Novel Hardware Verification Methods for FPGAs

Nieuwe hardwareverificatiemethoden voor FPGA's

Alexandra Kourfali

Promotor: prof. dr. ir. D. Stroobandt
Proefschrift ingediend tot het behalen van de graad van
Doctor in de ingenieurswetenschappen: computerwetenschappen

UNIVERSITEIT
GENT

Vakgroep Elektronica en Informatiesystemen
Voorzitter: prof. dr. ir. K. De Bosschere
Faculteit Ingenieurswetenschappen en Architectuur
Academiejaar 2018 - 2019

ISBN 978-94-6355-252-3
NUR 958, 959
Wettelijk depot: D/2019/10.500/60

Examination Commission

Prof. Dr.	Filip De Turck, Chairman Department of Information technology - INTEC Faculty of Engineering and Architecture Ghent University, Belgium
Prof. Dr.	Ingrid Moerman, Secretary Department of Information technology - INTEC Faculty of Engineering and Architecture Ghent University, Belgium
Prof. Dr.	Fernanda Lima Kastensmidt Computer Science & Microelectronics Program - PGMICRO Faculty Instituto de Informatica Federal University of Rio Grande do Sul, Brazil
Prof. Dr.	Dionisios Pnevmatikatos Department of Electronic & Computer Engineering - ECE Faculty of Electrical & Computer Engineering Technical University of Crete, Greece
Prof. Dr.	Nele Mentens Department of Electrical Engineering - ESAT Faculty of Engineering Technology Katholieke Universiteit Leuven, Belgium
Prof. Dr.	Joni Dambre Department of Electronics and Information Systems - ELIS Faculty of Engineering and Architecture Ghent University, Belgium
Prof. Dr.	Lieven Eeckhout Department of Electronics and Information Systems - ELIS Faculty of Engineering and Architecture Ghent University, Belgium
Em. Prof. Dr.	Erik D'Hollander Department of Electronics and Information Systems - ELIS Faculty of Engineering and Architecture Ghent University, Belgium

Ghent University
Faculty of Engineering and Architecture
Electronics and Information Systems department

M. Eng. Alexandra Kourfali

Tel.: +32-9-264.34.25
Fax.: +32-9-264.35.94
Email: Alexandra.Kourfali@UGent.be

Advisor: Prof. Dr. Dirk Stroobandt

Ghent University
Faculty of Engineering and Architecture
Electronics and Information Systems (ELIS) department
Computer Systems Lab (CSL)
Hardware and Embedded System (HES) research group
iGent, Technologiepark - Zwijnaarde 126, B-9052 Gent, Belgium

Tel.: +32-9-264.34.01
Fax.: +32-9-264.35.94
Email: Dirk.Stroobandt@UGent.be

This work was supported by a IWT/FWO fundamental research doctoral grant, by a HiPEAC collaboration grant and by the European Commission in the context of the FP7 FASTER project (#287804) and the H2020 EXTRA project (#671653).

Acknowledgements

It was my lifelong dream to be a scientist and pursue a PhD. Now that this journey is almost complete, I would like to thank everybody that helped me reach this goal.

First, I would like to thank my advisor *Prof. Dirk Stroobandt*, for all of the support that he provided me during all these years. He always gave me freedom and flexibility to pursue my academic interests, to explore new ideas and to create my own research path. This gave me the confidence to pursue my own research and to slowly grow as an independent researcher. I would like to thank him for giving me amazing professional opportunities, including the possibility to present my work in Silicon Valley companies and patenting my research.

I would also like to thank the members of my examination committee for taking the time to read and evaluate my thesis, giving excellent feedback, that contributed to significantly improve my dissertation. I would like to thank them for the fruitful and engaging discussions during the internal defense, that further improved my work. Special thanks to *Prof. Dionisios Pnevmatikatos* of TUC in Greece, for traveling to Ghent, and to *Prof. Fernanda Kastensmidt* of FURGS for traveling from Brazil to serve in my internal defense and for giving me constructive feedback in the conferences we both attended.

This thesis would not have been the same without *David Merodio Codinachs* from the European Space Agency. I would like to thank him for hosting me in the microelectronics group at ESA during the academic year 2015-2016, for his willingness to help advance my work and for expanding my knowledge. This invaluable experience gave me a unique insight on reliability and space applications, and resulted in multiple contributions in this thesis.

I would like also to thank the consortium of the H2020 EXTRA project that provided engaging discussions on my work and excellent collaboration. A special thanks to *Prof. Michael Huebner* and *Florian Fricke* for hosting me at RUB in June 2018 and collaborating with me. I would like to thank my former and current colleagues, and especially *Amit* and *Poona* for their support and friendship, *Dries, Yun* and *Vijay* for taking time to review my presentations. I would also like to thank my colleagues at the *Design Project* that we co-tutored, for the nice moments and the knowledge they helped me gain on entrepreneurship.

Science is built on the shoulders of giants, but a Ph.D. is built on the shoulders of colleagues, friends and family. Therefore, this thesis would not have been the same without their constant support.

I would like to deeply thank all my friends in Ghent and especially *Anastasia, Marios, Konstantinos, Ola, Vasilis, Thanos, Maria*, the "Zwijnaardians" *Giorgia, Francesco, Chiara, Domenico*, the many "amici y amigos" *Vince, Kaat, Raffa, Eva, Ursula, Olga* and so many more for giving me nice moments, alongside excellent Belgian beers! Thank you, *Vincent*, for the hundreds of cups of coffee & discussion, and for being there for me since I arrived in Belgium. I would also like to thank my *long distance friends* that are either spread around the world *(Thanos, Kostis, Giorgos, Nikos)* or in Greece *(Thodoris, Sakis, Alexis, Giannis, Kiki, Iro)* for keeping tabs on me, visiting me in Ghent and always supporting me.

Infinite thanks to *Domenico*, my partner in crime! He came in my life a while back, bringing joy and happiness! Thank you for the amazing moments, for all the advice you gave me and for supporting me during immensely difficult times and impossible decisions. I know that this last period was challenging, but you helped me gain enough confidence and perseverance to reach to the finish line. I could not be more grateful for all the love and the intellectual and emotional support you are giving me, for all the moments that we laugh out loud in random places, and for enjoying the little things in life! The best is yet to come! Grazie mille!

I would like to deeply thank my amazing sister *Giota*. She has always supported me and given me amazing memories and precious moments, as she never shies away from a new adventure (with me, her and Bella)! She has always been there for me, no matter the distance. Words cannot describe how important her emotional support has been during my whole life. I will be forever grateful that she chose to come to Ghent for her master studies and to finally live in the same city with me, after a decade. I cannot stress enough about the amazing moments that I had in Gent with Giota and Domenico. Without them, this Ph.D. would have been if not impossible, definitely less enjoyable!

I am very grateful to my awesome family, my mom *Anastasia* and my dad *Aristotelis*, that has redefined the term of unconditional support. They have been encouraging me constantly to pursue my goals and dreams. I would like to thank my dad that always supported my choices, learned to use apps to communicate with me abroad and he started learning English to be able to be a part of my new life; and my grandpa *Panagiotis*, that in his eighties, started using Skype to keep in touch with me. I would especially like to thank my tech-savvy mom that raised me as a feminist, gave me the confidence to be independent, to value education above all and to chase my STEM dreams and never stopped supporting me.

Γιώτα, μαμά. μπαμπά, παππού, οι λέξεις δεν μπορουν να περιγράψουν το πόσο σας ευχαριστώ για την απεριόριστη ηθική και συναισθηματική υποστήριξη που μου προσφέρατε όλα αυτά τα χρόνια, από την στιγμή που ξεκίνησα τις σπουδές μου στον Βόλο, μέχρι και σήμερα.

Ghent, June 16, 2019
Alexandra Kourfali

Samenvatting

Omdat de **Wet van Moore** het aantal transistoren in geïntegreerde circuits (IC) blijft opdrijven door het naar beneden schalen van de transistordimensies, worden steeds complexere digitale circuits gerealiseerd. De voortdurende technologische evolutie leidt tot steeds complexere architecturen. Het **verifiëren en valideren** van deze ontwerpen is een steeds moeilijkere taak geworden. Bovendien bevatten meerdere ontwerpen veiligheids-kritische eigenschappen en moeten ze voldoen aan specifieke veiligheidsstandaarden. Dit maakt van **IC-betrouwbaarheid** een fundamenteel aandachtspunt in het ontwerpproces.

Er bestaan verschillende **hardware-verificatietechnieken**, zoals formele verificatie, simulatie, foutinjectie, debugging, fouten vermijden, online test, enz. Ze worden gebruikt naargelang de vereisten van elk ontwerp en in verschillende stadia van de ontwerpcyclus. Er is een afweging tussen snelheid, ontwerpscomplexiteit en vereiste foutdekkingsgraad. Deze afweging bepaalt de verificatiemethodologie en het te volgen verificatieplan voor elk ontwerp. Verificatie, en zeker debugging, zijn essentieel om de functionele correctheid van het volledige systeem te garanderen. **Pre-Silicium-Verificatie** slaat op de activiteiten vooraleer de silicium-chip beschikbaar is. Deze processen bevatten o.a. het testen van circuits in een virtuele omgeving met simulatie en formele verificatie.

Historisch gezien is de meest gebruikte verificatietechniek simulatie- gebaseerd, door zijn flexibiliteit en gebruiksgemak. Simulatie-gebaseerde technieken kunnen echter de toegenomen complexiteit van ontwerpen niet aan, zeker nu de processorfrequentie niet langer schaalt. Bovendien ontsnappen vele fouten aan pre-silicium-verificatie en kunnen ze pas na de creatie van de eerste silicium-chip gevonden worden. In **post-silicium** debuggen hebben ontwerpers snel toegang tot fouten omdat de testen direct op de hardware uitgevoerd worden. Tests lopen op échte schakelingen op ware snelheid op échte systeemborden. De industrie is dus overgeschakeld naar post-silicium-validatie, door gebruik van prototypering en **FPGA-emulatie**.

Een **Field Programmable Gate Array (FPGA)** is een programmeerbare digitale elektronische chip. In FPGA-ontwerp moet de ontwerper de functie bepalen door het implementeren van een digitaal circuit op de beschikbare FPGA-blokken. De FPGA kan geherprogrammeerd (geherconfigureerd) worden door het wijzigen van de configuratiedata die de FPGA-functionaliteit bepaalt. Deze data kunnen aangepast worden naargelang de noden van de gebruiker. Door partiële herconfiguratie (PR) kunnen we een deel van het configuratiegeheugen configureren tijdens de looptijd via de interne configuratietoegangspoort (ICAP).

Onderzoekers van de HES-groep hebben een techniek ontwikkeld, genaamd **geparameteriseerde herconfiguratie**, om een geparameteriseerd ontwerp te creëren en te implementeren. Een ontwerp wordt geparameteriseerd genoemd als sommige ingangspoorten minder frequent van waarde veranderen dan andere ingangspoorten. In plaats van het implementeren van deze ingangen (parameters) als reguliere ingangen, worden ze geïmplementeerd als constante ingangen en het ontwerp wordt geoptimaliseerd naar deze constanten. Voor elke verandering in de parameterwaarden wordt het ontwerp geheroptimaliseerd (gespecialiseerd) tijdens de looptijd en geïmplementeerd door het geoptimaliseerde ontwerp te herconfigureren voor een nieuwe set van paramaters. De bitstromen van het geparameteriseerde ontwerp worden voorgesteld als **Boolese functies** van de parameters. Voor elke verandering in de parameters die af en toe optreedt, wordt een gespecialiseerde FPGA-configuratie gegenereerd door het evalueren van de overeenkomstige Boolese functies en de FPGA wordt geherconfigureerd met de gespecialiseerde configuratie.

In dit doctoraatswerk stel ik innovatieve technieken en tools voor om **geïntegreerde betrouwbaarheid en verificatie** aan te bieden in FPGA's met aangepaste geïnte-greerde componenten. Verder onderzoek ik methoden die de betrouwbaarheid en de interne observeerbaarheid van zowel commerciële als academische FPGA's verhogen tijdens het debuggen. De doel-FPGA's kunnen één of twee FPGA-lagen aan.

Eerst introduceer ik een methode om efficiënte **debugging** te introduceren in elk ontwerp. Door gebruik te maken van de PConf-techniek, voeg ik virtuele lagen toe die de debug-functionaliteit integreren in het ontwerp. Doordat de lagen virtueel zijn, is de impact op het oppervlaktegebruik minimaal. Er is ook de garantie van virtuele verbindingen tussen signalen en traceergeheugens. Vervolgens rangschik ik de interne signalen gebaseerd op klassificatiecriteria om de optimale set van signalen te vinden die toelaat sneller fouten op te sporen.

Daarna onderzoek ik hoe een **on-silicium debug-infrastructuur** geïntroduceerd kan worden in een FPGA met minimale invloed op de oppervlakte. Ik start met het bestuderen van de rol van FPGA-debug-structuren op **Virtual-Coarse-Grained Reconfigurable Arrays (VCGRAs)**. Ik gebruik twee verschillende technieken gebaseerd op de doel-FPGA-architectuur en ik creëer de Superimposed Debugging Architecture (SDA) die geïntegreerd is in de VCGRA.

Na deze twee hoofdstukken ga ik over tot het tweede deel van de thesis, dat focusseert op ontwerpmethoden om de **betrouwbaarheid** van **Commercial-Off-The-Shelf (COTS)** FPGA's te verhogen. Eerst stel ik twee methoden voor die toelaten het configuratiegeheugen te **herschrijven** gebaseerd op microherconfiguratie. Nadien stel ik een eigen FPGA-structuur voor (herconfiguratiecontroller) om de betrouwbaarheid te verhogen en de belangrijkste overhead (herconfiguratietijd) te verkleinen voor het **vermijden van zachte fouten (of soft errors)**. Deze structuur reduceert niet alleen de herschrijftijd maar heeft ook een versie die fouttolerant is en de algemene tolerantie verhoogt van COTS FPGA's tegenover stralingseffecten.

Daarnaast stel ik ook een fouttolerantieschema voor toekomstige zeer betrouw-

bare toepassingen in stralingsomgevingen voor met meerdere niveaus van **foutmitigatie**. Ik gebruik de meerlagige VCGRA-architectuur en zijn natuurlijke weerstand tegen stralingseffecten. Ik pas een spatiale redundantiemethode (TMR) toe en integreer de twee eigen herschrijfmechanismen. Daarna breid ik de PConf-methode uit, alsook de VCGRA tool-flow om een meer complete CAD flow te verkrijgen. Dit werk levert snelle herschrijfmethodes op met minder FPGA-middelen en heeft als doel een geïntegreerd foutmitigatiesysteem te bouwen dat de betrouwbaarheid van COTS- FPGA's verhoogt. Ik verifieer ten slotte de haalbaarheid van deze methoden in een foutinjectieprogramma.

Als laatste focusseer ik op **foutinjectie**. Ik heb twee varianten van foutinjectiemethoden ontworpen die gebruikt kunnen worden op de gekozen FPGA-architectuur. Ik deel ze in in twee types: de partieel geparameteriseerde foutinjectie (enkel LUT's) en de volledig geparameteriseerde foutinjectie (TLUT's en TCON's), afhankeijk van de gebruikte graad van parameterisatie. Het foutinjectieschema is geïntroduceerd in schakelingen op poortniveau. Dit geeft ons het voordeel dat we de technieken kunnen toepassen op een cluster van toepassingen die op een gelijkaardige manier gemodelleerd kunnen worden. In deze doctoraatsstudie kunnen de voorgestelde methoden gebruikt worden om ofwel permanente fouten te injecteren (voor testen post-silicium) of voor het injecteren van zachte fouten (soft errors) voor veiligheidskritische systemen. Deze techniek vermindert de oppervlakte-overhead van het met fouten geïnjecteerd ontwerp drastisch.

Summary

As **Moore's Law** continues to drive the number of transistors inside integrated circuits (IC) higher, alongside the scaling down of transistor dimensions, more complex digital designs are being realized. Continuous technology evolution has led to increasingly more complex architectures and designs. **Verifying and validating** these designs has become an increasingly difficult task. Additionally, multiple designs contain a safety-critical feature, and are in need to comply with specific safety-critical standards, making IC **reliability** a fundamental concern in the design and manufacturing process.

Various **hardware verification** techniques exist, such as formal verification, simulation, fault injection, debugging, fault mitigation, online test, etc. They can be used based on the requirements of each design and at different stages in the design flow. There is a trade-off between speed, design complexity and required fault coverage. This trade-off normally affects the verification methodology and plan that will be followed for each design. Verification, and especially debugging is essential to ensure the functional correctness of the entire system. **Pre-Silicon Verification** indicates the activities before the silicon chip is available. These processes include testing devices in a virtual environment with simulation and formal verification tools.

Historically, the most accepted verification method has been simulation-based, due to its flexibility and ease of use. However, simulation-based techniques are unable to handle the increasing complexity of the designs. As processor frequency scaling levels off, simulation-based techniques are unable to keep up with today's growing complexity. Additionally, many bugs escape pre-silicon verification, and can only be discovered after the creation of the first silicon. In **post-silicon** debug designers have early access to bugs, in high speeds, since tests are executed directly on the silicon. Tests occur on actual devices running at-speed in real-world system boards. Hence, the industry has shifted towards post-silicon validation, by using prototyping and **FPGA emulation**.

A **Field Programmable Gate Array (FPGA)** is a programmable digital electronic chip. In FPGA design the developer has to define its function through implementing a digital circuit on the FPGA resources. The FPGA can be reprogrammed (reconfigured) by changing the configuration data that defines the FPGA functionality. These data can be modified according to the user's needs. Partial Reconfiguration (PR) enables us to configure a part in the configuration memory during run-time, via the Internal Configuration Access Port (ICAP).

Researchers at the HES group have created a technique called **parameterized**

reconfiguration technique to create and implement a parameterized design. A design can become parameterized if some of its input values change less frequently than the rest. Instead of implementing these inputs (parameters) as regular inputs, they are implemented as constants, and the design is optimized for these constants. For every change in the parameter values, the design is re-optimized (specialized) during run-time and implemented by reconfiguring the optimized design for a new set of parameters. The bitstreams of the parameterized design are expressed as **Boolean functions** of the parameters. For every infrequent change in parameters, a specialized FPGA configuration is generated by evaluating the corresponding Boolean functions, and the FPGA is reconfigured with the specialized configuration.

In this dissertation I propose innovative techniques and tools to provide **integrated reliability and verification** in FPGAs that have integrated custom components. Furthermore, I investigate methods that increase the reliability and the internal observability during debugging, in both commercial and academic FPGAs.

First, I propose a method to efficiently introduce **debugging** to any given design. By leveraging the Parameterized Configurations (PConf) technique I add a virtual overlay that integrates debugging functionality in a design. Since it is virtual, the impact on the area is minimal. A guarantee of virtual connections between signals and tracing memories is provided. Then, I rank internal signals based on classification criteria, to find optimal signal sets that will be able to trace the bugs faster.

After that, I investigate how an **on-silicon debugging** infrastructure can be introduced into an FPGA, in such a way that it introduces minimal area overhead. I start studying the role of FPGA debugging structures on **Virtual-Coarse-Grained Reconfigurable Arrays (VCGRAs)**. I use two different techniques based on the target FPGA architecture and I create the Superimposed Debugging Architecture (SDA) that is integrated in the VCGRA.

After these chapters, I move on to the second part of the thesis, that focuses on design methodologies that increase the **reliability** of **Commercial-Off-The-Shelf (COTS)** FPGAs. First, I propose two configuration memory **scrubbing** techniques that are based on a fine-grained form of reconfiguration used for the PConf, called microreconfiguration. Afterwards, I propose a custom FPGA structure (reconfiguration controller) to increase the reliability and reduce the main overhead (reconfiguration time) of **soft error mitigation**. This structure not only reduces the scrubbing time, but it also has a fault tolerant version, that aims to increase the overall tolerance of a COTS FPGA against radiation effects.

Additionally, I create a fault-tolerant scheme for future high-reliability applications in radiation environments, with multiple level **fault mitigation**. I leverage the multi-layer VCGRA architecture and its natural resilience against radiation effects. First, I apply a spatial redundancy method (TMR) and the two custom scrubbing mechanisms. Then, I extend the PConf and the VCGRA tool-flow to create a more complete CAD flow. This work provides fast scrubbing with less FPGA resources and it aims at building an integrated fault mitigation scheme that

enhances the reliability of COTS FPGAs. I then verify the feasibility of these methods with a fault injection campaign.

Finally, I focus on **fault injection**. I have designed two variants of fault injection, that can be used based on the underlying FPGA architecture. I classify them into two types: the partially parameterized fault injection, and fully parameterized fault injection depending on the level of parameterization used. The fault injection scheme is introduced in gate-level designs. This gives us the advantage of being able to apply the techniques for a cluster of applications, that can be modeled in a similar way. In this dissertation, the proposed method can be used either for injecting permanent faults (for post-silicon testing), or for soft errors (for safety-critical systems). This technique drastically reduces the area overhead of the fault injected design.

Table of Contents

List of Figures

List of Tables

List of Acronyms

A

ABFT	Algorithm-Based Fault Tolerance
AIG	And-Inverter- Graph
ASIC	Application-Specific Integrated Circuit
ATPG	Automatic Test Pattern Generation

B

BRAM	Block Random Access Memory

C

CAD	Computer Aided Design
CERN	European Organization for Nuclear Research
CGRA	Coarse-Grained Reconfigurable Array
CLB	Configurable Logic Blocks
CMS	Compact Muon Solenoid
CNN	Convolutional Neural Network
CPU	Central Processing Unit

D

DCS	Dynamic Circuit Specialization
DD	Displacement Damage
DMS	Discrete Microscrubbing
DSP	Digital Signal Processor

DUT	Design Under Test
DWC	Duplication with compare

E

EDAC	Error detection and correction
ELA	Embedded Logic Analyzers
ESA	European Space Agency
ESD	Electro-Static Discharge

F

FAR	Frame address register
FF	Flip-flops
FIFO	First In First Out
FPGA	Field Programmable Gate Array
FT-SRC	Fault- Tolerant Superimposed Reconfiguration Controller

G

GCR	Galactic Cosmic Rays
GEO	Geo-magnetic
GPU	Graphics Processing Unit

H

HDL	Hardware Description Language
HWICAP	Hardware Internal Configuration Access Port

I

IC	Integrated Circuit

ILA	Integrated Logic Analyzer
IOB	Input/Output Blocks
ISS	International Space Station

L

LFSR	Linear Feedback Shift Register
LHC	Large Hadron Collider
LUT	LookUp Table

M

MBU	Multi-bit upset
mCW	minimum Channel Width
MEO	Medium Earth Orbit
MPSoC	Multi-Processor System-on-Chip
MS	Microscrubbing
MUX	Multiplexer

P

PConf	Parameterized Configurations
PE	Processing Elements
PL	Programmable Logic
POR	Power-on-reset
PPC	Partial Parameterized Configuration
PS	Processing System

R

RAM	Random Access Memory
RPR	Reduced-Precision Redundancy
RPU	Real-time Processing Unit
RTL	Register-Transfer Level

S

SAA	South Atlantic anomaly
SB	Switch block
SCG	Specialized Configuration Generator
SDA	Superimposed Debugging Architecture
SEE	Single Event Effects
SEFI	Single Event Functional Interrupts
SEP	solar energetic particles
SET	Single Event Transient
SEU	Single Event Upset
SFT	Superimposed Fault- Tolerant
SMAP	SelectMAP
SRAM	Static Random-Access Memory
SRC	Superimposed Reconfiguration Controller

T

TC	Template Configuration
TID	Total Ionizing Dose
TLUT	Tuneable LUT
TMR	Triple Modular Redundancy
TPaR	Tunable Place and Route tool
TTM	Time-to-Market

V

VC	Virtual Channel
VCGRA	Virtual Coarse-Grained Reconfigurable Array
VHDL	Very-high-speed Hardware Description Language
VPR	Versatile Placement and Routing
VTR	Versatile To Routing

W

WL	Wire Length

Part I

Introduction

1 Introduction

In this chapter, the background on Integrated Circuits design and the digital design and verification flow is given. The distinction between different computing paradigms, such as CPUs, GPUs, ASICs and FPGAs is presented, as well as the importance of heterogeneous computing in modern digital electronic designs. Reconfigurability plays an important role in heterogeneous computing, hence, an overview of the different steps involved in the design and verification of reconfigurable designs is given. Next, the importance of verification is explained, as well as the most common verification methodologies, for reconfigurable architectures. Then, it is stated where this thesis stands for each of the mentioned matters. Finally, the overall structure of this dissertation is presented.

1.1 Design Flow of Digital Integrated Circuits

In 1965, Intel co-founder Gordon Moore observed that the number of transistors per square inch on integrated circuits (ICs) had doubled every 1-2 years since their invention, while their cost was halved. This later became known as Moore's law, and is the driving force in chip design since then. Although in recent years the doubling of installed transistors on silicon chips occurs closer to every 24-36 months instead of annually.

Transistors are the building blocks of ICs. Up to billions of interconnected transistors can form networks of Boolean gates, used to process digital signals and produce an output for the IC. ICs are currently the core of every electronic device,

from FM transmitters, to mobile phones, to nuclear power plants and rockets.

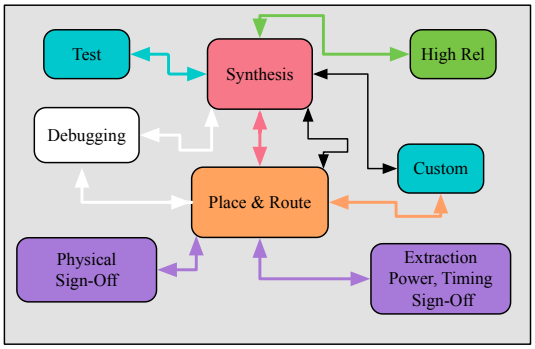
The increasing demand for performance from ICs pushes operation to higher signal bandwidths, while rapid advances in manufacturing capabilities have significantly reduced the feature size and increased the density of these devices. Due to the continuously shrinking transistor sizes, anomalies are caused in the design, that need contingency plans, as the new transistors impacts of power, thermal integrity, and reliability are becoming critical at ultra-low voltages. These contingency plans need to be available as soon as possible in the design of an IC, to avoid costly redesigns, reworks and other associated delays. Additionally, as the number of transistors inside ICs increases, thus allowing more complex digital designs to be realized, both verifying and validating these designs has become an increasingly difficult task [36].

These constant technological and scientific advancements are forcing a transformation of the traditional IC design flow as well. In the past, designers were assembling their own flows by cobbling together a set of disconnected tools from different vendors, as it is shown in Figure 1.1(a). The move away from such Frankenstein flows is already underway as vendors have rolled out first-generation platforms with loosely-coupled tools that feature connected and/or cloned engines through the flow. This is depicted in Figure 1.1(b).

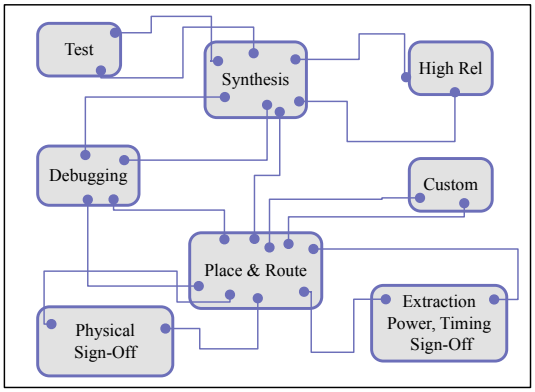
However, IC design is becoming unfeasible even with these first-generation platforms, because there are still clear lines of demarcation between the major phases of the design flow: Synthesis, Mapping, Place-and-Route and Sign-off. These functional boundaries inevitably cause major rework due to the loosely-connected tools, when transitioning from one design phase to the next, and from one verification procedure to the following. This makes for non-linear, non- monotonic, non-convergent progress that is far from optimal, with a direct impact on time-to-market (TTM) constraints.

Although these fragmented, loosely-connected, multiple-vendor flows were used in order to gain better consistency, they are still a sum of parts and only as good as the weakest link. Large numbers of iterations at the tool boundaries lead to poor full-flow time-to-results and unmet targets.

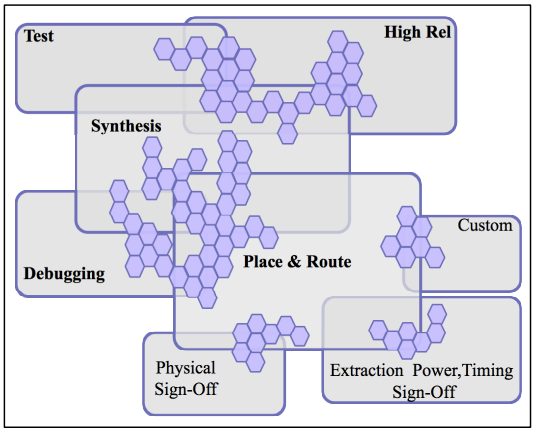
Nowadays, the need for a fusion of technologies across the entire spectrum of the IC design and verification flow has become evident. The technology boundaries that existed in the past must be revisited and redefined. The next-generation flow should have a backbone that assimilates and smooths out the boundaries leading to a monotonic, convergent flow that significantly accelerates the full design flow and provides various forms of verification and reliability. This is illustrated in Figure 1.1(c).



(a) Multi-vendor Flows



(b) 1st Generation Flows



(c) 2nd Generation Flows

Figure 1.1: Evolution of design and verification flow

1.1.1 Heterogeneous and Reconfigurable Computing

Additionally to Moore's law, which leads to the prediction that the number of on-chip cores of a processor doubles for every two years (and is already slowing down), stands the Dennard scaling [26]. According to Dennard scaling, the power density stays constant as transistor sizes become smaller. Therefore, the power usage remains in proportion with the area of the transistor [26, 106]. The Moore's law combined with Dennard scaling, means that performance per Watt grows at this same rate, doubling about every two years. However, it is not possible anymore to continue to ride the transistor count growth curve as per Moore's law, since the Dennard scaling on power density has failed for the technology nodes below 65 nm, causing the phenomenon of dark silicon: due to the failure of power density, the processors are unable to be powered-on at the same time.

In order to overcome this problem, heterogeneous computing has emerged, allowing to use transistors more efficiently. This enabled the creation of heterogeneous computing architectures, which can be broadly classified into two main categories, namely performance and functional heterogeneity.

Performance heterogeneous multi-cores share a common Instruction Set Architecture and they are further classified into static asymmetric multi-core (ARM big.LITTLE, Cortex-A7, Cortex-A15) and dynamic asymmetric multi-core (Bahurupi architecture).

Functional heterogeneous multi-cores allow for formerly discrete components to become integrated parts of Multi-Processor System-on-Chip (MPSoC) used in embedded products consisting of Central Processing Unit (CPU) cores, Graphics Processing Unit (GPU) cores, Real-time Processing Unit (RPU) cores, Digital Signal Processor (DSP) blocks and Programmable Logic (PL) accelerators. The PL is a Field Programmable Gate Array (FPGA), that is used to implement application specific specialized hardware.

In general, modern ICs can be divided in general purpose and application specific (ASIC). In the first case, a general-purpose processor provides full flexibility via hardware programming, but the performance and energy efficiency is much lower than in an ASIC. On the other end, an ASIC provides high performance and energy-efficient implementations with no flexibility. Reconfigurable computing is a computer architecture that combines some of the flexibility of software programming with the high-performance of hardware, by processing with very flexible high-speed computing fabrics like FPGAs [77]. Hence, reconfigurable computing fills the gap between the CPU flexibility and ASIC like performance.

1.1.2 Debugging and Reliability

Ensuring integrity in ICs begins early in the design process and continues throughout all aspects of manufacturing a device, releasing it, and its eventual end of life.

Therefore, it is always more efficient to make sure that the device is correctly designed, properly configured, and loaded with high-quality software before it is released, as even the smallest defect in software and hardware can compromise the entire system. Once a vulnerable piece of hardware is out in the market, there are real costs associated with service recalls or physical replacement. For a mobile device, the cost may be as simple as a software update or a full unit replacement. However, for an automotive recall, or for a high-reliability application, such as a satellite, a nuclear plant, or an aircraft, the costs of recall or replacement may be in the order of millions of euros. Furthermore, a malfunction can have devastating effects, including loss of lives and environmental disasters. It is therefore evident, that it is of utmost importance to verify and validate each IC, prior their product release.

Continuous technology evolution has led to increasingly more complex architectures, with a large amount of embedded memories, combined with the scaling down process of transistor dimensions (following Moore's law), power supply reduction, and operating speed increase. However, these constant advancements have compromised IC reliability. Reliability is defined as the probability of no failure in a given operating period. It is used to measure how good a system is and how frequently it goes down.

Various internal and environmental sources of noise and bombardment of particles of radiation led to an increase in the susceptibility of the circuit [74]. In particular, the probability of upsets (also called Single Event Effects (SEE)) due to radiation effects has increased significantly. SEE is the main concern in safety critical applications. It occurs when charged particles hit the silicon transferring enough energy in order to provoke a fault in the system. It can have a destructive or transient effect, according to the amount of energy deposited by the charged particles and the location of the strike in the IC. The main consequences of the transient effect, also called Single Event Upset (SEU), are bit flips in the memory elements.

This effect, in addition to the increased complexity of IC designs, have caused the convergence of two major fields to emerge in both ASICs and FPGAs: Debugging and Reliability. In the nanotechnology domain, reliability is a fundamental concern in the design and manufacturing process of IC circuits, whereas debugging is essential to ensure the functional correctness of the entire system. Thus, modern designs suffer from two basic problems at the same time: bugs and inevitable soft errors. Due to the increasing design size and complexity of IC circuits, the cost of IC verification and debug has significantly increased [121]. Additionally, the fabrication process is now approaching a point where it will be unfeasible to produce ICs that are free from SEEs. Therefore, more and more terrestrial applications that are determined as critical, such as data centers, bank and telecommunication servers, and automotive and avionics systems, are using fault-tolerant techniques

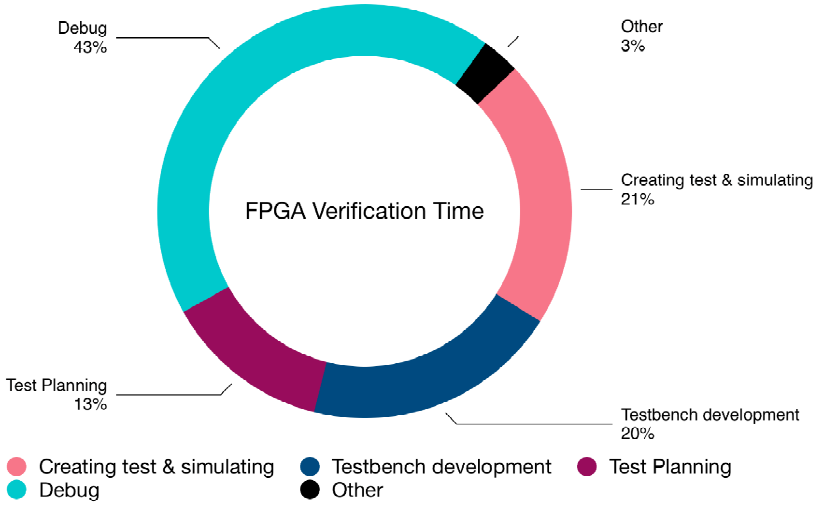


Figure 1.2: Time spend on different verification processes [36].

to ensure reliability.

1.2 Design and Verification Trends

The electronic industry continues to move towards larger designs. In fact, 31% of designs contain more than 80 million gates and 20% of the designs have more than 500 million gates. 72% of the designs have embedded processors and in FPGAs specifically, 59% of the designs have embedded processors, according to broad, vendor-independent studies of design verification practices around the world [36].

In order to be able to handle the increased design complexity, novel verification areas have emerged. Hence, additionally to the conventional functional verification domain, more requierements need to be verified, such as clock, power and security, as well as software. Regarding the security domain, up to 46% of the designs, contain a safety-critical feature, and are in need to comply with specific safety-critical standards, such as ISO26262, DO-254, IEC60601, etc. In order to comply with all the verification requirements, design engineers spend from 51% up to 80% of their time in verification [36].

In particular, debugging occupies the biggest portion during verification [36]. It is from 39% in ASIC/IC designs and 43% of the verification process in FPGA designs, as it is shown in Figure 1.2. However, up to 84% of FPGA design projects have non-trivial bugs that escape into production, and can manifest after the product is shipped.

For safety-critical designs, where the validation process is more detailed, in order to comply with the required standards, 34% of the IC designs require 3 or more spins, due to bugs. In spite of the rigorous testing and validation safety-critical ICs endure, only 25% of the designs have (non-trivial) bugs after production. These bugs that remain undetected and escape into production are mostly logic and functional. They are due to design errors or to changes in their specifications, as it is depicted in Figure 1.3.

Various verification techniques exist, such as simulation, emulation, prototyping, formal verification, etc. They can be used based on the requirements of each design and at different stages in the design flow, as is illustrated in Figure 1.4. There is a trade-off between speed, design complexity and required fault coverage. This trade-off normally affects the verification methodology and plan that will be followed for each design. Figure 1.4 illustrates the main verification plans and their stance regarding efficiency, coverage and speed.

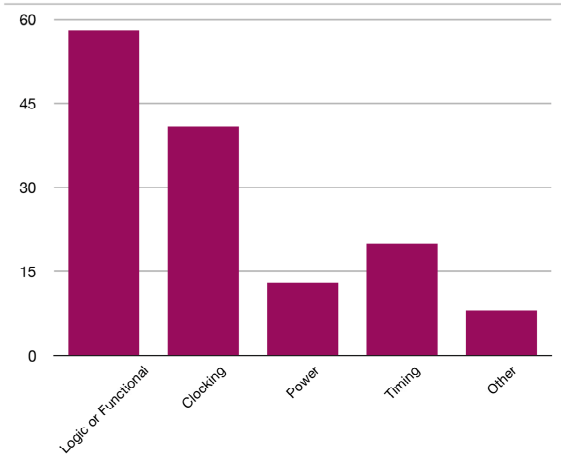
Historically, the most widespread verification method has been simulation, due to its flexibility and ease of use. However, simulation-based techniques are unable to handle the increasing complexity of the designs, especially when the designs under simulation include both software and embedded processors. Hence, acceleration techniques, such as FPGA prototyping and emulation are required for large designs (up to 80 million gates). By using FPGAs, the regression time (re-running functional and non-functional tests) is 9-17 hours, from multiple days required for an ASIC. Therefore, 35% of the industry has currently adopted emulation and 33% has adopted prototyping.

Emulation and FPGA-based prototyping are two technologies used for verifying complex hardware designs and validating systems with large software components. Architecturally, these tools are quite different, but they overlap in various their capabilities and applications.

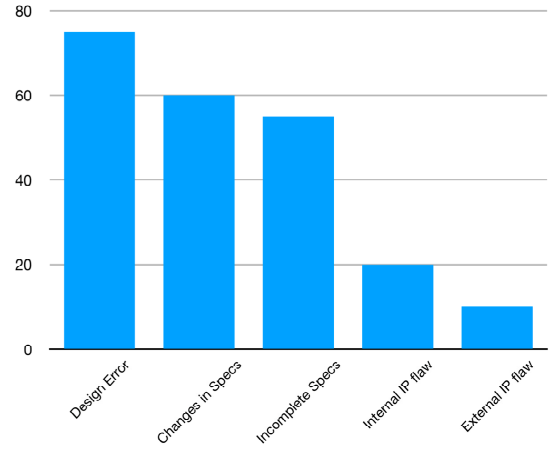
Emulators are recognized for their strong debug capabilities that are comparable to those of software simulators, they are flexible but they have a high cost. Additionally, emulators can reach speeds of up to 1 MHz and they often have to settle for 500 KHz. However, FPGA prototypes are regularly clocking in between 10 to 50 MHz, with some approaching 500 MHz (Xilinx Ultrascale) and they are low-cost, small and lightweight and can allow multiple platforms to be deployed, thereby accelerating overall performance. The largest drawback to FPGA-based prototypes is that it can take a long time to set up (up to two months or more). Moreover, FPGA prototypes can incur long FPGA place-and-route times when iterating the design, and are typically weaker than emulators at hardware debug.

Both techniques have some serious drawbacks, despite their speed and ease of use. Debug visibility is one of the biggest drawbacks of FPGA emulation and it is addressed in Part II (Chapters 3-4) of the thesis.

Another major setback is represented by the FPGA capacity issues, due to



(a) Errors contributing to FPGA rework



(b) Route cause of Logic and Functional errors

Figure 1.3: Route source and types of FPGA errors [36]

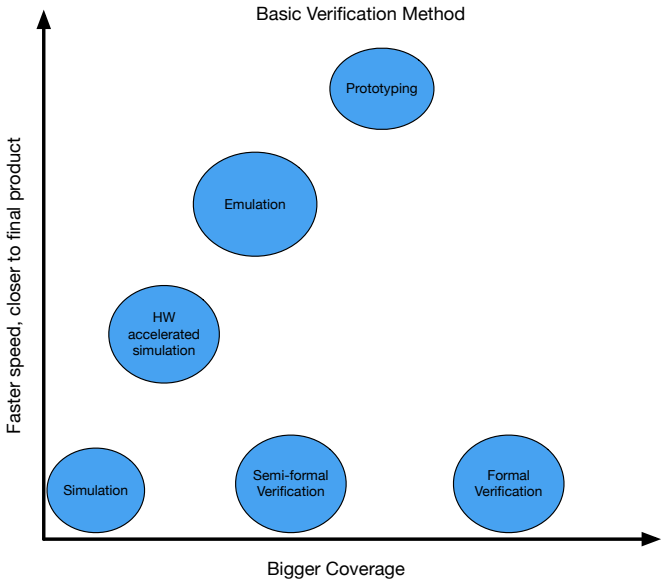


Figure 1.4: Verification Methods

hardware requirements that are essential to install the verification plan. This is addressed in Parts II and III. Finally, power, performance and area overheads are one of the biggest drawbacks of fault mitigation techniques for safety-critical applications. This is addressed in Part III (Chapters 5-7) in this thesis.

1.3 Design and Verification Methodology

A typical modern digital design is complex with multiple clock and reset domains, memory interfaces, specialized I/O, and integrates third party IP. Therefore, it is anticipated that the design and verification flow will be complex as well, where continuous advancements are needed to locate and fix bugs earlier in the design cycle while providing good results for performance and area to reduce system cost. A wide variety of errors can manifest in different steps in the design process. As the complexity and product requirements increase, designers need diverse flows that are able to handle different faults at various stages of the design process. A generic representation of a design flow with various integrated verification steps is depicted in Figure 1.5.

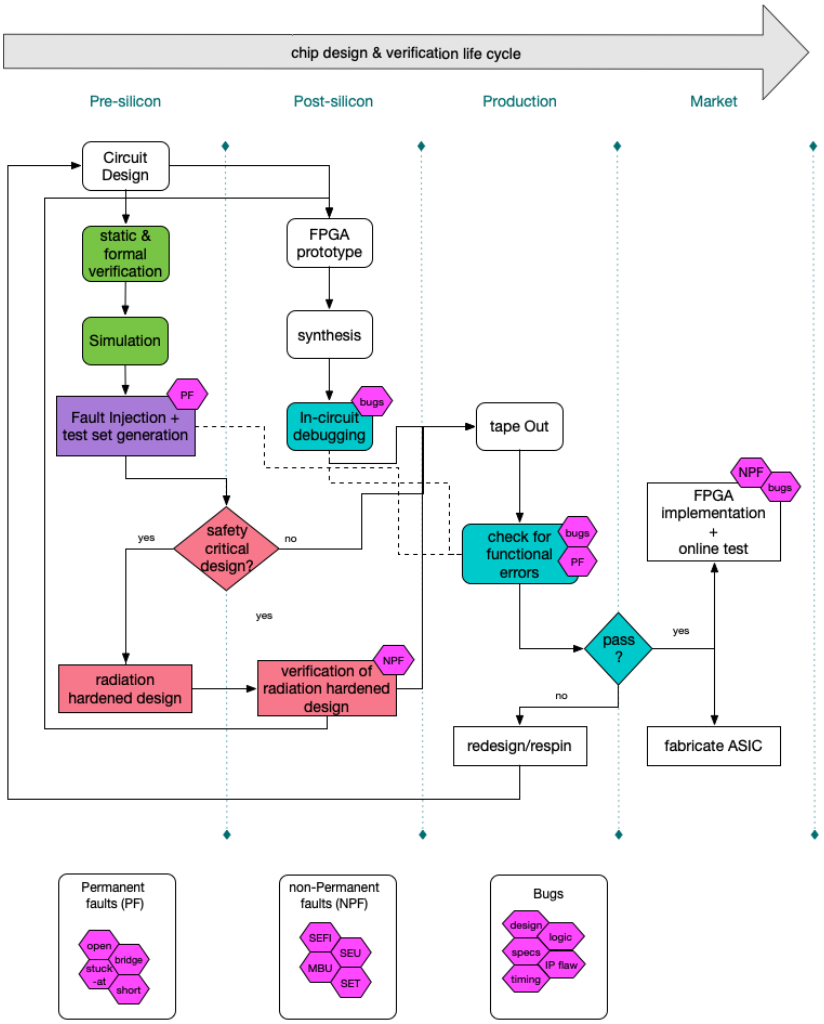


Figure 1.5: Design, verification and reliability flow

1.3.1 Pre-silicon Validation

Pre-Silicon Validation (or Functional verification) indicates the activities before the silicon chip is available. These processes include mainly formal verification and simulation.

Formal Verification

In formal verification the designer attempts to prove the correctness of the target design with respect to a certain formal specification or property, using formal methods of mathematics. There are various widely-known techniques for formal verification, including model checking, equivalence checking, static analysis, etc. These are tools aimed at verifying that the synthesis tool created a valid and equivalent netlist to the original Register-Transfer Level (RTL). With a comprehensive list of design rules, if these tools are employed correctly and each error reported by the tools is adequately verified this process should yield a 1 to 1 equivalence of the design to the resulting product in silicon. Except for equivalence checking tools, which are targeted at low-level design errors, formal verification tools do not currently scale well to large designs, as they often require exhaustive exploration of all the states of the design under test (DUT).

Simulation

Simulation-based verification is also a widely-used verification technique, that provides debugging at early stages of the design process. It offers full visibility and ease of use. However, simulation has not been able to keep up with the increasing design complexity, as exhaustive simulation of multi-million gate designs is extremely slow and often unfeasible. While simulators and software-based analysis tools have made significant progress over the years, these tools assume ideal, optimistic conditions. Therefore, due to the infinite number of potential design states in large designs, pre-silicon validation usually is unable to exhaustively check a design and is invariably incomplete.

1.3.2 Post-silicon validation

Post-silicon validation encompasses all that validation effort that is integrated in a system after the first few silicon prototypes become available, but before product release.

Debugging

Many bugs (mainly logical and functional) inevitably escape pre-silicon verification, and can only be discovered after the creation of the first silicon. In post-

silicon debug designers have early access to bugs, in high speeds, since tests are executed directly on the silicon. During the prototype debug phase of the design cycle in post-silicon debugging, design errors are discovered and isolated. The basic methodologies for in-circuit FPGA debugging are external test equipment, (oscilloscopes, external logic analyzers), embedded logic analyzers, FPGA-implemented scan chains, Readback and FPGA overlays.

During the pre-silicon process, engineers test devices in a virtual environment with simulation and formal verification tools. In contrast, post-silicon validation tests occur on actual devices running at-speed in real-world system boards using logic analyzers, assertion-based methods and external test equipment. Whereas simulation-based methods have nearly perfect internal observability, current commercially available silicon debug tools often offer limited visibility and impose significant overhead in terms of resource utilization. These drawbacks are addressed in Part II of the dissertation.

Fault Injection

Automatic Test Pattern Generation (ATPG) is a process where a test sequence is identified that can be later used to distinguish between the correct circuit behavior and the faulty circuit behavior caused by permanent faults. The generated patterns are used to test the DUT after fabrication, or to assist with determining the cause of an FPGA failure. The faults that can be identified are the typical fault modes, such as the stuck-at, open, bridging, short and delay faults. An ATPG is identified either with simulation-based techniques (PODEM, D algorithm, etc), or with FPGA-based techniques that facilitate fault injection. The first techniques can be inefficient for complex designs, while the latter can result in significant resource overhead. This is addressed in Part III (Chapter 7) of the dissertation.

Fault Injection is also a way to test the fault tolerance and the efficiency of an applied mitigation scheme. Fault injection can be done in two ways: either by physically injecting faults on a gate level or by manipulating the bitstream. The first methodology usually results in an extensive usage of the FPGA resources, as the faults are physically injected and thus the area is increased. In the second methodology it is essential to identify the exact bit locations in the bitstream that need to be flipped to execute a specific fault in the desired circuit element. By using fault injection, the system verification and validation engineers can construct test cases to guarantee that the complete system responds to SEUs as expected, and for testing the integration of the controller into larger system designs. This is addressed in Part III (Chapter 6) of the dissertation.

Fault Tolerance

Fault-tolerance on semiconductor devices has been researched since upsets due to radiation effects were first experienced in space applications several years ago. Since then, the interest in studying and advancing fault-tolerant techniques, to enhance the reliability of ICs in hostile environments has increased, driven by the need of radiation tolerant circuits in space missions, satellites, high-energy physics experiments and others. Based on the definition of fault-tolerance, the goal here is not to avoid the upsets, which by definition is not possible as they are caused by high-energy particles, but to maintain the IC's correct operation despite the existence of these upsets. Efficient fault-tolerant solutions are still a challenge for the semiconductor industry, especially because of the complexity of the new architectures.

Mitigation techniques to increase the fault-tolerance of the IC exist both in software-level and in hardware-level, as well as during pre-silicon and post-silicon validation. In pre-silicon mitigation techniques simulation prevails. At that stage, the system can be mitigated with a spatial redundancy technique before or after synthesis. Then, simulations can guarantee the design's correct operation. Although simulation is essential while designing a safety critical application to guarantee the reliability of the system, it is not possible to simulate the entire system. This is done during post-silicon validation, where a mitigated prototype design is more vigorously tested, after it is implemented on an FPGA. At this stage, more fault-tolerance techniques can be applied, besides spatial redundancy, such as configuration scrubbing. This is addressed on Part III in the dissertation.

1.4 Thesis Contribution

This PhD thesis proposes innovative techniques and tools to provide integrated reliability and verification in FPGAs that have integrated custom components. Furthermore, the reliability and debugging techniques are applied both in commercial and academic [1] FPGAs, that support (one or two) virtual FPGA overlays. All the techniques developed during my PhD research are described in the chapters of the thesis

- Introduce the proposed verification methodology
- Analyze the architecture used to support the verification strategy
- Outline the tools expanded to support the new methodology

[1] Academic FPGAs are architectures very similar to the commercial ones, but all their architecture and CAD tools are free and available to be used for academic purposes. [97].

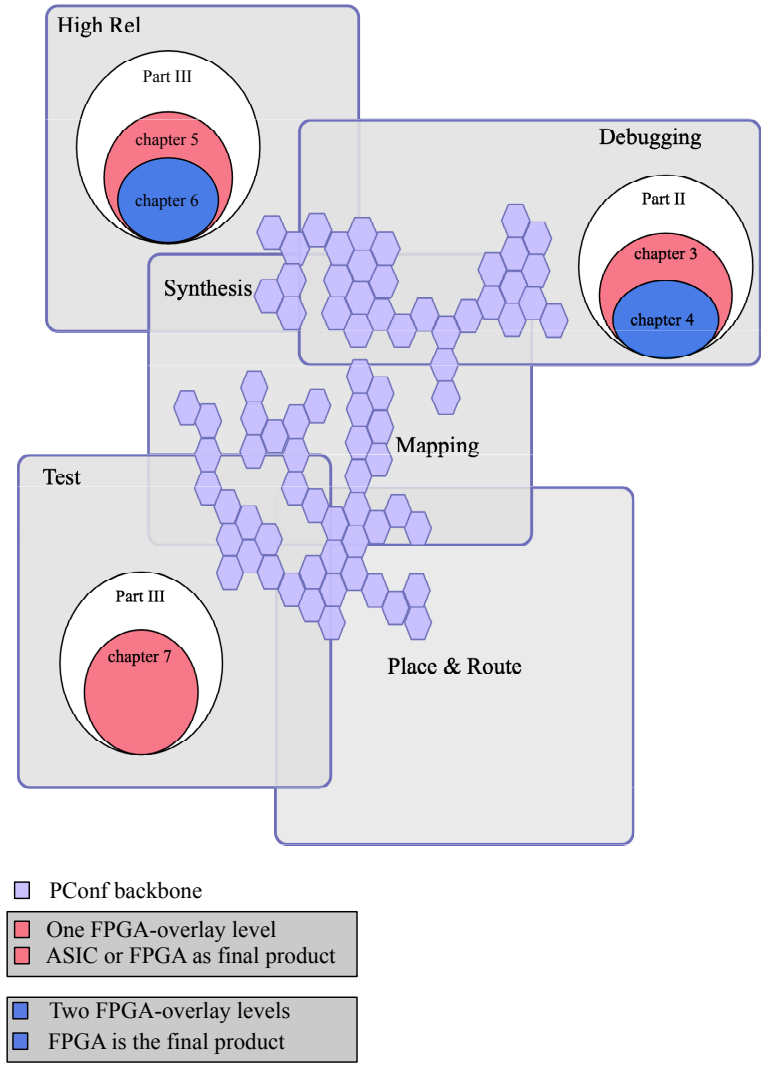


Figure 1.6: Thesis Contribution

- Discuss the overheads of the proposed approach and validate its accuracy, by comparing it with state-of-the-art tools and commercial or academic architectures.

In more detail, the thesis contributions are analyzed in five distinctive parts. The first part serves as an introduction and provides a background for the contributions. Parts II and III provide the main contribution of the thesis. Each part describes a verification process and the proposed techniques and tools. Part IV provides the conclusion and finally, an appendix gives a tutorial on one of the verification problems. A brief description of each chapter is presented here with their contribution in the overall research, as shown in Figure 1.6. It is illustrated how each chapter is contributing towards the three verification areas discussed in this thesis, namely debugging, testing and high-reliability. Additionally, it is shown whether each contribution targets FPGAs as a final product, as an intermediate step during ASIC design, or both.

1.5 Thesis Structure

The thesis structure is illustrated in Figure 1.7 and outlined below.

Part I: Introduction

Chapter 1

The current chapter describes the basics of design, reconfigurable computing and verification, as well as the thesis motivation and structure.

Chapter 2

Chapter 2 is used as a preface, to provide the basic background of the architecture of FPGAs. Here, the background analysis regarding FPGA architectures and the custom FPGA components needed to enable techniques that are used throughout this thesis are introduced, such as parameterized configurations, DCS and Virtual Coarse-Grained Reconfigurable Arrays (VCGRAs). For all the custom techniques, an overview is also given of their architecture, the tool flows needed to enable their functionality, and the impact of these techniques compared to traditional FPGA design.

Part II: In-Circuit Debugging

Chapter 3

In this chapter, a novel method for hardware verification is proposed, to facilitate debugging, to reduce the area and reconfiguration overhead and to increase

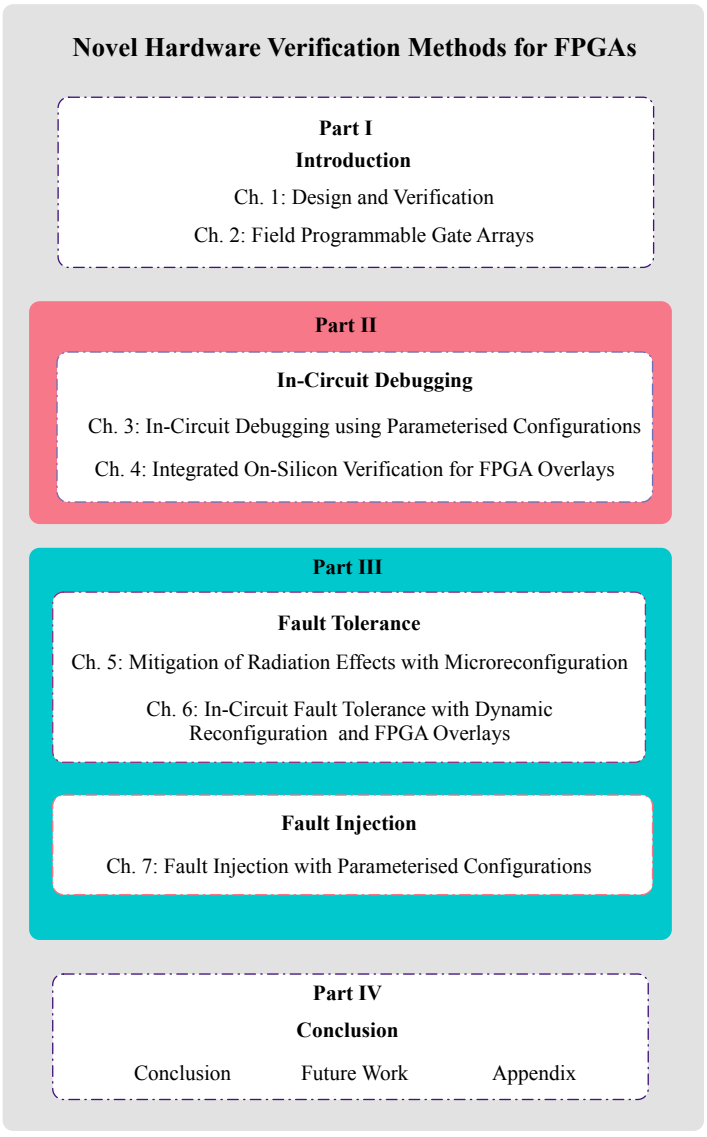


Figure 1.7: Thesis Structure

the internal signal observability. This technique can be either applied for FPGA prototyping during ASIC verification, or when FPGA is the final product. First, an in-circuit debugging with parameterized configurations approach is introduced. Then, a method to enable functionality that allows the reconfiguration of the basic logic blocks (LUTs) and the routing infrastructure is proposed, alongside internal signal classification techniques. The results of the system are presented and compared with commercial and custom debuggers. The area, time and power results and the trade-offs between internal signal observability and overhead are also explained.

Chapter 4

This chapter presents an approach to enhance the observability of VCGRA-based designs for functional debugging. The proposed technique is a custom-made in-circuit-debugger that can be used for debugging when the FPGA is the final product. It is installed during the initial recompilation and can be used to rapidly trace functional errors in high-performance computing applications, that can be implemented as FPGA overlays. This chapter describes first how the VCGRA is constructed and then how the proposed architecture can be integrated in a VCGRA, so that the latter can be debugged. In this chapter, the basics of debugging with FPGA overlays are also presented, alongside their characteristics.

Part III: Fault Tolerance

Chapter 5

This chapter analyses and classifies radiation effects and mitigation methods. It provides the basic reliability techniques for SRAM-based Commercial Off-the-Shelf (COTS) FPGAs. Then, it discusses the proposed technique on integrated fault tolerance for COTS FPGAs, with multiple-level configuration scrubbing. Two reconfiguration controllers are also proposed that support the novel technique. This approach can achieve reliable and low-cost configuration scrubbing for SRAM-based COTS FPGAs. The proposed techniques can be either applied during ASIC validation, or when a commercial FPGA is the final product and it requires mitigation.

Chapter 6

This chapter focuses on the major problem that reduces the fault tolerance of a design, the SEUs. It targets commercial-off-the-shelf FPGAs as the final product. Therefore, a Triple Modular Redundancy (TMR) method is proposed, for a virtual coarse-grained reconfigurable architecture, with an embedded on-demand fault-mitigation technique tailored for FPGA overlays. This method performs spatial

redundancy and run-time recovery and can achieve fault tolerance and fast scrubbing, with less resources than conventional tools, by providing integrated layers of fault mitigation. The technique first integrates into an FPGA overlay state-of-the-art mitigation technique and shows how VCGRAs can be leveraged to increase the reliability of a target design without compromises in area and time.

Chapter 7

This chapter discusses how the parameterized configurations method can be used to inject faults in a gate-level design, with very small area overhead. This technique can be either applied during ASIC testing, or when an FPGA is the final product and it requires testing for permanent faults and soft errors. It adds additional circuitry in order to model and apply permanent faults and soft errors without increasing the area requierements. This is done in two different methods, one for commercial FPGAS (partially parameterized) and one for academic FPGAs (fully parameterized).

Part IV: Conclusion

The concluding remarks and potential future directions of this research are presented in Chapter 8. Finally, an appendix that provides a theoretical background on the sources of radiation effects in microelectronics situates at the end of the thesis.

1.6 Publications

Journal Papers

- **Alexandra Kourfali**, Florian Fricke, Michael Huebner and Dirk Stroobandt, *Integrated On-Silicon Verification Method for FPGA Overlays*, Journal of Electronic Testing, Springer Nature, March 2019.

  (Contributes to Chapter 4)

- **Alexandra Kourfali** and Dirk Stroobandt, *In-Circuit Fault Tolerance for FPGAs using Dynamic Reconfiguration and Virtual Overlays*, Microelectronics Reliability, Elsevier, (accepted with minor revision).

  (Contributes to Chapters 5 and 6)

- **Alexandra Kourfali** and Dirk Stroobandt, *In-Circuit Debugging Methods with Dynamic Reconfiguration of FPGA Interconnects*, ACM Transactions on Reconfigurable Technology and Systems (TRETS), submitted.

  (Contributes to Chapter 3)

- Georgios Tzimpragos, Amit Majumdar, **Alexandra Kourfali**, Dirk Stroobandt, Hussain Al-Asaad and Tim Sherwood, *A Survey of Debugging Methods for FPGA-based Systems*, to be submitted.

  (Contributes to Chapter 3)

Patents

- **Alexandra Kourfali** and Dirk Stroobandt, *Integrated Circuit Verification using Parameterized Configuration*, US10295594B2, WO2015181389A3, patent granted.

  (Contributes to Chapters 3 and 7)

Conference Papers

- **Alexandra Kourfali** and Dirk Stroobandt, *Superimposed In-Circuit Debugging for Self-Healing FPGA Overlays*, IEEE Latin-American Test Symposium, 2018, Sao Paulo, Brazil

  (Contributes to Chapter 4)

- **Alexandra Kourfali**, Amit Kulkarni and Dirk Stroobandt, *SICTA: A Superimposed In-Circuit Fault Tolerant Architecture for SRAM-based FPGAs*, Proceedings of the 23rd IEEE International Symposium on On-Line Testing and Robust System Design, 2017, Thessaloniki, Greece.

  (Contributes to Chapter 5)

- **Alexandra Kourfali**, David Merodio Codinachs and Dirk Stroobandt, *Superimposed Fault Mitigation for Dynamically Reconfigurable FPGAs*, Radiations Effects on Components and Systems, RADECS 2017, CERN, Geneva, Switzerland

  (Contributes to Chapter 6)

- **Alexandra Kourfali** and Dirk Stroobandt, *Efficient hardware debugging using Parameterized FPGA Reconfiguration*, IEEE International Parallel and Distributed Processing Symposium Workshops, 2016, Chicago, USA.

  (Contributes to Chapter 3)

- **Alexandra Kourfali** and Dirk Stroobandt, *Test set generation almost for free using a Run-Time FPGA reconfiguration technique*, IEEE 16th Latin-American Test Symposium, 2015, Puerto Vallarta, Mexico.

  (Contributes to Chapter 7)

- **Alexandra Kourfali** and Dirk Stroobandt, *Parameterised FPGA reconfigurations for efficient test set generation*, IEEE Proceedings International Conference on Reconfigurable Computing and FPGAs, 2014, Cancun, Mexico

  (Contributes to Chapter 7)

- **Alexandra Kourfali**, Karel Bruneel and Dirk Stroobandt, *Pre-mapping Fault Injection in FPGA-based Parameterised Test Set Generation for ASIC Testing*, 23rd International Workshop on Logic & Synthesis, 2014, San Francisco, USA.

  (Contributes to Chapter 7)

- Vijaykumar Guddad, **Alexandra Kourfali** and Dirk Stroobandt, *VHDL Design Tool Flow For Portable FPGA Implementation*, (to be submitted).

Conference Posters / Short Papers

- **Alexandra Kourfali** and Dirk Stroobandt, *In-Circuit FPGA Debugging using Parameterised Reconfigurations* , 54th ACM/EDAC/IEEE Design Automation Conference (DAC'17), 2017, Austin, TX, USA

  (Contributes to Chapter 3)

- **Alexandra Kourfali**, *Post-Silicon Design Verification and Testing using Parameterised FPGA Reconfigurations*, 54th ACM/EDAC/IEEE Design Automation Conference (DAC'17), 2017, PhD Forum, Austin, TX, USA.

  (Contributes to Chapters 3-7)

- **Alexandra Kourfali**, David Merodio Codinachs and Dirk Stroobandt, *Enhancing the Reliability of COTS SRAM-based FPGAs with Microreconfiguration in Space Applications*, Military and Aerospace Programmable Logic Devices, MAPLD, 2016, San Diego, California, USA.

  (Contributes to Chapter 6)

- **Alexandra Kourfali** and Dirk Stroobandt, *Towards Efficient Hardware Debugging Using Parameterized FPGA Reconfiguration*, 52nd ACM/EDAC/IEEE Design Automation Conference (DAC'15), 2015, San Francisco, CA, USA.

  (Contributes to Chapter 3)

2

Field Programmable Gate Arrays

This chapter provides background on FPGA architectures and the custom FPGA components needed to enable techniques that are used throughout this thesis, such as Parameterized Configurations, Dynamic Circuit Specialization, and Virtual Coarse-Grained Reconfigurable Arrays. This chapter covers different aspects related to FPGAs and briefly discusses different FPGA technologies. First, an over-view of the FPGA architectures is presented. Details of basic FPGA logic blocks and different routing architectures are then described. After that, an overview of the different steps involved in the FPGA design flow is given and custom FPGA elements are presented. For all the custom FPGA elements used throughout this thesis, I give an overview of their architecture, the tool flows needed to enable their functionality, and their impact compared to traditional FPGA design.

2.1 FPGA Architecture

An FPGA is a semiconductor fabric that allows users to configure its functionality during run-time. FPGAs were first introduced almost two and a half decades ago. Since then they have seen a rapid growth and have become a popular implementation media for digital circuits. They are pre-fabricated silicon devices that can be electrically programmed in the field to become almost any kind of digital circuit or system. FPGAs are classified in SRAM-based, flash-based and anti-fuse devices.

SRAM cells are the basic cells used for SRAM-based FPGAs. Most commer-

cial vendors use static memory based programming technology in their devices. These devices use static memory cells which are distributed throughout the FPGA to provide the reconfiguration functionality. SRAM cells like the one shown in Figure 2.1 are used to program the routing interconnects and to program the Configurable Logic Blocks (CLBs) that are used to implement logic functions. SRAM-based FPGAs have become the dominant approach for FPGAs for a variety of applications, such as DSPs, data centers and in various environments, high-radiation, aerospace, nuclear, data centers.

FPGAs comprise of:

- Programmable logic blocks that implement logic functions.
- Programmable routing that connects these logic functions.
- I/O blocks that are connected to logic blocks through routing interconnects.

The main building blocks of FPGAs are CLBs, Input/Output Blocks (IOBs) and a routing network. Typically, an FPGA contains a grid of CLBs ranging from 10,000s to 100,000s (even millions in today's FPGAs) in number and hundreds to thousands of IOBs. The CLBs include LUTs and flip-flops (FF) that can be combined to realize any digital circuit.

The programmable logic and routing interconnect of FPGAs makes them flexible, low cost, general purpose and having faster TTM, but at the same time it makes them larger, slower and more power consuming than standard cell ASICs. Additionally, FPGAs need less than a minute to reconfigure and they cost less than ASICs (unless a very large number of he same ASICs are ordered). Moreover, many FPGAs offer partial reconfigurability, since a portion of the FPGA can be partially reconfigured while the rest of an FPGA is still running. This results in easy upgrades of any future updates in a final product, by simply downloading a new application bitstream. However, the main advantage of FPGAs i.e. flexibility is also the major cause of its draw back. These functionalities make FPGAs slower, larger and more power consuming compared to ASICs.

Two different styles of SRAM-based FPGA architectures exist. The first is the island style, that covers a basic version of the FPGA architectures and the column-based style. The island style architecture covers a general, basic version of the FPGA architectures, and the column-based style that is used in the modern commercial FPGA architectures.

2.1.1 Island-style FPGA architecture

Island style is the most commonly used architecture among academic FPGAs. It is called island-style architecture because in this architecture CLBs look like islands in a sea of routing interconnects. The CLBs include LUTs and flip-flops. Each

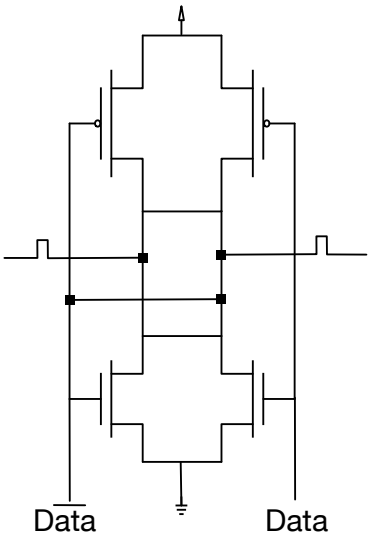


Figure 2.1: Static memory (SRAM) cell

LUT can implement any arbitrary Boolean function for the given inputs depending on the truth table entries stored in the configuration memory. The truth table entries of the LUT define the combinatorial logic for different values of the inputs. The flip-flops are used to store the output of a LUT and hence are used for implementing sequential logic. In this architecture, CLBs are arranged on a 2D grid and are interconnected by a programmable routing network and can realize any digital circuit. The routing network comprises of pre-fabricated wiring segments and programmable switches that are organized in horizontal and vertical routing channels. The network consists of multiple wires placed between the CLBs and form the routing channel width. In the routing network there are two primitives called Switch block (SB) and Connection block (CB). The IOBs on the periphery of the FPGA are also connected to the programmable routing network and to the external pins of the FPGA chip to establish communication with the external world, and therefore they are placed along the perimeter of the FPGA fabric. The Island-based FPGA architecture is depicted in Figure 2.2.

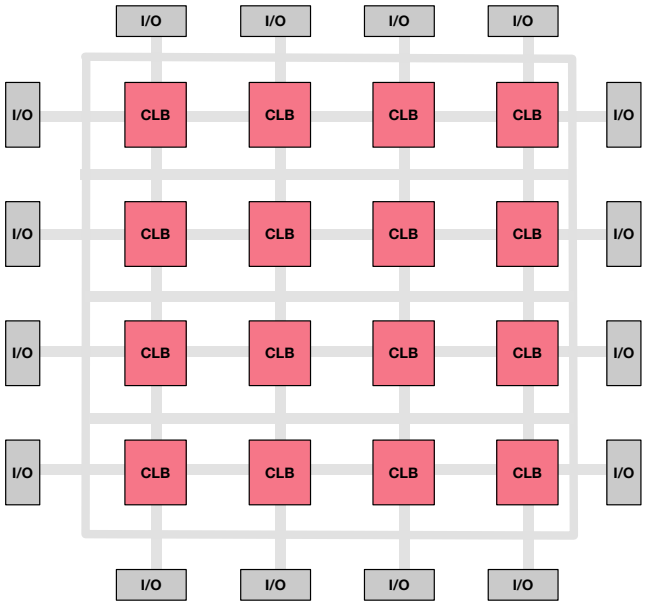


Figure 2.2: Island-style FPGA architecture

2.1.2 Column-style FPGA architecture

Today's commercial FPGA architectures can be best described as column style. They contain an array of CLBs which encapsulate LUTs, flip-flops and multiplexers. Each CLB contains 8 LUTs and is capable of realizing combinatorial and sequential logic. The array of CLBs is divided into a number of Clock Regions. Each clock region contains CLB columns with a fixed number of CLBs and the height of the CLB column remains the same in all the clock regions. There are multiple CLB columns adjacent to each other thus forming CLB rows. This is visualized in Figure 2.3.

There are other heterogeneous primitives available in the commercial FPGAs such as DSP columns (containing multiply accumulate operators), Block Random Access Memory (BRAM) columns, high speed IO protocols (PCIe, Gigabit Ethernet, etc.), clock management resources, ADC. Important primitives are embedded processors, such as ARM Cortex-A9, which is more powerful and efficient than the softcore processor (MicroBlaze) implemented on the programmable logic of the FPGA. These primitives help to meet the stringent performance requirements for a given application.

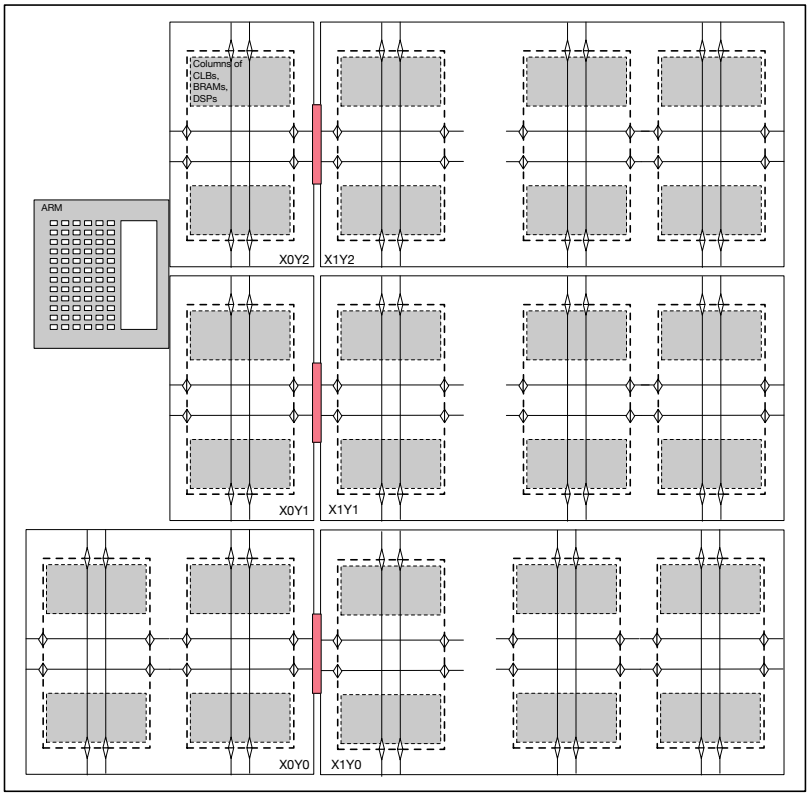


Figure 2.3: Column-style FPGA architecture

Configuration Bitstream

The FPGA configuration functionality is enabled via a bitstream, and it defines the hardware functionality. An FPGA comprises of an array of programmable logic blocks that are connected to each other through programmable interconnect network. Programmability in FPGAs is achieved through an underlying programming technology. The configuration bitstream is a stream of bits that set all the LUT values and the multiplexer selection bits and thus define the functionality of a digital circuit on an FPGA. It contains the configuration data and a set of commands that are used to orchestrate the programming of the FPGA.

FPGA Frame

An FPGA frame is the smallest addressable element of an FPGA configuration. In fact, the 7 series Xilinx FPGA configuration memory is arranged in frames that are

tiled about the device. Each frame consists of 101 32-bit words, as it is shown in Figure 2.4. Hence, each frame has identical length of 3,232 bits. A single frame can contain truth table entries of multiple LUTs which are located in a single CLB column. For every reconfiguration process, at least one frame has to be accessed via the HWICAP.

Configuration Interfaces

Configuration interfaces are ports that are used to load the configuration bitstream onto an FPGA. The basic interfaces are:

- The JTAG interface is a standard debugging port used by the external master. This port can be used for configuration and configuration read-back.
- The ICAP/ICAPE2 is a primitive providing the embedded processor access to the internal configuration of the FPGA. It is compatible with the external SelectMAP interface. The ICAP primitive connects to the Hardware Internal Configuration Access Port (HWICAP) driver with the bus so that it can be accessed by the embedded processor.
- The SelectMAP, Serial, SPI and BPI interfaces can reconfigure the FPGA during boot-up.
- PCAP: is a reconfiguration controller used for Partial Reconfiguration on the Zynq-SoC. It has a configuration interface similar to the ICAP, that is tightly coupled with the PS region of the Zynq-SoC. With the appropriate software drivers, the PCAP supports configuration read-back.

2.1.3 Other FPGA Architectures

SRAM-based FPGAs are the most widely used FPGAs. However, there are two other important FPGA programming technologies that are worth mentioning:

Flash-based FPGAs Flash-based FPGAs offer several advantages. First, they are non-volatile, which is a crucial requirement in high-reliability environments, as non-volatile FPGAs are immune to soft errors. Flash-based programming technology is also more area efficient than SRAM-based. However, unlike SRAM-based FPGAs, flash-based devices can not be reconfigured/reprogrammed an infinite number of times. This disadvantage places flash-based FPGAs out of the scope of this thesis.

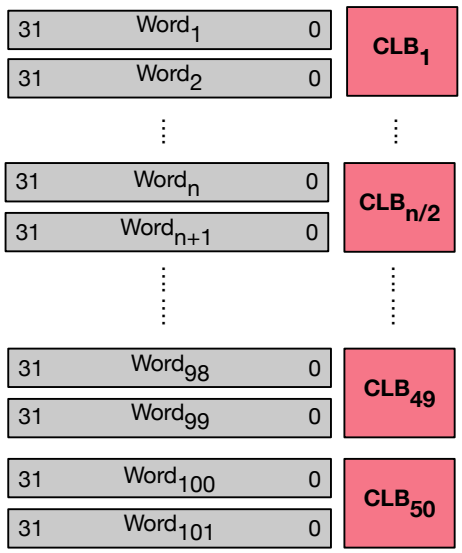


Figure 2.4: FPGA frame structure

Anti-fuse FPGAs An alternative to SRAM and flash-based technologies are the anti-fuse FPGAs. The main advantages of anti-fuse FPGAs is its low area, and their immunity against radiation effects. Also this technology has lower on resistance and parasitic capacitance and it is non-volatile. However, anti-fuse FPGAs cannot be reprogrammed more than one or two times, making them out of the scope of this thesis.

Ideally, one would like to have a programming technology which is reprogrammable, non-volatile, and that uses a standard CMOS process. Apparently, none of the above presented technologies satisfy these conditions.

SRAM-based programming technology is the most widely used programming technology. The main reason is its use of standard CMOS process and for this very reason, it is expected that this technology will continue to dominate the other two programming technologies. However, they are prone to soft-errors, due to their volatile nature, making flash-based and anti-fuse FPGAs a more preferable option for high-reliability environments. *Thus, one of the objectives of this work, is to create techniques that make volatile FPGAs as immune to soft errors as their non-volatile counterparts.*

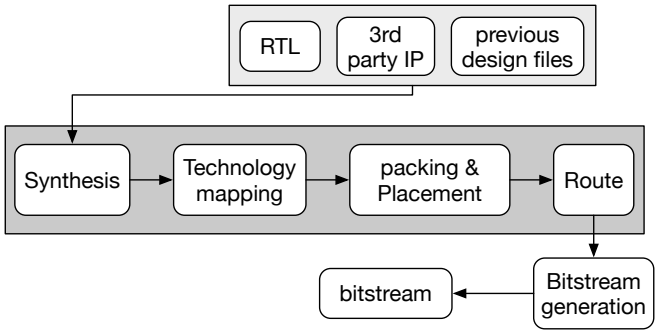


Figure 2.5: FPGA CAD Flow

2.2 FPGA CAD Tool Flow

A major aspect of FPGA architecture research is the development of Computer Aided Design (CAD) tools for mapping applications to FPGAs. It is well known that the quality of an FPGA is determined by the effectiveness of its accompanying CAD suite. The more optimized the accompanying tools, the more they exploit the features of the FPGA architecture. In general, the CAD tools take a design in a Hardware Description Language (HDL) or high-level language and they convert it into a bitstream to be loaded in the FPGA, that determines the logical function that the FPGA implements. This process is divided in five steps namely: synthesis, technology mapping, packing, placement and routing. The process is similar to the one used in ASIC design and is depicted in Figure 2.5.

2.2.1 Logic Synthesis

Logic synthesis transforms an HDL description (VHDL or Verilog) into a set of Boolean gates and Flip-Flops. The tool translates an HDL, Register-Transfer-Level (RTL), or high-level description of a design into a hierarchical Boolean network. The Boolean network of logic gates is represented by a graph, such as the And-Inverter- Graph (AIG) [53]. Each node in the graph represents a gate, flip-flop, primary input or primary output. Each edge in the graph represents a connection between two circuit elements. Afterwards various optimizations can be applied, such as a reduced number of logic gates, lower logic depth, etc.

2.2.2 Technology Mapping

During technology mapping, the tool tries to find a network of cells that implements the Boolean network. The library of cells is composed of k-input LUTs

and flip-flops. Therefore, this step involves mapping the Boolean network onto the available resources of the target FPGA. Other primitives, such as DSPs, BRAM blocks, etc. are directly inferred from the HDL, and hence they are not mapped during technology mapping. Technology mapping algorithms can optimize a design for a set of objectives including depth, area or power.

2.2.3 Placement

Placement algorithms determine which physical block within an FPGA should implement the corresponding logic block (instance) required by the circuit. Here the LUTs and flip-flops produced during the previous step, are clustered into CLBs, without any changes in the interconnections. This was a separate step, called packing. However in the last FPGA flows, it is merged with placement. Then, the packed CLBs are placed to specific blocks of the target FPGA. Extensive optimizations are applied, in order to minimize the wire length and the delay of the interconnects. This is one of the most time consuming steps during FPGA design.

2.2.4 Routing

The FPGA routing problem consists of assigning nets to routing resources such that no routing resource is shared by more than one net. It achieves that by configuring the physical switch blocks and connection blocks to achieve the required interconnect according to the circuit netlist. The routed netlist determines the critical path delay of the circuit. Therefore, routing is an iterative process that runs until it meets the given timing constraints [145]. For a large FPGA, the routing step can consume a huge amount of time.

2.2.5 Bitstream Generation

In the final step, a series of bits is generated that corresponds to the exact implementation of the routed netlist. The generated bitstream also consists of FPGA platform specific commands and settings that orchestrate the FPGA programming. The result of this step is a configuration bitstream.

2.3 Reconfiguration methods and techniques

A reconfigurable IC allows the user to change its functionality. Such systems ideally would require no user interventions while changing the functionality of the digital design. The FPGAs are configurable devices, since the digital implementation on the FPGA can be changed. Reconfiguration permits to trade-off between performance (speed and/or latency) and area (number of used primitives) of the

reconfigurable architecture. The process of changing the structure of a reconfigurable device at start-up-time (respectively run-time) is called (re)configuration, and it signifies the possibility to send new configurations to an FPGA.

Depending on the (re)configuration capability, FPGAs can be classified as depicted in Figure 2.6. FPGAs can be divided between one or two time configurable devices (flash-based) that can replace ASIC devices and reconfigurable FPGAs. The reconfigurable FPGAs are SRAM-based and can be classified into partially, dynamically reconfigurable and globally reconfigurable. Global reconfiguration is supported by previous Altera (now Intel) models. A complete FPGA configuration has to be swapped while performing global reconfiguration, which leads to a change in the internal state of the hardware and thus the FPGA has to restart its operation.

In partial reconfiguration, the user can change the function of part of the FPGA while other sections remain operational. Partial reconfiguration can be performed either statically by halting the operation of the application or dynamically, where the operation of the application can continue during the reconfiguration. Static reconfiguration is supported by Xilinx Spartan and NanoXplore models, whereas dynamic reconfiguration is supported by all recent Xilinx models (Kintex, Virtex 7-series).

2.3.1 Dynamic Partial Reconfiguration

The resources of a reconfigurable architecture can be reused by multiple modules over time. Only parts of a system might be updated while continuing operation of the remaining system. Dynamic Partial Reconfiguration [98] is the modification of a part of the FPGA's logic blocks, while the rest of the system remains active. The modification of the logic is achieved by downloading partial bitstreams. The application software triggers the reconfiguration when a set of conditions at a time are met. It triggers the reconfiguration by sending a reconfiguration request to the configuration manager. A reconfigurable system consists of an on-chip or off-chip CPU (PowerPC, ARM Cortex-A9 or MicroBlaze), a configuration database (SD card), a configuration interface (HWICAP) and the PL (FPGA's Programmable Logic). The configuration manager downloads an appropriate partial bitstream by fetching it from the configuration database. The partial bitstream is downloaded via a configuration interface called HWICAP, that can swap between partial bitstreams. Each partial bitstream represents a different partial reconfigurable module. Each module has to undergo all the conventional FPGA toolflow steps (Synthesis, Technology Mapping, Place and Route) to compile into a partial bitstreams. These modules are stored in the configuration database which is located in a memory. The memory can be internal in the FPGA (DRAM) or it can be external to the FPGA (SD card). This architecture is illustrated in the Figure 2.7.

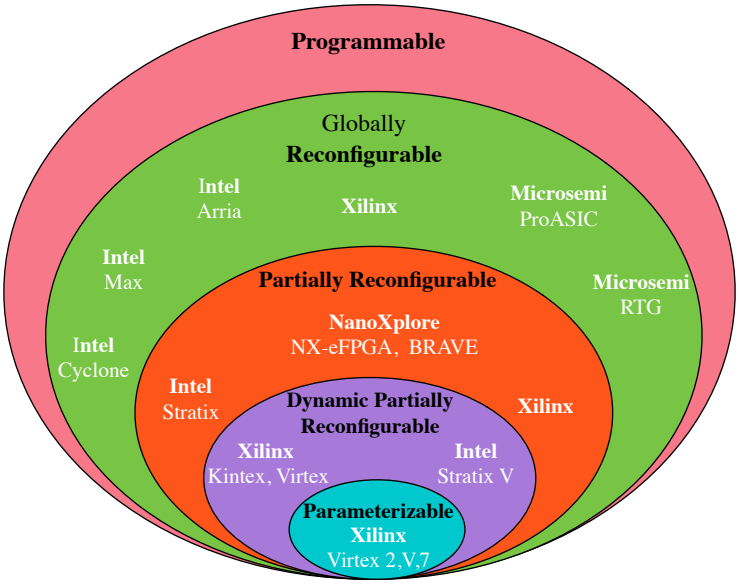


Figure 2.6: Classification of the main FPGA architectures based on their reconfigurability

2.3.2 Parameterized Reconfiguration

Parameterized Reconfiguration is suitable to implement parameterized applications. An application is said to be parameterized when some of its inputs, called parameters, are infrequently changing compared to the other inputs. Instead of implementing these parameter inputs as regular inputs, these inputs are implemented as constants, and the design is optimized for these constants. When the parameter values change, the design is re-optimized for the new constant values by reconfiguring the FPGA. A parameterized configuration contains bits that are not only static binary (0's and 1's) but also multi-valued Boolean functions of infrequently changing parameters. For specific parameter values, we can instantly derive specialized configurations by evaluating the Boolean functions for the given parameter values. This technique is also called Dynamic Circuit Specialization (DCS).

DCS is a technique used to optimize parts of a parameterized application and switch between the specialized parts for the current specific conditions utilizing Partial Reconfiguration (PR) at run-time. This technique improves the functional density (number of computations that can be performed per unit area and unit time) of the FPGA. [37, 150]. Figure 2.8 demonstrates this method. Parameterized reconfiguration is built on top of partial reconfiguration. Therefore, most parts in

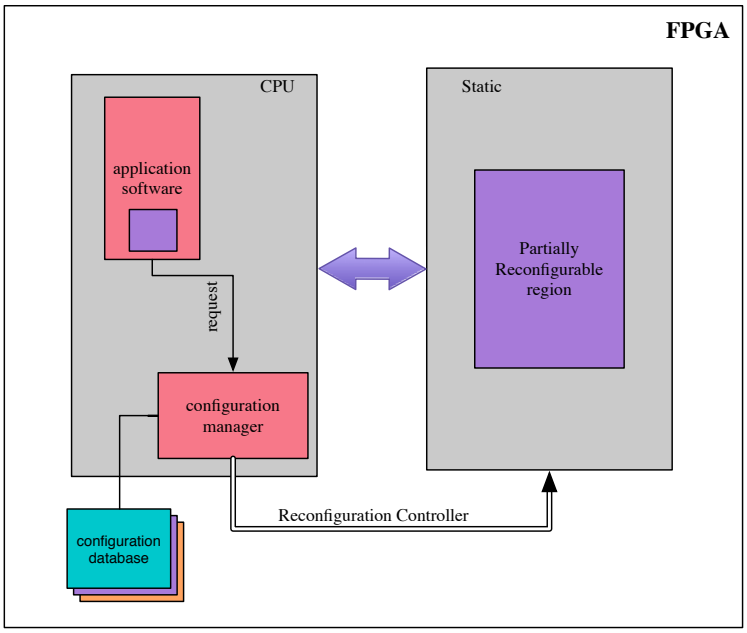


Figure 2.7: Dynamic Partial Reconfiguration System

the system remain the same. The application software running on the CPU constantly monitors the (parameterized) inputs. Once a change in parameter value is detected, the specialization is performed by the configuration manager by evaluating the Boolean functions (that are stored in the configuration database) for given parameter values, thus generating the specialized bitstreams. This is illustrated in Figure 2.9.

Tool Overview

The conventional FPGA tool flow cannot be used to generate parameterized configurations. Hence, an adapted version has been created, that is visualized in Figure 2.10.

Synthesis: During synthesis, the HDL design is converted into a network of logic gates. The infrequently changing inputs in the HDL design are annotated as parameters. This annotation distinguishes between parameter inputs and regular inputs. The parameter inputs are also a part of the Boolean network of logic gates produced after synthesis and are not treated differently in the synthesis step.

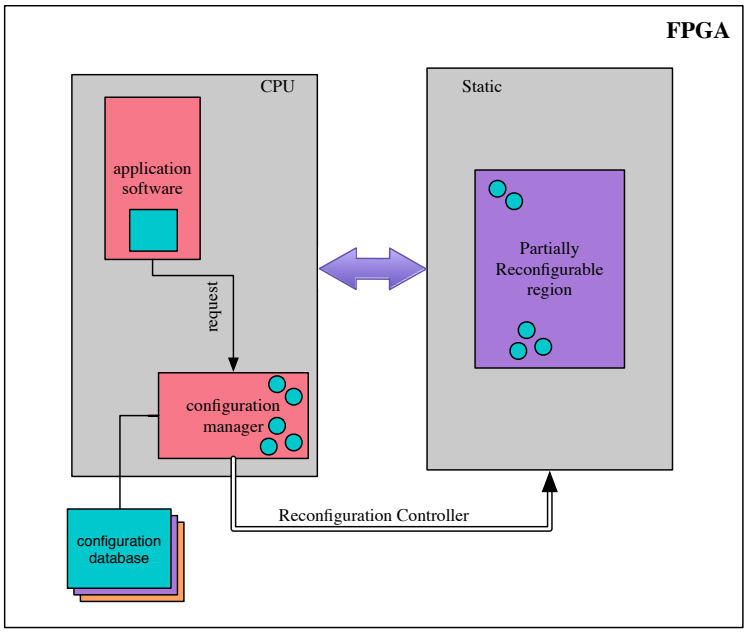


Figure 2.8: Parameterized Reconfiguration System

Hence, at this point any synthesis tool can be used, as long as it is able to handle the parameter annotations.

Technology Mapping: During technology mapping, the tool tries to find a network of cells that implements the Boolean network, in the same way as in conventional mapping. Here, the network is mapped in tuneable LUTs (TLUTs). These are virtual LUTs whose inputs are defined as the Boolean functions of the parameter inputs instead of ones and zeros. To generate a parameterized bitstream, authors of [55] changed the conventional mapping tool to a tuneable version, so that the Boolean functions of parameter inputs are mapped on to TLUTs or tuneable connections (TCONs). This process will eventually create the parameterized bitstream. This technology mapping algorithm can be integrated with the conventional Xilinx tool flow [12].

- TLUT: a virtual LUT of which the truth table entries are defined as Boolean functions of parameters instead of ones and zeros.
- TCON: a point-to-point connection which can be made or broken, based on

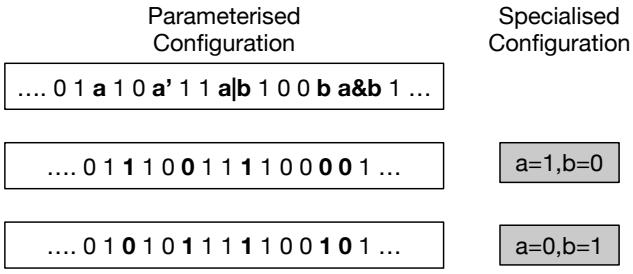


Figure 2.9: Parameterized configurations include Boolean functions of parameters and they are evaluated into specialized configurations

the value of a Boolean function of parameters. It is implemented using the FPGA's routing network. The parameterized configuration is derived from the Boolean function.

Placement and Routing As the parameters are already included in the LUT functionality through the Boolean functions, they are no longer present in the physical implementation of the netlist so the entire netlist can be placed and routed as if the parameters where not present. Therefore, a conventional placer and router can be used.

Bitstream Generation The final output of the generic stage is the Template Configuration (TC) and Partial Parameterized Configuration (PPC). TC is a static bitstream which contains static ones and zeros, which are used for configuring during the start of the FPGA. The PPC contains sets of Boolean functions of the parameter inputs. The PPC needs to undergo the specialization stage, along with parameter values to produce the specialized configuration. The specialization stage consists of a Specialized Configuration Generator (SCG), that takes the PPC and the parameter values as inputs and evaluates the Boolean functions of parameter inputs for given parameter values to produce a specialized configuration. During run time, the TLUTs are reconfigured by downloading the specialized configuration and thus accomplishing run time reconfiguration.

The SCG reconfigures the FPGA with the Internal Configuration Access Port (ICAP), that is a configuration interface. The ICAP swaps the specialized bitstreams into the FPGA configuration memory. Here, the HWICAP is used as a reconfiguration controller, encapsulates the ICAP primitive port of the FPGA and forms a controller that performs the swapping of specialized bitstreams via the ICAP. The bitstreams are accessed frame-by-frame. A frame is the smallest

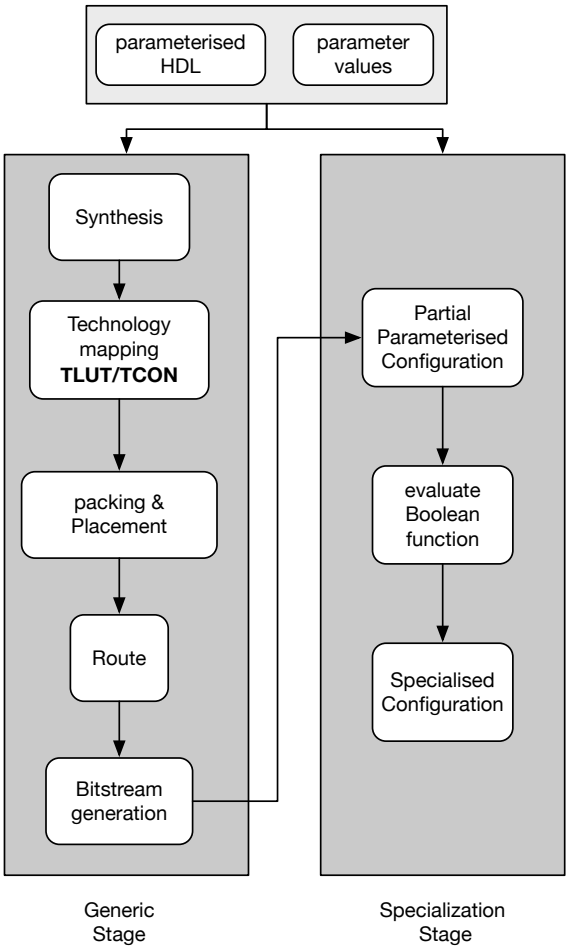


Figure 2.10: The two stage flow that supports parameterized configurations

addressable element of the FPGA configuration data. Each frame contains reconfiguration bits of tens of LUTs and has its unique frame address that can be used to point to the frame during the reconfiguration.

2.3.3 Micro-reconfiguration

Micro-reconfiguration is a technique to change the configuration of few FPGA resources frame-by-frame. The designer has to microreconfigure the frames that contain truth table entries of the TLUT that is implemented on this physical LUT. It is a fine-grained form of reconfiguration used for DCS.

A supporting reconfiguration controller reads, modifies and writes-back the frames from the configuration memory and a processor takes care of executing the cycle of read, modify and write-back of frames. The reconfiguration controller supports a software driver function called *XhwIcap_setClb_bits* to perform the reconfiguration. The performance of DCS is entirely dependent on the trade-off between the benefits and the costs of micro-reconfiguration. The changes in the FPGA architecture directly influence the micro-reconfiguration costs [87].

2.4 FPGA Overlay Architectures

The FPGA reconfiguration can be beneficially done by introducing an intermediate layer for the reconfiguration infrastructure. This intermediate layer exists in overlay architectures. Because we will use such architectures, we first describe them here. An FPGA overlay is an additional virtual architectural layer that is conceptually located between the user application and the underlying physical FPGA, as it is shown in Figure 2.11. It overlays on top of the physical FPGA configurable fabric and can carry out a variety of predefined computations [132]. With this additional layer, the user application is no longer implemented onto the physical FPGA directly, but in the overlay architecture, regardless of what the physical FPGA may be. An additional step is required, to translate the overlay architecture to the physical FPGA. The overlay is a *virtual* architecture because its features (multiplexers, LUTs, I/Os) may not necessarily be present in the physical FPGA. However, if the overlay is designed properly, any design that is mapped on the virtual overlay, can be executed unmodified on the actual FPGA without knowing its details.

The FPGA overlay can carry out certain computations. An FPGA Overlay may be designed as a virtual FPGA, a processor, a GPU, or as a (virtual) coarse-grained reconfigurable array (CGRA). The virtual overlay architectures provide more flexibility. The designers should be able to debug their virtual architecture designs in-system, without detailed knowledge of the hardware's limitations. However, none of the mentioned techniques can currently support efficient debugging of virtual architectures without extensive resource overhead or time-consuming recompilation.

2.4.1 FPGA Overlay Types

FPGA overlays are classified in three types, that are presented below:

Virtual FPGAs

Virtual FPGAs are built virtually on top of the commercial FPGA fabrics. The virtual FPGA overlays have a different set of configuration bits and features than the commercial FPGAs. Therefore, having a virtual FPGA layer over an FPGA fabric

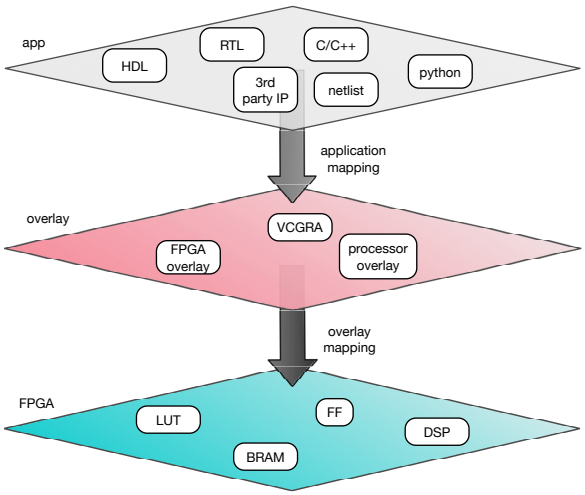


Figure 2.11: Overlay architectures

improves the application portability and compatibility. The virtual architectures proposed in [25, 48, 78, 99] are examples for virtual FPGAs.

Benefits of FPGA Overlays

FPGA overlays have three major benefits regarding designers' productivity:

- Significant reduction of the design space, by constraining the flexibility of an FPGA.
- Significant reduction of the CAD flow's run time.
- Significant reduction of the design completion time, by using models more familiar to software designers.

Therefore, FPGA overlays provide a trade-off between performance of the hardware and the flexibility of a software program for compute intensive parts of the applications.

Processor-like Overlays

A second category of overlays are processor-like overlays, that are using processor-like designs as an intermediate layer. Here, the goal is the usability of the overlay from a user's perspective. These overlays feature a high degree of control and provide ample data parallelism to make them suitable for FPGA accelerations.

Hence, customized soft-processors fall under this category. Examples of these overlays are presented in [69, 70, 128, 157].

CGRAs and VCGRAs

FPGA overlays are not limited to making virtual FPGAs only. On the contrary, many researchers have demonstrated the benefits of overlays that implement entirely different computing architectures such as coarse-grained reconfigurable arrays (CGRAs), by taking advantage of FPGA's general-purpose configurable fabric. The CGRAs are expensive to produce for low-volume products, as they are ASICs. FPGAs are relatively cheaper for low volume products but they are not so easily programmable. The combination of CGRAs and FPGAs results in implementing a Virtual Coarse-Grained Reconfigurable Array (VCGRA) on top of the fine-grained FPGA fabric. VCGRAs improve the power and the overall performance of the system by trading off design flexibility. Additionally, they can reduce the configuration granularity of an FPGA from its physical fine-grained configurable fabric such as LUTs to one with coarser reconfiguration granularity. VCGRAs can consist of small arithmetic blocks (adders, multipliers, DSP blocks), up to very complex microprocessors connected with a sophisticated network-on-chip.

VCGRAs can bridge the gap between FPGA implementations and high-level application descriptions, as with VCGRAs the time consuming design cycle of the FPGA (synthesis, mapping, place and route) can be moved forward to pre-compilation times. Hence, the entire development cycle is reduced, as (re)compilation is avoided for the (re)construction of the VCGRA. VCGRAs have been proposed before, either as optimized architectures, as a solution for long compilation times, or as a facilitator for high-level synthesis [25, 127]. This is achieved mainly because VCGRAs allow the designer to write the code in a higher abstraction level language, without requiring knowledge of the underlying hardware.

Many FPGA overlays are CGRAs that are built on top of the physical fine-grained configurable fabric. Their computations are carried out by arrays of processing elements (PEs) that are interconnected with virtual channels (VC). Instead of implementing the user application using low-level configurable logic of the FPGA, these operations are translated into computational tasks that take place in the PEs. Therefore, VCGRAs consist of a large number of PEs, laid out in a grid pattern, and VCs, that are a communication network connecting the PEs. This is shown in Figure 2.12. Each PE is a coarse-grained element (realized mainly using LUTs) and it is capable of computing incoming data and pass it on to the next PE, via an adjacent VC.

The VCGRA can be efficiently implemented using reconfigurable connections, with the assistance of the Parameterized Configurations (PConf) flow [90]. As PConf is a methodology used for implementing an application for which (part of)

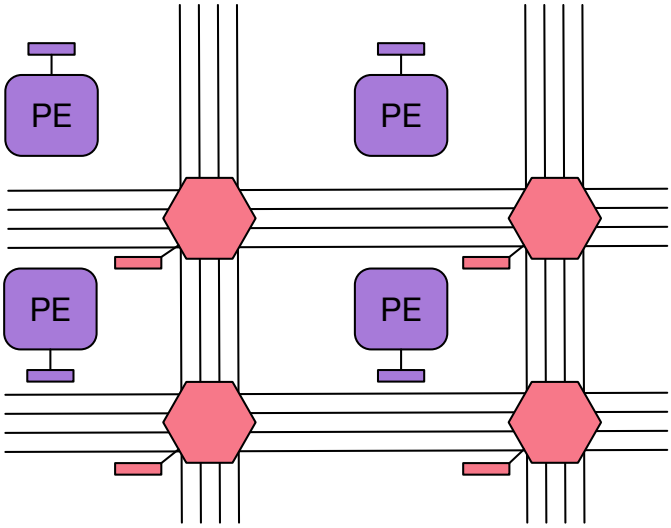


Figure 2.12: A part of a VCGRA grid with PEs, Virtual Switch Blocks (hexagons) and settings registers (rectangles)

the input values change infrequently, the VCGRA needs to be changed/adapted only infrequently. Hence, instead of implementing the VCGRA's inputs as regular inputs, with PConf these inputs are implemented as constants and the FPGA overlay is optimized for these constants. When these inputs change, the design is re-optimized for the new constant values by reconfiguring the FPGA. The interconnects are mapped onto parameterized physical Switch Blocks and Connection Blocks that are implemented using the PConf [147].

A VCGRA can be favored as a design option over traditional FPGA implementations, when architectural resources are scarce, reconfiguration is infrequent and computational tasks are demanding. In these cases, the overlay (e.g. VCGRA) can be chosen *specifically* to fit a specific application class. VCGRA implementations can use less area and power than the conventional ones [38], with a small impact in processing speed. Hence, by creating a VCGRA, less area resources are needed. Thus, it is often beneficial to transform an application into a VCGRA. The VCGRA, as similar coarse grained architectures, is constructed with multiple layers of PEs/VCs. The layers are physical pipelines in the VCGRA system and not abstraction layers.

Therefore, VCGRAs also provide a trade-off between performance of the hardware and the flexibility of a software program for computationally intensive parts of the applications. Therefore, VCGRAs are suitable to accelerate compute inten-

sive kernels. The overlays presented in [73, 81, 90, 91] are examples of VCGRAs.

2.4.2 VCGRA CAD Flow

Multiple flows have been proposed for VCGRAs [42, 57, 143]. In general, the tool flow is obtained by combining the standard FPGA tool flow with the VCGRA tool flow as it is depicted in Figure 2.13. The bottom side of the tool flow leverages Spatial programming to describe the mapping of an application on a given (V)CGRA architecture. There are different approaches to solve the CGRA mapping problem described, resulting in different generations of tools, as it will be analyzed below.

2.4.2.1 First generation

In order to map applications in CGRAs, the compilation time is shorter than mapping the application on a fine-grained FPGA. The VCGRA tool flow produces the VCGRA settings (for the PEs and VCs) and they are subsequently loaded into the settings register. The tool flow consists of a synthesis and a mapping tool. First, the application is converted into an array of PEs, interconnected with VCs. Next, the netlist of PEs is synthesized and mapped on virtual PEs of the VCGRA. All VCGRA interconnects are implemented on the communication network. Then, the Place and Route (P&R) tool provides the functionality of each PE and determines the functionality of the interconnects. The first version of the tool is shown in Figure 2.13. This work is further analyzed in [42].

2.4.2.2 Second generation

In order to further leverage the functionality of FPGA overlays, two new techniques have been created:

- Partially parameterized VCGRA: The functionality of the PE is decided by the different combinations of data. Different circuits are multiplexed within a parameterized PE. The multiplexers (intraconnects) within a PE are not parameterized, and only TLUTs are used. To reprogram the functionality of a PE, we perform micro-reconfiguration. This version can support commercial FPGAs.

- Fully parameterized VCGRA: Here, also the interconnects are parameterized. The parameterization is achieved by expressing the intraconnects of a PE as Boolean functions of parameters. Subsequently, both TLUTs and TCONs co-exist within a PE. This version can support academic FPGAs.

In more detail, this VCGRA tool flow builds on top of the parameterized reconfiguration tool flow. Here, there are also two stages: an offline version (generic)

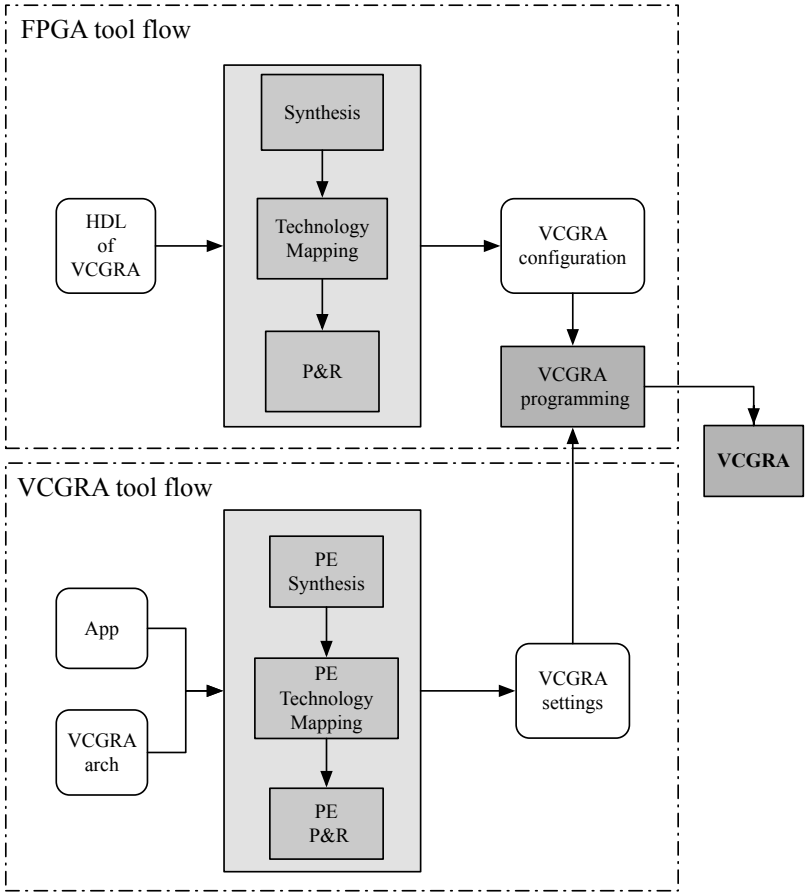


Figure 2.13: The two stage flow that supports a generic VCGRA implementation

of the tool and the online one (specialized). During the offline stage, the HDL design, has infrequent (parameterized) inputs. If a value changes in the parameter inputs, the configuration bits of the TLUTs (and TCONs) are reconfigured with specialized bits that are thus generated after evaluation of the Boolean functions for a specific set of parameter values. The Boolean functions are evaluated by a Specialized Configuration Generator (SCG) to generate specialized bitstreams. The SCG takes a specific parameter value and evaluates the Boolean functions to produce specialized bits. The SCG can be implemented on an embedded processor (PowerPC, ARM or MicroBlaze) present in the FPGA.

In this approach, the settings registers are mapped to the FPGA's configura-

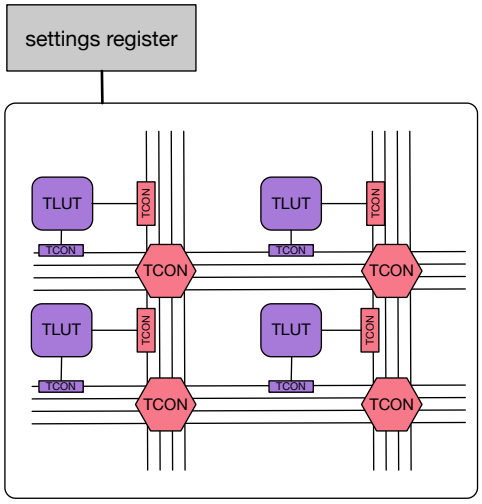


Figure 2.14: A schematic representation of a PE, with TLUTs, TCONs and settings registers

tion memory and not in flip-flops. This causes massive savings in flip-flops. If we are reconfiguring only LUTs, and not the interconnects, we can map FPGA overlays in Xilinx FPGAs as well. Otherwise, only theoretical, academic FPGAs can be used, since we do not have access to the low level routing reconfiguration infrastructure of commercial FPGAs. In general, in the second generation VCGRAs, the PEs are optimized by symbolic constant propagation, that is integrated in the parameterized configuration tool flow. VCGRA's intraconnects are mapped in TCONs (lower level reconfigurable routing switches), resulting in overall reduction of LUT utilization for implementing the connection network. This is depicted in Figure 2.14, and in Figure 2.15 and is further analyzed in [90,91].

2.4.2.3 Third generation

Here, the tool flow moves away from academic FPGA architectures, and creates a VCGRA that can be used alongside the Xilinx flow, as a third party IP. The researchers in [38] have proposed a VCGRA generator that is assisting the designer in creating architectures and configurations for them and the implementation of hardware using vendor tool-chains.

First, a VHDL generator creates the top-level entity and all the corresponding architectural components for the VCGRA. The description of the communication infrastructure and the processing elements are provided to the tool as input files. A toolset processes the HDL-description of architectural templates for the com-

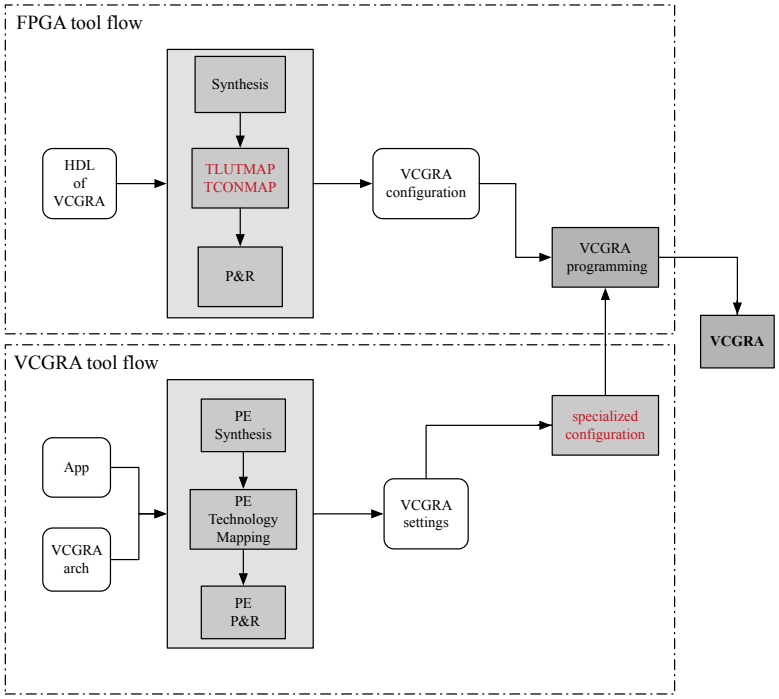


Figure 2.15: The two stage flow that supports a parameterized VCGRA implementation

ponents, and a definition of the desired target architecture as input. At this point, the VHDL generator can generate VHDL code and supplemental files, that are required to allow the mapping of algorithms and the creation of configuration vectors for the VCGRA.

A mapping tool has been designed for this architectures and it requires two graphs as input: a software graph of the application, and a hardware graph, of the internal description of the VCGRA. The output of this tool is a schedule for the graphs (if applicable), the description of the connection of inputs and outputs between the graphs and the VCGRA configurations for all the partitions of the algorithm's data-flow-graph. The output is the VCGRA's configuration which can be converted to bitstreams using the VCGRA generator [38]. This process is depicted in Figure 2.16.

With the 3rd generation of VCGRA tools, applications can be mapped in overlays in commercial architectures. The overlay design increases the usability over multiple hardware generations. Area reductions have been observed in comparison with traditional FPGA implementations. However, more than half of the whole processing time is used to transmit data values and coefficients, due to the limitations of the AXI interface. Moreover, frequent reconfigurations are not beneficial

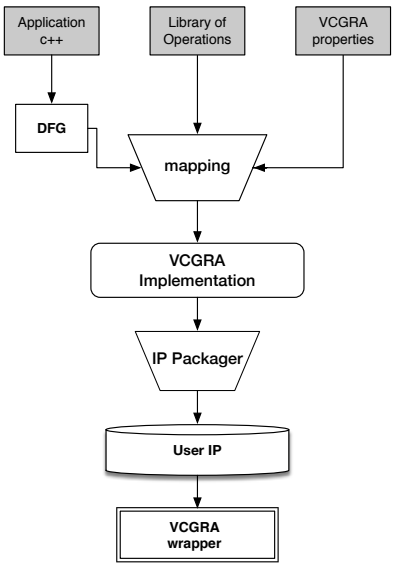


Figure 2.16: The tool flow that supports a Xilinx VCGRA implementation [38]

for the execution timings.

2.4.3 FPGA Overlays for Verification

Overlay architectures have become a popular option to support a variety of high-performance computing applications implemented on heterogeneous computing platforms. However, most of these architectures cannot offer an efficient way to dynamically debug and repair them.

With the emergence of FPGA overlays and VCGRAs, the current debugging techniques cannot efficiently debug the new designs. In order to perform FPGA-based debugging, it is essential to store signal values to trace-buffers (internal or external memories), as the signals cannot be accessed directly. Thus, the use of current in-circuit debuggers will have two possible effects on debugging VCGRAs. First, there is a large area needed for trace-buffers in order to store all the debugging information. Also, the designer will need to sit through long re-compilation cycles to debug a design with the conventional tools. Hence, for a high-performance application the design will need at least a day to recompile, to adjust a new set of signals. The second effect is the fact that, if a conventional debugger is used, the high-performance application designers will have to do a hands-on debug.

There are a number of drawbacks associated with Static Random Access Mem-

ory (SRAM)-based programming technology and FPGA overlays. For example, an SRAM cell requires 6 transistors which makes the use of this technology costly in terms of area compared to other programming technologies. Furthermore, SRAM cells are volatile in nature and external devices are required to permanently store the configuration data in high-reliability environments. These external devices add to the cost and area overhead of FPGAs. Additionally, the internal observability of FPGAs is non-existent, making debugging a challenge. These drawbacks are the focus of this thesis. Throughout this thesis, the two identified problems in design: technology shrinking that causes radiation effects and SoC complexity that causes complex bugs are investigated in par with FPGA overlay solutions. First with one-level FPGA overlays, and then with two-level overlay architectures.

2.5 Conclusion

In this chapter the basics of FPGA architectures and their respective tool flows are presented and the concepts that will be used throughout in the thesis are analyzed. In more detail, the parameterized configurations approach is presented, alongside its architecture and tool flow, and the basics of FPGA overlays. At the end of the chapter it is described how these custom FPGA techniques can be used to facilitate verification and reliability techniques. This chapter concludes the first part of the dissertation, that is used to provide to the reader the necessary background information needed in the remainder of the thesis.

Part II

In-Circuit Debugging

3

In-Circuit Debugging using Parameterized Configurations

In this chapter, a novel method for in-circuit debugging is introduced, that allows the insertion of low-overhead debugging infrastructure, by exploiting the technique of parameterized configurations. This allows the parameterization of the LUTs and the routing infrastructure, to create a virtual network of debugging multiplexers. It aims to facilitate debugging, to increase the internal signal observability and to reduce the debugging (area and reconfiguration) overhead. Signal ranking techniques are also introduced, that classify signals that can be traced during debug. Finally, the results of the method are presented and compared with a commercial tool. The area, time and power results and the trade-offs between internal signal observability and area and reconfiguration overhead are also explored.

3.1 Introduction

Ensuring functional correctness of implementations in hardware despite the rising levels of design complexity has been a major focus of research and development since the beginning of digital system design. This focus has led to remarkable advances in the verification and debugging of digital designs. While failures can be caused by various defects, such as logic, timing, circuit, layout, mask and process errors, there is no straight forward way to locate the root cause of these errors.

Therefore, specific verification (simulation-based) procedures have been used to detect the failures in the design. These procedures are generally considered costly and time consuming. Thus, verification and particularly hardware debugging has become one of the biggest challenges in hardware design. Ensuring a design's functional correctness, within time-to-market constraints continues to stand as one of the biggest challenges for today's hardware design teams.

Designers have been using software-based analysis tools, such as simulators, to ensure the design's functional correctness. Simulators give the designer a view of the system under test and its behaviour at any time. However, the rising complexity of embedded systems, makes it inherently more difficult to isolate the silicon failures and to find their root-causes. Moreover, simulators are time-consuming and they test the design under ideal conditions, which can differ from run-time operation. Thus, simulation can be incomplete and bugs can potentially escape from the attention of the simulator.

A growing number of designers are now opting to prototype their design using one or more FPGAs. Such FPGA emulation is much faster than simulation and enables higher verification coverage compared to simulation, allowing designers to test their design using realistic scenarios. The main fundamental drawback of using FPGA emulation is the limited internal signal observability. While simulation gives full visibility, FPGA emulation allows the designer to observe only a restricted number of signals due to the scarce number of output pins. This limits the productivity of debugging via FPGA emulation, as the designer has to recompile the entire FPGA in order to observe different signals through the output pins.

There is a growing need, as the designs become more complex, for software-like debugging. The designer should have a hands-off debugging without a detailed knowledge of the underlying hardware. Moreover, the designer should have early and easy access to all signals with an automated tool. Last but not least, as the designs scale, the available FPGA resources can become scarce. As FPGAs scale to even larger capacities, we need simulator-like observability. Even though FPGA emulation operates orders of magnitude faster than simulation, it is critical to provide simulator-like observability for FPGA emulation during debugging and verification and within TTM constraints.

3.1.1 Motivation

In-circuit debuggers offer a practical solution to efficient debugging and verification. However, the widely used, commercially provided debug tools, such as ILA by Xilinx, Quartus by Intel and Identify by Synopsys, offer limited internal-signal observability and they induce large additional resource overhead. Usually, designers use their own experience to overcome the limited internal signal observability by intuitively selecting the most critical signals to be observed, in order to avoid

the long recompilation cycles. Additionally, vendors want more software designers to also use FPGAs and they may not have the knowledge to select the critical signals, adding more pressure to get software-like debugging capabilities. In order to overcome the limited observability, the commercially available solution tends to insert trace-buffers to record various cycles of a trace of a subset of internal signals, while the device operates. Xilinx and Altera and technology independent vendors both provide in-circuit debuggers with trace-buffer integration [65, 155]. The main drawback of these techniques is that the subset of signals has to be predetermined before the nature of the bug is known, during compile-time. Then, if the source of the bug is not found, the designer has to recompile the design, in order to pick a new subset of signals. Often many recompilations and/or reconfigurations are required during debugging, to narrow down the root cause of the unexpected behaviour.

Researchers have proposed the use of incremental routing in order to connect signals to trace-buffers without a complete recompilation [61]. However, these techniques are still slow. Moreover, these tools perform independently, after the design is finalized. There is a need though, for a debugging tool flow that is completely integrated within the normal CAD flow, in order to limit hands-on, experience-based signal selections.

3.1.2 Contribution

In this chapter I propose a novel in-circuit verification toolflow that integrates debugging functionality in the typical FPGA design flow, before the design is finalized. It operates in two stages. During compile time, it automatically generates an overlay debugging infrastructure, based on the parameterized reconfiguration principles, and an adaptive number of trace-buffers. Instead of signal pre-selection, all signals are parameterized and virtually connected to on-chip memories. Then, during debug time, the tool enables a trigger-to-debug infrastructure. In this mode, the adaptive debugging starts to connect signals to trace-buffers by using the reconfigurable routing switch blocks of the target FPGA.

The proposed method combines the advantages of enhanced internal signal observability and fast FPGA reconfiguration in one tool flow. It has three main parts: a design step to add the debugging infrastructure incrementally, a parameterization step that applies the parameterized configurations technique in the design under test and an online step that applies the debugging methodology to a target design. This approach has various benefits over other debugging methods:

- Extended internal observability: With integration with Parameterized FPGA Configurations, the designer can trace a new set of signals fast, completing the debugging process within TTM constraints.
- Reduced area resources: By parameterizing all possible internal signals, and

dynamically adapting the added on-chip memories, the extra resources are reduced.

- Minimal debugging cycles: By prioritizing between signals with ranking algorithms, the source of the bug can be found faster, hence reducing the total number of debugging turns needed.

- Automated tool: The designer has only to select the DUT and to run the flow to perform hands-off signal tracing. Minimal user interventions assure that the DUT can be less prone to errors that can potentially occur during debugging. The less gate-level manipulations the user has to perform, the more the original design can stay unaffected.

The main contribution is a novel methodology where all internal signals of a given design can be selected automatically for tracing. Then, a signal selection algorithm adds the possibility of ranking internal signals. The ranking is done based on various criteria. In this way, the signals are organized based on their expected impact on a design. A signal-selection step has been added to ensure the signal selection efficiency and quality. The generated netlist has integrated debugging functionality and can be enabled and disabled without affecting the design under test.

The added functionality enables the user to debug large designs without significant area overhead. We use the standard benchmarks that are widely accepted and used by the testing community (ITC99), to validate the usage of the in-circuit debugger in realistic designs. In the experiments section, we perform design space exploration to validate the scalability of the method and of the signal selection algorithms, while increasing the design size and decreasing the size of the available FPGA resources. Additionally to the added functionality, the efficiency of the proposed CAD tool is explored alongside its compilation times for large designs. The proposed on-the-fly adaptation of the debugging infrastructure achieves a balanced trade off between area overhead, signal observability and compilation time. The benchmark suite is used in order to calculate compilation times and area overhead of the debugging infrastructure.

The main contributions are:

1. Dynamic data folding of the internal signals of a given design under test. Signal compression of mutually exclusive signals.

2. Two Signal Ranking techniques, that classify the internal signals to allow debugging of larger designs.

3. A tool flow that applies the above-mentioned techniques to a given design.

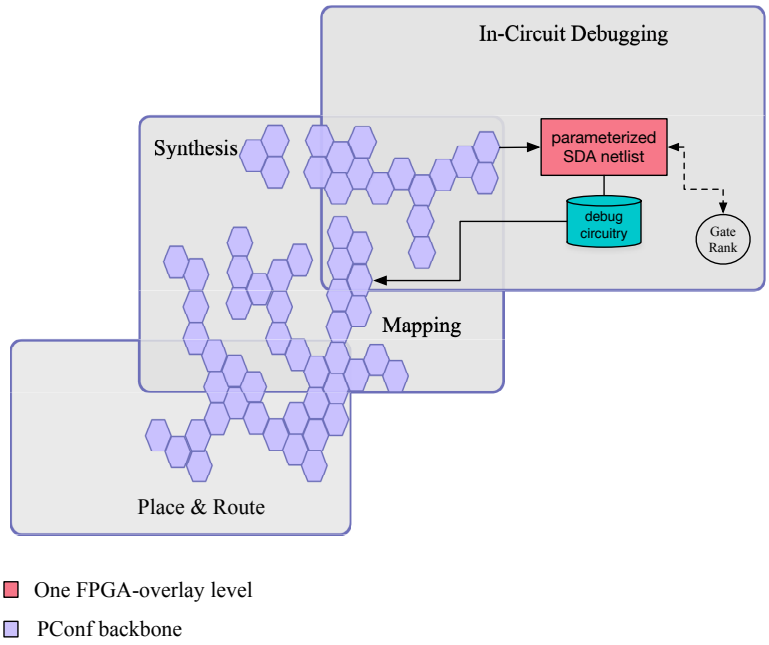


Figure 3.1: Visualization of this chapter's contribution

The contribution can be visualized in high-level in Figure 3.1 and is a first step towards adding debugging in the next generation tool flow that was depicted Figure 1.1(c) in Chapter 1.

The remainder of this chapter is organized as follows: Section 3.2 reviews the state-of-the-art techniques for in-circuit debugging. Section 3.3 presents the proposed technique and Section 3.4 the proposed tool-flow. Section 3.5 presents optimizations on the original technique and the adapted design flow that supports it. Finally, Section 3.6 describes the experiments that were performed and the results, followed by the Conclusion in Section 3.7.

3.2 Related Work

Most FPGA designers use pre-silicon design verification and analysis tools before downloading a design into the physical device. However, the designers cannot rely only on pre-silicon design verification to detect all design bugs, as it is not possible to reproduce realistic environmental conditions that a design will encounter with pre-silicon verification. It is several orders of magnitude slower than the actual silicon. It is very useful for some situations, such as the verification of individual bugs, but it faces scalability issues for full chip-level verification [36]. Therefore, it is often impractical to simulate complete systems, because software simulation scales badly, as it becomes slower when the design is bigger. Also, it is not always easy to create realistic scenarios and fault models to locate design errors. Furthermore, the complexity of integrated circuits continues to increase, making simulation impractical.

There are four basic methodologies for in-circuit FPGA debugging, namely Embedded Logic Analyzers (ELAs), Embedded scan chains, external test equipment and readback.

Debugging with External Test equipment, such as oscilloscopes and external logic analyzers is convenient if there is access to the necessary equipment [58]. External Test equipment operates by monitoring the external output pins of the integrated circuit. It analyzes how the states of the external output pins respond to certain signals placed on the external input pins, in order to determine if an error has occurred within the integrated circuit. However, since the external logic analyzers are capable of accessing the integrated circuit only through the external pins, internal conditions of interest for debugging cannot be monitored directly. With these models, it may be impossible to accurately determine the precise source of a bug, as there is no direct access to the internal inputs and outputs of the components under test.

On-chip observability can be enhanced using trace-based techniques. Trace-based techniques operate by allocating a significant amount of the FPGA's memory resources to record a small subset of internal signals while the FPGA is operating in real time. Various trace IP solutions are provided by the commercial vendors [64, 142, 146]. Here, the subset of signals are connected to on-chip memories, called trace-buffers, and it must be determined by the designer before the circuit is implemented. The trace-buffers are used to record a history of the important signals (or nets), during normal device operation [59]. Then, an engineer can use this information to understand the behaviour of the system. The drawback of this technique is that only a limited amount of such instruments can be inserted due to resource constraints.

Debugging with ELAs is one of the basic methodologies for in-circuit FPGA debugging [64, 142, 146]. However, it offers limited observability, as only a few

signals can be observed at the same time. Therefore a recompilation is needed if a new debug cycle starts. Additionally, their area overhead increases with the number of signals to be observed. ELAs are provided by commercial vendors, as debug IPs. Both Xilinx's Vivado and Intel's Quartus use general-purpose incremental compilation to avoid full recompilation if the debug core is updated during debug iterations, whereas Intels SignalTap II compiles the debug core with the user circuit as a separate partition (and recompiles this partition separately if the property of the debug core is updated). Synopsys Identify requires a recompilation if new logic is introduced to the debug core.

There are tools that avoid recompilation, such as Certus by Mentor Graphics [103]. However, these tools still require the designer to predetermine a few signals that they can observe. Therefore, the subset of signals that can be observed is small as well. Another drawback is that the signals have to be selected before synthesis. Hence, observing a new subset of signals requires the circuit to be re-instrumented and recompiled, a process that can take hours [22]. Additionally, the insertion of the debug circuitry and the pre-selection of signals can alter the place and route of the design and can potentially create other problems, such as the user circuit may no longer fit in the FPGA device, or artificial timing limitations caused by the debugging circuitry. Therefore, complete on-chip visibility is still not provided and a full recompilation is not always avoided.

All the above-mentioned techniques are slow and/or have large area overheads. Also, they offer limited observability, as their area overhead increases with the number of signals to be observed. The approaches described below aim at enhancing the limited on-chip observability by recording more signals for longer periods of time, by minimizing the area and time overhead and by avoiding recompilation or re-instrumentation of the design.

Embedded scan chains offer very high observability and operate in virtual-time, but they have reduced performance [79,148]. Most commercial FPGAs offer a way to read out the contents of its flip-flops and block-RAMs through a suitable interface, without destroying the machine-state. It does so without any overhead on the FPGA fabric and hence no perturbation of the application. These FPGAs can support readback-based techniques. Readback stands between embedded logic analyzers and scan chains, as they introduce less area overhead and they don't need to restore to the initial state.

Readback is used to retrieve the entire state of the circuit. It is initiated by sending a sequence of commands to the device, from where the machine-state captured in LUT registers, block RAMs and distributed RAMs can be read out through an interface. Readback approaches offer full observability without requiring re-synthesis, nor additional resources. Effort has been made to enhance their functionalities by adding features that enable the design to run unsupervised until it finds an error and then launch Modelsim automatically. However, this can be

done only for a single clock domain [75]. This is solved at [144], where the authors extend the debugging functionalities to support multiple asynchronous clock domains in a clock-deterministic manner at negligible cost. Most importantly, no BRAMs or additional routing circuitry are required.

Readback-based solutions have an advantage over their counterparts, as they offer full observability, without requiring re-synthesis when changing or adding debug signals. However, they cannot provide the same level of flexibility as do overlay architectures. Moreover, they also need to halt the circuit before scan-out. Also they may cause data loss without a specific support mechanism. Moreover, the introduction of a trigger mechanism can potentially affect the critical path. This can greatly slow down their use for real-time debugging [4, 144, 156]. Thus, in order to efficiently debug the hardware designs in-system, the current limitations of the in-circuit debuggers have to be addressed.

Trace buffers are on-chip memories that are used to record a specific set of signals for off-line analysis, during debug. Academic works have used overlay-based incremental techniques for signal tracing while avoiding full recompilation during debug [31, 47, 61, 62]. In [47, 62] the researchers used incremental routing techniques to connect signals from the user circuit to trace buffers, whereas in [31, 61] the researchers extended these techniques into a virtual overlay routing network and an accompanying routing algorithm to route trace signals to trace-buffers, by connecting all on-chip signals to the available trace buffers. Then, during debug, the designer can choose any set of the design signals for observation. If this set has to be changed, then only the control signals of the overlay's routing multiplexers have to be updated. In that way, the need for a full recompilation is eliminated and the original mapping remains unaffected. In [32, 60, 63] academic works demonstrate the post-implementation insertion of debugging infrastructure. All these techniques distribute the debugging logic over unused resources. However, these works demand adequate empty regions to install their circuitry. These techniques can be proven problematic in highly utilized FPGAs. Additionally, they can create a new longer critical path and cause routing congestion. In order to handle the resource overhead, researchers use the properties of compiler-optimized HLS circuits and adaptable trace-buffers to enable dynamic tracing [43, 44] with limited resource overhead. They also adapt the trace-buffer architecture for even more efficiency. Dynamic signal tracing techniques increase observability. However, on a highly-utilized FPGA, it may be hard or even impossible to find a region large enough to implement the additional logic. In that case as well, a trigger mechanism, can also potentially introduce a new long critical path. Most importantly, this technique is only available for compiler-optimized HLS designs. Even though readback solutions are more efficient in terms of resource requirements, the overlay and dynamic tracing architectures provide more flexibility. However, all of the above-mentioned techniques can introduce a longer critical path and due to the

fact that they perform non-automatic manipulations, they can be time consuming, introduce errors to the original design and affect the circuit's optimization.

3.3 In-Circuit Debugging using Parameterized Configurations

In this chapter, the first contribution of this thesis is presented. This chapter aims at building a debugging infrastructure, where internal signals are (virtually) connected to trace-buffers in an efficient manner. The proposed technique targets either ASIC designs that are prototyped in FPGAs to expedite verification, or designs that use FPGAs as a final product. First, it is proposed a new trace-based debugging approach that is combined with the Parameterized Configurations method, where the latter is used to virtualize the internal signals. Then, it is shown how these signals can be prioritized in order to minimize the number of debugging cycles.

Hence, the remainder of this chapter discusses the proposed technique, where the debugging instrumentation is fully automated and it occurs during the original compilation of the design. The debugging infrastructure is added incrementally as a latency insensitive design and alongside the original RTL design, making it less prone to errors. The debugging circuitry is fully adapted, integrated and optimized alongside the original circuit. Additionally, since the tool is automated, the designer does not have to indulge in manual-level optimization and hands-on a priori signal selections.

Efficient Routing of Debugging resources

In order to increase the routing efficiency in large designs, the concept of a TCON is used [147]. A TCON is defined as a connection with a connection condition expressed in terms of parameters. A TCON is implemented by a set of wires and switches, and the dynamic reconfiguration of some of the switches in the set. Regular connections have a source and a sink. A TCON has a source and a sink as well. Additionally, every TCON has a connection condition:

$$\zeta(p) : B^n \rightarrow B$$

where n is the number of parameters, as it is shown in Figure 3.2. This is a Boolean function of the parameters that returns true when the design requires the connection to be active. The connection conditions of the TCONs reflect the fact that not all connections are needed at the same time, since in debugging not all signals need to be observed at the same time.

TCONs allow us to distinguish which signals are mutually exclusive in time. These signals will be allowed to share the FPGA routing resources. TCONs are

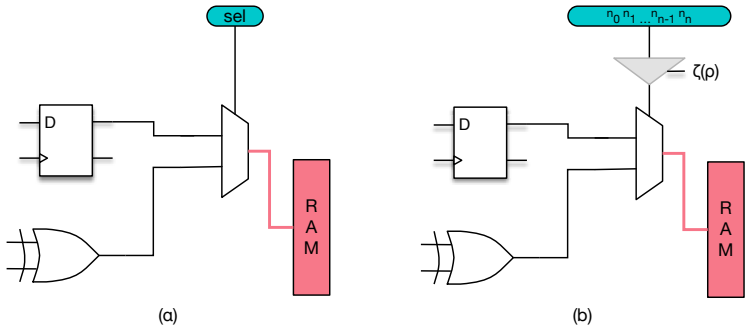


Figure 3.2: Description of normal routing connections (a) and tuneable connections (b)

abstractly generated during technology mapping and are later refined to concrete FPGA resources during placement. After placement, the endpoints of the TCONs are fixed to the specific signals-multiplexers to be traced and the router further refines the DUT by reserving the switches and wires to realize the connection between the subset of multiplexers and trace-buffers.

During routing the tool flow has to route a large set of TCON-containing signals. These signals have more than one resource sharing possibility. They may share resources if they have the same source or if they are not active at the same time. Two TCONs, sgn_1 and sgn_2 with their connection conditions ζ_1 and ζ_2, are not active at the same time, if their connection conditions are not simultaneously true for every parameter value (n_0 and n_1). Thus, if sgn_1 and sgn_2 are not active simultaneously for each parameter p in a set of parameters P then

$$\forall(p) \in (P) \Rightarrow \neg(\zeta_1(p) \wedge \zeta_2(p)) \tag{3.1}$$

The results of Equation 3.1 are reflected in Figure 3.3. Here, the tool has detected during technology mapping that the sgn_0 and sgn_2 that represent the gates $NAND_0$ and $NAND_1$ are not used at the same time, so they can share resources. Therefore, during debugging, the signal set will contain the sgn_0 and not sgn_2. This is defined by the Boolean function that uses as parameter inputs the multiplexers' selection bits: $n_0, n_1, \ldots, n_n$. Hence, the tool will create a subset of signals that for each multiplexer-cluster are never used at the same time.

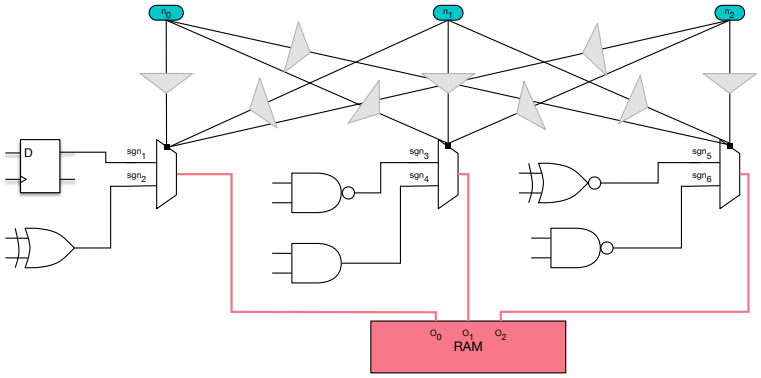


Figure 3.3: Signal select. Multiple signals are controlled via a set of parameters. Mutually exclusive signals form a Boolean function of the parameters.

$$\begin{aligned} O_0 &\Leftarrow i_0 \Leftarrow n_0 \neg n_1 \\ O_1 &\Leftarrow i_1 \Leftarrow n_0 n_1 \\ O_2 &\Leftarrow i_2 \Leftarrow \neg n_0 \neg n_2 \end{aligned} \tag{3.2}$$

The mutually exclusive signals are now the outputs of the debugging multiplexers $i_0, i_1, \ldots, i_n$. New connections are made to the on-chip memory for observation. These connections are not parameterized, since they are active all the time, tracing different signal sets for multiple cycles for each debugging turn. The corresponding RAM inputs are depicted in Equation 3.2.

The proposed method depends solely on time-exclusive internal signal multiplexing. This means that the tool will attempt to route all signals to the trace buffers while minimizing their wirelength. As a first step, in this method a fixed number of trace-buffers is initiated. Then, the tool tries to route all possible signals to the trace-buffers. In this case a maximum signal observability is defined, with a penalty on area resources and possible routing congestion. Here, the design is fully parameterized, with all signals being connected to virtual multiplexers and all the new multiplexers are guided to a fixed number of on-chip RAMs. Since the time-multiplexing aspect exists, that uses the same resources for signals that are not used during the same clock tick, it is more resource-efficient than the conventional ELAs. This algorithm is efficient, in the case that there are available FPGA resources.

The multiplexers' selection bits that have been incrementally added, are annotated as parameters, as they change (but less frequently than the other parts of the design) depending on the set of signals that will be observed during the verification

phase. These parameters indicate whether or not a single signal has to be observed in a certain debugging run. All the debugging multiplexers are implemented in the FPGA's reconfiguration resources (instead of the regular resources) reducing the overhead significantly. This is achieved with the TCON mapper, that has been explained in Chapter 2.

The parameterized Boolean network that implements the debugging infrastructure is generated during the synthesis step. Then, it is not mapped onto the resource primitives available in the target FPGA architecture, but on abstract primitives in the routing infrastructure that represent parameterized versions of these resource primitives. This procedure is further explained in Section 3.4. This process has to be performed in such a way that after the new modifications, the new description remains synthesizable. In our case, it is guaranteed due to the fact that the incrementally added parts are pre-verified and synthesizable. During the signal parameterization, the internal signals are interconnected with the debugging infrastructure, based on the latency insensitive design theory [14]. This means that we incrementally add pre-verified design components (multiplexers and memories) that are insensitive to communication delays and we integrate them in the design, in order to meet performance criteria. With latency-insensitive designs the added logic has no strict constraints on the number of clock cycles in which it must return a result. On-chip memories that are used as trace-buffers are a good example of latency-insensitive logic, as the trace-buffers are used to record multiple cycles of on-chip signal activity for debugging. This improved flexibility usually comes at the cost of area overhead. However, in our case, most of the added components (the multiplexers) have some virtual inputs. Thus, even though they are added, they are parameterized. The parameterized infrastructure will be able to assign signals to trace-buffers.

3.4 In-Circuit Debugging Tool

In this section, we present the proposed in-circuit debugging method, which consists of two phases, as with the conventional design flow depicted in Figure 3.4: the design (generalized) phase and the debugging (specialised) phase.

The first stage is the design phase and it is dedicated to creating a virtual network of multiplexers that contains Boolean functions and describes multiple signal sets, according to the PConf flow. The parameterized debugging architecture in fact is created during this phase, by annotating as parameters the selection bits of the internal signals. The debugging infrastructure is added on a design and the design is then synthesized, placed and routed and a generalized bitstream (with Boolean logic) is created. This varies from the conventional bitstreams, as a virtual FPGA overlay is created that has a Boolean function that describes the way signals are connected through trace buffers.

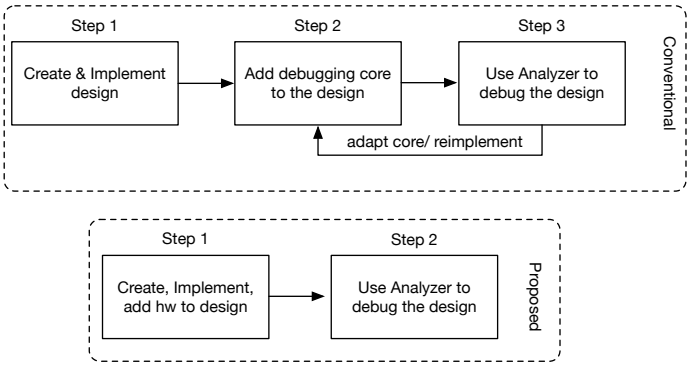


Figure 3.4: Outline of the conventional versus the proposed debugging process. The elimination of one step boosts runtime efficiency.

At the end of these steps, before the bitstream is loaded within the FPGA configuration memory, the tool evaluates the Boolean function with the parameter values denoting which signals are observed and creates a specialised (conventional) bitstream. The intermediate steps that complete the design phase are the main focus of this work.

The second stage is called the debugging phase and performs the actual debugging. During this phase, for each debugging turn, the design can be reconfigured with different signals. The signals that are not traced at the same time can share routing resources (based on the parameter settings). This is outlined in Figure 3.5. Each phase is described step-by-step in the remainder of this section.

3.4.1 Design phase

The method used to apply the proposed technique enables automatic generation of PConfs starting from the DUT and is based on the same steps as conventional FPGA tool flows: synthesis, technology mapping, placement and routing [56]. Figure 3.6 outlines this stage of the tool flow.

Custom Hardware Addition

At the beginning of the design phase, the tool locates the internal signals and adds the multiplexer network, that is connected to the internal signals. Then, the trace-buffers are integrated in the design.

In more detail, the tool reads the DUT and locates the internal signals and connects them with the debugging infrastructure randomly. The multiplexer's selection bits are annotated as parameters. The parameters will indicate whether or not

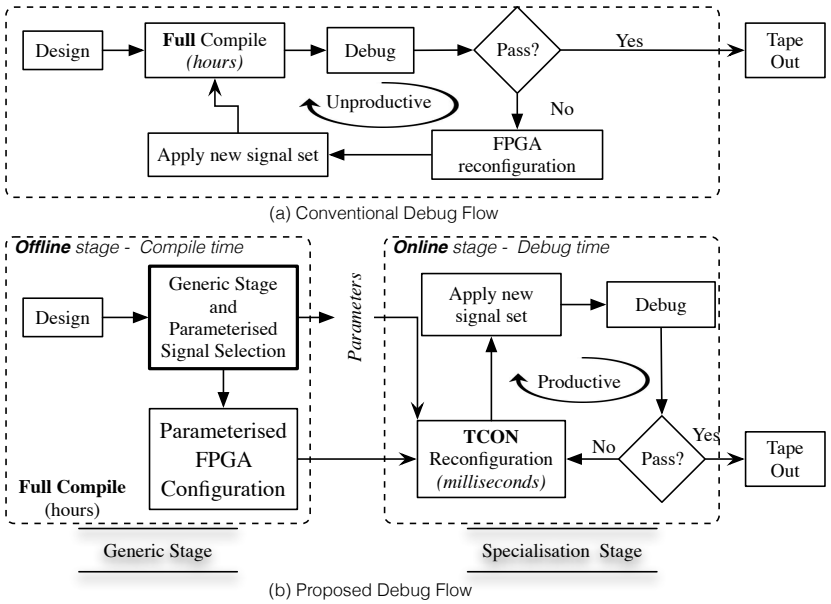


Figure 3.5: Proposed debugging flow. The proposed two discrete offline and online stages boost runtime efficiency.

a single signal has to be selected to be observed in a certain debugging run. In order to achieve that, these multiplexers are implemented not in the regular resources but in the FPGA's reconfiguration resources, reducing the overhead significantly.

An overlay network that routes the possible signals to the available trace-buffers is created and the appropriate annotation is automatically generated, that will enable the parameterization of the added hardware, at a later stage. Figure 3.7 demonstrates in different layers how the signal parameterization is achieved. The middle layer shows the FPGA and the signals that need to be observed. Then the virtual level adds the infrastructure that multiplexes the signals to trace-buffers (top layer). Therefore, after the hardware addition the DUT has an integrated debugging infrastructure, that is almost the same size as the original circuit, but for an extended circuit with tracing infrastructure installed at its signals. Hence, dedicated FPGA resources (that are claimed before implementation) for the multiplexer network and for the trace-buffers are no longer needed.

The parameterized debugging infrastructure is integrated inside the normal CAD flow, in order to alter as less as possible the critical path delay, to prevent additional routing stress when new signals are to be traced, and to offer automation of the process. Since the debugging infrastructure is incrementally added during compilation, it is optimized alongside the original design.

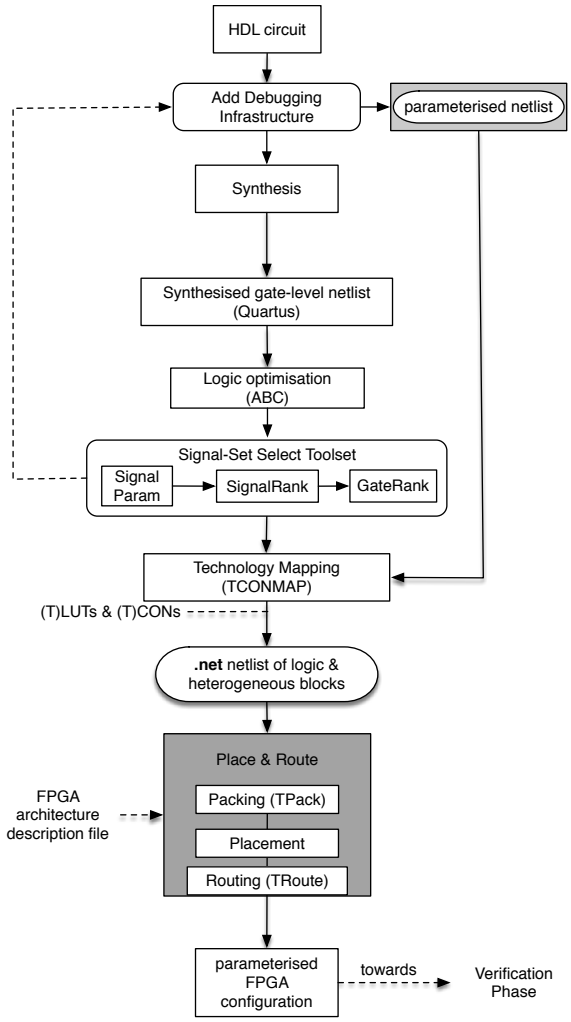


Figure 3.6: Schematic of the generic stage of the proposed tool flow.

Synthesis

At this point, the DUT is synthesized. The synthesis step can be performed by ABC, or by any tool that is able to synthesize functional blocks to an FPGA flow and communicate directly with it. ABC is a part of the VTR flow [97] (a common academic FPGA CAD flow). This is needed, as the next stage we will use the graphs that are generated by ABC. Additionally, a synthesis tool that can pass the parameterization information without altering the design is needed.

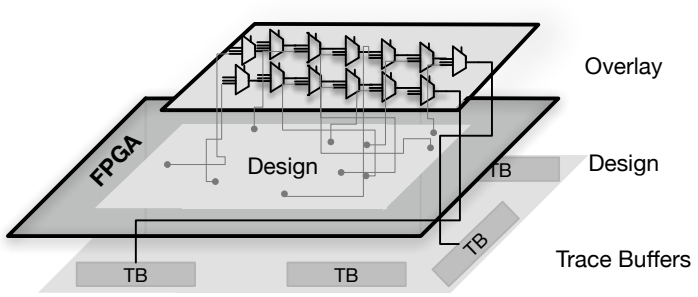


Figure 3.7: Demonstration of the separate layers. The user circuit, the parameterized multiplexers and the trace-buffers respectively.

TCON Technology Mapping

During technology mapping, the parameterized Boolean network of the added debugging infrastructure that was generated during signal parameterization is not directly mapped onto the resource primitives available in the target FPGA architecture, but intermediately on abstract primitives that introduce and allow the reconfigurability of the logic and routing resources. Therefore, the extra multiplexers added to guide the internal signals to the (also added) trace-buffers have their selection bits parameterized into Boolean functions and mapped in the virtual abstract primitives.

TPaR Placement and Routing

Next, the Tuneable Place and Route tool (TPAR) places and routes the netlist and performs packing, placement and routing with the algorithms TPack, TPlace and TRoute accordingly [12, 56, 147]. These algorithms route the DUT in a way that their routing resources can be reused to route a new subset of signals to the trace-buffers during debugging. Hence, the signals that can be traced can share routing resources (based on the parameter settings).

3.4.2 Debugging phase

At the end of the computationally intensive design phase the TPaR creates a PConf. The debugging phase is the *specialised stage* of the flow, where the DUT's trigger is initiated and the debugging can start. During this phase, for each debugging turn, the design can be reconfigured with different signals. The signals that are

not traced at the same time can share routing resources (based on the parameter settings). For each debugging cycle the network that is partially reconfigured with the exact signals the designer wishes to trace, the new signal selection translates directly into a new evaluation of the function that represents the selected signals. Then it can be reconfigured with Dynamic Partial Reconfiguration.

The multiplexer network added with the signal parameterization tool is reconfigured with the specialised solution which is evaluated according to the new signals that have to be observed. In order to generate a new set of signals, a specialised configuration is automatically performed, with minimal user interaction and without halting the DUT. The Boolean functions are evaluated for a specific parameter value by the Specialized Configuration Generator to generate a specialized bitstream. Usually it is implemented on an embedded processor. The embedded processor is responsible to swap the specialized bitstream into the configuration memory using the HWICAP. It is worth noting that here, only the configuration cells of all the routing switch boxes and the connection boxes for the memory resources will be reprogrammed, instead of the full recompilation and/or reconfiguration, (as it is the case in related work). Therefore, by implementing a PConf the extra recompilations are avoided during debugging, since only an evaluation of a Boolean function is needed, instead of recompilation and/or reconfiguration. Moreover, there are no FPGA resources dedicated to the inserted multiplexers.

3.5 Efficient signal selection

In order to multiplex signals in the on-chip memories efficiently, the random-set selection implementation is not enough. The basic drawback is the fact that optimally the designer would want to randomly enable all signals to possibly connect to a set number of trace buffers. The efficiency of the PConf lies in the mutual exclusivity of the signal parameters. Therefore, the mutual exclusivity in the case of multiple parameters (all internal signals) cannot be guaranteed and/or introduces routing congestion. In order to overcome this, two options are currently available. The verification engineer can either allocate the trace-buffers in the spare FPGA resources [61], or pre-select a subset of signals to trace, so that the trace-buffers needed are significantly less than the total signals. The first option is not viable, since the debugging infrastructure is added after synthesis and optimized alongside the DUT. Thus, the leftover resources are not yet known. The second option, regarding signal pre-selection is not optimal, since in that case the designer needs to reserve a large area for the signals.

However, since the designs are becoming more complex, on-chip debugging based on signal sets that are decided by the design engineer, is not always efficient. Moreover, the exploitation of the routing infrastructure has its limitations. Additionally, the designer often needs an indication of the criticality of each part

of the design, without having to indulge in the design itself. Hence, avoiding random signal selection and giving a set of signals that are more prone to bugs, could reduce the number of debugging cycles. Furthermore, the set itself can provide an indication of the design's complexity and even on the total amount of resources needed to be pre-installed during this first compilation, to ensure a smooth debugging process. In that way, the verification engineer doesn't need to have detailed knowledge for the underlying hardware and about the design itself to perform debugging in an efficient manner. Therefore, in the remainder of this chapter, the signal selection process is transformed from random into a classification problem and we propose two signal selection techniques that select signals based on specified criteria and not solely on randomization.

SignalRank is an algorithm that we created that partitions the signals between critical and non critical signals. The critical signals are the subset of the total number of signals that have many connections within the design. Hence, if the gate u has a large fanout (F_o) and/or fanin (F_i), according to SignalRank, it will be added at the top of the list. That means that during debugging, the signals of the gate u are immediately traced. Moreover, SignalRank can give an indication of the congestion of the netlist and adapt the trace-buffers accordingly.

SignalRank has significant limitations. Despite the fact that it can give an indication of the criticality of a single signal or gate, there is no way to tell which signal/gate is more critical, if they share the same fanin and fanout. In order to take into account these facts, alongside fault propagation, SignalRank has been extended to GateRank.

3.5.1 GateRank

A gate-level netlist can be described as a set of nodes that represent the gates and a set of edges that represent the wires that connect these gates. This structure is similar to the representation of the World Wide Web, where pages are represented as nodes and hyperlinks as edges. The World Wide Web contains countless number of web pages. An objective measure of the importance of a web page can be found by looking at the pages that link to it. The PageRank algorithm created by Page and Brin [114], is the first and primarily used algorithm by the Google search engine. PageRank assigns a rank to every web page and influences the order in which Google displays its search results. It is based on the link structure of the web graph and it does not depend on contents of web pages nor on the query.

Since gate-level netlists can be represented in similar graphs as webpages, one can use some principals of the PageRank algorithm to rank the impact that each gate has on the design, in a same way the PageRank determines the importance of its webpage based on the amount of hyperlinks that are connected to it. The problem of ranking gates is similar as ranking pages. If a gate has a higher number

of inputs (and/or fanout), it has a lot higher chance of influencing the output and hence is more important than other gates. This is similar to the number of links in web pages.

GateRank creates a stochastic matrix with adjacency functions that represent whether its gate is connected with all other gates. The graph shows not only the adjacent nets to each gate, but also how important each gate is based on propagation. Not only the designers can see the ranking for every gate, but they can have a first estimation of the fault propagation. Thus, the critical and non critical region of each design can be identified. In other words, the GateRank for every gate u, (with nets v_i) is the sum of the GateRank of all gates contained in a set B_u (having an input from u) divided by the fanout of each gate $v \in B_u$. This is generalized in Equation 3.3, where $v_1, v_2, \ldots, v_n$ are the logic gates under consideration, B_{u_i} is the set of gates that have an output to u_i (so the number of elements of it is equal to the fanin of B_{u_i}), $F_o(v_j)$ is the fanout of the gate v_j, $F_i(v_j)$ is the fanin of this gate and N is the total number of gates.

$$G_{rk}(u) = \zeta \times SR \times \sum_{v_j \in B_{u_i}}^{N} \frac{G_{rk}(v_j)}{F_o(v_j)}, \tag{3.3}$$

Finally, ζ is a factor for normalization, and it is $0 < \zeta < 1$ because there are a number of gates with no forward links and their weight is lost from the system, due to the fact that some gates have primary inputs and outputs. The eigenvector contains all the values of the GateRanks for each individual gate:

$$\forall v_j \in B_{u_i}^N, \;\; R = \begin{bmatrix} G_{rk}v_0 \\ G_{rk}v_1 \\ \vdots \\ G_{rk}v_{i-1} \\ G_{rk}v_i \end{bmatrix} \tag{3.4}$$

A gate-level netlist can be easily described as a finite graph (AIG during mapping). This graph can be represented as an adjacency matrix, which is a square matrix used to represent a finite graph. The elements of the matrix indicate whether its pairs of vertices are adjacent or not in the graph. Subsequently, let A be a square matrix with the rows and columns corresponding to logic gates.

Let $A_{u,v} = \frac{1}{N_u}$ if there is an edge from u to v and $A_{u,v} = 0$ if not. If we treat R as a vector over gates, then we have $R = \zeta AR$. Therefore, R is an eigenvector of A with eigenvalue ζ. Normally, since we have initially a Markov matrix, ζ should be 1. However it is described below why $\zeta \neq 1$.

A limitation with the custom gate ranking functions is that there is no way to indicate whether there are not only gates pointing only to the same gates in the graph, forming a closed loop. During the iterations, this loop will neither

accumulate ranking nor distribute any ranking, since there is no way to indicate from where the GateRank function can start nor finish. In order to overcome this problem we introduce the F_o, F_i ranks alongside the normalization factor ζ:

$$\forall v_j \in B_{u_i}^N \rightarrow F_{snk}^{src} = \begin{cases} G_{rk_{source}} = \frac{Prim.F_i(v_j)}{F_i(v_j)} \\ \\ G_{rk_{sink}} = \frac{1-Prim.F_o(v_j)}{F_o(v_j)} \end{cases} \tag{3.5}$$

In Pagerank, the graph is unweighted. However, weights are added, based on the net's fanout data. In Eq. 3.5 we describe that $\forall u \in B_{u_i}$, if gate u contains a source as an input (primary input) it has a priority value equal to the number of its sources F_i. That means, that the gate with the largest number of primary inputs will be computed first, then the next one, and so on, until the gates that are starting to compute their rank have only internal signals as F_i or F_o. This value is not included in the GateRank, only in the priority of computations. The normalization factor is defined as:

$$\forall v_j \in B_{u_i}^N, \;\; \zeta = \begin{cases} 0.95, \; if \; \frac{Prim.F_i(v_j)}{F_i(v_j)} = 1, \\ 0.75, \; if \; \frac{Prim.F_i(v_j)}{F_i(v_j)} > 0, \\ 0.25, \; if \; \frac{Prim.F_o(v_j)}{F_o(v_j)} > 0, \\ 0.05, \; if \; \frac{Prim.F_o(v_j)}{F_o(v_j)} = 1, \\ 0.5, \;\; otherwise \end{cases} \tag{3.6}$$

Additionally, instead of adding weights, we can also just duplicate every output connection as many times as the fanout (change a net into a set of connections) and then apply GateRank. Both options produce the same results.

In a similar way, $\forall v \in B_{u_i}$, if u is a sink, then the priority value is the lowest. That means that the gates with at least one of their outputs being a primary output will be computed last. Hence, the more primary inputs an individual logic gate has, the more priority it gains to compute its GateRank.

$$R = \zeta \times \frac{F_{snk}^{src}}{N} + \zeta \begin{bmatrix} w(g_0, g_0) & \dots & w(g_0, g_N) \\ w(g_1, g_0) & \ddots & w(g_1, g_N) \\ \vdots & & \vdots \\ w(g_N, g_0) & \dots & w(g_N, g_N) \end{bmatrix} \times R \tag{3.7}$$

In Eq. 3.7, the $w(g_i, g_j)$ represent whether or not a gate connects to another, via a single wire. Therefore for every column j we have:

$$\forall j \in N, A = \begin{cases} \sum_{i=1}^{N} w(g_i, g_j) = 1, \; gates \; connected \\ 0, \quad otherwise \end{cases} \tag{3.8}$$

The sum for each column equals to 1, making R a stochastic matrix. Finally, sparse matrix properties have been used, to construct a weighted matrix representation of the adjacency matrix for the calculations. Since a sparse stochastic matrix has to be solved, it will scale very well and the scaling factor for large netlists would be roughly linear in $\log(N)$, where N is the total gate number.

3.5.2 Signal Classification Tool

The signals can be classified based on the GateRank computation. This is an optional step that can be used if the FPGA resources are scarce. It can be applied in synthesized designs, as it is shown in Figure 3.6 in Section 3.4. Therefore, after synthesis, the GateRank algorithm runs and calculates the adjacency matrices. In order to create a graph that describes the netlist, the AIG from ABC [97] is used. After the graph's creation, the netlist can be adapted based on the new information, by adding debugging infrastructure only in the signals with the highest GateRank. The newly formed signals and trace-buffers can then be added to the netlist incrementally.

At this point, the designer can decide about the amount of signals that can be observed at the same time. The factors here that are crucial for the internal signal selection are the GateRank number, the routing congestion (also estimated by the GateRank density graph) and an estimation of the FPGA's available resources. A pre-processing step can be also executed to assess these metrics. First the tool is executed once to estimate the available area. The maximum fanin/fanout (derived from SignalRank) describes the trace-buffer number N, as long as there is still enough space on the FPGA. Then, the N trace-buffers are connected to the N most significant signals (derived from GateRank), from total signals M. After this process is complete, the tool can synthesize, map, place and route the design (that now contains the debugging infrastructure) and form a parameterized configuration.

However, this reduces flexibility as we can no longer observe signals that were not selected by GateRank. If we need to debug one of these, we do need another recompilation step. So, this is a new trade-off between area overhead and debugging flexibility. Therefore, we can use the GateRank optimizations when the available FPGA resources are scarce and a signal selection for the debugging infrastructure needs to be made anyway.

3.5.3 Trace-buffer Optimization

Trace-buffers are implemented as RAMs. They have to be kept minimal, to avoid draining the FPGA resources, while being able to store and trace the maximum possible number of data. Increasing the number of signals causes more RAMs to be consumed by the tool and can adversely affect design performance. If the

designer needs to observe more signals, more parameters are needed and more area will be subsequently be consumed (for the trace-buffers).

Therefore, the more signals will be observed at the same time, the larger the area penalty will be. Additionally, due to the parameterization of the debugging infrastructure, multiple (parameterized) reconfigurations will probably be needed, to observe more signals. Hence, there is a reconfiguration time penalty as well.

By using signal classification techniques before adding the debugging instrumentation, some properties of the signals can be known before the integration with trace-buffers into the design. Thus, the number of signals to-be-traced can be dynamically adapted, in order to limit the area requirements (and possibly the number of reconfigurations).

Firstly, the signal classification algorithm is performed to have a clear view of the fault propagation, by creating ranked signal sets. This is based on the fact that at the moment that a bug is singled out during tracing, the backward propagation defines the next signal set, until the source of the bug is found. In that way, there is always a constrained number of debugging cycles, with an upper bound defined by the design's depth, that can be easily extracted from the AIG. Hence, there can be a maximum number of the times a parameterized configuration can occur, in order to reconfigure to create a new signal set until the source of the bug is reached.

Secondly, the length of trace-buffers added to the design is found by constraining the maximum number of debugging cycles needed during debug. The designer can determine for how many cycles the signals will be monitored. This is done by creating a linked list during the construction of the fault propagation method that has width and depth arguments. The width defines the maximum number of signals that can be traced each time (from the previous step) and the depth defines the maximum number of linked gates (connected directly via a single wire).

The trace-cycles can be adapted dynamically, based on the density of the GateRank graph. [1] In a dense graph, more tracing cycles and/or debugging resources are needed, as there are more connections among signals, compared to graphs with smaller density. Therefore, it is possible that more signals will need to be traced in a dense graph.

The amount of trace-buffers and their respective depths and widths can be adapted. Therefore, the classification of the signals and the density of the DUT's AIG can influence the size of the trace-buffers. Since the width defines the number of debugging cycles and the depth the number of trace-buffers to be added, both the width and the depth of the tracing instrumentation needs to be adaptable. That means that the RAMs that will be used as trace-buffers need to be adaptable. Since the trace-buffers can have an adapted number of trace-cycles and signal sets, the designer can use asymmetric RAMs as trace-buffers instead of the conventional symmetric ones.

[1] A dense graph is a graph in which the number of edges is close to the maximal number of edges

Therefore, asymmetric trace-buffers are added in the design incrementally. An asymmetric trace-buffer is modelled in a similar way as a symmetric trace-buffer, with an array object. Its aspect ratio corresponds to the port with the lower data width (larger depth). Trace-buffer resources can be configured with two asymmetric ports. One Port accesses the physical memory with a specific data width defined by the GateRank classification and the second port accesses the same physical memory with a different data width. Both ports access the same physical memory, but see a different logical organization. Such an asymmetrically configured trace-buffer has ports with different aspect ratios. A typical use of port asymmetry is to create storage and buffering between two data flows. One data flow for storage and one data flow for trace. Hence, they can operate at asymmetric speeds. Like RAMs with no port asymmetry, trace-buffers with asymmetric ports are modelled with a single array of array object. The depth and width characteristics of the modelling signal or shared variable match the trace-buffer port with the lower data width (subsequently the larger depth). As a result of this modelling requirement, describing a read or write access for the port with the larger data (signal trace) width no longer implies one assignment, but several assignments (multiple trace cycles). The number of assignments equals the ratio between the two asymmetric datawidths.

3.6 Results and Discussion

The architecture is evaluated in terms of area and time overhead. The designs used have been derived from the ITC99 benchmark suite. This benchmark suite meets all of our requirements, including a variety of realistic algorithms, defined inputs, scalability, and portability. These benchmarks are specifically designed for testing and include a set of input vectors that are designed by the automated test pattern generation community. Here, they utilize approximately 1% of the commercial FPGA (Xilinx Virtex-7).

3.6.1 Offline Comparison of Area Utilization

First, the debugging infrastructure that incrementally adds a multiplexer network to be connected with trace-buffer infrastructure is integrated in the design. It is assumed that the trace-buffers pre-exist in an FPGA and the user can add only an overlay network that routes all possible signals to all possible trace-buffers.

For the comparison of the area utilization, the benchmarks were synthesized with Vivado 2016.1 and the two PConf tool flows. The FPGA architectures used were a Virtex 7 (in Vivado) and a theoretical architecture that is mimicking the basic structures of commercial architectures, but with full transparency on the configurable infrastructure, so that it is supported by the PConf tool flows. All

	Vivado		ILA		ABC	PPD	FPD
	LUT	FF	LUT	FF	LUT	LUT	LUT
b12	224	119	15538	19715	309	343	329
b13	37	51	4092	5552	51	72	59
b14	2165	219	46749	53316	1047	1165	1101
b15	1948	416	26710	33464	1952	2015	1979

Table 3.1: Logic Utilization of the debugging infrastructure for the benchmarks implemented with different tools. The original design is mapped with Vivado and ABC. The two proposed methods are the PPD (partially parameterized debug) and FPD (fully parameterized debug). ILA is the trace-based debugger by Xilinx and ABC an academic synthesis tool that supports PConf.

architectures contain 6-input LUTs for fair comparison.

Multiplexer Network

The proposed debugging architecture (the one attached to each internal signal) needs tuneable connections and tuneable LUTs. If it is implemented with the conventional tools, it needs up to 56× more LUTs to ensure all signals can be linked to the trace-buffers. The proposed architecture has a small impact on physical LUTs for each design. This happens because the debugging infrastructure is parameterized and integrated in the reconfiguration resources. Hence, the number of physical resources remain almost constant. The multiplexer network creates virtual connections between the signals and the trace-buffers, as it was explained in Section 3.3, reducing the debugging overhead significantly.

Integrated Logic Analyzer

In order to support commercial architectures, a similar design and verification process as the one described in Section 3.4 needs to be followed with minor modifications to support commercial FPGA architectures (Xilinx, Virtex-7). The basic difference is that the Vivado's Integrated Logic Analyzer (ILA) core is used, instead of any parameterization-based optimizations. The core has been configured in such way that it is able to debug as many signals as possible, with a trigger and probe set to minimal, to maximize the area savings. However, even though the area overhead of the debugging core is kept minimal, there is a massive increase in the used resources. This effect results in two designs (b14, b15) not being able to be placed on the FPGA, requiring a larger device and repetition of the whole process. The results are depicted in the ILA column in Table 3.1.

With our proposed methods, the only area overhead that exists, is a minimal increase in the number of physical FPGA resources (LUTs) after the installation of the debugging architecture, which is expected, as not all the multiplexers' inputs

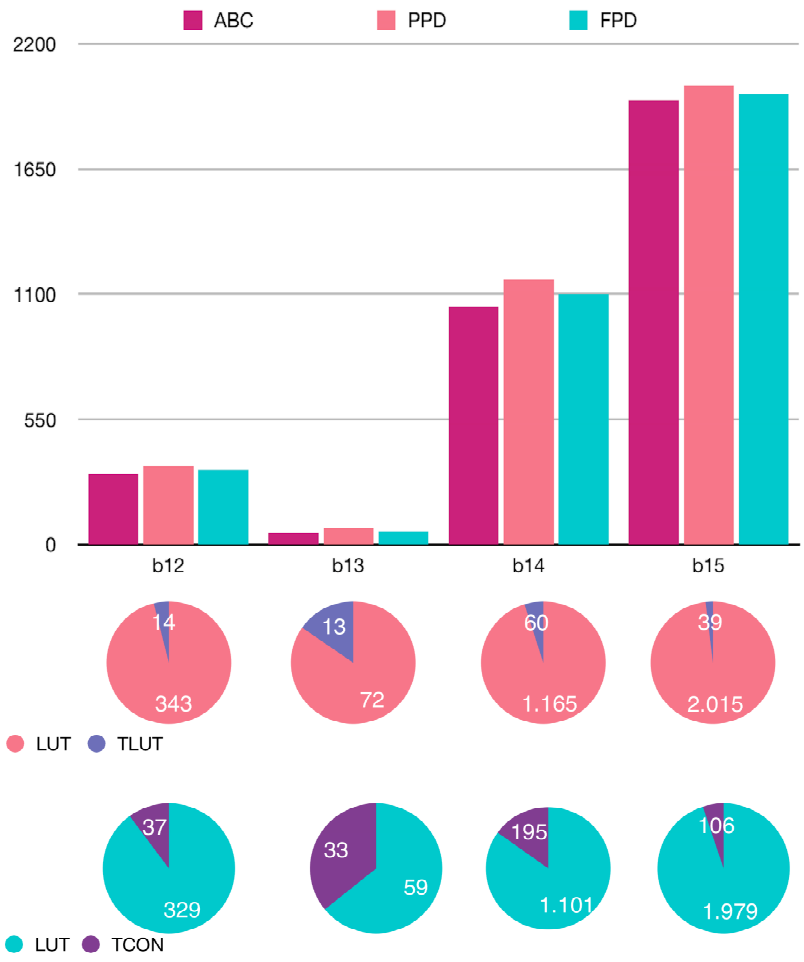


Figure 3.8: Graphical representation of the area results (in LUTs) and the number of parameterized resources (TLUTs, TCONs). The LUT *in the pie diagrams represents the total number of LUTs needed (including the ones with the parameterized resources needed). The* TLUT *and* TCON *show the number of parameterized resources are needed in respect to LUTs.*

are parameterized. There is a large increase in the virtual FPGA resources (TLUTs and TCONs), but with minimal impact on the design's area and performance. The results are shown in Table 3.1. In the table, the benchmarks were first synthesized with the Vivado and ABC tools. The columns Vivado and ABC show the area in number of LUTs for each benchmark without any debugging infrastructure. The ILA column shows the area impact after the instantiation of the debugging core on a commercial device. The PPD column shows the area results obtained with the TLUT flow, were only the LUTs were parameterized, presented as the Partially Parameterized Debugging (PPD) results. The FPD results in the last column indicate the Fully Parameterized Debugging (FPD) infrastructure, where all the debugging infrastructure has been fully parameterized with the TCON tool flow (parameterized routing infrastructure). We can observe that with the proposed tool, we have significantly less debugging resources (shown in the last two columns), compared to the vendor tool (ILA).

Figure 3.8 visualizes the number of actual and parameterized resources needed for the debugging infrastructure. The lower part of the figure shows the area of the virtual multiplexer network mapped in TLUTs and TCONs instead of actual resources. In this figure is illustrated how the the existence of virtual resources (TLUTs in PPF and TCONs in FPD) allows the designs to maintain low area overhead, despite the insertion of the debugging infrastructure. We can observe that due to the virtualization, the area overhead increases up to 10% with the proposed method.

The results indicated as *PPD* contain LUTs and TLUTs, while the results indicated as *FPD* contain LUTs and TCONs. In general, some Vivado implementations can use fewer LUTs for the golden design (Vivado column) in commercial tools compared to the tools that target theoretical architectures (ABC column). This is due to the fact that these tools probably contain better optimizations for numerical operations, compared to the generic theoretical FPGA architectures that are used in academic tools. However, designs with more multiplexers are better optimized using the PConf tools. Especially TCONs are heavily used for the implementation of the multiplexing between internal signals and trace-buffers for the debugging architecture, because multiplexers are well suited for optimizations of shared resources.

One can observe in Table 3.1, that with Vivado 2016.1 the implementations on a Virtex-7 are on average $21.3\times$ larger than the proposed implementations. The proposed architecture (FPD) with the integrated debugging functionality is in fact $11\times$ smaller than the design integrated with Vivado's ILA and $1.9\times$ larger than the golden application(ABC column). Therefore, with an area penalty of approximately 10%, the debugging functionality can be integrated in a given design. However, there are some kinds of design that debug can be more complicated, such as with third-party IPs that realize high frequency, high-dense communication (neural

	Vivado		Proposed	
	AXI-HWICAP	AXI-HWICAP (DMA)	HWICAP	MiCAP-Pro
b12	480	1690	3.2	1.03
b13	500	1684	2.9	0.83
b14	2050	7500	13.8	3.846
b15	2420	8530	8.97	2.49

Table 3.2: Comparison of reconfiguration time between the proposed technique and Vivado-based implementation

networks), or high DSP utilization. In these cases, the proposed approach should be integrated in the Vivado toolflow in a way that it can still access the internal signals within the IP. This is investigated in Chapter 4.

3.6.2 Reconfiguration Overhead

The speed of configuration is directly related to the size of the BIT file and the bandwidth of the configuration port. Here, a comparison of the reconfiguration times between different reconfiguration controllers and the target application is performed. Since the bitstreams that are generated from the application have similar size, the impact on the reconfiguration time is similar. This is depicted in Table 3.2.

If the proposed architecture is integrated in the benchmarks, there is a small impact in the configuration time (from the Boolean function evaluation), as no calculation of a new reconfiguration is needed to evaluate a new signal set, only an evaluation of a Boolean function. Moreover, the proposed technique is instantiated during the original implementation of the design, resulting in a negligible compilation overhead. However, with the conventional tools, in order to instantiate a debug core, the CAD tools need a significant amount of time (minutes to hours). Then, in order to adapt the core, the tools need to repeat the process. All this compilation overhead does not exist in the proposed technique, as instead of debug core adaptations we use Boolean-function evaluations.

There is a significant acceleration in the reconfiguration time, if the MiCap-Pro controller [89] is used instead of the conventional HWICAP. With a PConf adaptation, there is no need to recompile the entire FPGA in order to create a new (specialized) reconfiguration, as only an evaluation of a parameterized configuration needs to be performed. The MiCap-Pro controller takes care of this process. Then, the design needs to perform only micro-reconfigurations in the FPGA and not a full conventional reconfiguration. One microreconfiguration of 1 TLUT takes $64.1\mu s$. Therefore, in order to reconfigure each design we need maximum $3.8s$ (for the b14 design that has the maximum number of TLUTs). This is calculated based on the total number of TLUTs in each benchmark. This metric is 3 orders

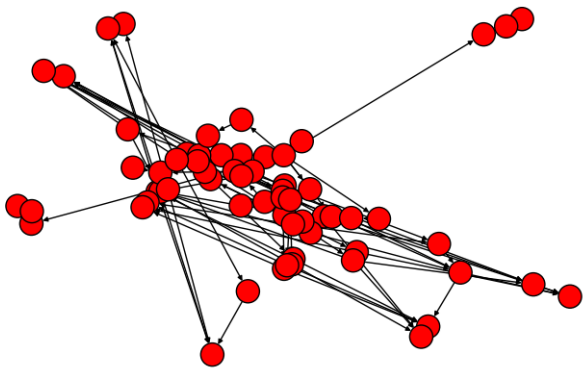


Figure 3.9: Graphical representation of the DUT

of magnitude faster than the conventional reconfiguration time without parameterized configurations and the Xilinx AXI-HWICAP reconfiguration controller, as it is depicted in in Table 3.2.

3.6.3 Signal Ranking

Here the results of the signals selected from Vivado ILA and the proposed technique are compared.

With Vivado's ILA, only up to 30% of the internal signals could be instantiated in the ILA core. However, only 10% of the signals could be fully integrated in the design (to be routed and create a bitstream). Moreover, with ILA, the designer needs to manually select all the signals of the synthesized design. Then the tool emerges on an additional compilation overhead to install the extra infrastructure. This dramatically increases the overhead of the FPGA in terms of LUTs (the b12 benchmark for example becomes up to $70\times$ larger) and an additional compilation overhead is needed to install the debugging functionality (up to one hour). This process needs to be repeated if the correct signal set is not selected. However, with the proposed method, the integration of the debugging functionality is done without any compilation overhead. The debugging infrastructure is automatically integrated in the DUT before synthesis. Additionally, the designer doesn't preselect signals. All internal signals are selected automatically. Then the signals can be transferred for analysis. Therefore, more signals can be traced (up to 100%)

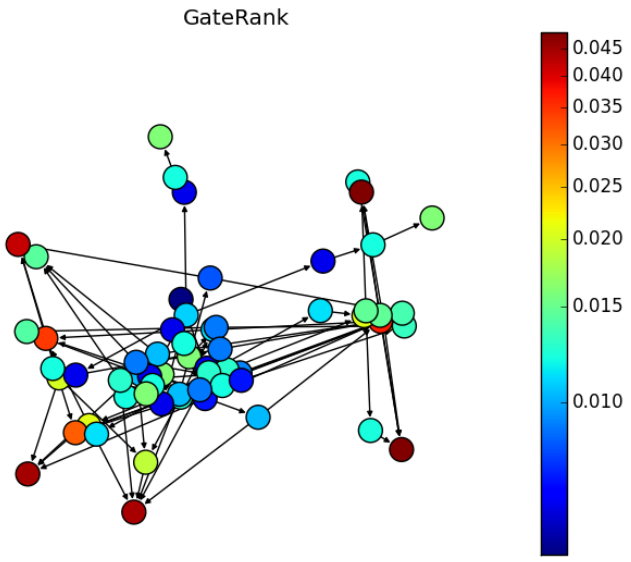


Figure 3.10: Graphical representation of the DUT, with the different colors showing the gaterank of each node

with the proposed technique, in comparison with ILA that can trace up to 10% of the target application's signals.

In the case that there is increased routing congestion, the designer can opt to use the SignalRank and GateRank algorithms to further analyze the design before implementation. These two techniques are integrated in the normal FPGA flow and can be initiated when the design is synthesized. After completion of the ranking, the techniques can influence the debugging framework installation. SignalRank will provide the initial signal congestion to define the number of trace-buffers needed. GateRank will provide the fault congestion to create efficient signal sets, based on their ranking.

For this work the designs of the ITC benchmarks have been used, to demonstrate the impact of the proposed algorithms. After synthesis, a directed graph has been created, of the AIG representation of the circuit. This is shown in Figure 3.9. Then, the GateRank algorithm has been applied. The impact of the algorithm in Figure 3.10 can be observed. The trace-buffers of the design can be adapted by choosing to observe the signals that are above the median GateRank. This totals in observing only 20% of the signals, reducing the area overhead by 17%.

The FPGA can suffer from bugs that are either physical or design errors. Phys-

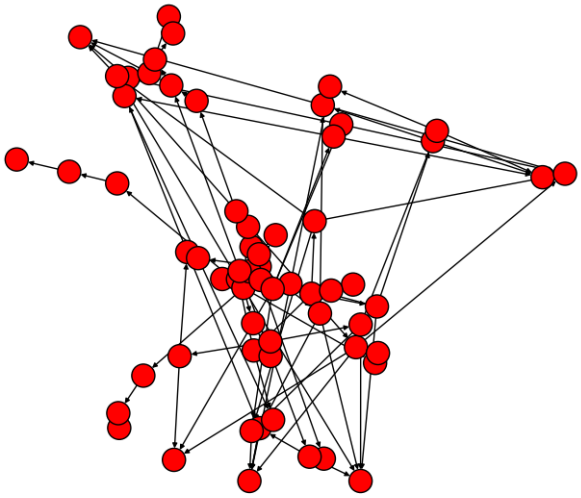


Figure 3.11: Graph of the DUT, with the accumulated bugs

ical errors are the errors that happen spontaneously in the FPGA fabric and they can normally be solved with reconfiguration. They include bit flips, two nets used by the same node, two nodes used by the same net, opens and redirected nets. Design Errors are the ones that have been proven to be discovered late in the design and verification process and include signal source errors (30% of all errors), missing logic, such as missing instance, missing input, missing term, or missing state (26% of all errors) and wrong inversions and unconnected inputs (17% of all errors).

In order to evaluate the quality of the GateRank-based signal-sets, the design is injected with different types of bugs. It has been assumed that the design was synthesized despite the critical physical and design errors that can occur after the fault injections. In order to achieve that, the errors have been injected directly to the graph of the synthesized design. All the errors were assumed to be detected with the worst case scenario. That means, at the end of the fault propagation list. The results indicate the maximum number or steps (tracing cycles) needed to locate the source of the error. The errors injected count for 10% of the total nodes of the DUT and they are both physical and design errors. This is considered an extreme scenario, as the design would never be able to synthesize with so many bugs.

For the DUT, all errors were detected in the first signal set, as the first set was the one selected with the highest GateRank. This made it more likely to detect the faults, as the nets with the highest GateRank had the most connections. Thus, the

source of the error was always located in the first step. The graphic representation of the modified design is shown in Figure 3.11. Here, we can observe the difference in the connections compared to Figure 3.9. The connections of the nodes are significantly less, due to all the accumulated bugs (on 10% of all signals), that have a total effect on approximately 50% of the signals. This is an extreme scenario that is designed only for experimental purposes, as a design would never be synthesizable with so many bugs.

3.7 Conclusion

FPGAs have been increasingly used as prototyping platforms and in various applications. Due to their reconfiguration centered functionality they can achieve significantly higher operating frequencies allowing designers to increase their verification coverage by several orders of magnitude. However, in case of unexpected behaviour, the internal FPGA observability is limited. In this chapter we have presented an integrated debugging method that offers extended internal observability. With integration with the parameterized Configurations tool-set, the designer can rapidly trace different sets of signals. The proposed tool offers low area overhead for the debugging infrastructure and less debugging cycles, with the use of signal ranking techniques. The proposed techniques integrate the hardware infrastructure and optimize it alongside the original design, with an area penalty of 10% - 30%.

4

Integrated Post-Silicon Validation for FPGA Overlays

This chapter presents my approach to enhance the observability of VCGRA-based designs for functional debugging. The proposed approach is a custom-made in-circuit-debugger that is installed during the initial recompilation and can be used to rapidly trace functional errors in high-performance computing applications, that can be implemented as VCGRAs. This chapter describes first how a VCGRA is constructed and then how the proposed architecture can be integrated in a VCGRA, so that the latter can be debugged.

4.1 Introduction

With the emergence of FPGA overlays and VCGRAs, the current debugging techniques cannot efficiently debug these new VCGRA implementations. The use of current in-circuit debuggers will have two possible effects on debugging VCGRAs: a large area needed for trace-buffers in order to store all the debugging information and manual debug.

In the previous chapter, we were focused in providing a low-overhead debugging architecture for ASIC designs, that use FPGA prototyping as a step to reduce the (simulation-based) debugging and verification cycle. Here, we target FPGA architectures that implement FPGA overlays and VCGRAs.

As the FPGA architectures become more complex and as they integrate cus-

tom components (such as VCGRAs), there is a growing need for more efficient FPGA-based debugging. The designer should have a hands-off debugging without a detailed knowledge of the underlying hardware, and early and easy access to multiple signals with an automated tool. In this work, we investigate each of these problems, and provide a framework that is able to integrate debugging infrastructure into a VCGRA, for efficient FPGA-based debugging.

In more detail, in this chapter, a semi-automated in-circuit debugging method is introduced, for theoretical and commercial FPGA architectures, that integrates FPGA overlays with debugging properties, with minimal user intervention. A custom two level overlay architecture is proposed where the debugging instrumentation is virtual, automated and it is integrated in the design during the original compilation. The debugging infrastructure is added incrementally and optimized alongside the target FPGA. The first overlay level is the VCGRA, whereas the second overlay implements the proposed debugging functionality.

4.1.1 Prerequisites

The virtual and overlay architectures provide more flexibility. The designers should be able to debug their virtual architecture designs in-system, without detailed knowledge of the hardware's limitations.

VCGRAs consist of a large number of processing elements (PEs), laid out in a grid pattern, and Virtual Channels (VCs), that are a communication network connecting the PEs. Each PE is a coarse-grained element (realized mainly using LUTs) and it is capable of computing incoming data and pass it on to the next PE, via an adjacent VC. The VCGRA is implemented using reconfigurable connections, with the assistance of the PConf flow. A VCGRA needs to be changed when the application implemented on the FPGA needs to change or adapt. Since some of the VCGRA's inputs are implemented based on the PConf tool flow, that means that the VCGRA needs to be changed/adapted infrequently, as the reconfiguration time is the bottleneck of these architectures. Hence, instead of implementing the VCGRA's instruction inputs as regular inputs, with PConf these inputs are implemented as constants and the FPGA overlay is optimized for these constants [90], as it was analyzed in Chapter 2.

Academic works have leveraged FPGA overlays to improve their debugging methods. In debugging with the use of FPGA overlays, the authors have used mainly incremental compilation, to insert trace-buffers, after the design is mapped and finalized on the FPGA [31,32,61]. These tools support only academic FPGAs and they use the remaining FPGA logic to construct virtual routing networks for signal tracing and to add debugging circuitry. However, the size of the overlay and the amount of its resources and routing needed can make its construction challenging and can affect the original design's timing and critical path. Moreover, in the

above-mentioned methods, the researchers create overlays to assist the debugging process, without the possibility to debug the FPGA overlay itself.

4.1.2 Contribution

The proposed method combines the advantages of enhanced internal signal observability and fast FPGA reconfiguration in one tool flow. It has four main parts: a parsing and mapping step to create the VCGRA, a design step to add the debugging infrastructure incrementally, a parameterization step that is used in the case of virtual FPGA architectures in the DUT and an online step that adapts the debugging methodology to a target design. This approach has some benefits over other debugging methods:

- Extended internal observability: By integrating with the PConf approach, the designer can trace sets of signals fast, completing the debugging process within time constraints.
- Reduced area resources: By adding the debugging infrastructure on a higher-abstraction level (overlay architecture) and dynamically adapting the added on-chip memories, the extra resources are reduced.
- Automated tool: The designer has just to select the DUT and to run the flow to create the debugging infrastructure for the VCGRA in order to perform signal tracing. Semi-automated interventions reassure that the DUT's debugging hardware remains minimal and fully parameterized.

The main contribution in the debugging integration is the possibility of adding debugging hardware on a higher abstraction level. This is done when the application is transformed to a VCGRA. In this way, the signal tracing hardware is co-designed alongside the VCGRA. A netlist is generated that has integrated debugging functionality and can be adapted without affecting the VCGRA under test [80].

The contribution of this chapter is illustrated in Figure 4.1, where it can be visualized how it is connected with the rest of the thesis. It is shown how this chapter enhances the debugging step with a second overlay (VCGRA generation) and a more automated debugging process (SDA).

This technique integrates in-circuit debugging functionality in an FPGA overlay. The added functionality enables the user to debug large designs mapped on FPGAs without significant area overhead. A realistic application is used in order to validate the feasibility of the debugging architecture. In the experiments section, an exploration of the area and runtime overhead is realized, to validate the scalability of the method and of the signal integration algorithms, while increasing the design size and decreasing the size of the available FPGA resources. In

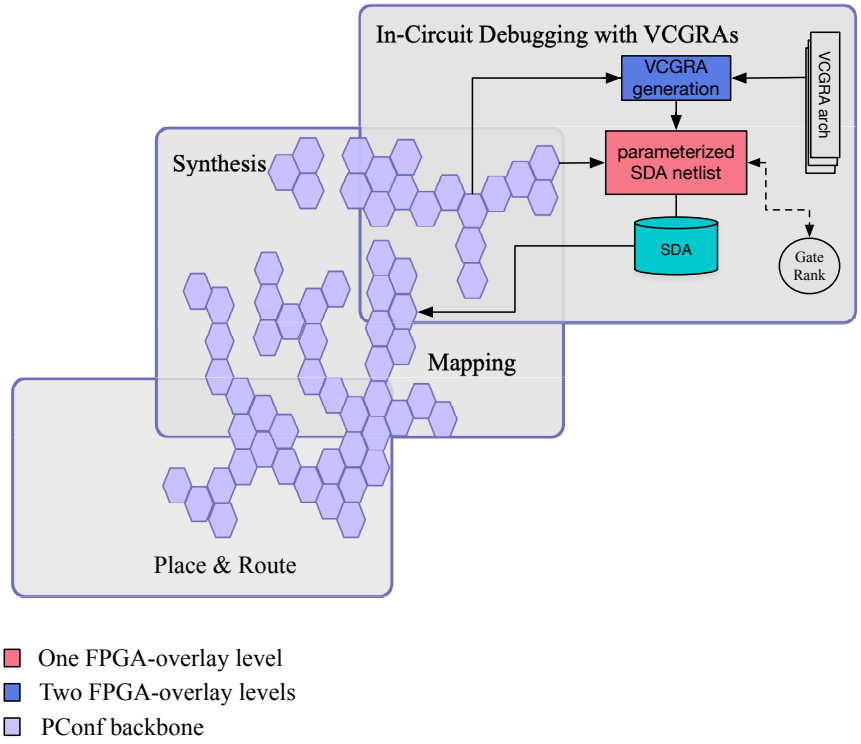


Figure 4.1: Visualization of this chapter's contribution

addition to the added functionality, the efficiency of the proposed tool is explored, for both academic and commercial FPGA architectures. Moreover, an on-the-fly adaptation of the debugging infrastructure is proposed, that achieves a balanced trade-off between area overhead, signal observability and compilation time.

This chapter analyzes how a second FPGA overlay can be applied on a specific overlay (VCGRA) architecture, in such a way that it will provide in-circuit debugging functionality and without altering the original design. First, parameterization of the internal signals of a given design under test, on a VCGRA level is proposed. Then, signal compression of mutually exclusive signals (during runtime) is used to reduce the area overhead of the debugging circuitry, while increasing its flexibility. The entire concept of debugging with FPGA overlays is applied for two different FPGA architectures. Thus, three contributions are made:

1. This chapter introduces FPGA overlay debugging methodologies in both academic and commercial FPGAs. Integration with state-of-the-art FPGA-design platforms is also presented, that allows researchers to map their designs in commercial FPGAs.

2. The unique architecture of the VCGRAs is leveraged, alongside its layer-by-layer data path, to selectively add debugging functionality in such a way that with minimal monitoring, it can perform on-silicon debug, where the designer can observe the results and adapt the VCGRA. Both parameterized configurations and conventional dynamic reconfigurations are used, as well as an AXI-wrapper that packages the VCGRA-SDA system. This is introduced as a third-party IP. The creation of an IP will allow real applications of the proposed system. This method can be used by integrating the IP in a design.

3. A tool flow that applies the above-mentioned techniques to a given design automatically. A complete tool that co-creates the VCGRA and the debugging circuitry is provided, where the same automations that can create a VCGRA can generate an SDA as well. A second layer is created, that provides dynamic signal tracing in the VCGRA. The tool co-designs the VCGRA and the SDA in such a way that they are interconnected. The tool is integrated in the Vivado tool flow and it can rapidly generate (offline) the files needed for the tool to create the IP and install both the VCGRA and the SDA.

The remainder of this chapter is organized as follows: Section 4.2 presents the proposed two-overlay technique and its architectures and Section 4.3 presents the adapted design flows that support them. Section 4.4 describes the experiments that were conducted and their results, followed by the Conclusion in Section 4.5.

4.2 In-Circuit Debugging for FPGA Overlays

This section presents the main approach to enhance the observability of VCGRA-based designs for functional debugging. The Superimposed Debugging Architecture (SDA) approach is a custom-made in-circuit-debugger that is used to rapidly trace functional errors in high-performance computing applications, that can be implemented as VCGRAs. In order to use the SDA, first a VCGRA needs to be constructed from a dataflow graph representation (or a VHDL design) of an application. Hence, this section describes first how the VCGRA is constructed and then how the SDA can be integrated in a VCGRA, so that the latter can be debugged.

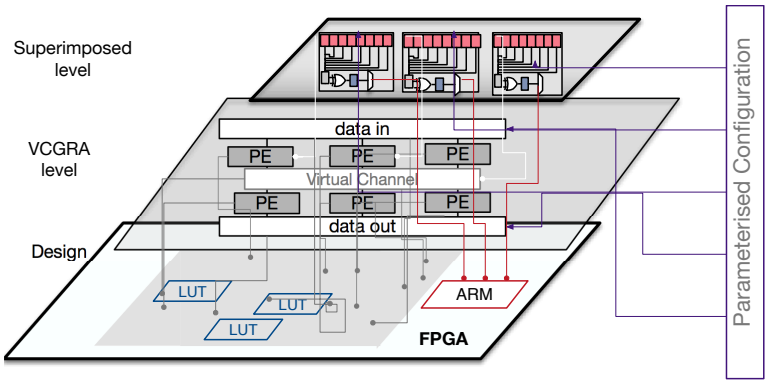


Figure 4.2: Overview of the multiple-level architecture showing the FPGA and the two virtual levels (the VCGRA application and its adjacent superimposed debugger).

4.2.1 VCGRA Overview

A VCGRA is a hierarchical architecture consisting of several levels, where each level consists of logic functions (PEs) and connections between them (VCs). In order to build a custom VCGRA application, the first step is to design the PE and VC, according to the target application. Different functionalities form different PEs. Multiple PEs are connected with VCs, forming one level of the VCGRA. Then, multiple layers of the VCGRA form the VCGRA grid. The grid's structure is described by the number of PEs in each level of the architecture and the elements' input and output bandwidths. The VC configuration controls the data flow among the VCGRA's levels, and the PE configuration controls the operation of a PE. The size of the configuration bitstreams is defined by the number of PE operations and the number of PEs within a VCGRA level and is determined during design time. The VCGRA is depicted in the middle layer of Figure 4.2 and in the left part of Figure 4.3.

All components of the VCGRA are described in VHDL and they contain parameters that later will be used by the PConf backend, in order to reduce the utilization of LUTs in the target-FPGA. The PEs are created by parsing and converting a textual description of a synthesized application into a netlist of PEs. PEs are designed as state machines. Their inputs perform an operation and their output is a buffer that saves the result for one cycle. The inputs and outputs of the PE can be multi-bit and they are defined during the implementation stage, based on the target application. VCs have one multiplexer at each output in order to connect one specified input with its configured output. In order to select the inputs, a multiplexer with multiple select inputs is added (Figure 4.2). Consequently, this

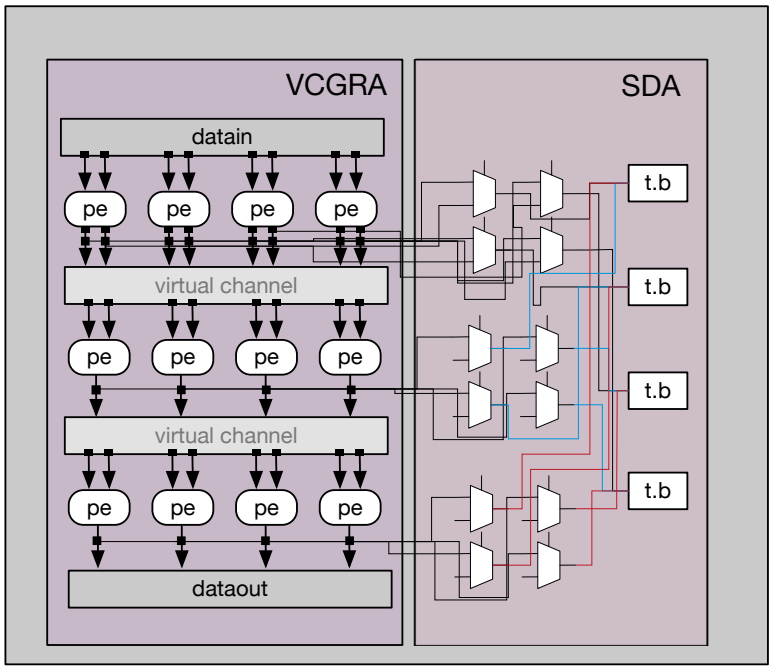


Figure 4.3: The architecture that integrates the SDA in a VCGRA.

architecture requires a lot of routing resources. In order to reduce the routing resources, all select-lines of the multiplexers are considered infrequently-changing constants, and subsequently parameterized. Therefore PConf can be used on these parameters. Since PConf uses resources sharing, the debugging resources can be reduced.

4.2.2 Architecture: SDA Overview

The SDA is a network of multiplexers that will connect to trace-buffers that store inputs and outputs of a VCGRA layer. This circuitry is co-designed during the initial VCGRA implementation. The SDA technique aims to tackle the drawbacks of the current in-circuit debuggers in the following way:

- The use of on-chip resources (which can also impact the design's timing performance and create additional debugging requirements) is optimized, by minimizing the signals to be traced with the SDA and by using a small amount of multiplexers and trace-buffers, tailored to the requirements of the VCGRA application.

- The need to recompile and reprogram the design to observe different sets of signals is eliminated with the use of the PConf approach.
- By creating a layer-by-layer debugging approach and reconfiguring to debug each VCGRA layer, the area overhead is controlled.
- The observability of the internal signals is enhanced with the debugging meta-layer (SDA).

The number of the trace-buffers needed for debugging is equal to the number of PEs and their size (their length) is equal to the length of the traces one wants to store. The tracing cycles (number of times tracing has to be performed in a debugging cycle) can vary based on the demands of the application. Some of the debugging infrastructure's elements, such as multiplexers, are incrementally added at the outputs of the PEs, in such a way that they use minimal LUTs. This is achieved by tracing only specific signals of each layer (based on the application) and not all possible signals. For commercial FPGAs we cannot yet use PConf on the routing infrastructure so the multiplexers have to be actually implemented in LUTs (or only TLUTs). However, for academic FPGAs the debugging circuitry is completely implemented in the FPGA's reconfiguration resources. This network of multiplexers will subsequently be connected to trace-buffers to perform functional debugging. The SDA is shown in the upper layer of Figure 4.2 and in the right part of Figure 4.3.

The SDA distinguishes from the state-of-the-art debuggers in two basic ways: First, the basic elements that are observed are layers of PEs and not solely signals. Hence it operates on a higher abstraction level, accelerating the debugging process, without reducing the high observability, as there is a guarantee of a connection between an important signal and a trace-buffer. The second distinguishing element is the fact that the SDA can be installed on specific components of the design (tracing specific signals) and rapidly check for possible bugs, compared to the long recompilation cycles that are needed to check different sets of signals in the whole design, as in conventional in-circuit debugging.

The SDA is structured in layers, similar to the VCGRA. Each layer has a number of multiplexers that is in accordance to the number of PEs per layer. Each multiplexer is connected with one PE. All layers are then interconnected with trace-buffers using parameterized connections. These are statically determined at design time. The SDA's components are described in VHDL and contain annotations that can also be used by the PConf backend to control the reconfiguration, determining which VCGRA layer is being traced at a certain time.

The VCGRA-SDA system mitigates the use of on-chip resources in two different ways. For academic FPGAs is achieved by creating a PConf, that compresses signals that are not used at the same time. These signals are the selection bits of multiplexers that connect the internal signals of various layers with trace-buffers.

Multiple signals can be described in one (tuneable) LUT, as it was analyzed in the previous chapter. These LUTs contain Boolean functions that upon evaluation can describe different signal sets, as it is used in the PConf tool flow. This mechanism is thoroughly explained in Chapters 1 and 3 and in previous work [82, 83].

The VCGRA architecture is leveraged to build an efficient SDA. The architecture of the VCGRA is detected by our tool and signal tracing infrastructure is installed on specific signals (outputs of the PEs of each layer). In that way, not all signals need to be observed, but only a subset. Due to the architecture of the VCGRA, the debugging can be done layer-by-layer, allowing massive savings in area resources as the installation of debugging cores is avoided. Additionally, the internal observability remains constant, as the designer can select the exact signals he/she wishes to be traced, in RTL level. This is an added benefit that is an obstacle during conventional debugging of a design, where the internal observability of the third-party IPs is limited.

The in-circuit debugger is extended with a (minor) repairing functionality. In that case, the designer doesn't aim to locate the source of the bug in order to correct the design error, but to detect the PE where the bug occurs and then reload the entire bitstream of the PE. The elements of one PE can be reconfigured to be repaired, instead of repairing the bug by redesigning. This can be used for soft-error and for design errors. This technology can be used as a fail-safe mechanism that can be enabled in the case of safety-critical applications. This functionality can be applied with *microreconfiguration* [89]. Microreconfiguration is a 3 step technique that involves reading, modifying and writing back specific frames: using the frame address, the frames containing the truth table entries of a PE or VC, are read from the configuration memory. Then, the current truth table entries are replaced by the requested bits (saved in an FPGA memory). Finally, by using the same frame address, the modified four frames are written back to the configuration memory, thus accomplishing the microreconfiguration and efficiently repairing the erroneous subgrid, with the use of the with the use of the XhwIcap_setClb_bits function [88], that returns the exact location of a bug. Hence, as long as the bug is located in four (or less) consecutive frames, these frames of the erroneous PE are reloaded to create the self-healing overlay.

4.3 VCGRA-SDA Toolflow

In order to create the VCGRA-SDA architecture (or similar architectures), from a target application, it is necessary to use a toolset that allows the creation of such architectures, configurations, debugging functionalities, hardware implementations and integration with the vendor tools, with minimal user intervention. Therefore, the third generation VCGRA tool described in Chapter 2 and depicted in Figure 2.16, and the PConf tools have been extended, to integrate the SDA in the

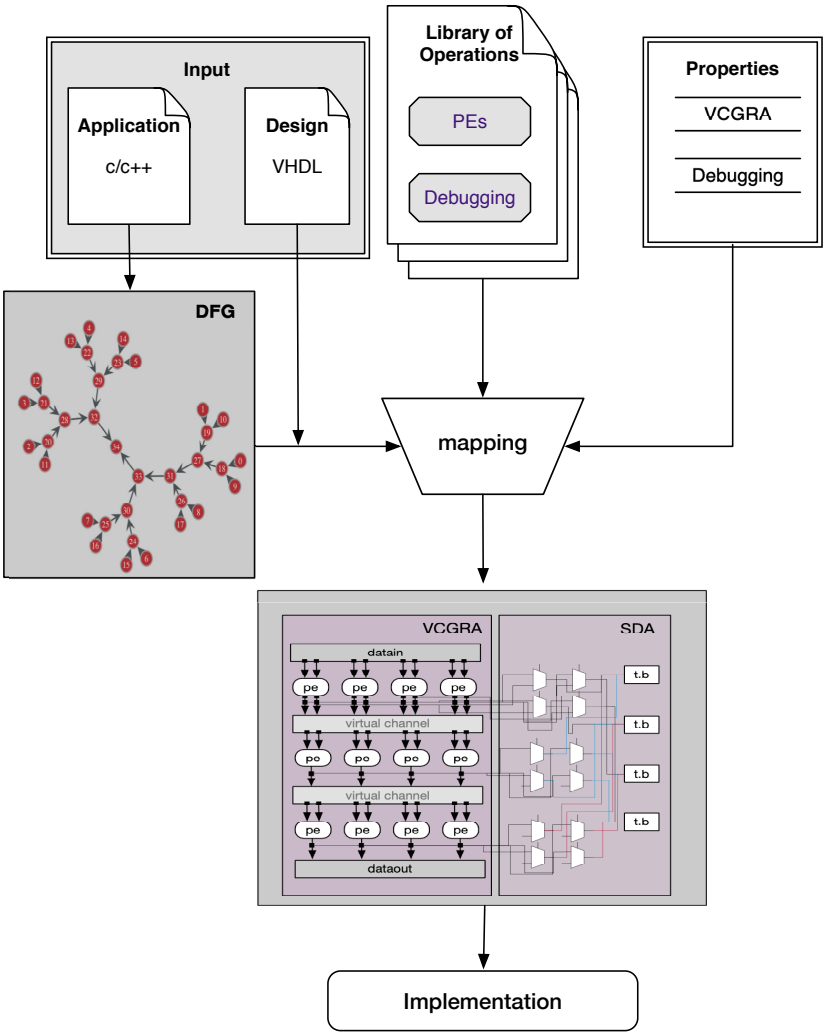


Figure 4.4: The tool flow that enables the SDA-integrated VCGRA implementation.

target application automatically. Figure 4.4 offers a high-level representation of the tool flow that has been used, to analyze and instrument systems with VCGRA and SDA. This is based on two standalone tool flows [38, 90], with debugging tooling integrated in both toolchains.

The SDA could be designed in an HLS language as well, as debugging would be easier when using a higher abstraction hardware programming language. Once the implementation in the high-level programming language is debugged, the final circuit should also be correct, assuming that the HLS compiler is error-free. Hence, it would be possible to incorporate this contributions in higher abstraction hardware programming languages. However, it would be difficult to connect these techniques with the microreconfiguration system, that is used for repairing erroneous frames, as the underlying functions of this functionality couldn't be combined with a high-level language. Therefore, HLS languages can be used for the SDA, by inserting the functionality at a higher level (and not when the VCGRA is constructed).

4.3.1 Application Mapping Toolset

In order to map an application in a target (pre-defined) VCGRA architecture, an algorithm (represented as a graph or as a VHDL design), the properties of the overlay and the library of operations are given as inputs. More specifically, the library of operations is an HDL representation of all possible operations needed to create the functionality of the PEs and the debugging infrastructure. The properties of the VCGRA are the shape (number of layers, and dimensions of each layer) and the number of inputs of the VCGRA. From this file, the properties of the SDA are automatically extracted (the shape and number of inputs and outputs). The input graph is a dataflow graph and can be automatically obtained using a small tool, which has been developed with the help of third-party libraries, such as the graph-tool [116], in the case that the application is described in a language such as C/C++. However, it is also possible to provide as an input a VHDL design. The output is a VCGRA architecture that can be mapped either with the PConf tool (for less resource utilization) or with the vendor tools (for better performance).

4.3.2 VCGRA-SDA Toolset

While processing the HDL templates of the architecture and its components, the tool generates VHDL and other associated files, that facilitate the mapping of a target application to a target architecture, the creation of the SDA and the configuration vectors. The SDA Generator creates the modules and the architectural components for the SDA integration in the VCGRA. The proposed tool requires the description of the applications and the description of the VCGRA as inputs.

First, the tool analyses the inputs required by the applications and whether they can fit in the VCGRA. If they fit, the tool adds multiplexers in the appropriate positions to interconnect PE outputs and trace-buffers. The output of the SDA tool is the description of the connection of inputs (PE outputs) and outputs (trace-buffer inputs) between PEs and trace-buffers and the necessary multiplexing and the VCGRA configurations of the target applications. This is visualized in Figure 4.4.

Additional inputs required in order to construct the SDA are the shape of the target overlay (VCGRA), the number of the VCGRA's inputs and the levels of PEs/VCs. Moreover, a library of operations is also needed, that describes the functionality of the PEs and the debugging elements. The output is a configuration that can be later translated into bitstreams with the use of the CGRA-Configuration Generator [38].

At the end of the SDA-generation step, the SDA tool has created a meta-layer with the debugging infrastructure on-the-fly, that is subsequently added in the VCGRA incrementally. In order to create the debugging infrastructure, the trace-buffers are installed. They are equal to the number of PEs of one layer, as it is adequate to trace one layer of the VCGRA at any given moment, since changing between VCGRA layers is very efficient, due to the rapid reconfigurations. Instead of pre-reserving an area for trace-buffers (as in state-of-the-art ELAs) or instead of using all possible leftover resources after place and route (as in state-of-the-art academic tools), the minimum possible amount of trace-buffers and additional logic needed to debug just one VCGRA layer are used. Therefore, this architecture can reuse the same trace-buffers among different layers. In fact, the routing is constructed between the SDA, the trace-buffers and the target VCGRA layers. Therefore, the number of the trace-buffers scales based only on the size of one layer and not of the entire VCGRA. Similarly, the debugging requirements scale in a linear way for larger applications, as they are based on the number of PEs per layer and not on the application. In that way, the architecture can scale well.

4.3.3 Online SDA Generator

In order to be able to reconfigure a VCGRA architecture in vendor tools, certain steps need to be followed. After the generation of the VCGRA and the SDA, an AXI-Lite template is adapted specifically for the VCGRA and the SDA. The AXI protocol is a point to point interconnect designed for high performance, high speed microcontroller systems. AXI interfaces can be connected to the PS. The AXI-Lite interface is an IP that provides a point-to-point bidirectional interface between a user IP core and the AXI. Then, the entire application that is now mapped on the VCGRA, containing the adjacent SDA, is added as a user IP in the vendor tools. Hence, all associated IP files, alongside the hardware description of the application, are packaged in a form of a user IP and an AXI-wrapper is created.

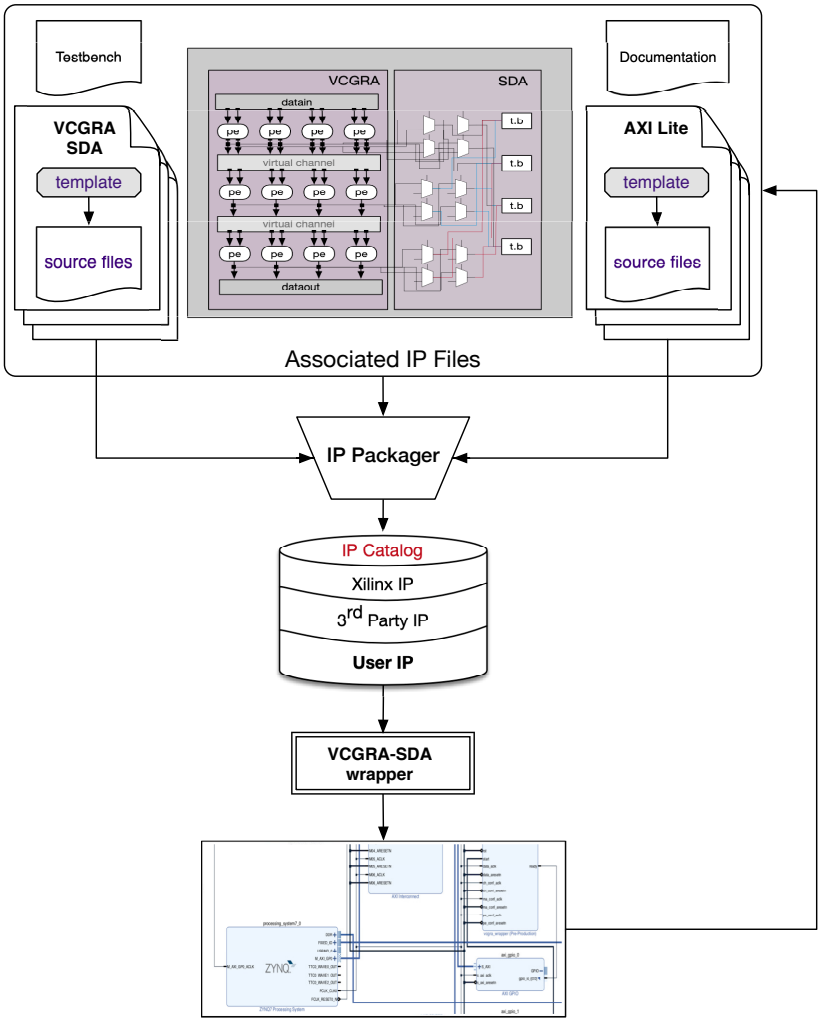


Figure 4.5: IP design flow for the SDA-integrated VCGRA

This wrapper is needed, because the VCGRA itself has been implemented in pure VHDL and is not limited to a specific FPGA architecture. It is used to enable communication between the VCGRA and the host system and provides access to the inputs needed to configure the VCGRA and to the data inputs and outputs. The interface-templates provided can be used to create interfaces which are usable on Xilinx Zynq-SoCs, further interfaces for different FPGAs from other vendors are possible. A little software library that is provided eases access to the AXI interfaces used for transferring configuration and data. Additionally, some binary signals are also provided by the interface and are used to synchronize data transfers and to report the hardware state. The online part of the flow is shown in Figure 4.5.

The processor examines each new layer (set of signals). In case a bug is detected, the process can be interrupted in order to microreconfigure the VCGRA, (if the repair functionality is enabled in the case of soft-errors). In that way, the method has integrated self-correction capabilities, that can be used for safety-critical applications.

4.3.4 Theoretical Architectures Tool Flow

In the case of theoretical architectures (instead of using the vendor tools), the PConf toolflow is used. A distinction is made at this point between theoretical and commercial architectures. It is worth to be noted that the PConf tool flow conceptually works for any (SRAM-based) architecture but the vendors do not allow access to information on the routing resources (which is a requirement of the PConf) and therefore, for the time being (as long as the vendors have not integrated this tool flow in their own tools) designers cannot use the PConf concept on commercial architectures. We therefore can only show the PConf (and the debugging methods that leverage it) on theoretical architectures.

The PConf option can be used when the FPGA resources are scarce, and more flexibility (more integration and faster reconfiguration) is needed for the SDA. Here, the user determines the VCGRA's settings that will reconfigure the VCs and PEs to realize a target application, as described above. During synthesis and mapping, a textual description of the target application is converted into PEs. The PEs are mapped into *virtual* PEs of the VCGRA. Next, with a custom router, optimal connections are created between the PEs and VCs. The multiplexers connecting the signals to the trace-buffers (whose values change infrequently compared to the rest) of the PEs and VCs are mapped on virtual LUTs and virtual Connections with the PConf tool [147]. In that way, the required number of LUTs is reduced, as the debugging infrastructure is implemented using the reconfiguration resources (that are usually not available to the user).

As soon as the architecture of the VCGRA is constructed, textual settings are also extracted from the VCGRA, in order to construct the SDA. The settings

needed are the data bandwidth and the basic PE operations. Then, the multiplexers are inserted into the VCGRA's reconfiguration resources. At this point the granularity and the functionality of the VCGRA are defined, alongside all the possible ways the PEs and VCs can be interconnected with the debugging network.

Figure 4.6 is a high-level representation of the tool flow that we used, to analyse and instrument systems with VCGRA and SDA. This is based on [57], with in-circuit debugging tooling inserted. The functions in the VCGRA endure very little changes. If the frequency of the reconfigurations is too high, this will induce a time overhead during run time, which may be unacceptable. Next, with a custom router, optimal connections are created between the PEs and VCs. By using TLUTs and TCONs the utilization of the LUTs is reduced, leaving more flexibility for the installation of the debug infrastructure for the VCs and PEs.

4.3.5 PConf Adaptations and Limitations

In terms of debugging, PConf is very efficient for debugging theoretical architectures due to the fact that the reconfiguration resources are manipulated. This cannot yet be done in a commercial FPGA [82]. Therefore, it is possible to use PConf for commercial FPGAs but with some restrictions. Additionally there are limitations in the tools, due to memory requirements. The tools that support the VCGRA mapping on academic FPGAs can support a VCGRA of a few hundred LUTs. Therefore, it is able to synthesize one PE (of approximately 200 LUTs) but not an entire VCGRA (few thousand LUTs). Hence, due to these two limitations (memory shortage and inefficient handling of debugging resources), the commercial version of the approach without the PConf has been implemented, that is leveraging architectural benefits of the VCGRA, the Vivado tool and conventional reconfigurations instead of parameterized.

4.4 Results and Discussion

To verify the applicability of the SDA, we used for the experiments a design that can be transformed into a VCGRA. An image processing application has been used that implements a Sobel Edge Detection filter [129] and occupies 1% of the FPGA's capacity (Virtex-7). Edge detection is applied in image processing applications with the use of convolution filters, such as the Sobel filter. A pixel is calculated by the following equation:

$$G_x(x,y) = \sum_{i=1}^{3}\sum_{j=1}^{3} I(x+i-a, y+j-a) \times S_x(n+1-i, n+1-j) \quad (4.1)$$

Where the filter's set-point is in the middle of the filter-mask, and:

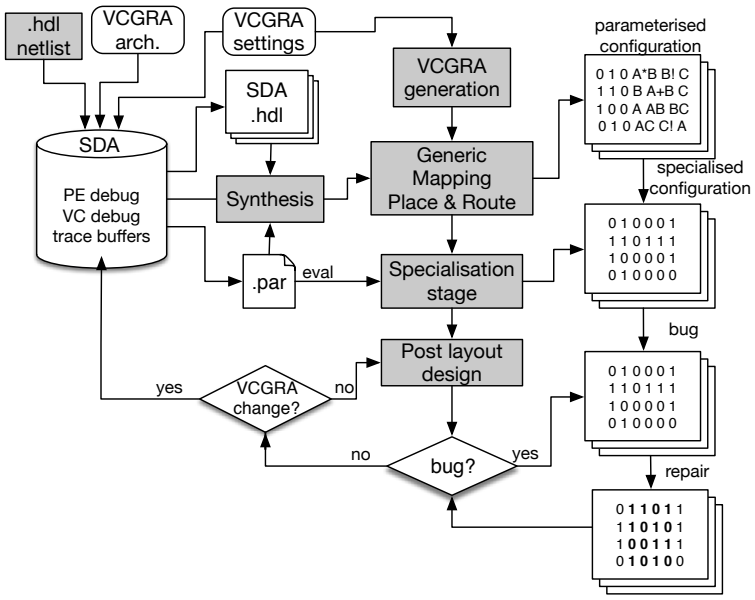


Figure 4.6: The tool flow that enables the SDA-integrated VCGRA implementation. The repair phase shows the microreconfiguration of specific frames, to repair a bug by resetting the parameters.

- $G_x(x, y)$ is a pixel in the result image of a convolution in vertical or horizontal direction.
- I is the input image. The corresponding pixels underneath the mask are addressed relatively, depending on the filter set-point.
- S_x is a filter in either vertical or horizontal direction.

For the experiments, first, the application's architecture is converted into a VCGRA and then the debugging infrastructure is added incrementally. It is shown how it can be implemented with a smaller area using the proposed techniques.

Starting from a graph representation of a Sobel edge algorithm, a VCGRA grid can be created. The implementation depends on the requirements of the design, such as the graph's depth, the PE-operations, the data paths' bitwidth and whether there is a need for (parameterized) reconfiguration. If the target-FPGA is large enough, there might not be a need for reconfiguration. However, when the target FPGA is not large enough, PConf can be used.

The DUT has two different operations (add, mul), between two neighboring pixels and their corresponding filter coefficients. Hence, the operations are mod-

eled as PEs, with 2 inputs: a pixel and a filter coefficient. The edges of the graph can be modeled as VCs and a VCGRA can be constructed. The architecture has been implemented with the proposed tool-flow. Hence, the PEs describe the different mathematical operators to be used. The tool can use the different operators to create the VCGRA grid. In this case the SDA needs only a trace buffer and some additional logic. The tracing infrastructure can debug one layer. Hence, the added circuitry is equivalent to the maximum number of PEs per layer.

Here, the minimum possible SDA that is needed is multiplexers that connect the output signals to trace-buffers for each layer. With a conventional ILA, the designer would have needed a fixed amount of trace-buffers pre-installed, that trace a large fixed number of signals, before the nature of the VCGRA application is known. Here, after a trigger, the SDA is reconfigured and the multiplexer network realizes point-to-point connections between the trace-buffers and the PEs performing the operations.

4.4.1 Offline Comparison of Area Utilization.

For the comparison of the area utilization, the DUT was synthesized with Vivado 2016.1 and the PConf tool flow. In order to compare the two tools, the PEs, VCs and the whole VCGRA target application of the DUT were used. The FPGA architectures used were a Virtex 7 (in Vivado) and a theoretical architecture that is supported by the PConf tool flow. All architectures contain 6-input LUTs for fair comparison.

Commercial FPGA Architectures

In order to debug with the SDA on commercial FPGA architectures (Xilinx, Virtex 7), instead of tracing sole PEs per reconfiguration (as in academic FPGAs), one VCGRA layer was traced per reconfiguration. This design choice has been realized in order to reduce the reconfigurations and to increase the performance, as the proposed debugging technique works better when the VCGRA is infrequently reconfigured. The results are depicted in Table 4.1.

Different variations of the application have been designed to evaluate the area of the proposed technique:

- A conventional implementation (without a VCGRA) of the target application, for fair comparison (static).
- A VCGRA wrapper with an AXI-Lite module without any debugging functionality.
- A VCGRA-SDA wrapper with an AXI-Lite module that has integrated debugging functionality (proposed approach).

Component	**Area**		**Power**
Design	LUTs	FF	W
Static	1230	1420	15.775
VCGRA	768	1542	3.604
VCGRA-SDA	1006	1542	3.324
VCGRA-ILA	5303	8093	3.844

Table 4.1: Comparison of the Sobel Edge detection filter design implemented on a Virtex-7 with Vivado 2016.1. The proposed (VCGRA-SDA) technique is smaller than the static (no-VCGRA) implementation (with no debugging infrastructure) and significantly smaller than the conventional debugging core.

- A VCGRA wrapper with an AXI-Lite module and an ILA connected that can trace the VCGRA's internal signals (conventional approach).

With Vivado 2016.1, the VCGRA implementation on a Virtex-7 is up to 39% smaller than the static implementation. The proposed architecture with the integrated debugging functionality (VCGRA-SDA) is $5.2\times$ smaller than the VCGRA design integrated with Vivado's ILA and 18% smaller than the static application (without any debugging functionality). In the VCGRA, there is an area penalty of 25%, to add the debugging functionality. However, this is considered small, since the area penalty to add debugging functionality with the ILA core is up to 860% for a small design (PE). A schematic representation of these results is found in Table 4.1 and in Figure 4.7. Here, even though the area penalty of the proposed technique (VCGRA-SDA) is 23%, it remains 22% smaller than the original application. Therefore, by transforming an application into a VCGRA, the debugging can be included in the application without any additional overhead. Thus, the on-silicon debugging circuitry can be integrated in an application, even post-development, and can be re-activated, to debug bugs that escape after deployment, which is the case in most designs [36].

Power

The power results are depicted in Figure 4.7 and in Table 4.1. The power consumption of using a VCGRA instead of the conventional implementation of the Sobel filter is decreased by $4.3\times$. There is no power increase for the VCGRA-SDA architecture. In fact, it needs 13% less power compared to integrating a logic analyzer (ILA core) in a VCGRA with Vivado.

The results indicated as *PConf* in Table 4.2 are compared with the results obtained from ABC, the standard tool to synthesize the academic FPGA used for the PConf. For the proposed technique there is a decrease of 25% of the FPGA resources with the proposed technique compared to an implementation with the

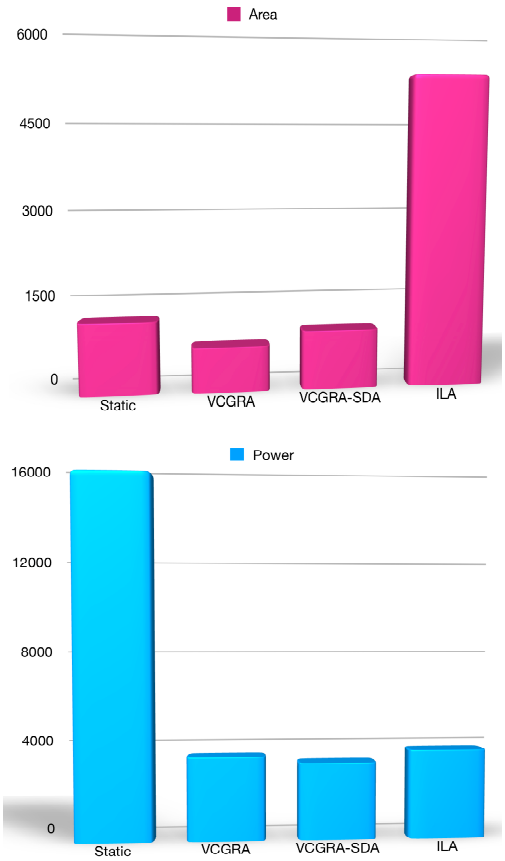


Figure 4.7: Schematic representation of the results after the implementation of the target application. The comparison is between a static implementation, a VCGRA, a VCGRA with debugging with a vendor tool (VCGRA-ILA) and the proposed one (VCGRA-SDA).

conventional tools. This is achieved due to the usage of the reconfiguration interface for the debugging infrastructure, that is possible with the PConf but not with ABC.

In general, the implementation of a PE uses fewer LUTs in commercial tools compared to the PConf tool. This is due to the fact that these tools probably contain better optimizations for numerical operations and have far more optimizations than explicitly for the specific target-architectures, compared to the generic theoretical FPGA architectures that are used in academic tools. However, designs with more multiplexers are better optimized using the PConf tool. Especially tuneable connections are heavily used for the implementation of the VCs and the debugging architecture, because multiplexers are well suited for optimizations of shared

Component	**Commercial**	
	LUT	FF
VCGRA		
PE	61	27
VC	10	34
VCGRA	768	1542
VCGRA-SDA		
PE	62	35
VC	10	50
VCGRA	1006	1542
VCGRA-ILA		
PE	3318	4468
VC	3267	4475
VCGRA	5303	8093

Component	**Academic**	
	ABC	PConf
VCGRA		
PE	85	76
VC	34	16
VCGRA	-	-
VCGRA-SDA		
PE	102	75
VC	66	16
VCGRA	-	-
VCGRA-ILA		
PE	n/a	n/a
VC	n/a	n/a
VCGRA	n/a	n/a

Table 4.2: Logic Utilization for VCGRA-SDA designs implemented with different tools. The area results are shown in terms of LUTs and flip-flops. For academic FPGAs the area is compared in LUTs with the ABC tool [97].

resources. The results are shown in Table 4.2.

The two tools are not directly comparable, as they are implemented on different architectures. Due to some limitations with the PConf tool currently it is impossible to implement large designs. Therefore, the results for the complete VCGRA using this tool cannot be shown.

Theoretical FPGA Architectures

For the following experiments, it has been assumed that the application requirements exceed the available FPGA's resources, hence the (parameterized) configurations approach is selected over a fully application-specific design. The components were synthesized, placed and routed, with the tailor-made CAD tool that enables the parameterized configurations. Place & Route was performed using a 6LUT_sanitized FPGA architecture. This VCGRA has been designed based on [90]. The results are presented in Table 4.3.

- The PE that realizes the VCGRA's mathematical operations has been designed in VHDL in two different versions: a fixed point PE and a floating point PE. The P&R results of the fixed point and floating point PE are tabulated in Table 4.3. The logic resources (in terms of LUTs) used by the fixed point PE are optimized by 5% and for the floating point PE design are optimized by 24%, compared to the non-parameterized implementation. The wire length has decreased by 2% for the fixed point and by 25% for

the floating point design. This optimization can be achieved by investing a reconfiguration time costs of $3.4ms$ for the fixed point and $88.5ms$ for the floating point. Moreover, 13% and 82% of the PE's design (for fixed and floating point PEs respectively) is responsible to form the reconfigurable part of the processing element.

- The VC is also described in VHDL. The channel has parameterized connection multiplexers whose select lines are the parameter inputs. Since we parameterize the connections, the major part of the VC doesn't need LUTs. Here, 82% of the logic is mapped on the reconfigurable physical switches instead of physical LUTs and multiplexers (as per the conventional implementation). We also observe a significant decrease of 76% in wire length (WL) between conventional and parameterized implementation due to the fact that the minimum channel width (mCW) is reduced by 42%. This optimization can be achieved at the cost of a reconfiguration time of 4.6 ms.

The SDA is also described in VHDL. Since we parameterize the connections between VCGRA and the debugging infrastructure, the SDA doesn't consume any additional LUTs and has minimal impact on the total routing. The results are presented in Table 4.3, alongside the VCGRA results. The minimum SDA architecture (the one attached to a VC) needs 16 parameterized connections and no LUTs. If it is implemented with the conventional tools, it needs 17 LUTs. The proposed architecture has 0% impact on physical LUTs for PEs, VCs and the grid. It introduces 18% routing overhead in VC, 66% in fixed point PEs, and 5.6% in floating point PEs. The total routing impact in a VCGRA grid is 10.2%. The total wirelength increases by 4% for a VC, by 1% for a fixed point PE, by 0.6% for a floating point PE and 0.08% for the total grid. The results are summarized in Table 4.3, where we can observe the LUT reduction after the VCGRA parameterization. The area requirements are presented in terms of LUTs. We can observe that there is no increase in the number of physical FPGA resources (LUTs) after the installation of the SDA. There is an increase in the virtual FPGA resources (TLUTs and TCONs), with no impact on the design's performance. After the SDA installation on the VCGRA, the wirelength impact is also minimal. For the experiments, it is assumed that each level of PEs is successfully debugged and self-corrected, after the completion of the algorithm. However, this is not always the case. In some scenarios, the bug is located in multiple PEs. In that case, a full VCGRA reconfiguration is needed.

	LUTs (TLUTs)	TCONs	Logic Depth level	WL	mCW
SDA	0(0)	16	0	34	2
VC Conventional	176(0)	0	2	3186	7
VC parameterized	32(0)	72	1	782	4
VC SDA	32(0)	88	1	816	4
PE Conventional	408(0)	0	47	3832	8
PE parameterized	387(32)	22	47	3769	8
PE SDA	387(32)	70	47	3817	8
PE_FP Conventional	2191(0)	0	47	23388	10
PE_FP parameterized	1668(584)	798	47	17676	10
PE_FP SDA	1668(584)	846	47	16778	10
Grid Conventional	17066(0)	0	155	176200	14
Grid parameterized	16099(976)	561	153	169560	12
Grid SDA	16099(976)	625	153	169696	12

Table 4.3: Resource utilization and P&R results of VCGRA with and without SDA

4.4.2 Online Execution Times

Commercial FPGA Architectures

The design has been implemented on a Zedboard with Xilinx Vivado 2016.1, as this technique is based on dynamically reconfigurable SRAM-based FPGAs. For this work Xilinx FPGAs have been targeted. However, other commercial tools that support SRAM-based FPGAs can be used, such as Quartus II. With some amount of engineering effort it will be possible to support this technique for any SRAM-based FPGAs, such as Intel's, but not tools such as Libero, that support flash-based FPGAs. In that case, the entire concept of debugging, VCGRAs and SDA has to be revisited.

The four different design variations (static, VCGRA, VCGRA- SDA and VCGRA -ILA) of the application are used for comparison of its execution times. In more detail, an AXI-Module is added to read and write GPIOs of the processing system (PS) for the synchronization between the PS the VCGRA and the SDA. Additionally, a VCGRA wrapper and a VCGRA-SDA was created with four AXI-Lite interfaces to send and receive data and to parameterize the implemented VCGRA. Using this interface, the configuration data can be transferred from the PS to the programmable logic where it is saved locally within configuration registers.

On average, the current design needs around 160μ to calculate a single result pixel [38]. In order for new PEs to be instantiated for debug, we need 6.4μs to reconfigure them. PE reconfiguration is 8.25% of the total reconfiguration time. For the whole VCGRA, more than half of the whole processing-time is used to transmit data-values and coefficients. Frequent reconfigurations are not beneficial for the execution timing. The bottleneck in the design is the AXI-Lite interface between the PS and the VCGRA. Currently, the obtainable performance is limited by the AXI-lite interface, because AXI-lite does not support bursts or data-streaming. Thus, for each data package of four bytes, the overhead of a full AXI-communication sequence is needed.

Theoretical FPGA Architectures

In order for a new PE to be instantiated for debug, the tool needs $280ms$ to place the SDA architecture and $27ms$ to route it. Hence, SDA can be in-place to start tracing the erroneous behaviour of a new virtual channel in $307ms$. Then, after a bug is confirmed, 11.28 ms are needed to microreconfigure one frame. Therefore, $318.28ms$ are needed to install a new SDA element in a new VCGRA location and self-heal the erroneous element, in case the self-healing functionality is enabled. The timing information was measured with the PConf tool.

In theoretical FPGAs the debugging infrastructure is on a separate layer. In that way it has small effects on the interconnects and any other element that can influence the timing of the design itself. However, it is possible that the SDA will

take away resources that may no longer be used by the actual implementation and therefore can result in congestion for the implementation and thus longer routes and lower clock frequencies.

In commercial FPGAs, the SDA is connected with the VCGRA. They are co-implemented and co-exist. That means that a specific VCGRA is paired with an SDA and they have a slower (or faster) timing compared to another VCGRA-SDA pair. This will not affect the entire system much, since the entire VCGRA-SDA will be faster or slower (since the timing will be different). Therefore, the on-silicon debugging infrastructure will not impact the timing of the original implementation in the sense that it is co-designed alongside the original implementation during the original (VCGRA) compilation.

PConf Approximation

With a PConf adaptation of a VCGRA, only an evaluation of a parameterized configuration needs to be performed to create the new configuration. Then, the design needs to perform only microreconfigurations in the FPGA and not full reconfigurations. One microreconfiguration of 1 TLUT takes 64.1μs. Therefore, in order to reconfigure the entire VCGRA approximately 82ms are needed. This is calculated based on an approximation of the total number of TLUTs in a VCGRA-SDA design. This metric is closer to the theoretical FPGA architecture reconfiguration time and is 3 orders of magnitude faster than the reconfiguration time without parameterized configurations and the conventional AXI-HWICAP reconfiguration controller. Since some limitations with the PConf tool for large designs exist, the complete VCGRA results with the PConf adaptations are based on approximations.

Different reconfiguration controllers

The speed of configuration is directly related to the size of the BIT file and the bandwidth of the configuration port. Here, the reconfiguration times between different reconfiguration controllers and the target application are compared. Since the bitstreams that are generated from the application have similar size, the impact on the reconfiguration time is negligible. Therefore, if the SDA is integrated in the VCGRA, there is no impact in the configuration time for a Virtex-7 FPGA. However, there is a significant acceleration if the MiCap-Pro controller [89] is used instead. This is shown in table 4.4.

4.4.3 Internal Signal Analysis

With Vivado's ILA, only 10% of the internal signals could be selected and integrated in one ILA core. Above this threshold, the design was unrouteable. More-

Design	**HWICAP**	**MiCap**	**MiCapPro**	**JTAG**	**SelectMAP**
Static	8.49	7.34	0.59	2.4	8.49
VCGRA	8.493	7.4	0.596	2.457	8.493
VCGRA-SDA	8.493	7.33	0.593	2.445	8.493

Table 4.4: Configuration time (in s) with different reconfiguration controllers.

over, with ILA, the designer needs to manually select all the signals of the synthesized design. Then the tool performs an additional compilation to install the extra infrastructure. This infrastructure dramatically increases the overhead of the FPGA in terms of LUTs (the design becomes $7.7\times$ larger) and the additional compilation overhead that is needed to install the debugging functionality adds an overhead of additional multiple minutes. However, with the VCGRA-SDA method, the integration of the debugging functionality is done without any significant compilation overhead. In fact, the no-SDA and the SDA designs took the same amount of time to be compiled. However, for larger designs, we expect a small compilation overhead.

The SDA is automatically integrated in the VCGRA before synthesis. At this point, the designer doesn't need to pre-select signals. However, he/she can choose to run the GateRank algorithm. The internal signals can be selected automatically. Then the signals can be transferred for analysis. Therefore, the verification engineer will be able to trace more signals with the proposed technique (up to 100%), in comparison with ILA that can trace up to 10% of the target application's signals. Additional signals can be traced only after adapting the ILA core and repeating the whole process.

4.4.4 GateRank Optimization

Here, we apply the GateRank (GR) algorithm, that was presented in Chapter 3. The GateRank graph is shown in Figure 4.8.

Since only few of the nodes have higher GR than the rest of the nodes ($3\times$ higher than the average), we trace them back in the sources of their respective signals. Then, we adapt the debugging infrastructure to debug only the signals with the highest GR. The results are presented in Figure 4.9.

A significant reduction in the resources can be observed. This has occurred due to the fact that the SDA was applied only in specific signals (that are assumed as the most critical ones by GR). Then, the design was fully parameterized and synthesized and mapped with various mappers. It can be seen in the results that the PE with the SDA installed on its most critical signals (TCON Debug_GR bar) is 10% smaller than the original design (ABC DUT) that doesn't have any param-

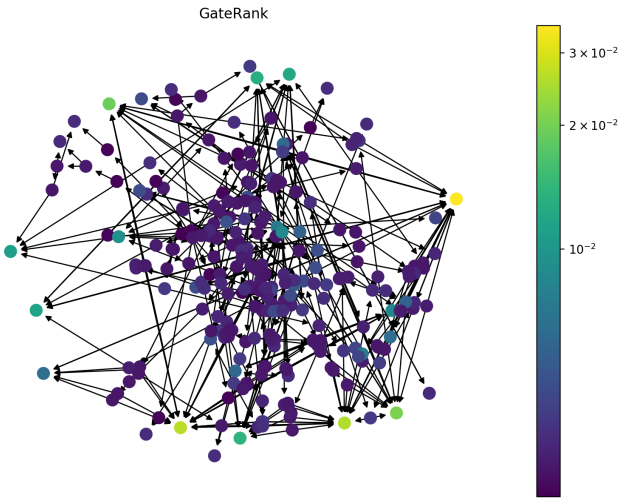


Figure 4.8: Graphic representation of the GateRank of a PE, for the target theoretical FPGA.

eterization nor any debugging infrastructure.

Debugging with the SDA is defined as on-silicon debug. It can also be introduced during the implementation stage to assist in debugging the data path, as according to [36], bugs always escape after debugging, post-development, even for safety-critical applications. The structure of the co-designed VCGRA-SDA is developed in that way that the concern can be bugs in both the control path and the data path. It depends on the signals that will be determined to be traced (control or data). In this work, the signals that were selected to be traced were part of the data path. However, for a different VCGRA study, control path signals would be of (debugging) interest. In that case, the VCGRA-SDA design will adapt accordingly, as it is relatively easy with the accompanying tools to add tracing infrastructure on the VCGRA. The acquisition can start either manually, or via triggering. This work involved on-silicon debugging, that is realized while the VCGRA is being developed. In that way it can collect raw debug data for offline analysis, to finalize the design. In that case, no specific events are in need to start the acquisition, as raw data can be used.

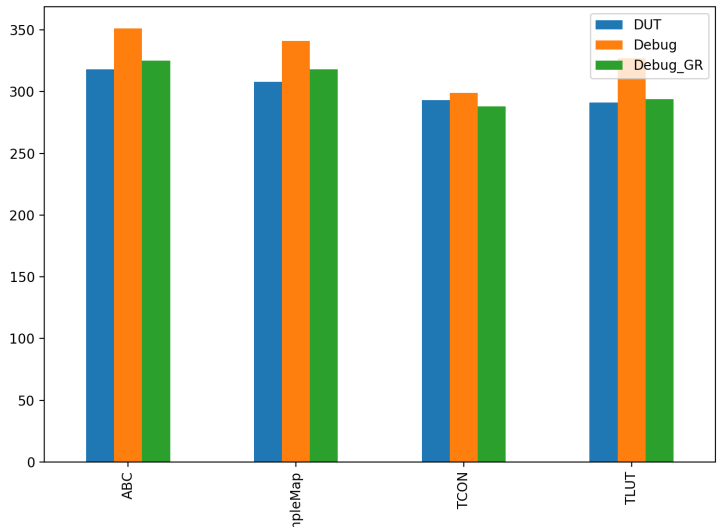


Figure 4.9: Results from various Mappers after installing the SDA in all signals (Debug) and only in the signals with the highest GateRank (Debug_GR) and two different forms of parameterization.

4.5 Conclusion

In this chapter, a low-power / low-overhead superimposed debugging architecture is proposed, for heterogeneous computing platforms. This technique is used to integrate debugging infrastructure to efficiently debug FPGA overlays, such as VCGRAs. In order to create this novel architecture, a supporting flow and a new two-level virtual architecture have been constructed. The custom made tools introduce resource sharing on the two-level virtual architecture, reducing the actual FPGA's resources. Additionally, by parameterizing the PEs and VCs, their resources are further reduced. Hence, a low-overhead debugging mechanism can be integrated in virtual FPGA architectures, without any power or reconfiguration overhead and with a minimal area penalty of 30%.

Part III

Fault Tolerance

5

Mitigation of Radiation Effects with Microreconfiguration

Soft errors from radiation are the primary concern in digital electronic reliability. This phenomenon is more important than all other causes of computing reliability put together. As we enter the era of ubiquitous computing, with interlaced intelligence controlling our machines and information, system crashes without hardware defects are the major threat of complex designs. This chapter analyses the basic reliability techniques for SRAM-based COTS FPGAs, to provide integrated fault tolerance on a COTS FPGA, with multi-level configuration scrubbing. Two reconfiguration controllers are also proposed that support the novel technique.

5.1 Radiation Effects

The three main sources contributing to a radiation environment are the galactic and extra-galactic cosmic rays (GCR), solar energetic particles (SEPs), and trapped particles. They are the main source for causing a radiation environment. The radiation environment is divided in the space environment, the aerospace, the ground (sea) level and the underground level. It is also divided in natural environment (space radiation) and human-made radiation environment, like the nuclear plants and particle accelerators. This is extensively analyzed in Appendix A. In this sec-

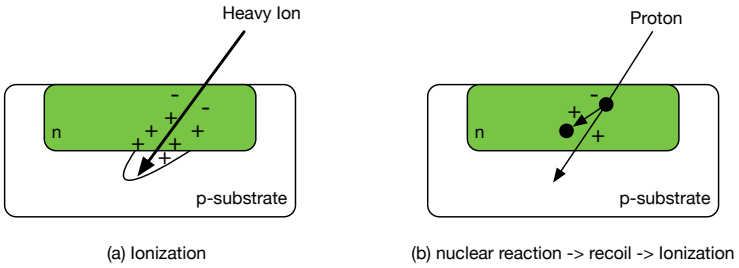


Figure 5.1: Charged particles strike the silicon

tion, the energetic particle radiation is analyzed, alongside its source, and its monitoring and mitigation techniques.

5.1.1 Radiation Effects in FPGAs

The COTS FPGAs do not undergo a radiation hardening procedure and they are not tested for harsh radiation environments, leaving the system susceptible to radiation-induced permanent and non-permanent faults. Hence, the system must integrate built-in mitigation and validation techniques, on system or architecture level, for run-time recovery from failures. Fault injection should also be offered in-system, to rapidly evaluate at any moment that the mitigation process remains robust. The objective of my research (and of Part III of the thesis) is to introduce mitigation, validation and fault injection techniques in FPGA overlays, to enhance their fault tolerance.

The radiation environment is composed of various particles. The particles can be classified as two major types: charged particles such as electrons, protons and heavy ions, and electromagnetic radiation (photons), which can be x-ray, gamma ray, or ultraviolet light. The main sources of charged particles that contribute to radiation effects are protons and electrons (trapped in the Van Allen belts), heavy ions trapped in the magnetosphere, GCR and SEPs. This is further analyzed in Appendix A.

When a single heavy ion strikes the silicon, it loses its energy via the production of free electron-hole pairs, resulting in a dense ionized track in the local region, whereas protons and neutrons can cause nuclear reaction when passing through the silicon. The recoil also produces ionization. The ionization generates a charge deposition that can be modeled by transient current pulse that can be interpreted as a signal in the circuit causing an upset. This is illustrated in Figure 5.1.

The term soft errors is used for nondestructive functional errors, that are induced by energetic ion strikes. They are a subset of single event effects (SEE),

Figure 5.2: Radiation Effects in SRAM-based FPGAs.

and include Single Event Upsets (SEU), multiple-bit upsets (MBU), Single Event Functional Interrupts (SEFI), Single Event Transients (SET) (that if latched, become SEU), and Single Event Latchups (SEL). In SEL, the formation of parasitic bipolar action in CMOS wells induce a low impedance path between power and ground, producing a high current condition (SEL can also cause latent and hard errors). An overview of common radiation effects that must be mitigated in SRAM-based FPGAs is given in Figure 5.2. The main effects in the remainder of the section.

Total Ionizing Dose

The ionization of electronic components is caused by electrons and protons and bremmstrahlung. It causes leakage currents and various other effects [29]. Ionization is caused due to processes related to photon interaction, such as the photoelectric effect, the Compton effect, and pair production, all leading to the production of free electrons and hole-electron pairs. The accumulation of these effects leads

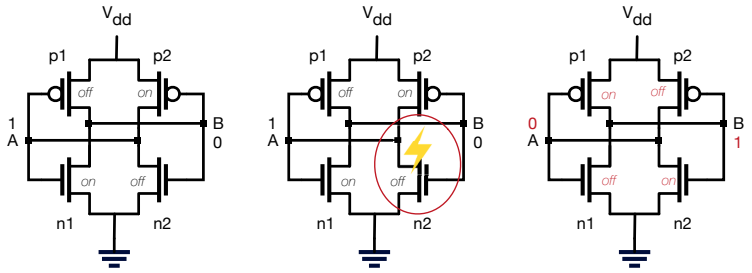


Figure 5.3: SEU effect in a SRAM Memory cell

to degradation. It is called called Total Ionizing Dose (TID) and is measured in krad.

Single Event Latchups

A SEL is a SEE that can be potentially destructive. It can trigger parasitic PNPN thyristor structures in a device [29]. SEL effects are of no concern for radiation hardened devices, such as Virtex-4QV and Virtex-5QV, as both devices have a guaranteed latchup immunity. However this is not the case for COTS FPGAs.

Single Event Upsets

SEUs are the most common effects for SRAM-based FPGAs, as they may affect the configuration memory as well as memory cells that are used as part of the user logic (flip-flops, embedded RAM). They are soft errors that change the state of a bi-stable element, as it is illustrated in Figure 5.3. SEUs are caused by heavy ions and protons and result from ionization by a SEP or the nuclear reaction products of an energetic proton. The ionization induces a current pulse in a $p - n$ junction whose charge may exceed the critical charge that is required to change the logic state of the element. As a result, the value of a memory bit can be flipped [131].

Single Event Transients

A SET is a momentary voltage or current disturbance that may propagate through subsequent circuitry. It can eventually manifest as a SEU once it reaches a flip-flop or other memory elements [28].

Single Event Functional Interrupts

SEFI interfere with the normal operation of the FPGA. It is typically used to classify failures that affect the circuits needed to operate the FPGA [131]. A SEFI is often associated with an upset in a control bit or register. Various types of SEFIs exist, namely Power-on-reset (POR), SelectMAP (SMAP), Frame address register (FAR), Global signal , Readback and Scrub SEFIs.

Multi-bit Upsets

Many of the SEUs are in fact Multi-bit Upsets (MBUs) [120]. When a single high-energy ion passes through the silicon it can energize two or more adjacent memory cells. MBUs can be induced by direct ionization. The energy of the particle is more likely to provoke double bit upsets while multiple bit upsets are caused by an increase of the particle incident angle [111].

5.1.2 SRAM-based FPGAs in Radiation Environments

FPGAs have become one of the core components for high-reliability computing systems, as they are used for in-situ high-speed signal processing, for data categorization and data compression in radiation environments, such as space, aerospace and in particle accelerators. The need to modify the design as the high-reliability system is upgraded, the low-volume cost of FPGAs, and the need for a flexible and adaptable system all point toward the use of FPGAs as a hardware implementation solution.

SRAM-based FPGAs specifically are a convenient solution, as they can offer hardware reconfiguration for high-reliability applications in later design, after upgrades or even during space missions. They are best suited for high-performance on-board tasks due to their flexibility and their embedded DSP blocks, compared to single-board computers and DSP processors. There is a growing demand for Commercial Off-The-Shelf (COTS) SRAM-based FPGAs for experiments in high-reliability environments, such as the Compact Muon Solenoid (CMS) Trigger of the Large Hadron Collider (LHC) particle accelerator at European Organization for Nuclear Research (CERN), that performs real-time pre-processing of collision data. Moreover, SRAM-based FPGAs are used in Low-Earth Orbit satellites and in space missions with less dependability constraints.

During the design process various techniques need to be applied to mitigate SEEs in SRAM-based FPGAs. A classification of these techniques is presented below, which uses the terminology introduced in the *NASA Fault Management Handbook* [108]. Although targeting flight systems in general, this terminology can be used to describe soft error mitigation techniques for FPGAs as well, in various radiation environments.

An abnormal state of a system is described with three terms: fault, error, and failure. A *fault* is defined as the cause of an error. An *error* is defined as the cause of a failure. Finally, the *failure* is defined as the termination of the ability of an element, to perform a function as required.

According to the *NASA Fault Management Handbook*, failures can be either prevented or tolerated. In the first case, actions are taken to avoid failures either at design-time or runtime. The design-time fault avoidance includes fault management capabilities to minimize the risk of a fault and resulting failure, whereas operational failure avoidance predicts that a failure will occur in the future and takes action to prevent it. According to these guidelines, with failure tolerance, failures are either accepted or mitigated. Failure masking techniques allow a lower level failure to occur, but mask its effects so that it does not affect the higher level system function. Failure recovery methods allow a failure to compromise the system function temporarily, but respond before the failure compromises a mission goal. Finally, goal change strategies allow a failure to compromise the system function, and respond by changing the systems goals to new, usually degraded goals that can be achieved.

Throughout this PhD research, any erroneous FPGA output is defined as a failure. Although the failure is always caused by a fault, a fault does not necessarily lead to a failure, as it can be masked. In an FPGA design, such a fault could be a soft error caused by a flipped bit in a flip-flop, a reprogrammed logical operation due to a falsified LUT, or a permanent fault.

5.2 Mitigation of Radiation Effects in SRAM- Based FPGAs

If an ion hits an SRAM-based FPGA, it can affect memory resources either in the configuration memory, or in the user logic layer. When this happens, an upset can be seen as a fault that may or may not lead to a failure. If the incoming particle strikes the configuration cells for logic modules it can lead to misconnected signals or to functional changes. If it strikes the routing matrix, it can cause a misrouted signal, or a missing signal. This is depicted in Figure 5.4.

Since the SRAM-based FPGAs can be reconfigured, the affected memory cell can be updated with the correct values. Hence, the configuration memory are the main concern to create a mitigation strategy. A fault in the configuration memory may lead to a failure in case the affected configuration bit controls a resource that is utilized by the design. Xilinx classifies the configuration memory bits as non-essential, essential and critical bits. Essential bits are defined here as those bits associated with the circuitry of the design. Thus, a fault affecting an essential bit *may* lead to a failure.

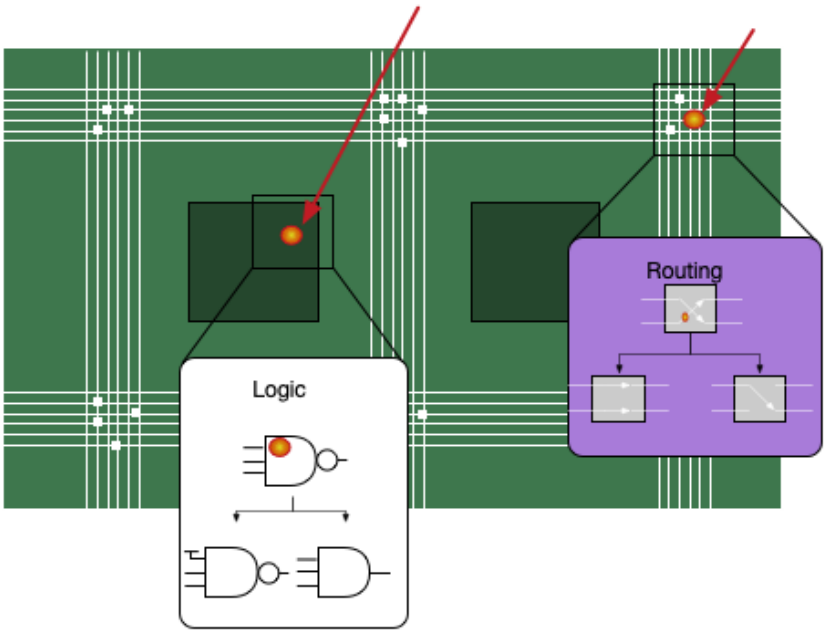


Figure 5.4: Neutrons and alpha particles cause upsets in SRAM-based FPGAs.

Critical bits are the subset of the essential bits that guarantee a failure in the case of a fault. They are defined here as those bits that cause a functional failure if they change state. Both essential and critical bits are a subset of the device configuration bits [94]. This is depicted in Figure 5.5. All other bits in the FPGA's configuration memory are classified as non-essential bits and they cannot cause a failure. A failure can propagate through the system until it becomes measurable at the output. It is often only of transient nature. A fault in a BRAM cell can lead to a failure with the next read access.

Mitigation of radiation-induced hardware faults in COTS FPGAs considers both permanent and non-permanent faults, such as SEUs, SETs, etc. These faults are caused by high-energy subatomic particles that are originating from the cosmic radiation. They can cause extensive damages in the functionality of FPGA components such as logic blocks, I/O blocks and DSP blocks. Hence, it is essential to apply SEU mitigation for fault tolerance. The basic approaches include mitigation of the user logic, of the configuration memory and of the BRAMs. The specific mitigation techniques are analyzed in the remainder of this section.

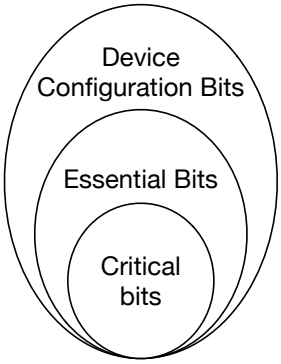


Figure 5.5: Relationship of Device Configuration Bits, Essential Bits and Critical Bits

5.2.1 User Logic Mitigation

The failure masking is implemented on the user logic layer using a method of redundancy. Most commonly, spatial redundancy is used, but information and temporal redundancy can be found as well for specific components.

Spatial Redundancy

The most common mitigation technique that uses spatial redundancy is TMR. Here, all components of a circuit are triplicated, and a majority voter is then placed at the end that chooses the correct output. This is shown in Figure 5.6. To decrease the probability of failure, the voter can be triplicated as well. In principle, when any of the design domains fail, the other two domains continue to operate correctly, and the voter passes the correct behavior to the outside world. SEUs that affect feedback loops, such as counter or state machines, can be problematic because the failure is trapped in the loop. To overcome this problem, voters can be placed inside feedback loops. This technique, sometimes also referred to as XTMR [15], synchronizes the flip-flops automatically after repair.

Radiation tolerance can also be improved with software-based mitigation techniques, that use temporal and spatial locality to protect the data flow. Error detection is done via duplicated or triplicated instructions, or via lower level duplications of registers and selected instructions [18, 20, 21, 113]. This introduces large overheads in terms of program size, memory usage, and execution time, in the same way TMR introduces area overheads in hardware. Recently, compiler-based techniques have been used to introduce information redundancy with both fine-grained and coarse-grained approaches [9, 27, 35, 118, 119]. For scenarios where latency is not critical, these program replication techniques can provide high re-

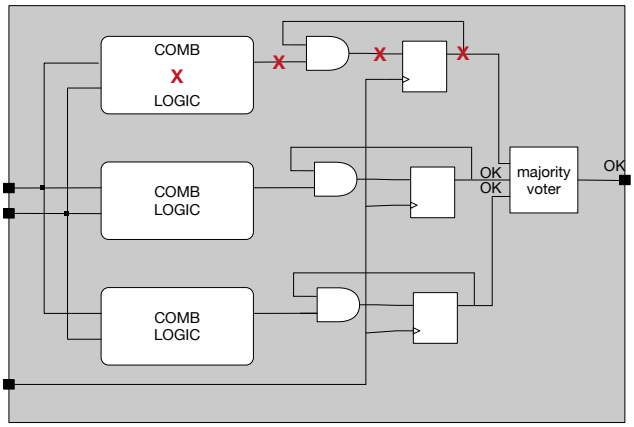


Figure 5.6: Traditional TMR Implementation

liability for COTS systems. However, most of these techniques often slow down program execution, target specific architectures, languages, or features, and lack automation tools, making them of limited use for future research or commercial projects. Additionally, the majority of these techniques are limited to server-like environments, targeting high-performance, super-scalar processors, and are not tailored for FPGAs.

In order to create a fault tolerant design, the designers currently employ existing CAD tools, such as Synplify Premier, XTMR, Frame ECC, or custom error correction codes [122] and then run the conventional design flow, which introduces significant resource overhead ($\geq 3\times$) and reduces the overall performance. Moreover, a reconfiguration controller is used that increases the area overhead by more than 1000 LUTs and is prone to radiation effects as well. TMRG [85] is able to integrate predefined mitigation techniques semi-automatically, in a target design by merging Verilog code. However, it doesn't support fault prediction, nor multiple (FPGA) architectures.

TMR is usually applied to a netlist using CAD tools. Several commercial and academic software tools are available, including the TMRTool by Xilinx [15], Precision Hi-Rel by Mentor [102], and Synplify Premier by Synopsys [141]. Despite its demonstrated reliability, TMR has some drawbacks. The basic drawbacks are the area overhead and the decrease in the design's performance. Several techniques have been proposed in the literature to overcome the drawbacks, such as partial TMR and modular TMR. These techniques have increased flexibility, however they are less robust than traditional TMR.

With spatial redundancy techniques, other faults can be mitigated that manifest

as bit flips, such as Silent Data Corruption (SDC). A SDC materializes as bit flip in storage (both volatile memory and non-volatile disk) or even within processing cores. Therefore it can be mitigated with a spatial redundancy method.

During runtime, failure masking techniques can be used to tolerate failures. Failure masking allows the fault to happen, but masks its effect and additionally it reduces its probability to propagate. This is usually achieved by redundancy. Most commonly, spatial redundancy is applied, such as full or partial TMR, duplication with compare (DWC) [72], or reduced-precision redundancy (RPR) [39].

RPR differs from TMR, as instead of using three redundant copies of a module, one module processes data with full precision while the two other modules process the data with reduced precision. This makes RPR is suitable for data processing algorithms, such as fixed-point numerical problems. A decision block determines if a failure has occurred as follows. In DWC a circuit is duplicated and the output of the redundant circuits is compared by a comparator. This is a less popular mechanism, as it is only able to detect failures instead of masking them. However, it can be useful for systems that need a low-overhead failure detection mechanism that triggers scrubbing on demand.

Information Redundancy

Although information redundancy techniques are mainly applied to memory and communication channels, ICs can benefit from them as well. Information redundancy techniques function by adding redundant bits to data to be able to detect or even correct falsified information. An example for the first case is the CRC code, whereas error correction can be achieved by Hamming codes. In fact, parity bits can be added to achieve a Hamming code, which enables the detection or correction of bit upsets. State machines with a Hamming with a distance of 2 (H2) have have fewer overall errors and state machines Hamming with a distance of 3 (H3) codes can fully handle single errors and are least affected by double bit errors.

For SRAM-based FPGAs, however, these results should be regarded with care because the influence of configuration memory upsets is not taken into account. Using fault injection, authors in [107] concluded that the additional required logic can potentially add more unreliability than the reliability it adds to the original circuit.

Alternatively, information redundancy techniques can be used to detect and mask failures in certain types of circuits, such as algorithm-based fault tolerance (ABFT) that is used to implement fault-tolerant matrix operations [68], or error detection and correction (EDAC) [92]. EDAC techniques are one of the most popular solutions for mitigating state machines. The states can be encoded using different coding schemes, (binary, one-hot, Grey). In addition, parity bits can be added to achieve a Hamming code, to detect or correct SEUs.

5.2.2 Configuration Memory Mitigation

Multiple errors can break TMR in FPGA-based systems. These systems require dedicated mechanisms that refresh the memory by scrubbing (refreshing) the contents of the configuration memory. These approaches are often coupled with hardware redundancy techniques. Thus, configuration scrubbing is used as an enhancement to TMR to avoid fault accumulation [10, 54, 109, 117, 123–125].

Scrubbing is a mitigation technique that corrects SEUs in an FPGA's configuration memory by writing incorrect bits with correct data while the system is fully operational, via (partial) reconfiguration. Scrubbing protects against accumulation of SEUs. It utilizes a dedicated area of the FPGA and performs the necessary operations to repair SEUs by reconfiguring a portion of the design while continuously scrubbing the entire FPGA [51, 52]. In [45], a faulty configuration frame[1] is first identified and then scrubbed. The drawback of this work is long diagnosis time, bounded by reading the configuration frame and checking its correctness. This involves coarse-grained modules and is further analyzed in Chapter 6.

The Xilinx 7-Series FPGA provides a number of fault mitigation features, such as a built-in internal configuration scan and Frame Error Correction Codes. These can catch isolated bit errors, but are prone to missing multiple errors occurring at the same time. However, results from radiation testing on the 28-nm Kintex 7 FPGA suggest that intra-frame multi-bit upsets (MBU) account for 9.9% of the events observed [149]. That means that only 90,1% of the bits can be repaired with the existing technology, as scrubbing will not protect TMR against a MBU, where a single strike can hit multiple nodes at one time. Additionally, scrubbing needs re-synchronization of the system, which introduces time overhead. Significant time overhead is also introduced by reconfiguration during scrubbing, which may jeopardize the entire design [1].

Blind Scrubbing

During blind scrubbing the configuration memory is periodically updated with a known to be correct copy of the original bitstream. This copy that is sometimes referred to as the golden copy is stored in an external, radiation-hardened memory. An external or internal configuration controller controls the download of the bitstream via one of the configuration interfaces of the FPGA. Blind scrubbing is an operational failure avoidance methodology because faults are handled in a preventive manner without any knowledge about the current health state of the system. It is the most basic methodology. However, since this scrubbing is by definition blind, that means that potential unnecessary bitstream downloads and writes occur to the configuration memory. Scrubbing can also be effective in cases of Detected

[1] By frame, we mean the smallest addressable element of an FPGA's configuration memory

Uncoverable Error, that is an error that is detected but not corrected. The solution can be scrubbing with FPGA reconfiguration or parameterized reconfiguration.

Readback Scrubbing

The readback feature of the FPGAs can also be utilized for scrubbing. In that way, the bitstream can be read for SEUs and unnecessary write accesses to the configuration memory can be avoided. Before writing a correct bitstream to the device, the current bitstream is read and checked for upsets. Only if upsets are detected is the correct bitstream eventually written. This is categorized as a failure recovery technique [153].

Device and Frame Scrubbing

The device-based scrubbing, requires a simple implementation where the configuration memory is scrubbed with a full bitstream (or frame-by-frame). Then, the bitstream can be directly downloaded from a memory to the configuration interface. The basic drawback of this scrubbing method is the susceptibility of the configuration interface to soft errors. If a soft error occurs during bitstream download, the whole design is likely to become corrupted.

Frame-based scrubbing requires a more complex configuration controller implementation because each frame must be prepared before download. However, the benefit of this approach is the possibility to isolate the effects of a soft error to a single frame. Aside from the increased complexity of implementation, the scrubbing speed is decreased as well [16].

Periodic and On-Demand Scrubbing

In many designs, the scrubbing process is independent of other mitigation techniques. The configuration memory can be periodically scrubbed and scanned for upsets with a fixed scrubbing rate.

The scrubbing process can also be triggered by a failure detection mechanism. Such a methodology can be advantageous in systems where continuous scrubbing is unwanted. Eventually, the availability of a system depends on the time a faulty component remains unrepaired [110]. This time can be minimized either by increasing the scrubbing frequency or by implementing a mechanism that can trigger a repair process immediately after failure detection.

External and Internal Scrubbing

A configuration interface, such as the SelectMAP can be used for external scrubbing due to its high throughput rates [112]. ICAP (Internal Configuration Access Port), the internal counterpart to SelectMAP, can be used if the scrubbing logic is

implemented on the user logic layer. External scrubbing via the SelectMAP interface is commonly seen as the more robust approach and is also recommended by Xilinx [16]. The authors at [8] come to similar conclusions, after comparing an external blind scrubber to an internal ECC readback scrubber by Xilinx.

Internal scrubbing does not necessitate an external configuration controller and a memory for the golden bitstream copies and is considered a low-budget solution. However, in most applications for high-reliability environments, a radiation-hardened supervisor and reliable memory for the initial bitstream configuration is available anyway.

5.2.3 Block Random Access Memory

The embedded RAM blocks in Virtex devices need special care regarding the mitigation because similar to the configuration memory, upsets in BRAM can accumulate, leading to an ever-decreasing reliability of the memory. Although the BRAM content can be read out via the configuration interface, external scrubbing is not possible during operation because the RAM cannot be accessed by configuration and user logic at the same time [16].

5.2.4 Mitigation Design Techniques for Design-Time Fault Tolerance

During design time, several techniques can help to avoid faults in advance, including tools for the susceptibility analysis (e.g., for the quantification of sensitive configuration bits) and tools that place and route a circuit design in a reliability-oriented way. Analytic approaches are available from Politecnico di Torino, that analyze the sensitivity of a design and then are reducing it, by rerouting the design [133, 136, 137, 139]. In this work the critical configuration bits of a circuit design are identified, faster than fault injection. Then, by rerouting the circuit, the tools can avoid the problem of SEUs. However, the rerouting decreases the performance of the circuit. With these tools, mitigation of MBUs is also applied with novel placement algorithms.

5.2.5 Overview of Mitigation Techniques

As has been analyzed in this section, mitigating SRAM-based FPGAs can be done by introducing spatial redundancy in the user logic and the BRAMs and/or by adding scrubbing functionality to repair the configuration memory. Various techniques can be used, based on the frequency of the reconfiguration and the available FPGA resources. Figure 5.7 summarizes the most common mitigation techniques for SRAM-based FPGAs that are used in high-reliability applications in radiation environments.

The decision graph presented in Figure 5.7 guides the designer in choosing the best possible mitigation scheme. The method to be used can be chosen based on the speed of reconfiguration, the number of available FPGAs, the size of the design and the availability of external (radiation hardened) equipment.

5.3 Contributions

In this chapter a multi-level scrubbing technique is proposed and two reconfiguration controllers that support it. The aim is to improve the reliability of COTS FPGAs in radiation environments, with rapid configuration memory scrubbing. Hence the following contributions are made:

- A scrubbing technique based on microreconfiguration that speeds up the scrubbing process.
- An on-demand scrubbing technique that targets reconfiguration frames under SEU.
- Two custom reconfiguration controllers (a lightweight and a fault-tolerant one) that implement fault injection, fault detection and correction functions on a target design. They are low-overhead and they support the proposed scrubbing techniques.

5.4 Configuration Memory Microscrubbing

After the installation of a spatial redundancy scheme, scrubbing occurs in the configuration memory bits to avoid SEUs, MBUs and fault accumulation. The first approach (Discrete Microscrubbing - DMS) has been designed to correct the FPGA from SEUs. Hence, DMS, is used to target a SEU on a specific frame. During the FPGA's operation, if one of the mitigated outputs has a different result than the other two, this means that a SEU has been detected at a specific location. The tool then triggers one of the proposed scrubbing mechanisms by providing the exact configuration frames that need to be repaired, with the use of the with the use of the XhwIcap_setClb_bits function [88]. This is called microreconfiguration and it is able to restore a frame after scrubbing.

5.4.1 Prerequisites: Microreconfiguration

Micro-reconfiguration is a technique to change the configuration of minor or few resources of the FPGA. Therefore, micro-reconfiguration is the main application of DCS, as it is analyzed in Chapter 1. The Specialized Configuration Generator (SCG) is realized on an embedded processor (ARM Cortex-A9 dual core processor

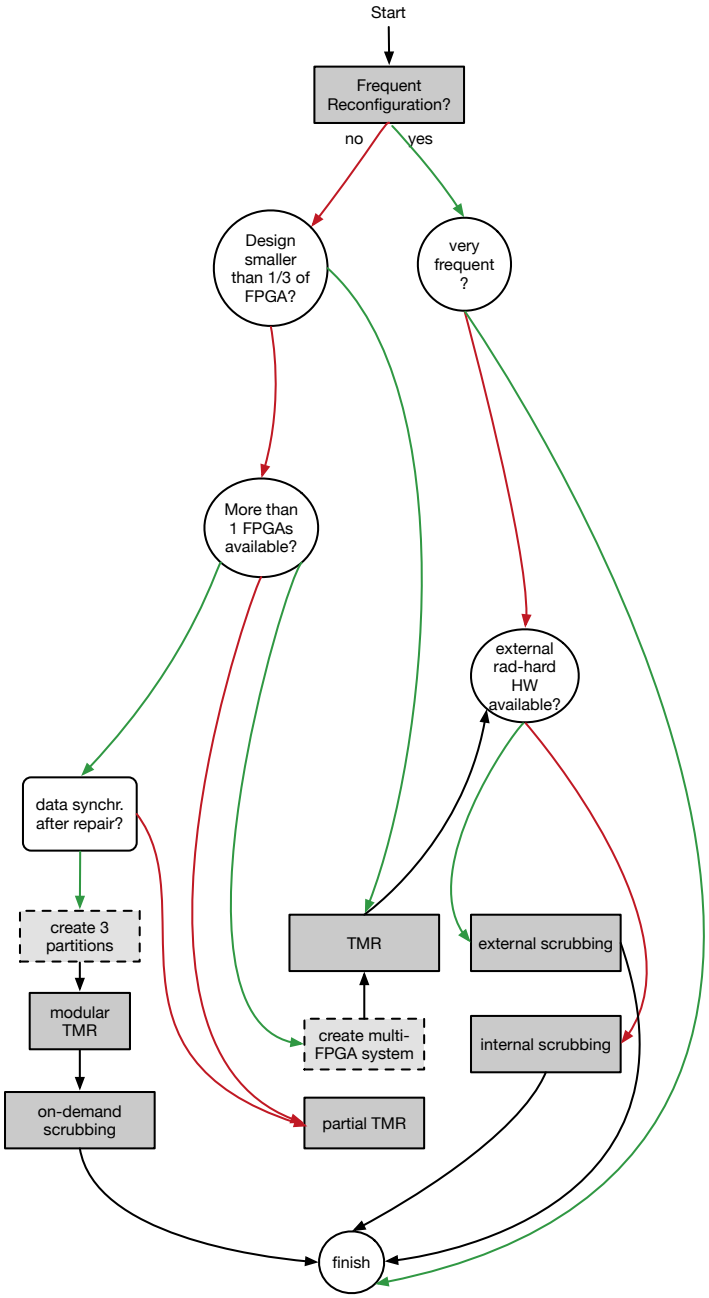


Figure 5.7: Mitigation Design Decision Graph. If the FPGA is constantly reconfigured, there is no need for a separate mitigation scheme.

or a MicroBlaze soft core processor). The PPC Boolean functions are stored in the memory such as DRAM memory of the Zynq-SoC. The ICAP is used as a configuration interface. The HWICAP reconfiguration controller is responsible for orchestrating the replacement of the stale frames with specialized frames present in the configuration memory of the FPGA. The HWICAP supports a software driver function called *XhwIcap_setClb_bits* to perform the reconfiguration. This function accepts two crucial function arguments:

- Location co-ordinates of a LUT is used to generate the frame address, which can point to a specific frame. This frame contains the truth table entries of the TLUT (that is implemented on this physical LUT).
- Truth table entries, that contain the specialized bits generated after the specialization stage of the DCS tool flow. The LUT truth table entries need to be overwritten with these specialized bits.

5.4.2 Multiple-level Configuration Scrubbing

A golden bitstream needs to be maintained at the processor side before the scrubbing is triggered. After writing back (scrubbing) the respective frame, the element's state can be restored.

Different levels of scrubbing are activated by different triggers, creating an online test mechanism for a fault-tolerant architecture. The proposed scrubbing technique operates on three different levels and it is described in the remainder of this chapter.

The proposed scrubbing tools scrub only the frame (lowest configuration granularity) under SEU, using golden bits (stored in a non-volatile memory). The location is obtained via the *XhwIcap_setClb_bits* function which provides the association between specific bits and their exact location. Therefore, if a SEU is detected, the exact frame where the error occurs can be reconfigured, instead of the whole FPGA. This involves frame detection and scrubbing.

TMR deals mainly with SEUs. However, MBUs still need to be prevented from having an adverse effect in the design. Therefore, two different types of scrubbing are needed. The first technique deals with SEUs, while the second technique handles fault accumulation and MBUs. Each voter can be connected with circuitry that will trigger configuration scrubbing on up to four frames. In that case, MBUs that occur on neighboring frames (or in one CLB) can be scrubbed. However, if the MBUs occur in more than 4 consecutive frames, they will not be scrubbed with this technique, as the microreconfiguration approach can currently reconfigure up to four frames, because a single CLB has configuration bits in 4 different frames [86]. Hence, different levels of scrubbing are activated by different triggering mechanisms, aiming to create a fault-tolerant application.

5.4.2.1 Discrete Microscrubbing

After the detection of a SEU, Discrete Microscrubbing (DMS) is triggered on-the-fly by providing the exact configuration frame that needs to be repaired. Then, DMS scrubs only the frame under SEU using the golden bits. This process is triggered automatically without any user interference. Therefore, if a SEU is detected, DMS scrubs only the SEU infected area and reconfigures only the targeted frames. In that way, we are able to reconfigure an exact VCGRA frame instead of the whole FPGA. This process involves the following steps:

- Online test (Detect frames): after a SEU, the affected frames are flagged and the frame with the same address containing the golden configuration bits is returned from the non-volatile memory.
- Scrubbing (Write-back frames): the same frame address is scrubbed and the correct frames are written back to the configuration memory.

First, the TMR or partial TMR is integrated in the target design. Then, during the FPGA's operation, if a voter's output is different than the current output, a SEU has been detected at a specific location. The tool then applies DMS by providing the exact configuration frames that need to be repaired, by triggering the microreconfiguration process [88]. We need to maintain a golden bitstream at the processor side before we trigger the scrubbing using microreconfiguration. After writing back (scrubbing) the respective frame, the state can be restored. Subsequently, DMS scrubs only the frame under SEU, using golden bits. Therefore, if a SEU is detected, we are able to reconfigure an exact frame instead of the whole FPGA. This involves frame detection and scrubbing.

5.4.2.2 Microscrubbing

The second level is called Micro-Scrubbing (MS) and repairs periodically the FPGA's bits, to deal with Multiple Bit Upsets (MBUs) and fault accumulation, that are 10% of the total faults [149]. This process is completed in three steps:

- Read frames: The FPGA's frames are read from the configuration memory.
- Modify frames: the current truth table entries are replaced by the specialized bits, from a copy stored on a non-volatile memory.
- Write-back frames: using the same frame address, the modified frames are written back to the configuration memory, thus accomplishing MS and efficiently repairing the erroneous VCGRA.

Microscrubbing is enabled in the case the FPGA's architecture needs to be adapted, or more than 1 bits have been affected (a MBU has occurred). When this trigger is enabled, the tool scrubs the FPGA.

5.4.2.3 Parameterized scrubbing

Dynamic reconfiguration capabilities are highly valuable in aerospace applications, since on-site reconfiguration allows the ability to adapt and upgrade electronic functions, or create designs that can overcome a specific computational request. Thus, the parameterized FPGA configuration approach can be used, as it can allow the design to be adapted via parameterized reconfiguration without FPGA interruption, to rapidly evaluate a specific computation [13]. In fact, the third and final level is called parameterized scrubbing. It is enabled in the case the (parameterized) application needs to be adapted, or a critical error has occurred in a parameterized application. When this trigger is enabled, a parameterized scrubbing is initiated. In this approach the parameterized elements (infrequently changing inputs) have been implemented as constants and the design has been optimized for these constants. Then, the proposed tool scrubs the entire bitstream, by evaluating a new specialized bitstream (from the protected copy) and rewrites the bitstream.

5.5 Custom Reconfiguration Controller

5.5.1 Introduction

A reconfiguration controller is needed for any reconfiguration, including scrubbing. Commercial FPGAs include reconfiguration controllers [152, 154]. However, these controllers are more expensive than needed for scrubbing in terms of area needed and reconfiguration speed. Hence, in this section, two custom reconfiguration controllers are introduced, that have been designed to support the proposed configuration scrubbing schemes on a self-reconfigurable platform for the Zynq-SoC. In the previous section, dedicated scrubbing of certain frames has been introduced that targets the Xilinx FPGA column-based architectures. In the remainder of this section two reconfiguration controllers are introduced, that have been designed to support this functionality.

FPGA vendors offer controllers that oversee autonomous reconfiguration, fault detection and recovery operations and the golden copy of the recovery bitstreams. These controllers enable operations such as reading and writing the FPGA configuration memory. Various IP and academic controllers have been developed, such as HWICAP [154], the SEM controller [152] and the PCC controller [46], that handle specific requirements of the radiation fault-tolerant applications and offer solutions to detect, manage and correct soft errors in the configuration memory. Despite their ease of use, these controllers suffer from a large area overhead, slow performance, and lack of flexibility, as they cannot be adapted alongside design changes, they are not fault-tolerant and they don't offer protection for custom (overlay) applications. They are able to detect and recover faults 50% of the time and the recover process is time-consuming [152].

5.5.2 Prerequisites

ICAP

The Xilinx SRAM-based FPGAs contain a set of components to execute the reconfiguration, such as the ICAP, a data access bus and an embedded processor (PowerPC or ARM Cortex-A9). The ICAP is a built in hardware macro, which has direct access to the configuration memory and it requires a reconfiguration controller that is built as part of the design, to manage bitstream movement between the ICAP macro and the processor. The HWICAP is a reconfiguration controller that contains a complex state machine and a First In First Out (FIFO) buffer that accesses the bitstreams from the configuration memory of the FPGA. Reconfiguration time is a major limiting factor for a Xilinx FPGAs, due to the complexity of the HWICAP architecture and the lower communication bandwidth between the processor and the ICAP controller.

The ICAP contains two data ports for reading and writing the data. Each bus supports a data width of 32-bits. It has a clock input and an active-low ICAP enable input. A "RDWRB" signal selects the direction of the data. ICAP's maximum clock frequency is 100 MHz.

There is a series of commands used to access the configuration bitstreams of an FPGA. They have to be written to the ICAP's input for every rising edge of the clock cycle. These commands help the user to orchestrate the ICAP to read the configuration data or write the configuration data to the configuration memory. The ICAP commands include read from and write to the configuration memory.

With a clock input of 100 MHz and a data width of 32 bits, the maximum throughput of the ICAP is 400 MBps [50]. However, the HWICAP that encapsulates the ICAP port supports only 19 MBps due to its inefficient architecture that contains a complex state machine and a communication overhead between the ICAP and the processor which is unnecessary for an application that has PConf enabled. In that case, the designer can use the MiCAP.

MiCAP

Various optimizations have been proposed, to improve the reconfiguration speed of reconfiguration controllers, such as introduction of placement constraints and custom reconfiguration controllers for the Xilinx 7 series [88, 89]. MiCAP [88] consists of two asynchronous FIFO buffers, a state machine and the ICAP, synchronized by a clock with a frequency of 100 MHz. The ICAP read and write commands and the (specialized configuration) data which are to be written into the configuration memory are stored in the first asynchronous FIFO buffer, the input buffer. The second asynchronous FIFO buffer, that is the output buffer stores the data fetched by the ICAP. The ICAP gives access to the FPGA's configuration data. The ICAP state machine controls the controller's read/write activities.

5.5.3 Motivation

In order to efficiently implement the fault mitigation through reconfiguration, we need an efficient reconfiguration controller. With that in mind, a lightweight reconfiguration controller is proposed, that supports the proposed mitigation and scrubbing schemes. Additionally, its fault-tolerant version is also discussed. The benefit of the proposed controllers is that it requires less area resources, it supports fault injection and performs fast reconfiguration to repair SEUs.

5.5.4 Superimposed Reconfiguration Controller

The Superimposed Reconfiguration Controller (SRC) has been extended from [88]. It has been designed in order to control the fault mitigation process. It is a reprogrammable hardware/software co-design that can be easily extended to support different fault detection techniques. In the core functionality of the SRC there is an optimized state machine for microreconfiguration [88]. The depth of its FIFO is long enough to store multiple frames that define configuration of one LUT. This results in an improved reconfiguration speed. It is also light weight and hence power efficient compared to HWICAP. The SRC provides fault monitoring, detection, injection and correction, by rapidly reading and writing specific frames in the FPGA configuration system.

5.5.4.1 Architecture

The SRC has an ICAP state machine and it is connected with a bidirectional Input buffer, two asynchronous connections to the Output buffer and to the ICAPE2 Macro. The architecture is similar to MiCAP, however the Input Buffer contains a wider variety of commands and the state machine allows fault injection and fault tolerance processes. The Input buffer is an asynchronous FIFO buffer that holds the ICAP read, write and reliability commands along with specialized configuration data which are to be written into the configuration memory. All the data read from the configuration memory via the ICAP is stored in the output buffer. Once the data is ready, the processor has to read the frames from the output buffer. Using this element the commands and data can be read or written into the FPGA configuration memory with the ICAP module. This is depicted in Figure 5.8.

The SRC is a fully synchronous design using the src_clk as the single clock. All elements are synchronous to the rising edge of this clock. As a result, all interfaces are also synchronous to the rising edge of this clock. The SEU correction interface and the fault injection interface use the microscrubbing functionality enabled by the SRC interface to monitor and correct SEUs, by reading modifying and writing back frames. When the SRC is in the IDLE state, the ability of the controller to detect and correct errors is suspended. If it detects an input from the

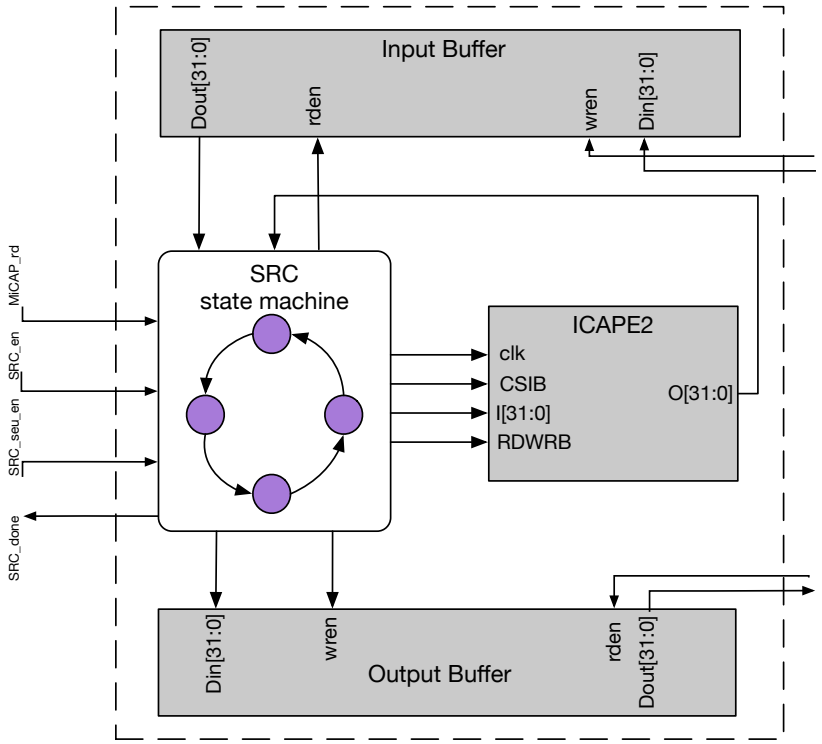


Figure 5.8: Details of the SRC architecture

user to inject errors into the configuration memory, the fault injection interface provides a specific location and commands the controller to create a bit flip into the configuration memory via microreconfiguration.

5.5.4.2 State Machine

The SRC's state machine has additional states compared to MiCAP that are designed to improve the reliability of the design. Additionally to the Read and Write state, the SRC has the SEU and FI state. The SEU_en state, is the state that is activated if a SEU has occurred. The FI_en state, is the state that is activated if the system needs to undergo fault injection.

The SRC State machine contains various states that orchestrate the controller's read and write activity and mitigation processes. The state machine handles the multiplexing of data between the frames of the input buffer and the commands of

the ICAP, from the single port RAM to establish the reconfiguration process. The state machine contains three major states: a wait state, a read state and a write state.

In the *Wait* state the SRCs state machine waits until all the data (frames + ICAP commands) are filled into the input FIFO.

In the *Read* state,

1. SRC_en MiCAP_rd is set to high
2. RDWRB is set to high
3. CSIB is set to low so ICAP primitive is enabled
4. The read command in the input buffer is fetched and written to the ICAPs input port. The command is written in the rising edge of the clock.
5. Once the read command is sent, the ICAP starts fetching the configuration data. The frames fetched from the ICAP are written into the output buffer.
6. CSIB is set to high so the ICAP is disabled
7. SRC_done is set to high

In the *Write* state,

1. SRC_en is set to high and MiCAP_rd is set to low
2. RDWRB is set to low
3. CSIB is set to low so the ICAP primitive is enabled
4. The write command is fetched from the input buffer and the command is written to the ICAP's input port. The command is written in the rising edge of the clock.
5. Once the write command is sent, the ICAP knows that next incoming data is the configuration data that has to be written into the configuration memory of the FPGA. Now the state machine reads the data from the input buffer and writes the data into the ICAP input port. The ICAP continues to write the data sent from the input buffer into the configuration memory until the input buffer is empty.
6. CSIB is set to high so the ICAP is disabled
7. SRC_done is set to high

The fault injection state in the state machine is a process were the reconfiguration controller after reading the data, reverses them, creating a bitflip. Therefore the 0 becomes 1 and vice versa. Therefore, in the *Fault Injection* state,

1. SRC_en and SRC_SEU_en is set to high and MiCAP_rd is set to low
2. RDWRB is set to low
3. CSIB is set to low so the ICAP primitive is enabled
4. The write command is fetched from the input buffer and the command is written to the ICAP's input port. The command is written on the rising edge of the clock.
5. Once the write command is sent, the ICAP knows that next incoming data is the configuration data that has to be written into the configuration memory of the FPGA. Now the state machine reads the data from the input buffer and writes the data into the ICAP input port based on the stuck-at fault (0 or 1).
6. CSIB is set to high so the ICAP is disabled
7. SRC_done is set to high

One more state added in the state machine is the SEU state. Here, if this state is enabled, the data are under SEU and they are scrubbed. Hence, in the *SEU* state,

1. SRC_en, SRC_SEU_en MiCAP_rd are set to high
2. RDWRB is set to high
3. CSIB is set to low so the ICAP primitive is enabled
4. The read command in the input buffer is fetched and written to the ICAPs input port. The command is written in the rising edge of the clock.
5. Once the read command is sent, the ICAP starts fetching the configuration data. The frames fetched from the ICAP are written into the output buffer.
6. RDWRB is set to low
7. MiCAP_rd is set to low
8. The write command is fetched from the input buffer and the command is written to the ICAP's input port.
9. Once the write command is sent, the ICAP knows that next incoming data is the configuration data that has to be written into the configuration memory of the FPGA. Now the state machine reads the data from the input buffer, inverses the data and writes the inversed data into the ICAP input port. The ICAP continues to write the data sent from the input buffer into the configuration memory until the input buffer is empty.

10. CSIB is set to high so ICAP disabled

11. SRC_done is set to high

12. SRC_SEU_en is set to low, so the reliability aspect is disabled.

5.5.5 Fault Tolerant Reconfiguration Controller

Reconfiguration controllers can be designed/extended to provide reliability to a COTS FPGA designed to operate in a High-Reliability environment. However, there are no mitigation techniques to ensure that the actual controller is reliable.

In this work, partial TMR is added to the SRC controller. A widely-used method for SEU mitigation is applied, where the flip-flops are triplicated and voting logic is added [49].

Partial application of TMR is a technique that can be used for reducing SEUs on an FPGA design. In that scheme, the sequential logic is triplicated, instead of the entire component. This is a trade-off between resource utilization and fault tolerance. The TMR flip-flops have been designed taking synthesis tool optimizations into account, to avoid the removal of any redundant parts. In fact, the VHDL code utilizes attributes specific to the Synplify synthesis tool [141]. The FT-SRC Controller is depicted in Fig. 5.9.

Partial TMR provides redundancy on the sequential elements only. It provides only a single set of input and output ports for the entity, except for the clock and reset ports that can be triplicated. The triplicated sequential elements can be voted with a single output voter that is also implemented in the logic. The importance of feeding back the voted result to all voted sequential elements is discussed in [15, 54]. It is mainly done to restore the state of all redundant sequential elements and to avoid error build up. A schematic of partial TMR is depicted in Figure 5.10.

With SRC and FT-SRC the bottleneck is the data transfer between the processor and the ICAP during the configuration. In order to overcome this it is possible to use the high performance port (HP0) of the Zynq-SoC. Since the HP ports can be accessed only by a master from the PL region, a DMA controller can be used that acts as a master to the high performance ports. The data transfer that occurs through the HP0 port of the Zynq-SoC, can establish a very high speed data transfer that will contribute to faster reconfiguration speed ($3\times$ faster). However, this design option was not used for this work, as the goal of this dissertation was to provide a fault-tolerant reconfiguration controller that remains relatively small in size. However, if an AXI-DMA controller and the HP0 were used, the area of the controller would be increased dramatically (at least 2 orders of magnitude).

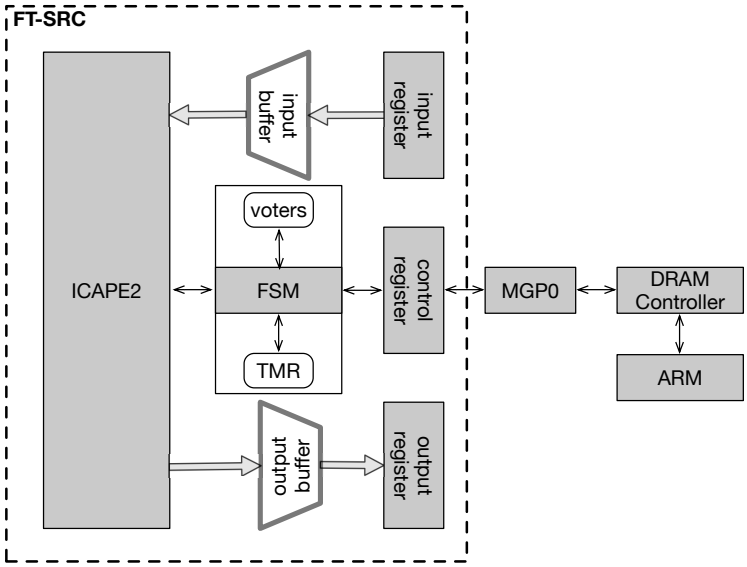


Figure 5.9: Overview of the Fault Tolerant Controller on a Zynq-SoC.

5.5.6 Results

5.5.6.1 Microscrubbing

Both DMS and MS target the lowest granularity of configuration for 7-series FPGAs, which is the configuration frame. For the 7-series FPGA, each frame is 101 words of 32-bits each (3,232 bits per frame). Both techniques target the error correction on a frame level.

In a conventional implementation, the HWICAP is used as a reconfiguration controller. However, the MS and DMS are realized using a custom reconfiguration controller. The reconfiguration speed with this controller is improved by 3× over the HWICAP. The MS and DMS prove to be more energy efficient compared to the conventional configuration. The results are shown in Table 5.1. Power analysis shows that both DMS and MS have a more energy efficient reconfiguration controller (14.8× & 7.6× respectively) than the conventional Xilinx controller (HWICAP).

MS can periodically reload the reconfiguration bits from a golden instance. Since there is no need for full or partial reconfiguration, only Boolean function evaluation, there is a significant decrease in time (5 orders of magnitude less generation time) [57].

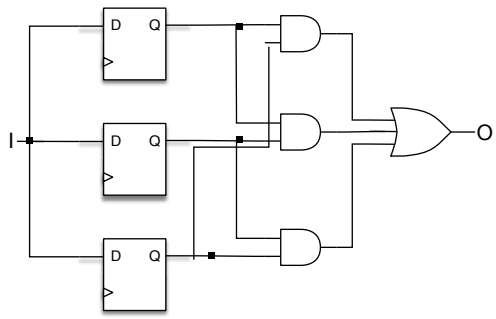


Figure 5.10: Partial TMR: Mitigation of faults by triplication of sequential elements and voting

DMS has only to write-back the correct frames from the correct instance (provided by the FT infrastructure). In this case, there is no need for full or partial reconfiguration, nor Boolean function evaluation. Therefore, there is a $10\times$ decrease in time compared to conventional scrubbing and $3\times$ compared to MS. Finally, in the case the FPGA needs to be changed, parameterized scrubbing (full parameterized configuration) is needed. This is 3 orders of magnitude faster compared to a full conventional FPGA reconfiguration.

Scrubbing technique	**Time (μs)**	**Power (mW) Reconf. (mW idle)**
Conventional Scrubbing	234	62.2 (3.3)
Microscrubbing	64.1	8.1 (1.6)
Discrete Microscrubbing	23.1	4.2 (1.6)

Table 5.1: Comparison of reconfiguration time and power for reconfiguring one LUT with different controllers and scrubbing techniques.

5.5.6.2 Reconfiguration Controller

The results of the design of custom controllers are presented and compared with the standard reconfiguration controllers. The power measurement results of the controllers and the trade-off between their reconfiguration speed and area (resource utilization) are also explained.

The results in Table 5.2 show that the proposed controller (SRC) can be up to $3.6\times$ smaller than the above-mentioned proprietary and academic non-TMR controllers. The results in Table 5.2 also show that the proposed fault-tolerant controller (FT-SRC) requires less resources than the above-mentioned proprietary and

RC	Resources			Optional features	
	Slices	**LUTs**	**FFs**	**Fault Inj.**	**Fault Tolerance**
HWICAP	276	544	587	no	n/a
SEM	250	863	681	optional	n/a
PCC	573	1758	1048	yes	n/a
SRC	120	221	290	yes	no
FT-SRC	242	336	402	yes	yes

Table 5.2: Resource overhead comparison of various standalone reconfiguration controllers for a Xilinx Zynq-7000 design.

	Golden		Synplify TMR/DTMR		Proposed	
	Area	Freq	Area	Freq	Area	Freq
SRC	138	278.5	168/786	228.6	336	270

Table 5.3: Area and frequency of the proposed reconfiguration controller. Comparison between the golden version (SRC), the SRC mitigated with a commercial tool (with TMR and DTMR) and the proposed version (FT-SRC) mitigated with TMR.

academic non-TMR controllers. Direct comparison with fault-tolerant controllers was not possible, as to the best of our knowledge, none of the current proprietary controllers are fault tolerant.

The fault tolerance of FT-SRC has been verified with a fault injection campaign, with the FT-UNSHADES2 system, where 1000 random bit flips haven't resulted to any SEUs, as due to the partial TMR, none of the bit-flips affected any of the mitigated area. The fault injection system that was used will be thoroughly described and analyzed in the following chapters.

5.6 Conclusion

In this chapter, a fault-tolerant scheme has been presented, suitable for future high-reliability applications in radiation environments that include overlay architectures. A technique has been proposed that applies discrete microscrubbing at a target FPGA and a fault-tolerant reconfiguration controller has also been designed. This work provides fast scrubbing with less FPGA resources and it aims at building an integrated fault mitigation scheme that enhances the reliability of COTS FPGAs.

6

In-Circuit Fault Tolerance for FPGAs using Dynamic Reconfiguration and Virtual Overlays

Reassuring fault tolerance in computing systems is an important problem in high-reliability applications. With the interest in commercial SRAM-based FPGAs in radiation environments, it is beneficial to provide continuing operation after an error occurred and runtime reconfigurable recovery from a failure. In this chapter, a two-level FPGA overlay is proposed, with an embedded on-demand fault-mitigation technique. The proposed method exploits spatial redundancy and run-time recovery and can achieve fast run-time recovery with less additional resources in FPGA devices, compared to conventional tools, by providing integrated layers of fault mitigation.

6.1 Introduction

Radiation affects the digital electronic reliability, reducing the overall computing reliability. In contrast to their antifuse radiation hardened counterparts, COTS SRAM-based FPGAs are in principle more susceptible to radiation effects, such as SEUs, as they can alter the hardware configuration. The radiation effects are caused by high-energy subatomic particles from the cosmic radiation. A detailed description of this process is analyzed in Appendix A. These particles can cause

extensive damages in the functionality of FPGA components such as logic blocks, I/O blocks and DSP blocks.

As it was analyzed in Chapter 5, in COTS FPGAs, there is no radiation hardening. In that case, fault mitigation is used. Mitigation of radiation-induced hardware faults in COTS FPGAs considers a wide variety of faults, such as SEUs, SETs, etc. The basic approaches include Triple Modular Redundancy (TMR) and configuration scrubbing.

6.1.1 Motivation

In order to create a fault tolerant design, the designers currently employ existing TMR tools, such as Synplify Premier, XTMR and Frame ECC and then run the conventional design flow, which introduces significant resource overhead ($\geq 3\times$) and reduces the overall performance. Also, these tools don't offer any protection against MBUs and fault accumulation. Additionally, SRAM-based FPGA overlays have been gaining traction among researchers, as it was analyzed in Chapter 4. Hence, FPGA overlays, such as VCGRAs are often used because of their potential to reduce development costs in a range of High-Performance Computing applications [57].

The COTS FPGAs do not undergo a radiation hardening procedure, as it was discussed in Chapter 5, and they are not tested for harsh radiation environments, leaving the system susceptible to radiation-induced faults.

The previous chapter outlined the radiation effects that cause soft errors in SRAM-based FPGAs and proposed scrubbing techniques to recover from the failures and reconfiguration controllers to support these functionalities. The objective of this chapter is to introduce mitigation, validation and fault injection techniques in FPGA overlays, to enhance their fault tolerance.

6.1.2 Related Work

Radiation tolerance can be improved with mitigation techniques, that use temporal and spatial locality to protect the COTS FPGA. The basic approaches include TMR and configuration scrubbing, as it was analyzed in Chapter 5 [11, 85, 102].

CGRAs and FPGA overlays can also be used for fault tolerance. Coarse-grained architectures have also been used in [40, 93, 126], where a coarse-grained reconfigurable array (CGRA) is used as a de-facto natural reliability architecture against SEUs [93]. The authors apply hardware redundancy and voting. However, this technique is not applied on FPGAs and therefore is not equipped with a scrubbing mechanism. Additionally, the computational overload in these architectures gives limited flexibility to add mitigation mechanisms without a performance penalty.

Academic works have used overlay-based incremental techniques to add additional logic [31, 47, 61, 62], to trace internal signals, as it was analyzed in Part II of the dissertation. Additionally, in [32, 60, 63] academic works demonstrate the post-implementation insertion of verification infrastructure. Researchers have also used the properties of compiler-optimized HLS circuits to enable dynamic tracing to increase internal signal observability [43, 44]. They have also adapted some of the added circuitry for even more efficiency. However, these works demand adequate empty regions to install their circuitry. On a highly-utilized FPGA, it may be hard or even impossible to find a region large enough to implement additional logic, that is essential for techniques such as TMR. These techniques can be proven problematic in highly utilized FPGAs and hence cannot be used to generate spatial redundancy circuitry. Additionally, they can create a new longer critical path and cause routing congestion. Most importantly, these techniques are only available for compiler-optimized HLS designs or academic FPGAs and don't offer any spatial redundancy or scrubbing mechanism.

The virtual and overlay architectures provide more flexibility. However, none of the mentioned techniques can currently support efficient fault tolerance of virtual architectures without extensive resource overhead or time-consuming recompilation. Thus, in order to efficiently mitigate FPGA overlays and virtual designs in-system, the current limitations of the fault mitigation techniques need to be addressed.

6.1.3 Contribution

In this chapter an in-circuit reliability-aware infrastructure is proposed, tailored for overlay architectures on FPGAs, with an integrated fault tolerance mechanism. With the use of the PConf tool [57], we aim to improve the reliability of COTS FPGAs in radiation environments, with minimal area overhead. Hence, we make the following contributions:

- An in-circuit reliability technique with fault tolerance for an overlay design.
- An extension for the parameterized Configurations tool flow, in order to provide the reliability functionality, by integrating spatial redundancy, scrubbing and fault injection to the target design.
- Adaptations for commercial FPGA architectures
- An experimental study that demonstrates how the proposed techniques can be applied to create reliable FPGA overlays.
- A fault injection design flow to validate the results.

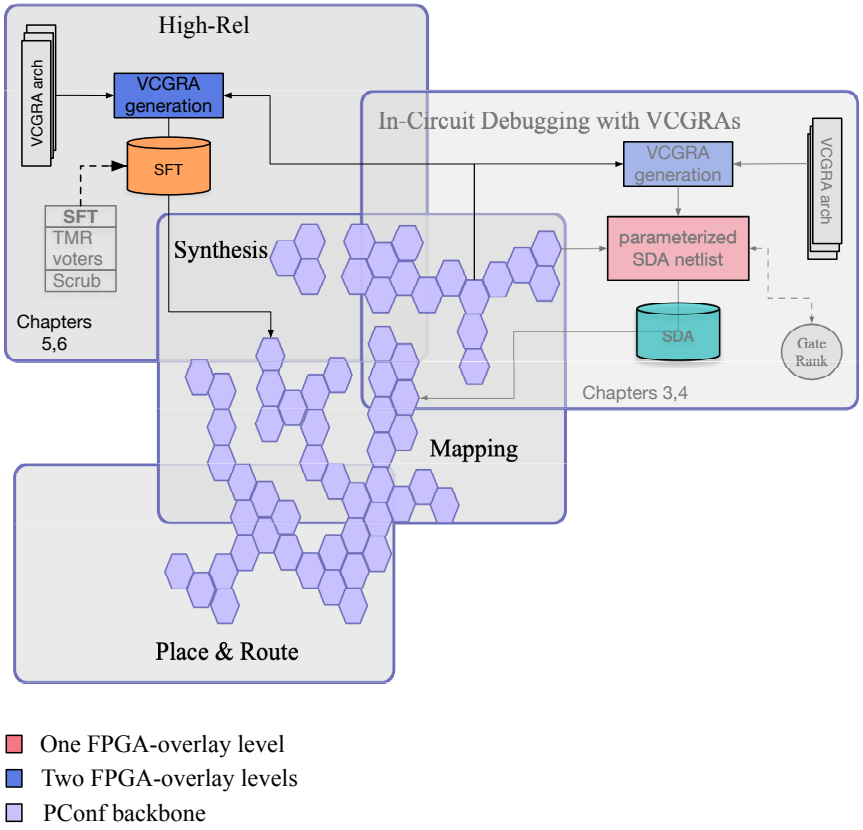


Figure 6.1: Visualization of this chapter's contribution

The contribution can be visualized in Figure 6.1 and where it is placed in this dissertation regarding the rest of the contributions. It adds a high-reliability step in the next generation tool-flow, that was first depicted in Chapter 1 in Figure 1.1(c) and extended in Part II. In more detail, this chapter adds fault tolerance techniques (spatial redundancy, voters and scrubbers) and FPGA overlays.

The remainder of this chapter is organized as follows: First, the proposed technique and the proposed methodology are introduced, that are dedicated to inserting SEU mitigation infrastructure in overlay FPGA architectures. The proposed work leverages the PConf and the VCGRA architecture and introduces SEU mitigation in (overlay) FPGA architectures. Additionally, a tool is also introduced that adds mitigation techniques (TMR, voters) to the design with significantly less area overhead than the proprietary tools and it also contains an in-circuit fault injection

infrastructure that can validate the design during runtime.

The accompanying tool adds mitigation techniques (TMR, voters) to the design with significantly less area overhead than the proprietary tools and it also contains an in-circuit fault injection infrastructure that can validate the design during runtime. Then, this technique is paired with the multi-level scrubbing introduced in Chapter 5 and it is shown how a designer can handle fault accumulation by using multiple-level scrubbing techniques. Finally, a case study is presented, alongside the subsequent steps taken to provide SEU mitigation in two image processing applications: a Sobel Edge Detection filter and a Convolutional Neural Network (CNN). The proposed technique is validated with a fault injection system based on the FT-UNSHADES2 platform in the end of this chapter.

6.2 Architecture

This section describes the proposed fault mitigation scheme for safety-critical applications that use COTS SRAM-based FPGAs and that can be integrated with an overlay architecture, such as a VCGRA.

6.2.1 Superimposed Fault Mitigation

The VCGRA is constructed in such a way, that there are clear layers of PEs and VCs [90]. The proposed architecture is leveraged by inserting the mitigation scheme on each individual VCGRA layer. In that way the design's robustness is increased and fault propagation is reduced, as each layer is mitigated and voted. Additionally, it allows to create a mitigation scheme and then adapt it during runtime if it is needed.

In safety-critical applications it is crucial to be able to have a detection mechanism that can assess if a SEU occurs. Hence, during the construction of the VCGRA at the target FPGA, TMR is directly applied. TMR is performed on the overlay infrastructure at a PE level and not at a gate or block level as in the state of the art. This creates an overlay TMR infrastructure on top of the VCGRA architecture. In that way the TMR can be adapted on-the-fly, as soon as the VCGRA is created (or it is adapted after deployment). Moreover, since it is performed on a higher abstraction level, not all TMR resources are translated directly to real FPGA resources, as LUTs and flip-flops can be more efficiently designed to implement the overlay and map it efficiently, reducing the resource overhead of the mitigation scheme.

The VCGRA, as similar coarse grained architectures, is constructed with multiple layers of PEs/VCs. TMR is co-designed alongside the VCGRA. Therefore, TMR is applied after each layer of PE/VC, as it is shown in Fig. 6.2. One voter is placed directly at the output of the triplicated PE. The voted output serves as an in-

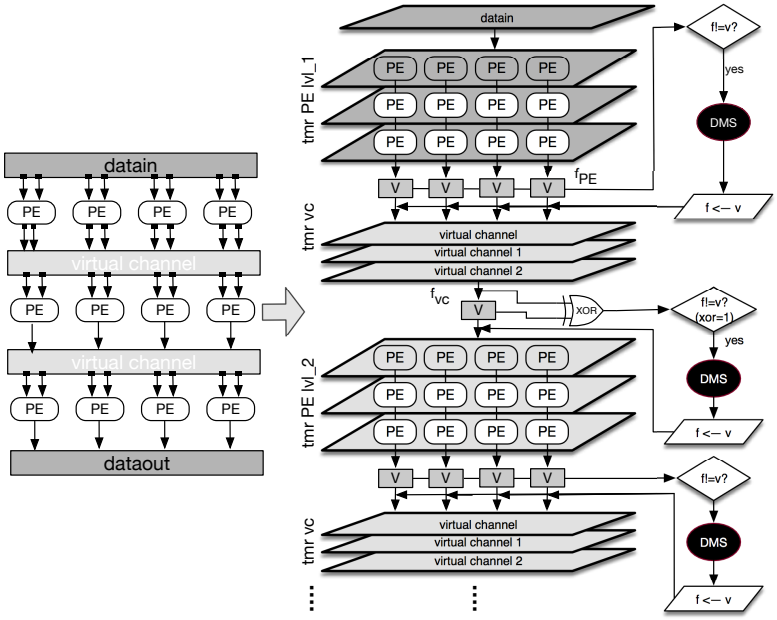


Figure 6.2: Overview of a VCGRA grid integrated fault mitigation for PEs and VCs.

put for the next level (and not the PE's output). Hence, the VCGRA has redundant PEs and VCs, as each element is triplicated and voted. Thus, by leveraging the VCGRA architecture, the SEU effect can be eliminated (via redundancy), before it can be propagated to another layer of PE/VC.

The voters are a standard mitigation technique. They provide a majority voting functionality that ensures correctness at least at the two out of three copies of the system, in order to remain operational. Thus, the voter plays a pivotal role in ensuring the correct operation of the system. This logic is extended to provide a repair functionality alongside TMR and to avoid fault accumulation. A majority voter is installed after each triplicated element. Hence, when the TMR result is different than the original target PE or VC, the voters can trigger on-the-fly circuitry that can repair the current VCGRA layer. In that way, the SEU can be detected and repaired on each layer, eliminating the need to scrub the entire VCGRA (or FPGA) periodically. This is visualized in Fig. 6.2, where reconfiguration is enabled in a target PE and the target PEs are scrubbed.

It is also common practice to triplicate also the voters, to provide an additional

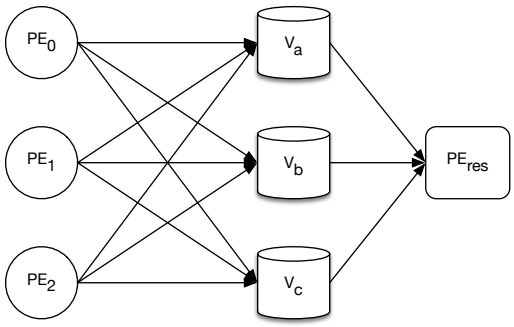


Figure 6.3: Triplicated voters

level of mitigation. This method requires additional area resources, but it includes an additional level of redundancy. Hence, in this work, we provide the possibility of triplicating the voters as well, if there are sufficient FPGA resources. In that way, the PE outputs are triplicated and voted. Then, the voters are also triplicated and their result (PE_{res}) is used as an input for the next VCGRA layer. This is visualized in Fig. 6.3.

6.2.2 Superimposed Fault Injection

Here, a combination of location specific fault injection is proposed, with the reuse of the microscrubbing methodology that is already integrated in the design, as it was presented in Chapter 5. In that way, a bit-flip can be generated through microscrubbing since the location of the bits is already known. Hence, since the SEUs in a PE can be modeled as bit flips, the bit at the register output can be inverted. Subsequently, its effect on the behavior or functionality can be observed during simulation. Moreover, the complexity of the fault injection campaign is reduced, as the bit-flip is applied on the output of a specific PE (group of LUTs), reducing the amount of extra hardware resources needed. The fault injection is on a higher abstraction level (PEs and VCs) and not on specific gates nor bits.

The fault injection infrastructure is added incrementally, alongside the mitigation scheme, by reusing the TMR resources that are allocated in the reconfiguration resource of the FPGA. The infrastructure that is reused are multiplexers that are installed at the PE's output that connects it with the triplicated PEs and voters. To integrate the fault injection functionality, the multiplexers are redesigned, to accept one more parameter-input value (one additional selection bit). The virtual LUTs and routing are implemented in the reconfiguration infrastructure and hence do not need extra resources. Hence, the LUT utilization remains constant, since

virtual LUTs and routing connections have been used exclusively, to describe all the additional multiplexers, in the reconfiguration infrastructure of the FPGA. The designer has no access in the reconfiguration infrastructure with the conventional methods, leaving routing resources unexploited. In that way, by exploiting these resources, a fault injection functionality can be introduced at specific locations in the FPGA, that are accessed via reconfiguration. In these locations a stuck-at fault or a bit-flip can be introduced.

6.2.3 Configuration Scrubbing

After the installation of the TMR, the scrubbing occurs to avoid SEUs, MBUs and fault accumulation. Here, the approaches that are used (MS, DMS) have been first introduced in Chapter 5. Here, they target the PEs specifically. The first approach has been designed to correct the VCGRA from SEUs. Hence, the first scrubbing technique, is to target a SEU on a specific PE. The second technique, is to handle fault accumulation and MBUs on various PEs/VCs, by periodically scrubbing the VCCGRA. Hence, different levels of scrubbing are activated by different triggering mechanisms, without disrupting the FPGA operation, aiming to create a fault-tolerant application.

Hence, DMS, is used to target a SEU on a specific PE. After a subgrid is constructed, during the design stage, the fault mitigation scheme is integrated alongside the VCGRA. Then, during the FPGA's operation, if one of the triplicated outputs has a different result than the other two, this means that a SEU has been detected at a specific PE or VC. The tool then triggers DMS by providing the exact configuration frames that need to be repaired via microreconfiguration [88], that is also able to restore the state of the PE after scrubbing. A golden bitstream needs to be maintained at the processor side before the scrubbing is triggered. After writing back (scrubbing) the respective frame, the PE's state can be restored. Subsequently, DMS scrubs only the frame (lowest configuration granularity) under SEU, using golden bits (stored in a non-volatile memory). The location is obtained via a function which provides the association between specific bits and their exact location. Therefore, if a SEU is detected, the exact VCGRA frame where the error occurs is reconfigured, instead of the whole FPGA. This involves frame detection and scrubbing.

Microscrubbing (MS) repairs periodically the VCGRA's bits, to deal with Multiple Bit Upsets (MBUs) and fault accumulation, that are 10% of the total faults [149]. The MS is enabled by the processor after multiple DMS cycles. This process is completed in three steps that include reading the VCGRA's frames from the configuration memory, replacing the current truth table entries of a PE/VC with the specialized bits (from a copy stored on a non-volatile memory) and writing back the modified frames to the configuration memory. Microscrubbing is built on top

of the parameterized FPGA configuration technique. It is enabled in the case the FPGA's overlay architecture needs to be adapted, or a critical error has occurred. When this trigger is enabled, a parameterized reconfiguration is initiated. Then, the proposed tool scrubs the entire VCGRA, evaluates the new VCGRA and integrates it with the adjacent TMR infrastructure. MS is similar to the technique described in Chapter 5. However, here it is used for scrubbing PEs.

The microscrubbing technique can be offered for any SRAM-based FPGA that is able to apply DCS. The VCGRA-TMR adaptations can be applied in a design that can benefit from such architecture and has relatively scarce resources. Therefore, the mitigation decision graph that was depicted in Figure 5.7 from Chapter 5 can be extended into Figure 6.4.

6.3 Tool Flow

Before the mitigation schemes (spatial redundancy, fault injection, monitoring) can be included in the design, the respective back-end tools that generate them need to be integrated in the conventional (parameterized configurations) tool-flow [55].

The generic design tool flow consists of the basic steps, namely synthesis, technology mapping, place & route and bitstream generation. The additions are introduced before the place & route. A pre-synthesis step is needed to create the VCGRA and the mitigation infrastructure, while during synthesis the supplementary files are generated. The supporting tool-flow that implements a fully parameterized VCGRA integrated with the mitigation schemes is shown in Fig. 6.5.

In more detail, in order to create a VCGRA, the user provides an HDL design and the VCGRA architecture and settings. In this work the tool has been adapted to add the mitigation schemes and to create a fault-tolerant design. At this point, the design can be synthesized placed and routed according to [57], in order to allow the designer to create the generic (parameterized) infrastructure. The tool is divided in two parts, namely the offline phase and the online phase. The first (offline) part of the flow occurs only when the VCGRA is designed (or adapted). The second part (online phase) is executed during runtime, where upon a detection of a SEU, the design is repaired by reloading the frames that have been scrubbed. This process is analyzed in the remainder of this section.

6.3.1 Offline phase: Superimposed Fault Tolerant Tool

In order to setup the SEU mitigation scheme, the VCGRA settings first need to be extracted. The settings needed are the number of PE levels used in the application, the function of each PE (or the set of functions), the (possible) connections between PEs and the number of PEs that define each level.

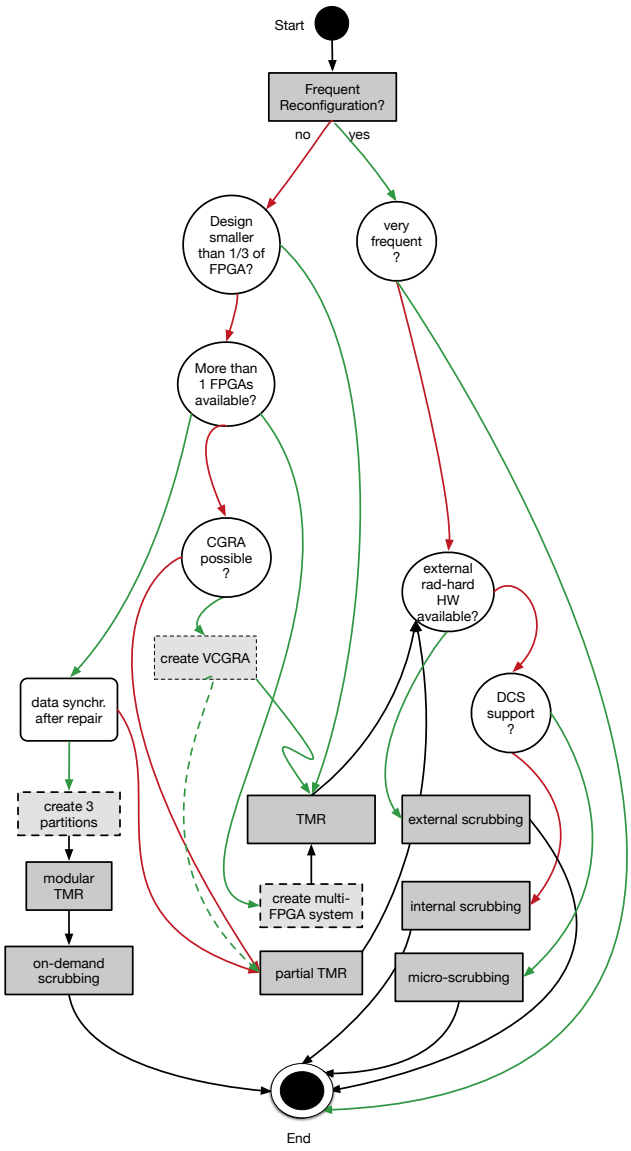


Figure 6.4: The mitigation decision graph that supports DCS, VCGRA and microscrubbing.

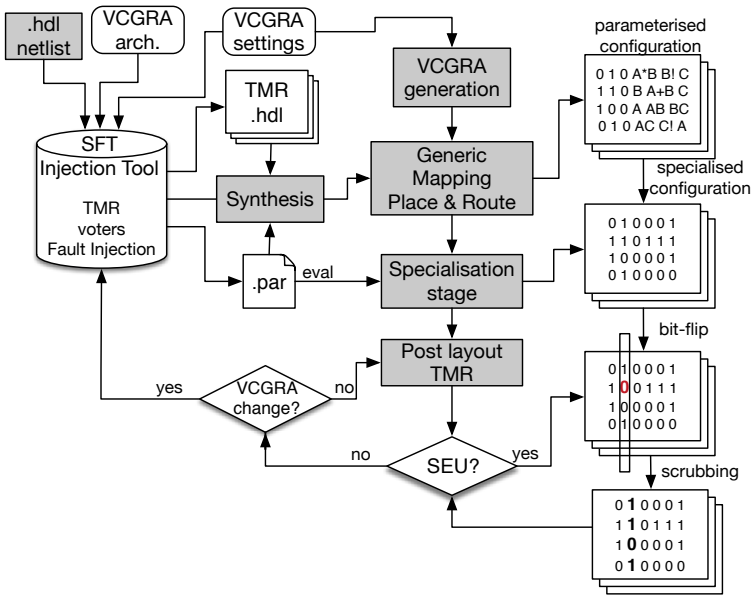


Figure 6.5: The tool flow that integrates the VCGRA implementation with mitigation schemes.

First, the proposed tool constructs a tailor-made TMR scheme, starting from a target application and the overlay infrastructure settings. The VCGRA settings define the levels of PE needed, alongside some information for the .par file that will be generated. In more detail, the user provides the original netlist, written in a HDL language (VHDL or Verilog), the VCGRA settings and the VCGRA architecture as inputs to the Superimposed Fault Tolerant (SFT) Injection tool, that is shown in Fig. 6.5. As soon as the architecture of the VCGRA is defined, the TMR is integrated on the VCGRA, as a second overlay. The output of the tool is an HDL design. The tool can repeat the process for each VCGRA level, by incrementally adding two additional identical versions of each PE on each level for full TMR, or by triplicating only the sequential logic for partial TMR. The tool gets as an input the number of levels and the VHDL description of a PE and extracts a VHDL output where each level of PEs is triplicated and connected with a majority voter.

The hierarchy introduced in the VCGRA is leveraged to our benefit in the TMR design. The output of the majority voter is the input of the next level of the VCGRA. The voted output is the input of the VC. The tool repeats this process numerous times, until all levels are triplicated. This architecture provides a very convenient method to add spatial redundancy, as the hierarchy allows the introduc-

tion of TMR on each PE or on each layer. When the design is in fault injection mode, the tool reuses the multiplexers from TMR. The multiplexer injects a bit-flip at the target PE. The bit-flip is a single operation, where the current frame is modified and written back.

At this point the SFT Injection tool has already defined the granularity, the functionality and the mitigation scheme of the VCGRA, alongside all the possible ways the triplicated and voted PEs and VCs can be interconnected.

For FPGAs that support parameterized and fully parameterized configurations, as it was discussed in Chapter 4, the VCGRA's parameter inputs are mapped in virtual LUTs (with Boolean logic as well as 1's and 0's) that allow their lookup entries to be defined as Boolean functions of the parameter inputs instead of static ones and zeros. The truth table entries (Boolean values) will be microreconfigured upon every change in the VCGRA. This part of the flow creates a parameterized VCGRA configuration. This configuration is a virtual intermediate layer and describes the target application (depicted on the top right schematic of Fig. 6.5). Here, the PEs are mapped into virtual PEs of the VCGRA. Next, with the custom router, optimal connections are created between the voted outputs.

The parameterized configurations approach has some benefits (reduction of the FPGA resources and fast reconfiguration). However, in some cases it is more beneficial to use the conventional approach, such as in the occurrence of many numerical operations and when reconfiguration is frequent and the FPGA resources are not scarce. In these cases, the VCGRA can be implemented on a commercial FPGA without using parameterized configurations. Then, the fault tolerance can be integrated on top of the VCGRA. In order to achieve this, it is essential to follow a very specific procedure, to create a user IP:

1. Describe the application in a form of a graph
2. Create the settings of the VCGRA
3. Generate TMR functionality
4. Create the AXI settings
5. Create a user-IP that contains an AXI wrapper of the VCGRA
6. Implement it on the FPGA with vendor tools.

The outcome of this procedure, is a third-party IP of the VCGRA, where each layer is triplicated. Since here each level is triplicated, without any parameterization, the area is increased. However, since some optimizations still exist, due to the VCGRA architecture, the area overhead is more constrained. This flow is applied for Xilinx Virtex-7 FPGAs. This procedure is depicted in Figure 6.6.

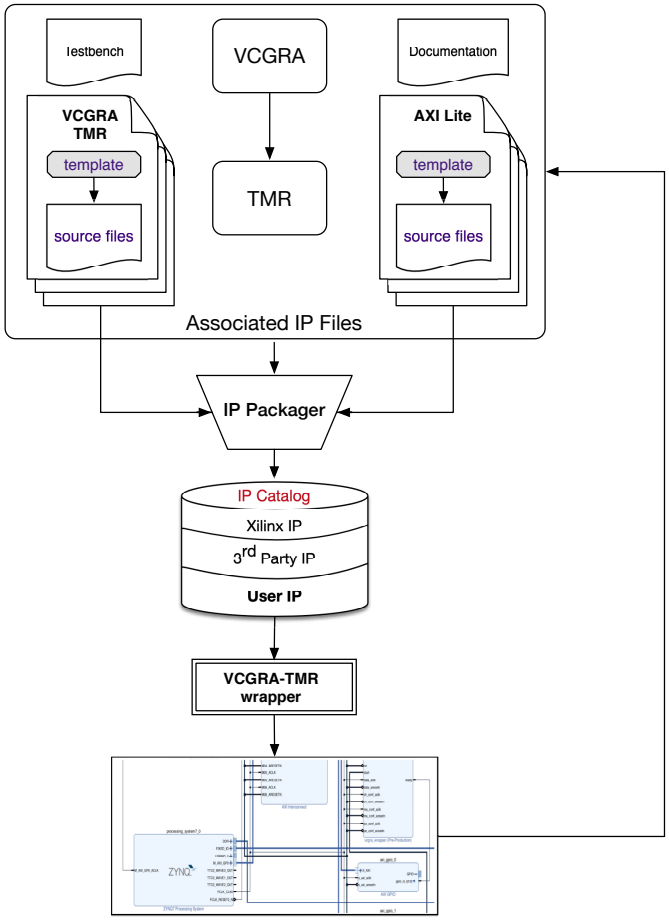


Figure 6.6: Representation of the procedure to create the fault-tolerant VCGRA IP.

6.3.2 Online phase: SEU Mitigation

During runtime (later design stage), the FPGA can be rapidly repaired from SEUs and MBUs on-the-fly via targeted (discrete) microscrubbing. By using the proposed flow the two-level VCGRA gives a new set of triplicated components. The voter examines the PE, until its output doesn't match with the original. When it doesn't pass, the new settings for the components are merged with the specialization stage and create a new specialized configuration with the repaired components. Hence, in case a SEU is detected, the controller activates the microscrubbing process to repair one frame and return to SEU checking.

At the current frame address that a SEU has been detected (after voting), four

frames containing all the truth table entries of a column of LUTs are read from the configuration memory. Then, a function locates the truth table bits of all the LUTs that are present in the frame. The current truth table entries of these LUTs are replaced with the original (golden copy) specialized truth table bits. Finally, with the help of the same frame address, the modified truth table values are updated in all the LUTs of the column by swapping in multiple frames into the configuration memory of the FPGA. This updates all the truth table entries of multiple LUTs that are placed in a single column and completes the Discrete Microscrubbing process.

6.4 Results and Discussion

For the experiments we have used two benchmarks to showcase how the proposed mitigation scheme is applied and how it impacts the original design, in terms of usage and timing. Then, a Fault Injection campaign is executed to validate the mitigation technique. We have used two case studies for image processing applications: one is a Sobel Edge Detection filter [129], and the other is a Convolutional Neural Network (CNN) [84].

First we map the applications into a VCGRA architecture. Then, we integrate our designs with SEU mitigation components, such as TMR, voters and configuration scrubbing and perform an area exploration study by comparing it with a commercial tool and subsequently a fault injection campaign with the use of FT-UNSHADES2 (FTU2). Alongside the case studies, we perform fault injection to verify the fault tolerance of the proposed fault-tolerant reconfiguration controller (FT-SRC).

For the area overhead, we compare the benchmarks with conventional FPGA architectures. We have implemented it on a Xilinx Virtex-5 FPGA and applied TMR with the Synplify Premier tool by Synopsys. The FPGA was selected as the one that supports all the vendor tools that are needed for the experimental study. Then, in order to create the proposed fault-tolerant VCGRA grid with our method, we have applied TMR on each PE.

6.4.1 Sobel Edge Detection Algorithm

The Sobel filter is used for edge detection in image processing. It was first presented in Chapter 5. Starting from a graph representation of a Sobel edge algorithm, we can create a VCGRA grid, as it was explained in Section 4. The algorithm has two different operations (add, mul), between two neighboring pixels and their corresponding filter coefficients. Hence, the operations are modeled as PEs, with 2 inputs: a pixel and a filter coefficient. The edges of the graph can be modeled as VCs. Therefore, we can construct a VCGRA. The architecture has been implemented with the proposed tool-flow.

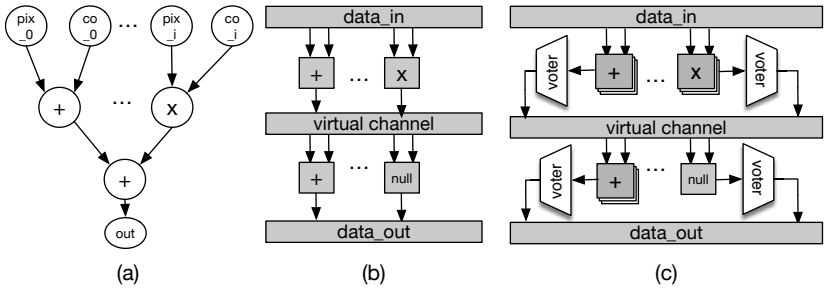


Figure 6.7: Model of a Sobel edge detection filter as a reprogrammable PE. Fig. (a) shows the graph of the filter, Fig. (b) shows how it can be mapped on a VCGRA and Fig. (c) how the fault tolerance can be applied.

	Usage		Timing
	LUTs	FF	Freq (MHz)
PE Golden	62	81	319.5
PE TMR	147	27	189.5
PE TMR (Logic & Voters)	164	43	248.9
PE syn_TMR	205	45	141.5

Table 6.1: Sobel Edge Filter utilization comparison between the golden version, the proposed, the proposed with triplicated voters and with Synplify. The design used is a processing element of the DUT.

For the area utilization of commercial FPGA architectures, we compare the Sobel filter with a conventional architecture without an overlay. We have implemented it on a Xilinx Virtex-5 FPGA and applied TMR with the Synplify Premier tool by Synopsys. Then, in order to create the proposed fault tolerant VCGRA grid we have applied TMR on each PE.

The area results of the area overhead needed, is shown in Table 6.1. We have added 5 VCGRA layers, with 9 PEs per layer. Thus, each PE is triplicated and voted, as it is shown in Fig. 6.7(c). Since it is also integrated with the use of FPGA overlays (VCGRA), we notice an overall reduction up to 30% of the total TMR resources compared to full TMR. To implement the entire Sobel filter we need 45 PEs and 4 VCs. The results are compared with the area overhead produced with Synplify Pro.

6.4.2 Reconfiguration Speed

The time overhead is calculated based on the time to reconfigure all the processing elements and virtual channels. This takes 156 ms and 18.4 ms of reconfiguration

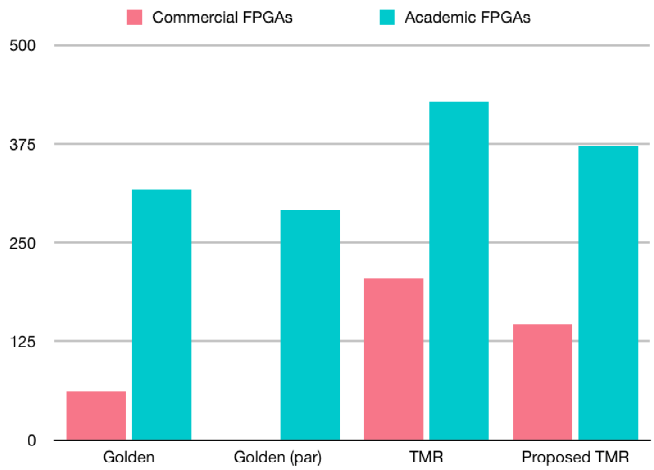


Figure 6.8: Comparison of area results for a Virtex-5 FPGA and an academic architecture that allows parameterized configurations. The y axis is the area in terms of LUTs and the x-axis each design mapped on commercial and academic FPGAs respectively.

time respectively. For this application, the reconfiguration speed is improved by $3\times$ over the HWICAP. Additionally, DMS has a $10\times$ decrease in time compared to conventional scrubbing, as it was analyzed in Chapter 5 and $3\times$ compared to MS, as it was also presented in Section 6.2.3. Both DMS and MS target the lowest granularity of configuration for 7-series FPGAs, which is the configuration frame.[1]

Parameterized Configurations Optimization

The Processing Element has been designed in VHDL and a parameterized application has been compared to a conventional one. In the parameterized application, the parameters are used in order to define the two different mathematical operations of the filter. Due to dynamic data folding, the logic resources (in terms of LUTs) can be reduced up to 24% and the wire length is decreased up to 25% for a parameterized application compared to a conventional one. This has a reconfiguration time costs from $3.4ms$ up to $88.5ms$. A major part of the virtual channel doesn't need LUTs to make them reconfigurable, due to routing multiplexing (parameterization) of the graph's edges. Hence, up to 82% of its logic is mapped on the reconfigurable physical switches instead of physical LUTs and multiplexers (as per the conventional implementation). The wire length is decreased by 76% and the minimum channel width is reduced by 42%. This optimization can be achieved

[1] For the 7-series FPGA, each frame is 101 words of 32-bits each (3,232 bits per frame)

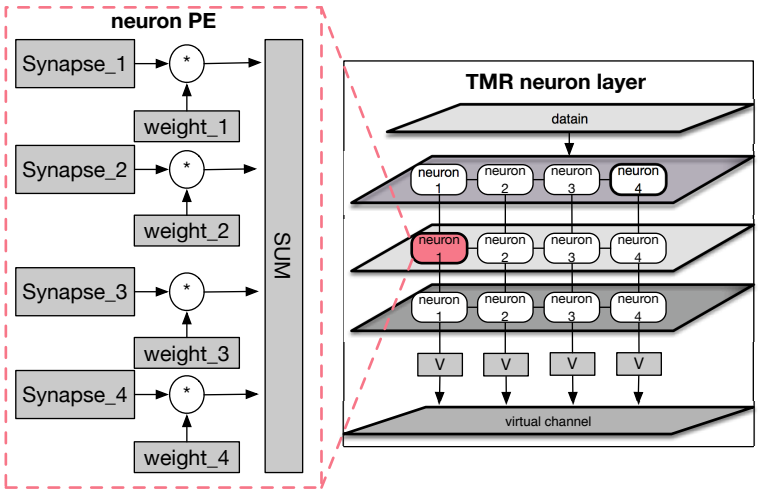


Figure 6.9: Model of a single neuron structure as a reprogrammable PE

at the cost of a reconfiguration time of 4.6 ms. In order to add fault-tolerance (by triplicating the asynchronous logic) in the parameterized design, we have an area increase of 30%. However, with the commercial architectures, the area utilization is more optimized for numerical operations, making the non-parameterized method more efficient in terms of LUTs. However, the area overhead in academic FPGAs is less than the area overhead in commercial FPGAs. This is depicted in Fig. 6.8, where we can observe that the DUT (golden circuit) is only 62 LUTs with the commercial FPGA and 317 LUTs with the academic FPGA. Hence, the original implementation is $5\times$ larger if it is synthesized with academic tools.

6.4.3 Convolutional Neural Network

A CNN is a class of deep, feed-forward artificial neural networks that has successfully been applied to analyze visual imagery. CNNs use a variation of multilayer nodes designed to require minimal pre-processing. Each node is a neuron that uses a nonlinear activation function. Because of this design, CNNs make ideal candidates for overlay architectures, such as VCGRAs, after minimal modifications. Hence, each neuron is modelled as a PE, and the multiple layers of these PEs model a 2-dimensional VCGRA, similar to Fig. 6.2.

The neuron architecture is designed in VHDL and it is depicted in the left part of Fig. 6.9. The neuron allows parallel multiplication of the inputs by the weights. Timesteps are implemented using a clock signal. Our implementation on a VCGRA is smaller compared to the conventional implementation in terms of LUTs.

	Golden		Synplify TMR/DTMR		Proposed	
	Area	Freq	Area	Freq	Area	Freq
1 PE $_{\text{Neuron}}$	36	490.9	155/605	198.1	112	499.7
VCGRA $_{\text{CNN}}$	432	451.7	1954/6574	184.7	1220	457

Table 6.2: Area and Frequency comparison between the golden design (the original circuit without mitigation), with the Synplify tool (TMR and distributed TMR) and with the proposed method. The designs compared are a neuron of a CNN and a complete CNN.

The fault tolerant neuron has been designed in VHDL, by adding TMR at the flip-flops of each neuron. The reliable neuron is smaller if it is implemented with our technique instead of the Synplify Premier [141]. The components that add fault mitigation were implemented with the proposed tool. Since we virtualize the connections between VCGRA and the TMR infrastructure, TMR consumes less additional LUTs than full TMR with Synplify. This gain exists due to the fact that overlay architectures don't always translate in real FPGA resources. The results are shown in Table 6.2. From the results, we can observe that the proposed technique occupies less FPGA resources than with Synplify and its frequency is always higher.

6.4.4 Fault Injection Campaign

In order to validate our example, we performed a fault injection campaign. We have used the FT-UNSHADES2 (FTU2) system [2, 104]. FTU2 is an automated FPGA-based fault injection platform that consists of an FPGA motherboard connected with two custom daughterboards via PCI-Express. The motherboard stores stimuli and configuration data and compares responses. Each daughterboard features a service FPGA and a test FPGA. The service FPGA configures the test FPGA, applies input stimuli, and records output responses from the test FPGA. FTU2 can inject errors to user registers in the design or the configuration memory. All FPGAs in the FTU2 system are Xilinx Virtex-5 FPGAs. Therefore, the target FPGA used in the test described in this paper is a Xilinx Virtex-5 FX70T. Subsequently, all of our designs were adapted and re-implemented for this specific FPGA. We have created a flow (Fig. 6.10) that prepares the design and executes the fault injection campaign.

The designs followed each step of the system described in Fig. 6.10. The RTL description of each design is mitigated and synthesized with the Xilinx flow and with the proposed flow. Then, the designs are implemented on the Virtex-5 FX70T FPGA. The bitstream, the readback and the register location files are generated. At this point the fault injection system is initiated. Three designs were compared during the campaign. The processing element of the Sobel edge detection filter,

	Injector	Campaign		Errors	
		Injections	Runs	Synplify	Proposed
PE: Sobel Golden	Random	100	10	-	354
PE: Sobel TMR	Random	100	10	0	3
PE: CNN TMR	Random	100	10	10	5
FT-SRC	Random	100	10	2	2

Table 6.3: Fault Injection Campaign

a neuron of a CNN, and the fault-tolerant reconfiguration controller. The logic of the designs was mitigated with the Synplify Pro and with the proposed tool.

Multiple fault injection campaigns were executed for the PEs. The Reconfiguration Controller, even though it is originally implemented for Xilinx Series 7 FPGAs making use of the AXI interface, it has been re-implemented, to offer consistency for the fault injection results, as FTU2 currently supports Xilinx 5 FX70T FPGAs. However, the main functionality has remained intact. The service FPGA configured the test FPGA, applied input stimuli, and recorded output responses from the test FPGA. The FTU2 injected 1000 errors to user registers in the design or the configuration memory. By error we mean that an erroneous output has been reported after a SEU occurred. In fact, during the campaign, 100 runs were executed, with 10 induced SEUs per run. Hence, the following results are per 1000 flipped bits.

The results of the fault injection of the designs are summarized in Table 6.3. The fault injections were realized at the same locations, for both Synplify and the proposed method. For the designs tested, the results show that the proposed method (last column) is equally robust compared to traditional TMR with Synplify for the Reconfiguration controller, as both tools eliminate all the SEUs, more robust for the CNN, where 5 errors are reported per 1000 injections and less robust for the Sobel filter, as 3 more errors occurred, compared to the design hardened with the Synplify Pro tool. However, with the proposed fault-tolerant scheme, *351* less errors were reported per 1000 SEUs and the critical bits were reduced significantly (from 354 to 3). After cross-reference of the location of the triplicated registers and the locations of the SEUs that caused an error that propagated to the output, none of SEUs caused an error at these locations, nor it was propagated to the output. Therefore, the tool is robust, as no errors from SEUs were recorded in the critical bits of any of the designs, nor in the reconfiguration controller.

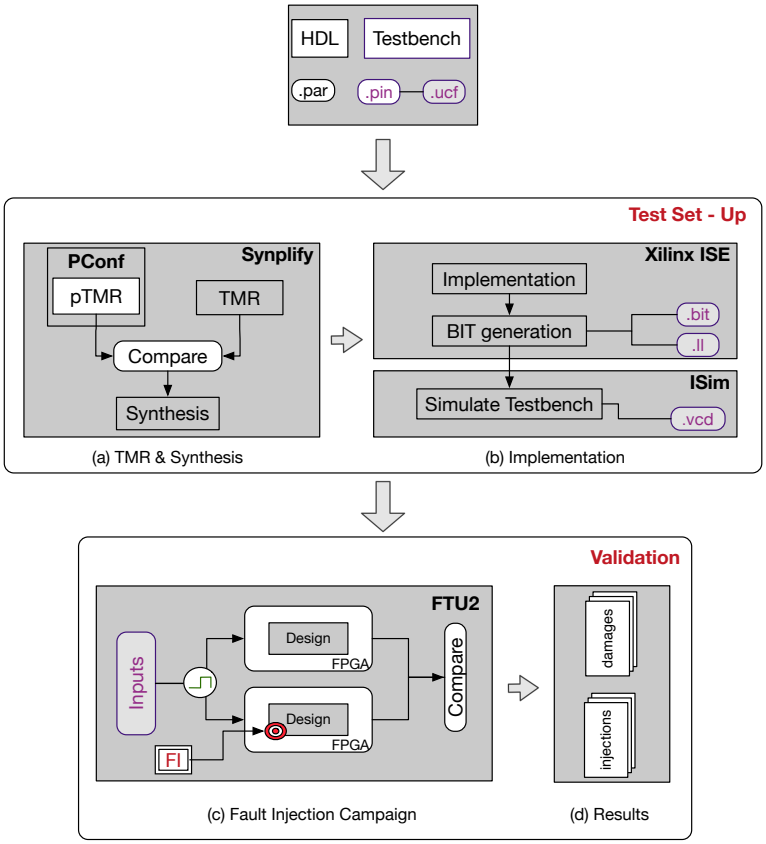


Figure 6.10: The Fault Injection System with its complete flow.

6.5 Conclusion

A fault-tolerant scheme for future high-reliability applications in radiation environments has been presented. A technique that applies TMR and discrete microscrubbing at the FPGA's overlay has been proposed. This work provides fast scrubbing with less FPGA resources and it aims at building an integrated fault mitigation scheme that enhances the reliability of COTS FPGAs. The fault tolerant techniques have been applied to a Sobel Edge Detection filter and to a Convolutional Neural Network and have been verified by a Fault Injection Campaign.

7

Fault Injection with Parameterized Configurations

In this chapter, the parameterized configurations technique is leveraged, to create a novel fault injection method with very low area overhead. This method can be used to test the ability of a design to detect future faults by introducing faults and seeing how the design reacts. It can inject both permanent faults for test set generation, and soft errors, for high-reliability environments. Experimental results demonstrate the practicality of the new technique as, compared to conventional tools, speedups of up to 3 orders of magnitude are observed, alongside $8\times$ *area reduction, and no increase in critical path delay.*

7.1 Introduction

Ensuring a design's functional correctness is very crucial in current technologies. In Chapters 3 and 4 of the thesis, techniques that detect logic and functional errors were proposed. Then, in Chapters 6 and 7, novel fault tolerance and mitigation methodologies were presented, that mitigate the effect of soft errors in the FPGA configuration memory and were validated with long fault injection processes. This chapter introduces a fault injection methodology that is able to significantly expedite the process of injecting faults in a design to locate permanent faults or soft errors. This contribution is added in this thesis because integrated circuits are becoming more and more susceptible to permanent and non-permanent errors, due

to the constant decrease of CMOS feature sizes, or due to radiation environments. Hence, one of the biggest challenges for today's design teams is the complexity of testing and ensuring reliability. It is therefore essential that a design is extensively tested for multiple errors, during various steps of the chip design flow and under possible technological deviations.

One of the most important step in the final testing of fabricated ASICs, or the functional testing of ASIC and FPGA designs is the generation of a complete test set that is able to find the possible errors in the design. Automatic Test Pattern Generation (ATPG) is often done by fault simulation which is very time-consuming. Speed-ups in this process can be achieved by emulating the design on an FPGA and using the actual speed of the hardware implementation to run proposed tests. However, faults then have to be actually built in into the design, which induces area overhead as (part of) the design has to be duplicated to introduce both a faulty and a correct design. The area overhead can be mitigated by run-time reconfiguring the design, at the expense of large reconfiguration time overheads.

In testing a design (after fabrication or FPGA implementation), the crucial point is having the right test set which should be small enough, not to take too much time to run, but also have enough tests to cover most of the possible errors. Within ATPG, fault simulation is important. The main reason to use fault simulation is that one cannot test all input combinations so one has to drastically limit the total number of input combinations. In order to be sure that this limited set covers all (or most) possible errors, the test engineers need to guess what errors may occur and therefore fault simulation is needed. However, due to necessary sequential computations, the time needed for fault simulation is prohibitively large.

FPGA-based fault emulation has proven more efficient than software-based methods to detect if a design will function properly. Therefore, reconfiguration itself can be used to expedite the ATPG process, with often an impact on area and time. Reducing this impact is the main focus of this chapter.

Most ATPG methods rely on fault injection. Fault injection can be realized either by fault simulation or fault emulation. Fault simulation is based on the insertion of a fault model into a Design Under Test (DUT), and the simulation of the design under faulty conditions in order to find a set of input values that can produce an error at one of the outputs for that particular fault. While software-based simulations offer results of good quality they are often very impractical due to their limited speed and memory consuming calculations. Furthermore, as the complexity of integrated circuits continues to increase, consistent with Moore's Law, fault simulation becomes unfeasible. In order to overcome these limitations, circuit designers have turned to FPGAs for the emulation of their complete systems. In FPGA emulation, the faults are built in the hardware and tests can be applied at actual hardware speeds, thus gaining significantly in test set generation time. However, as all faults need to be emulated in hardware, this induces a very

large area cost, unless the same hardware can be reused to emulate different faults. This can be done in FPGAs, as explained next.

Fault injection is a crucial step to prove the reliability of a mitigation technique in safety critical systems. It is used as a step before radiation testing, as it needs no dedicated facilities (radiation chambers) and is significantly faster. Fault injection in safety critical applications has been traditionally done with simulation with vendor tools, such as Questasim [101]. However, this faces the same complications as fault injection for test set generation. As the designs become larger and more complex, the long simulation times become restrictive. FPGA-based fault injection, systems exist that expedite fault injection, such as the one that was used in Chapter 6, where emulation-based fault injection is used (and not its slower counterpart, simulation). Multiple methods have been proposed to expedite fault injection, namely masking (that can mask faults and reduce fault injections), machine learning, and bitstream-based fault injection. However, the main focus of this thesis is FPGA emulation and how is can expedite verification methods. Hence, emulation-based fault injection is the main focus of this chapter.

In order to create a fault-injected circuit with emulation, traditionally the initial design is injected with a single fault and synthesized, mapped, placed and routed on the target FPGA device. Then, a bitstream is generated that can be programmed into the FPGA. Normally, if a fault is observed by a different output value than the correct one, the test set is stored and then the device is reconfigured for another fault. Therefore, for each possible fault, the device needs to be reconfigured. The process is repeated until a test set is created for a large enough set of faults. At this point, we can observe that this process requires significantly long time. In order to reduce the time, a method can be used that creates multiple instantiations of the same design, each one implemented with different faults. This technique doesn't need the time consuming reconfigurations, however, it introduces area overhead and makes it practically impossible to be used for large designs.

7.1.1 Motivation

Fault injection has already been presented in Chapter 5 as a frame-level fault injection system, supported by the SRC, and in Chapter 6 as a validation system of TMR. However, these techniques are either tailored for specific FPGA architectures (frame-based architectures) or they need dedicated tools (FTU2) and multiple FPGAs. In this chapter, a different approach to fault injection is considered, that can target either ASIC designs that used FPGA prototyping as an intermediate step to expedite fault injection, or FPGA designs as a final product. Here, virtual faults are injected in a gate-level design, to provide a virtual layer of potential faults, without being depended on frame-based injections or in specific FPGA architectures.

Physical fault injection is well suited when a prototype of the system is already available, or when the system itself is too large to be modeled and simulated at an acceptable cost. Simulation-based fault injection is very effective in early and detailed analysis of designed systems, since it can be exploited when a prototype is not yet available, and allows the analysis of any possible fault, but requires very high CPU time to simulate the model of the system. An intermediate solution, between physical fault injection and simulation-based fault injection, that provides most of the benefits of both techniques, while avoiding most of their disadvantages, is emulation-based fault injection. It is also an efficient alternative before the radiation testing campaign. By being orders of magnitude faster than simulation and without needing a working prototype and/or specified lab infrastructure (such as a cyclotron), it is able to be applied fast and easily, to evaluate the efficiency of a mitigation technique, or to generate a test-set, for ASIC testing.

However, current fault injection techniques require either massive area resources, as they need to physically inject logic to emulate bit-flips and stuck-at faults, or they apply bit-flips to the reconfiguration bitstreams. Given the fact that the reconfiguration bitstream is protected as IP, a significant amount of engineering is needed as a pre-injection step.

7.1.2 Contribution

In this chapter, a novel fault injection methodology is proposed, that virtually injects faults in a gate-level design, without affecting its area resources, nor its critical path. The proposed technique targets either ASIC designs that are prototyped in FPGAs to expedite fault injection, or designs that use FPGAs as a final product.

Therefore, the following contributions are made:

1. A partially parameterized fault-injection methodology, based on the parameterized configurations tool-flow for commercial FPGAs.

2. A fully parameterized fault injection technique that is tailored for academic FPGAs.

3. An extension to the parameterized configurations tool-flow, to support fault injection.

The contribution can be visualized in high-level in Figure 7.1 and is a first step towards adding a fault injection step in the next generation tool-flow, that was first depicted in Chapter 1 in Figure 1.1(c).

The remainder of this chapter is organized as follows: Section 7.2 provides the background and Section 7.3 the proposed method. Section 7.4 discusses the alterations made to the FPGA flow, in order to support the proposed method. Then, experimental results are presented in Section 7.5 and Section 7.6 concludes this chapter.

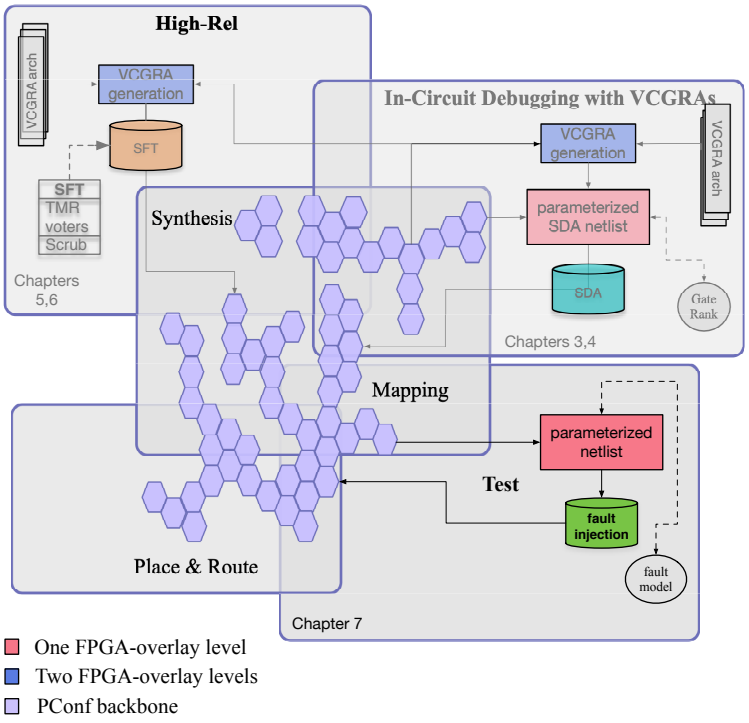


Figure 7.1: Visualization of this chapter's contribution

7.2 Related Work

It is necessary to validate any mitigation methodology applied to a design. Therefore, various techniques have been created, to simulate and/or emulate SEUs and permanent faults.

Fault injection investigation is an efficient method to qualify the design before submitting the part to radiation. Radiation testing is the prime technique to validate mitigation techniques. SEUs can be induced by protons. Linear and cyclotron accelerators can generate protons with adequate energy to simulate proton belt conditions and solar flares. GREs and SEPs on ground are simulated with particle accelerators. Generally, the estimation of the SEU sensitivity using this concept is rather conservative [7]. The machine most commonly used for heavy ion SEU

testing is the cyclotron. Bombarding a chip with radiation can test how susceptible the chip is to radiation but it cannot control where exactly the radiation hits. This is much better controlled by fault injection, hence less tests are needed in a fault injection campaign than a radiation campaign.

In SRAM-based FPGAs, fault injection is defined as a bit flip in specific bits of the configuration bitstream, (or as a stuck-at fault in a specific location, that will be later also a bit flip in the bitstream).

By modelling permanent faults and SEUs in that way, it is possible to evaluate the effects of an upset in sensitive areas of the programmable matrix. Some of these bits are directly related to the user designed logic, and some of them are related to the FPGA architecture and design implementation.

With fault injection, SEUs can be emulated in the configuration memory. As it was mentioned in the previous chapter, various different implementations have been created. The main structure is similar for most projects. Usually, two devices, one of the DUT, that is used as a golden (original) copy of the design and one that undergoes fault injection are needed. The faults are injected at the second (identical) DUT. The basic methodology for fault injection of SEUs is based on bitstream alterations.

A configuration frame is usually read back from the FPGA via one of the configuration interfaces. Then, one or more bits are flipped, and the frame is written back to the device. The outputs of both designs are compared to conclude whether the fault injection led to a failure (a difference in the output). Alternatively, only one DUT can be used, and its response is compared to golden answers during the fault injection campaign.

There are many ways to perform fault injection. First, it is important to have a good model for the type of fault to be injected. For example, a SEU in a memory cell can be modeled as a bit-flip [74]. So, a SEU can be easily injected in a design described in a hardware description language, such as VHDL or Verilog, by performing a XOR operation between the original stored value and a mask, which defines the bit to be flipped.

The most common techniques for FPGA-based fault injection are presented below. They try to avoid the large area or large computation times needed for emulation-based fault injection. There are two basic methodologies, in order to create an FPGA-based fault injection tool.

1. *Reconfiguration based fault emulation*, that modifies directly the configuration bitstream of the design to emulate faults. Fault emulation platforms can be used such as JBits and [96, 138], but they cannot support state-of-the-art FPGAs yet [5]. Direct bitstream manipulation has also been proposed to inject faults in LUTs [115], however not all faults can be covered with this technique. Also, changes to the HDL design have been proposed, but this

again requires recompilation for every new fault injection (or one needs a lot of memory to store all possible bitstreams for every fault).

2. *Circuit instrumentation based fault emulation* techniques modify the structural descriptions of a circuit by adding extra hardware for fault injection. This avoids the time-consuming recompilations for generating a new bitstream every time a new fault needs to be injected [23]. Injecting multiple faults to avoid unnecessary recompilations has also been proposed [19]. However, the size of the new design is directly proportional to the hardware complexity of the injector making the technique again unfeasible for large designs.

Several works already explored the usage of FPGAs for speeding-up fault simulation of permanent single stuck-at faults [23, 67, 130]. However, these methods avoid circuit reconfiguration but introduce significant area overhead (and this limits the size of circuits that can be analyzed), or they have to perform a full reconfiguration for every fault injection. In general, both circuit instrumentation-based and reconfiguration based approaches introduce specialization overhead, which is the extra resources and the extra time needed in order to generate a test set for a specific design.

Below, two European research efforts are explained, under European Space Agency (ESA) funding, that successfully perform fault injection:

FLIPPER

The system comprises of one (Virtex-II) FPGA as a controller that can be connected to a DUT board hosting the FPGA under test [3]. The controller communicates with a software running on a host PC via USB. Here, only one DUT is used, and its response is compared to stored and correct answers. Test vectors can be converted from testbench stimuli and are fed into the DUT after one or more faults have been injected into the bitstream. Compared to radiation testing, results from FLIPPER concluded that it is effective, however its failure rate is underestimated, as it emulates only SEUs in the configuration memory. Moreover, FLIPPER supports only up to the outdated Virtex-2 FPGAs.

FT-UNSHADES2

The second fault injection system developed is FT-UNSHADES2 (FTU2) from the University of Seville. Compared to FLIPPER, its initial aim was the emulation of SEUs and MBUs that originate from SETs and thus manifest in flip-flop cells rather than the emulation of configuration memory upsets. All data management is processed in hardware, leading to very high fault injection rates. The system can also be used to inject faults into the configuration memory. The system is based

on Virtex-V FPGAs, and the circuit under test is duplicated and compared within one FPGA.

During a fault injection campaign, the application is driven to the desired fault injection time, the clock is stopped, the fault is injected into the desired flip-flop(s) of one of the circuits, the clock is restarted, and the outputs of both circuits are compared to detect any mismatch [105]. The system only uses one circuit under test whose output is compared to the correct output [104]. FTU2 has increased usability, compared to other academic tools, due to a simplified design flow and a Web browser-based user interface. FTU2 has been essential to perform fault injection experiments in Chapter 6 of this thesis, to evaluate the proposed TMR schemes.

7.3 Fault Injection with Parameterized Configurations

In this chapter an emulation-based method is proposed for fault injection, that virtually injects faults in a given design.

Instead of manipulating the configuration bitstreams to introduce bitflips, or by adding extra logic, fault injection is performed by introducing parameterized configurations. A virtual multiplexer network is integrated in the design, that injects the right faults at the right place, instead of creating a separate circuit for every fault (or set of faults). This virtual network is implemented via reconfiguration, limiting the amount of extra resources needed. This is visualized in Figure 7.2.

Before a fault is injected in a specific location, it needs to be modelled. Since we take two types of faults under consideration for fault injection, we need two models. The first is the stuck-at fault. A stuck-at fault by definition is a gate output that is always either 0 or 1. This is modelled with a multiplexer, as it is shown in Figure 7.2.

The second fault category that is modelled is a soft error. Soft errors in gate-level designs are demonstrated as bitflips. There are two main ways to create a bitflip and to inject it at a specific location. One is with the SRC and FT-SRC reconfiguration controllers that were presented in the previous chapter, where a specific frame is read, inversed and written back. The other is by adding hardware circuitry in a specific location in the gate-level design. In that way, it is easier to control and test a specific area, if the information of the location of each frame is not known. Therefore, in order to achieve that, specific circuitry is added in the design. This circuitry needs to inverse the output of a specific gate, which actually encapsulates the damage a radiation particle inflicts on the configuration memory, resulting to a bitflip. This is achieved by adding in the output of the target gate a 2:1 multiplexer. Therefore, in a specific location, soft error can be introduced by

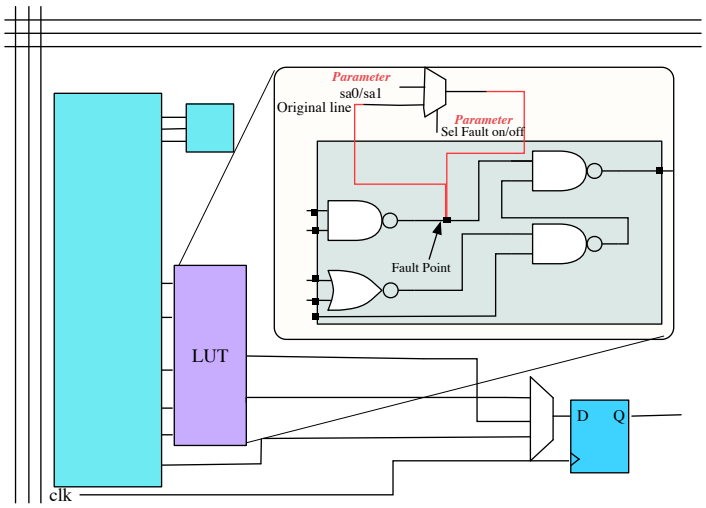


Figure 7.2: Visual representation of a stuck-at fault in a SRAM-based FPGA.

adding the same circuitry, as with the stuck-at fault.

Therefore, with the same circuitry, we can introduce two different types of faults, a stuck-at fault and a soft error. The next step, is to introduce these faults in various locations in the design and virtualize them with two different methods, the partially parameterized and fully parameterized fault injection:

- The first method leverages the LUTs and adds logic in them, according to the parameterized configurations technique. Here, each fault location is identified, and circuitry is added. This is repeated with multiple possible fault locations. At this point, all additional circuitry is parameterized. In that way, all fault locations (multiplexer inputs) are treated as constants, instead of actual LUT inputs, and they are optimized as such. A parameterized configuration is then created, based on these constants. The evaluation of the parameterized configuration creates a new circuit, but with a specific fault, located at a specific location in the gate-level design.

- The second method leverages the routing infrastructure with the TCON tool. Here, the fault injection problem is transformed into a routing problem, as during the fault injection cycle the only aspects of the FPGA that have to be reconfigured with this method are the routing resources and specifically, only the configuration cells for all the multiplexers in the routing switch boxes and the connection boxes.

Both of the proposed methods allow the FPGA to be reconfigured during run-

time very efficiently based only on the evaluation of the Boolean functions in order to reconfigure and create a bitstream. An intermediate bitstream is first produced, that represents all the bits as Boolean functions of parameters that describe the behaviour of faults.

As compared with other fault emulation methods, almost no area increase is achieved. Different faults can be controlled with the use of the dynamic specialization of the FPGA's logic and routing resources. The advantage of this technique is that there is minimal area and runtime overhead during fault injection and it needs no iterative executions of the computationally intensive design re-synthesis, mapping, placement and routing. Finally, with the use of this technique the critical path delay also stays the same as in the initial design, before the fault injection. Therefore, the proposed fault injection method has minimal impact on the original design and requires almost no additional overhead.

7.4 Fault Injection Tool

At a high level, the tool works in two phases: the offline stage (that has two parts) and the online stage, as in the in-circuit debugging method from Part II. The first stage creates a new design that has a selected fault model injected in the initial design. Then, this fault model is mapped to abstract logic and routing resources of the FPGA, in a way that it doesn't occupy extra space. During the online stage, the FPGA is reconfigured multiple times to apply each time a new fault. This dynamic mapping in the abstract logic and routing resources, can create a design with various injected faults.

The parameterized configurations tool has been extended in such way, that it can be used as an automatic tool that injects faults into a design and produces a Boolean bitstream (a bitstream that contains Boolean functions depending on the fault parameters) and is able to reconfigure the FPGA's logic and routing resources. For this purpose, new modules have been created that can support the new design (with the injected faults).

Figure 7.3 shows the proposed tool-flow, that is an extension of the original parameterized configurations tool-flow.

7.4.1 The Fault Injection phase (offline)

According to the PConf methodology, the design has to pass each of the above steps (synthesis, mapping, P&R) only once. This computationally intensive stage is the offline stage of the tool flow. At the end of the offline stage, there is a parameterized bitstream generation, with a virtual multiplexer network of injected faults. The details of how this is achieved are analyzed in the remainder of this section.

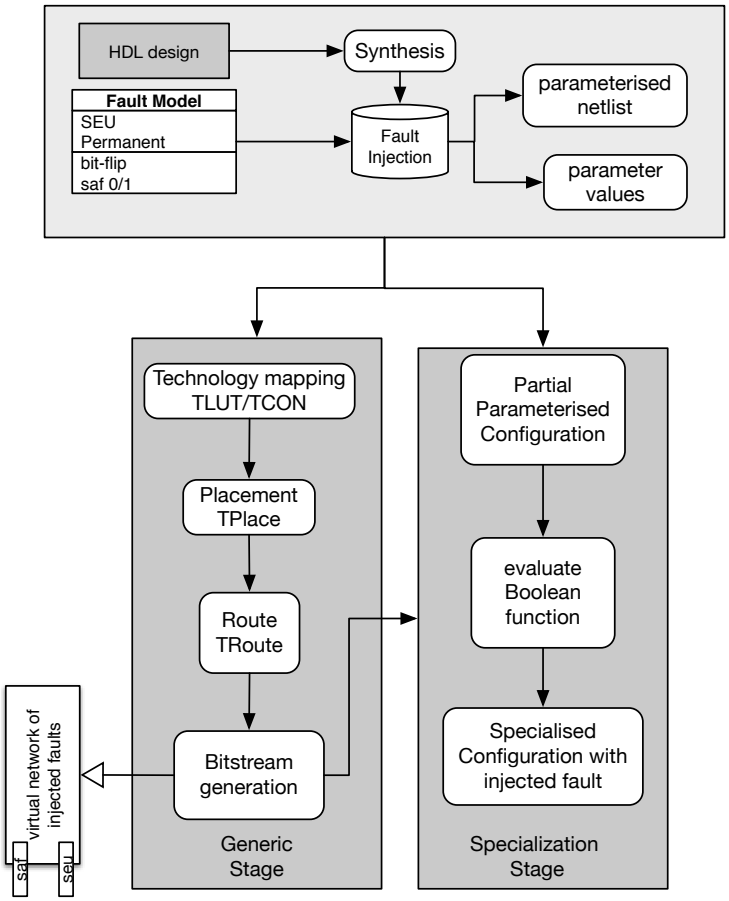


Figure 7.3: The three-stage fault injection tool flow.

During the first stage, the design is introduced. The synthesis step doesn't need to be modified, as the RTL circuit is unaltered. Then, a new step is added in the flow, fault injection. Here, the fault model is defined, and the appropriate hardware is inserted in all possible locations. This is done in the .blif circuit, that is the gate-level representation of a synthesized design with the tools Quartus or ABC. The new design (including all faults as parameters) has to pass all the typical compilation steps, namely synthesis, technology mapping, packing, placement and routing, similar to the tool flow shown in Figure 2.10 [55].

In order to use the parameterized reconfiguration tool flow for our solution, we have transformed it into a three-stage flow. Our tool, during the offline part uses fault injection with a new (instrumentation-based) technique and adds faults to each possible fault location, creating the virtual multiplexer network. So, it starts from a gate-level netlist and modifies it, by virtually adding extra hardware that represents a new fault. However, as every fault is annotated as a parameter, say for example fault *X*, the tool creates a generic bitstream (the PConf) that represents all circuit changes needed for every specific fault in a single bitstream, reducing the online changes needed to reflect a new fault to a simple Boolean function evaluation and a reconfiguration of the FPGA for the obtained new bitstream. This parameterized Boolean bitstream can be rapidly evaluated for a specific fault automatically with the parameterized configurations tool flow. The design is never recompiled, only reconfigured.

The design can originally be described in various HDLs, such as VHDL/Verilog. The synthesis step can be performed by any tool that is able to extract a .blif format. At this point the design is ready for fault injection.

The faults need to be added into the design at every possible fault location in such a way that after the new modifications, the new description remains synthesizable. In order to achieve that, the solution is to add multiplexers into each fault point to introduce a logic one or zero in order to mimic such fault, as shown in Figure 7.2. The tool reads the netlist and locates all the possible fault checkpoints, i.e. locations where faults need to be inserted. The selection signals are annotated as *parameters*, as they will change (but less frequently than the other signals) depending on the type of fault and whether or not the fault should be injected. In our approach, we basically add logic (multiplexers) without adding more LUTs because they are depending only on parameters and can hence be implemented in the reconfiguration resources. So we basically have almost the same size as for the original circuit but now for an extended circuit with all faults injectable.

7.4.2 The Parameterized Configuration creation phase (offline)

In this work, the technique focuses on permanent faults and soft errors, and we assume that the injected fault set is either optimized, or it can be optimized by ex-

isting techniques such as fault dropping and fault collapsing. The main fault model that is used (single stuck-at fault model) is widely applied within the testing community and an easy-to-implement method in order to introduce faulty behaviour in the circuit. However, our methodology is not limited to this model alone.

During technology mapping, the parameterized Boolean network generated by the synthesis step is not directly mapped onto the resource primitives available in the target FPGA architecture, but intermediately on abstract primitives that introduce and allow the reconfigurability of the logic and routing resources.

Next, the Tunable Place & Route tool (TPAR) places and routes the netlist and performs packing, placement and routing with the algorithms TPack, TPlace and TRoute. These algorithms can enable routing of circuits where their routing resources can be reused during the fault emulation and drastically reduce the area usage. At the end of the computationally intensive offline stage the TPaR creates a *PConf*, a *virtual intermediate FPGA configuration* in which the bits are Boolean functions of the parameters.

7.4.3 The Fault Testing phase (online)

Fast test set generation is essential. It is faster to generate random test inputs and select a viable test by emulation, than to effectively search for a test that detects the fault in simulation. So, we use a Linear Feedback Shift Register (LFSR) to generate random input vectors. Both the initial circuit output and the fault-injected one have to be compared (with a XOR gate) for every different input set. If a fault is detected (by a difference in both output vectors) the input vector is stored as a test vector that detects the specific fault at hand. If the required fault coverage is not yet achieved, a new fault is chosen and the FPGA is reconfigured to match the new fault. The stored vectors form the test set. This is depicted in Figure 7.4. The overhead of the reconfiguration for every fault is comparable but always smaller than the overhead we would have in the traditional reconfiguration based fault emulation, as we have less bits to reconfigure than in the original case. However, the big gains of this method lie in the offline part (which in other cases would also be online and very time-consuming or produce a circuit with a very large area).

7.5 Results and Discussion

We have used synthesized gate-level benchmarks for our experiments [66]. The benchmarks selected (ISCAS89) where chosen for their wide acceptance in the testing community and their available and certified gate-level design descriptions (.blif files). Despite the fact that we have used small benchmarks (up to 1% of the FPGA area), we will also discuss the scaling behaviour and show that we can have interesting results also for larger designs. Since our tool follows the flow of

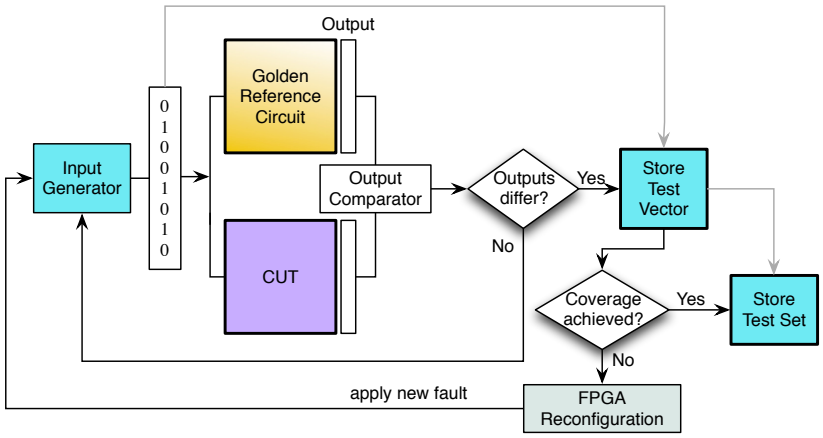


Figure 7.4: The fault injection tool flow for test set generation.

the parameterized configurations, it targets theoretical architectures. However, the results are representative for designs implemented in any LUT-based configurable logic of commercial FPGAs as well. In our experimental setup, our tool adds multiplexers that can introduce stuck-at fault logic in all possible fault locations. Each multiplexer now describes one possible fault at one specific location. The faults can easily model both SEUs and stuck-at faults.

7.5.1 Area Overhead

The proposed approach is compared with the conventional approach, mapped with ABC [97]. In the current approaches such as ABC, we can only include the fault injection with a large area overhead, while we can do the same with almost no area overhead if we use the TCONMap (from Section 2.1). As described in this chapter, the conventional fault modelling adds circuitry (multiplexers) wherever a fault needs to be introduced. So the traditional techniques scale worse than our approach and therefore cannot be applied to larger designs. With our methodology, after the parameterization of the selection bit of multiplexers and the SA0/SA1 fault we have minimal area overhead.

For the experiments, we compared the area of the fault injected circuit to the reference (fault-free) design. Four possible structures are evaluated. The first proposed structure is a normal implementation (without any parameterized reconfiguration), mapped with a conventional mapper. The second structure is an implementation, where only the logic (functional) resources are dynamically reconfigured. The third approach is a structure where we reconfigure both logic and routing resources and finally, a fourth approach, where the stuck-at fault signal was also

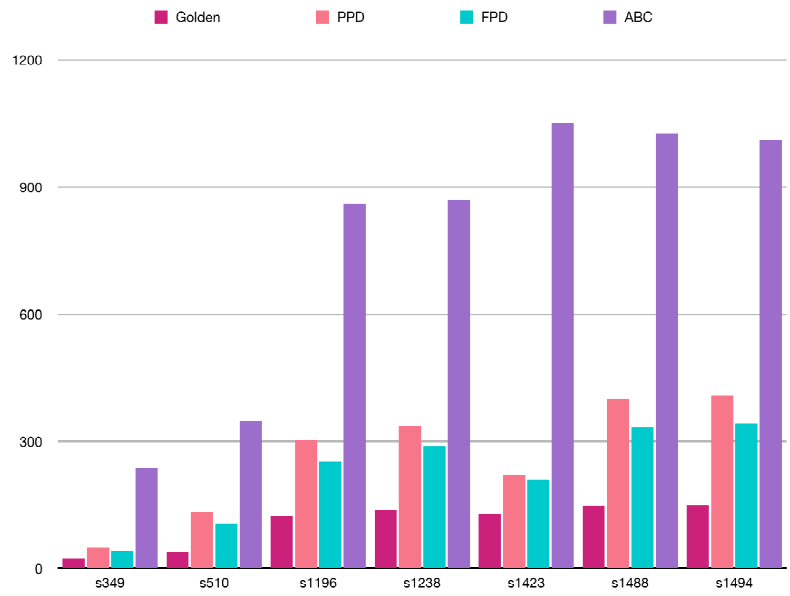


Figure 7.5: Area comparison with the conventional mapper(ABC) and the partially parameterized fault-injection(PPD with TLUTMap) and fully parameterized fault-injection (FPD with TCONMap). The initial area (of the golden circuit) that is needed for the benchmark is compared with different fault injection approaches for the fault injected circuit, FPD,PPD and ABC.

identified as an infrequently varying input, alongside dynamic reconfiguration of logic and routing resources. All these approaches are compared in Table 7.1, with the initial, fault-free design (golden).

In the conventional fault injection approach, the lack of resource parameterization causes a massive area overhead (8 times more area), making this approach unfeasible for large designs (as the extra area needed is proportional to the number of faults). A step forward is mapping of the design with parameterization of its logic resources only (first Proposed). Here, we can see an area reduction compared to VTR's conventional mapper (up to a factor of 4.7). However, there is still a significant area overhead, compared to the golden circuit. The next approach applies parameterized reconfiguration of the FPGA's logic and routing resources, reducing the area even more. Finally, the last approach, builds upon approach 3. Now, the stuck-at fault (or the bitflip) circuitry is fully parameterized, instead of only its selection bit. This results in the minimal area overhead, (needs up to $8\times$ less LUTs than the conventional method, and is only 3% larger than the initial design). Moreover, the problem can scale very well, making the technique feasible

	Golden	Conventional	Proposed(1)	Proposed(2)	Proposed(3)
s349	24	237	50	41	25
s510	39	346	133	106	45
s1196	124	861	302	253	126
s1238	137	869	337	289	144
s1423	129	1049	220	210	130
s1488	148	1027	400	334	153
s1494	149	1010	408	343	153

Table 7.1: Area results in #LUTs: The first column is for the fault-free design, the other columns contain fault injections. For the proposed method, we distinguish mapping with parameterized configuration of (1) logic, (2) logic & routing, (3) fully parameterized circuitry (logic & routing).

for larger designs.

Parameterized configurations are more beneficial than full reconfiguration, regardless the size of the initial design. Even in an almost full FPGA we can apply our technique, as the area overhead is almost zero. In more details, full reconfiguration often needs to change the entire circuit (depending on how incremental synthesis, place and route is done), whereas with parameterized configurations the design does not differ that much from the initial design. In fact, the difference lies only in changing one set of parameters. Therefore, our technique can scale very well. As can also be seen from the experimental results in Figure 7.5 and in Table 7.1, the area overhead remains constant regardless the design, whereas with the conventional methods, the area is increased linearly, with larger benchmarks. This means that the gain in reduction of the overhead in our technique compared to the conventional one is larger for the larger benchmarks.

7.5.2 Critical Path Delay

Our mapper reduces the critical path delay of the added functionality for the faults by reducing the number of lookup tables and the routing infrastructure on the critical path. From Table 7.2, we can observe that the logic depth (inversely related to clock speed) of the design remains constant after the fault injection and the use of our mapper. This is very different from the conventional fault injection technique that introduces MUXs also in the critical path. In fact, the logic depth decreases with a factor of 5 to 8 in our method, compared to conventional fault injection.

7.5.3 Runtime Impact Estimation

The conventional approach as described above has a prohibitive area overhead for large designs. However, as no reconfiguration is needed, it does not have any

#LUTs	Golden	Proposed	Conventional
24	3	3	20
39	3	3	14
124	5	5	25
137	5	5	25
129	10	10	61
148	3	3	18
149	3	3	18

Table 7.2: Depth comparison between the initial circuit and the fault injected version mapped with 6-input LUTs, with the proposed mapper (FPD with TCONMap) and the conventional one (ABC), respectively.

reconfiguration time overhead. In this section we discuss the reconfiguration time overhead of our method.

The runtime overhead depends on the number of times the emulator needs to be reconfigured and on the reconfiguration overhead, the time to evaluate the PConf and to reconfigure the bits that changed. The frequency of reconfiguration depends on the time between two fault injections and hence on the number of tests that need to be performed with the same fault. The number of faults to be injected gives the number of reconfigurations needed. The FPGA needs to be reconfigured when a new fault needs to be activated. Therefore, the time overhead can be expressed as the single specialization time (for specializing the FPGA once) multiplied by the number of times a new fault will be activated. This has to be related to the total time needed. Additionally, the overhead as the time of a single fault injection compared to one complete test run (for one fault) should be considered. This is the relative overhead and this overhead then has to be paid for every new fault (but that does not change the relative overhead if the number of tests per fault is roughly constant).

For large designs, a lot of random tests are needed on average before a fault is detected. As long as the time needed for the evaluation of all these tests is significantly larger than the single specialization time, our technique will have a relatively low time overhead. The single specialization time depends on the evaluation time and the time required for reconfiguration.

The time is needed to evaluate the Boolean functions in the parameterized configuration produced by the offline generic stage of the TCON tool flow. The time needed for one parameterized reconfiguration is highly dependent on the complexity of the Boolean function, and needs maximum 50 μs. Thus, each parameterized configuration can be 3 orders of magnitude faster than a full reconfiguration, which is typically 176 milliseconds for a Xilinx Virtex-5 FPGA. Also, assuming the FPGA design runs at 400 MHz (which is quite fast for an FPGA implementation) and the test vector generation loop in figure 7.4 can be executed in 4 clock

ticks (which requires a fully pipelined design), the 50 μs overhead corresponds with the time needed to perform 5000 tests on the FPGA fabric. This is a reasonable number for large designs. So for larger designs, the overhead becomes smaller relative to the test set generation time.

Besides the methods that have area overhead but no reconfiguration overhead, there is also a common technique that does not have an area overhead but requires execution of the entire tool flow online [30]. In this case, bitstream manipulations are performed on the fly, which takes a lot of time. Since they have to reconfigure the entire chip for every different fault, they would need 176 ms + 169.738 ms for each bitstream manipulation. Comparing this number with our online tool flow that takes 176 ms + 50 μs we are about twice as fast. However, the online time needed is only 50 μs in our case and thus orders of magnitude faster. This shows that the proposed method can introduce minimal time overhead as well. Also, our tool flow automatically produces the bitstream from simple additions to the HDL code, without the need for low-level bitstream manipulations to be done by the designer (as in the other approach).

7.6 Conclusion

Emulation-based fault injection provides numerous advantages over fault simulation techniques, but, up to now, at a considerable cost. In this chapter a novel technique is introduced, where faults can be generated efficiently (with minimal area overhead and minimal time overhead at the same time) and thus enhance the time consuming fault injection procedure during the conventional design flow. The experimental results have demonstrated the feasibility of the approach, as it obtains an area and delay reduction of a factor of 8, and operates within 3 orders of magnitude faster, compared to conventional methods.

Part IV

Conclusion

8

Conclusion and Future work

FPGAs have been increasingly used as prototyping platforms and in various applications, including safety-critical ones. Due to their bit-level parallelism that is possible in HW, they can achieve significantly higher throughput than CPUs, while being more cost-efficient than ASICs. This has made FPGAs ideal prototyping and verification platforms and it has allowed designers to increase their verification coverage by several orders of magnitude. FPGAs are excellent candidates to create the next generation of design and verification solutions, including debugging, fault injection and fault tolerance. However, the observability of their values on internal signals is limited and excessive area is used. If reconfiguration is used, the long recompilation/reconfiguration times often prohibit designers to offer efficient FPGA-based hardware verification methods. Additionally, the FPGA configuration memory is prone to soft errors, in radiation environments.

In order to conclude this dissertation I recapitulate the goal of the thesis:

The primary goal of this dissertation is the creation of novel, efficient hardware verification methods that, alongside custom FPGA components (reconfiguration controllers, VCGRAs), are providing on-silicon debugging and fault mitigation, in commercial-off-the-self FPGAs. Different improvements to the FPGA conventional tool flows are proposed, by adding a pre-synthesis step (for debugging and fault mitigation) and a post-synthesis step (for fault injection). After adapting custom FPGA structures with integrated verification schemes (TMR and scrubbing), it is investigated how FPGA overlays and parameterized configurations can assist

in reducing the overheads of verification on FPGAs. Reducing the overheads is important to allow the proposed techniques to be useful in commercial designs.

8.1 Conclusions

In this section, I present the overall conclusions of the research focusing on how the thesis' goals are achieved for a given set of problems.

The results throughout the thesis demonstrate that an FPGA overlay called VCGRA can be used to provide both debugging and fault mitigation with very low area and performance overheads. Additionally, parameterized configurations can be used alongside all the verification methods that were studied in this thesis, to reduce the area and reconfiguration overhead, for two different architectures of dynamically reconfigurable FPGAs. In more detail:

8.1.1 In-Circuit Debugging

In this dissertation I presented two different debugging schemes, that can be tailored for both commercial and academic FPGA architectures.

In chapter 3, an integrated debugging method is introduced, that offers extended internal observability. Thanks to integration with the parameterized configurations tool-set, the designer can rapidly trace different sets of signals. The proposed tool offers low area overhead for the debugging infrastructure and less debugging cycles, with the use of signal ranking techniques. Hence, in Chapter 3, in-circuit debugging infrastructure can be integrated and optimized, alongside the original design, with an area penalty of 10% - 30%. Here, the penalty is the overhead for adding verification to an otherwise unverified design. That penalty, however, is much smaller than with conventional methods.

To improve the applicability of the debugging technique, the techniques from Chapter 3 have been extended into CGRA architectures. Conventional CGRAs suffer from high reconfiguration overhead and have no parameterization options. Hence, in Chapter 4, a low-power / low-overhead superimposed debugging architecture for heterogeneous computing platforms is described. First, it is demonstrated how to efficiently implement a CGRA in an FPGA, and then how to extend it with an on-silicon debugging infrastructure, called SDA. The SDA is used to integrate a debugging infrastructure to efficiently debug FPGA overlays, such as VCGRAs, with a small area penalty. In order to create this novel architecture, a supporting flow and a new two-level virtual architecture have been constructed. The custom made tools introduce resource sharing on the two-level virtual architecture, reducing the actual FPGA resources. Additionally, by parameterizing the VCGRA, its resources can be further reduced. Hence, in Chapter 4, a novel method of low-overhead debugging mechanism is introduced, that can be integrated in vir-

tual FPGA architectures, without any power or reconfiguration overhead and with a relatively small area penalty (up to 30%), that is still $5\times$ smaller than in commercial tools.

8.1.2 Fault Tolerance

COTS SRAM-based FPGAs have been extensively used in high-reliability environments and in safety-critical applications. So far, fault mitigation for COTS FPGAs is provided at a high cost of area and time. In Chapter 5, a fault-tolerant scheme is presented, for future high-reliability applications in radiation environments that include multi-level reconfiguration scrubbing and fault-tolerant reconfiguration controllers. First, two microscrubbing techniques (MS and DMS) are proposed. The MS can periodically reload the reconfiguration bits from a golden instance, resulting in a significant decrease in time. DMS improves this result by achieving $10\times$ decrease in time compared to conventional scrubbing and $3\times$ compared to MS, as in DMS only the correct frames that are affected by the SEU are written and not all the frames.

Next, two reconfiguration controllers that support the custom scrubbing techniques have also been designed (SRC and its fault-tolerant version, FT-SRC). This work provides fast scrubbing with less FPGA resources and it aims at building an integrated fault mitigation scheme that enhances the reliability of COTS FPGAs. DMS and MS can be combined with the proposed reconfiguration controllers that are more efficient than the conventional Xilinx controllers in terms of area, energy and runtime.

In Chapter 6, a spatial redundancy scheme that provides fault-tolerance in radiation environments, for overlay architectures has been introduced. The proposed technique applies TMR on an FPGA overlay and achieves an overall reduction up to 30% of the total TMR resources compared to full TMR with vendor tools. The mitigation scheme and the FPGA overlay are co-designed and co-optimized in such a way that they result in a fault-tolerant architecture with less overhead in terms of area, time and power. This work aims at building an integrated fault mitigation scheme that enhances the reliability of COTS FPGAs. The fault tolerant techniques have been verified by a fault injection campaign that has been designed specifically for this dissertation. During the fault injection campaign, no wrong outputs were recorded as a result from SEUs and the critical bits of the designs under test have been significantly reduced, making the proposed method equally robust to commercial mitigation tools.

8.1.3 Fault Injection

Emulation-based fault injection provides numerous advantages over fault simulation techniques, but, in general, at a considerable cost. In Chapter 7, a new fault

injection technique has been proposed, where faults can be generated efficiently (with minimal area overhead and minimal time overhead) , thus enhancing the time consuming fault injection procedure during the conventional design flow. The experimental results have demonstrated the feasibility of the approach, as it obtains a significant area reduction, and operates 3 orders of magnitude faster, compared to conventional methods. The proposed technique can apply fault injection either for test set generation, or for detection of soft errors, with the same fault injection model. Hence, a low-overhead fault injection approach is proposed, where a virtual multiplexer network applies fault injection in a given design, with minimal impact on the DUT.

8.2 Future Work

I conclude this dissertation with a possible scope for future research on debugging, reliability and FPGA overlays.

The main objectives for future research are the development of a design & verification toolflow that contains innovative scalable methods to mitigate, predict and debug a large variety of faults (including MBUs) in complex digital integrated systems for various FPGAs. A possible visualization is depicted in Figure 8.1, that is consistent with the tool-flow that has been used throughout this thesis.

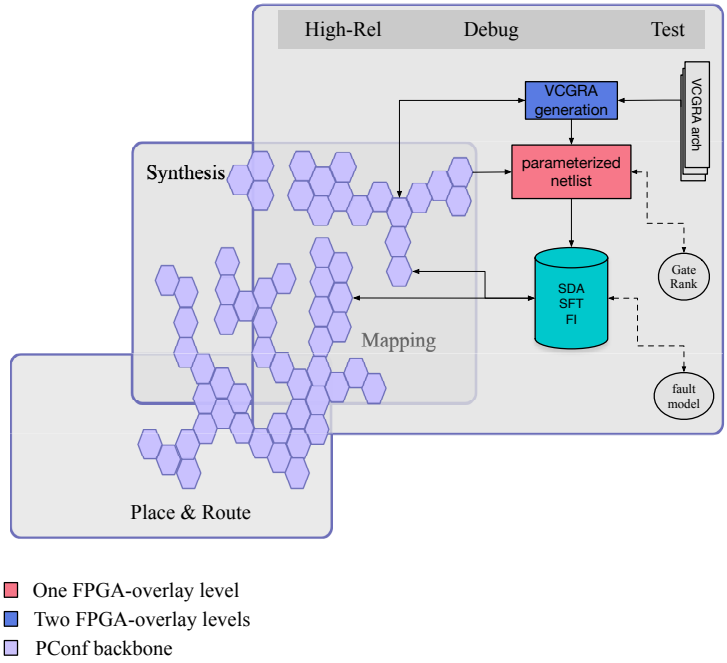


Figure 8.1: Visualization of the future tool-flow

A future direction can be to introduce a framework that adds mitigation techniques to the design that will also contain a fault prediction mechanism able to assist in predicting faults before they occur and to validate the design during runtime. MBUs should also be addressed in the next generation of high performance digital systems that utilize commercial FPGAs, as due to the constant technology shrinking, the probability of MBUs is increased.

The proposed algorithms from Chapters 5 and 6 could be used as a basis and

extended to cover a wider variety of radiation effects, such as MBUs, SETs and SELs. Alongside the basic mitigation technique, the FT-SRC custom reconfiguration method that has been introduced in Chapter 5 could be extended with a high performance port (HP0) of the Zynq-SoC. Since the HP ports can be accessed only by a master from the PL region, a possibility will be to use a DMA controller that acts as a master to the HP0. The data transfer will then occur through the HP0 port of the Zynq-SoC, thus establishing a very high speed data transfer that will contribute to faster scrubbing and fault injection.

In the future, the proposed methods of this dissertation could be valorized through the development of custom CAD tools to implement on-the-fly radiation hardened digital designs, as currently, the vendor tools do not offer techniques that mitigate multiple radiation effects without compromising the performance (area, speed, power) of the system. Finally, it would be interesting as a future direction to demonstrate a prototype chip that employs these techniques to realize a radiation hardened digital design. Therefore, I pinpoint the future work in various different objectives:

1. Methods to apply in-circuit debugging and fault tolerance for complex digital designs. An FPGA overlay can be used to automatically generate and instantiate various verification schemes in a target design for multiple FPGA architectures.

2. It will be interesting to create a failure prediction model that will analyze data acquired from fault injection or radiation testing, to determine what failure may occur in the future. It can be used to predict the possibility of an error, and to calculate the error rate.

3. An Intermediate Representation language could be also designed, that will be able to translate design requirements, mitigation circuitry, debugging circuitry, fault injection methods and error locations for various FPGA architectures. This language could then be translated in different hardware description languages (VHDL, Verilog) or any other languages that the system is described (in C++, python, etc). It will be interesting to first integrate this language in the initial design, during development, but making it inactive, until mitigation is needed. Then, at the moment that redundancy is needed, or a new technology is needed, the new requirements could be applied immediately.

4. A framework that operates alongside the typical FPGA flow and rapidly integrates the above-mentioned techniques in the conventional FPGA flow. The FPGA overlays will reduce the runtime of the CAD tools, and will decrease the resource overhead. In the first level of the framework the system will be

trained to analyze complex designs. The tool will be able to adapt the design during runtime, based on the time series predictions of the single event effects. The second part of the framework will apply selective redundancy and will integrate debugging circuitry, in the areas that it is more possible for an upset or a bug to occur. Then, the tool will be extended with a preprocessing step, that will instantiate mitigation circuitry, to support future FPGA architectures and different FPGA families. Hence, according to the target FPGA family and the design language, it will generate the mitigation scheme, the debugging method, create a new overlay and install it in the target FPGA automatically. This step, will decrease the implementation time dramatically, as no reimplementation nor redesign will be needed, to adapt to a new FPGA architecture.

The outcome of these future directions could result in the ability to mitigate and debug a wide variety of faults, and to develop fault tolerant designs that can be reused with future state-of-the-art architectures, without the need to be redesigned. Then, novel techniques can be designed that will automatically integrate various debugging techniques, and prevent failures due to radiation effects, by finding the most effective reliability techniques, tailored for SRAM-based and flash-based FPGAs. In more detail, possible future directions can be summarized in:

- Assessing hardware-based techniques that use spatial redundancy (TMR, DMR, etc) to efficiently prevent a wide variety of faults.
- Creating an intermediate representation language that translates the fault-tolerance, debugging and fault prediction models into a specified FPGA architecture and design language.
- Reacting to faults, immediately upon their detection via a CAD tool that enables just-in-time scrubbing, real-time on-chip monitoring and readback techniques via novel reconfiguration methods and controllers.

8.2.1 Overlay-based fault tolerance and debug

An MBU detection technique can be designed, for future high-reliability applications in radiation environments that include overlay architectures. In more detail, an in-circuit reliability technique with fault mitigation of multiple bit upsets for an FPGA overlay can be created. The overlay could be extended from Chapter 6, instantiated alongside the design and triplicate the logic. Moreover, scrubbing methods will also be integrated to the target design, alongside the spatial redundancy.

A fault-tolerant custom reconfiguration controller that implements the proposed functionality, can also be integrated, as it will detect erroneous configuration

frames during configuration scrubbing. Then, it can also apply various injection models and debugging infrastructure. The fault-tolerant reconfiguration controller that was proposed in Chapter 5 can be extended in order to assist and control the novel verification schemes.

The output of this future work can be a user IP that is able to install the overlay in the FPGA and adapt it to apply different verification methods.

Error detection can be done at much lower cost than error correction. Thus, a fault detection technique can also be created to detect erroneous configuration frames before configuration scrubbing, by predicting the time and location of the radiation effect, by using neural networks, or one of the Mean Time to Failure (MTTF) models. In order to efficiently create a model for fault prediction, single event upset data can be collected with radiation testing, or fault injection. Then, the data can be organized to facilitate the model, that will be designed based on the availability of the training data.

8.2.2 Intermediate Representation language for multiple architectures

In the future, the proposed debugging and mitigation techniques from this dissertation could be further investigated to be applied in other FPGA architectures. The circuitry generated from the tools described above could be integrated at the target FPGA. Then, an Intermediate Representation (IR) language could be created that during runtime can be translated in specific code description, based on the inputs of the framework.

A language could be created, with a simple set of description rules and registers that describe the design-level fault mitigation and debugging scheme, such as the mitigation method, a signal set to be debugged, the scrubbing technique, the scrubbing frequency, etc. An example of this translation is depicted in Fig. 8.2. We believe it would be beneficial to design the IR in such a way, that can allow to adapt the validation schemes automatically, without the need to redesign nor reimplement the circuit. In that way, the radiation tolerance can be adapted, and the debugging scheme changed, only by evaluating the IR and adapting the overlay.

In order to support the IR language, a compiler could be designed that uses the standard UNIX yacc parser generator and lex scanner generator tools. First it can create a state machine and generate expressions that represent the values of each identifier in each state. If the end of the input is reached without an error, the second pass can perform some simple optimizations on the circuit, generate the FPGA overlay and output the netlist.

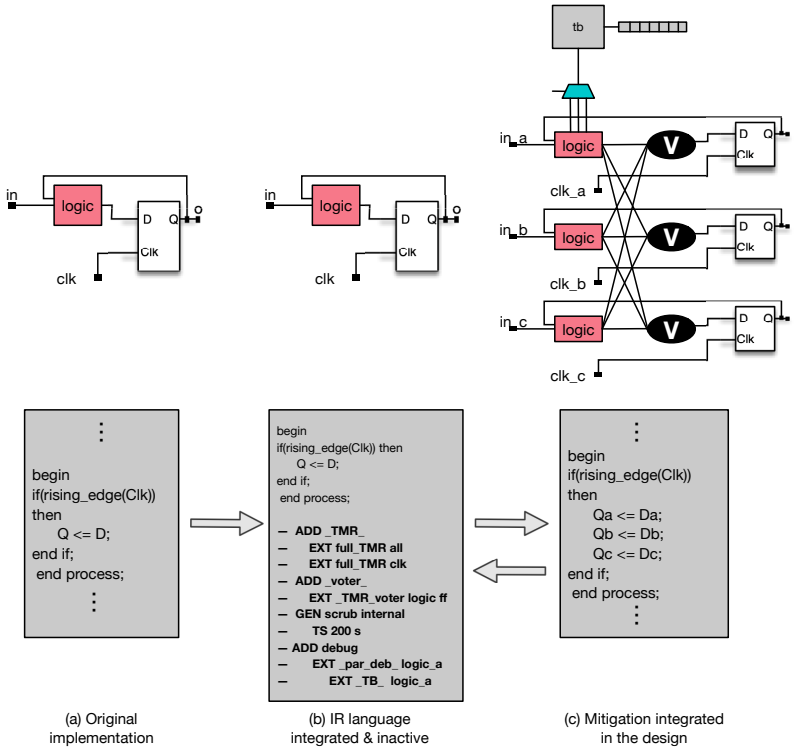


Figure 8.2: Scheme that shows in (a) an example design, in (b) the Intermediate Representation language having no impact in the design and (c) translation of the IR and mitigation of the design.

8.2.3 CAD Framework

Another possibility is to extend the above-mentioned methods in a framework that could be designed following the principles of modularity and technology independence. In that way, the framework will be expandable. The main components of the platform could be the inputs provided by the user and the final generated artifacts are presented in Fig. 8.3. We believe that this platform will allow the user to specify the application logic in various languages, such as C/C++, VHDL, python. It is also possible to extend the supported languages by modifying and replacing the modules that analyze the user application.

Along with the code, the platform will also require a HW description that defines the system template, the proposed mitigation method, the debugging model, the target architecture, and a description of the overlay that will be instantiated. The flow could then be organized in three main components: the frontend, the overlay and mitigation optimization component and the backend. All the components will operate on the selected node architectural templates, which specify how the reconfigurable nodes of the target system should be mitigated. In particular an architectural template could specify both the mitigation model, the debugging scheme within the reconfigurable hardware, the levels of overlay, the maximum number of observed signals and the specific technology used to implement the model.

The frontend can analyze the users code and will provide suggestions on the architectural templates, debugging circuitry and mitigation schemes that best fit the application, allowing also the user to consider multiple mitigation schemes for the subsequent phases. It will also give the possibility to add or remove the debugging infrastructure. The frontend could return a mitigation technique to be instantiated on the overlay, within the FPGA target architecture. The HW description of the system (overlay and mitigation) could then be fed to the backend component, which, after some implementation steps, can produce a final fault-tolerant bitstream for FPGA configuration and the application that the user can use to execute the mitigated version of his/her application. In the future, we believe that such framework can be essential in providing a mean to integrate multiple mitigation and debugging methods.

Ideally, these techniques could be complimentary to the Vivado tool flow that already adds incrementally the FPGA overlay in Chapter 4 and 5. The FPGA could then be updated during run-time, alongside its mitigation scheme. The reconfiguration controller could also be integrated automatically.

Multiple fault tolerant mechanisms and the MTTF-like predictions for various architectures could be used as either inputs or templates at the framework. Ideally, the IR language will be able to be translated in specific designs that support mitigation and debugging overlays for various FPGA architectures, from different vendors.

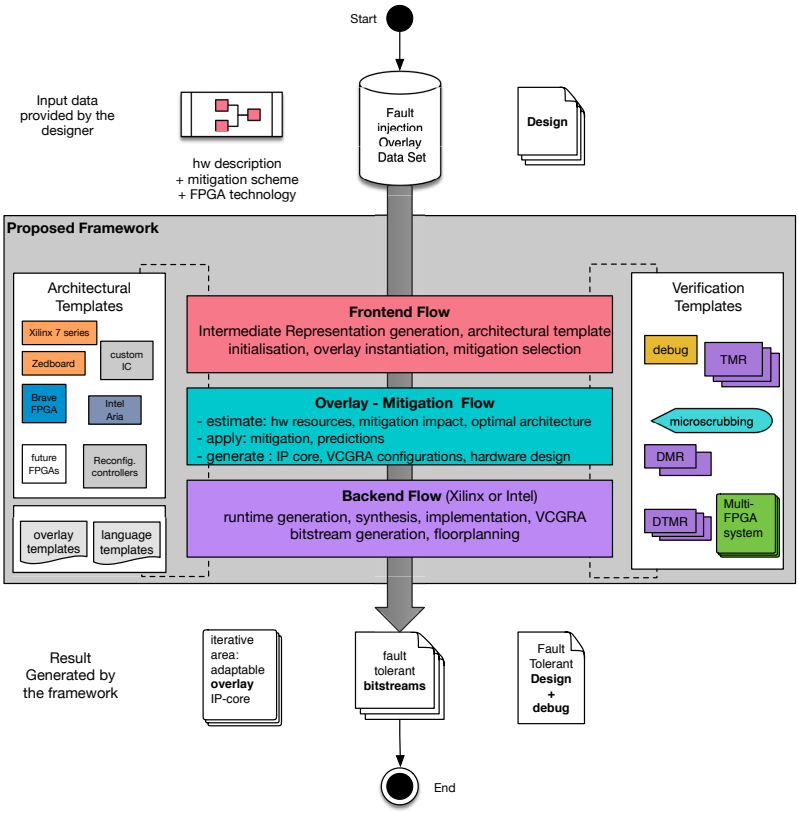


Figure 8.3: High level overview of the framework, that includes all the possible directions that can be targeted for future work

At a later stage, the tools could be extended in a homogeneous design and verification tool-flow, similar to Figure 1.1(c), presented in Chapter 1. The tool could start from a target design and create all the above-mentioned techniques, using as a back-bone the PConf tool, that by then could be already integrated in the vendor tools.

8.2.4 Integration With Vendor Tools

A very interesting future direction will be if the vendors (Xilinx, Intel, etc) would include the methods proposed in this thesis, in their own tools. In that way, the tools of this thesis will be beneficial the most, by leveraging the internal FPGA structure (that is currently protected). So, in order to benefit from this thesis the most, the proposed methods would have to be integrated in the vendor tools itself so the tool has access to the internal structure and can use it to the method's benefit. According to industry, there is a need to extend the debugging methodology for the last generation of Xilinx FPGAs. Then, it will be possible to integrate these techniques in FPGA prototyping platforms. Since the results of this dissertation have demonstrated that it is beneficial to apply in-circuit debugging by leveraging the FPGA's reconfiguration resources, it is anticipated that these techniques will introduce large area savings in FPGA prototyping platforms. Hence, for future work, the techniques should first be extended to support the Xilinx UltraScale FPGAs. Additionally, an integration with the RapidWright tool from Xilinx, will be a first step to leverage the reconfiguration resources of Xilinx FPGAs. This will be the first step towards integrating the tools in the conventional toolflow of commercial FPGAs.

8.2.5 Beyond FPGAs

For future work, the techniques could also be generalized into a technique for mapping ICs automatically via the framework, by adding the appropriate template in the framework. Optimizations can be designed to support ASIC designs. Another possibility for future work, could be to extend the IR model to cover ASICs. The outcome of this could be an ASIC flow that creates a tolerant version of the target design for SoC. First, a custom recovery circuit can be developed that implements the techniques from this thesis. Fault injection data can be applied to classify the probability of errors and apply targeted mitigation. This could possibly eliminate the performance overhead requirements of the new design and it might reduce respins. A methodology can then be created, to automatically add debugging, fault tolerance and implement a design.

Appendices

A

Radiation Effects

From space to ground and below

The Part III of this dissertation, focuses on the impact of space radiation that imposes on microelectronics, and especially in SRAM-based FPGAs. Therefore, this appendix includes supplementary material, to provide background information on the source of radiation effects. Hence, in this chapter *I analyze the energetic particle radiation, its source, and its monitoring and mitigation techniques*. The three main sources contributing to a radiation environment are the galactic and extra-galactic cosmic rays (GCR), solar energetic particles (SEPs), and trapped particles. These sources are the main source for causing a radiation environment. The radiation environment is divided in the space environment, the aerospace, the ground (sea) level and the underground level. It is also divided in natural environment (space radiation) and human-made radiation environment, like the nuclear plants and particle accelerators.

A.1 The space radiation environment

The space radiation environment is defined as an environment that is complex and hostile, with risks to space-mission survival coming from a variety of sources, including meteoroids, space debris, residual atmosphere, energetic particle radiation, and lower energy plasma. Here, I focus on particle radiation and its effects on microelectronics.

A.1.1 Galactic cosmic rays

Galactic Cosmic Rays (GCR) originate from outside of the Solar System and consist of charged particles travelling near the speed of light. Their average energy is $< E > \ 1GeV/u$, while the most energetic particles have an energy reaching $< E > \ 1021eV$. The energy of cosmic rays is believed to derive from acceleration by interaction with moving magnetic fields, such as in moving shocks originating from supernovae [34].

Primary cosmic rays in outer space manifest themselves as a continuous flux of about 4 particles $/(cm^2 s)$ on average, with intensity in anti-correlation with solar activity. The majority of the GCRs is made of atomic nuclei, and about 1% is made of electrons. Among the nuclei, protons constitute about 90%, alpha particles 9%, and the remaining 1% of the particles consists of heavier ions. Due to their high atomic number and high energy, this numerically small proportion of very penetrating particles contribute significantly to SEEs in microelectronics (and to the predicted biological dose in astronauts in long duration interplanetary human spaceflight).

A.1.2 Trapped radiation

The particles able to penetrate the geo-magnetic (GEO) field and reach the Earth are shielded by the atmosphere. Collisions of the primary cosmic rays in the upper atmosphere produce *secondary* particles. Part of the secondary particles, generated by GCR interactions, reach ground level. This flux of secondary particles, causes in particular SEEs and it has an impact on the performance of microelectronics (including SRAM-based FPGAs).

In the near-Earth region the magnetic field keeps electrons and protons populations trapped in a gyration, bounce and drift movement [135]. This is visualised in Figure A.1. The energetic electrons and ions are magnetically trapped in the Van Allen radiation belts (which extend from 100km to 65000km above the surface of the Earth) and consist mainly of electrons up to a few MeV and protons of up to several hundred MeV energy.

The Earths magnetic field is not symmetrical. This leads to local distortions, such as the South Atlantic anomaly (SAA), that exposes passing spacecraft passing, from this area, to an increased level of radiation[1]. This is visualised in Figure At a low altitude, variations of the proton fluxes are observed, anti-correlated with solar cycle changes in thermosphere density (neutral atmosphere). Effects to microelectronics in this altitudes include Displacement Damage (DD), SEEs, and nuclear activation. The SAA is a significant source of effects for all polar orbits,

[1] SAA is a manifestation of the lower inner belt proton altitude in the South Atlantic region, due to the Earth magnetic field being tilted by 11 degrees with respect to the Earth rotation axis and offset by 500km towards the north Pacific

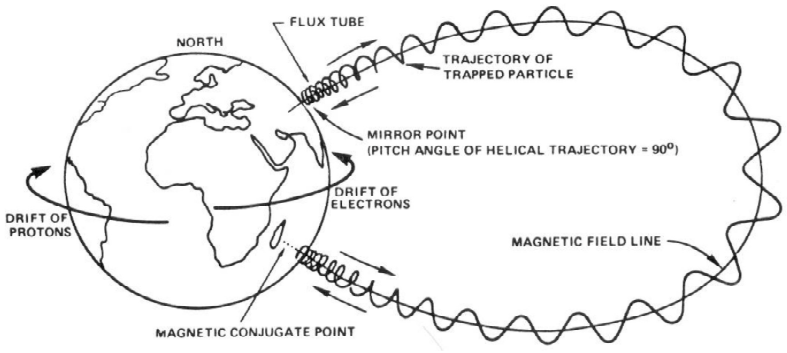


Figure A.1: A descriptive drawing of the three types of motion of particles trapped in the Earth's magnetic field [135].

and it is the only significant radiation exposure on many low altitude orbits, especially with low inclination or heavy shielding (e.g. for the International Space Station - ISS).

The outer belt is dominated by a dynamic population of energetic electrons, with (kinetic) energy $E < 10MeV$. Trapped electrons dominate the geostationary orbit (GEO) and the medium Earth orbit (MEO) environment. Electron effects traditionally include total dose and electro-static discharges (ESD) from surface and internal charging, and component susceptibility to electron-induced SEEs [76].

A.1.3 Solar energetic particles during solar flares

SEPs like solar flares, are generating particles such as protons, heavier ions, electrons, neutrons, gamma rays, X-rays. They are emitted from the sun, and in some cases are accelerated (either in the vicinity of the sun, or by shocks in interplanetary space). The energy spectrum of the SEPs is highly variable, and is much softer than that of GCRs.

Charged SEPs are normally shielded by the Earth's magnetic field, but can get closer to the Earth surface at higher latitudes (closer to the polar regions). If the energy spectrum of the SEPs is high enough, from their interactions in the upper layers of the atmosphere, secondary particles are generated that can reach aircraft altitude, or even ground level. Cumulative fluence of heavy ions of solar origin have been shown to exceed GCRs during solar maximum, with an impact on SEEs in microelectronics [151].

Therefore, GCRs, trapped electrons and protons, and SEPs have been introduced as the main sources of particle radiation in space. Their impact on electronic

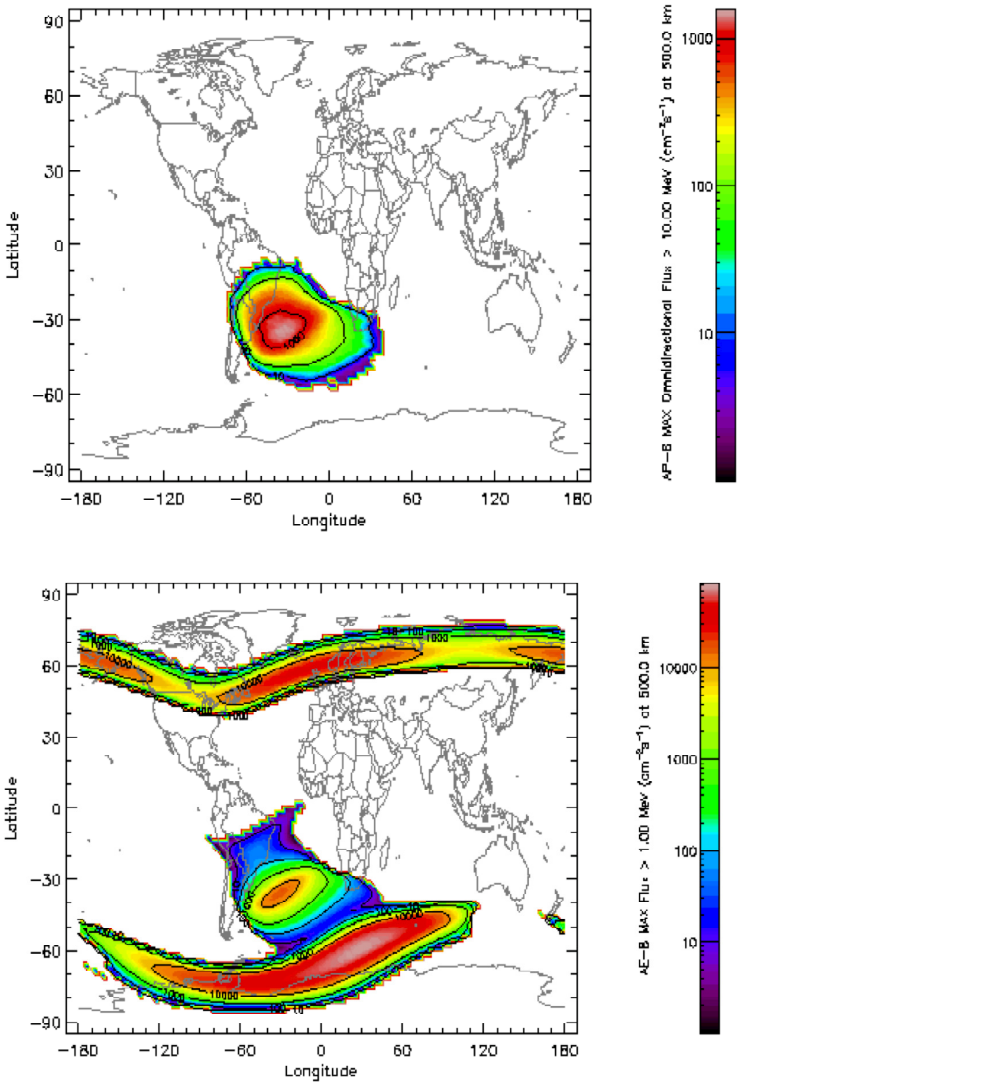


Figure A.2: World map at 500 km altitude of the trapped proton ($> 10MeV$) and trapped electron ($> 1MeV$) distributions, respectively. The SAA shows up clearly in both maps [134].

systems on board spacecraft is quantified in terms of TID, non-ionising dose and SEEs. However, the variability of the intensity and energy spectrum of the environment is still not fully captured in the prediction models. A general trend is that the high exposures that are seen at very thin shielding, can decrease behind an aluminum structure of just a few mm (as the low-energy portion of the particle spectra is absorbed). Then, the dose levels can decrease much more with increasing shielding (either due to the hard proton spectra, or to the penetrating secondary Bremsstrahlung gammas in the electron environment).

A.2 Aircraft Radiation Environments

The different sources of cosmic radiation can affect spacecrafts for interplanetary missions and in the near-Earth environment. In addition, the interactions of GCRs and SEPs with the atmosphere lead to the generation of a variety of secondary particles which affect the lower reaches of the atmosphere. These particles affect microelectronics at aircraft altitudes (and on the ground) through SEEs [95]. The interactions of energetic ($10sMeV/nuc$ to many $100sGeV/nuc$) protons and heavier ions include nuclear spallation[2] potentially with the production of many other nucleons and other nuclear fragments, and if sufficient energy is present, more rare particles such as pions (at incident energies $> 290MeV$), and other mesons appear. The ambient dose equivalent rate as a function of altitude, and the contributions made by these particle species to the dose is shown in Figure A.3. These products can themselves undergo further interactions, that can result in a flux of lower-energy particles from a single cosmic-ray or SEP, striking the higher reaches of the atmosphere. Furthermore, many of the particles produced can lead to γ - ray production from the de-exciting nuclei, and γ-rays, electrons and positrons from the decaying mesons. Depending upon the application and the component used in the mission, the effects of these particles must also be considered.

The intensity of the atmospheric environment depends on the altitude and (magnetic) latitude, with the Earth geo-magnetic shielding (that is less effective closer to the poles). Intensity, composition and spectrum of the particle environment are also dependent on the depth in the atmosphere. Here, neutrons are the major contributor to the SEE rates, especially at lower altitudes. The contributions to SEEs from heavier ions (both residual GCRs and nuclear fragments) account for $<$15% of the event rates (up to 15 km).

[2] spallation is the process in which a heavy nucleus emits a large number of nucleons as a result of being hit by a high-energy particle, thus greatly reducing its atomic weight.

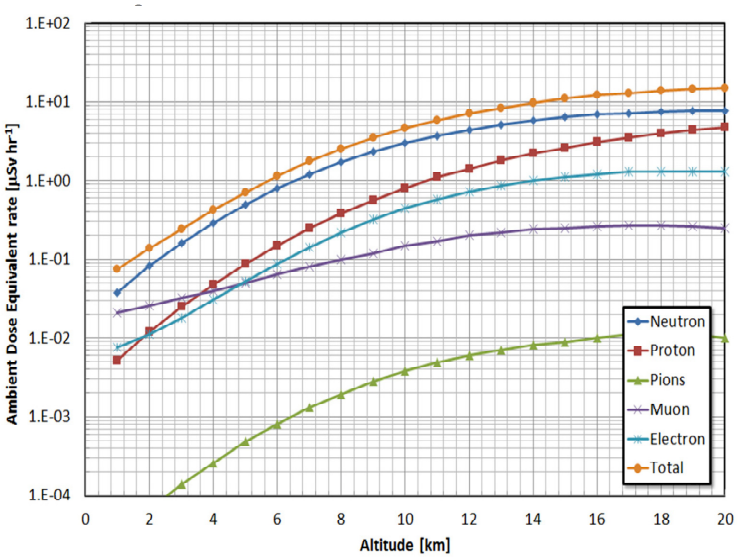


Figure A.3: The ambient dose equivalent rate as a function of altitude, and the contributions made by different particle species to the dose [95, 134].

Altitude km	Neutron [events/year]	Proton [events/year]	Heavier Ions [events/year]	Total [events/year]
12	25	5	5	35
20	43	20	12	75
99	11	26	112	149

Table A.1: Predicted ESA SEU Monitor upset rates due to GCR-induced atmospheric radiation, including contributions from neutrons, protons and heavier ions. The predictions are based on the MAIRE model for location $45^{o}N, 74^{o}W$, which has a vertical cut-off rigidity of $1GV$, and solar minimum conditions.

A.3 Radiation environment at ground level

The radiation environment at ground level consists mainly of muons, neutrons, electrons and photons and it can affect modern electronic components, circuits and systems. It has two main origins: the cosmic ray induced particles showers that propagate from the top atmosphere to ground and radioactive species in the earth.

Human-made activities produce a huge quantity of radiation in Nuclear Fission Power Plants experimental reactors [140]. Additionally, medical applications use more and more accelerators with possible side leakage of radiation. In these facilities the radiation environment can be compared to natural radiation level. In addition, microelectronics may suffer from a radiation effects linked to residual ra-

dioactivity located in the active device materials or inside the package close to the active device. Additionally, high energetic ions (protons, alphas and heavy ions) from galactic or extragalactic origin can reach the upper atmosphere and interact with the atmospheric species (oxygen, nitrogen, argon, water vapor) [6, 24].

The flux of particles is much lower in ground level than at aircraft altitudes and in space, but the huge number of COTS electronic devices that may be sensitive to SEEs has a direct impact on microelectronics. In that case, neutrons are giving the most important contribution to SEEs on microelectronics. But sensitivity of devices varies continuously with the technology node. Thus, the contribution of other particles may become important. Radiation in human-made radiation environments may reach tens of kGy/h and the total cumulative dose may reach tens of MGy.

A.4 Radiation environment below ground level

The radiation fields that occur in close vicinity of a high-energy accelerator such as the Large Hadron Collider (LHC) are extremely complex in nature, with intensities and composition which strongly depend on the location and operation mode. The LHC [33] at CERN is the worlds largest and most powerful particle accelerator. The CERN accelerator complex, from the injector chain up to the LHC is visualized in Figure A.4. Its main purpose is to provide large enough energies and intensities to generate rare events, that will allow us to answer fundamental questions and improve our understanding of the high-energy physical universe laws. The LHC consists of a 27km ring of superconducting magnets in which two high-energy ($7TeV$) beams travel in opposite directions and separate ultra-high vacuum beam pipes before they are made to collide.

A.4.1 LHC radiation source

In the so-called high-luminosity experiments (CMS and ATLAS) the circulating proton bunches, of roughly 1011 protons each, and separated by $25ns$, collide producing approximately 20 inelastic interactions per crossing. The number of collisions per unit cross section produced in a detector is defined as the luminosity. The nominal design luminosity value for the LHC is $10^{34}cm^{-2}s^{-1}$. The main performance parameter for a high-energy particle collider is the integrated luminosity. It is expressed in units of inverse cross section, typically as inverse femtobarns (fb^{-1}) which for TeV energies corresponds to roughly 10^{14} collisions.

In the vicinity of the accelerator, a mixed radiation field is present consisting of multiple particle species and energies and that can therefore negatively impact the availability of the accelerator equipment through radiation effects on electronic components and systems. Additionally, a large number of COTS FPGAs are in a

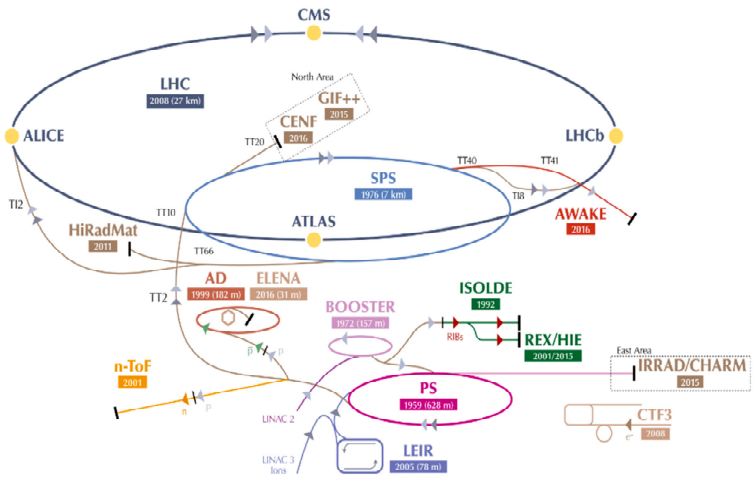


Figure A.4: The CERN accelerator complex, from the injector chain up to the LHC [17].

close vicinity of the accelerator. The radiation effects to electronics can negatively impact the performance of the accelerator either through the induction of failures leading to premature beam dumps and/or the need of intervention to repair them, increasing the accelerator downtime.

A.4.2 LHC Radiation Environment

The LHC radiation environment largely depends on the angle and distance with respect to the source, as well as on the type and amount of shielding (if any) in between. It is originated by three sources of radiation:

- particle debris induced by colliding beams
- direct losses in beam intercepting devices
- interactions between the circulating beam and the residual gas inside the beam pipe

In the LHC, the contribution of the various high-energy particles and its respective radiation damage, depends highly on the specific radiation environment. In general, hadrons (neutrons, protons and pions) are the main SEE contributors. Charged hadrons, electrons/positrons and photons dominate Total Ionizing Dose (TID), and neutrons are the primaru cause of the DD.

A.5 Monitoring and Characterization of Radiation Environments

The radiation-induced effect rate, or failure risk, is determined by complex calculations, from the radiation environment source, through the transport through spacecraft structures, down to the specific effects in electronics. Currently, no end-to-end risk assessment is available for radiation effects to electronics. For each element of the calculation, a confidence level is calculated that is based on statistical uncertainties from the observed environment variability, from conservative predictions, or from engineering margins (that have been already introduced to ensure safety). Recent engineering tool developments are aiming at providing model predictions based on statistical analysis. Additionally, various efforts include model prediction of the dynamic nature of the environments, and predictions that associate mission confidence levels and their consistency with the variability of the particle fluxes [41, 71, 100].

A.6 Conclusion

The atmospheric radiation environment has always existed. However, due to the technology scaling according to Moore's law, the trends in microelectronics technologies towards smaller feature- sizes, require constantly much lower critical charges to upset devices. The Concorde for example, flew at higher altitudes than almost all current commercial jets, but the 1960s/70s technology-base of that airliner meant that there was negligible chance of GCR or SEP radiation affecting its systems. Today there is much greater awareness of the vulnerabilities of microelectronics in terrestrial applications from the natural cosmic-radiation-induced radiation environment, and it is recognized by semiconductor manufacturers as a key risk affecting the reliability of future state-of-the-art components, in safety-critical applications, such as automotive systems, flight and aircraft engine control, data centers and nuclear plants.

Bibliography

[1] P. Adell, G. Allen, G. Swift, and S. McClure. Assessing and mitigating radiation effects in Xilinx SRAM FPGAs. In *Radiation and Its Effects on Components and Systems (RADECS), 2008 European Conference on*, pages 418–424, Sept 2008.

[2] M. A. Aguirre, J. N. Tombs, F. Munoz, V. Baena, H. Guzman, J. Napoles, A. Torralba, A. Fernandez-Leon, F. Tortosa-Lopez, and D. Merodio. Selective protection analysis using a SEU emulator: Testing protocol and case study over the leon2 processor. *IEEE Transactions on Nuclear Science*, 54(4):951–956, Aug 2007.

[3] M. Alderighi, F. Casini, S. D'Angelo, M. Mancini, D. M. Codinachs, S. Pastore, C. Poivey, G. R. Sechi, G. Sorrenti, and R. Weigand. Experimental validation of fault injection analyses by the flipper tool. *IEEE Transactions on Nuclear Science*, 57(4):2129–2134, Aug 2010.

[4] H. Angepat, G. Eads, C. Craik, and D. Chiou. NIFD: Non-intrusive FPGA Debugger – Debugging FPGA 'Threads' for Rapid HW/SW Systems Prototyping. In *2010 International Conference on Field Programmable Logic and Applications*, pages 356–359, Aug 2010.

[5] L. Antoni, R. Leveugle, and B. Fehér. Using run-time reconfiguration for fault injection applications. *Instrumentation and Measurement, IEEE Transactions on*, 52(5):1468–1473, 2003.

[6] J. L. Barth, C. S. Dyer, and E. G. Stassinopoulos. Space, atmospheric, and terrestrial radiation environments. *IEEE Transactions on Nuclear Science*, 50(3):466–482, June 2003.

[7] J. L. Barth, K. A. LaBel, and C. Poivey. Radiation assurance for the space environment. In *2004 International Conference on Integrated Circuit Design and Technology (IEEE Cat. No.04EX866)*, pages 323–333, May 2004.

[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*, 55(4):2259–2266, Aug 2008.

[9] M. Bohman, B. James, M. J. Wirthlin, H. Quinn, and J. Goeders. Microcontroller compiler-assisted software fault tolerance. *IEEE Transactions on Nuclear Science*, 66(1):223–232, Jan 2019.

[10] C. Bolchini, A. Miele, and C. Sandionigi. A novel design methodology for implementing reliability-aware systems on SRAM-based FPGAs. *IEEE Transactions on Computers*, 60(12):1744–1758, Dec 2011.

[11] B. Bridgford, C. Carmichael, and C. W. Tseng. Single-event upset mitigation selection guide. *Xilinx Application Note, XAPP987 (v1. 0)*, 2008.

[12] K. Bruneel, F. Abouelella, and D. Stroobandt. Automatically mapping applications to a self-reconfiguring platform. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, pages 964–969, 3001 Leuven, Belgium, Belgium, 2009. European Design and Automation Association.

[13] K. Bruneel, W. Heirman, and D. Stroobandt. Dynamic data folding with parameterizable FPGA configurations. *ACM Transactions on Design Automation of Electronic Systems*, 16(4):43:1–43:29, Oct. 2011.

[14] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli. Theory of latency-insensitive design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(9):1059–1076, Sep 2001.

[15] C. Carmichael. Triple module redundancy design techniques for virtex FPGAs. Retrieved from: https://www.xilinx.com/support/documentation/application_notes/xapp197, 2018.

[16] C. Carmichael and C. W. Tseng. Correcting single-event upsets in virtex-4 FPGA configuration memory. Retrieved from: http://citeseerx.ist.psu.edu/viewdoc/download, 2018.

[17] CERN. CERN document server. Retrieved from: https://cds.cern.ch/, 2019.

[18] J. Chang, G. A. Reis, and D. I. August. Automatic instruction-level software-only recovery. In *International Conference on Dependable Systems and Networks (DSN'06)*, pages 83–92, June 2006.

[19] K.-T. Cheng, S.-Y. Huang, and W.-J. Dai. Fault emulation: A new methodology for fault grading. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(10):1487–1495, Oct 1999.

[20] E. Chielle, G. S. Rodrigues, F. L. Kastensmidt, S. Cuenca-Asensi, L. A. Tambara, P. Rech, and H. Quinn. S-SETA: Selective software-only error-detection technique using assertions. *IEEE Transactions on Nuclear Science*, 62(6):3088–3095, Dec 2015.

[21] E. Chielle, F. Rosa, G. S. Rodrigues, L. A. Tambara, J. Tonfat, E. Macchione, F. Aguirre, N. Added, N. Medina, V. Aguiar, M. A. G. Silveira, L. Ost, R. Reis, S. Cuenca-Asensi, and F. L. Kastensmidt. Reliability on ARM processors against soft errors through SIHFT techniques. *IEEE Transactions on Nuclear Science*, 63(4):2208–2216, Aug 2016.

[22] S. Y. L. Chin and S. J. E. Wilton. An analytical model relating FPGA architecture and place and route runtime. In *2009 International Conference on Field Programmable Logic and Applications*, pages 146–153, Aug 2009.

[23] P. Civera, L. Macchiarulo, M. Rebaudengo, M. S. Reorda, and M. Violante. An FPGA-based approach for speeding-up fault injection campaigns on safety-critical circuits. *Journal of Electronic Testing*, 18(3):261–271, Jun 2002.

[24] N. Combier, A. Claret, P. Laurent, V. Maget, D. Boscher, A. Ferrari, and M. Brugger. Improvements of fluka calculation of the neutron albedo. *IEEE Transactions on Nuclear Science*, 64(1):614–621, Jan 2017.

[25] J. Coole and G. Stitt. Intermediate fabrics: Virtual architectures for circuit portability and fast placement and routing. In *2010 IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 13–22, Oct 2010.

[26] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, Oct 1974.

[27] M. Didehban and A. Shrivastava. nzdc: A compiler technique for near zero silent data corruption. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2016.

[28] P. E. Dodd, M. R. Shaneyfelt, J. A. Felix, and J. R. Schwank. Production and propagation of single-event transients in high-speed digital logic ICs. *IEEE Transactions on Nuclear Science*, 51(6):3278–3284, Dec 2004.

[29] J. Drabbe. ECSS-E-ST-10-12C: methods for the calculation of radiation received and its effects, and a policy for design margins. Retrieved from: https://ecss.nl/standard/ecss-e-st-10-12c-methods-for-the-calculation-of-radiation-received-and-its-effects-and-a-policy-for-design-margins/, 2019.

[30] M. Ebrahimi, A. Mohammadi, A. Ejlali, and S. G. Miremadi. A fast, flexible, and easy-to-develop FPGA-based fault injection technique. *Microelectronics Reliability*, 54(5):1000 – 1008, 2014.

[31] F. Eslami, E. Hung, and S. J. E. Wilton. Enabling effective FPGA debug using overlays: Opportunities and challenges. *CoRR*, abs/1606.06457, 2016.

[32] F. Eslami and S. J. E. Wilton. An adaptive virtual overlay for fast trigger insertion for FPGA debug. In *2015 International Conference on Field Programmable Technology (FPT)*, pages 32–39, Dec 2015.

[33] L. Evans and P. Bryant. LHC machine. *Journal of Instrumentation*, 3(08):S08001–S08001, aug 2008.

[34] E. Fermi. On the origin of the cosmic radiation. *Phys. Rev.*, 75:1169–1174, Apr 1949.

[35] C. Fetzer, U. Schiffel, and M. Süßkraut. An-encoding compiler: Building safety-critical systems with commodity hardware. In B. Buth, G. Rabe, and T. Seyfarth, editors, *Computer Safety, Reliability, and Security*, pages 283–296, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[36] H. D. Foster. Trends in functional verification: A 2014 industry study. In *Proceedings of the 52Nd Annual Design Automation Conference*, DAC '15, pages 48:1–48:6, New York, NY, USA, 2015. ACM.

[37] P. W. Foulk. Data-folding in SRAM configurable FPGAs. In *[1993] Proceedings IEEE Workshop on FPGAs for Custom Computing Machines*, pages 163–171, April 1993.

[38] F. Fricke, A. Werner, K. Shahin, and M. Huebner. CGRA tool flow for fast run-time reconfiguration. In N. Voros, M. Huebner, G. Keramidas, D. Goehringer, C. Antonopoulos, and P. C. Diniz, editors, *Applied Reconfigurable Computing. Architectures, Tools, and Applications*, pages 661–672, Cham, 2018. Springer International Publishing.

[39] A. Gavros, H. H. Loomis, and A. A. Ross. Reduced precision redundancy in a radix-4 FFT implementation on a field programmable gate array. In *2011 Aerospace Conference*, pages 1–15, March 2011.

[40] M. G. Gericota, L. F. Lemos, G. R. Alves, and J. M. Ferreira. On-line self-healing of circuits implemented on reconfigurable FPGAs. In *13th IEEE International On-Line Testing Symposium (IOLTS 2007)*, pages 217–222, July 2007.

[41] G. P. Ginet, T. P. O'Brien, S. L. Huston, W. R. Johnston, T. B. Guild, R. Friedel, C. D. Lindstrom, C. J. Roth, P. Whelan, R. A. Quinn, D. Madden, S. Morley, and Y.-J. Su. Ae9, ap9 and spm: New models for specifying the trapped energetic particle and space plasma environment. *Space Science Reviews*, 179(1):579–615, Nov 2013.

[42] R. Gnanaolivu, T. S. Norvell, and R. Venkatesan. Mapping loops onto coarse-grained reconfigurable architectures using particle swarm optimization. In *2010 International Conference of Soft Computing and Pattern Recognition*, pages 145–151, Dec 2010.

[43] J. Goeders and S. J. E. Wilton. Using Dynamic Signal-Tracing to Debug Compiler-Optimized HLS Circuits on FPGAs. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 127–134, May 2015.

[44] J. Goeders and S. J. E. Wilton. Signal-tracing techniques for in-system FPGA debugging of high-level synthesis circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 36(1):83–96, Jan 2017.

[45] M. Gokhale, P. Graham, E. Johnson, N. Rollins, and M. Wirthlin. Dynamic reconfiguration for management of radiation-induced faults in FPGAs. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, pages 145–, April 2004.

[46] 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)*, pages 117–124, Dec 2016.

[47] P. Graham, B. Nelson, and B. Hutchings. Instrumenting bitstreams for debugging FPGA circuits. In *The 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'01)*, pages 41–50, March 2001.

[48] D. Grant, C. Wang, and G. G. Lemieux. A CAD framework for Malibu: An FPGA with time-multiplexed coarse-grained elements. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, pages 123–132, New York, NY, USA, 2011. ACM.

[49] S. Habinc. Gaisler research, functional triple modular redundancy (FTMR). Retrieved from: https://www.gaisler.com/doc/fpga_003_01-0-2.pdf. Accessed: 2019-01-26.

[50] S. Hansen, D. Koch, and J. Torresen. High speed partial run-time reconfiguration using enhanced ICAP hard macro. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 174–180, May 2011.

[51] J. Heiner, N. Collins, and M. Wirthlin. Fault tolerant ICAP controller for high-reliable internal scrubbing. In *2008 IEEE Aerospace Conference*, pages 1–10, doi. 10.1109/AERO.2008.4526471, March 2008.

[52] J. Heiner, B. Sellers, M. Wirthlin, and J. Kalb. FPGA partial reconfiguration via configuration scrubbing. In *2009 International Conference on Field Programmable Logic and Applications*, pages 99–104, Aug 2009.

[53] L. Hellerman. A catalog of three-variable or-invert and and-invert logical circuits. *IEEE Transactions on Electronic Computers*, EC-12(3):198–223, June 1963.

[54] I. Herrera-Alzu and M. Lopez-Vallejo. Design techniques for Xilinx Virtex FPGA configuration memory scrubbers. *IEEE Transactions on Nuclear Science*, 60(1):376–385, Feb 2013.

[55] K. Heyse, B. Al Farisi, K. Bruneel, and D. Stroobandt. TCONMAP: Technology Mapping for Parameterised FPGA Configurations. *ACM Trans. Des. Autom. Electron. Syst.*, 20(4):48:1–48:27, Sept. 2015.

[56] K. Heyse, K. Bruneel, and D. Stroobandt. Mapping logic to reconfigurable FPGA routing. *Proceedings of the 22nd International Conference on Field Programmable Logic and Applications*, pages 315–321, 2012.

[57] K. Heyse, T. Davidson, E. Vansteenkiste, K. Bruneel, and D. Stroobandt. Efficient implementation of virtual coarse grained reconfigurable arrays on FPGAS. In *2013 23rd International Conference on Field programmable Logic and Applications*, pages 1–8, Sept 2013.

[58] A. B. T. Hopkins and K. D. McDonald-Maier. Debug support for complex systems on-chip: a review. *IEEE Proceedings - Computers and Digital Techniques*, 153(4):197–207, July 2006.

[59] E. Hung, A. Jamal, and S. J. E. Wilton. Maximum flow algorithms for maximum observability during FPGA debug. In *2013 International Conference on Field-Programmable Technology (FPT)*, pages 20–27, Dec 2013.

[60] E. Hung, T. Todman, and W. Luk. Transparent insertion of latency-oblivious logic onto FPGAs. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, Sep. 2014.

[61] E. Hung and S. J. Wilton. Towards simulator-like observability for FPGAs: A virtual overlay network for trace-buffers. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '13, pages 19–28, New York, NY, USA, 2013. ACM.

[62] E. Hung and S. J. E. Wilton. Incremental trace-buffer insertion for FPGA debug. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 22(4):850–863, April 2014.

[63] B. L. Hutchings and J. Keeley. Rapid post-map insertion of embedded logic analyzers for Xilinx FPGAs. In *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 72–79, May 2014.

[64] Intel. Quartus Prime Standard Edition Handbook, Volume 3: Verification: Design and Debugging with the SignalTap II Logic Analyzer. Retrieved from: https://www.mouser.com/pdfdocs/qts-qps-5v3.pdf, 2019. Accessed: 2019-01-02.

[65] Intel. Quartus SignalTap: embedded logic analyzer. Retrieved from: https://www.intel.com/content/pdfs/literature/ug/signal.pdf, 2019. Accessed: 2019-04-22.

[66] ISCAS89. ISCAS89 : Benchmark Suite. Retrieved from:https://ddd.fit.cvut.cz/prj/Benchmarks/, 2018.

[67] R. K. Iyer and D. Tang. Fault-tolerant computer system design. In D. K. Pradhan, editor, *Fault-tolerant Computer System Design*, chapter Experimental Analysis of Computer System Dependability, pages 282–392. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[68] 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 Trans. Reconfigurable Technol. Syst.*, 5(4):21:1–21:30, Dec. 2012.

[69] A. K. Jain, S. A. Fahmy, and D. L. Maskell. Efficient overlay architecture based on DSP blocks. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 25–28, May 2015.

[70] A. K. Jain, X. Li, P. Singhai, D. L. Maskell, and S. A. Fahmy. DeCO: A DSP block based FPGA accelerator overlay with low overhead interconnect. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 1–8, May 2016.

[71] P. Jiggens, A. Varotsou, P. Truscott, D. Heynderickx, F. Lei, H. Evans, and E. Daly. The solar accumulated and peak proton and heavy ion radiation environment (SAPPHIRE) model. *IEEE Transactions on Nuclear Science*, 65(2):698–711, Feb 2018.

[72] J. Johnson, W. Howes, M. Wirthlin, D. L. McMurtrey, M. Caffrey, P. Graham, and K. Morgan. Using duplication with compare for on-line error detection in FPGA-based designs. In *2008 IEEE Aerospace Conference*, pages 1–11, March 2008.

[73] M. A. Kadi and M. Huebner. Integer computations with soft GPGPU on FPGAs. In *2016 International Conference on Field-Programmable Technology (FPT)*, pages 28–35, Dec 2016.

[74] F. Kastensmidt, L. Carro, and R. Reis. *Fault-tolerance techniques for SRAM-based FPGAs*, volume 32. Springer, 01 2006.

[75] A. Khan, R. N. Pittman, and A. Forin. gNOSIS: A Board-Level Debugging and Verification Tool. In *Proceedings of the 2010 International Conference on Reconfigurable Computing and FPGAs*, RECONFIG '10, pages 43–48, Washington, DC, USA, 2010. IEEE Computer Society.

[76] M. P. King, R. A. Reed, R. A. Weller, M. H. Mendenhall, R. D. Schrimpf, B. D. Sierawski, A. L. Sternberg, B. Narasimham, J. K. Wang, E. Pitta, B. Bartz, D. Reed, C. Monzel, R. C. Baumann, X. Deng, J. A. Pellish, M. D. Berg, C. M. Seidleck, E. C. Auden, S. L. Weeden-Wright, N. J. Gaspard, C. X. Zhang, and D. M. Fleetwood. Electron-induced single-event upsets in static random access memory. *IEEE Transactions on Nuclear Science*, 60(6):4122–4129, Dec 2013.

[77] D. Koch. Self-adaptive reconfigurable networks. In *Partial Reconfiguration on FPGAs: Architectures, Tools and Applications*, pages 285–305, Cham, 2012. Springer Science & Business Media.

[78] D. Koch, C. Beckhoff, and G. G. F. Lemieux. An efficient FPGA overlay for portable custom instruction set extensions. In *2013 23rd International Conference on Field programmable Logic and Applications*, pages 1–8, Sept 2013.

[79] D. Koch, C. Haubelt, and J. Teich. Efficient Hardware Checkpointing: Concepts, Overhead Analysis, and Implementation. In *Proceedings of the 2007 ACM/SIGDA 15th International Symposium on Field Programmable Gate Arrays*, FPGA '07, pages 188–196, New York, NY, USA, 2007. ACM.

[80] A. Kourfali, F. Fricke, M. Huebner, and D. Stroobandt. An integrated on-silicon verification method for FPGA overlays. *Journal of Electronic Testing*, Mar 2019.

[81] A. Kourfali, A. Kulkarni, and D. Stroobandt. SICTA: A superimposed in-circuit fault tolerant architecture for SRAM-based FPGAs. In *Proceedings of the IEEE 23rd International Symposium on On-Line Testing and Robust System Design*, page 4. IEEE, 2017.

[82] A. Kourfali and D. Stroobandt. Efficient Hardware Debugging Using Parameterized FPGA Reconfiguration. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 277–282, May 2016.

[83] A. Kourfali and D. Stroobandt. Superimposed in-circuit debugging for self-healing FPGA overlays. In *2018 IEEE 19th Latin-American Test Symposium (LATS)*, pages 1–6, March 2018.

[84] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, pages 1097–1105, USA, 2012. Curran Associates Inc.

[85] S. Kulis. Single event effects mitigation with TMRG tool. *Journal of Instrumentation*, 12(01):C01082, 2017.

[86] A. Kulkarni, T. Davidson, K. Heyse, and D. Stroobandt. Improving reconfiguration speed for dynamic circuit specialization using placement constraints. In *Proceedings International Conference on Reconfigurable Computing and FPGAs*, pages 1–6. IEEE xplore, 2014.

[87] A. Kulkarni, K. Heyse, T. Davidson, and D. Stroobandt. Performance evaluation of dynamic circuit specialization on xilinx FPGAs. In *Proceedings of the FPGA World Conference 2014*, FPGAWorld '14, pages 1:1–1:6, New York, NY, USA, 2014. ACM.

[88] A. Kulkarni, V. Kizheppatt, and D. Stroobandt. MiCAP: a custom reconfiguration controller for dynamic circuit specialization. In *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–6, doi. 10.1109/ReConFig.2015.7393327, Dec 2015.

[89] A. Kulkarni and D. Stroobandt. Micap-pro: a high speed custom reconfiguration controller for dynamic circuit specialization. *Design Automation for Embedded Systems*, 20(4):341–359, Dec 2016.

[90] A. Kulkarni, D. Stroobandt, A. Werner, F. Fricke, and M. Huebner. Pixie: A heterogeneous virtual coarse-grained reconfigurable array for high performance image processing applications. In *3rd International Workshop on Overlay Architectures for FPGAs (OLAF 2017)*, pages 1–6, 2017.

[91] A. Kulkarni, E. Vansteenkiste, D. Stroobandt, A. Brokalakis, and A. Nikitakis. A fully parameterized virtual coarse grained reconfigurable array for high performance computing applications. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops*, pages 265–270. IEEE xplore, 2016.

[92] M. Lanuzza, P. Zicari, F. Frustaci, S. Perri, and P. Corsonello. Exploiting self-reconfiguration capability to improve sram-based FPGA robustness in space and avionics applications. *ACM Trans. Reconfigurable Technol. Syst.*, 4(1):8:1–8:22, Dec. 2010.

[93] V. Lari, A. Tanase, J. Teich, M. Witterauf, F. Khosravi, F. Hannig, and B. H. Meyer. A co-design approach for fault-tolerant loop execution on coarse-grained reconfigurable arrays. In *2015 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pages 1–8, June 2015.

[94] R. Le. Soft error mitigation using prioritized essential bits. Retrieved from: https://www.xilinx.com/support/documentation/application_notes/xapp538-soft-error-mitigation-essential-bits, 2019.

[95] F. Lei, A. Hands, S. Clucas, C. Dyer, and P. Truscott. Improvements to and validations of the qinetiq atmospheric radiation model (qarm). In *2005 8th European Conference on Radiation and Its Effects on Components and Systems*, pages D3–1–D3–8, Sep. 2005.

[96] C. Lopez-Ongil, L. Entrena, M. Garcia-Valderas, M. Portela, M. A. Aguirre, J. Tombs, V. Baena, and F. Munoz. A unified environment for fault injection at any design level based on emulation. *IEEE Transactions on Nuclear Science*, 54(4):946–950, Aug 2007.

[97] J. Luu, J. Goeders, M. Wainberg, A. Somerville, T. Yu, K. Nasartschuk, M. Nasr, S. Wang, T. Liu, N. Ahmed, K. B. Kent, J. Anderson, J. Rose, and V. Betz. VTR 7.0: Next Generation Architecture and CAD System for FPGAs. *ACM Trans. Reconfigurable Technol. Syst.*, 7(2):6:1–6:30, June 2014.

[98] P. Lysaght, B. Blodget, J. Mason, J. Young, and B. Bridgford. Invited paper: Enhanced architectures, design methodologies and cad tools for dynamic reconfiguration of xilinx FPGAs. In *2006 International Conference on Field Programmable Logic and Applications*, pages 1–6, Aug 2006.

[99] R. L. Lysecky, K. Miller, F. Vahid, and K. A. Vissers. Firm-core virtual FPGA for just-in-time FPGA compilation (abstract only). In *FPGA*, 2005.

[100] D. Matthi, T. Berger, A. I. Mrigakshi, and G. Reitz. A ready-to-use galactic cosmic ray model. *Advances in Space Research*, 51(3):329 – 338, 2013.

[101] Mentor. Modelsim: Simulation of hardware description languages. Retrieved from: https://www.mentor.com/products/fv/modelsim/, 2019.

[102] Mentor. Precision Hi-Rel:Advanced FPGA Synthesis. Retrieved from: https://www.mentor.com/products/fpga/synthesis/precision-hi-rel, 2019.

[103] Mentor Graphics. Certus Silicon Debug, 2016.

[104] H. Michel, H. Guzman-Miranda, A. Dorflinger, H. Michalik, and M. A. Echanove. SEU fault classification by fault injection for an FPGA in the space instrument SOPHI. In *2017 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pages 9–15, July 2017.

[105] J. M. Mogollon, H. Guzmn-Miranda, J. Npoles, J. Barrientos, and M. A. Aguirre. Ftunshades2: A novel platform for early evaluation of robustness against SEE. In *2011 12th European Conference on Radiation and Its Effects on Components and Systems*, pages 169–174, Sep. 2011.

[106] G. E. Moore. Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff. *IEEE Solid-State Circuits Society Newsletter*, 11(3):33–35, Sep. 2006.

[107] 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*, 54(6):2065–2072, Dec 2007.

[108] NASA. Fault management handbook. Retrieved from: https://snebulos.mit.edu/projects/reference/NASA-Generic/NASA-HDBK-1002.pdf, 2019.

[109] G. L. Nazar. Improving FPGA repair under real-time constraints. *Microelectronics Reliability*, 55(7):1109 – 1119, 2015.

[110] G. L. Nazar, L. P. Santos, and L. Carro. Accelerated FPGA repair through shifted scrubbing. In *2013 23rd International Conference on Field programmable Logic and Applications*, pages 1–6, Sep. 2013.

[111] G. Neuberger, F. de Lima, L. Carro, and R. Reis. A multiple bit upset tolerant SRAM memory. *ACM Trans. Des. Autom. Electron. Syst.*, 8(4):577–590, Oct. 2003.

[112] M. Nielson. Using a microprocessor to configure 7 series FPGAs via slave serial or slave SelectMAP mode. Retrieved from: https://www.xilinx.com/support/documentation/application_notes/xapp583-fpga-configuration.pdf, 2019.

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

[114] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.

[115] A. Parreira, J. P. Teixeira, and M. Santos. A novel approach to FPGA-based hardware fault modeling and simulation. In *Proc. of the Design and Diagnostics of Electronic Circuits and Syst. Workshop*, pages 17–24, 2003.

[116] T. P. Peixoto. The graph-tool python library, 2014.

[117] M. Psarakis and A. Apostolakis. Fault tolerant FPGA processor based on runtime reconfigurable modules. In *2012 17th IEEE European Test Symposium (ETS)*, pages 1–6, May 2012.

[118] H. Quinn, Z. Baker, T. Fairbanks, J. L. Tripp, and G. Duran. Software resilience and the effectiveness of software mitigation in microcontrollers. *IEEE Transactions on Nuclear Science*, 62(6):2532–2538, Dec 2015.

[119] H. Quinn, Z. Baker, T. Fairbanks, J. L. Tripp, and G. Duran. Robust duplication with comparison methods in microcontrollers. *IEEE Transactions on Nuclear Science*, 64(1):338–345, Jan 2017.

[120] 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 *2012 IEEE Radiation Effects Data Workshop*, pages 1–8, July 2012.

[121] B. Quinton and S. Wilton. Post-silicon debug using programmable logic cores. In *Proceedings. 2005 IEEE International Conference on Field-Programmable Technology, 2005.*, pages 241–247, Dec 2005.

[122] P. M. B. Rao, M. Ebrahimi, R. Seyyedi, and M. B. Tahoori. Protecting SRAM-based FPGAs against multiple bit upsets using erasure codes. In *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2014.

[123] M. S. Reorda, L. Sterpone, and A. Ullah. An error-detection and self-repairing method for dynamically and partially reconfigurable systems. In *2013 18th IEEE European Test Symposium (ETS)*, pages 1–7, May 2013.

[124] A. Sari and M. Psarakis. Scrubbing-based SEU mitigation approach for systems-on-programmable-chips. In *2011 IEEE International Conference on Field-Programmable Technology*, pages 1–8, Dec 2011.

[125] A. Sari, M. Psarakis, and D. Gizopoulos. Combining checkpointing and scrubbing in FPGA-based real-time systems. In *2013 IEEE 31st VLSI Test Symposium (VTS)*, pages 1–6, April 2013.

[126] T. Schweizer, P. Schlicker, S. Eisenhardt, T. Kuhn, and W. Rosenstiel. Low-cost TMR for fault-tolerance on coarse-grained reconfigurable architectures. In *2011 International Conference on Reconfigurable Computing and FPGAs*, pages 135–140, Nov 2011.

[127] L. Sekanina. Virtual reconfigurable circuits for real-world applications of evolvable hardware. In *Proceedings of the 5th International Conference on Evolvable Systems: From Biology to Hardware*, ICES'03, pages 186–197, Berlin, Heidelberg, 2003. Springer-Verlag.

[128] A. Severance and G. Lemieux. VENICE: A compact vector processor for FPGA applications. In *2011 IEEE Hot Chips 23 Symposium (HCS)*, pages 1–5, Aug 2011.

[129] M. Sharifi, M. Fathy, and M. T. Mahmoudi. A classified and comparative study of edge detection algorithms. In *Proceedings. International Conference on Information Technology: Coding and Computing*, pages 117–120, April 2002.

[130] J.-H. H. Shih-Arn Hwang and C.-W. Wu. Sequential circuit fault simulation using logic emulation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(8):724–736, Aug 1998.

[131] F. Siegle, T. Vladimirova, J. Ilstad, and O. Emam. Mitigation of radiation effects in SRAM-based FPGAs for space applications. *ACM Comput. Surv.*, 47(2):37:1–37:34, Jan. 2015.

[132] H. K.-H. So, C. Liu, F. Hannig, and D. Ziener. *FPGA Overlays*, pages 285–305. Springer International Publishing, Cham, 2016.

[133] M. Sonza Reorda, L. Sterpone, and M. Violante. Multiple errors produced by single upsets in FPGA configuration memory: a possible solution. In *European Test Symposium (ETS'05)*, pages 136–141, May 2005.

[134] SPENVIS. ESA's SPace ENVironment information system. Retrieved from: https://www.spenvis.oma.be/, 2019.

[135] Spjeldvik and Rothwell. Handbook of geophysics and the space environment. *Phys. Rev.*, 75:1169–1174, 1985.

[136] L. Sterpone and N. Battezzati. A novel design flow for the performance optimization of fault tolerant circuits on SRAM-based FPGAs. In *2008 NASA/ESA Conference on Adaptive Hardware and Systems*, pages 157–163, June 2008.

[137] L. Sterpone, M. S. Reorda, and M. Violante. Rora: a reliability-oriented place and route algorithm for SRAM-based FPGAs. In *Research in Microelectronics and Electronics, 2005 PhD*, volume 1, pages 173–176 vol.1, July 2005.

[138] L. Sterpone and M. Violante. A new partial reconfiguration-based fault-injection system to evaluate seu effects in SRAM-based FPGAs. *IEEE Transactions on Nuclear Science*, 54(4):965–970, Aug 2007.

[139] L. Sterpone and M. Violante. Static and dynamic analysis of seu effects in SRAM-based FPGAs. In *12th IEEE European Test Symposium (ETS'07)*, pages 159–164, May 2007.

[140] D. Stork. Demo and the route to fusion power. Retrieved from: https://fire.pppl.gov/eu_demo_Stork_FZK2009.

[141] Synopsys. Synplify premier: Accelerate implementation of FPGA designs and FPGA-based prototypes. Retrieved from: https://www.synopsys.com/implementation-and-signoff/fpga-based-design/synplify-premier.html, 2019. Accessed: 2018-02-22.

[142] D. Synopsys. Identify: simulator-like visibility into hardware debug. Retrieved from: https://www.synopsys.com/implementation-and-signoff/fpga-based-design/identify-rtl-debugger.html, 2019. Accessed:2019-01-10.

[143] M. A. A. Tuhin and T. S. Norvell. Compiling parallel applications to coarse-grained reconfigurable architectures. In *2008 Canadian Conference on Electrical and Computer Engineering*, pages 001723–001728, May 2008.

[144] G. Tzimpragos, D. Cheng, S.Tapp, B. Jayadev, and A. Majumdar. Application Debug in FPGAs in the Presence of Multiple Asynchronous Clocks. In *2016 IEEE International Conference on Field-Programmable Technology, 2016. (FPT). Proceedings.*, Dec 2016.

[145] UG903. Vivado Design Suite User Guide : Using Constraints, ug903 (v2018.1). Retrieved from: https://www.xilinx.com/support/documentation/sw_manuals/, 2018.

[146] UG973. Programming and Debugging: Vivado Design Suite User Guide, ug973 (v2018.1). Retrieved from: https://www.xilinx.com/products/design-tools/vivado.html, 2018.

[147] E. Vansteenkiste, B. A. Farisi, K. Bruneel, and D. Stroobandt. TPaR: Place and route tools for the dynamic reconfiguration of the FPGA's interconnect network. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 33(3):370–383, March 2014.

[148] T. Wheeler, P. Graham, B. Nelson, and B. Hutchings. *Using Design-Level Scan to Improve FPGA Design Observability and Controllability for Functional Verification*, pages 483–492. Springer Berlin Heidelberg, 2001.

[149] M. Wirthlin, D. Lee, G. Swift, and H. Quinn. A method and case study on identifying physically adjacent multiple cell upsets using 28-nm, interleaved and SECDED-protected arrays. *IEEE Transactions on Nuclear Science*, 61(6):3080–3087, Dec 2014.

[150] M. J. Wirthlin and B. L. Hutchings. Improving functional density through run-time constant propagation. In *Proceedings of the 1997 ACM Fifth International Symposium on Field-programmable Gate Arrays*, FPGA '97, pages 86–92, New York, NY, USA, 1997. ACM.

[151] M. A. Xapsos, C. Stauffer, T. Jordan, J. L. Barth, and R. A. Mewaldt. Model for cumulative solar heavy ion energy and linear energy transfer spectra. *IEEE Transactions on Nuclear Science*, 54(6):1985–1989, Dec 2007.

[152] Xilinx. Soft error mitigation controller (pg036) v4.1. Retrieved from: https://www.xilinx.com/support/documentation/ip_documentation/sem/v4_1/pg036_sem.pdf. Accessed: 2019-01-26.

[153] Xilinx. Virtex-5 FPGA Configuration User Guide (ug191). `https://www.xilinx.com/support/documentation/user_guides/ug191.pdf`. Accessed: 2019-03-14.

[154] Xilinx. Xilinx, logicore IP AXI HWICAP (pg134), 2016. Retrieved from: www.xilinx.com/support/documentation/ip_documentation/axi_hwicap/v2_03_a/ds817_axi_hwicap.pdf. Accessed: 2019-01-26.

[155] Xilinx. ChipScope Integrated Logic Analyzer , 2016.

[156] Xilinx, Inc. Configuration Readback Capture in UltraScale FPGAs. Application Note, XAPP1230, Nov 2015.

[157] P. Yiannacouras, J. G. Steffan, and J. Rose. Fine-grain performance scaling of soft vector processors. In *Proceedings of the 2009 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '09, pages 97–106, New York, NY, USA, 2009. ACM.