGA within the ATLAS Liquid Argon (LAr) calorimeter,” *Journal of Instrumentation*, vol. 9, no. 01, p. C01025, 2014. [Online]. Available: <https://doi.org/10.1088/1748-0221/9/01/c01025>
- [138] Xilinx Inc., *Correcting single-event upsets in Virtex-II platform FPGA configuration memory application note v1.1. (XAPP779)*, 2007.
- [139] —, *Configuration readback capture in UltraScale FPGAs application note v1.1 (XAPP1230)*, 2015.
- [140] —, *AXI HWICAP LogiCORE IP product guide v3.0 (PG134)*, 2016.
- [141] —, *Xilinx TMR tool user guide v3.1.2 (UG156)*, 2017.
- [142] —, *7 series FPGAs configuration user guide v1.13.1 (UG470)*, 2018.
- [143] —, *Device reliability report user guide v10.9 (UG116)*, 2018.
- [144] —, *Radiation-hardened, space-grade Virtex-5QV family data sheet: overview data sheet v1.6 (DS192)*, 2018.
- [145] —, *Soft error mitigation controller product guide v4.1 (PG036)*, April 2018.
- [146] —, *Versal: The first Adaptive Compute Acceleration Platform (ACAP) white paper v1.0 (WP505)*, 2018.
- [147] —, *Vivado design suite user guide v2018.1 (UG909): Partial Reconfiguration*, 2018.

- [148] Z. Zhao, “Reconfiguration control networks for FPGA-based space applications,” Master’s thesis, University of New South Wales (UNSW) – School of Computer Science and Engineering, Sydney, Australia, 2016. [Online]. Available: <http://handle.unsw.edu.au/1959.4/56974>