UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

GABRIEL LUCA NAZAR

Fine-Grained Error Detection Techniques for Fast Repair of FPGAs

Thesis presented in partial fulfillment of the requirements for the degree of Doctor of Computer Science

Prof. Dr. Luigi Carro
Advisor

Porto Alegre, July 2013.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Nazar, Gabriel Luca

Fine-Grained Error Detection Techniques for Fast Repair of FPGAs / Gabriel Luca Nazar. -- 2013.

125 f.

Orientador: Luigi Carro.

Tese (Doutorado) -- Universidade Federal do Rio Grande do Sul, Instituto de Informática, Programa de Pós-Graduação em Computação, Porto Alegre, BR-RS, 2013.

1. FPGA. 2. Detecção de erro. 3. Tempo médio de reparo. I. Carro, Luigi, orient. II. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos Alexandre Netto
Vice-Reitor: Prof. Rui Vicente Oppermann
Pró-Reitor de Pós-Graduação: Prof. Vladimir Pinheiro do Nascimento
Diretor do Instituto de Informática: Prof. Luís da Cunha Lamb
Coordenador do PPGC: Prof. Luigi Carro
Bibliotecário-Chefe do Instituto de Informática: Alexsander Borges Ribeiro

TABLE OF CONTENTS

LIST OF ABBREVIATONS AND ACRONYMS 5
LIST OF FIGURES.. 7
LIST OF TABLES 9
RESUMO 11
ABSTRACT 13
1 INTRODUCTION... 15

1.1 Main goals and contributions.. 20
1.2 Outline 22

2 FPGAS AND THEIR DEPENDABILITY THREATS........................... 23

2.1 FPGA Architecture basics.. 23
2.2 Dependability threats for FPGAs 25
2.2.1 Radiation effects... 26
2.2.2 Aging effects 29
2.2.3 Technology scaling and dependability metrics 30

3 FAULT TOLERANCE TECHNIQUES FOR FPGAS 33

3.1 Techniques based on redundancy... 33
3.2 Techniques based on bitstream manipulation 39
3.3 Contributions of this thesis... 43

4 FAULT INJECTION FOR FPGAS... 45

4.1 Fault injection Basics... 45
4.2 Fault injection for FPGA-based systems.. 48
4.2.1 Radiation experiments.. 48
4.2.2 Artificial bitstream fault injection 50

4.3 Fault injection platform.. 51
4.3.1 Platform components.. 52
4.3.2 Area costs 54
4.3.3 Injection Rate 55

5 FINE-GRAINED ERROR DETECTION.. 57

5.1 Fine-grained detection with carry propagation chains...................... 57
5.2 Experimental setup 59
5.3 Experimental results... 61
5.3.1 Area... 62
5.3.2 Clock period.. 64
5.3.3 Error detection... 65

5.3.4 Detection acceleration 68

5.4 Radiation Experiments 69
5.4.1 Tested circuit 70
5.4.2 Neutron experiments results 72
5.4.3 Comparison to fault injection results 74

6 FINE-GRAINED DIAGNOSIS AND LOCAL REPAIR 77

6.1 Challenges 77
6.2 The SURFER approach 78
6.2.1 Overview 78
6.2.2 Reducing the MTTR through optimized starting frames 79
6.2.3 Optimum frame identification 80

6.3 Extended experimental setup 81
6.4 PST - Perfect Signature Translation 84
6.5 HST - Heuristic Signature Translation 85
6.5.1 Heuristic table generation 85
6.5.2 Area and delay costs 90
6.5.3 MTTR Results 92
6.5.4 Evaluating the impact of the *maxSize* parameter 95

7 CONCLUSIONS 97

7.1 Summary of contributions 97
7.1.1 Fault injection platform 97
7.1.2 Platform for radiation experiments 97
7.1.3 Carry chain circuits for fine-grained comparison 98
7.1.4 Making use of fine-grained diagnosis with SURFER 98

7.2 Future works 99
7.2.1 Choosing intermediate redundancy grains 99
7.2.2 Further exploring the SURFER design space 99
7.2.3 Diagnosing permanent faults and aging 100
7.2.4 Performing radiation testing over a complete SURFER platform 100
7.2.5 Finding other uses for the signature translation heuristic 100

7.3 Publications 100
7.3.1 Book chapters 100
7.3.2 Journal 101
7.3.3 Conferences and workshops 101

REFERENCES 103
APPENDIX A – TAXONOMY OF DEPENDABLE SYSTEMS 109
APPENDIX B – USING NON-RANDOM INPUT VECTORS 115
APPENDIX C – MAXSIZE EVALUATION RESULTS 119
APPENDIX D – RESUMO EM PORTUGUÊS 123

LIST OF ABBREVIATONS AND ACRONYMS

ALM	Adaptive Logic Module
ALU	Arithmetic and Logic Unit
ASIC	Application Specific Integrated Circuit
AUT	Area Under Test
BRAM	Block Random Access Memory
BTI	Bias Temperature Instability
CED	Concurrent Error Detection
CG-DMR	Coarse-Grained Dual Modular Redundancy
CLB	Configurable Logic Block
CMOS	Complementary Metal-Oxide-Semiconductor
CRC	Cyclic Redundancy Check
CUT	Circuit Under Test
DMR	Dual Modular Redundancy
DSP	Digital Signal Processing
DUT	Device Under Test
DWC	Duplication With Comparison
ECC	Error Correcting Code
FG-DMR	Fine-Grained Dual Modular Redundancy
FGTMR	Fine-Grained Triple Modular Redundancy
FIT	Failures In Time
FPGA	Field Programmable Gate Array
FSD	Faulty State Description
HCI	Hot Carrier Injection
IC	Integrated Circuit
ICAP	Internal Configuration Access Port
IOB	Input/Output Block
LFSR	Linear Feedback Shift Register
LUT	Lookup Table
MBU	Multiple Bit Upset
MCNC	Microelectronics Center of North Carolina
MTBF	Mean Time Between Failures
MTTF	Mean Time To Failure

MTTR	Mean Time To Repair
NBTI	Negative Bias Temperature Instability
NMOS	N-type Metal-Oxide-Semiconductor
NMR	N Modular Redundancy
PBTI	Positive Bias Temperature Instability
PI	Primary Input
PMOS	P-type Metal-Oxide-Semiconductor
PO	Primary Output
QFDR	Quadruple Force Decide Redundancy
RAM	Random Access Memory
ROM	Read-Only Memory
RoRA	Reliability-oriented place and Route Algorithm
SBU	Single Bit Upset
SEB	Single Event Burnout
SEE	Single Event Effect
SEGR	Single Event Gate Rupture
SEL	Single Event Latchup
SES	Single Event Snapback
SET	Single Event Transient
SEU	Single Event Upset
SEUPI	Single Event Upset Probability Impact
SPOF	Single Point Of Failure
SRAM	Static Random Access Memory
STAR	Self-Testing Area
SURFER	Scrubbing Unit Repositioning for Fast Error Repair
TDDB	Time-Dependent Dielectric Breakdown
TID	Total Ionizing Dose
TMR	Triple Modular Redundancy
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
XST	Xilinx Synthesis Technology

LIST OF FIGURES

Figure 1.1: Fault-free circuit and its associated configuration bits (a) and faulty circuit due to a configuration upset (b) ... 17
Figure 1.2: Fault-free output (a), one cycle error (b) and 56 cycles error (c). The markings on the *x* axes show the fault duration ... 18
Figure 1.3: Iterative development cycle of dependable systems ... 19
Figure 1.4: Coarse-grained (a) and fine-grained (b) DMR ... 21
Figure 2.1: Example of a 3-input LUT implementing the XOR function ... 24
Figure 2.2: Schematic of a Virtex 5 slice (XILINX, INC., 2010) ... 24
Figure 2.3: Effects of an energetic ion on a silicon device ... 26
Figure 2.4: Different outcomes of a single event transient (SET) ... 27
Figure 2.5: Single Event Upset on an SRAM cell ... 27
Figure 2.6: Technology scaling and the bathtub curve ... 30
Figure 3.1: DMR (a), TMR (b) and TMR with tripled voters (c) ... 34
Figure 3.2: TMR, DMR and time-redundancy hybrid technique (LIMA, CARRO and REIS, 2003) ... 35
Figure 3.3: 6-input LUT built with two 5-input LUTs (a), and with the XOR gate added for comparison (b) (KYRIAKOULAKOS and PNEVMATIKATOS, 2009) ... 37
Figure 3.4: FPGA system with external reconfiguration controller (a) and partition scrubbing mechanism (b) (BOLCHINI, MIELE and SANDIONIGI, 2011) ... 38
Figure 3.5: Total scrub time for the largest Xilinx FPGA of each family ... 41
Figure 3.6: The roving STARs approach with horizontal (H-STAR) and vertical (V-STAR) testing areas (EMMERT, STROUD and ABRAMOVICI, 2007) ... 42
Figure 3.7: System with partial reconfiguration controller and multiple error signals (STRAKA, KASTIL and KOTASEK, 2010) ... 43
Figure 4.1: Basic components of a fault injection platform ... 45
Figure 4.2: Static cross-section per configuration bit, as reported by (XILINX, INC., 2012c) ... 49
Figure 4.3: Static cross-section for the configuration of the largest device of each family ... 50
Figure 4.4: Fault injection base architecture ... 52
Figure 5.1: Carry chain circuit applied to fine-grained comparison ... 58
Figure 5.2: Incorrect (a) and correct (b) routing in FG-DMR. Dashed lines denote critical routing paths ... 58
Figure 5.3: Redundant heterogeneous comparators ... 59
Figure 5.4: Experimental design flow ... 60
Figure 5.5: Locations of faults of each category ... 61
Figure 5.6: Area overheads for CG-DMR (left-hand bar of each circuit) and FG-DMR (right-hand bar) ... 62

Figure 5.7: Minimum clock period T_{Clk} for the unhardened circuit, CG-DMR and FG-DMR 65
Figure 5.8: Undetected error variation. Positive values indicate a smaller amount for CG-DMR. 67
Figure 5.9: Fault coverage for FG-DMR and CG-DMR. 67
Figure 5.10: Reduction in cycles to detect an error 69
Figure 5.11: Reduction in error detection time 69
Figure 5.12: ISIS spectrum compared to those of the LANSCE and TRIUMF facilities and to the terrestrial one at sea level multiplied by 10^7 and 10^8 (VIOLANTE, STERPONE, MANUZZATO, *et al.*, 2007) 70
Figure 5.13: Placement of replicas and control unit. Even numbered replicas (in light gray) used the proposed FG-DMR while odd numbered ones used CG-DMR. 70
Figure 5.14: Disposition of multi-CUT events on the FPGA. All such events occurred with neighboring CUTs. 73
Figure 5.15: Events reported at each instance. 74
Figure 6.1: Fine-grained detection and the generated error signature 77
Figure 6.2: Overview of a system with SURFER 79
Figure 6.3: Histograms of two signatures for the *pdc* circuit 80
Figure 6.4: Extended experimental setup 82
Figure 6.5: MTTR of standard scrubbing, PST with training and testing signatures 85
Figure 6.6: HST Generation algorithm 86
Figure 6.7: Schematic of a HST circuit 87
Figure 6.8: The weight of an edge $\{u, v\}$ 87
Figure 6.9: Redundant translation table 89
Figure 6.10: Area overhead of circuits with standard CG-DMR (left-hand bars) and circuits with FG-DMR and HST tables (right-hand bars) 90
Figure 6.11: Minimum clock period T_{Clk} for CG-DMR and FG-DMR with pipelined and combinational HST 92
Figure 6.12: MTTR for the HST mechanism (with training and testing lists). PST and standard scrubbing are shown for comparison. 93
Figure 6.13: MTTR increase due to faults affecting the translation table 94
Figure 6.14: MTTR and table area for different *maxSize* values 95
Figure A.1: The bathtub curve 112
Figure B.1: Average detection cycles (a) and detection time (b) for different input sets 116
Figure B.2: Experimental flow for testing SURFER with varied input vectors 117
Figure B.3: MTTR with different translation tables and signature sets 117

LIST OF TABLES

Table 4.1: Required resources and device occupation for a fault injection platform..... 54
Table 5.1: Input benchmark circuits.. 61
Table 5.2: Area costs in LUTs (comparators, error aggregation and total, including the two circuit copies) 63
Table 5.3: Minimum clock period in nanoseconds.. 64
Table 5.4: Amount of faults in each category 66
Table 5.5: Average amount of cycles and associated time to detect an error................. 68
Table 5.6: Received events classification.. 72
Table 5.7: Cross-section and failure in time at New York City 73
Table 5.8: Fault injection and radiation results for “Pre” FSDs................................ 75
Table 5.9: Fault injection and radiation results for “Post” FSDs 75
Table 6.1: Total signature size S_{size}, amount of received signatures (*O*) and of different signatures (/*S*/) for each circuit.. 83
Table 6.2: MTTR of standard scrubbing and SURFER with training and testing signatures (in µs) 84
Table 6.3: Area and delay results for SURFER 91
Table 6.4: MTTR (in µs) with fault-free table and with faults in the translation circuit 93
Table C.1: MTTR (in µs) for the first set of circuits 119
Table C.2: Area (in LUTs) for the first set of circuits... 120
Table C.3: MTTR (in µs) for the second set of circuits 120
Table C.4: Area (in LUTs) for the second set of circuits 120
Table C.5: MTTR (in µs) for the third set of circuits.. 121
Table C.6: Area (in LUTs) for the third set of circuits.. 121

Técnicas de Grão Fino de Detecção de Erros para Reparo Rápido de FPGAs

RESUMO

Field Programmable Gate Arrays (FPGAs) são componentes reconfiguráveis de hardware que encontraram grande sucesso comercial ao longo dos últimos anos em uma grande variedade de nichos de aplicação. Alta vazão de processamento, flexibilidade e tempo de projeto reduzido estão entre os principais atrativos desses dispositivos, e são essenciais para o seu sucesso comercial. Essas propriedades também são valiosas para sistemas críticos, que frequentemente enfrentam restrições severas de desempenho. Além disso, a possibilidade de reprogramação após implantação é relevante, uma vez que permite a adição de novas funcionalidades ou a correção de erros de projeto, estendendo a vida útil do sistema. Tais dispositivos, entretanto, dependem de grandes memórias para armazenar o *bitstream* de configuração, responsável por definir a função presente do FPGA. Assim, falhas afetando esta configuração são capazes de causar defeitos funcionais, sendo uma grande ameaça à confiabilidade. A forma mais tradicional de remover tais erros, isto é, *scrubbing* de configuração, consiste em periodicamente sobrescrever a memória com o seu conteúdo desejado. Entretanto, devido ao seu tamanho significativo e à banda de acesso limitada, *scrubbing* sofre de um longo tempo médio de reparo, e que está aumentando à medida que FPGAs ficam maiores e mais complexos a cada geração. Partições reconfiguráveis são úteis para reduzir este tempo, já que permitem a execução de um procedimento local de reparo na partição afetada. Para este propósito, mecanismos rápidos de detecção de erros são necessários para rapidamente disparar este *scrubbing* localizado e reduzir a latência de erro. Além disso, diagnóstico preciso é necessário para identificar a localização do erro dentro do espaço de endereçamento da configuração. Técnicas de redundância de grão fino têm o potencial de prover ambos, mas normalmente introduzem custos significativos devido à necessidade de numerosos verificadores de redundância. Neste trabalho, propomos uma técnica de detecção de erros de grão fino que utiliza recursos abundantes e subutilizados encontrados em FPGAs do estado da arte, especificamente as cadeias de propagação de vai-um. Assim, a técnica provê os principais benefícios da redundância de grão fino enquanto minimiza sua principal desvantagem. Reduções bastante significativas na latência de erro são atingíveis com a técnica proposta. Também é proposto um mecanismo heurístico para explorar o diagnóstico provido por técnicas desta natureza. Este mecanismo tem por objetivo identificar as localizações mais prováveis do erro na memória de configuração, baseado no diagnóstico de grão fino, e fazer uso dessa informação de forma a minimizar o tempo de reparo.

Palavras-chave: FPGA, detecção de erro, tempo médio de reparo.

ABSTRACT

Field Programmable Gate Arrays (FPGAs) are reconfigurable hardware components that have found great commercial success over the past years in a wide variety of application niches. High processing throughput, flexibility and reduced design time are among the main assets of such devices, and are essential to their commercial success. These features are also valuable for critical systems that often face stringent performance constraints. Furthermore, the possibility to perform post-deployment reprogramming is relevant, as it allows adding new functionalities or correcting design mistakes, extending the system lifetime. Such devices, however, rely on large memories to store the configuration bitstream, responsible for defining the current FPGA function. Thus, faults affecting this configuration are able to cause functional failures, posing a major dependability threat. The most traditional means to remove such errors, i.e., configuration scrubbing, consists in periodically overwriting the memory with its desired contents. However, due to its significant size and limited access bandwidth, scrubbing suffers from a long mean time to repair, and which is increasing as FPGAs get larger and more complex after each generation. Reconfigurable partitions are useful to reduce this time, as they allow performing a local repair procedure on the affected partition. For that purpose, fast error detection mechanisms are required, in order to quickly trigger this localized scrubbing and reduce error latency. Moreover, precise diagnosis is necessary to identify the error location within the configuration addressing space. Fine-grained redundancy techniques have the potential to provide both, but usually introduce significant costs due to the need of numerous redundancy checkers. In this work we propose a fine-grained error detection technique that makes use of abundant and underused resources found in state-of-the-art FPGAs, namely the carry propagation chains. Thereby, the technique provides the main benefits of fine-grained redundancy while minimizing its main drawback. Very significant reductions in error latency are attainable with the proposed approach. A heuristic mechanism to explore the diagnosis provided by techniques of this nature is also proposed. This mechanism aims at identifying the most likely error locations in the configuration memory, based on the fine-grained diagnosis, and to make use of this information in order to minimize the repair time of scrubbing.

Keywords: FPGA, error detection, mean time to repair.

1 INTRODUCTION

Over the past decades, the amount of transistors that can be placed within a single silicon die has grown exponentially, as foreseen by Moore's Law. These advances have fueled an increase in the amount and complexity of functionalities one can integrate in a single chip. Thereby, complex systems require only a small amount of Integrated Circuits (ICs) to carry out their functions, reducing project costs and time. On the other hand, the productivity of IC designers does not evolve at this same rate, leading to the phenomenon known as the *productivity gap* (ITRS, 2011). In other words, the amount of transistors made available by new manufacturing processes is so overwhelming that designers are unable to make the most efficient use of them. Furthermore, the time expected for the release of new products, known as *time-to-market*, becomes progressively shorter for most application niches, reducing the time available for design and further worsening the mentioned productivity gap.

To alleviate this problem, efficient design techniques that maximize the reuse of modules, reducing the burden on designers, are desirable. Designs with high regularity, i.e., that are mostly composed of replicas of smaller and simpler blocks, are therefore very effective to tackle the productivity gap. Regularity is also a valuable feature to reduce testing time and cost, as well as increasing the manufacture yield. The yield is the fraction of fabricated chips that are usable (and sellable), being an important metric to maintain profits. In this context, Field Programmable Gate Arrays (FPGAs) become a viable alternative that has found great commercial success in the past years.

FPGAs are reconfigurable devices that contain large amounts of generic logic and storage components, interconnected by flexible routing structures. On its own, an FPGA performs no useful operation, much like a processor without instructions to execute. This generic structure, however, can be programmed by uploading an appropriate configuration stream of bits in order to behave as virtually any digital circuit, provided it fits within the logic capacity limitations of the chosen device. Thus, FPGAs bring benefits of general purpose processors, as they are able to perform virtually any required function once properly programmed. And they also bring benefits of Application Specific Integrated Circuits (ASICs), as the function is computed by a dedicated circuit with potentially very high performance.

As FPGAs comprise thousands and even millions of identical generic logic, memory and routing blocks, they intrinsically present high regularity. From the manufacturers' point of view, this translates to the possibility of greatly simplifying the design of the different models within a same FPGA family or even of new families. Thus, manufacturers have been consistently able to release new products using very recent technologies, such as Xilinx's UltraScale architecture (XILINX, INC., 2013), expected reach a 16 nm feature size, and Altera's Stratix 10 (ALTERA CORPORATION, 2013), using a 14 nm process. From the users' perspective, an FPGA provides the ability to

quickly implement the desired circuit much faster than by manufacturing it as an ASIC, which is also a precious asset in times of pressing time-to-market restrictions.

Evidently, these benefits come with costs. Due to its generic and flexible nature, a circuit implemented in an FPGA is usually slower, larger and more power consuming than its ASIC counterpart. Nonetheless, fueled by the advances in semiconductor manufacturing technologies, FPGAs have shown a steady increase in their logic capacity and throughput in the past years. State-of-the-art FPGAs may include over 1.2 million lookup tables (LUTs) (XILINX, INC., 2012a), which are the basic building blocks for circuit logic in current devices. A better perspective on what this number means may be obtained by considering that a 32-bit MIPS-compatible *softcore*[1] processor requires approximately 2,750 LUTs to be implemented. Thus, such a device is able to include over 400 processors, an unthinkable amount when FPGAs were first created. The high throughput available in newer FPGA devices, coupled with the offered flexibility and fast prototyping capabilities mentioned, made FPGAs very successful in a variety of niches. Nowadays, FPGAs are used in military, automotive, data center and telecommunication applications, among many others (ALTERA CORPORATION, 2012), (XILINX, INC., 2012b).

The field programmability is also a very important feature in FPGAs, since it allows the addition of new functionalities after deployment, increasing the system lifetime. It also allows the correction of design mistakes with a much lower cost, when compared to ASICs. These possibilities are very interesting for *critical systems*, where efficient and high throughput computing may be required and a long lifetime is also desirable. Moreover, as these systems are frequently difficult to reach physically after deployment (e.g., space applications), the possibility to perform remote programming is of great relevance.

A system is deemed *critical* when its malfunction may have severe adverse effects. Such effects include, e.g., when human lives are at stake. The braking system of a car and the control of airplane wings are examples of systems considered critical as human lives are put in danger whenever they do not perform their operation properly. Other systems are considered critical for environmental causes, such as the control of an oil extraction platform. Finally, economic reasons may characterize a critical system. The data base of a bank or a high throughput router in a network backbone may bring severe losses to their owners and users if they fail to operate as expected.

Therefore, critical applications face stringent *dependability*[2] constraints that must be satisfied in order to minimize unwanted service failures. Unfortunately, the same advances in semiconductor manufacturing processes that have allowed the continued reduction in transistors' feature sizes also bring dependability threats. They increase the susceptibility of devices to several physical phenomena such as aging effects (e.g., negative bias temperature instability and hot carrier injection), which reduce the device lifetime. Radiation-induced single event effects (SEEs) also become more frequent, causing failures if not counteracted. Thus, efficient techniques able to tolerate hardware

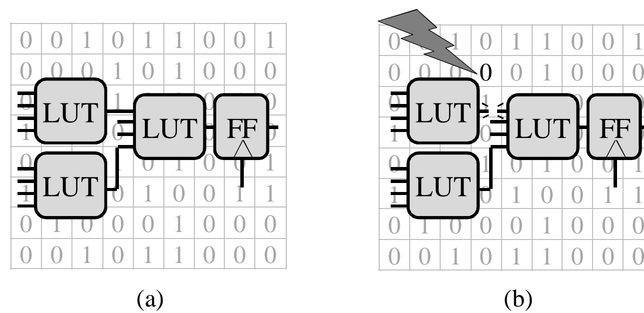
---

[1]A *softcore* is a processor implemented in the reconfigurable fabric of an FPGA, i.e., with LUTs, flip-flops, etc.

[2] Please see Appendix A or (AVIZIENIS, LAPRIE, RANDELL, *et al.*, 2004) and (PRADHAN, 1996) for a detailed description of dependability as well as other basic concepts and terminology of dependable systems.

faults are required to achieve the expected dependability levels. *Fault tolerance* techniques are traditionally based on some form of redundancy, which consists in performing a computation in a manner that allows checking its correctness. The most basic form of redundancy is repetition, either spatial or temporal. The regularity of FPGAs may also aid in the provisioning of redundancy, especially of the spatial nature, providing new possibilities for the designers of critical systems.

One of the first challenges faced when providing fault tolerance for FPGAs is to understand the effects that faults have on such devices, which differ in many aspects from those of traditional ASICs. As the device functionality is user-specified, it must be stored in a special configuration memory. Figure 1.1(a) shows a simple circuit and the configuration bits that describe its correct implementation. The configuration memory is frequently implemented with cells that are susceptible to radiation-induced single event upsets (SEUs). For example, SRAM cells, which are used for most high end devices, may have their stored value flipped if hit by an energetic particle. Thus, SEUs can modify the user circuit function, as shown in Figure 1.1(b), something that does not occur for a hardwired ASIC. The effects that a flipped bit in the configuration memory has on the user circuit are hard to predict, due to the complex effects that a configuration pattern unforeseen by the manufacturer may have on the fabric. Moreover, the lack of low level schematics available to users further increases the complexity of developing (and evaluating) fault tolerance techniques for FPGAs.



(a) (b)

Figure 1.1: Fault-free circuit and its associated configuration bits (a) and faulty circuit due to a configuration upset (b)

Furthermore, as the routing between the logic components is also configurable, faults in the configuration bitstream may affect it as well. This leads to soft errors that modify how the components are interconnected, which does not occur for dedicated hardwires. The example in Figure 1.1(b) shows the breaking of a routing wire caused by a bitflip. This property has severe implications on well-established fault tolerance techniques, since a single fault may even affect multiple independent user nets (LIMA, CARMICHAEL, FABULA, *et al.*, 2001).

It is also important to note that such faults remain in the system until explicitly removed, since in a traditional FPGA-based system the configuration memory is only loaded after power up. Thus, even when no actual permanent damage is caused to the storage cell, the user circuit may present erroneous behavior for a long time. To minimize this issue, one alternative is to periodically overwrite the configuration memory, in a procedure called *scrubbing* (CARMICHAEL, CAFFREY and SALAZAR, 2000). However, this approach may take a long time to reach the faulty bit, due to the large size of the configuration memory and the limited bandwidth available to access it. The mean time to repair (MTTR) is associated with how long it takes to traverse the entire configuration, and it is in the order milliseconds even for mid-range

FPGAs (CHAPMAN, 2010). Thus, systems that must meet real-time deadlines, for example, may find it insufficient to rely solely on configuration scrubbing, as the system is likely to be unavailable during the presence of the fault.

Real-time systems require an answer that is both correct and within the expected timeframe whenever possible, preferably even after the occurrence of a fault. Hence, for such systems, the ability to detect and correct an error may not be sufficient if it takes too long to recover and many deadlines are missed. Even more drastic effects can occur in control or Digital Signal Processing (DSP) systems where the next output is highly dependent on the previous states. Let us take, for example, a simple digital biquad filter with an 8 KHz sampling frequency and with a 200 Hz sawtooth input. Figure 1.2(a) shows the output of the filter without faults. Now let us assume that a fault occurs and it modifies one of the coefficients from 0.9 to 1.9, leading the filter to a potentially unstable behavior. If the error lasts for the time of one sample (125 µs), the output of the filter becomes that shown in Figure 1.2(b), almost identical to the correct one. On the other hand, if the error lasts for 7 ms (or 56 samples), which is a relatively short repair time for a state-of-the-art FPGA, it causes the output to become that in Figure 1.2(c). Note that it severely disrupts the output values far longer than the duration of the error, a behavior that is typical in circuits with logic feedback (PRATT, CAFFREY, GRAHAM, *et al.*, 2006). Therefore, faster means to detect and remove errors are required to allow the application of FPGAs in such systems.

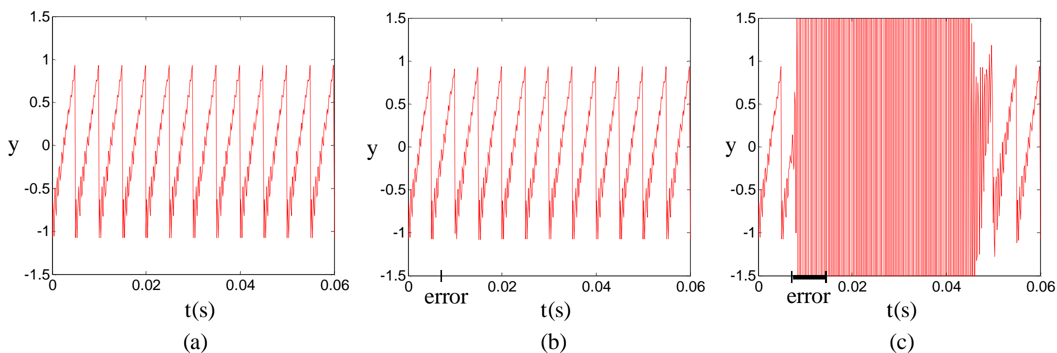


Figure 1.2: Fault-free output (a), one cycle error (b) and 56 cycles error (c). The markings on the *x* axes show the fault duration

Even in systems that do not necessarily have real-time constraints, fast error detection and removal can be crucial. A softcore processor that has its program counter moved to an unknown memory location, for example, may never recover if only scrubbing is used, since removing the configuration error does not restore its execution flow. *Checkpoint* and *rollback* procedures can be used for such cases. The former consists in periodically saving the system state, while the latter is the action of returning to one of these states once a fault is detected. The longer it takes to detect the occurrence of a fault, the farther its effects may have propagated throughout the system. Thus, it becomes more costly to maintain backup copies of the system state and to return to one of these checkpoints if the system takes long to detect an error. For example, if a processor is unable to detect the presence of an error before it propagates to the main memory, rolling back to a safe state becomes very costly or even unfeasible, as a backup of the entire memory is required. On the other hand, if an error is detected while it is restricted to the register file, or even before it reaches a register, the rollback is greatly simplified.

The specific properties of faults on FPGAs dictate that hardening techniques applied to such devices must be specifically tailored to cope with faults that affect the configuration memory. Moreover, it is crucial to keep in mind that the routing resources are not reliable and that configuration faults last for long periods of time if only straightforward scrubbing is applied. Therefore, other techniques must be applied in conjunction, ensuring that the probability of service failure is kept at an acceptable level. On the other hand, the system is still subject to the other constraints found in embedded devices, such as performance, power, energy and area. As fault tolerance techniques are traditionally based on redundancy, they will have a negative impact on at least some of these parameters. For example, Triple Modular Redundancy (TMR) consists in instantiating three replicas of the original design and performing a majority vote on their results. Thereby, it allows masking errors on a single module, but comes with area and power costs of approximately 200%. Such costs may be prohibitive for systems with stringent budget or power constraints. Error detection alternatives with lower costs, such as Dual Modular Redundancy (DMR) can become attractive in such situations. DMR consists in duplicating the components to be hardened and in comparing their outputs, thus presenting approximately 100% area and power costs. Reaching the required dependability levels, while meeting the remaining design constraints and minimizing costs, is a challenging task that frequently requires iterative fine tuning, as shown in Figure 1.3. Several iterations may be required until all constraints are met and costs are minimized, making it crucial to be able to quickly and efficiently modify the design, as well as to quickly and accurately evaluate the dependability and costs associated.

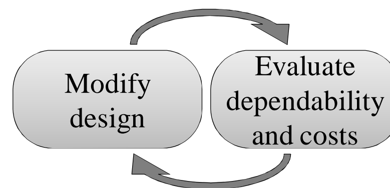


Figure 1.3: Iterative development cycle of dependable systems

Therefore, just as important as providing efficient means to cope with faults is evaluating the effectiveness of such means. A critical system specification should include values for metrics that determine the desired level of dependability. For example, “the system should not be offline for longer than 1 hour per year” or “the probability of a mission failure should be below 1 in a million” are the kind of specification that makes sense from the end user perspective. They must be appropriately translated into metrics that can be measured during design time, so that engineers are able to tune the techniques applied until the constraints are met with minimum costs.

Traditional approaches include the use of mathematical models to estimate reliability, which may become too complex or imprecise for large systems, and fault injection, which may take a long time to become statistically significant, especially when based on simulation software. Furthermore, due to the complex effects of faults in FPGAs and the unavailability of low level schematics to the users, the evaluation of fault tolerance techniques for FPGAs most frequently uses an actual device to perform experiments. Doing so provides more precise results in a reduced time, compared to simulation-based approaches. Experiments using radiation sources to stimulate the occurrence of errors in the FPGA are also common and are valuable especially to estimate the expected error rates for the system after deployment. They are important to

perform the mentioned conversion of the dependability expected by the users into metrics that are manageable by designers, as they allow estimating the fault rates that would be observed in the deployment environment.

1.1 Main goals and contributions

The goal of this work is to tackle one of the main challenges found when providing fault tolerance for FPGAs: the long time required to detect and remove a configuration memory error. As discussed previously, this long repair latency can cause missed deadlines in real-time systems and it increases the costs of performing checkpoint and rollback procedures. It also increases the probability of faults accumulating in the system, which may break techniques built upon the single-fault assumption. This thesis focuses on *soft errors*, i.e., those that do not permanently damage the device and that can be removed by overwriting the correct memory contents. Soft errors are a very critical concern for digital systems with deeply scaled transistors, and for FPGAs the configuration memory consists in a particular concern (FULLER, CAFFREY, SALAZAR, *et al.*, 2000). Other resources in FPGAs, such as internal block RAM memories, are also susceptible to SEUs. Faults on these components, however, have a similar behavior to that observed in ASICs and can be mitigated with the same established techniques, such as error correcting codes (ECCs). Thus, in this work, we focus on providing means to efficiently detect and remove soft errors from the configuration memory of FPGAs.

In order to quickly detect and remove an error, one cannot rely solely on periodically overwriting the contents of the configuration memory. As discussed previously, the time required to do so is long enough for the error to cause missed deadlines and to propagate throughout the system logic, making it very costly to return to a consistent state. In this work, thus, we propose the use of techniques that allow *fast error detection*, i.e., a short latency between fault occurrence and detection. Such techniques can be used to perform a *triggered scrubbing*, i.e., one that ensues once an error is effectively detected. Furthermore, it is preferable that an accurate fault location is provided. With *precise diagnosis*, one can perform localized removal procedures in a much shorter time than with global scrubbing. Furthermore, by repairing a smaller portion of the memory, one can save energy, as fewer memory accesses are necessary. The key concept explored in this work to achieve both fast error detection and precise diagnosis is the *granularity* of the error detection mechanism.

The granularity of a fault tolerance mechanism determines how the system is divided into modules for the sake of applying the technique. In other words, it determines how large and complex one allows each of these modules to be. Let us take DMR, one of the most traditional techniques, as an example. Figure 1.4(a) shows the basic coarse-grained approach. It allows detecting any single fault that occurs in one of the two copies and that propagates to the comparator. The latter condition, however, is frequently non-trivial, as a fault may be masked by circuit logic for long periods of time, depending on the nature of the function computed by the component. The granularity of the modules plays a significant role on this error detection latency. Note that small and simple components, as those shown in Figure 1.4(b), are more likely to quickly propagate an error to one of its outputs. This will in turn trigger the associated comparator, warning the system of the presence of an error. The approach in Figure 1.4(a), on the other hand, will only allow detecting the error once it has propagated to a primary output. For FPGAs, the possible granularities range from single LUTs up to entire complex modules, such as softcore processors.

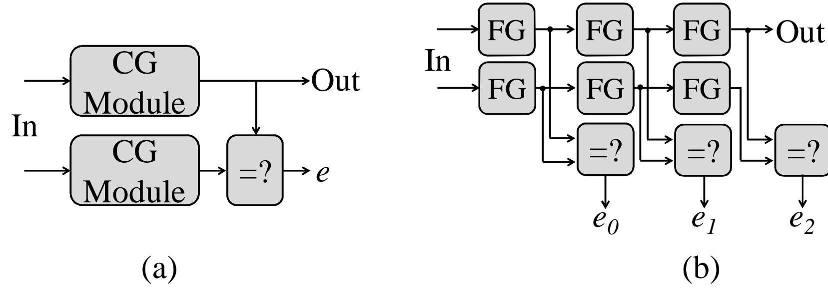


Figure 1.4: Coarse-grained (a) and fine-grained (b) DMR

The granularity also affects the precision with which the location of the detected error is known. As the comparator is only able to indicate that the output of the modules diverged, it cannot further specify the location of the error, which may be anywhere in the two modules and also in the comparator itself. Thus, the smaller the modules are, the more precise is the knowledge regarding fault location. In the example in Figure 1.4(a) a single bit of error detection is provided, which only allows indicating that a fault was detected, with no information about is location. On the other hand, in Figure 1.4(b) we are able to narrow the fault location down to a smaller portion of the system, depending on which signal was raised. Fine-grained error detection is, hence, an important feature to reduce the error removal time, as it provides improved diagnosis, allowing for localized repair procedures.

This thesis, thus, focuses on fine-grained error detection techniques for FPGAs and how they may be applied to achieve fast error detection and removal. One of the presented contributions is a technique that exploits abundant and underused resources found in state-of-the-art FPGAs to perform fine-grained comparison of replicated LUTs. As one of the main drawbacks of fine-grained fault tolerance is that it typically has an increased cost in area due to additional comparators (or voters), finding alternative mechanisms to implement them can help saving resources. Related works have even proposed the insertion of hardwired comparators in the FPGA fabric to minimize this area overhead (KYRIAKOULAKOS and PNEVMATIKATOS, 2009). Thus, the technique proposed here allows providing the benefits of fine-grained DMR while minimizing its main disadvantage, namely the increased area. It does not require any modification in the FPGA substrate, being applicable to devices that contain carry propagation chains, which are dedicated circuits for the efficient implementation of adders found in many state-of-the-art devices.

The use of very fine-grained diagnosis to perform accelerated error removal is also a challenging task, especially when one aims at doing so with acceptable costs. In this thesis, an approach to deal with very large error signatures (i.e., numerous individual error flags) is presented and evaluated. It relies on statistical information to build a relation between error signatures and the most likely error locations. It then identifies the optimum starting point of a scrubbing procedure aiming at minimizing the mean time required to actually reach the erroneous frame and correct it.

As discussed previously, evaluating the effectiveness of fault tolerance techniques for FPGAs is a demanding task as well. For this purpose, this work also presents a high speed and low cost fault injection platform that allows performing extensive experimental campaigns in a timely manner. The platform requires a single FPGA to carry out all the required functions, reducing the complexity and cost of the experimental setup, while avoiding off-chip accesses that reduce the injection rate.

Radiation experiments were also performed with a particle accelerator at the VESUVIO facility in ISIS, Rutherford Appleton Laboratories in Didcot, United Kingdom. The purpose of these experiments is twofold: first, asserting the effectiveness of the proposed techniques in an actual radiation environment; second, validating that the results attained with the fault injection tool accurately represent the effects of radiation on the device.

1.2 Outline

This work is structured as follows. Chapter 2 describes a standard FPGA architecture and discusses the main components found in current devices. It also presents the dependability threats faced by FPGA-based systems. Related works on fault tolerance techniques for FPGAs are discussed in chapter 3, while chapter 4 focuses on radiation experiments and fault injection platforms. Chapter 4 also presents the fault injection platform developed in the context of this work, describing its implementation and presenting its costs and advantages. In chapter 5 we present the techniques required to leverage fine-grained fault tolerance as a means for fast configuration error removal in FPGAs. We also present experimental results, including the expected failure rates for the proposed approaches. Chapter 6 presents the Scrubbing Unit Repositioning for Fast Error Repair (SURFER) technique and evaluates it regarding area and delay costs, as well as reductions on repair time. Conclusions drawn from the conducted work are presented in chapter 7. Some of the most promising future works envisioned at this time are also presented in chapter 7. Appendix A presents the taxonomy on dependable systems adopted in this work. It is suggested especially for readers not familiar with the nomenclature and basic concepts of this area. Appendix B discusses and evaluates the impact of non-random input stimuli on the figures reported in chapters 5 and 6. Appendix C presents additional experimental results exploring the design space offered by the heuristic algorithm proposed in chapter 6.

2 FPGAS AND THEIR DEPENDABILITY THREATS

The configurability of FPGAs is, at the same time, the key to their commercial success and the main source of area, delay and power costs. It must also be taken into account when a critical FPGA-based system is being designed, as it provides new possibilities, but also additional concerns. In this chapter we present the basic concepts of FPGA architectures in section 2.1. Then, in section 2.2 we present the main dependability threats for current FPGA devices.

2.1 FPGA Architecture basics

FPGAs are designed to be highly flexible, easily configurable and also to present high performance. A good source for further reading on the basic concepts of FPGA architecture is (KUON, TESSIER and ROSE, 2008). One of the most relevant aspects of such architectures, from both design and reliability perspectives, is the existence of a large configuration memory that stores the configuration bitstream. It is usually divided into *frames*, which are the smallest addressable units of the bitstream. The contents of this memory configure each and every element inside the device, including the behavior of each logic circuit and the routing between them. Thus, it stores the entire circuit functionality expected by the user, making its integrity a key requirement for the correct behavior of the system. Moreover, the configuration memory may be implemented with different manufacturing technologies, such as static RAM (SRAM) (LESEA, DRIMER, FABULA, *et al.*, 2005), flash (MICROSEMI CORPORATION, 2011) or antifuse (MICROSEMI CORPORATION, 2012), each with its advantages and drawbacks, to be discussed in more details section 2.2.

With regard to configuration memories, an important development of newer SRAM-based FPGAs is the possibility to perform dynamic partial reconfiguration. It consists in modifying the bitstream while the device operates normally, which has many applications. The most straightforward is having a dynamically reconfigurable area that has its behavior modified to assist the system in its current needs, avoiding the use of a larger area. Many other applications exist, especially when considering the requirements of fault tolerant systems. Devices also frequently provide special components to access the configuration from within the FPGA logic, allowing the creation of self-modifying designs and creating many new opportunities also for critical systems.

In order to be able to efficiently describe the required system function with the configuration memory, a flexible and powerful logic component is required. For that purpose, current FPGAs employ configurable lookup tables (LUTs) as their most basic functional building block. A *k*-input LUT is basically a multiplexer that selects one[3] out

[3] Some devices contain multiple output LUTs that can implement two or more functions that use the same input signals.

of 2^k binary values which are stored in memory cells. It can implement, thus, any boolean function with up to k inputs, with the configuration memory cells holding the desired function's truth table, as shown in Figure 2.1.

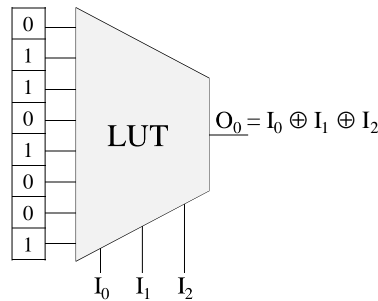


Figure 2.1: Example of a 3-input LUT implementing the XOR function

LUTs are combined, along with other basic components such as flip-flops, into small modules called configurable logic blocks (CLBs). Xilinx 7-Series CLBs, for example, are divided into two slices, with each slice containing four 6-input LUTs and eight flip-flops (XILINX, INC., 2012a). Similarly, in an Altera's Stratix V device, each CLB (called ALM – Adaptive Logic Module by the manufacturer) contains two adaptive LUT structures and four flip-flops (ALTERA CORPORATION, 2013). An adaptive LUT is an 8-input structure that can implement two 4-input functions, any 6-input function and certain 7-input ones. Furthermore, each CLB typically contains carry propagation chains that greatly simplify the implementation of adders or subtracters in the FPGA fabric. Inside each CLB there are also multiplexers able to realize interconnections between the LUTs, the carry chain circuitry and the flip-flops. Figure 2.2 shows the schematic of a Virtex 5 slice, which comprises four 6-input LUTs and

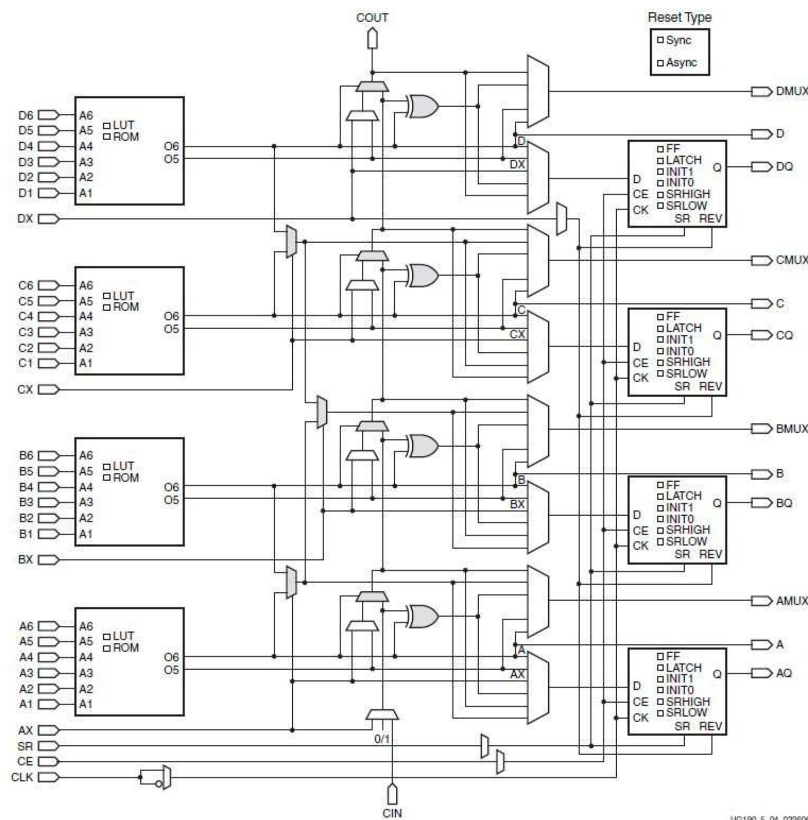


Figure 2.2: Schematic of a Virtex 5 slice (XILINX, INC., 2010)

four flip-flops. Each LUT can also be used as a read-only memory (ROM) and flip-flops have several configuration options, such as their initialization value and whether they are sensitive to clock level (latch) or edge (actual flip-flop). Each LUT can also be used to implement two different 5-input functions, as long as they share common inputs. The multiplexers and XOR gates in the center compose the carry chain circuit.

Aside from the CLBs, FPGAs contain dedicated blocks that implement some functions which are commonly required by the end users. For example, FPGAs are frequently used in digital signal processing (DSP) applications and such algorithms make heavy use of multiplication operations. Therefore, as multipliers are complex blocks that would require many LUTs to be implemented, FPGAs usually include hardwired DSP blocks. Each DSP block is able to compute a fixed point multiplication which may or may not be followed by an accumulation (XILINX, INC., 2010). Other more specific functions, such as dedicated transceivers and clock management units are also found in state-of-the-art FPGAs.

Another important feature found in FPGAs is the embedded memory blocks. The flip-flops found in the CLBs are very efficient to implement registers, such as counters or timing barriers in a pipelined design. However, when larger random access memories (RAMs) are required, flip-flops become inappropriate for two main reasons: 1) they are not so abundant in the device and are commonly heavily used as purpose-specific registers or pipeline barriers, leaving few spare resources and 2) their access is made by general purpose routing wires, which means that the multiplexing required to make a random access memory would have to be LUT-implemented, leading to further resource waste. For this reason, FPGAs usually include block RAMs (BRAMs), which are hardwired arrays of SRAM cells with dedicated access circuitry, much like a cache inside a processor chip. They can be used as small instruction memories for simple programs or as buffers for incoming or outgoing data frames of a specific application.

The communication with external components is done via FPGA pins, which are connected to the internal circuit using configurable input/output blocks (IOBs). These blocks can be configured to work as input, output or bidirectionally, according to different coding and electrical standards (XILINX, INC., 2010). Since they are the beginning and ending points of the system contained in the FPGA, these components play a significant role in reaching high reliability levels. Some techniques make use of pin redundancy to improve reliability (D'ANGELO, METRA, PASTORE, *et al.*, 1998), (LIMA, CARRO and REIS, 2003).

Finally, there is the great concern of interconnecting all the components of the FPGA: CLBs, BRAMs, dedicated hardwired logic and IOBs. The need for flexible routing resources, in the sense that they must be able to realize the interconnection topologies required by the user, and that are efficient in terms of area, delay and power, makes such resources a chief concern in FPGA design. The required flexibility imposes the need for a large amount of configuration bits associated with routing. In fact, the vast majority of the configuration bits actually configure how components are interconnected and not their behavior (XILINX, INC., 2011a), making such resources great concern regarding device reliability as well.

2.2 Dependability threats for FPGAs

The aggressive scaling of semiconductor devices, which leads to increased performance and lower energy consumption, frequently has adverse effects on dependability. The effects of energetic particles that may hit the silicon and disturb the

circuit operation, which are discussed in section 2.2.1, are of particular interest to this work. We also briefly discuss the effects of the aging of devices in section 2.2.2. Section 2.2.3 discusses the relation between dependability threats and the dependability metrics for FPGAs.

2.2.1 Radiation effects

With the reduction of transistors' dimensions and of supply voltage, the amount of electrical charge in a transistor is significantly reduced. Thus, the *critical charge*, i.e., the electrical charge that needs to be collected after a radiation event in order to induce an error, is also reduced. With reduced critical charge, the rate at which radiation-induced errors are observed tends to increase.

Several different particle types may induce errors on silicon devices, by generating energetic ions either directly or as a secondary effect. Alpha particles, neutrons, protons and heavy ions are among the most commonly reported sources of errors. Figure 2.3 shows the effects of an ion in the silicon. As the energetic ion passes through the device, it produces electron hole pairs (a), which are then rapidly collected in a funnel-like shape (b) and then more slowly over a long period of time by a diffusion process (c). If the amount of charge collected during this process exceeds the critical charge, then an error may occur. More in-depth discussions on the interaction of energetic particles and integrated circuits can be found at (SEXTON, 2003) and (BAUMANN, 2005). Of greater relevance to this work are the effects of such particles on a higher abstraction level, i.e., on circuit logic and on the service provided by the system. The current pulse induced by the particle may lead to several different *single event effects* (SEEs), especially when one considers the particular properties of FPGAs.

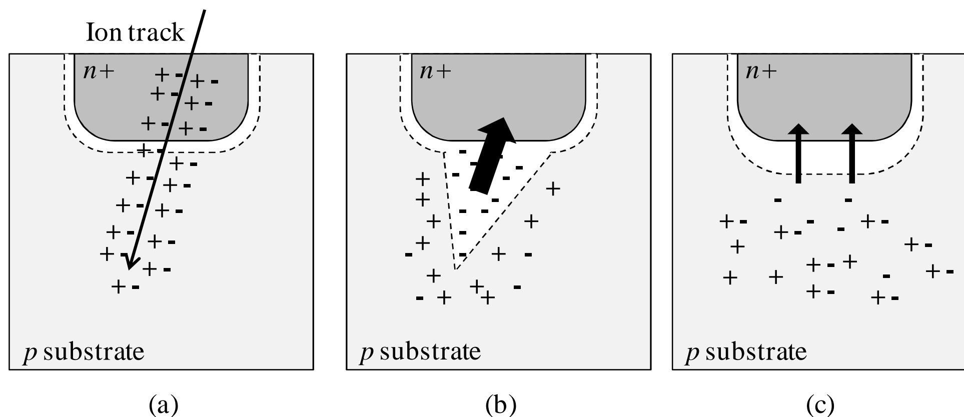


Figure 2.3: Effects of an energetic ion on a silicon device

2.2.1.1 Single Event Transient (SET)

If the affected part of the circuit contains combinational logic, the fault may manifest itself as a glitch on the output of a given logic gate. This phenomenon is referred to as a *single event transient* (SET).

Let us consider the simple circuit shown in Figure 2.4(a), in which the OR gate is subject to a fault that temporarily raises its output, n_0, to a logic '1'. Figure 2.4(b) shows the situation in which the SET propagates through the combinational logic and is stored in a register, leading to the occurrence of a soft error. However, it may not lead to such an error due to several reasons. First of all, it may be masked by circuit logic, as shown in Figure 2.4(c) (note that the value of i_2 has changed). Second, the storage cell may

have an "enable" input which, when deactivated, prevents the cell from reading its input value, as depicted in Figure 2.4(d). Finally, the fault may not be present during the occurrence of a latching window, as shown in Figure 2.4(e). The latching window is the period of time in which the storage cell updates its output value. It comprises the time when the clock is high (or low) for a latch or a narrow timeframe around the rising (or falling) edge of the clock for flip-flops (defined by the setup and hold times of the cell). With the reduction of the critical charge and with the increase of the operating frequency of newer devices, there is a possibility that the SET will last for more than a clock cycle. The impact of the long duration transient faults and several techniques to mitigate these effects are presented in (LISBÔA, 2009).

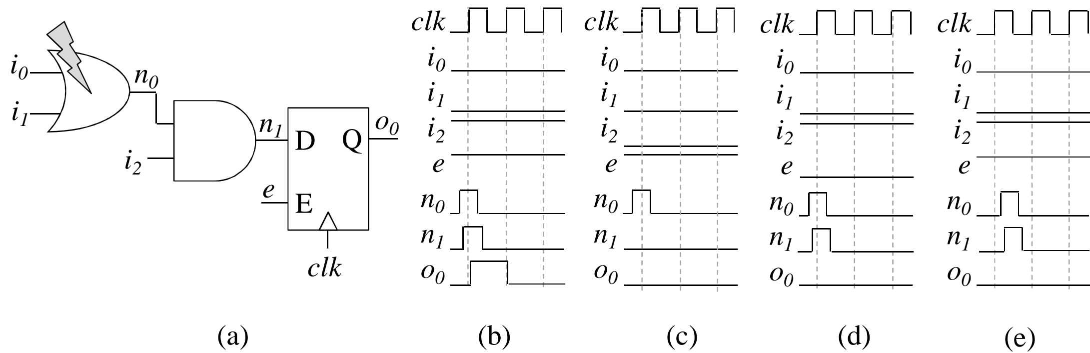


Figure 2.4: Different outcomes of a single event transient (SET)

Even though SETs on combinational logic should still be taken into consideration, they are far less common in FPGAs than in traditional ASICs (LESEA, DRIMER, FABULA, *et al.*, 2005). This is mainly due to the higher capacitance found in the routing of signals in an FPGA, which makes it much less likely that a particle will have sufficient energy to induce an error.

2.2.1.2 Single Event Upset (SEU)

An energetic particle may also directly hit a storage element and potentially alter the stored value. This phenomenon, called *single event upset* (SEU), most frequently affects a single memory cell leading to a *single bit upset* (SBU). Due to the greater integration and reduced dimensions of transistors, a single particle may also cause a *multiple bit upset* (MBU), which may have undesirable effects on systems that rely on error correcting codes (ECC), for example.

Figure 2.5 shows how a SEU occurs for a standard 6-transistor SRAM cell. Figure 2.5(a) shows the initial (correct) state of the cell, which is storing '1'. As shown in Figure 2.5(b), the particle creates a pulse in the output of one of the cross-coupled inverters that form the cell, similarly to a SET. In this case, however, if the pulse lasts long enough, it drives the input of the other inverter, which in turn reinforces the effect

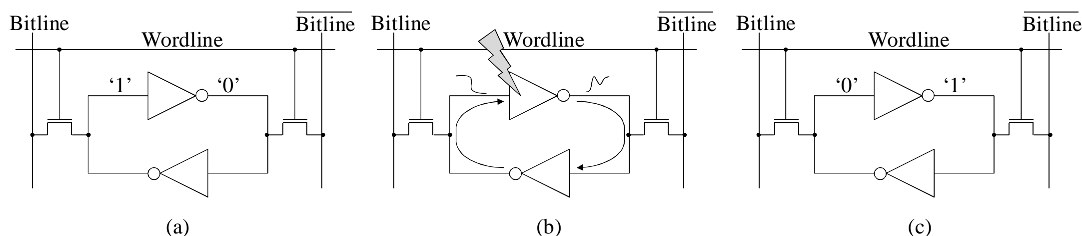


Figure 2.5: Single Event Upset on an SRAM cell

of the pulse on the first one through the feedback loop, altering the stored state, as shown in Figure 2.5(c).

SEUs are of particular interest for FPGAs due to their reliance on a configuration memory to store the desired circuit functionality. Thus, if such memory is subject to a SEU, the circuit function may be modified until the memory is rewritten at that position. As any configurable element is subject to this type of faults, both logic and routing resources may be affected. For instance, a LUT may have its function altered, meaning that it will yield a wrong output if the inputs choose that specific value. On the other hand, as discussed in section 2.1, the majority of the configuration bits are related to the routing resources. Therefore, faults affecting the configuration bitstream are likely to affect the interconnection between the logic elements. Furthermore, the effects of such faults on the user circuit behavior are not straightforward. For example, a single fault in a routing configuration bit may corrupt multiple nets of the user design, with undesirable effects on traditional error mitigation schemes (LIMA, CARMICHAEL, FABULA, *et al.*, 2001), (STERPONE and VIOLANTE, 2006). Such effects are a major concern for error detection and/or correction techniques for FPGAs and represent a major threat to the dependability of FPGA based systems.

The technology used to implement the configuration cells has significant impact on the expected SEU rate. Flash-based FPGAs present, along with the non-volatility of flashes, the advantage of a higher tolerance to radiation-induced SEUs. However, for the most aggressively scaled technologies, even flash memories may be subject to such faults (IROM, NGUYEN, HARBOE-SøRENSEN, *et al.*, 2011), creating the need for mitigation schemes even for such FPGAs. Furthermore, flash-based FPGAs present a much reduced logic capacity, in comparison to the SRAM-based ones. For example, the largest flash-based FPGA made available by Microsemi (formerly known as Actel) presents a logic capacity of 75,264 *VersaTiles* (each VersaTile can be configured to work as a D flip-flop or as a 3-input LUT) and 504Kb of BRAM (MICROSEMI CORPORATION, 2011). The largest SRAM-based FPGA made available by Xilinx, on the other hand, presents 1.22 million 6-input LUTs, over 45Mb of BRAM and 2.44 million flip-flops (XILINX, INC., 2012a).

Antifuse cells are also an alternative for the implementation of the configuration memory. This technology is highly resistant against radiation-induced faults, but presents the significant drawback of being programmable only once. This property prevents designers from including new functionalities or correcting design mistakes after the FPGA has been programmed. It also prevents the use of many techniques based on partial reconfiguration to avoid permanent faults. Moreover, antifuse-based FPGAs are also limited in terms of logic and embedded memory capacities, when compared to the SRAM ones. The largest antifuse-based FPGA made available by Microsemi contains 20,160 radiation-hardened flip-flops and 40,320 combinational cells (C-cell) (MICROSEMI CORPORATION, 2012). Each C-cell can implement over 4,000 5-input functions (which are not all possible 5-input functions).

The non-configuration memory cells (mainly BRAMs and flip-flops), which are usually implemented with SRAM cells, may also be subject to SEUs. However, as they are similar in purpose to the cells found in an ASIC, they may rely on the same traditional mitigation schemes, such as modular redundancy and ECCs. Such techniques should be used together with those targeting the configuration bitstream in order to provide comprehensive fault coverage.

2.2.1.3 Destructive Radiation Effects

Differently from what is observed with SETs and SEUs, there are also single event effects that cause permanent, destructive damage to the system. These effects are much rarer than SETs and SEUs, usually requiring more specific conditions and higher energy particles to occur. In (SEXTON, 2003) a detailed discussion is presented for each of the different mechanisms that may cause permanent damage, which is summarized here.

A s*ingle event latchup* (SEL) occurs when a particle activates parasitic *pnpn* bipolar structures found in CMOS devices. A low impedance path is created between supply voltage and ground, though which a high current flows. As this effect can only be removed through a power cycle, i.e., by completely powering off the circuit, there is a high probability that the current will permanently destroy the affected region. Thus, the latchup is not destructive by itself but the high current it creates may damage the device. A SEL is similar in effect to a *single event snapback* (SES). SESs, however, do not require the *pnpn* structure, occurring within a single transistor. Other destructive phenomena are more common in power transistors, such as *single event burnout* (SEB) and *single event gate rupture* (SEGR), being of less concern for FPGAs. A SEB is caused by heavy ions that trigger an avalanche effect, that in turn create large currents, potentially damaging the circuit. It may also cause a SEGR, a phenomenon in which the particle causes the dielectric that separates gate and channel to fail, also permanently damaging the transistor.

2.2.1.4 Cumulative Radiation Effects

The long term exposure to radiation may also have negative effects on the device dependability. These effects, thus, are not due to a single particle that hits the device, but due to the accumulated effects of radiation. The *total ionizing dose* (TID) over time may cause charges to be trapped in the field and gate oxides, the latter causing changes in the transistor's threshold voltage (V_{Th}) (SCHRIMPF, 2007). This in turn degrades the transistor's performance, until it eventually starts violating the timing constraints of the design, leading to intermittent and even permanent errors. Energetic particles may also cause displacement of atoms in the silicon lattice, also modifying its physical properties and potentially leading to intermittent or permanent faults. The cumulative radiation effects, due to their long term nature, are similar to aging effects, which are the subject of section 2.2.2.

2.2.2 Aging effects

The continued use of silicon devices leads to several physical phenomena that may, over time, cause intermittent or permanent faults. Such effects, collectively called *aging effects*, are the main responsible for the increased failure rate during the wear-out phase of semiconductor devices, as shown in the bathtub curve (Figure A**.**1 in Appendix A). In this section we briefly review the main physical sources of integrated circuits aging (BANSAL and RAO, 2011).

Among the most common sources of aging faults in recent technologies are those related to *bias temperature instability* (BTI). The voltage and temperature stress of gate dielectric (silicon oxide – SiO_2) may cause charges to become trapped in the interface between the silicon channel and the dielectric. BTI effects are divided into negative (NBTI), which affects PMOS transistors, and positive (PBTI), which affects NMOS transistors. Traditionally, due to the operating mode of CMOS gates, NBTI was of greater concern. However, with the introduction o hafnium oxide (HfO_2) in newer technology nodes, PBTI may also become a concern (BANSAL and RAO, 2011).

SRAM cells are also susceptible to aging effects. NBTI reduces the static noise margin of these cells and may increase the fault rates. Specifically for FPGAs, most of the PMOS transistors are used in the configuration cells, as the LUTs and routing resources are made mostly with NMOS pass transistors (MEHTA and DEHON, 2011). The effect of NBTI on SRAM cells can be relevant for FPGAs, leading to configuration cell instability (RAMAKRISHNAN, SURESH, VIJAYKRISHNAN, *et al.*, 2007). This may be observed as an increased SEU rate for FPGA devices that have been in use for a long time.

Hot carrier injection (HCI) also creates charge traps in the SiO_2-Si interface, but as a direct consequence of the high kinetic energy electrons that occur near the drain junction. These particles may also generate secondary particles through impact ionization, which may also become trapped in the oxide. As HCI affects NMOS transistors, in the case of FPGAs it may lead to faults in the routing pass transistors as well. Both HCI and BTI effects cause an increase in the threshold voltage, leading to slower device response and potential timing violations.

Time-dependent dielectric breakdown (TDDB) is another consequence of the traps that occur in the gate oxide. These traps may accumulate until a conductive path is formed between gate and channel, thus breaking the dielectric. The result of this breakdown is a sudden increase in gate current and consequently in power consumption.

2.2.3 Technology scaling and dependability metrics

As discussed herein, technology scaling increases the susceptibility of integrated circuits to many adverse phenomena that negatively affect dependability. The increased susceptibility to radiation SEEs, for example, reduces the MTBF throughout the entire lifetime of systems, while the accelerated aging anticipates the wear-out phase, where the failure rate starts to increase severely. Other phenomena, such as process variability and the increased complexity of testing and performing burn-in on manufactured devices may extend the infant mortality phase as well (LI, KIM, MINTARNO, *et al.*, 2009), if not counteracted. Thus, the failure rate function $z(t)$ gets higher at all phases of the devices' lifetime, modifying the bathtub curve described in section A.4 of Appendix A, as shown in Figure 2.6. Note that the plateau region that defines the useful life of the system gets shorter, due to early-life failures and the anticipated wear-out phase. Furthermore, it gets higher, due to the increased susceptibility to random soft errors, such as radiation induced SEUs.

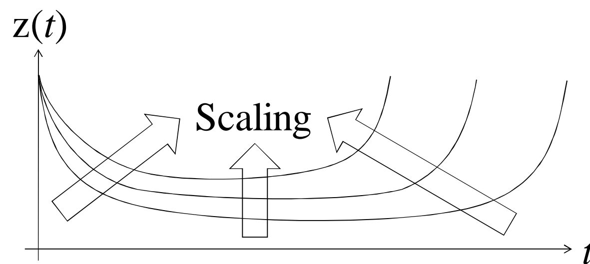


Figure 2.6: Technology scaling and the bathtub curve

Specifically for FPGAs, some efficient alternatives exist to mitigate manufacture defects (HATORI, SAKURAI, NOGAMI, *et al.*, 1993), (HOWARD, TYRRELL and ALLINSON, 1994) and process variability (GOLSHAN, KHAJEH, HOMAYOUN, *et al.*, 2011), (MEHTA and DEHON, 2011), usually relying on the fabric's regularity for

this purpose. Furthermore, due to FPGAs' improved energy efficiency when compared to general purpose processors, they tend to operate on lower temperatures, which in turn delays aging processes (MEHTA and DEHON, 2011). As discussed in section 2.2.2, one of the most critical effects of aging on FPGAs is the increased susceptibility to soft errors on the configuration cells (RAMAKRISHNAN, SURESH, VIJAYKRISHNAN, *et al.*, 2007). Thus, being able to efficiently mitigate the effects of SEUs on FPGAs is crucial during all phases of the devices' lifetime.

Another relevant side-effect of technology scaling observed in FPGAs is that, as a general rule, the bitstream size increases faster than the configuration interface speed, increasing the total programming time. As configuration scrubbing remains the main alternative to remove errors from the bitstream, the MTTR attainable with this approach tends to increase as technology advances. The probability of timing failures in real-time systems is therefore also increased, as well as the downtime of systems with availability constraints. A more in-depth analysis on configuration rates and bitstream sizes for different device families (and manufacturing technologies) is presented in section 3.2.

3 FAULT TOLERANCE TECHNIQUES FOR FPGAS

The many advantages and new concerns brought by FPGAs to the field of dependable systems have fueled a significant amount of works on fault tolerance techniques for such devices. Some techniques are adaptations of traditional redundancy schemes while others make explicit and intensive use of the underlying configuration memory to detect and possibly correct faults. Frequently, techniques make use of both approaches concurrently to provide a more comprehensive reliability solution. In section 3.1 we review the main works on this area, focusing on those that are based on redundancy schemes. Works that heavily exploit bitstream manipulation are discussed in section 3.2. Works that combine both are discussed in the category where the most significant contributions were made. The contributions of this thesis are contrasted with related works in section 3.3.

3.1 Techniques based on redundancy

Redundancy is the repetition of information or computation. While minimizing it is the goal in many situations (e.g., logic circuit minimization, data compression), it remains an essential tool to provide fault tolerance. Earlier works, such as (HATORI, SAKURAI, NOGAMI, *et al.*, 1993) and (HOWARD, TYRRELL and ALLINSON, 1994) propose to introduce redundancy in the form of spare resources that are used to improve the manufacture yield, i.e., the fraction of total produced chips that is usable. These resources are activated if, and only if, the manufacture test detects a fault, replacing the defective components in the chip. This approach allows maintaining the total logic capacity of the device and increases the probability that each chip is usable. Post-manufacture faults, however, were not a concern for these works. In the following years, several other works were concerned with improving the yield of FPGA architectures. Such techniques, however, are not the main concern of this work and we focus on techniques able to tolerate faults occurring at runtime.

In (MOJOLI, SALVI, SAMI, *et al.*, 1996) the importance of also mitigating post-manufacture faults is presented. The work focuses on permanent faults. The authors assume a non-reprogrammable FPGA, thus ignoring any possibility of run-time reconfiguration. The proposed technique makes use of modular redundancy, i.e., the replication logic blocks followed by comparison or voting, to provide fault tolerance.

Modular redundancy can be implemented with a varied number of replicas, with the most common variations being dual modular redundancy (DMR) and triple modular redundancy (TMR). As discussed previously, DMR allows the detection of single faults by comparing the outputs of two circuit copies, thereby signaling possible failures to other modules. TMR, on the other hand, allows masking single faults by voting the value given by the majority of three replicas. Figure 3.1(a) and Figure 3.1(b) show the basic DMR and TMR techniques, respectively. The reliability of the comparator or

voter component is also of critical concern for such systems. For this reason, redundancy is also frequently applied to them. Figure 3.1(c) shows a frequently used technique that triples the voters to avoid single points of failure (SPOFs). Modular redundancy may be applied with an arbitrary amount *N* of replicas (NMR) as well.

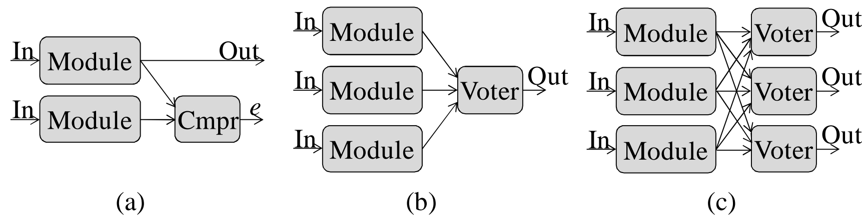


Figure 3.1: DMR (a), TMR (b) and TMR with tripled voters (c)

Mojoli, Salvi, Sami, *et al.* (1996) make use of four replicas of each module, thus allowing not only the correction of single faults but also the detection of double faults. This technique is also able to correct two faults if they do not happen simultaneously. The authors argue for the need of using placement constraints to ensure isolation between the different replicas, even though the complex faults associated with routing resources were not yet discussed. Furthermore, the authors present different granularities of implementation. Previous works, which aimed at improving the manufacture yield, required a very deep understanding of the FPGA fabric, which is not always available or convenient for the end user. Hence, in (MOJOLI, SALVI, SAMI, *et al.*, 1996) redundancy is implemented with the granularity of functions that can be handled by the user CAD tools. The presented results show that the technique has a high probability of providing correct service. However, they were obtained by means of high level equations that estimate the probability of faults affecting the control circuitry, without any experiments on a real device.

Standard TMR is applied in (FULLER, CAFFREY, SALAZAR, *et al.*, 2000). The authors present extensive radiation experiments on a Virtex device, showing that configuration upsets are a major concern in FPGAs. The addition of TMR combined with configuration scrubbing showed a 15× improvement on proton fluence-to-failure measurements. The authors also identified a critical component in the fabric of those devices, called “weak-keepers”. They were responsible for driving constant values that could be required by other components in the fabric and were susceptible to upsets that were not detectable in the configuration bitstream. Replacing the use of these circuits with other means to drive a constant value further improved the results. Very high availability measures (up to 99.9998%) could be achieved with the use of such techniques.

In (LIMA, CARMICHAEL, FABULA, *et al.*, 2001) the authors also make use of TMR on a Virtex device, which is evaluated through bitstream fault injection and confirmed through radiation ground testing. Experiments showed that single bit flips could lead to unexpected functional failures. Using proprietary tools, the source of such situations was identified: a single bit flip could connect signals from independent redundant modules, corrupting multiple nets and causing the voting scheme to fail. The relevance this property led to several further researches on how to mitigate this issue (STERPONE and VIOLANTE, 2006), (KASTENSMIDT, FILHO and CARRO, 2006). A Reliability-oriented place and Route Algorithm (RoRA) is presented in (STERPONE and VIOLANTE, 2006), with the purpose of deliberately avoiding the instantiation of routing paths that can lead to such faults, relying on TMR as redundancy mechanism. In

(KASTENSMIDT, FILHO and CARRO, 2006), the proposed technique consists in introducing redundant routing paths that are able to maintain a reliable connection between the components in the presence of faults. Both approaches show very significant reductions in the sensitivity to this kind of faults.

Besides spatial techniques, such as modular redundancy, time redundancy may also be used for fault tolerance. It consists in computing repeatedly or with additional delay in order to detect and/or mask errors. However, as all computations are performed on the same hardware, permanent faults are likely to repeatedly produce the same erroneous results, leading to undetectable situations and making time redundancy techniques more suitable for transient fault detection. This problem is of special concern for FPGAs, where faults affecting the bitstream linger until removed, which usually takes at least milliseconds, as discussed previously.

In (LIMA, CARRO and REIS, 2003) a technique combining time redundancy, DMR and TMR is proposed, aiming at reducing the hardware costs of TMR, especially concerning the usage of IO pins. Figure 3.2 shows the proposed technique. During normal operation, the combinational logic to be protected is duplicated in modules dr0 and dr1, following a standard DMR approach. Then, if the DMR comparator points out the occurrence of a fault, one extra cycle is used to determine which of the modules is faulty by means of time redundancy. In order to avoid the situations in which a bitstream error is not detected, the circuits operate with encoded inputs and decoded outputs. In the example in Figure 3.2, encoding consists in shifting the inputs 1 bit to the left, while decoding shifts the result two bits to the right. This approach allows stimulating the circuits differently, and potentially identifying which is the faulty one. Note, however, that not all circuits allow simple encoding and decoding to perform this kind of detection. Furthermore, there is no guarantee that the encoded inputs will stimulate the present fault, as they activate different paths in the circuit. Still, for some

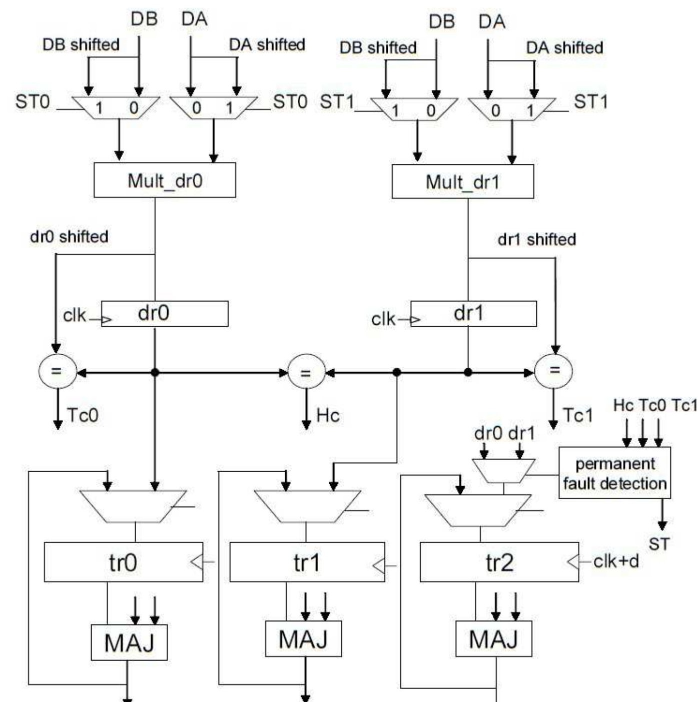


Figure 3.2: TMR, DMR and time-redundancy hybrid technique (LIMA, CARRO and REIS, 2003)

classes of circuits, such as arithmetic functions, it is possible to achieve relevant gains. The output flip-flops are tripled, with two of them receiving the outputs of dr0 and dr1 and the third one receiving the output of the module currently deemed fault-free. The presented results show area reductions and high fault coverage for a multiplier example, assuming a stuck-at fault model.

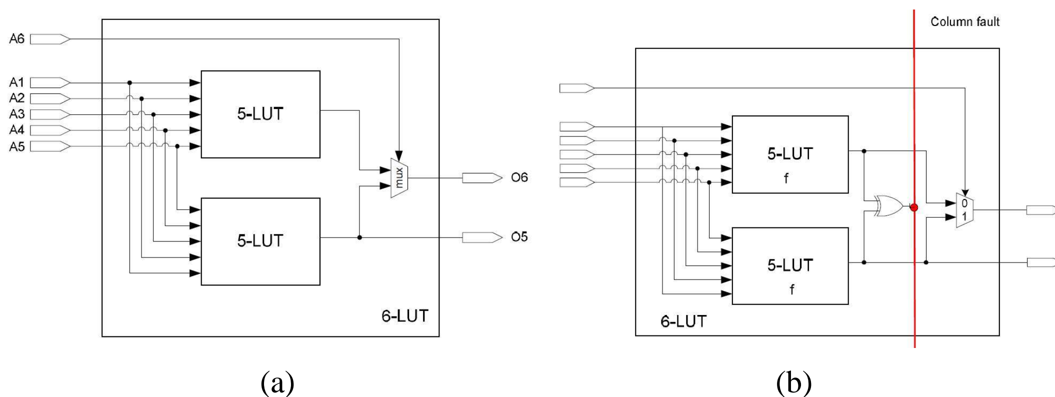
Another study concerning the use of TMR is presented in (KASTENSMIDT, STERPONE, CARRO, *et al.*, 2005). In this work, the authors evaluate the trade-offs regarding different granularities of application. Entire large modules can be voted at their primary outputs (i.e., a coarse granularity) or additional voters can be inserted at the output of simpler modules (i.e., a fine granularity). The main drawback of fine TMR granularities is that they cause additional area overhead, due to the extra voters, similarly to what occurs with DMR and the additional comparators. On the other hand, there is a reduced likelihood that faults will affect two redundant modules that share a same voter, thus potentially improving the fault coverage. Kastensmidt, Sterpone, Carro, *et al.* (2005) present a case study considering three different voting granularities and performing fault injection on the bitstream of a Xilinx Spartan device. The results showed that the intermediate granularity presents the best fault coverage (99.02%). All uncovered faults were associated with routing bits, as expected.

In (PRATT, CAFFREY, GRAHAM, *et al.*, 2006) an approach to reduce the costs of applying TMR to a design is presented. The observation done in that work is that the divergence of the system's output from the expected one may persist even after the configuration error is removed. This occurs mainly when the error modifies the behavior of a feedback structure of the design, i.e., a structure whose current state depends on its own previous state. For such parts of the design, simply removing the configuration error is not sufficient, as this operation does not restore the system state to a consistent one. The configuration bits that lead to this kind of situation are named persistent bits by the authors. The proposed approach is to apply TMR only to those parts of the design identified as feedback structures, aiming at reducing the amount of persistent bits. This is a valid approach for those applications that may accept short interruptions of service, but not a permanent one, such as audio or video decoding. The presented results show that a DSP application kernel could benefit from the technique more significantly than a synthetic design based on multiple linear feedback shift registers (LFSRs), which had more feedback loops that required TMR, reducing the gains of the technique.

TMR is also used in (GERICOTA, LEMOS, ALVES, *et al.*, 2007), but with a coarse granularity. The use of error detection to trigger configuration repair through partial scrubbing is evaluated. Error detection is performed by means of a scan chain that allows comparing internal signals of the TMR modules. Diagnosis information is not used to further divide the TMR modules, i.e., once an error is detected, the entire module is reconfigured. And, since coarse-grained TMR is used, large configuration areas, associated with large modules, must be repaired. Moreover, the time required to detect an error is associated with the time to sequentially compare the entire scan chain, thus presenting a linear dependence with the circuit size that limits scalability, similar to a global configuration readback mechanism.

Kyriakoulakos and Pnevmatikatos (2009) present another discussion regarding the different granularities of implementation of modular redundancy. They argue for the use of very fine grains, based on the fact that it is intuitively less likely that two faults will strike a single comparison or voting domain. A domain comprises the original module,

its replica(s) and the comparator or voter responsible for its operation. More specifically, they consider that each LUT is a module, thus making use of the finest grain available for an FPGA. Furthermore, they exploit the fact in Virtex 5 devices, each 6-input LUT can implement two different 5-input functions, as long as they share the same set of inputs, as shown in Figure 3.3(a). Thus, if the original LUT has only 5 inputs, it is possible to implement two replicas of the same function on a single 6-input LUT. They propose to modify the FPGA fabric to add a dedicated comparator to assess that the two 5-input functions are equal, thereby detecting any fault directly affecting one of them. Figure 3.3(b) shows the scheme, where a column fault wire is also added to indicate that a fault occurred at that column. Moreover, they synthesize the circuit to 5-input LUTs, so that every LUT in the design can be mapped to this structure. They present another modified substrate in which a voter is added, being able to realize a TMR scheme in which every three 6-input LUTs of the fabric implement two tripled functions.



(a) (b)

Figure 3.3: 6-input LUT built with two 5-input LUTs (a), and with the XOR gate added for comparison (b) (KYRIAKOULAKOS and PNEVMATIKATOS, 2009)

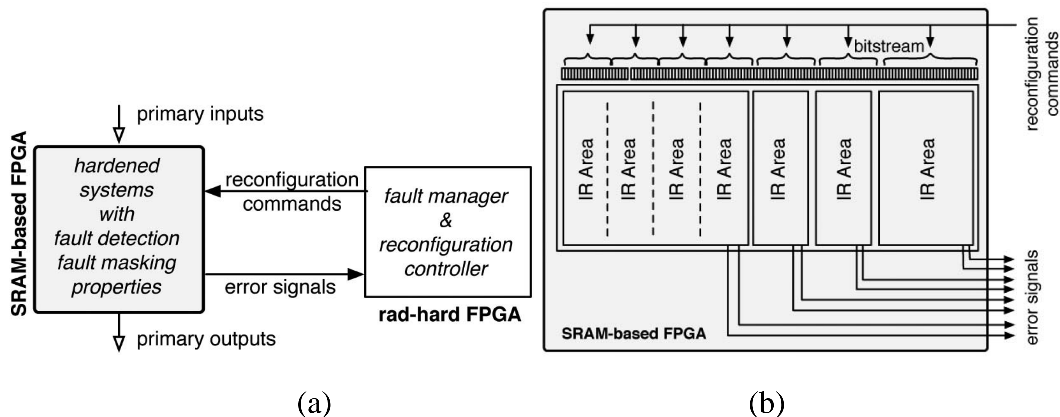
No fault injection experiments were presented in (KYRIAKOULAKOS and PNEVMATIKATOS, 2009). The faults associated with routing resources, which are likely to pose significant threats to this technique, are also left unchecked. The input signals of the replicas are the same, thus any fault affecting them is likely to remain undetected (even faults that affect a single net). Moreover, as a single voter is used for the TMR case, faults affecting the voter's output may also disrupt the technique. Additionally, no results on the area and latency costs of the proposed modifications were presented. Nonetheless, the significant reduction in costs (1.76 instead of at least 3 times for fine-grained DMR, measured by number of LUTs), points out an interesting research direction.

In (SHE and SAMUDRALA, 2009) the authors present an approach to minimize the costs of TMR similar to that of (PRATT, CAFFREY, GRAHAM, *et al.*, 2006). The idea is to apply redundancy selectively, only on those parts of the design that are deemed sensitive by a heuristic approach. The heuristic aims at maximizing the probability that a fault is masked, either by circuit logic or by the inserted TMR parts. It relies on input signal probabilities that state, for each primary input of the design, the likelihood of it being '1'. These probabilities are propagated throughout the circuit, considering the function computed by each LUT. Based on how likely it is for a LUT to propagate a fault (i.e., none of its other inputs has a dominant value), the "SEU sensitive probability" of each LUT is calculated. LUTs with a probability above a user-specified threshold are considered sensitive and receive TMR. The threshold probability can be

used to increase or decrease the amount of redundancy inserted. Furthermore, all LUTs computing NOT and XOR functions are considered sensitive, as they always propagate faults, as well as those that generate the primary outputs of the circuit. The proposed scheme was able to maintain high fault coverage with reduced overhead compared to TMR. The evaluation, however, made use of gate-level fault injection instead of using an actual FPGA, meaning that the employed fault model differs from what is observed in practice. For instance, the duration of the injected fault is not specified, and it is known that faults in FPGAs' bitstream are likely to linger for a long time (until explicitly removed).

Fine-grained redundancy schemes are also discussed in (NIKNAHAD, SANDER and BECKER, 2011). The authors present two approaches to tolerate massive fault scenarios. One is fine-grained TMR (FGTMR), which triples each LUT and each flip-flop and votes the output values with individual tripled voters. Thus, each LUT of the original design becomes 6 (3 replicas and 3 voters) and each flip-flop becomes 3, also requiring 3 additional LUTs for voting. Such high overheads make this technique only affordable for systems with extremely high resilience requirements as well as sufficient financial and power budgets available. The other approach uses Quadruple Force Decide Redundancy (QFDR), which requires each function to be computed four times and each time with duplicated inputs. Thus, aside from the cost of instantiating each LUT four times, they must use duplicated inputs. Assuming 6-inputs LUTs, every function with more than 3 inputs will have to be split, since one LUT will not accommodate its version with duplicated inputs. The experimental results show that the area overhead of the QFDR approach is even larger than that of FGTMR. Furthermore, the proposed techniques have a high impact on circuit latency, as they introduce several additional layers of logic to the design. These costs, however, are not measured by the authors. Still, both approaches are likely to be able to withstand very harsh scenarios due to their multiple faults masking capabilities.

As shown in (NIKNAHAD, SANDER and BECKER, 2011), very fine-grained approaches can introduce overwhelming costs. A design space exploration framework is presented in (BOLCHINI, MIELE and SANDIONIGI, 2011) to automatically identify the optimum partitioning granularity and hardening technique to satisfy a given set of system requirements. The framework takes into account costs and goals such as area overheads and average configuration time, with the purpose of accelerating repair. An external reconfiguration controller, shown in Figure 3.4(a), monitors detection signals that trigger scrubbing procedures. Reconfigurable partitions with individual



(a) (b)

Figure 3.4: FPGA system with external reconfiguration controller (a) and partition scrubbing mechanism (b) (BOLCHINI, MIELE and SANDIONIGI, 2011)

detection capabilities are defined to allow localized scrubbing, as shown in Figure 3.4(b) with the finest grains providing faster repair but increased area overhead. DMR and TMR variations are considered as potential redundancy techniques in the presented case study. Very significant MTTR reductions are achieved over global scrubbing (at least 80%). The problem of mapping the error indication signals, which can become numerous for fine granularities, is not addressed.

Psarakis and Apostolakis (2012) present a similar approach to exploit the benefits of fine-grained redundancy. The goal of their work is to avoid the costly maintenance of checkpoints to perform rollback procedures once an error is detected. The authors propose to divide a design into smaller modules, such as pipeline stages of a processor, and to apply a concurrent error detection technique to each of them individually. Thus, they are able to detect the presence of an error before it has propagated to structures such as the register file or the main memory and to indicate in which stage the error occurred. Furthermore, each module is implemented in an individual reconfigurable partition that includes spare resources to mitigate the effects of permanent faults. And as each module is smaller than the total design, they are able to perform a localized scrubbing to reduce the MTTR. When scrubbing is unable to restore the module's functionality, it is deemed as permanently faulty. In (PSARAKIS and APOSTOLAKIS, 2012), however, they assume the existence of additional pre-compiled configurations to activate the spare resources and to avoid using the faulty ones within the module, similarly to (LACH, MANGIONE-SMITH and POTKONJAK, 1998).

In their case study, they apply DMR to three modules of an OpenRISC processor, namely instruction decode, execute and the multiply-accumulate module. These modules represent only over 20% of the design, but the observed area overhead is 40.2%, showing that the proposed approach indeed causes significant costs due to the addition of reconfigurable partitions and reconfiguration controllers. The authors do not present any study regarding fault coverage. Also, the memory access and the writeback stages are not addressed in the paper. They are very critical to the proposed technique, since they have write access to the main memory and the register file, respectively, and faults on these modules can lead to the introduction of errors in these storage structures.

A set of modifications on the carry chain-based comparison technique presented in this thesis is proposed in (SONZA REORDA, STERPONE and ULLAH, 2013). The presented mechanism performs fine-grained comparison with carry propagation chains as well, and attempts to improve multiple-bit error detection properties. Since it is applicable only to LUTs that have up to 5 inputs, those with this property are separated from the rest and receive the technique, creating "multiple error regions". All error flags associated with one slice column in a frame row (i.e., 20 slices or 80 LUTs in height) are joined into a single bit that indicates the presence of an error on that column. These multiple error detection bits are then used to locate the error and perform local correction. Experimental results showed promising gains on repair time, which was reduced from 20.65 ms for global scrubbing to the order of tens to hundreds of microseconds for a set of benchmark circuits.

3.2 Techniques based on bitstream manipulation

As discussed previously, FPGAs contain a configuration memory that stores the circuit functionality and that is the basis of their flexibility. Especially for SRAM-based devices, this memory can be manipulated during runtime in order to provide fault tolerance. In this section, we briefly discuss the main approaches that heavily exploit

this feature. Features found in newer devices, such as partial reconfiguration, have further expanded the possibilities offered by such techniques.

Perhaps the most basic and intuitive approach is the configuration *readback* (CARMICHAEL, CAFFREY and SALAZAR, 2000), which consists in periodically reading the configuration and comparing it to a golden copy, which may be stored in a more reliable, off-chip medium. Doing so provides the ability to detect any fault striking the bitstream. Note, however, that such approach is not without costs. First, there is the energy consumed by the accesses performed to both memories (configuration and golden copy). There is also the financial cost of the golden copy itself. However, since the device needs to be programmed after power on, the system is likely to already possess some sort of non-volatile off-chip storage. Alternatively, data redundancy techniques may be employed, such as cyclic redundancy check (CRC) or checksum. Such approaches allow the detection of most errors with very high probability. They do not point, however, the error location. In (GOKHALE, GRAHAM, JOHNSON, *et al.*, 2004), a per-frame CRC calculation is performed, which allows locating the fault. The faulty frame can then be solely repaired.

Error repair is usually performed through *scrubbing* (CARMICHAEL, CAFFREY and SALAZAR, 2000). In its most basic form, instead of reading the bitstream in search of errors, it consists in directly overwriting the current configuration with its desired contents, regardless of the existence of errors. Errors may also be removed by means of error correcting codes (ECCs). For Virtex 5 devices, for example, each configuration frame, which comprises 1,312 bits, also contains 12 dedicated ECC bits that allow correcting a single bit flip or detecting double flips in that frame (XILINX, INC., 2011a). The device also includes a hardwired component that simplifies the verification of the correctness of the ECC embedded in each frame. Error correction and removal procedures must be performed by user-implemented circuitry. ECC-based approaches are interesting as they avoid the need to constantly access an off-chip memory to scrub the device.

Readback and scrubbing, even when based on ECC codes, suffer from long times to detect (or remove) an error, which significantly increase the achievable MTTR. The time required is associated with how long it takes to traverse the entire configuration memory, which determines the worst case detection/correction time. This time tends to get longer as devices get more complex and, consequently, with larger configuration memories. For the largest Virtex 7 device, for instance, it can be as high as 125 ms. Figure 3.5 shows the total scrub time for the largest device of each Xilinx family. Note that there is a sharp increase in the latest families, since no improvement in the configuration speed is observed since the Virtex 4 family, when the 100 MHz 32-bit SelectMAP programming interface was first introduced (XILINX, INC., 2009a). This development also explains the significant reduction from the scrubbing time observed in comparison to Virtex II Pro devices, which relied on a 50 MHz 8-bit interface (XILINX, INC., 2011b). Spartan devices show a similar trend, where the increase in the configuration speed is unable to compensate the increased configuration size.

The average time for the readback or scrubbing mechanism to reach the fault in the configuration is half of the worst case, assuming a uniform fault distribution over configuration frames. Even the average time may be too long for some applications, such as critical control loops. Furthermore, the circuit may not recover its functionality even after fault removal (PRATT, CAFFREY, GRAHAM, *et al.*, 2006). However, as such approaches are among the few able to effectively remove the fault from the

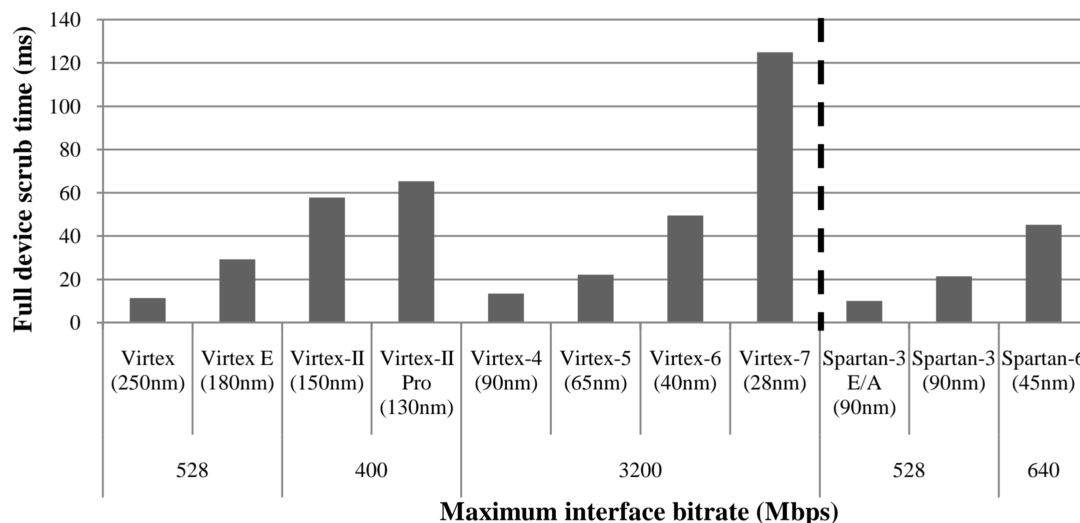


Figure 3.5: Total scrub time for the largest Xilinx FPGA of each family

bitstream, they are frequently used combined with redundancy approaches applied at the user circuit level, as in (FULLER, CAFFREY, SALAZAR, *et al.*, 2000), (LIMA, CARRO and REIS, 2003) and (SHE and SAMUDRALA, 2009).

Aside from the basic approaches of readback, scrubbing and those based on data redundancy codes, there are more complex schemes that periodically modify the bitstream in order to find permanent errors on the reconfigurable fabric. As a common drawback, these approaches usually present very high detection latencies, leading to a higher MTTR. They are, however, very efficient approaches to bypass the faulty components and eliminate them from the system, an important feature to avoid the accumulation of permanent faults in modular redundancy schemes, for example.

In (SHNIDMAN, MANGIONE-SMITH and POTKONJAK, 1997) the authors present a technique to perform on-line testing of the resources in the FPGA by means of partial reconfiguration. The proposed approach consists in leaving one of the columns of the FPGA offline, while its functionality is tested. The test is performed by exhaustively stimulating all the resources of the column in parallel. The correctness of the LUTs' outputs is assessed by comparing them with dedicated configuration memories that are included in the devices specifically for this purpose. Similarly, additional flip-flops are included to work as replicas of the original ones. After the test of that column is complete, the following one is tested, iteratively scanning the entire device. In order for the system to remain functional, a free column in the device computes the function of the column being tested.

Another approach that exploits partial reconfiguration to mitigate permanent faults is presented in (LACH, MANGIONE-SMITH and POTKONJAK, 1998). In this work, however, the authors focus on how to divide the design into clusters and to allocate spare resource to each of them. Alternative configurations for each cluster are pre-compiled, each using a different subset of the available resources and all of them maintaining the same interface with regard to the inputs and outputs of the cluster. This allows replacing the configuration of a cluster in order to avoid the use of a faulty resource without modifying the entire design, as the interfaces between each cluster are maintained. The technique had a very small impact in area (worst case 9.8%) and a reasonable impact on delay (from 14% to 45%). The reliability results, estiamted through probability equations, show that the technique is able to increase the reliability

considerably for different benchmark circuits. The equations, however, only consider faults in the CLBs, and not in the routing resources. This is likely to pose a main concern for the technique, as faults affecting the interfaces of the CLBs would make it impossible to use the pre-compiled configurations. Furthermore, the task of detecting the presence and the location of a fault remains an open question in (LACH, MANGIONE-SMITH and POTKONJAK, 1998). The introduction of error detection with the granularity demanded by the technique is likely to significantly increase the presented area overhead.

An approach that combines both scan-based testing (SHNIDMAN, MANGIONE-SMITH and POTKONJAK, 1997) and fault mitigation through spare resources (LACH, MANGIONE-SMITH and POTKONJAK, 1998) is the roving self test areas (STARs) technique, presented in (ABRAMOVICI, STROUD, HAMILTON, *et al.*, 1999) and (EMMERT, STROUD and ABRAMOVICI, 2007). The roving STARs technique also provides other benefits, such as being able to detect faults affecting the routing resources. Furthermore, it provides a very precise fault diagnosis, including the identification of the failure mode of a resource. Thus, a faulty resource may still be used if a function that is not affected by that particular fault can be mapped to it. It relies on vertical (V-STAR) and horizontal (H-STAR) areas to identify faults in wires of both directions. Figure 3.6 shows the proposed approach. The system state must be transferred from a column (row) whenever it is about to be tested, in order for the system to remain functional. Thus, the system function must be stopped for this operation to take place. Moreover, routing wires must cross the STARs to allow for communication between components on opposite sides, imposing delay overheads. Finally, the times to transfer configuration and state while roving the STARs lead to repair latencies in the order of seconds (estimated in 1.34s for an ORCA 2C15A, a very small device by current standards). These times are likely to be even greater for newer devices, as the size of configurations grew considerably more than the operating frequency of the programming interfaces.

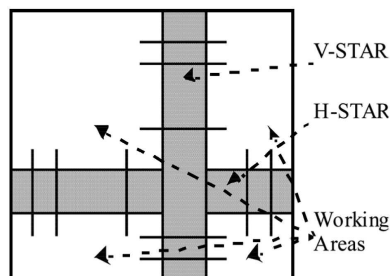


Figure 3.6: The roving STARs approach with horizontal (H-STAR) and vertical (V-STAR) testing areas (EMMERT, STROUD and ABRAMOVICI, 2007)

As discussed previously, one of the alternatives to reduce error detection and correction times is to use detection techniques with fine granularities. In order to effectively exploit the potential benefits, however, some challenges need to be addressed not only from a redundancy point of view, but also from a configuration perspective. In (STRAKA, KASTIL and KOTASEK, 2010) the authors focus providing a generic controller to perform local reconfiguration of modules. This module receives one error signaling bit from each of the reconfigurable partitions, as depicted in Figure 3.7, and upon detection accesses a table containing the initial and final addresses of the faulty module, which is then scrubbed to remove soft errors. If the error persists after reconfiguration, the fault is considered permanent. Thus, the proposed controller allows

exploring the fault indication bits provided by each module to reduce the MTTR and the probability of timing failures. The proposed approach does not specify a granularity of operation, but is intrinsically limited to the finest grain available for dynamically reconfigurable partitions. Furthermore, it requires the use of such partitions to identify beforehand the configuration frame addresses associated with each module, which imposes area and delay costs. The presented results concern only frequency of operation and area occupied, not evaluating the possible gains on reliability or repair time.

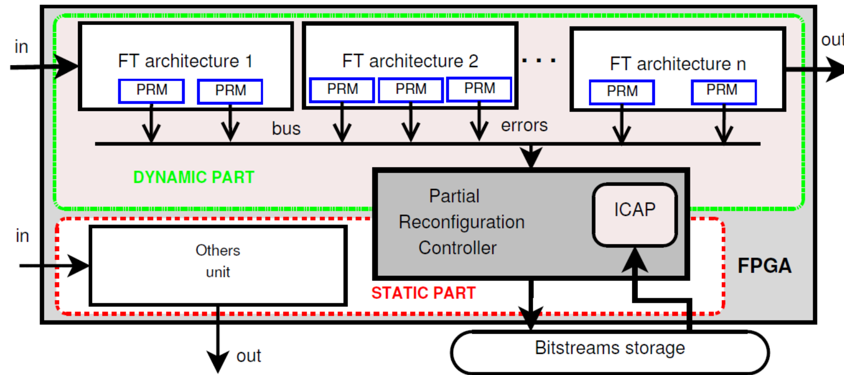


Figure 3.7: System with partial reconfiguration controller and multiple error signals (STRAKA, KASTIL and KOTASEK, 2010)

3.3 Contributions of this thesis

Fine-grained redundancy has been explored in previous works with many different goals. In (KASTENSMIDT, STERPONE, CARRO, *et al.*, 2005), different TMR granularities are evaluated in order to minimize the probability of a single fault affecting multiple redundancy domains. In (NIKNAHAD, SANDER and BECKER, 2011), the goal is to withstand very harsh environments, exploiting the fact that each individual TMR domain can mask the presence of one error. Other works make use of fine-grained partial fault tolerance (PRATT, CAFFREY, GRAHAM, *et al.*, 2006) (SHE and SAMUDRALA, 2009) to reduce the costs of full redundancy, compromising fault coverage to reduce area costs.

In this work, we make use of fine-grained redundancy with the main purpose of reducing repair time. When triggered repair procedures are used, which is the case in here and in related works such as (STRAKA, KASTIL and KOTASEK, 2010), (BOLCHINI, MIELE and SANDIONIGI, 2011), (PSARAKIS and APOSTOLAKIS, 2012) and (SONZA REORDA, STERPONE and ULLAH, 2013), two features of fine-grained redundancy become particularly valuable: reduced detection latency and precise diagnosis. The intuitive property is that the finest redundancy grains have the greatest potential to minimize the MTTR, since smaller modules have reduced masking probabilities and fewer associated configuration bits. But they also introduce the greatest overheads, which stem from the need of additional comparators or voters. Works such as (NIKNAHAD, SANDER and BECKER, 2011) present area costs that surpass 6 times, while others try to avoid them with modifications in the underlying fabric (KYRIAKOULAKOS and PNEVMATIKATOS, 2009). We propose a very fine-grained error detection mechanism that relies on the carry propagation circuitry found in current FPGAs to implement comparators. Since such resources are frequently underused, as will be shown in the experiments discussed in chapter 5, they are likely to not conflict excessively with the remainder of the design. Thus, a fine-grained

redundancy mechanism with manageable area costs can be devised for unmodified commercial FPGAs. Due to its very fine granularity, the technique is able to detect errors in a reduced timeframe when compared to coarser approaches. Reducing this error latency is important not only to minimize system downtime but to also to avoid the accumulation of errors.

The fact that fine-grained diagnosis can be used to perform a localized repair procedure is also explored in this work. In (LACH, MANGIONE-SMITH and POTKONJAK, 1998), (STRAKA, KASTIL and KOTASEK, 2010), (BOLCHINI, MIELE and SANDIONIGI, 2011) and (PSARAKIS and APOSTOLAKIS, 2012) reconfigurable partitions are used to delimit the minimum scrubbed area. Each partition has an individual error detection mechanism, which allows the use of partial scrubbing on a reduced range of the configuration memory. However, the definition of partitions has costs: they have a fixed interface with other modules, which restrict placement and routing choices. Moreover, fragmentation due to unused components within the partition space can also lead to wasted resources. Such costs tend to become more significant as smaller partitions are defined. If the most significant gains are desired, therefore, very small partitions have to be used, introducing additional costs.

In (SONZA REORDA, STERPONE and ULLAH, 2013) a modified version of the carry chain-based comparison mechanism presented in this work is used to detect errors as well. The fine-grained diagnosis is also used to accelerate repair, and differently from the other mentioned works, the minimum scrubbed unit is independent from reconfigurable partitions. The technique proposed in here also avoids the use of reconfigurable partitions as minimum scrubbed area, but with a different approach: we exploit the fact that the scrubbing does not necessarily start at the first configuration frame of a partition, and that starting it closer to the actual error location can significantly reduce repair time. As a result, partitions can be defined by designers as they see fit, following the recommended practices for design modularization. Moreover, when fine grains are used, the amount of error signals can increase quickly, and their mapping to error locations can be challenging, as will be discussed in chapter 6. We aim at providing a scalable mechanism able to handle numerous error detection bits and to extract useful information from them in a low cost, fast and reliable manner.

4 FAULT INJECTION FOR FPGAS

Fault injection is an important and frequently used means to evaluate the dependability of systems. Specifically for FPGAs, several challenges and opportunities are found. In this chapter we first briefly discuss the basic aspects of fault injection techniques, such as the desired features and the basic approaches. Section 4.1 presents this discussion. In section 4.2 we present the particularities found in FPGA-based systems, as well as the main fault injection platforms available in the literature. The fault injection platform developed in this work is detailed in section 4.3.

4.1 Fault injection Basics

Fault injection consists in artificially inserting faults in a system or in a system model in order to evaluate its response to faults of a particular model. Thus, in order to do so, a fault injection platform typically requires the basic components shown in Figure 4.1. First, an instance or model of the circuit being evaluated, often called circuit or device under test (CUT or DUT, respectively), is required. The input generator unit applies input vectors that stimulate the operation of the CUT. At some point during or before the execution the fault injector disturbs the circuit behavior according to the specified fault model. The output vectors produced by the CUT must be evaluated, either by making use of a *golden copy*, i.e., a copy of the CUT that is kept free of faults, or by some other means to determine whether they are correct or not. This task is carried out by a fault classification unit, which determines what the effect of the injected fault on circuit behavior was, i.e., if it caused a functional failure and/or if it was detected by some sort of detection mechanism.

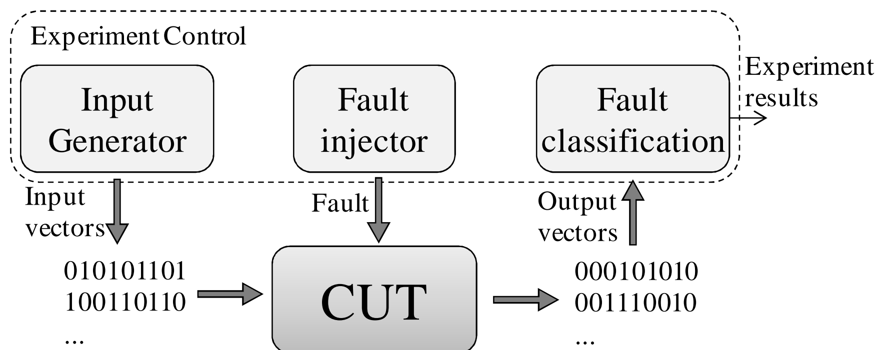


Figure 4.1: Basic components of a fault injection platform

Fault injectors are frequently used to measure dependability metrics such as reliability, availability and the fault coverage of a given fault tolerance technique. As the results of fault injection campaigns are used to guide the following steps of a project, the techniques and platforms used to perform such experiments play a critical

role in the overall project costs, time and quality. Thus, several assets are expected from such systems, from which we highlight the following:

- *Accuracy*: a fault injector should be able to accurately mimic the effects of the faults that the system will be subject to after deployed. This includes the definition of an appropriate fault model and the correct application of it to the system instance or model being evaluated. The amount of faults applied also plays an important role in the overall accuracy, as it should be statistically significant and also able to identify unexpected faulty behaviors that may occur under particular circumstances.
- *Injection rate*: as discussed, injecting a large amount of faults is important to achieve accurate results. Thus, being able to inject many faults in a short time interval is crucial to provide results quickly to designers, reducing the overall design time. As the system must execute for some time after each injection in order to observe its response to the fault, achieving high injection rates may be a challenging task. The achievable injection rate is closely related to the abstraction level adopted and to the complexity of the system being evaluated.
- *Flexibility*: designers frequently must inject faults following different models, such as permanent and transient faults that may strike the system. Thus, a fault injection platform which provides flexibility, allowing the modification of fault models or other simulation parameters, such as initial and termination conditions, allows a more comprehensive assessment of the system's dependability.
- *Controllability and reproducibility*: another important asset is that of being able to inject faults at specific areas or components of the systems, which have already been identified as critical, for example. Furthermore, it is frequently important to reproduce an experiment, in order to evaluate if a system modification was able to improve the dependability. Thus, being able to choose exactly when, where and how to inject a fault is a relevant feature of a fault injection mechanism.
- *Cost*: finally, but not less relevant, is the cost of the fault injection experiments. Those requiring expensive components, powerful simulation mainframes or several instances of the system under test may be unattractive for projects with a lower budget.

Providing all of the above advantages at once is a complex task, as most approaches present trade-off situations. Faults may be injected into a system by several different means, depending on the current stage of the project and on the desired properties from the experimental flow.

For the early stages of a project, when a hardware prototype is not yet available, a simulation model may be used, such as done by the MEFISTO tool (JENN, ARLAT, RIMEN, *et al.*, 1994). The MEFISTO tool simulates a VHDL model of the system to perform the required fault injection experiments. As the VHDL language allows the use of different abstraction levels, it is possible to achieve higher accuracy with a reduced injection rate when using a structural description, or the opposite when a behavioral description is used. For the case study, a very simple 32-bit processor was used, described in the two mentioned abstraction levels, showing a 3.2 times difference is simulation time. The injections may be performed using commands of the simulation software that artificially modify the values of the signals and variables of the system. Alternatively, *saboteur* components or modified versions of system modules, called

mutants, may be used to inject faults. Such approaches allow the use of very complex fault models. As is typical of simulation-based fault injection techniques, the MEFISTO tool presents high flexibility, controllability and reproducibility, as well as a low cost, as no hardware prototype or special components are required. On the other hand, the achievable injection rate and accuracy are conflicting properties, which are tuned by the chosen abstraction level and the amount of faults to be injected. Furthermore, even when working on the lowest abstraction levels allowed by VHDL, complex electrical phenomena, such as cross talking wires and overheating, are not detectable. Simulation-based fault injection platforms are presented in several other works, such as (CHA, RUDNICK, PATEL, *et al.*, 1996), (AIDEMARK, VINTER, FOLKESSON, *et al.*, 2001) and (KAMMLER, GUAN, ASCHEID, *et al.*, 2009).

FPGAs have brought an interesting opportunity for fault injection campaigns, even those aiming at ASIC designs. Since FPGAs are easily configurable, they can be used to emulate an ASIC design with a significantly increased speed compared to that of software simulators. The emulated circuit can then be instrumented with additional hardware in order to perform fault injection according to the specified fault model and to assess the effects of each fault. Thus, FPGAs are able to greatly enhance the injection rates of simulation based approaches. Note, however, that such approaches are limited by the size circuit that fits the available FPGA device. Works such as (CIVERA, MACCHIARULO, REBAUDENGO, *et al.*, 2002), (DE ANDRES, RUIZ, GIL, *et al.*, 2008) and the FT-UNSHADES platform (AGUIRRE, TOMBS, MUOZ, *et al.*, 2007) are examples of fault injection platforms that use FPGAs to increase the injection rates.

Once designers are able to make use of a system prototype, several other approaches become available, which allow overcoming some of the shortcomings found in simulation-based techniques. In (HSUEH, TSAI and IYER, 1997) the basic aspects of such approaches are presented, classified into hardware and software fault injection. Hardware fault injection techniques are further divided into injection with contact and injection without contact. Software-based approaches are divided into compile-time and runtime injection.

Hardware fault injection without contact typically makes use of electromagnetic fields or beams of energetic particles to interfere with the device's operation. Such approaches are valuable to measure not only the effectiveness of fault tolerance techniques but also to characterize manufacturing processes regarding their sensibility to the chosen source of faults. Thus, they present very high accuracy when the goal is to evaluate the effects to such physical phenomena, being an important step to validate systems that are to be used in harsh environments, such as space or industrial applications. However, the achievable injection rate is very limited, usually being orders of magnitude lower than that of simulation-based approaches, for example. Also, the only fault model to be addressed is that of the chosen physical source, limiting the flexibility. The controllability and reproducibility are also poor, as there is little choice regarding which parts of the system will be affected. Finally, the costs associated with such experiments may be high, due to the potentially expensive equipments that are required.

Hardware fault injection with contact consists in the use of active probes or sockets that intercept the communication between the circuit and its board. Thus, only the values available at the external pins are accessible and/or modifiable. As the actual system is running, such techniques have a possibility of presenting a higher accuracy and injection rates than simulation-based approaches. On the other hand, the flexibility

is reduced, as only those faults applicable at the external pins are injectable, limiting the possible fault models and also jeopardizing the accuracy. The use of scan chains may improve such properties. Since engineers control precisely when and where faults are injected, such techniques present good controllability and reproducibility. The cost of these mechanisms may be high when very sensitive probes and sockets are required.

Compile-time software fault injection consists in modifying the software prior to execution, either at the source code or at the executable binary. The main advantage of this approach is the reduced cost, especially due to its simplicity that greatly reduces the required engineering effort. They are useful to emulate permanent faults, as the modifications embedded in the code linger throughout the entire execution. On the other hand, the approach presents a low flexibility, being limited to those faults that can be mimicked with static modifications in the code. Conversely, runtime software fault injection is triggered by timers or exceptions and is able to model transient faults more accurately than compile-time approaches. Common to both software fault injection techniques are the limitations in the fault model, as not all parts of the hardware are reachable from the software's perspective. Furthermore, the accuracy may be threatened by the intrusiveness of the injection and evaluation mechanisms. Also, they are only applicable to processor-like systems, since the existence of software is required. Finally, the controllability and reproducibility of both approaches are related to how much influence those parts outside the designers' control have on the experiments. For example, the scheduler of the operating system may heavily modify the results of a fault injection campaign, especially for multi-threaded applications.

4.2 Fault injection for FPGA-based systems

The techniques discussed in section 4.1 were thought as means to evaluate the dependability of integrated circuits in general, regardless of whether they are FPGAs or not. Therefore, some of them are not directly applicable for many FPGA-based systems. For example, since software techniques require the existence of software in the first place, many FPGA systems lie out of scope, as they do not necessarily contain a processor.

Furthermore, simulation-based approaches require in-depth knowledge of how the system works and, in order for them to achieve accurate results, the system should be simulated in a low abstraction level. However, low level schematics of FPGA devices are rarely available to the end-users, as this is not in the best interest of FPGA manufacturers. This makes it nearly infeasible to evaluate the impact of SEUs affecting the configuration memory by means of simulation. Moreover, the complex scenarios of configurations unexpected by the manufacturer (such as two independent wires being in short circuit) would have to be modeled in a very low abstraction level, such as using an electrical simulator, in order for their outcome to be precisely determined. Working on such low levels brings an enormous computational burden, especially when an entire complex system needs to be simulated. Thus, it is nearly mandatory to use an actual FPGA device to perform fault injection with satisfactory accuracy, especially when the impact of SEUs in the configuration memory is to be evaluated. The use of hardware fault injection techniques described in section 4.1 becomes, hence, not only very attractive for FPGA devices but also one of the few remaining alternatives.

4.2.1 Radiation experiments

Experiments with particle accelerators are an important step to evaluate the impact of radiation on these devices. Some of the works discussed in chapter 3 conducted such

experiments (FULLER, CAFFREY, SALAZAR, *et al.*, 2000), (LIMA, CARMICHAEL, FABULA, *et al.*, 2001) to measure the reliability of circuits or to validate results achieved with other fault injection approaches. Such works measure the *dynamic* cross-section, which is related to the susceptibility to the effects of radiation on the user design atop the fabric (FULLER, CAFFREY, SALAZAR, *et al.*, 2000). It is an important metric as it measures the effectiveness of any fault tolerance technique that may be in use and is valuable to estimate the MTBF.

Fuller, Caffrey, Salazar, *et al.* (2000) also report *static* cross-sections for the evaluated Virtex FPGA. This measurement is performed by reading back the device's configuration memory and comparing it to the expected value. It is, thus, not related to the user circuit currently implemented, being an important metric to characterize the manufacturing process and the cell design employed with regard to SEU susceptibility. In (LESEA, DRIMER, FABULA, *et al.*, 2005), a series of experiments called Rosetta attempts to quantify the amount of faults to be observed in Xilinx FPGAs. Boards consisting of a hundred devices are constantly monitored and left at different places and altitudes. Accelerated experiments were also performed at the Los Alamos Neutron Science Center (LANSCE). Quarterly updated results of the Rosetta experiment can be found at (XILINX, INC., 2012c). Figure 4.2 shows the neutron cross-sections per configuration bit measured at LANSCE for different FPGA families.

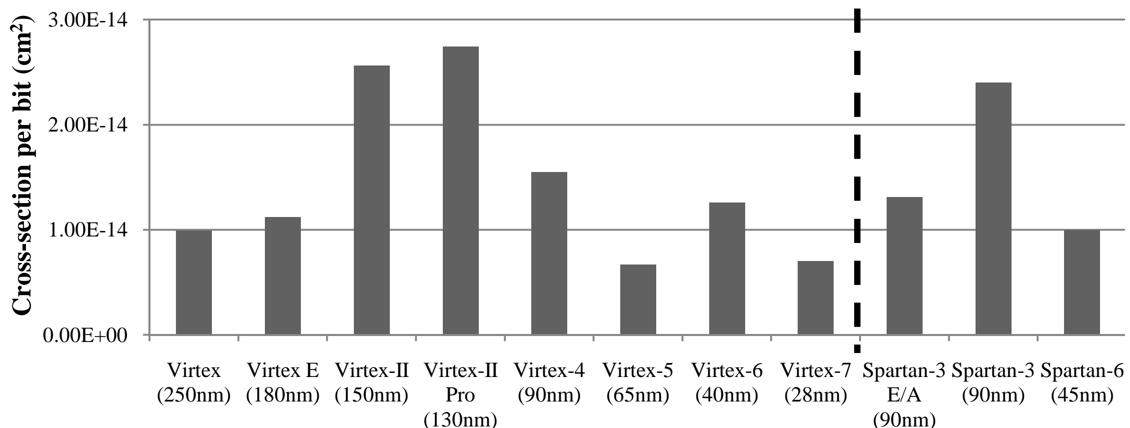


Figure 4.2: Static cross-section per configuration bit, as reported by (XILINX, INC., 2012c)

As the manufacturing technology scales, so does the capacitance of the transistors, as well as the supply voltage V_{dd}. This in turn reduces the critical charge required to change a storage cell's state, as was discussed in section 2.2.1. On the other hand, a smaller transistor is less likely to be struck by a particle. Furthermore, advances in the design of the storage cell may also improve its resilience against such particles. The result of these opposing factors is the non-monotonic variation of the cross-section per bit observed across different technologies shown in Figure 4.2.

The cross-section per bit, however, is not the only information necessary to evaluate the sensibility of a given device, as the amount of bits grows significantly from one generation to another. When multiplying the cross-section per bit by the amount of configuration bits in the largest device of each family, one gets a very different plot, as can be seen in Figure 4.3. The coupled effect of a larger cross-section per bit and a larger configuration size drove quickly the total cross-section until Virtex II Pro

devices, which were manufactured with a 130 nm process. Then, Virtex 4 and Virtex 5 families were able to compensate the increase in configuration size, slightly reducing the total cross-section. Until this point, a similar trend is observed for the Spartan series, as the Spartan 6 shows approximately the same total cross-section as the Spartan 3. However, Virtex 6 and Virtex 7 devices showed an aggressive increase in the total configuration size, while not significantly reducing the cross-section per bit. This results in a much larger total cross-section for these two families, reinforcing the need for fault tolerance techniques able to mitigate the effects of configuration errors.

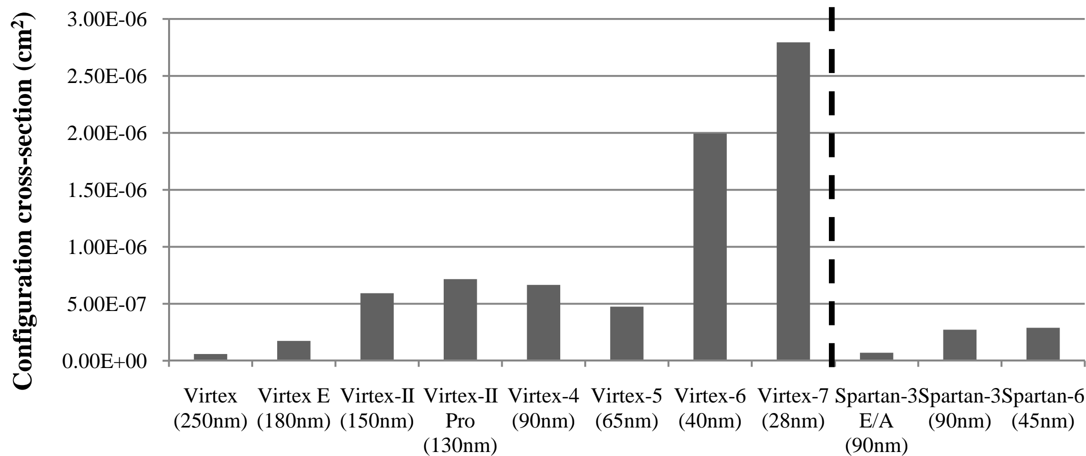


Figure 4.3: Static cross-section for the configuration of the largest device of each family

4.2.2 Artificial bitstream fault injection

The fast prototyping provided by FPGAs is a valuable asset for evaluating the dependability of FPGA systems. As the chip is usually available at the early stages of the project (unless a project is being developed for an unreleased device), designers perform very accurate reliability estimations without waiting for the manufacture of the chip at a foundry. The unlimited reconfigurability provided by SRAM-based FPGAs allows one to perform efficient fault injection campaigns on the actual device, providing timely and accurate results. As the configuration memory is programmable, one can artificially flip one or more bits on its content, artificially emulating the effects of SEUs that affect such memory.

Several platforms have been developed aiming at performing fault injection on the configuration memory of FPGA devices. The experiments conducted in (LIMA, CARMICHAEL, FABULA, *et al.*, 2001) made use of a control panel and two FPGA boards to inject bitflips in the configuration memory. The CUT is placed on one of the FPGAs, while the other one, along with the control panel, controls the experiment and communicates with a host PC. In (WIRTHLIN, JOHNSON, ROLLINS, *et al.*, 2003) a similar platform is presented, making use of three FPGAs. The first device contains the CUT, while the second contains a golden copy of it. The third device is responsible for applying the input vectors and checking the correctness of the CUT outputs. The fault rate was approximately 100 μs per fault.

The FLIPPER platform is presented in (ALDERIGHI, CASINI, D'ANGELO, *et al.*, 2007). It uses two FPGA boards. One contains the management circuit, which flips configuration bits and applies input vectors, while the other contains the CUT. The reported fault injection time was 50 μs per fault. Both input vectors and golden outputs

are derived from simulation software and stored in the on-board RAM before beginning the injection campaign. A software application, running on a PC, allows configuring the tests, choosing parameters such as clock rate, fault type (single or multiple bitflips) and stop conditions.

The FT-UNSHADES-C platform used in (STERPONE, AGUIRRE, TOMBS, *et al.*, 2008), which is an extension of FT-UNSHADES (AGUIRRE, TOMBS, MUOZ, *et al.*, 2007) to perform fault injection in the configuration bits, makes use of a similar approach. A control FPGA provides the interface between a host PC and the system FPGA, which holds both the CUT and a golden copy. Input stimuli are also derived from simulation hardware, as does the FLIPPER platform. No results were presented regarding the possible injection rates.

The fault injection platforms discussed so far make use of multiple FPGAs and, in some cases, of additional components, increasing the cost and complexity of the system. Furthermore, due to the need of off-chip communication, the use of multiple FPGAs is likely to also reduce the injection rate. A first system making use of a single FPGA was presented in (BERNARDI, SONZA REORDA, STERPONE, *et al.*, 2004). It relies on a host PC, however, to inject a fault in the configuration bitstream and to reprogram the device, increasing the injection time to approximately 6 s. Such long times make it infeasible to use this system to perform exhaustive fault injection campaigns on current FPGAs, due to the increased configuration sizes. This concern is addressed in (STERPONE and VIOLANTE, 2007), which presents a platform that places all the required components (CUT, input stimuli generation, fault injection and fault classification) in a single FPGA. In this platform, the host PC is only responsible for receiving and displaying the experiment results. The experiment control is implemented in software and executes on a hardwired PowerPC processor that is available on some Xilinx FPGAs. Fault injection is performed by writing a faulty configuration frame through the internal configuration access port (ICAP), a component that allows accessing the configuration memory from a user circuit in the same FPGA. The time strictly required to inject a single bitflip with this platform is 10.1µs. A more detailed classification framework was presented in (BOLCHINI, CASTRO and MIELE, 2009), using the injector described in (STERPONE and VIOLANTE, 2007). It allows the individual evaluation of the effects of each fault, which is a valuable resource when one desires to improve the reliability of a design.

4.3 Fault injection platform

As was discussed in chapters 1 and 4, fault injection is among the most traditional means to measure the dependability of systems. Furthermore, FPGAs present a very particular fault model that requires dedicated experimental platforms to accurately measure metrics such as fault coverage and failures in time. The most traditional method is to flip configuration bits of an actual FPGA device to observe the effects on the user circuit running on the reconfigurable fabric. Several approaches available in the literature were discussed in section 4.2.2, and in this chapter we present the fault injection platform developed in this work.

The main advantages of the proposed platform are:

- Low cost and low complexity, since it requires only a single FPGA and a host computer to carry out its functions;
- High injection rate, as no external memories or controllers are required to inject faults and to apply stimuli to the CUT;

- Applicability to other devices, as the system is composed only of LUTs, flip-flops and a small memory, requiring no complex hardwired component, with the exception of an internal configuration access port (ICAP);
- Modularity and extensibility, which allows adapting the system to different types of circuits and different fault models.

4.3.1 Platform components

As for any fault injection platform, the basic components shown in Figure 4.1 must be present in order to inject faults and to evaluate their effects on the operation of the CUT. The components that form the proposed platform are shown in Figure 4.4 and described in the following subsections. The function of each component may vary depending on the specific needs of each experiment campaign. Thus, the components described herein can be modified to satisfy different needs, and some of the possible variations are described in the following subsections as well.

Furthermore, some adaptations are required to implement the platform in FPGAs from different manufacturers or different families of the same manufacturer, especially regarding the injector block, which must handle configuration addressing and interface with the reconfiguration port available in the device. The platform herein described was implemented and tested on Virtex 5 XC5VLX110T FPGA, and some of the details provided focus on this device family. The proposed approach, however, remains applicable to any device that allows the user circuit to access the configuration memory.

4.3.1.1 Injector

The injector unit is responsible for actually modifying the current bitstream according to the specified fault model. For that purpose, it must first choose the specific bit(s) of the configuration to be flipped. A bit is univocally identified by its frame address and its position within the frame. As discussed in section 2.1, a frame is the smallest addressable unit of the configuration memory. For example, a Virtex 5 frame is composed of 41 words of 32 bits, for a total of 1,312 bits.

The frame address generation unit, thus, is responsible for choosing a valid frame address for injection. This choice may be pseudo-random or sequential, if exhaustive fault injection is to be performed. Frame addresses, however, are not organized in a straightforward continuous fashion. Each frame address is divided into fields that may

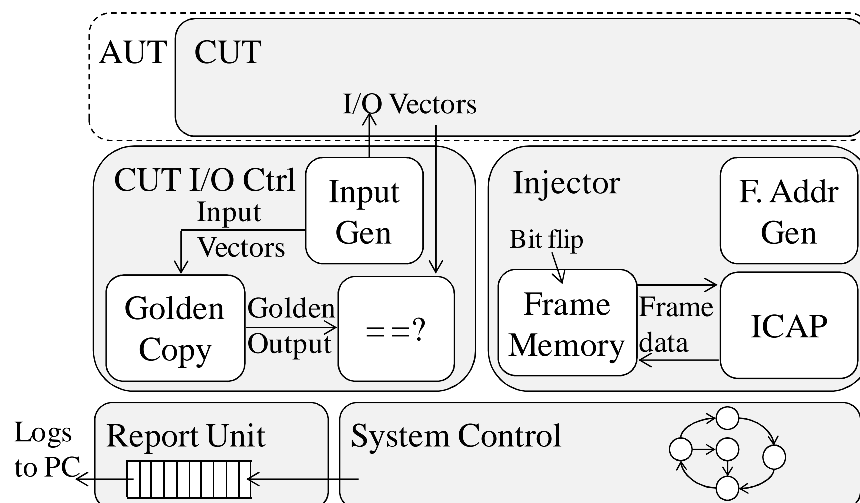


Figure 4.4: Fault injection base architecture

vary from one device family to the other. Virtex 5 devices divide their frame addresses into block type (3 bits), top/bottom (1 bit), row address (5 bits), major address (8 bits) and minor address (7 bits), for a total of 24 bits. More details regarding frame addressing and organization can be found at (XILINX, INC., 2011a).

The choice of an appropriate frame address must also take into account that the fault injection platform is on the same FPGA and it must not disturb its own operation. Thus, the concept of area under test (AUT) is defined, which restricts which configuration frames are eligible to suffer fault injection. The circuit under test must be placed within the AUT and the experiment control circuitry out of it, to ensure that it will maintain its own integrity. This is achieved by means of placement constraints. Using the address fields to aid in this process can greatly simplify determining the frames associated with the AUT. In this work, we use an AUT limited within the top frame row, comprising 2,000 slices (8,000 LUTs and 8,000 flip-flops) and with approximately 2.6 million configuration bits. If larger circuits are to be tested, then larger AUTs can be defined. Doing so, however, also extends the experiment time.

Once the injector has defined a target frame and bit, it must read the desired frame from the configuration. The read frame is stored in the frame memory. Then, the chosen bit is flipped and the frame is written back, thus corrupting the bitstream. The read frame remains in the frame memory for the following fault removal, once requested by the system control. Bits are flipped back to their original values and the correct frame contents are restored.

Interacting with ICAP requires the following of a specific protocol, that includes issuing read and write commands. Moreover, the results of each read command are preceded by a dummy frame. Likewise, after completing a write command, one dummy frame must be pushed in the ICAP data port. The costs of these commands and the dummy frames will be quantified and taken into account when estimating the reachable injection rate, in section 4.3.3. It is also important to keep in mind that configuration frames have addressable non-existing bits. In other words, there are, scattered throughout the memory, addressable bits that have no actual associated memory cell (XILINX, INC., 2011a). Such bits should not be considered by the injection platform. Thus, each injection is followed by a frame read to confirm that the injection was successful and that only real configuration cells will be taken into account.

4.3.1.2 CUT I/O Controller

The CUT I/O Controller is responsible for interfacing with the CUT in order to stimulate its operation and evaluate the effects of each injected fault. Thus, it must be able to generate input stimuli and to compare the circuit outputs to the expected values. This can be achieved by different means. The controller shown in Figure 4.4 assumes that inputs are generated and applied both to the CUT and to a golden copy of it, which allows evaluating the correctness of outputs at each cycle. If the CUT is a softcore processor, however, the developer may be interested in the final state of the memory, instead of a cycle per cycle comparison. The system may perform an initial fault-free run of the software and store the final (golden) state of the memory, which is then used as a reference for the subsequent faulty executions. The system requires, in this case, three memories: one stores the initial memory state, one stores the golden final state, and one is used as work memory, i.e., the memory that the CUT uses during execution.

Several different options exist regarding the input stimuli generation as well. They may be pseudo-random, generated by a linear feedback shift register (LFSR), for

example. Alternatively, the developer may be interested in a specific set of input vectors that represent the typical use case of that hardware, which may be stored in a memory.

4.3.1.3 Report Unit

The report unit is responsible for transmitting the experimental results to the host PC for analysis. The specific information transmitted may differ depending on the purpose of the experiment and the nature of the CUT. If it possesses some sort of error detection mechanism, for example, the developer may be interested in determining which faults were detected and which were not. There may also be situations in which the developer is interested in the specific output values the CUT generated when faulty. For such situations, the transmission of results may become a serious bottleneck of the system. For straightforward error detection evaluations, however, low speed interfaces, such as a serial port, are sufficient.

4.3.1.4 System Control

The system control unit coordinates the operation of all other modules in order to realize a complete fault injection campaign. It starts by requesting a fault injection from the SEU injector unit (1). Once the fault is injected, it activates the I/O controller so that it starts applying input vectors (2). For a pre-specified number of cycles it monitors the correctness of outputs and any error detection signal that may be triggered (3). The system control then halts the I/O controller and requests the fault removal (4). While the fault is removed, the report unit transmits the outcome of that particular fault (5). Steps (1) through (5) are repeated until the desired amount of faults is injected. Then, the report unit transmits any final results that were obtained during the experiment (6) and finishes the execution.

4.3.2 Area costs

In order to leave as many resources as possible available for the CUT, allowing larger and more complex circuits, it is important to maintain the entire SEU injection and control system as small as possible. Table 4.1 shows the amount of resources used by each component and by the entire system as well as the proportional occupation of the device, considering a Virtex 5 XC5VLX110T.

Table 4.1: Required resources and device occupation for a fault injection platform

	Required resources			*Device Occupation*		
Module	*LUTs*	*FFs*	*BRAM*	*LUTs*	*FFs*	*BRAM*
SEU Injector	431	149	1	0.62%	0.22%	0.68%
CUT I/O Controller	14	137	0	0.02%	0.20%	0.00%
Report Unit	30	15	0	0.04%	0.02%	0.00%
System Control	647	384	0	0.94%	0.56%	0.00%
Total	1122	685	1	1.62%	0.99%	0.68%

The exact area occupation depends on the specific version of the platform being used. For example, if the input vectors are stored in BRAMs, then the occupation of this type of component will be increased. If the system uses a LFSR to generate pseudo-random inputs, then additional flip-flops and LUTs will be required instead. The area results provided herein refer to a platform injecting faults in a 32-bit ALU, which has 69 input bits (two 32-bit operands and 5-bit operation code) and 33 output bits (the result value and an overflow flag). The injection platform system requires 1122 LUTs and 685 flip-flops, occupying 1.62% and 0.99% of the total device respectively, which contains

69,120 of each. The BRAM occupied by the SEU Injector unit stores the read frames before they are written back to the configuration. Note that, even though 98.38% of the LUTs are still available, not all of them are usable by the CUT. Some extra areas are required to ensure the isolation of the CUT and of the control system. Still, the vast majority of the FPGA may be used to accommodate the CUT.

4.3.3 Injection Rate

Since current FPGAs allow multiple clock domains, the components shown in Figure 4.4 do not need to run at the same frequency. In this work, the SEU injector and the ICAP run at 50 MHz to ensure that there is no timing violation, as is done in (CHAPMAN, 2010). If required, however, the ICAP can be used with frequencies up to 100 MHz, further accelerating the process.

As described in section 4.3.1.1, each fault is injected by reading the frame, applying the required modification and then writing the frame back. The frame is then read back to verify that the injection was successful. For each read command, one invalid dummy frame must also be retrieved, due to the internal implementation of the ICAP. Similarly, for each write command, one dummy frame must be inserted in the data input port of the ICAP. As each frame contains 41 words, each read or write access requires 82 cycles to be completed. Furthermore, there is the action of sending the read or write instructions, which require several smaller commands. More details about the write and read sequences can be found at (XILINX, INC., 2011a). Thus, the total time to read or write a frame is the sum of the time required to send the command, to read or write a dummy frame and to read or write the actual data. For the implementation done in this work, the total times are 108 cycles to read a frame and 107 cycles to write a frame. Hence, the total time strictly required to inject a fault is 215 cycles (one frame read and one frame write). When considering also the time required to confirm the injection (one read operation) and to remove the fault (one write operation), 215 additional cycles are required. Thus, the strict injection time is 430 cycles, or 8.6 µs, considering the 50 MHz clock frequency.

This injection rate allows, for example, exhaustively injecting faults in an intermediate-sized FPGA, such as the Virtex 5 XC5VLX110T used in this work (with approximately 24 Mbits of configuration), in less than 4 minutes. Further optimizations are possible, especially when performing sequential fault injection, since in such cases the following injected fault is likely to be in the same frame of the previous one. However, the injection latency is so short that the total campaign time is likely to be dominated by the stimulation of the CUT or the transmission of results. Therefore, further reducing injection time will have little impact on the final experiment for most cases. For example, assuming the CUT will run for 100,000 cycles at 50 MHz for each fault, injection time represents less than 0.5% of the total experiment time. The total time, in this case, is approximately 1 hour and 30 minutes for the AUT described in section 4.3.1.1 (2,000 slices on the top frame row of the device).

5 FINE-GRAINED ERROR DETECTION

In this chapter, we present the developed fine-grained error detection mechanism. The basic approach is described in section 5.1 and in section 5.2 the experimental setup and design flow used to evaluate the proposed technique are detailed. Experimental results including area, delay, error detection and detection acceleration are presented and discussed in section 5.3. The radiation experiments conducted are described and discussed in section 5.4.

5.1 Fine-grained detection with carry propagation chains

When building an adder or a subtracter in a LUT-based FPGA, emerges the problem of calculating the most significant bits of the output. As they depend on all the least significant bits, the amount of LUTs required to compute each one increases significantly. For this reason, FPGA manufacturers include, along with each LUT, a small circuit that comprises basically a multiplexer and an XOR gate to compute both the carry out and the sum bits. Even though this circuit can be used to compute other functions (XILINX, INC., 2010), synthesis tools rarely use them, unless an adder is explicitly declared.

Figure 5.1 shows a simplified view of a carry chain circuit and the LUTs coupled to it, based on a slice of a Virtex 5 device (XILINX, INC., 2010), shown in Figure 2.2. A Virtex 5 slice comprises 4 LUTs, 4 flip-flops, the carry chain circuit and some multiplexers for internal routing. The labels in Figure 5.1 indicate how one may use the carry circuit to compare two pairs of duplicated LUTs. At the first stage (the bottommost one), the multiplexer inputs are set to constant values ‘1’ and ‘0’, forcing it to propagate the output of LUT A, *X*, to the next stage. The first XOR gate has one input set to ‘1’ and the other to *X*. It behaves, thereby, like an inverter. Through the internal routing of the slice, the output of the first multiplexer is directly connected to the inputs of the second stage’s multiplexer and XOR gate. Through the external routing, i.e., the global routing wires of the FPGA, one may set the other input of the second multiplexer to $\overline{X}$, as shown by the dashed arrow Figure 5.1.

The two inputs of the multiplexer in the second stage are set to *X* and $\overline{X}$, for when the selection signal equals ‘1’ or ‘0’, respectively. As the selection signal is the output of LUT B, *Y*, this is equivalent to calculating the XNOR function of *X* and *Y*. The inputs of the XOR gate in the second stage are also equal to the outputs of the two first LUTs. The carry circuit is computing, hence, both the XNOR and XOR functions of *X* and *Y*, and these values can be connected in a similar manner to the third carry stage and so on, realizing the computation of the XOR and XNOR functions of the entire slice. One can configure the top two LUTs in the slice to compute the same functions of the bottom two. Thus, under normal circumstances, the output of the XOR gate at stage 4 will always be ‘0’. And if any LUT diverges from its correct value, the error signal will be

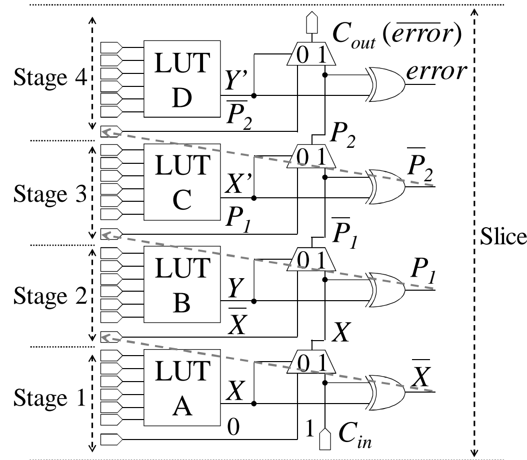


Figure 5.1: Carry chain circuit applied to fine-grained comparison

raised. Thereby, a slice-wise error detection signal is implemented completely avoiding the need to use LUTs to implement comparators.

As the carry out bit at the top of the slice, shown in Figure 5.1, is connected directly to the carry in of the next slice in the same FPGA column, the amount of LUTs that can be compared this manner, by a single comparator, is limited only by the amount of rows in the device. The latency of such circuit, however, could be too large for the application at hand. Thus, several smaller checkers can be stacked in a same column to keep delay penalties to a minimum. This actually allows one to find the best trade-off for each application, regarding error detection granularity and delay penalty. The example in Figure 5.1 assumes that each slice will produce an individual error indication signal at the output of the topmost XOR gate. Also, as the last carry out of a checking group is actually the inverted error signal, it will always be ‘1’, unless a fault was already detected. This allows the stacking of arbitrarily long comparators in a same column even without respecting slice boundaries, since the only requirement is that the bottom carry in is equal to ‘1’.

In order to minimize undetectable errors, it is crucial to maintain an appropriate routing between modules. Figure 5.2(a) shows an approach in which one of the modules drives both replicas of the following logic stage. There is a potentially critical routing segment created in between the two stages, shown by a dashed line. Faults on that segment may not be detected by e_0, since it is past the point in which the comparator is connected, nor by e_1, since whichever effect the fault has on the wire will be observed by both LUTs l_1 and l_1’, leading to incorrect results on both. The scheme used in Figure 5.2(b) removes that segment by connecting l_0 to l_1 and l_0’ to l_1’. Thus, unless a single fault corrupts both nets, which can be minimized by using reliability-aware routing as in (STERPONE and VIOLANTE, 2006), faults affecting one of them will be detected by either e_0 or e_1. Similar situations occur to PIs and POs. The branching of PIs should be done as soon as possible, as shown in Figure 5.2(b). This is particularly important for

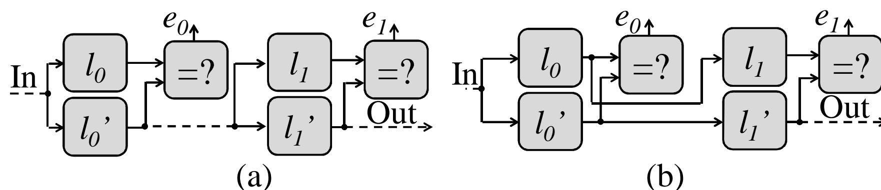


Figure 5.2: Incorrect (a) and correct (b) routing in FG-DMR. Dashed lines denote critical routing paths

fine-grained approaches, because an unaware routing algorithm could tend to split these nets near the modules, as in Figure 5.2(a), due to the closeness of the redundant modules. In this work we use duplicated PIs to eliminate these critical segments. Alternatively, a modified routing algorithm could force them to split as soon as possible to minimize undetectable faults. Reducing the length of PO routing is also an alternative to reduce the length of critical routing. Moreover, POs can be duplicated as well, if the module is followed by another duplicated circuit.

There are some limitations to the applicability of the proposed comparison mechanism. If the carry chain is already occupied to perform another function, such as addition, then it naturally cannot be used for comparison. Furthermore, the extra slice inputs that are required to route the partial comparison signals (shown with dashed arrows in Figure 5.1) must be free. When the synthesis tool allocates them to other resources, such as the multiplexers that are used to implement arbitrary 7-input and 8-input functions (MUXF7 and MUXF8, respectively), then regular LUT-based comparators must be instantiated.

Once all comparators are defined, one is left with numerous error detection signals. If these signals are to be used to trigger a local scrubbing procedure, then they must be combined into a single bit. This is done by computing the OR function over all signals. We refer to this operation as *error aggregation*. In all results presented in this chapter, the existence of the error aggregation circuit is taken into account.

When instantiating redundancy checkers, it is also always important to also take into account the reliability of the checker itself. The use of redundant checkers is a traditional approach to assert the detection of faults affecting the checking circuit (KUNDU and REDDY, 1990). Specifically for FPGAs, if a single checking bit is used, errors affecting the bitstream portion associated with the comparator may set its output to ‘0’ (assuming ‘1’ indicates an error). Such errors may stay dormant for a long period, affecting the overall reliability. In order to avoid this issue, redundant checkers may be used. Figure 5.3 shows how they are implemented in this work. In order to avoid the excessive area overheads of fine-grained LUT-based comparators, we implement a redundant checker that operates only on the primary outputs (POs) of the circuit, avoiding the propagation of the error to other modules in the system. The use of LUT-based comparators for POs is also useful as it allows reducing the length of the critical PO routing segments shown in Figure 5.2 by placing them close the end of the net (an IOB, for example).

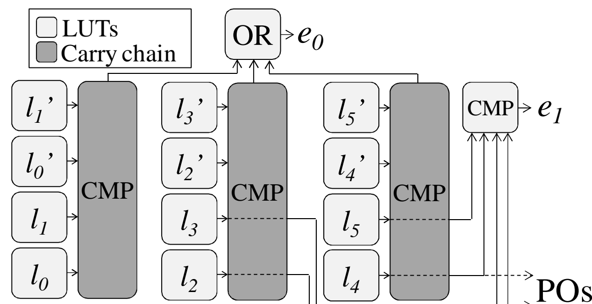


Figure 5.3: Redundant heterogeneous comparators

5.2 Experimental setup

Figure 5.4 shows the design flow used. It starts with an unhardened description of the user circuit in a standard hardware description language, which is synthesized using

Xilinx Synthesis Technology (XST). The post-synthesis netlist is converted from its native format into a structural VHDL description using Xilinx *netgen* (XILINX, INC., 2011c). At this point, the circuit is already described using the basic components found in the FPGA fabric, such as LUTs, flip-flops, carry chains and multiplexers.

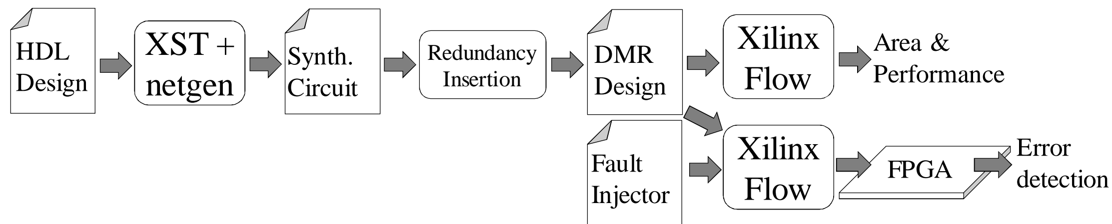


Figure 5.4: Experimental design flow

A redundancy insertion tool was developed in C++ to automatically apply the proposed technique. It parses the post-synthesis netlist and builds an internal representation of the circuit. Then, it duplicates all components and instantiates carry chain comparators that cover one slice for those LUTs that have available the required resources, as shown in Figure 5.1. All internal signals are duplicated as well, in order to maintain the routing redundancy shown in Figure 5.2. For those LUTs to which the technique is not applicable, regular LUT-based comparators are inserted. The error aggregation circuit is also introduced by the tool, if requested by the user. The tool generates a structural VHDL description of the hardened circuit, also using the, which goes through Xilinx standard flow to determine area and delay costs.

The hardened circuit is then subject to fault injection, using the platform described in section 4.3. Exhaustive fault injection is used, i.e., every configuration bit associated with the CUT is flipped (2,628,288 bits). The AUT used is that described in section 4.3.1.1 (2,000 slices on the top frame row). Circuits receive pseudo-random inputs, which are applied to a golden copy of the circuit as well. For each injected fault, 100,000 input vectors are applied. And for each applied vector, the correctness of the outputs is verified, along with the state of the error detection bits. Each vector can be classified into one of four categories (shown here in ascending severity order):

1. *No event*: the outputs are correct and the error detection bits are low. This occurs frequently, since not all configuration bits are able to corrupt the circuit operation. Furthermore, not all input vectors are able to stimulate an error, even when it indeed affected the circuit.
2. *Detected only*: the outputs are correct but an error detection bit was raised. This happens mainly for one of three reasons: the secondary copy of the circuit was struck by the fault, i.e., the one not driving POs; the checking circuit was struck; the primary copy was hit and the error was detected by an internal comparator, but it did not yet propagate to a PO, i.e., it was masked by the circuit logic.
3. *Detected error*: the outputs are incorrect and the error was detected. This is the straightforward situation in which the error propagated to a primary output and was detected by the comparators.
4. *Undetected error*: the outputs are incorrect but the error detection bits remained low. This is by far the most severe case, which happens mainly when a PO is affected past the point in which it is compared to its copy. It may also happen due to single faults that affect multiple nets in both redundant circuits, (LIMA, CARMICHAEL, FABULA, *et al.*, 2001).

Each injected fault is classified into the highest severity category it presented among all applied input vectors, as is done in (BOLCHINI, CASTRO and MIELE, 2009). Figure 5.5 shows some of the most likely locations of faults in each category, in a simple coarse-grained DMR circuit for the sake of clarity.

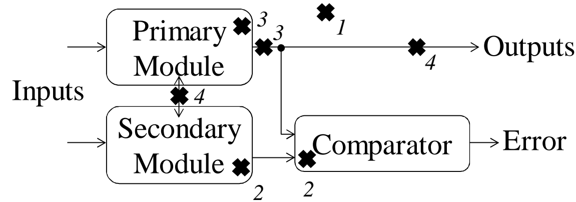


Figure 5.5: Locations of faults of each category

The platform monitors not only the specific outcome of each fault (i.e., if it caused errors in the circuit POs and/or if it was detected) but also the amount of cycles it takes for error detection to be triggered. This allows determining the average detection time, which is important for systems relying on triggered scrubbing to remove faults.

5.3 Experimental results

Table 5.1 characterizes the input benchmark circuits regarding the amount of LUTs, flip-flops, primary inputs (PIs), primary outputs (POs) and minimum clock period T_{Clk}. A set of 22 benchmark circuits was used, 20 of which are from the MCNC

Table 5.1: Input benchmark circuits

	LUTs	*FFs*	*PIs*	*POs*	T_{Clk} *(ns)*
alu4	402	0	14	8	4.94
alu_32b	342	0	69	33	6.77
alu_64b	721	0	133	65	8.01
apex2	798	0	39	3	6.26
apex4	655	0	9	18	6.39
bigkey	575	224	264	197	3.63
clma	1269	34	384	82	7.25
des	550	0	256	245	4.26
diffeq	470	244	29	3	4.64
dsip	635	224	230	197	2.78
elliptic	143	71	20	2	3.46
ex1010	487	0	10	10	4.59
ex5p	128	0	8	63	2.99
frisc	1718	853	21	116	8.30
misex3	699	0	14	14	5.55
pdc	1253	0	16	40	6.18
s298	17	14	5	6	2.78
s38417	1709	1447	30	106	5.26
s38584.1	2001	1233	40	304	4.84
seq	846	0	41	35	5.27
spla	221	0	16	46	3.98
tseng	598	260	53	122	5.17

(Microelectronics Center of North Carolina) benchmark suite and were obtained at (MINKOVICH, 2011). For all of these, described by means of boolean equations and flip-flops, the synthesis tool was unable to make any use of the carry propagation circuit. The other two circuits are ALUs with 32 and 64 bits (*alu_32b* and *alu_64b*), described with a higher level behavioral VHDL and explicitly using additions and subtractions. As a result, the synthesis tool was able to infer adders/subtracters for these circuits. However, even for such cases, only approximately 10% of the LUTs had their associated carry circuitry occupied. This shows that for many cases the carry propagation chain is highly unused, and can be available for the application of the proposed technique.

In order to set baseline values for each evaluation axis presented herein, the proposed fine-grained DMR (FG-DMR) is compared to traditional coarse-grained DMR (CG-DMR). It consists in duplicating the entire circuit and comparing the primary outputs only, also with redundant comparators. The benchmark circuits can be viewed as individual modules in a larger system, in which case the baseline CG-DMR is in keeping with the approaches used in (BOLCHINI, MIELE and SANDIONIGI, 2011) and (PSARAKIS and APOSTOLAKIS, 2012). Note that the benchmark circuits have very diverse sizes and some of them have a large amount of POs compared to their own total sizes (such as *ex5p*, *des* and *bigkey*). For such cases, as a relevant amount of internal signals are also POs, the two approaches are likely to behave similarly on some of the comparison axes. All results are shown both in tables, with absolute values, and charts, in order to highlight the relations between techniques for different circuits.

5.3.1 Area

Since minimizing the area is among the motivations of the proposed FG-DMR technique, it is important to assess if the observed overhead is indeed comparable to that of standard CG-DMR. For both, the increase in number of flip-flops is exactly 100%, since these are used only in the payload circuit itself, i.e., the circuit computing the user-specified function. Thus, we focus our analysis on the use of LUTs, which are the basic logic building blocks of FPGA circuits and are exactly the resource FG-DMR aims at saving. Table 5.2 shows the absolute costs, in number of LUTs. Figure 5.6 shows proportional overheads over the unhardened circuit, for both approaches, with CG-DMR on the left and FG-DMR on the right for each circuit.

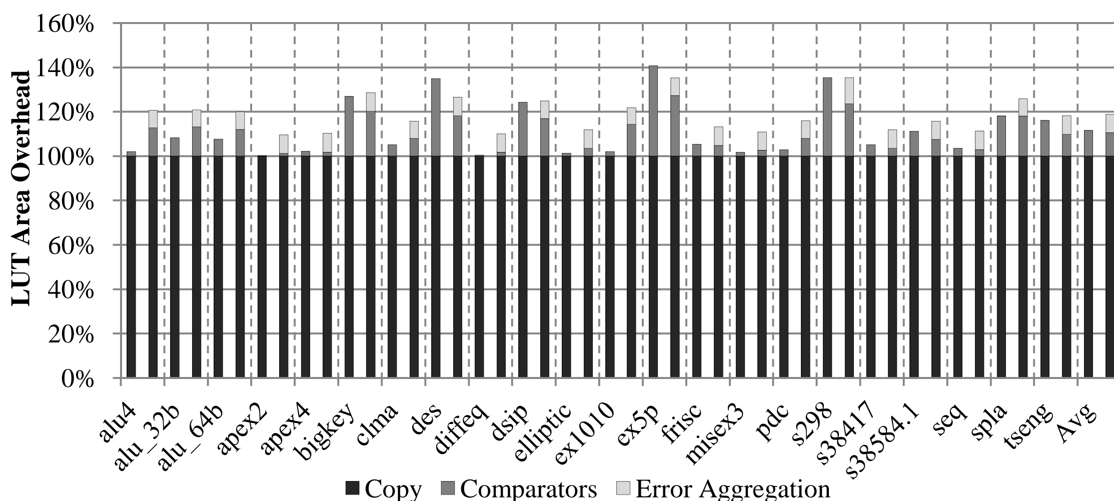


Figure 5.6: Area overheads for CG-DMR (left-hand bar of each circuit) and FG-DMR (right-hand bar)

Table 5.2: Area costs in LUTs (comparators, error aggregation and total, including the two circuit copies)

	CG-DMR		*FG-DMR*		
	Comp.	*Total*	*Comp*	*Error Aggreg.*	*Total*
alu4	8	812	51	32	887
alu_32b	28	712	45	26	755
alu_64b	55	1497	87	57	1586
apex2	2	1598	10	66	1672
apex4	14	1324	12	55	1377
bigkey	154	1304	116	48	1314
clma	65	2603	101	98	2737
des	192	1292	100	46	1246
diffeq	2	942	8	39	987
dsip	154	1424	107	51	1428
elliptic	2	288	5	12	303
ex1010	10	984	70	36	1080
ex5p	52	308	35	10	301
frisc	92	3528	83	144	3663
misex3	12	1410	18	58	1474
pdc	34	2540	99	100	2705
s298	6	40	4	2	40
s38417	88	3506	60	143	3621
s38584.1	224	4226	152	164	4318
seq	30	1722	25	70	1787
spla	40	482	40	17	499
tseng	96	1292	59	50	1305

The costs in Figure 5.6 and Table 5.2 are divided into the main components of each approach. CG-DMR comprises a 100% cost due to the copy of the circuit and, on average, additional 11.6% due to comparators, for a total of 111.6%. FG-DMR, on the other hand, introduces a 10.5% average overhead due to comparators and 8.3% to perform error aggregation, for a total of 118.8% overhead when considering the circuit replica as well. The comparator cost of FG-DMR comprises both the redundant output comparators and the fine-grained LUT comparators for situations in which carry chains could not be used.

The CG-DMR costs are particularly more pronounced for those circuits with a high amount of POs compared to its total size. Most notably, *des* (550 LUTs and 245 POs) and *ex5p* (128 LUTs and 63 POs) present such high costs for CG-DMR that FG-DMR actually requires fewer LUTs. For *s298* both approaches present exactly the same area. The costs of FG-DMR depend not only on the amount of POs (since it also has a PO-only comparator) but also on the amount of LUTs to which the carry chain comparison is not applicable. For *alu4*, for example, 24.4% of the LUTs make use of their associated MUXF7 multiplexer, imposing the need for many LUT-based comparators and increasing the comparator costs of FG-DMR when compared to CG-DMR. For the remaining circuits, however, these situations occur more rarely. As a result, the average

area overhead of FG-DMR over CG-DMR is 3.57%, showing that the proposition of maintaining a manageable overhead was achieved.

5.3.2 Clock period

Although usually smaller than that of temporal redundancy techniques, spatial redundancy techniques also introduce performance penalties. For DMR, the delay overhead is caused mainly by the checking circuits that are introduced in series with the critical path of the original circuit. Table 5.3 shows the minimum clock period T_{Clk}, in nanoseconds, for each circuit. For the combinational circuits, T_{Clk} comprises the complete circuit delay.

Table 5.3: Minimum clock period in nanoseconds

	Unhardened	*CG-DMR*	*FG-DMR*
alu4	4.94	6.55	7.52
alu_32b	6.77	8.27	9.83
alu_64b	8.01	9.59	11.98
apex2	6.26	7.84	10.86
apex4	6.39	7.79	9.78
bigkey	3.63	4.56	8.12
clma	7.25	7.29	11.87
des	4.26	7.08	8.46
diffeq	4.64	4.97	9.55
dsip	2.78	4.14	8.26
elliptic	3.46	3.50	6.91
ex1010	4.59	6.61	7.59
ex5p	2.99	4.83	5.59
frisc	8.30	8.33	15.08
misex3	5.55	7.37	10.03
pdc	6.18	8.94	10.67
s298	2.78	2.80	4.09
s38417	5.26	5.61	12.11
s38584.1	4.84	7.06	11.20
seq	5.27	7.39	10.04
spla	3.98	5.81	6.94
tseng	5.17	5.77	9.11
Average	5.15	6.46	9.34

Figure 5.7 highlights that the two techniques have very diverse behaviors for each circuit. For example, for *alu4* both present comparable costs (the delay of FG-DMR is 14.8% longer than that of CG-DMR). For other circuits there may be more significant increases when introducing FG-DMR. For *frisc*, e.g., FG-DMR presents 81.1% overhead over CG-DMR. On average, FG-DMR presents an 86.3% T_{Clk} increase over the unhardened circuit and 48.7% over CG-DMR. The additional delay is due to the nature of the introduced comparators. When using the proposed FG-DMR with carry chain comparators, intermediate signals are routed through the global wires of the device, as was discussed in section 5.1, introducing additional delay. Moreover, since

many error signals are generated, the error aggregation circuit imposes further overheads.

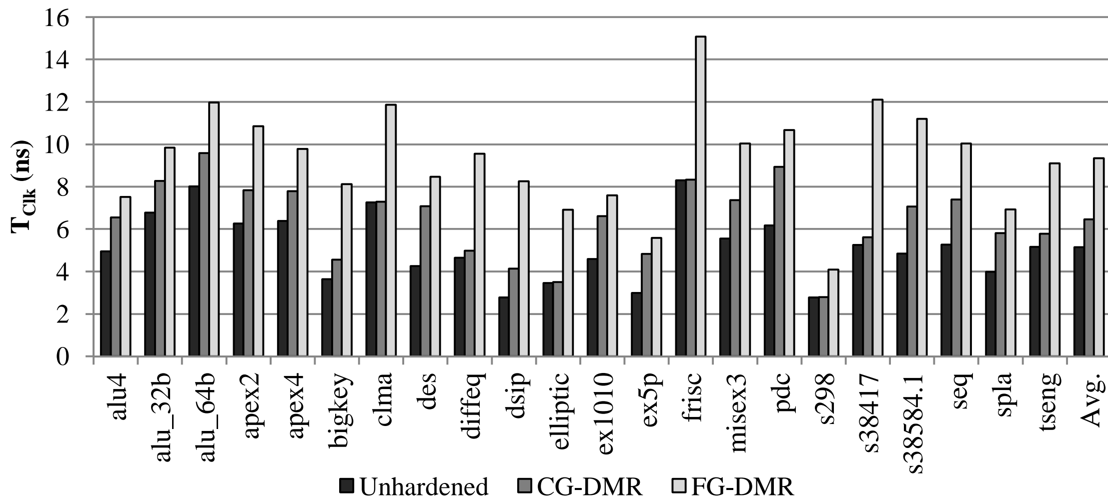


Figure 5.7: Minimum clock period T_{Clk} for the unhardened circuit, CG-DMR and FG-DMR

The fact that CG-DMR compares only POs further fuels this difference, especially for sequential circuits. For instance, if all primary outputs of the circuit are registered, then the introduction of output comparators may not change the critical path at all. Sequential circuits such as *clma*, *frisc* and *s298* show negligible differences between the unhardened versions and CG-DMR. As a result, the increase in T_{Clk} of FG-DMR over CG-DMR is particularly more pronounced for sequential circuits (78.9%, on average) than for combinational ones (23.6%). However, when comparing only primary outputs, it may take longer to detect the occurrence of a fault, leaving the error latent for a longer time, as will be shown in section 5.3.4. Moreover, when the error reaches the comparators, internal registers are likely to be corrupted, increasing the complexity of checkpoint and rollback procedures, as discussed in (PSARAKIS and APOSTOLAKIS, 2012). Finally, these measurements consider straightforward clock period, which may not reflect directly in the performance. The performance of the system may be limited by other modules, that may present clock period or throughput limitations, or by the input bandwidth. It can also be the case in which the system is able to meet the real-time deadlines with spare time. In such cases, it may be more relevant to provide fast error detection and correction than the fastest possible circuit operation.

5.3.3 Error detection

In this section we discuss the results of fault injection regarding the classification of faults into the categories described in section 5.2. Table 5.4 shows the absolute amount of faults in each category (category 1 is omitted for clarity). It can be seen that FG-DMR shows a very significant increase in the amount of faults in category 2. This, in fact, is firstly related to the greatly increased observability that FG-DMR introduces. Since it compares individually the output of each and every LUT, it has a much increased probability of detecting the presence of an error. Therefore, there is an increased likelihood that an error effectively affecting the circuit is classified as “no event” for CG-DMR simply because it never propagated to a primary output. This is more pronounced for sequential circuits due to the increased difficulty in propagating faults in such circuits, a property long identified by researches on automated test pattern generation (ABRAMOVICI, BREUER and FRIEDMAN, 1990). Second, the fine-

Table 5.4: Amount of faults in each category

	CG-DMR			*FG-DMR*		
	2) Det. Only	*3) Det. Error*	*4) Undet*	*2) Det. Only*	*3) Det. Error*	*4) Undet*
alu4	29286	29199	156	55065	38414	83
alu_32b	29892	27610	416	48210	33473	377
alu_64b	62604	63174	749	104927	76443	557
apex2	45437	45087	92	156147	59282	47
apex4	57943	56686	197	90299	73852	196
bigkey	67352	51496	1319	79904	56617	1225
clma	4576	2770	185	98612	3469	213
des	69231	52351	1366	88262	61817	1612
diffeq	577	547	22	40749	747	12
dsip	73524	62001	1394	105406	73560	975
elliptic	300	245	16	13681	466	30
ex1010	30077	29781	109	60643	39891	160
ex5p	11185	8157	305	15753	9989	401
frisc	63096	56604	407	203668	88283	864
misex3	46839	46146	130	98855	67283	168
pdc	106608	104191	356	197795	141581	434
s298	848	740	27	1270	1059	33
s38417	24744	19732	353	245031	27188	244
s38584.1	168111	149026	1059	384042	230289	1161
seq	71711	72449	242	136585	96316	368
spla	17873	15631	203	28319	19371	405
tseng	9236	4340	556	78902	6400	439

grained comparators naturally demand additional configuration bits, which also impacts on the amount of faults observed on category 2. Fine-grained comparison also increases, but on a reduced scale, the amount of faults in category 3, which are output errors that were detected. This is due to the modified placement and routing that the proposed technique imposes, which can add sensitive bits to the primary circuit as well.

Of special interest is the comparison on the amount of faults in category 4, which are those that caused a PO error and went undetected for at least one input vector. Figure 5.8 presents the variation in the amount of undetected errors for the two techniques. Negative values indicate benchmark circuits for which FG-DMR had fewer such events, i.e., $F_{4CG} > F_{4FG}$. FG-DMR showed fewer faults in category 4 for 10 circuits, while CG-DMR was better on the remaining 12. On average, FG-DMR presents 12.73% more undetected faults.

Another, and also relevant, perspective on the meaning of these figures can be found by analyzing the fault coverage. As discussed in section A.5 of Appendix A, fault coverage is the fraction of total events that was appropriately handled by a given mechanism, being an important metric of its efficacy. It is usually calculated as the ratio between covered faults (detected, in this case) and total faults. The total amount of faults, however, can be defined in different ways for this kind of experiment. Simply taking into account the total amount of injected faults can be misleading, since most of

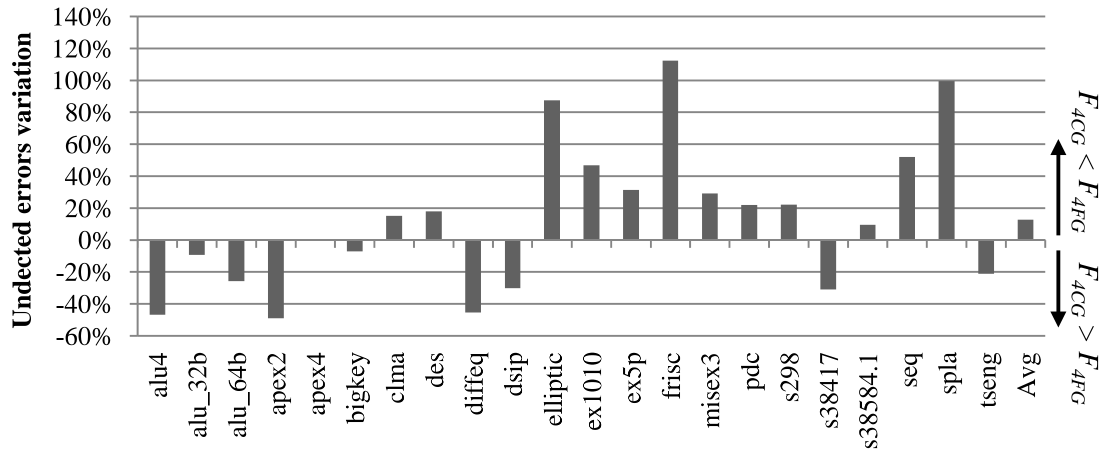


Figure 5.8: Undetected error variation. Positive values indicate a smaller amount for CG-DMR.

them did not actually hit the CUT. As discussed in (LESEA, DRIMER, FABULA, *et al.*, 2005), most bits are bound to have no effect on the system, even in high occupation scenarios, due to the great over provisioning required from routing resources. Thus, we consider only sensitive bits, i.e., those that modified circuit behavior in some way. Let F_x denote the amount of faults in category x. The total amount F_T can be calculated as $F_T = F_2+F_3+F_4$. However, when comparing two different techniques, the total amount of faults F_T should ideally be the same for both. Otherwise, a technique with more uncovered faults could in fact present higher fault coverage simply because it presents a much higher F_T. Note that the high fault masking observed for CG-DMR can significantly reduce its F_T, leading to an apparent reduced coverage. Thus, in order to maintain a fair comparison, we use the F_T values of FG-DMR also when calculating the fault coverage of CG-DMR.

Figure 5.9 shows the obtained results, which are quite high for both techniques. CG-DMR presents an average coverage of 99.62%, whereas for FG-DMR it is 99.58%, i.e., a 0.04% difference. The circuits with lower coverage are the ones with a large amount of primary outputs per LUT (mainly *bigkey*, *des*, *ex5p* and *s298*), which is in keeping with the discussions section 5.1, i.e., that POs introduce critical routing segments. If higher coverage is required, reliability-oriented routing can be used, such as the approaches presented in (KASTENSMIDT, FILHO and CARRO, 2006) and (STERPONE and VIOLANTE, 2006).

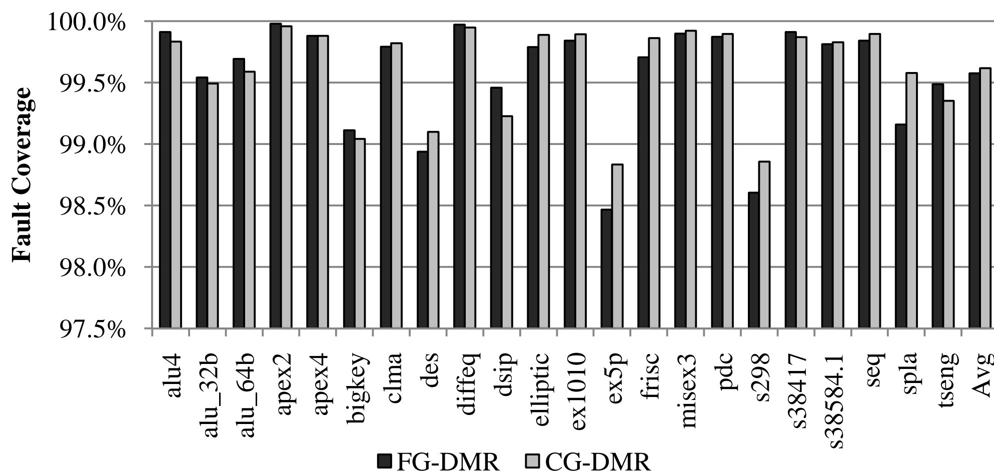


Figure 5.9: Fault coverage for FG-DMR and CG-DMR.

5.3.4 Detection acceleration

As the reduction on error detection time is among the advantages of fine-grained techniques, in this section we compare the average time each technique requires to detect the presence of an error. Since many of the faults in category 2 observed for FG-DMR were not detected by CG-DMR during the experiment due to the reduced observability, it is not possible to determine their detection time. Thus, we focus our analysis on those faults that indeed propagated to primary outputs (i.e., category 3). The developed fault injection tool monitors C_{Det}, the amount of cycles required to detect the presence of each error, and reports it to the host PC. The fine-grained circuits, however, do not necessarily operate at the same frequency of the coarse-grained ones. When considering that each circuit operates at its maximum frequency, the average error detection time T_{Det} is calculated as shown in (5.1), where T_{Clk} is the clock period and C_{Det} is the average amount of cycles to detect. Table 5.5 shows the results.

$$T_{Det} = T_{Clk} \cdot C_{Det} \tag{5.1}$$

Table 5.5: Average amount of cycles and associated time to detect an error

	CG-DMR		*FG-DMR*	
	C_{Det}	T_{Det} *(ns)*	C_{Det}	T_{Det} *(ns)*
alu4	1800.71	11791.02	42.66	320.73
alu_32b	394.04	3259.90	110.59	1087.54
alu_64b	387.05	3709.84	114.48	1370.93
apex2	11482.31	89998.34	1099.86	11941.15
apex4	211.61	1647.40	193.21	1889.43
bigkey	37.19	169.67	15.71	127.53
clma	20.13	146.64	12.66	150.31
des	66.98	474.00	46.83	396.26
diffeq	3708.65	18446.82	19.52	186.44
dsip	92.91	384.66	45.84	378.40
elliptic	2096.16	7342.85	5.63	38.88
ex1010	480.60	3178.23	33.53	254.51
ex5p	109.23	528.02	62.75	350.66
frisc	11139.11	92766.53	5769.30	87024.11
misex3	1327.88	9783.81	407.68	4090.67
pdc	4655.00	41620.39	171.72	1832.23
s298	2076.20	5802.97	2020.98	8255.69
s38417	982.62	5512.52	289.59	3507.26
s38584.1	5354.82	37794.30	648.89	7270.22
seq	8187.95	60525.35	2200.91	22090.51
spla	3724.51	21654.28	466.82	3237.39
tseng	1985.94	11466.82	547.50	4985.54
Average	2741.89	19454.74	651.21	7308.47

Figure 5.10 shows the reduction in C_{Det} observed with the use of FG-DMR. Circuits display very diverse values as different functions have different masking probabilities. For example, the XOR function will always propagate an error on one of its inputs,

while the AND function will mask it as long as another input equals '0'. On average, a 66.2% reduction was observed, with a maximum of 99.73% for *elliptic*.

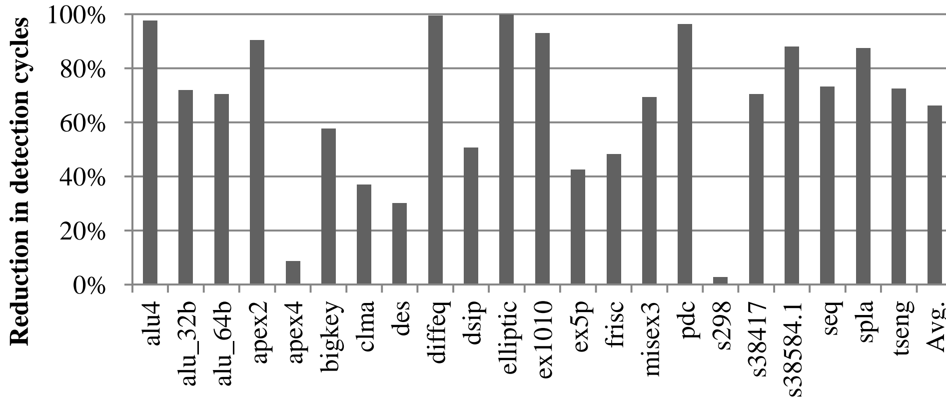


Figure 5.10: Reduction in cycles to detect an error

Figure 5.11 shows the reduction in error detection time made possible by FG-DMR. Negative values indicate situations for which the coarse-grained approach was faster, i.e., $T_{DetCG} < T_{DetFG}$. This happens due to a combination of two factors: 1) the circuit masking probabilities are low, leaving small room for improvements on C_{Det}; 2) the clock period is longer for FG-DMR, meaning that fewer input vectors are applied per unit of time. For the majority of circuits, however, the improvements in fault observability were able to significantly reduce the error detection time, with *elliptic* displaying once more the most significant reduction (99.47%). This is in agreement with other hints that this circuit had a high masking probability: it has only 2 PO bits, and its variation in F_2 for CG-DMR and FG-DMR is very significant (~45×). On average, a 50.15% reduction was observed.

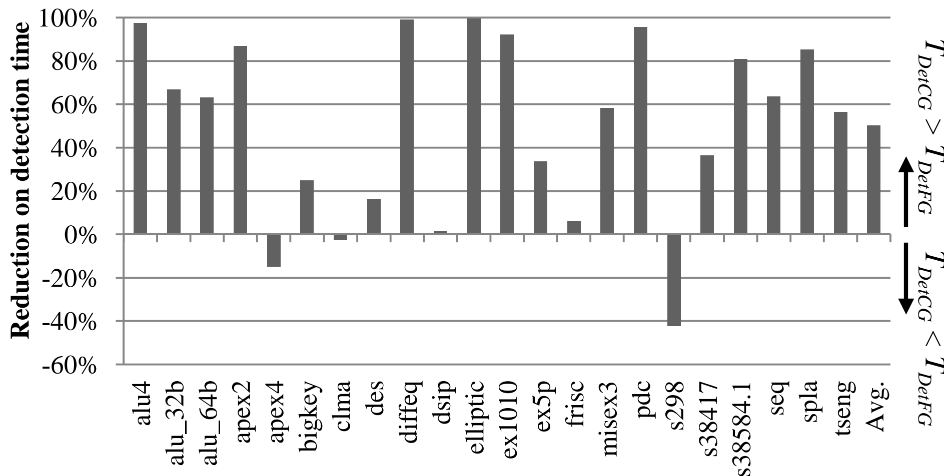


Figure 5.11: Reduction in error detection time

5.4 Radiation Experiments

With the purpose of evaluating the resilience of the proposed mechanism when subject to actual radiation and also of validating the conducted fault injection campaigns, radiation experiments were performed and are herein described and analyzed. Experiments took place at the VESUVIO facility in ISIS, Rutherford Appleton Laboratories in Didcot, United Kingdom. We irradiated the device with a fluence of approximately $1.5 \cdot 10^{10}$ n/(cm^2) with the available spectrum (shown in Figure

5.12), which has already been demonstrated to be suitable to emulate the atmospheric neutron flux (VIOLANTE, STERPONE, MANUZZATO, *et al.*, 2007). The available flux was of approximately $4.5 \cdot 10^4$ n/(cm$^2 \cdot$s) for energies above 10 MeV. The beam was focused on a spot with a diameter of 3 cm plus 1 cm of penumbra, which is enough to cover the whole FPGA chip. Irradiation was performed at room temperature with normal incidence.

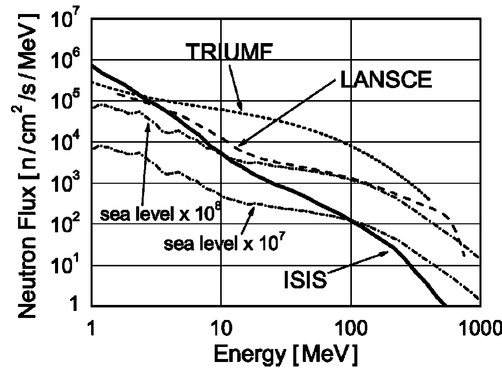


Figure 5.12: ISIS spectrum compared to those of the LANSCE and TRIUMF facilities and to the terrestrial one at sea level multiplied by 10^7 and 10^8 (VIOLANTE, STERPONE, MANUZZATO, *et al.*, 2007)

5.4.1 Tested circuit

The experiments were conducted on a XUPV5-LX110T board, which contains a Xilinx Virtex 5 XC5VLX110T FPGA, i.e., the same device and board on which the fault injection experiments were conducted. This FPGA is manufactured with a 65 nm process (XILINX, INC., 2009b). The board was connected to a host PC with a serial and a USB cable, needed for experimental result transmission and FPGA reprogramming, respectively.

The design implemented in the FPGA comprises a control unit and 26 copies of the circuit under test (CUT), aiming at increasing the device occupation and, thus, the amount of observed events. The CUT used was the *apex4* circuit. Figure 5.13 shows the placement of the modules, with individual CUT identifiers, as well as of the control unit (in dark grey). The even numbered CUTs (in light grey) are hardened with FG-DMR, while the odd numbered ones (in white) use the standard CG-DMR.

The *apex4* circuit was chosen among other benchmarks due to its intermediate size (655 LUTs) and relative small amount of inputs and outputs (9 and 18, respectively)

0	1	2
3	4	5
6	7	8
9	10	11
12	13	14
15	CTRL	16
17	18	19
20	21	22
23	24	25

Figure 5.13: Placement of replicas and control unit. Even numbered replicas (in light gray) used the proposed FG-DMR while odd numbered ones used CG-DMR.

compared to the other circuits of the suite. The reduced amount of inputs and outputs simplifies the routing of the multiple circuit instances, while the intermediate circuit size allows reaching a higher device occupation with fewer copies than with other circuits, simplifying the control circuitry and reducing its probability of being hit by faults. Each circuit with FG-DMR required 1377 LUTs, while those with standard CG-DMR required 1324 LUTs. The circuits used in these experiments do not use duplicated primary inputs, in order to minimize the routing interference over the CUTs.

Pseudo-random inputs are applied with a linear feedback shift register to all the CUTs and a "*golden*" instance. Each of the 26 CUT copies has the same 4 possible states described in section 5.2: 1) normal execution; 2) error in the configuration memory detected by the comparators but not observed at POs; 3) error detected and observed at POs; 4) error not detected but wrong POs. Whenever a CUT leaves the normal execution state, the transmission of a *faulty state description* (FSD) is triggered, informing the host PC about the current state of all CUT instances. It contains 2 bits per instance: one to indicate if the fault was detected and one to indicate if the fault manifested at a PO. Each FSD has, thus, 52 bits, which are transmitted in 7 bytes. Note that several upsets may occur in the configuration memory before an error is observed in the circuits, due to the single event upset probability impact (SEUPI) de-rating factor (LESEA, DRIMER, FABULA, *et al.*, 2005). All cross-section and failure rate values measured in the radiation experiments, thus, are dynamic, and reflect the susceptibility of the user circuit atop the FPGA fabric (FULLER, CAFFREY, SALAZAR, *et al.*, 2000).

A control unit was added for applying input vectors to all the CUT copies, checking the correctness of the outputs and transmitting FSDs to the host PC through the serial cable. The control unit was positioned in the center of the FPGA (see Figure 5.13) so that it enclosed the clock and serial transmission I/O pins, located at that region. It uses 1037 LUTs and 238 registers and comprises the golden instance of the original unhardened *apex4*, with 655 LUTs. Faults in the golden instance can be easily detected, as the system will inform that all the 26 CUTs have incorrect outputs, creating a FSD that differs radically from those received when a fault strikes one of the CUTs. The remaining 382 LUTs and 238 registers are responsible for monitoring and transmitting the FSDs.

As the control unit is embedded on the same FPGA of the CUTs, it requires a mechanism to monitor its integrity. Therefore, it periodically transmits an "*alive*" signal to the host PC through the serial cable. We add a watchdog on the host PC that reprograms the FPGA if the *alive* signal is not received for more than 3 seconds, allowing the detection and removal of faults on the control state machine, clock distribution or transmission circuitry. Finally, if no FSD is received after 10 minutes, the device is preventively reprogrammed, even if the *alive* signals are being received properly. This allows avoiding situations in which the system is still sending the *alive* signal but is no longer checking the output of the CUTs or is unable to send a FSD. All reconfigurations are performed by the remote host PC over a USB cable.

After the transmission of the FSD, the control unit waits 100,000 cycles, latching state changes that may occur for the CUTs during this period. A new FSD is then transmitted. This allows finding with greater accuracy if the fault could affect a primary output and is important especially when fine-grained detection schemes are used. If a scheme is able to perform early detection, then the first FSD may indicate it before error manifestation at a PO, while the second one, after 100,000 cycles, indicates if the fault

eventually propagated to a PO. As the system runs at 50 MHz, we consider the probability of another SEU occurring in the 2 ms waiting period to be negligible ($P \approx 7 \cdot 10^{-6}$). After the reception of the second FSD, the host PC reprograms the FPGA to initiate a new round of the experiment.

5.4.2 Neutron experiments results

Each of the events reported by the control unit was classified, according to its FSD, into the same categories described in section 5.2. Events of category 1 are not reported by the monitoring system, as FSD transmission is only triggered when one or more CUTs leave the normal execution state. Table 5.6 shows the amount of events reported for each category, for the two techniques. The results labeled "Pre" are those obtained by the FSD sent before the 100,000 cycles waiting period, and those labeled "Post" were obtained after it.

Table 5.6: Received events classification

	2) Det. Only	*3) Det & PO*	*4) PO Only*
CG-DMR "Pre"	244	221	6
FG-DMR "Pre"	471	211	5
CG-DMR "Post"	245	223	6
FG-DMR "Post"	396	287	5

For FG-DMR the amount of "Pre" errors in category 2 (detected but with correct output) is larger than that of "Post" errors in this same category. This is caused by the fact that the fine-grained comparators are frequently able to detect the error before it is present at a PO, signaling it to the control unit. Then, during the waiting period, different input vectors may make the error propagate to a PO, moving the event to category 3. As local repair procedures may commence after detection, this property is useful to reduce error removal times. And as a standard DMR scheme is only able to detect errors that have already propagated to a PO, a longer period of time has to be waited before starting repair procedures. The amount of errors in category 4 (error not detected but PO corrupted) does not present a statistically significant difference to allow comparing FG-DMR and CG-DMR, but was a small fraction of total amount of events for both approaches.

The different response times of the circuits also explain the slight increase in the total amount of events between the "Pre" and "Post" results. This occurs due to faults that strike routing resources in the border regions between CUTs and that disrupt the operation of multiple instances. There were three such events, to which we refer as multi-CUT events, as more than one instance of the circuits under test manifested the occurrence of a fault. There is a probability that a multi-CUT event is actually triggered by multiple and independent SEUs. However, as all such events occurred with neighboring CUTs and due to the short duration of the waiting period (2 ms), compared to the observed error rates (around one error every 5 minutes), we attribute them to single errors that affect multiple circuits, which is, as mentioned, a well known and documented effect (LIMA, CARMICHAEL, FABULA, *et al.*, 2001). Figure 5.14 shows the location of the multi-CUT events detected throughout the experiment.

In the first of such events, in the "Pre" FSD, sent immediately after the event occurrence, only CUT #20 reported that a fault was detected. Hence, the error is classified into category 2 for "FG-DMR Pre", since even numbered CUTs are the ones using the fine-grained scheme. In the "Post" FSD, sent after 100,000 cycles, CUT #20

0	1	2
3	4	5
6	7	8
9	10	11
12	13	14
15	CTRL	16
17	18	19
20	21	22
23	24	25

Figure 5.14: Disposition of multi-CUT events on the FPGA. All such events occurred with neighboring CUTs.

indicated that a fault was detected and that it manifested an error at a primary output. Thus, it is one of the events that shifted from category 2 to category 3 during the waiting period for FG-DMR. However, the "Post" FSD also indicated that an error was detected and present at a PO for CUT #21. These two CUTs are adjacent, as can be seen in Figure 5.14.

The second multi-CUT event started with CUT #2 indicating the detection of an error. Then, after the waiting period, CUTs #1, #2 and #5 indicated error detection, while only CUTs #2 and #5 manifested an error at a PO. As shown in Figure 5.14, all three involved instances lie on the top right corner of the device. This event was probably triggered by an SEU that affected the original instances of CUTs #2 and #5 and the redundant copy (or comparison circuit) of CUT #1. As occurred with the event described above, this event shifted from category 2 to category 3 for FG-DMR.

Finally, the third event began with CUT #21 indicating error detection at the "Pre" FSD. In the "Post" FSD, CUT #24 also reported error detection. None of the CUTs presented error at a primary output, indicating that the SEU probably affected only redundant copies or comparison circuits.

Table 5.7 shows the cross-sections found with the conducted experiments, considering the amount of undetected errors (category 4) of each technique. The cross-section is the ratio of errors to fluence, as described in section A.4 of Appendix A. To evaluate the effectiveness of the proposed approach on a typical terrestrial application, Table 5.7 also reports the expected failures in time (FIT) at New York City considering a flux of 13 n/($cm^2 \cdot h$) (above 10MeV) (JEDEC, 2006). Results are shown both for all the 13 CUTs of each circuit type and for one single instance.

Table 5.7 reports the cross-section and FIT figures for an unhardened circuit as well. As can be seen in Figure 5.5, the events in category 3 (errors detected and observed at a PO) of CG-DMR are likely to be the errors at the primary copies of coarse-grained circuits. Thus, we assume that they are a good estimate of the amount of failures that

Table 5.7: Cross-section and failure in time at New York City

	Total		*Per Instance*	
	Cross-section (cm^2)	*FIT*	*Cross-section (cm^2)*	*FIT*
CG-DMR	$3.875 \cdot 10^{-10}$	5.04	$2.98 \cdot 10^{-11}$	0.388
FG-DMR	$3.23 \cdot 10^{-10}$	4.2	$2.48 \cdot 10^{-11}$	0.323
Unhardened	$1.44 \cdot 10^{-8}$	187.25	$1.11 \cdot 10^{-9}$	14.4

would be observed in unmitigated circuits, thereby estimating their cross-section and FIT values. This assumption is based on the fact these are the faults that actually struck the original instance of the circuit, which is similar to the original circuit, since comparison is performed only at the POs. The FG-DMR technique was able to reduce the failure rate by 44.6 times, as the amount of undetected errors is much smaller than that of PO failures of the unmitigated design. Even when considering all circuit instances, the FIT values of DMR circuits are quite low, especially when compared to those of unmitigated designs.

Figure 5.15 shows the total amount of events for each circuit. All copies were subject to a significant amount of events, showing that the FPGA was homogeneously struck by the neutron beam. The cell in dark grey in Figure 5.15 is the control unit, which presented 57 failures, 36 of which are due to watchdog timeouts. The remaining 21 failures are due to invalid FSDs that indicate errors in the golden copy, as described in section 5.4.1. These FSDs indicate faults striking the golden instance or the FSD generation circuitry.

59	48	46
31	44	45
41	47	50
36	48	28
51	25	42
35	57	68
33	59	32
60	41	70
39	50	35

Figure 5.15: Events reported at each instance.

5.4.3 Comparison to fault injection results

As one of the purposes of the conducted experiments was to validate the accuracy of the fault injection tool, in this section we compare the results obtained with both evaluation approaches. Most specifically, we are interested in analyzing if the relations between FG-DMR and CG-DMR observed in fault injection are kept in the radiation experiments. For that purpose, fault injection experiments were conducted aiming at reproducing the radiation experiments. The *apex4* circuit with FG-DMR was placed at the position of CUT #0, while the one with CG-DMR was placed at the position of CUT #1. Both circuits were subject to exhaustive fault injection, leading to a significantly larger amount of events, when compared to the radiation experiments. The injection tool informs which faults were first only detected and then propagated to a PO (i.e., faults that would be in category 2 for “Pre” FSD and in category 3 for “Post”) and which faults were only detected when they had already propagated to a PO (i.e., faults that would be in category 3 already in the “Pre” FSD).

Table 5.8 shows the results for the “Pre” FSDs. For each of the experiments, the results for FG-DMR and CG-DMR are shown, as well as the ratio between them. For categories 2 and 3, as well as for the total amount of events, the ratios showed a strong similarity, with a maximum of 7.05% variation for category 2. Category 4 shows a very significant variation, confirming that the results found in radiation experiments were not sufficient to allow comparing both approaches regarding these ratios.

Table 5.8: Fault injection and radiation results for “Pre” FSDs

	Radiation			*Fault Injection*			
	FG-DMR	*CG-DMR*	*Ratio*	*FG-DMR*	*CG-DMR*	*Ratio*	*Ratio Variation*
2) Det. Only	471	244	1.93	103697	57797	1.79	7.05%
3) Det. & PO	211	221	0.95	55876	55664	1.00	-4.89%
4) PO Only	5	6	0.83	571	193	2.96	-71.83%
Total	687	471	1.46	160144	113654	1.41	3.52%

Similarly, Table 5.9 shows the results for “Post” FSDs. The ratios for categories 2 and 3 become even more similar, while category 4 remains the same, as expected. A small variation is observed in the total amount, due to the multi-CUT events that modifies the ratio for radiation experiments. A very strong similarity is maintained, showing that the fault injection and radiation experiments were consistent in those situations for which consistency was expected.

Table 5.9: Fault injection and radiation results for “Post” FSDs

	Radiation			*Injection*			
	FG-DMR	*CG-DMR*	*Ratio*	*FG-DMR*	*CG-DMR*	*Ratio*	*Ratio Variation*
2) Det. Only	396	245	1.62	89872	57755	1.56	3.73%
3) Det. & PO	287	223	1.29	69701	55706	1.25	2.86%
4) PO Only	5	6	0.83	571	193	2.96	-71.83%
Total	688	474	1.45	160144	113654	1.41	3.01%

6 FINE-GRAINED DIAGNOSIS AND LOCAL REPAIR

As discussed previously, the main advantages of fine-grained redundancy are twofold: faster detection and more precise diagnosis. In chapter 5 we have estimated the reductions on detection times that a very fine-grained mechanism can provide. In this chapter, we focus on the how the fine-grained diagnosis can be used, in a scalable manner, to allow localized scrubbing with significantly reduced repair times. This task presents several challenges, as will be discussed in section 6.1. Section 6.2 presents the devised approach to tackle the challenges found, deemed Scrubbing Unit Repositioning for Fast Error Repair (SURFER). The experimental setup used to evaluate the SURFER mechanism is described in section 6.3. It is based on that presented in section 5.2, but includes several extensions in order to allow proper evaluation of the techniques herein discussed. Section 6.4 presents the results obtained with SURFER assuming a precise translation mechanism, which is valuable to estimate the potential of the technique. Finally, section 6.5 introduces a heuristic mechanism that aims at implementing SURFER with manageable costs and maintaining relevant gains in repair time.

6.1 Challenges

Among the most promising features of fine-grained error detection mechanisms is the possibility of using the precise diagnosis, provided by multiple error detection bits, to perform a local and fast repair procedure. All error indication signals can be concatenated and seen as an *error signature*, as shown in Figure 6.1. The signature contains all the raw diagnosis information provided by the detection mechanism, which must be translated into information useful for repair. As the granularity gets finer, the size of the redundant modules is reduced and the amount of error signals increases. Therefore, circuits with very fine granularities have the greatest potential of reducing the MTTR, but present very large signatures to be handled. And several challenges are found when aiming at translating large signatures into error locations.

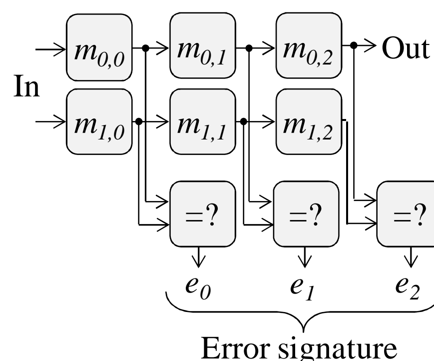


Figure 6.1: Fine-grained detection and the generated error signature

Firstly, the inputs of the two modules may present different values due to a fault in a preceding circuit. Such input difference may be propagated by the logic implemented in the module, thus also triggering its associated comparator. For example, a fault striking module $m_{0,0}$ in Figure 6.1 will be detected by comparator e_0 whenever it propagates to the module's output. If the change in the output of $m_{0,0}$ causes a change in the output of $m_{0,1}$ (i.e., $m_{0,1}$ propagated the error), then e_1 will also be raised, and similarly for e_2. Otherwise, i.e., if the error was masked by $m_{0,1}$, e_1 will remain low. The propagation or masking of an error by a module depends on many variables, both *static*, e.g., the logic function implemented by the module, and *dynamic*, e.g., the current value of the other inputs of the module or the state of internal registers. Thus, several different signatures are possible for a single fault, especially when complex circuit topologies and functions are considered, since dynamic factors may change which comparators are triggered. Assuming that [e_0, e_1, e_2] is the error signature for this circuit, [1, 0, 0], [1, 1, 0] and [1, 1, 1] are possible signatures for a fault in $m_{0,0}$ (or in $m_{1,0}$).

Furthermore, unless the function of $m_{0,0}$, $m_{1,0}$ are entirely configured by one single configuration frame, there are multiple candidate frames once a given signature is generated. The reconfigurable routing resources of FPGAs also play an important role on this matter. For example, a fault in the routing between $m_{0,0}$ and $m_{0,1}$ may occur either before or after the branching point of the wire connected to the comparator. If it occurs before, then it will behave similarly to a fault in $m_{0,0}$, as it will be detected also by e_0. On the other hand, if it is located after this point, then it will only be detected by e_1, provided $m_{0,1}$ propagates it. Thus, signatures such as [1, 0, 0] and [0, 1, 0] can be associated with the routing between the two modules. Depending on the choices of placement and routing algorithms, this routing path may be arbitrarily long and span across several different configuration frames. Therefore, as a general rule, it cannot be assumed that it is possible to narrow a fault location down to a single configuration frame, even when the finest available granularities are employed. Note, however, that the *probability* of each frame generating a given signature is different, depending also on the static and dynamic factors involved. This property can be explored to overcome the challenges herein discussed and minimize repair time, as will be seen in section 6.2.

To summarize, the problem at hand consists in identifying the most likely error locations for a given error signature, which may be very long for large circuits and fine granularities, and to make use of this information to reduce the MTTR. It must be also taken into account that: a same error may lead to different signatures depending on dynamic factors; a same signature may be caused by errors in different locations and with different probabilities.

6.2 The SURFER approach

6.2.1 Overview

The proposed Scrubbing Unit Repositioning for Fast Error Repair (SURFER) technique uses a signature translation (ST) mechanism to convert the error signature into an indication of the error location. This indication is provided in the form of a frame address, chosen according to the methodology described in the sections 6.2.2 and 6.2.3. As mentioned previously, the configuration of FPGAs is divided into frames, which are the smallest addressable units. For the Virtex 5 devices used as case studies in this work, addresses are composed of several subfields, such as the top/bottom bit, row number, major and minor addresses (XILINX, INC., 2011a). The ST mechanism is

defined in a manner that delivers the error location following this specification, in order to avoid additional complex post-processing that may increase the repair time.

Figure 6.2 shows an overview of a system using SURFER. Similarly to (BOLCHINI, MIELE and SANDIONIGI, 2011), we assume the existence of an external configuration controller that interacts with the non-volatile memory that stores the configuration. Note that a non-volatile memory and a controller able to interact with it are already required by any system using SRAM-based FPGAs. As this controller is very simple, it can be implemented in a lower performance radiation-hardened FPGA or ASIC. Alternatively, in very low budget situations and when the reduction in reliability is acceptable, it can be implemented within the same FPGA, as in (STRAKA, KASTIL and KOTASEK, 2010). The ST mechanism, on the other hand, is performed in the same FPGA to minimize its delay and to avoid excessive pin use.

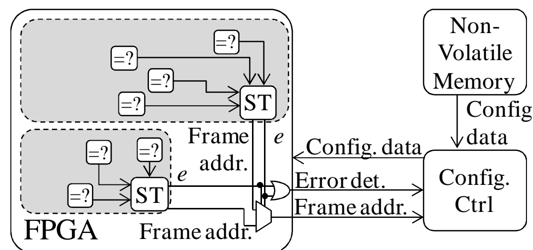


Figure 6.2: Overview of a system with SURFER

In Figure 6.2, the FPGA design is divided into modules, each with its own ST block (we present only two modules for the sake of clarity). Each block generates an error detection bit and a frame address. Moreover, in order to provide fault isolation between the modules, they can be defined as reconfigurable partitions as well. However, developers are free to divide the system into modules as they see fit, following the good practices of design modularization, since the gains in repair time are not limited by their size. Thus, the costs of defining very small reconfigurable partition can be avoided.

6.2.2 Reducing the MTTR through optimized starting frames

In this work we exploit the fact that the scrubbing procedure does not need to begin at the first frame of the configuration, but instead an improved *starting frame* can be identified for each signature. If, for example, the signature indicates that there is a strong probability of the error being in the 300th frame of the partition, a *shifted scrubbing* procedure, starting closer to that position, can significantly reduce the MTTR. If the end of the partition is reached and the error is not removed, then the procedure returns to beginning of the partition and continues until the previous starting frame. As discussed in section 6.1, each signature may be associated with errors in different frames with different probabilities and pointing to a single frame once the error is detected can be infeasible. One must rely on the information of which are the most likely faulty frames for each signatures and give them some form of priority. Thus, the first step to allow the use of SURFER is to measure the relations between errors in each frame and the generated signatures. Through fault injection experiments it is possible to identify which configuration bits are able to generate each signature when flipped. Thereby, one can build histograms that show, for each signature, which frames can lead to its occurrence and with which incidence. These histograms allow identifying the most likely error locations associated with each signature. Figure 6.3 shows two such histograms, for two different signatures of circuit *pdc*. More details on the conducted experiments will be provided in section 6.3.

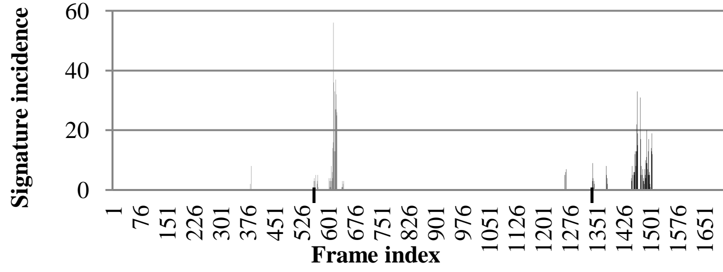


Figure 6.3: Histograms of two signatures for the *pdc* circuit

Once the relations between error locations and signatures are mapped, remains the problem of efficiently making use of this information. The histograms in Figure 6.3 present clear peak regions, where the error is most likely located. However, if one were to scrub only these peak regions, there would be a probability (although small) of not correcting the fault, as it can be located outside peak regions. Furthermore, two frame addresses would have to be stored per signature (the first and the last addresses of the area), creating the need for large and costly tables. Errors and approximations in this table would also be critical, as they could lead to scrubbing the wrong area. When setting a shifted starting frame, on the other hand, even if the signature translation module makes a poor guess regarding the error location, the entire partition will eventually be scrubbed if needed, thereby maintaining the reliability of a standard scrubbing procedure.

The user circuit can be halted when the error is detected, as in (PSARAKIS and APOSTOLAKIS, 2012), and scrubbing can ensue until the error is reached and removed. Correction can be detected, in many cases, by the lowering of error signals. Alternatively, it may be advantageous to perform a readback, comparing each frame to the expected value (or using redundancy codes) to first locate the error. The identified faulty frame is then solely scrubbed, similarly to (GOKHALE, GRAHAM, JOHNSON, *et al.*, 2004). The proposed scheme remains identical regardless of these device and application specific implementation choices. Once the error is removed, scrubbing can be halted and execution can resume.

The marks on the *x* axis of Figure 6.3 show the optimum starting frame for each of the two signatures. Note that, for both histograms in Figure 6.3, there is a possibility that the error is located before the chosen starting frame. These locations are only scrubbed after reaching the partition end and returning to its beginning. Placing the starting frame before those locations, however, would increase the time required to reach the highest concentration areas, increasing the average correction time.

6.2.3 Optimum frame identification

In order to identify the optimum starting frame for each signature *s*, we calculate the estimated $MTTR_s(f)$ for each possible starting frame *f*. It is defined as:

$$MTTR_s(f) = \frac{FS}{BR}\sum_{i=PB}^{PE}\frac{h_s[i]}{O_s}\cdot\left(dist(i,f)+1\right). \qquad (6.1)$$

Where *f* is the starting frame, *FS* is the frame size, *BR* is the configuration port bit rate, *PB* is the partition beginning and *PE* is the partition end. $h_s[i]$ is the histogram value for signature *s* for the *i*-th frame and O_s is the total amount of occurrences of *s*. Therefore, $h_s[i]/O_s$ is the probability that the error is located in the *i*-th frame, whenever signature *s* is received. $dist(i, f)$ is the distance between *f* and the *i*-th frame, i.e., the amount of frames that have to be written before reaching the *i*-th. It is defined as:

$$dist(i,f) = \begin{cases} i-f, \text{ if } i \geq f \\ PE - f + 1 + i - PB, \text{otherwise.} \end{cases} \tag{6.2}$$

The sum in (6.1) is, therefore, the "mean frames to repair" when s is received and f is used as starting frame. It is converted to a time unit with the time required to write a frame (FS/BR). There may also be additional costs associated with interacting with the programming interface, such as issuing a write command. Such costs are device-dependent and thus not shown in (6.1). Furthermore, they are usually negligible when compared to the time required to transmit the configuration data, but are nonetheless taken into account in the experimental results reported in this work.

In (6.2), the first condition is the distance between f and i if f, the starting frame, is before i. In this case, the error is corrected before reaching the end of the partition. The second condition occurs when the error is only corrected after reaching the end of the partition and returning to its beginning. In this case, $PE - f + 1$ is the amount of frames written until the partition end and $i - PB$ is the distance between the partition beginning and i.

With (6.1) and (6.2) one can calculate the expected MTTR for each possible starting frame and select the smallest one as the optimum choice for signature s. This is repeated for all the different signatures that occurred for the circuit. Let O denote the total amount of received signatures, as shown in (6.3) and S the set of all *different* signatures. The overall $MTTR$ is defined by the average of all signatures, weighted by their occurrences, as shown in (6.4).

$$O = \sum_{s \in S} O_s \tag{6.3}$$

$$MTTR = \sum_{s \in S} \frac{O_s}{O} \cdot MTTR_s(f_s) \tag{6.4}$$

One can then build a table that indicates, for each signature s, its optimum starting frame f_s. This table provides the optimum ST mechanism for SURFER in terms of MTTR reduction. For this reason, we refer to it as perfect signature translation (PST), and it is a relevant mechanism to measure the maximum gains of SURFER. Its benefits and drawbacks are discussed in section 6.4, following the experimental setup described in section 6.3.

6.3 Extended experimental setup

The experimental setup presented here extends that described in section 5.2 and consists in several tools required to evaluate the proposed techniques, as shown in Figure 6.4. The entire setup is divided into macro-steps for the sake of clarity, which are detailed in the remainder of this section.

The *first step* is the same performed in the setup described in section 5.2, i.e., a synthesized description of the original HDL design is generated with the standard synthesizer XST and *netgen*, which is then subject to the redundancy insertion tool that applies the carry chain-based fine-grained DMR. In this case, however, the error aggregation circuit is not instantiated, as we are interested in observing the individual error indication bits that form the error signature. Table 6.1 presents the total signature size S_{size} for each circuit.

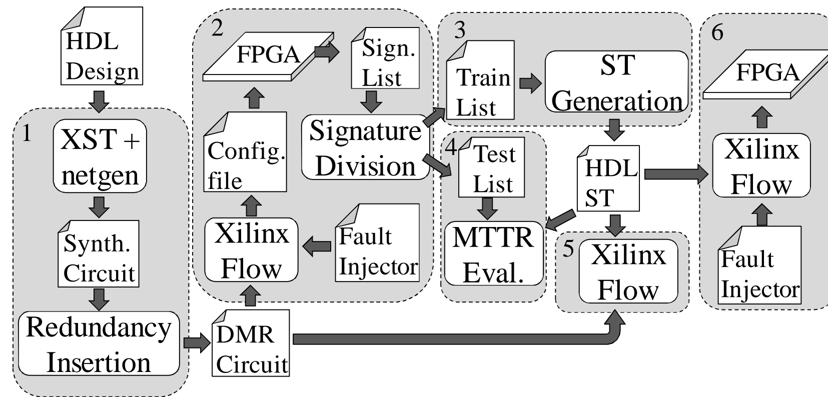


Figure 6.4: Extended experimental setup

The *second step* consists in extracting error signatures are associated with each injected fault. The injection tool described in section 4.3 was extended to transmit the generated signatures to the host PC, along with the frame address on which the fault was injected. We once more perform exhaustive injection, i.e., faults are injected on every bit associated with the CUT. As previously, 100,000 pseudo-random input vectors are applied to each circuit for each injected fault. However, as discussed in section 6.1, several different signatures may be generated for each fault, due to the dynamic factors that change propagation in the circuit. To maintain a tractable experiment time, we limit to 20 the amount of signatures transmitted per fault. Still, almost 3 million signatures were received per circuit, on average, as can be seen in Table 6.1. Table 6.1 shows, for each circuit, the total amount of signatures, i.e., *O* as seen in (6.3), and the total amount of different signatures, i.e., */S/*. The experiment is therefore exhaustive only regarding the possible faulty bits and not regarding the possible generated signatures, since faults are injected on all bits but only up to 20 signatures are taken from each.

It is important ensure that the signature sample is statistically significant and that the mechanism is not applicable only to that particular set of signatures. For that purpose, we use an approach similar to that traditionally used with neural networks (HAYKIN, 1998). The *signature division* step shown in Figure 6.4 generates two non-overlapping signature lists, one to be used in the generation of the translation module (*training list*) and one to measure the obtained MTTR (*testing list*). Thus, the evaluation is performed on a list of signatures not available to the generation algorithm. The first 15 signatures received for each fault are placed on the training list and the rest on the test list.

In the *third step*, the translation function is generated based on the signatures in the training list. It can either follow the straightforward PST mechanism described in section 6.2.3 or the heuristic signature translation (HST) algorithm, to be presented in section 6.5. Moreover, it calculates the expected MTTR for the signature distribution observed in the training list. This value, when compared to that obtained in the *fourth step*, i.e., when the test list is applied to the generated function, allows determining if the obtained signatures are representative of the error-to-signature relations for that circuit and if the generated mechanism is not strictly limited to signatures in the training list. All results assume the maximum operating speed of the Virtex 5 SelectMAP interface, which is a 32-bit wide port operating at 100 MHz. These figures can be converted if a reduced transfer rate is being used. We also take into account the time required to issue a write command to the interface (25 cycles in our implementation) and to write a dummy frame, which is required by SelectMAP (XILINX, INC., 2011a). Note that this must be done twice whenever a return to the partition beginning is required.

Table 6.1: Total signature size S_{size}, amount of received signatures (O) and of different signatures ($|S|$) for each circuit

	S_{size}	*Total Signatures*	*Different signatures*
alu4	167	1,785,081	24,017
alu_32b	359	1,756,168	48,215
alu_64b	192	3,567,880	89,343
apex2	395	3,819,021	25,941
apex4	332	3,232,288	31,271
bigkey	354	2,984,645	54,717
clma	609	1,373,711	16,413
des	355	2,962,133	57,043
diffeq	234	740,011	9,928
dsip	370	3,519,234	38,471
elliptic	73	205,020	649
ex1010	215	1,991,867	24,996
ex5p	81	502,924	1,990
frisc	894	4,412,457	54,924
misex3	349	3,245,937	31,787
pdc	603	6,588,236	64,214
s298	11	44,865	84
s38417	884	4,784,611	27,332
s38584.1	1,080	11,681,701	38,573
seq	430	4,215,089	22,344
spla	114	928,254	5,525
tseng	337	1,354,465	25,155
Average	383.55	2,986,164	31,497

The *fifth step* consists in submitting the generated translation table, described in VHDL, along with the DMR circuit to the Xilinx standard design flow to determine area and performance overheads. We also evaluate the resilience of the generated translation tables to faults affecting their configuration, since they are also embedded in the FPGA. This is done through a second round of fault injection experiments, in the *sixth step*. Actual error signatures are used as stimuli and the faulty outputs are transmitted to the host PC for analysis. Each faulty event is then categorized as described in section 6.5.1.2 and the increase it causes to the overall MTTR is computed.

The generated signatures and the resulting ST mechanism are strictly related to the decisions made by the placement and routing algorithms, since components (and routed wires) that change place may also change their associated frames. Thus, for the generated ST mechanisms to be applicable to the final design, it is important to maintain the same placement and routing used for signature generation (second step). This can be accomplished through several means, such as through automatically generated fine-grained placement and routing constraints (e.g., LOC, BEL and DIRECTED_ROUTING (XILINX, INC., 2011d)) or using an incremental design flow (ZEH, 2007), which allows creating partitions whose placement and routing are not modified by changes in other modules.

6.4 PST - Perfect Signature Translation

The Perfect Signature Translation (PST) consists, as described in section 6.2.3, in a table that maps each and every generated signature to the optimum starting frame address that minimizes the MTTR. It is, thus, an important mechanism to estimate the maximum gains made possible by the SURFER mechanism. In this section we present these gains and also discuss the shortcomings of this approach.

The extended experimental setup was applied to the same 22 benchmark circuits used in section 5.3. Table 6.2 shows the obtained MTTR results, in microseconds. The *Standard* approach consists in starting reconfiguration at the first frame of the circuit, i.e., it presents the MTTR obtained with straightforward partition-based scrubbing. *PST Train* is the MTTR associated with the signature list used in the generation of the translation circuit, whereas *PST Test* is that obtained when the testing signature list is applied to the translation function.

Figure 6.5 emphasizes the reductions achieved in MTTR with PST. The average MTTR reduction provided by *PST Test* over standard scrubbing is of 79.65%. The circuit with least gains is *s298*, which showed a 52.9% reduction, due to its very small size which leaves a reduced room for improvements with fine-grained diagnosis. The testing and training results are very similar for all circuits, indicating that signatures

Table 6.2: MTTR of standard scrubbing and SURFER with training and testing signatures (in µs)

Circuit	*Standard*	*PST Train*	*PST Test*
alu4	172.29	31.07	33.59
alu_32b	109.36	27.12	30.07
alu_64b	220.71	39.84	45.52
apex2	228.48	45.23	47.02
apex4	239.74	38.86	40.85
bigkey	194.87	36.28	38.42
clma	325.43	47.39	50.92
des	211.48	29.02	31.07
diffeq	169.67	34.63	36.81
dsip	342.18	52.67	55.85
elliptic	118.08	23.94	24.04
ex1010	196.57	40.55	42.93
ex5p	78.91	18.36	18.69
frisc	507	79.12	82.68
misex3	251.42	44.4	47.16
pdc	415.7	58.23	62
s298	35.84	16.83	16.88
s38417	436.67	74.37	76.34
s38584.1	450.28	84.04	85.95
seq	372.74	61.52	63.36
spla	206.74	31.12	31.9
tseng	137.49	28.36	30.85
Average	246.44	42.86	45.13

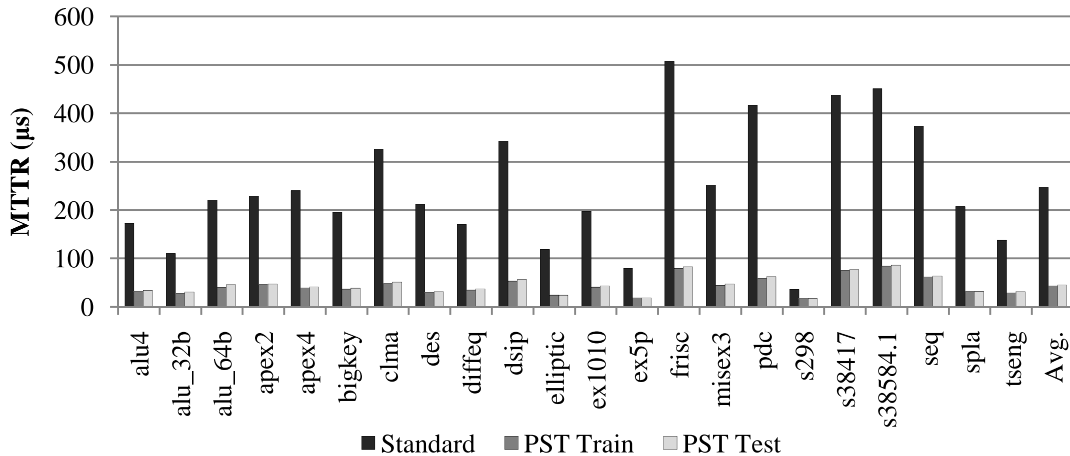


Figure 6.5: MTTR of standard scrubbing, PST with training and testing signatures

used in the testing list were able to appropriately capture most of the error-to-signature relations for each circuit. The average error in *PST Train* relative to *PST Test* is of 5.08%, with a maximum of 12.47% for *alu_64b*.

Implementing PST tables in the FPGA substrate, however, can be very challenging. For *s298*, the smallest circuit (17 LUTs), direct implementation of its PST table requires 119 LUTs, which is 7 times the size of the original circuit. In this case, due to its very small signature size, it is still possible to use BRAMs instead of LUTs to implement this direct translation. But it quickly becomes infeasible for circuits with larger signatures. Still among the smallest circuits (128 LUTs), *ex5p* has 1,990 different 81-bit signatures. Direct implementation of its PST table, however, required 25,290 LUTs (197.58 times the size of the original circuit), showing the poor scalability of this approach. In fact, the synthesis tool runs out of memory before being able to synthesize the PST table for even intermediate-sized circuits. Therefore, in order to provide a scalable variation of the SURFER approach, we propose a heuristic signature translation (HST) mechanism.

6.5 HST - Heuristic Signature Translation

6.5.1 Heuristic table generation

The proposed Heuristic Signature Translation (HST) must be able to quickly provide an initial frame address to be used by the reconfiguration controller. It must also be as small as possible, in order to minimize the area overhead. The mechanism proposed herein works similarly to a hardware-implemented hash table, generating a compressed version of the signature that is then used to access a table containing the target frame addresses. Most of the effort goes into defining an appropriate hash function, which will in turn lead to an efficient table implementation.

As occurs for any function to be used in a hash table, we want to minimize collisions, i.e., different signatures that are mapped to a same compressed counterpart. However, the algorithm should take into account the specific purpose and requirements of the translation being implemented. First of all, not all collisions have the same penalty in terms of the final overall MTTR. Several signatures may have neighboring starting frames, or even point to exactly the same frame. For these cases, collisions have a reduced penalty (or none at all). The hashing function should, therefore, give preference to causing this kind of collision rather than for signatures that point to far away locations. Second, the amount of occurrences O_s is different for each signature s. It is more important to have a precise output for those signatures that are more frequent

than for those that rarely occur. Finally, the signature translation (ST) block must also generate an error detection bit, as can be seen in Figure 6.2, in order to trigger repair procedures. This bit is basically the OR operation performed over the entire signature. If the hashed signature can also be used to generate this bit, then logic resources can be saved.

The pseudo-code shown in Figure 6.6 presents the main steps of the proposed HST mechanism. It consists in first identifying those signature bits that, when active, have a high probability of being associated with the same area of the circuit. We consider that two signatures are in a same area whenever their optimum frames are in a same row and major column. A major column of frames is associated with a column of resources in the FPGA. For example, in Virtex 5 devices most of the major addresses are associated with slice columns, and have 36 frames each (with individual minor addresses) (XILINX, INC., 2011a). Bits that, when active, have a high probability of indicating errors in a same column are iteratively organized into groups. Over each group, the OR function is applied, generating a hashed signature that has one bit per group. Figure 6.7 shows the logic schematic of the proposed mechanism compressing an 8-bit signature into a 2-bit one. And since the hash function is computed with ORs over the signature, the error detection bit e can be generated based on the hashed signature, as shown in Figure 6.7, saving resources. In the remainder of this section, we detail how the HST is generated.

Input: *signList*, a list of all occurring signatures and associated frame addresses, S_{Size}, the size of uncompressed signatures, and *maxSize*, the maximum acceptable compressed signature size.

Output: *gb*, a set of which bits must be grouped and *compAddrTable*, a table with the optimum frame address for each compressed signature

1. *signTable*:= parse(*signList*);
2. *addrTable*:= optimumTable(*signTable*);
3. *gb*:= initialGrouping(S_{Size});
4. **while** size(*gb*) >*maxSize* **do**
5. *G* := buildGraph(*gb*, *addrTable*);
6. *maxMatch*:= maxWeightedMatching(*G*);
7. *gb*:= join(*gb*, *maxMatch*);
8. **end while**;
9. *compSignTable*:= compressTable(*gb*, *signTable*);
10. *compAddrTable*:= optimumTable(*compSignTable*);

Figure 6.6: HST Generation algorithm

The first step of the algorithm (line 1) is to parse the signature list *signList* and to build an appropriate structure to store the information. The list contains all the signatures received by the host PC during the fault injection experiments. It also contains the frame address in which the fault was injected for each signature. These data are stored in the signature table *signTable* that maps each signature to its frame histogram. The histogram is a vector containing how many times that signature occurred for faults injected in each frame.

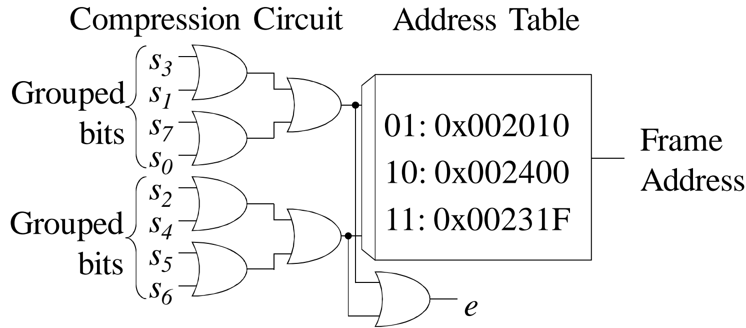


Figure 6.7: Schematic of a HST circuit

The second step (line 2) is to identify the optimum starting frame for each signature, following the methodology described in section 6.2.3. The resulting address table *addrTable* maps each frame address *f* to a set of signatures that have *f* as their optimum starting point.

The third step (line 3) initializes the set *gb* of grouped bits. This set contains the groups of bits that are going to be subject to the OR function, compressing them into a single bit. The initial grouping consists in creating one group for each bit, where that bit is placed alone.

The steps in lines 4 through 8 are repeated until we reach the maximum acceptable compressed signature size *maxSize*. This parameter defines how much effort will be put into compression and will be discussed in greater detail in section 6.5.1.1.

In line 5 the complete undirected group graph $G = (gb, E)$, on which the grouping decisions are to be made, is built. Each set of grouped bits $u \in gb$ is a vertex. As G is complete, there is an edge $\{u, v\}$ in E for every pair of distinct groups $u, v \in gb$. Each edge $\{u, v\}$ is weighted according to the frequency with which u and v are active for signatures that point to a same major address column. A group of bits u is said to be active for a given signature s if at least one of the bits in u is one in s, i.e., the OR over those bits would evaluate to one with s as input. Figure 6.8 describes how the weight of each edge $\{u, v\}$ is calculated. It sums the occurrences of all signatures that point to a

Input: An edge $\{u, v\} \in E$, the address table *addrTable* and the occurrence count O_s for each signature s.

Output: The weight w of edge $\{u, v\}$.

```
1.   w := 0;
2.   for each major column c
3.      o_u := 0; o_v := 0;
4.      for each frame f in c
5.         for each signature s in addrTable(f)
6.            if active(u, s) then o_u := o_u + O_s;
7.            if active(v, s) then o_v := o_v + O_s;
8.         end for;
9.      end for;
10.     w := w + min(o_u, o_v);
11.  end for;
```

Figure 6.8: The weight of an edge $\{u, v\}$

frame *f* in column *c* which *u* is active and does the same for *v*. Then, it adds the minimum of these values to the weight *w*. Thus, the increase in *w* will be zero if either *u* or *v* were never active for the signatures that point to *c*. Moreover, a large value will only be added to *w* when both groups are active for signatures with frequent occurrence. This may also be accomplished by a single signature in which both groups are active and that has a high occurrence count.

Line 6 (Figure 6.6) computes the maximum weighted matching on *G*. It consists in choosing a subset of non-adjacent edges (i.e., that do not share vertices) from *E* that maximizes the sum of their weights. The maximum weighted matching can be computed in polynomial time (EDMONDS, 1965). We use the implementation available with the LEMON graph library (DEZSő, JÜTTNER and KOVÁCS, 2011). One can then join the groups (line 7) according to this matching, maximizing the total frequency with which they are active for signatures that point to a same major column. Thereby, the signature size (i.e., amount of groups in *gb*) is approximately divided in half at each iteration. These steps are repeated until the maximum signature size *maxSize* is reached.

In line 9 the compressed table *compSignTable* is built. It is similar to *signTable* as it contains, for each compressed signature, its occurrence histogram. The compressed signature is computed by applying the OR function over the bits of each group in *gb*. Its histogram is the frame-wise sum of the histograms of all uncompressed signatures that are mapped to it when compressed.

Finally, on line 10, the same calculation of optimum frame address for each signature can be repeated, this time over the compressed table. The resulting compressed address table *compAddrTable* allows mapping the compressed signatures to their corresponding optimum starting frames.

6.5.1.1 The maxSize parameter

The *maxSize* parameter tunes the HST algorithm effort and has significant impact on the resulting translation mechanism. High *maxSize* values reduce the amount iterations of the compression loop (lines 4-8 in Figure 6.6) and allow large compressed signatures. Consequently, the address table stores many different addresses for different compressed signatures, leading to more accurate results but with a higher cost in area. As one reduces the value of *maxSize*, fewer signatures remain due to more collisions that occur, leading to smaller translation tables with less precise results. The design space made available by this parameter will be explored in section 6.5.4.

There are two corner cases that should be highlighted: if $maxSize \geq S_{size}$, where S_{size} is the uncompressed signature size, then the HST and PST tables will be identical, as no compression will take place. Conversely, if $maxSize = 1$, then all bits will be grouped, leading to single-bit compressed signatures and an address table with a single entry. Thus, for any signature, the resulting frame address will be the same. We refer to this address as the *best static address*. Since all signatures will be mapped to the same compressed counterpart, the best static address points to that starting frame that minimizes the MTTR considering all signatures (and their incidence counts) at once. This statically shifted scrubbing, therefore, does not actually exploit fine-grained signatures. Instead, it solely uses the non-uniform distribution of sensitive bits over the frames and may still present reductions in the MTTR compared to the standard approach.

6.5.1.2 Dealing with faults in the translation table

As the translation table is implemented in the same reconfigurable fabric of the circuit it is monitoring, it is also susceptible to the same faults. Thus, it is important to understand their possible effects, their impact in the overall MTTR and how can they be handled. For that purpose, we propose the use of a redundant error aggregation circuit, as shown in Figure 6.9. To minimize area overheads, this circuit does not generate a target frame address, but only the error detection bit (OR over all signature bits, as was done in chapter 5). This allows avoiding the most critical scenarios, as will be discussed herein.

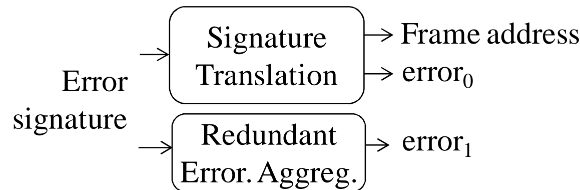


Figure 6.9: Redundant translation table

Two types of table faults are distinguishable: those that manifest themselves immediately and those that remain silent. The first type consists mostly in “false alarm” faults, i.e., faults that cause the error indication bit to be raised even though the input signature is zero. These may occur in the translation table or in the redundant copy, but are detectable, since they will diverge. Furthermore, some faults may cause the frame address output to change while the detection bits are kept low, thus also being detectable. Such faults must be removed upon detection to avoid accumulation.

Faults that remain silent present more complex scenarios. As they are not immediately detected, they may accumulate with faults in the payload circuit. The most evident possible outcome is that an incorrect frame address may be generated. In this case, the generated address may or may not be valid, i.e., among those that the table would normally produce (note that it only generates a restricted set of addresses under normal operation). A silent fault may also prevent the error detection signal from being raised. In this case, upon occurrence of a payload fault, the redundant checkers will diverge.

Considering the discussed scenarios, we propose the following approach. When both detection bits are raised and a valid frame address is generated, that address is used. If the generated address is invalid, the *best static address* is used instead. This avoids, for example, using addresses that are outside the configuration space of that particular partition. Whenever the detection bits diverge, the translation table is scrubbed first, returning it to correct behavior. Thereby the detection signal is lowered in case of false alarms. On the other hand, if it remains high then there is an error in the redundant checker preventing its triggering *and* an error in the payload circuit, which should be scrubbed with the current generated address. Finally, to avoid accumulation of faults, the translation circuitry should be scrubbed after every scrub of the payload circuit.

One can evaluate the impact in MTTR of faults in the translation table considering the overheads introduced by each situation. False alarms require the scrubbing of the translation table to be identified and removed. Faults that cause valid but incorrect frame addresses will have the MTTR associated with the use of that sub-optimal starting frame. Silent faults that prevent error detection require the time to scrub the translation circuit plus the time to scrub payload circuit. By considering the amount of configuration bits (and input signatures) that lead to each situation, one can determine

the total change expected in the MTTR. Moreover, the smaller the translation circuit is, in comparison to the payload, the less likely it is for it to be subject to faults. Thus, minimizing its area is also important to minimize its susceptibility to faults.

6.5.2 Area and delay costs

As discussed previously, the goal of SURFER is not only to provide MTTR reductions, but also to do so with manageable costs and in a scalable manner. In this section, we discuss the area and delay overheads of the proposed heuristic signature translation, considering *maxSize* = 7. The reason behind this choice and the impact of this parameter will be discussed in section 6.5.4.

We take into account two variations of the technique. One attempts to minimize delay overheads by processing error signatures in a pipelined fashion. It first stores the generated error signature to process it in the following cycle. As a result, it requires the use of additional flip-flops. If these are a scarce resource in that particular design, then the alternative combinational approach may be more attractive. Moreover, there may be situations in which the performance is limited by other components of the design and improving the frequency of the module at hand is unnecessary. In such cases, the combinational approach could also be preferable. It calculates the target frame address directly from the comparators' outputs. Therefore, it minimizes the use of flip-flops but introduces additional delay. Note that the single-cycle difference in MTTR observed between both approaches is negligible.

Table 6.3 shows the absolute area occupied by the each circuit, separated into its individual components: comparators, HST table and the redundant error aggregation (EA) circuit. The total figures include the two copies of the original circuits. Figure 6.10 shows the area overhead for each circuit, in terms of occupied LUTs. The results for CG-DMR are also included for comparison. For most circuits, the proposed translation mechanism was able to maintain low overheads. Those circuits with higher costs, *s298* and *ex5p* (212% and 154%, respectively), are also the ones with smallest areas. Most notably, the former has only 17 LUTs in its unhardened form, which leaves small room for the implementation of a translation mechanism with low relative costs. On average, only 15.5% of the amount of LUTs of the unhardened circuit is required to implement the HST translation mechanism. The average total SURFER overhead was 133.9% over the unhardened circuit and 10.5% over CG-DMR.

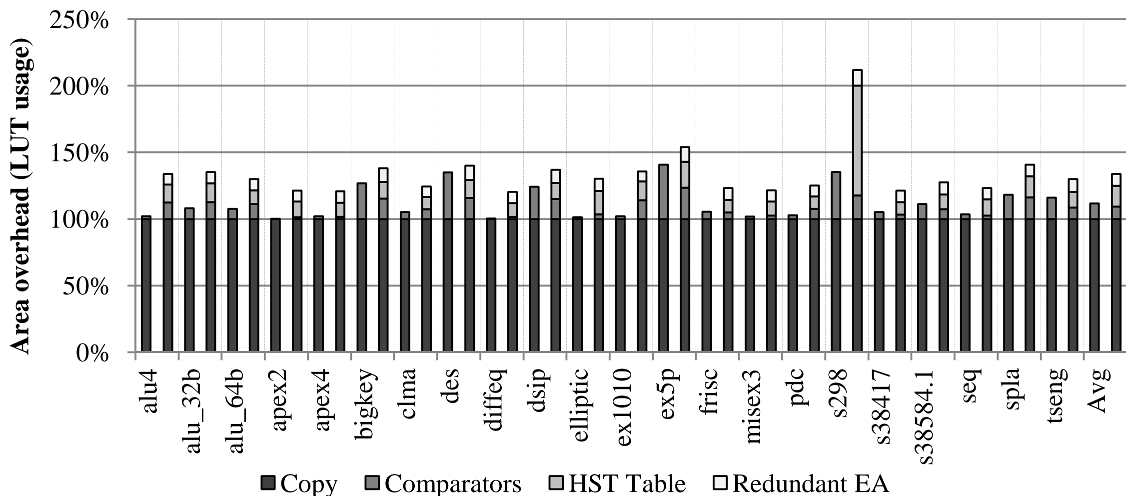


Figure 6.10: Area overhead of circuits with standard CG-DMR (left-hand bars) and circuits with FG-DMR and HST tables (right-hand bars)

Table 6.3: Area and delay results for SURFER

	Area (LUTs)				*Clock Period (ns)*	
Circuit	*Comparator*	*HST Table*	*Redund. EA*	*Total*	*Comb.*	*Pipe.*
alu4	50	54	32	940	8.47	6.81
alu_32b	43	49	28	804	9.83	8.21
alu_64b	81	75	60	1658	12.27	9.25
apex2	10	94	66	1766	11.63	10.39
apex4	11	69	56	1446	10.65	9.19
bigkey	88	72	59	1369	8.21	5.81
clma	92	116	103	2849	12.42	9.6
des	86	74	60	1320	9.25	6.02
diffeq	8	48	39	1035	10.18	7.58
dsip	96	76	62	1504	7.73	5.12
elliptic	5	25	13	329	6.84	5.28
ex1010	69	69	36	1148	8.92	6.43
ex5p	30	25	14	325	6.39	4.42
frisc	85	162	150	3833	15.56	14.65
misex3	17	75	59	1549	10.45	7.67
pdc	96	117	102	2821	11.67	9.5
s298	3	14	2	53	4.58	3.41
s38417	54	161	149	3782	12.16	10.41
s38584.1	149	220	181	4552	11.21	7.23
seq	22	103	72	1889	11.1	8.57
spla	36	35	19	532	7.11	5.33
tseng	51	71	57	1375	10.16	7.89
Average	53.73	82.00	64.50	1676.32	9.85	7.67

The use of flip-flops, on the other hand, depends on the applied variation of the technique. If we implement the translation mechanism as a purely combinational circuit, no flip-flops are introduced on combinational benchmarks, while sequential circuits have exactly 100% overheads, since flip-flops are also duplicated by FG-DMR. For the pipelined version, an amount of flip-flops equal to the signature size S_{size} (found in Table 6.1) has to be introduced. These two approaches, however, are corner cases of several possibilities that may insert flip-flops to register partially compressed signatures and find improved design points in terms of used resources and delay overhead, depending on the specific constraints of each design.

Figure 6.11 shows the minimum clock period for the two implementations of the HST mechanism and for CG-DMR for comparison. The introduction of the HST circuit directly after the comparators (i.e., the purely combinational approach) adds an average of 56.4% delay over standard DMR. As occurred for FG-DMR without SURFER, in section 5.3.2, this delay is particularly more pronounced for the sequential benchmarks (83.9%, on average) then for the combinational circuits (33.7%). This occurs mainly because internal flip-flops may divide the logic path in such a way as to hide the delay of the comparators.

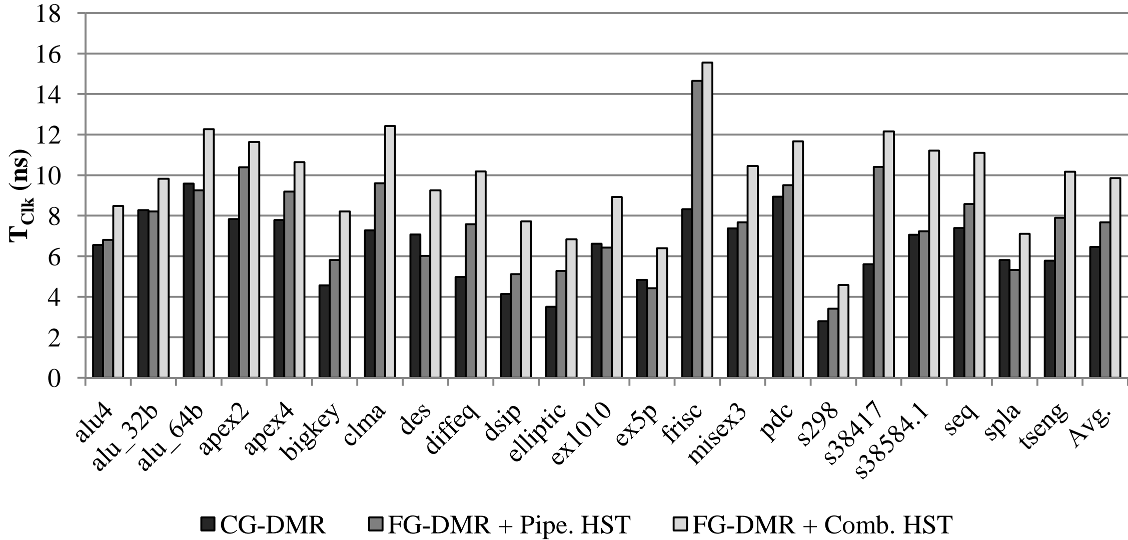


Figure 6.11: Minimum clock period T_{Clk} for CG-DMR and FG-DMR with pipelined and combinational HST

Depending on the requirements of each specific design, the delay overhead of the combinational approach may or may not be acceptable. As an alternative to minimize its effects, we consider the use of a pipelined version, which reduces this overhead by dividing in two steps the generation of the target frame address. The reduction over the combinational approach is very significant for most cases, as can be seen on Figure 6.11, leaving the pipelined version closer to CG-DMR. On average, pipelined HST presents a 20.5% delay increase over CG-DMR. As occurred for the combinational implementation, this difference is more significant for sequential benchmarks (40.8%) then for combinational ones (3.5%). Furthermore, in some cases, especially when the amount of primary outputs is very large compared to the circuit size (such as *des* and *ex5p*), the delay of comparing primary outputs may become very significant and the pipelined approach may even be faster than CG-DMR.

6.5.3 MTTR Results

Table 6.4 shows, in microseconds, the MTTR assuming a fault-free HST circuit (i.e., the results obtained at steps 3 and 4 of the experimental setup in Figure 6.4). It also contains the experimental results to evaluate the impact of faults in the translation table (i.e., obtained at step 6). Both scenarios are discussed in the following sub-sections. We set *maxSize* = 7 in this section as well.

6.5.3.1 MTTR reduction with a fault-free table

Figure 6.12 shows the MTTR obtained with HST for each circuit. It also shows those of standard scrubbing and of PST with testing signatures for the sake of comparison. Although unable to maintain the average gains of PST, as expected, HST presented only a 4.03% increase for the circuit with least gains, i.e., *s298*. As it presents very small signatures, the compression loop required one single iteration to reach the target *maxSize* for this circuit, leading to a very small difference between both techniques. On average, a 95.2% MTTR increase was observed due to the loss of precision caused by the compression heuristic. Nonetheless, HST was able to substantially accelerate repair, when compared to standard scrubbing. On average, a 61.9% reduction was achieved (with the testing list), showing that the proposed heuristic maintains the ability to significantly minimize repair time.

Table 6.4: MTTR (in μs) with fault-free table and with faults in the translation circuit

	Fault-free table		*Faulty table*	
Circuit	*Train*	*Test*	*Golden*	*Faulty*
alu4	57.04	57.06	56.41	58.55
alu_32b	49.83	49.84	49.04	50.58
alu_64b	86.25	86.28	90.45	91.66
apex2	94.02	94.39	92.66	93.80
apex4	89.35	89.37	88.08	89.36
bigkey	60.06	60.06	59.31	61.30
clma	139.31	138.74	138.04	139.82
des	51.88	51.87	50.91	53.69
diffeq	80.45	80.74	79.93	80.20
dsip	105.62	105.69	104.37	106.08
elliptic	50.50	50.60	50.52	51.55
ex1010	60.96	60.99	60.06	61.90
ex5p	33.77	33.85	33.73	36.64
frisc	192.14	190.37	191.10	190.99
misex3	105.93	105.61	104.47	105.72
pdc	134.58	134.53	129.87	130.31
s298	17.54	17.56	17.54	17.85
s38417	207.81	207.35	206.88	212.26
s38584.1	174.17	174.31	169.27	173.95
seq	128.50	128.02	127.24	127.92
spla	62.59	62.27	62.26	62.32
tseng	50.36	51.05	49.53	53.77
Average	92.39	92.30	91.44	93.19

It is also important to evaluate the difference between results with training and testing signature lists. Figure 6.12 highlights that they are very similar for all circuits. The average variation is of 0.26%, with a maximum of 1.36% for *tseng*. Such a small difference indicates that the applicability of the HST mechanism is not restricted to the

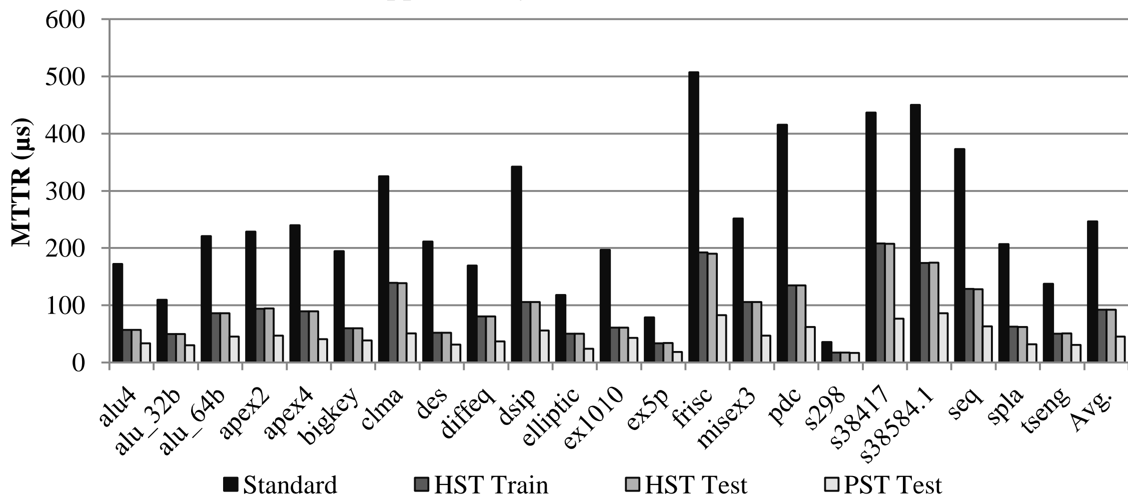


Figure 6.12: MTTR for the HST mechanism (with training and testing lists). PST and standard scrubbing are shown for comparison.

signatures used in its generation and that the experiments were able to adequately expose the error-to-signature relations for the circuits. Moreover, this difference is substantially smaller than that observed in for PST (5.08%, on average), showing that the HST mechanism is less susceptible to unexpected signatures or signature histograms that differ from those observed during table generation.

6.5.3.2 The impact of faults in the translation table

As discussed previously, it is important to assess the robustness of the proposed technique to faults in the translation table. For that purpose, faults are injected specifically on the translation table in step 6 (shown in Figure 6.4), which is stimulated with signatures obtained during the first injection campaign (step 2). Due to the large amount of signatures (shown in Table 6.1), which could not be stored within the FPGA memory, we limit the applied stimuli. For each injected fault, 1,000 different signatures are applied to the circuit, chosen as follows:

1. The all-zero input is applied to detect false alarm faults and faults that change the frame address output without triggering detection (as described in section 6.5.1.2);
2. For each possible compressed signature s_c, the most frequent signature that is mapped to s_c is chosen. This aims at stimulating the different circuit paths. As *maxSize* = 7, this represents at most 128 different signatures;
3. The remaining signature slots are filled with the most frequently occurring signatures that were not inserted during step 2, aiming at covering the most frequent signatures in the experiment.

Signatures chosen this way cover 90% of all occurrences observed in the first injection campaign. Faulty outputs are sent to the host PC, which categorizes them and calculates their effect on the MTTR, following the approach described in section 6.5.1.2. Table 6.4 shows, on the two rightmost columns, the MTTR associated with the chosen subset of signatures, assuming a fault-free (golden) table and when the effect of faults are included. Figure 6.13 shows the increase observed for each circuit, which had a 2.48% average. Overall, it can be seen that the technique is very robust to such faults. Even when their effects are considered, a 61.66% average reduction is maintained over standard scrubbing. First, because HST is able to significantly reduce the table size. Therefore, the amount of sensitive bits in the table is very small, compared to the payload circuit. Second, the redundant checker allows detecting those situations that

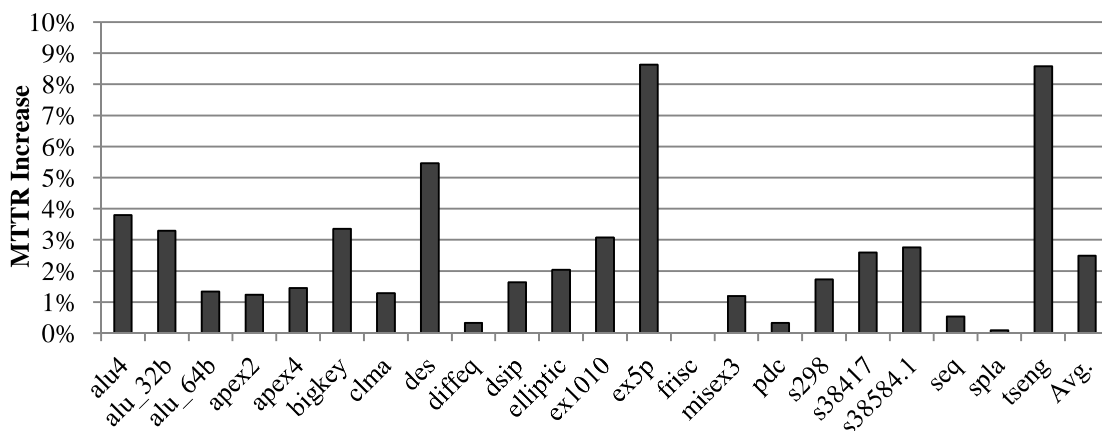


Figure 6.13: MTTR increase due to faults affecting the translation table

would have the highest impact, such as mistaking a false alarm for an actual payload circuit fault.

The values change significantly between each circuit as they are influenced by a number of factors. For example, the most significant increase is 8.6% for *ex5p*. For this case, the time required to repair the table when false alarms happen is particularly significant, for two reasons: the circuit has a low baseline MTTR (as can be seen in Figure 6.12) and the area occupied by the table is significant (shown in Figure 6.10). The short MTTR causes a small increase to be more significant, while the significant table area introduces more sensitive bits. This is also the case for *tseng*, which had a similar increase. For *s298*, on the other hand, even though it has the highest area overhead, it presents only moderate gains with SURFER, due to its reduced size. Moreover, its *best static address* presents comparable improvements, since the circuit area is very small. As a result, a reduced overhead is observed for those situations in which the table fault is detectable (e.g., invalid addresses). An interesting situation is also presented by *frisc*. As it is a large circuit with a longer MTTR and a relatively small table, the sensitive bits introduced by the table actually present a slightly reduced MTTR compared to the payload circuit. Thus, when all scenarios are considered, the overall MTTR remains virtually the same.

6.5.4 Evaluating the impact of the *maxSize* parameter

The *maxSize* parameter defines the maximum acceptable compressed signature length and is used to determine the heuristic compression effort. Figure 6.14 shows its impact on table area and MTTR for a representative subset of the benchmark circuits (for the sake of clarity). Results are shown for each iteration of the compression loop of the HST algorithm, which stand for different target *maxSize* values. Appendix C presents the results for all circuits. The rightmost points in the curves are associated with large signatures, which are iteratively reduced in the compression loop. Each point is associated with one such iteration. For *ex5p*, the rightmost point stands for the PST table. For the other circuits in Figure 6.14, XST was unable to synthesize PST tables.

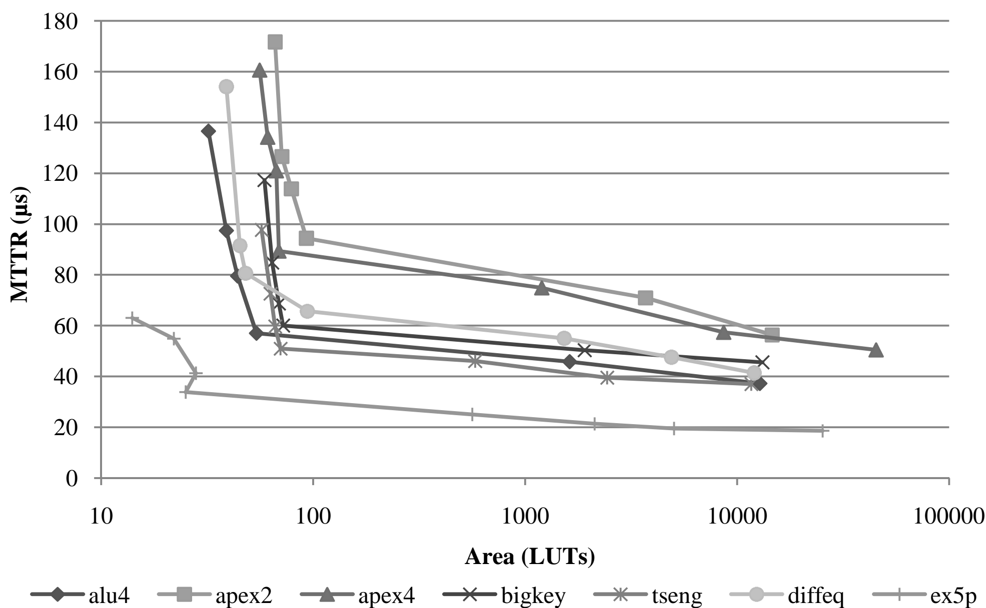


Figure 6.14: MTTR and table area for different *maxSize* values

The heuristic is able to provide multiple *Pareto points*, i.e., points that are not surpassed by any other in both metrics at once. The only non-*Pareto* point occurs for *ex5p*, when one of the compression iterations actually increases the area. This is a situation in which the compression circuit becomes larger but the translation table is not reduced accordingly, leading to a larger total area.

In general, there is a clear point up to which there are very significant area reductions. After this point, the MTTR continues to be increased, but the area is reduced less aggressively. This occurs when the compressed signature size approaches the amount of inputs in the device's LUTs, allowing efficient implementations of the address table seen in Figure 6.7. For Virtex 5 devices, LUTs have 6 inputs, but there are multiplexers to allow implementing any 7 or 8 input function with 2 or 4 LUTs, respectively (XILINX, INC., 2010). Thus, signatures around these sizes can be seen as optimal spots for the heuristic, considering a cost-benefit metric such as "MTTR reduction per area". The chosen value (*maxSize* = 7) for the experimental results in sections 6.5.2 and 6.5.3 is, therefore, in the middle of this space. The area reductions provided by further compression become less significant, as the compression circuitry starts to dominate the overall area. The leftmost point is associated with the circuit that compresses all bits into a single signature and responds with the best static address to all of them.

7 CONCLUSIONS

In this work, we have presented a study on the dependability threats faced by state-of-the-art FPGAs and on existing techniques aiming at mitigating them. Our attention was focused on a particular issue faced by such devices: with growing configuration memories, the time required to scrub away their transient errors becomes longer. And FPGAs tend become more and more susceptible to such errors, both due to the growing configuration size and the scaling of transistors. Thus, efficiently and quickly handling these errors becomes crucial to enable the use of FPGAs on critical systems, especially those on harsh environments, such as space applications. The use of fine-grained error detection techniques was put forth as a means to do so with manageable costs. We now summarize the main contributions of this work and the conclusions drawn, as well as possible future works. Publications achieved by the author both within the scope of this thesis and in cooperation with other researchers are listed in section 7.3.

7.1 Summary of contributions

7.1.1 Fault injection platform

A new fault injection platform was developed to evaluate the techniques proposed in this work. As main features, it requires one single FPGA to operate, reducing the costs of setting up the experimental setup. Moreover, it operates directly on the internal configuration access port (ICAP), without using softcore or hardwired processors. This reduces the injection latency and generalizes the platform's applicability, since it does not require special components, aside from LUTs, BRAMs, flip-flops and the ICAP. The modularity and extensibility of the injector allowed its adaptation to evaluate different attributes of circuits, such as fault coverage and detection latency. It was adapted to extract error signatures and to evaluate the susceptibility of the SURFER translation tables to faults, both being important aspects discussed in chapter 6. This platform is currently being used by other researchers to evaluate different mitigation mechanisms.

7.1.2 Platform for radiation experiments

The experiments conducted on ISIS, Rutherford Appleton Laboratories, required the development of a monitoring platform able to detect the occurrence of errors, to automatically reprogram the FPGA and to log all relevant events. This platform is described in section 5.4, along with the results that were obtained. As the fault injection system, this platform was developed in a modular and extensible manner. Both the on-chip monitoring circuit and the scripts on the host PC have been successfully adapted to be used with different circuits and fault tolerance techniques in cooperation researches.

7.1.3 Carry chain circuits for fine-grained comparison

The maintenance of low costs was among the main concerns of this work. And, since fine-grained redundancy typically demands additional area to implement the numerous required comparators, we have devised a method to use the abundant carry propagation chains found in FPGAs to implement these comparators. Thereby, the use of LUTs can be avoided. This translates to more LUTs being left available for other purposes (such as other functions to be integrated in the same FPGA) or even in the possibility to use a smaller (and lower cost) FPGA.

A tool to automatically apply the proposed technique was developed. Numerous features are supported, such as the instantiation of error aggregation circuitry, the use of redundant comparators and the duplication granularity. The technique was extensively evaluated under several axes and compared to a traditional coarse-grained DMR, showing similar area and significant reductions on detection latency at the cost of a slightly reduced fault coverage and an increased clock period.

7.1.4 Making use of fine-grained diagnosis with SURFER

7.1.4.1 Shifted scrubbing

The basic concept explored by SURFER is that one does not necessarily starts scrubbing an FPGA on the first configuration frame, i.e., it can be *shifted* in the addressing space. The idea to start scrubbing at a position closer to the actual error location was inspired by the rotational latency of hard disk drives: once the magnetic head reaches the desired track, it must wait for the disk rotation to bring the desired sector. If one could place the head just before this sector, then this time would be minimized. Similarly, the actual correction time of scrubbing depends on how far ahead the error is located, relative to the next frame to be accessed by the scrubbing unit. Therefore, one can choose a starting frame that minimizes the mean time to reach the actual error. This realization is, in fact, independent from fine-grained error detection mechanisms. Even without fine-grained diagnosis, one can estimate the areas with higher density of sensitive bits and start scrubbing immediately before that area.

7.1.4.2 Shifted scrubbing guided by error signatures

As discussed in section 6.1, several challenges are faced by systems aiming to explore very fine-grained diagnosis to accelerate repair. The dynamic factors that change masking and propagation through circuit logic cause multiple signatures to be generated by a same error. Furthermore, errors on different frames can cause a same signature, since a module’s functionality is not necessarily encompassed by a single frame and routing paths may cross long regions of the device. As a result, even when very fine-grained redundancy is used, it may not always be possible to narrow the error location down to a single frame. These signatures can, however, be used as meaningful hints for a shifted scrubbing system. The SURFER mechanism proposed herein was able to reduce the MTTR by 80% on average, when making use of a perfectly precise signature translation mechanism. This mechanism, however, turned out to have very high costs even for small circuits, creating the need for more efficient translation heuristics. It remains relevant, nonetheless, to show the maximum gains provided by SURFER, being useful as a goal for any such heuristic.

7.1.4.3 Heuristic for efficient signature translation

The heuristic signature translation (HST) proposed in this work is based on a compression circuit that joins signature bits with the OR function. It operates similarly

to a low cost hash table and heavily exploits the fact that not all collisions have the same impact on the final quality of the solution, since many signatures would be translated to neighboring frame addresses. By grouping those bits that are active (i.e., '1') for signatures that frequently appear in a same region, it attempts maximize collisions between such signatures and to minimize them between those that appear in far away locations. It allowed creating translation tables that provide an average 61.9% MTTR reduction at cost of 15.5% of the unhardened circuit area.

7.2 Future works

7.2.1 Choosing intermediate redundancy grains

In this work we have used a very fine granularity, since the outputs of all LUTs were compared to copies. This was made with reduced area costs by means of the carry propagation chains. But it did introduce delay penalties and generated very large error signatures, which increased the complexity of translating them to useful information. Therefore, identifying the most important observation nodes, both in terms of detection latency and of diagnosis, can be an interesting approach to reduce costs, similarly to what is done in partial redundancy works such as (PRATT, CAFFREY, GRAHAM, *et al.*, 2006) and (SHE and SAMUDRALA, 2009).

7.2.2 Further exploring the SURFER design space

The concepts introduced by SURFER open an enormous design space, in which many different research directions are possible. We highlight the following as promising approaches to further improve the benefits of SURFER.

7.2.2.1 Improved translation heuristics

The HST mechanism proposed here is one of many possible and had the main purpose of showing the feasibility of the SURFER approach. Different weight functions or grouping heuristics (not based on iterative maximum weighted matching) can be devised. For example, the current version of the heuristic does not always fully exploit the chosen maximum signature size *maxSize*. Since it approximately divides in half the size at each iteration, there may be situations in which the final compressed signatures are substantially smaller than *maxSize*. A final "relaxation" step can be introduced to ungroup bits and meet *maxSize* precisely, leading to less collisions and improved translation precision with very small area costs.

Other translation mechanisms can be found based on different paradigms as well. Meta-heuristics and neural networks, for example, may bring better results or at least interesting additional Pareto points. The time required to perform the translation, albeit relevant, can also be extended, if the quality of the chosen frame is improved substantially.

7.2.2.2 Multiple starting frames

The current SURFER mechanism points to *one* starting frame, based on the error probability distribution observed for that specific signature. It may be interesting in some situations to create multiple scrubbing areas, with different priorities, in order to skip "dead zones" in which the probability of finding an error, for that signature, is very small. This can be done with low costs if the compressed signatures are shared, at least partially, by the multiple translation tables.

7.2.2.3 Using other signals in the signature

The error signatures used in this work comprise all individual error detection signals, but additional informational can be included. As was discussed in section 6.1, error signatures may vary depending of dynamic factors, such as primary inputs (PIs). Therefore, adding information on the current state of the circuit can aid in the location of the error. For example, an indication of the current operation mode, of the software being executed in a softcore processor or registers which are particularly relevant for the component's operation can help improving the precision of the chosen frame address.

7.2.3 Diagnosing permanent faults and aging

This work focused primarily on locating and correcting soft errors. The improved diagnosis provided by fine-grained error detection, however, can also be used to identify areas of the FPGA which are subject to permanent faults or aging. If the incidence of a particular signature is significantly above its expected frequency, it may indicate that the associated FPGA area is facing aging or even a permanent fault. Alternative repair mechanisms, such as reallocating the module (or a part of it) to a spare area, can be adopted in this case.

7.2.4 Performing radiation testing over a complete SURFER platform

Implementing a complete SURFER platform for relevant applications, preferably with strict real-time restrictions, is an important step to validate the proposed flow. Once the complete system is implemented on a board with the required resources (i.e., an SRAM-based FPGA for payload application and a radiation-hardened device for scrubbing control) it can be subject to radiation testing, measuring the overall reliability of the entire platform.

7.2.5 Finding other uses for the signature translation heuristic

The HST algorithm proposed herein showed interesting results, being able to maintain repair acceleration with a very low area cost. It may be possible to apply this same heuristic (or variations of it) to other problems with the same requirements: large amount of inputs, small area, and approximate results. Comparing its performance with hardware-implemented neural networks, for example, is an interesting experiment to evaluate its efficiency.

7.3 Publications

The following publications were achieved by the author during this course.

7.3.1 Book chapters

BECK, A. C. S.; LISBÔA, C. A. L.; CARRO, L.; NAZAR, G. L. et al. Adaptability: The Key for Future Embedded Systems. In: BECK, A. C. S.; LISBÔA, C. A. L.; CARRO, L. **Adaptable Embedded Systems**. 1st. ed. New York: Springer, 2013. Cap. 1, p. 1-12.

NAZAR, G. L.; CARRO, L. Reconfigurable Memories. In: BECK, A. C. S.; LISBÔA, C. A. L.; CARRO, L. **Adaptable Embedded Systems**. 1st. ed. New York: Springer, 2013. Cap. 4, p. 95-117.

7.3.2 Journal

NAZAR, G. L.; RECH, P.; FROST, C.; CARRO, L. Radiation and Fault Injection Testing of a Fine-Grained Error Detection Technique for FPGAs. **IEEE Transactions on Nuclear Science**, Piscataway, (in press) 2013.

7.3.3 Conferences and workshops

AZAMBUJA, J. R.; NAZAR, G. L.; RECH, P.; CARRO, L. et al. Combining Hardware- and Software-Based Techniques to Detect and Diagnose Neutron Induced Single Event Effects in SRAM-Based FPGA. NUCLEAR AND SPACE RADIATION EFFECTS CONFERENCE (NSREC). San Francisco: [s.n.]. 2013.

ANJAM, F.; WONG, S.; CARRO, L.; NAZAR, G. L. et al. Simultaneous Reconfiguration of Issue-width and Instruction Cache for a VLIW Processor. INTERNATIONAL CONFERENCE ON EMBEDDED COMPUTER SYSTEMS: ARCHITECTURES, MODELING AND SIMULATION (SAMOS). **Proceedings...** Piscataway: IEEE. 2012. p. 183-192.

ITTURRIET, F. P.; NAZAR, G. L.; FERREIRA, R. R.; MOREIRA, A. F. et al. Adaptive parallelism exploitation under physical and real-time constraints for resilient systems. INTERNATIONAL WORKSHOP ON RECONFIGURABLE COMMUNICATION-CENTRIC SYSTEMS-ON-CHIP (ReCoSoC). **Proceedings...** Piscataway: IEEE. 2012. p. 1-8.

ITTURRIET, F.; FERREIRA, R.; GIRÃO, G.; NAZAR, G. et al. Resilient Adaptive Algebraic Architecture for Parallel Detection and Correction of Soft-Errors. EUROMICRO CONFERENCE ON DIGITAL SYSTEM DESIGN (DSD). **Proceedings...** Los Alamitos: IEEE CS. 2012. p. 136-139.

NAZAR, G. L.; CARRO, L. An Area Effective Parity-based Fault Detection Technique for FPGAs. INTERNATIONAL SYMPOSIUM ON DEFECT AND FAULT TOLERANCE IN VLSI AND NANOTECHNOLOGY SYSTEMS (DFT). **Proceedings...** Piscataway: IEEE. 2011. p. 27-33.

NAZAR, G. L.; CARRO, L. Energy Efficient Pseudo-Cache Architecture Through Fine-Grained Reconfigurability. INTERNATIONAL SYMPOSIUM ON CIRCUITS AND SYSTEMS (ISCAS). **Proceedings...** Piscataway: IEEE. 2011. p. 2317-2320.

NAZAR, G. L.; CARRO, L. Exploiting Modified Placement and Hardwired Resources to Provide High Reliability in FPGAs. ANNUAL INTERNATIONAL SYMPOSIUM ON FIELD-PROGRAMMABLE CUSTOM COMPUTING MACHINES (FCCM). **Proceedings...** Los Alamitos: IEEE CS. 2012. p. 149-152.

NAZAR, G. L.; CARRO, L. Fast error detection through efficient use of hardwired resources in FPGAs. EUROPEAN TEST SYMPOSIUM (ETS). **Proceedings...** Los Alamitos: IEEE CS. 2012.

NAZAR, G. L.; CARRO, L. Fast Single-FPGA Fault Injection Platform. INTERNATIONAL SYMPOSIUM ON DEFECT AND FAULT TOLERANCE IN VLSI AND NANOTECHNOLOGY SYSTEMS (DFT). **Proceedings...** Piscataway: IEEE. 2012. p. 152-157.

NAZAR, G. L.; RECH, P.; FROST, C.; CARRO, L. Experimental Evaluation of an Efficient Error Detection Technique for FPGAs. EUROPEAN CONFERENCE ON RADIATION AND ITS EFFECTS ON COMPONENTS AND SYSTEMS (RADECS). **Proceedings...** Piscataway: IEEE. 2012.

NAZAR, G. L.; SANTOS, L. P.; CARRO, L. Accelerated FPGA Repair through Shifted Scrubbing. INTERNATIONAL CONFERENCE ON FIELD PROGRAMMABLE LOGIC AND APPLICATIONS (FPL). **Proceedings...** Piscataway: IEEE (in press). 2013.

NAZAR, G. L.; SANTOS, L. P.; CARRO, L. Scrubbing Unit Repositioning for Fast Error Repair in FPGAs. INTERNATIONAL CONFERENCE ON COMPILERS ARCHITECTURE AND SYNTHESIS FOR EMBEDDED SYSTEMS (CASES). **Proceedings...** New York: ACM (in press). 2013.

SANTOS, P. C.; NAZAR, G. L.; ANJAM, F.; WONG, S. et al. A Fully Dynamic Reconfigurable NoC-based MPSoC: The Advantages of a Multi-Level Reconfiguration. WORKSHOP ON DESIGN TOOLS AND ARCHITECTURES FOR MULTI-CORE EMBEDDED COMPUTING PLATFORMS (DITAM). **Proceedings...** Berlin: [s.n.]. 2013.

SANTOS, P. C.; NAZAR, G. L.; ANJAM, F.; WONG, S. et al. A Fully Dynamic Reconfigurable NoC-based MPSoC: The Advantages of Total Reconfiguration. WORKSHOP ON RECONFIGURABLE COMPUTING (WRC). Berlin: [s.n.]. 2013.

SANTOS, P. C.; NAZAR, G. L.; CARRO, L.; ANJAM, F. et al. Adapting Communication for Adaptable Processors: A Multi-Axis Reconfiguration Approach. INTERNATIONAL CONFERENCE ON RECONFIGURABLE COMPUTING AND FPGAS (ReConFig). **Proceedings...** Red Hook: IEEE. 2012. p. 1-6.

TABORDA, T. B.; NAZAR, G. L.; CARRO, L. Evaluating the Weighted Fault Sensitivity of the Components of a VLIW Architecture. WORKSHOP ON DESIGN TOOLS AND ARCHITECTURES FOR MULTI-CORE EMBEDDED COMPUTING PLATFORMS (DITAM). Berlin: [s.n.]. 2013.

TABORDA, T. B.; NAZAR, G. L.; CARRO, L. Investigating Reliability-Critical Components of VLIW Processors. WORKSHOP ON DESIGN FOR RELIABILITY (DFR). Berlin: [s.n.]. 2013.

TAMBARA, L.; KASTENSMIDT, F. L.; AZAMBUJA, J. R.; CHIELLE, E. et al. Evaluating the Effectiveness of a Diversity TMR Scheme under Neutrons. CONFERENCE ON RADIATION EFFECTS ON COMPONENTS AND SYSTEMS (RADECS). **Proceedings...** Piscataway: IEEE (in press). 2013.

TONFAT, J.; AZAMBUJA, J. R.; NAZAR, G. L; RECH, P. et al. Analyzing the Influence of Voltage Scaling for Soft Errors in SRAM-based FPGAs. DATA WORKSHOP OF THE CONFERENCE ON RADIATION EFFECTS ON COMPONENTS AND SYSTEMS (RADECS). **Proceedings...** Piscataway: IEEE (in press). 2013.

REFERENCES

ABRAMOVICI, M.; BREUER, M.; FRIEDMAN, A. **Digital systems testing and testable design**. 1st. ed. New Jersey: Wiley-IEEE Press, 1990.

ABRAMOVICI, M.; STROUD, C.; HAMILTON, C.; WIJESURIYA, S. et al. Using roving STARs for on-line testing and diagnosis of FPGAs in fault-tolerant applications. INTERNATIONAL TEST CONFERENCE (ITC). **Proceedings...** Washington: IEEE Press. 1999. p. 973-982.

AGUIRRE, M. A.; TOMBS, J. N.; MUOZ, F.; BAENA, V. et al. Selective Protection Analysis Using a SEU Emulator: Testing Protocol and Case Study Over the Leon2 Processor. **IEEE Transactions on Nuclear Science**, Piscataway, v. 54, n. 4, p. 951-956, August 2007.

AIDEMARK, J.; VINTER, J.; FOLKESSON, P.; KARLSSON, J. GOOFI: Generic Object-Oriented Fault Injection Tool. INTERNATIONAL CONFERENCE ON DEPENDABLE SYSTEMS AND NETWORKS (DSN). **Proceedings...** Los Alamitos: IEEE CS Press. 2001. p. 83-88.

ALDERIGHI, M.; CASINI, F.; D'ANGELO, S.; MANCINI, M. et al. Evaluation of Single Event Upset Mitigation Schemes for SRAM based FPGAs using the FLIPPER Fault Injection Platform. IEEE INTERNATIONAL SYMPOSIUM ON DEFECT AND FAULT-TOLERANCE IN VLSI SYSTEMS (DFT). **Proceedings...** Los Alamitos: IEEE CS Press. 2007. p. 105-113.

ALTERA CORPORATION. Altera End Markets. **Altera**, 2012. Available at: <http://www.altera.com/end-markets/end-index.html>. Accessed in: 17 October 2012.

ALTERA CORPORATION. About Stratix Family High-End FPGAs and SoCs. **Altera**, San Jose, p. 580, 2013. Available at: <http://www.altera.com/devices/fpga/stratix-fpgas/about/stx-about.html>. Accessed in: 22 July 2013.

AVIZIENIS, A.; LAPRIE, J.; RANDELL, B.; LANDWEHR, C. Basic Concepts and Taxonomy of Dependable and Secure Computing. **IEEE Transactions on Dependable and Secure Computing**, Los Alamitos, v. 1, n. 1, p. 11-33, January-March 2004.

BANSAL, A.; RAO, R. M. Variations: Sources and Characterization. In: BHUNIA, S.; MUKHOPADHYAY, S. **Low-Power Variation-Tolerant Design in Nanometer Silicon**. 1st. ed. Dordrecht: Springer, 2011. p. 3-39.

BAUMANN, R. C. Radiation-Induced Soft Errors in Advanced Semiconductor Technologies. **IEEE Transactions on Device and Materials Reliability**, Piscataway, v. 5, n. 3, p. 305-316, September 2005.

BERNARDI, P.; SONZA REORDA, M.; STERPONE, L.; VIOLANTE, M. On the evaluation of SEU sensitiveness in SRAM-based FPGAs. IEEE INTERNATIONAL ON-LINE TESTING SYMPOSIUM (IOLTS). **Proceedings...** Los Alamitos: IEEE CS Press. 2004. p. 115-120.

BOLCHINI, C.; CASTRO, F.; MIELE, A. A Fault Analysis and Classifier Framework for Reliability-aware SRAM-based FPGA Systems. INTERNATIONAL SYMPOSIUM ON ON DEFECT AND FAULT TOLERANCE IN VLSI AND NANOTECHNOLOGY SYSTEMS. **Proceedings...** Los Alamitos: IEEE CS. 2009. p. 173-181.

BOLCHINI, C.; MIELE, A.; SANDIONIGI, C. A Novel Design Methodology for Implementing Reliability-Aware Systems on SRAM-Based FPGAs. **IEEE Transactions on Computers**, Los Alamitos, v. 60, n. 12, p. 1744-1758, Dec 2011.

CARMICHAEL, C.; CAFFREY, M.; SALAZAR, S. **Correcting Single-Event Upsets Through Virtex Partial Configuration**. Xilinx, Inc. San Jose, 12 p. 2000.

CHA, H.; RUDNICK, E. M.; PATEL, J. H.; IYER, R. K. et al. A Gate-Level Simulation Environment for Alpha-Particle-Induced Transient Faults. **IEEE Transactions on Computers**, Los Alamitos, v. 45, n. 11, p. 1248-1256, Nov 1996.

CHAPMAN, K. **SEU Strategies for Virtex-5 Devices**. Xilinx, Inc. San Jose, 16 p. 2010.

CIVERA, P.; MACCHIARULO, L.; REBAUDENGO, M.; SONZA REORDA, M. et al. An FPGA-Based Approach for Speeding-Up Fault Injection Campaigns on Safety-Critical Circuits. **Journal of Electronic Testing**, Dordrecht, v. 18, n. 3, p. 261-271, Jun 2002.

D'ANGELO, S.; METRA, C.; PASTORE, S.; POGUTZ, A. et al. Fault-Tolerant Voting Mechanism and Recovery Scheme for TMR FPGA-based Systems. IEEE INTERNATIONAL SYMPOSIUM ON DEFECT AND FAULT TOLERANCE IN VLSI SYSTEMS (DFT). **Proceedings...** Los Alamitos: IEEE CS Press. 1998. p. 233-240.

DE ANDRES, D.; RUIZ, J. C.; GIL, D.; GIL, P. Fault Emulation for Dependability Evaluation of VLSI Systems. **IEEE Transactions on Very Large Scale Integration (VLSI) Systems**, Piscataway, v. 16, n. 4, p. 422-431, Apr 2008.

DEZSő, B.; JÜTTNER, B.; KOVÁCS, P. LEMON – an Open Source C++ Graph Template Library. **Electronic Notes in Theoretical Computer Science**, v. 264, n. 5, p. 23-45, July 2011.

EDMONDS, J. Paths, Trees and Flowers. **Canadian Journal of Mathematics**, v. 17, p. 449-467, February 1965.

EMMERT, J. M.; STROUD, C. E.; ABRAMOVICI, M. Online Fault Tolerance for FPGA Logic Blocks. **IEEE Transactions on Very Large Scale Integration (VLSI) Systems**, Piscataway, v. 15, n. 2, p. 216-226, Feb 2007.

FULLER, E.; CAFFREY, M.; SALAZAR, A.; CARMICHAEL, C. et al. Radiation Testing Update, SEU Mitigation, and Availability Analysis of the Virtex FPGA for Space Reconfigurable Computing. MILITARY AND AEROSPACE APPLICATIONS OF PROGRAMMABLE DEVICES AND TECHNOLOGIES INTERNATIONAL CONFERENCE (MAPLD). **Proceedings...** Laurel: [s.n.]. 2000. p. 1-11.

GERICOTA, M. G.; LEMOS, L. F.; ALVES, G. R.; FERREIRA, J. M. On-Line Self-Healing of Circuits Implemented on Reconfigurable FPGAs. IEEE INTERNATIONAL ON-LINE TESTING SYMPOSIUM (IOLTS). **Proceedings...** Los Alamitos: IEEE CS. 2007. p. 217-222.

GOKHALE, M.; GRAHAM, P.; JOHNSON, E.; ROLLINS, N. et al. Dynamic reconfiguration for management of radiation-induced faults in FPGAs. INTERNATIONAL PARALLEL AND DISTRIBUTED PROCESSING SYMPOSIUM (IPDPS). **Proceedings...** Los Alamitos: IEEE. 2004. p. 1-6.

GOLSHAN, S.; KHAJEH, A.; HOMAYOUN, H.; BOZORGZADEH, E. et al. Reliability-aware placement in SRAM-based FPGA for voltage scaling realization in the presence of process variations. IEEE/ACM/IFIP INTERNATIONAL CONFERENCE ON HARDWARE/SOFTWARE CODESIGN AND SYSTEM SYNTHESIS (CODES+ISSS). **Proceedings...** New York: ACM. 2011. p. 257-266.

HATORI, F.; SAKURAI, T.; NOGAMI, K.; SAWADA, K. et al. Introducing redundancy in field programmable gate arrays. CUSTOM INTEGRATED CIRCUITS CONFERENCE (CICC). **Proceedings...** Los Alamitos: IEEE Press. 1993. p. 7.1.1-7.1.4.

HAYKIN, S. **Neural Networks - A Comprehensive Foundation**. 2nd. ed. Upper Saddle River: Prentice Hall, 1998.

HOWARD, N. J.; TYRRELL, A. M.; ALLINSON, N. M. The yield enhancement of field-programmable gate arrays. **IEEE Transactions on Very Large Scale Integration (VLSI) Systems**, Piscataway, v. 2, n. 1, p. 115-123, Mar 1994.

HSUEH, M. C.; TSAI, T. K.; IYER, R. K. Fault Injection Techniques and Tools. **Computer**, Los Alamitos, v. 30, n. 4, p. 75-82, Apr 1997.

IROM, F.; NGUYEN, D. N.; HARBOE-SøRENSEN, R.; VIRTANEN, A. Evaluation of Mechanisms in TID Degradation and SEE Susceptibility of Single- and Multi-Level High Density NAND Flash Memories. **IEEE Transactions on Nuclear Science**, Piscataway, v. 58, n. 5, p. 2477-2482, October 2011.

ITRS. **International Technology Roadmap for Semiconductos 2011 Edition - Design**. ITRS. [S.l.], 48 p. 2011.

JEDEC. **Measurement and Reporting of Alpha Particle and Terrestrial Cosmic Ray-Induced Soft Errors in Semiconductor Devices**. JEDEC. Arlington, 84 p. 2006.

JENN, E.; ARLAT, J.; RIMEN, M.; OHLSSON, J. et al. Fault injection into VHDL models: the MEFISTO tool. TWENTY-FOURTH INTERNATIONAL SYMPOSIUM ON FAULT-TOLERANT COMPUTING (FTCS). **Proceedings...** Los Alamitos: IEEE CS Press. 1994. p. 66-75.

KAMMLER, D.; GUAN, J.; ASCHEID, G.; LEUPERS, R. et al. A Fast and Flexible Platform for Fault Injection and Evaluation in Verilog-Based Simulations. THIRD IEEE INTERNATIONAL CONFERENCE ON SECURE SOFTWARE INTEGRATION AND RELIABILITY IMPROVEMENT (SSIRI). **Proceedings...** Los Alamitos: IEEE CS Press. 2009. p. 309-314.

KARLSSON, J.; LIDEN, P.; DAHLGREN, P.; JOHANSSON, R. et al. Using heavy-ion radiation to validate fault-handling mechanisms. **IEEE Micro**, Los Alamitos, v. 14, n. 1, p. 8-23, February 1994.

KASTENSMIDT, F. L.; FILHO, C. K.; CARRO, L. Improving Reliability of SRAM-Based FPGAs by Inserting Redundant Routing. **IEEE Transactions on Nuclear Science**, Piscataway, v. 53, n. 4, p. 2060-2068, Aug 2006.

KASTENSMIDT, F. L.; STERPONE, L.; CARRO, L.; SONZA REORDA, M. On the Optimal Design of Triple Modular Redundancy Logic for SRAM-based FPGAs. DESIGN AUTOMATION AND TEST IN EUROPE (DATE). **Proceedings...** Los Alamitos: IEEE CS Press. 2005. p. 1290-1295.

KUNDU, S.; REDDY, S. M. Embedded totally self-checking checkers: a practical design. **IEEE Design & Test of Computers**, Los Alamitos, v. 7, n. 4, p. 5-12, Aug 1990.

KUON, I.; TESSIER, R.; ROSE, J. FPGA Architecture: Survey and Challenges. **Foundation and Trends in Electronic Design Automation**, Delft, v. 2, n. 2, p. 135-253, April 2008.

KYRIAKOULAKOS, K.; PNEVMATIKATOS, D. A Novel SRAM-based FPGA Architecture for Efficient TMR Fault Tolerance Support. INTERNATIONAL CONFERENCE ON FIELD PROGRAMMABLE LOGIC AND APPLICATIONS (FPL). **Proceedings...** Los Alamitos: IEEE Press. 2009. p. 193-198.

LACH, J.; MANGIONE-SMITH, W. H.; POTKONJAK, M. Low Overhead Fault-Tolerant FPGA Systems. **IEEE Transactions on Very Large Scale Integration (VLSI) Systems**, Piscataway, v. 6, n. 2, p. 212-221, Jun 1998.

LESEA, A.; DRIMER, S.; FABULA, J. J.; CARMICHAEL, C. et al. The Rosetta Experiment: Atmospheric Soft Error Rate Testing in Differing Technology FPGAs. **IEEE Transactions on Device and Materials Reliability**, Piscataway, v. 5, n. 3, p. 317-328, September 2005.

LI, Y.; KIM, Y. M.; MINTARNO, E.; GARDNER, D. S. et al. Overcoming Early-Life Failure and Aging for Robust Systems. **IEEE Design and Test of Computers**, Piscataway, v. 26, n. 6, p. 28-39, November/December 2009.

LIMA, F.; CARMICHAEL, C.; FABULA, J.; PADOVANI, R. et al. A Fault Injection Analysis of Virtex FPGA TMR Design Methodology. 6TH EUROPEAN CONFERENCE ON RADIATION AND ITS EFFECTS ON COMPONENTS AND SYSTEMS (RADECS). **Proceedings...** Los Alamitos: IEEE Computer Society. 2001. p. 275-282.

LIMA, F.; CARRO, L.; REIS, R. Designing Fault Tolerant Systems into SRAM-based FPGAs. DESIGN AUTOMATION CONFERENCE (DAC). **Proceedings...** New York: ACM. 2003. p. 650-655.

LISBÔA, C. A. L. **Dealing with Radiation Induced Long Duration Transient Faults in Future Technologies**. Thesis (Doctoral Degree in Computing), Instituto de Informática - UFRGS. Porto Alegre, 113 p. 2009.

MEHTA, N.; DEHON, A. Variation and Aging Tolerance in FPGAs. In: BHUNIA, S.; MUKHOPADHYAY, S. **Low-Power Variation-Tolerant Design in Nanometer Silicon**. 1st. ed. Dordrecht: Springer, 2011. p. 365-380.

MICROSEMI CORPORATION. **Radiation-Tolerant ProASIC3 Low Power Spaceflight Flash FPGAs with Flash*Freeze Technology**. Microsemi Corporation. Aliso Viejo, 170 p. 2011.

MICROSEMI CORPORATION. **RTAX-S/SL and RTAX-DSP Radiation-Tolerant FPGAs**. Microsemi Corporation. Aliso Viejo, 278 p. 2012.

MINKOVICH, K. Kirill Minkovich's Home Page, 2011. Available at: <http://cadlab.cs.ucla.edu/~kirill/>. Accessed in: 10 October 2011.

MOJOLI, G. A.; SALVI, D.; SAMI, M. G.; SECHI, G. R. et al. KITE: A Behavioural Approach to Fault-Tolerance in FPGA-Based Systems. INTERNATIONAL SYMPOSIUM ON DEFECT AND FAULT TOLERANCE IN VLSI SYSTEMS (DFT). **Proceedings...** Los Alamitos: IEEE Press. 1996. p. 327-334.

NIKNAHAD, M.; SANDER, O.; BECKER, J. A study on fine granular fault tolerance methodologies for FPGAs. 2011 6TH INTERNATIONAL WORKSHOP ON RECONFIGURABLE COMMUNICATION-CENTRIC SYSTEMS-ON-CHIP (RECOSOC). **Proceedings...** Piscataway: IEEE Press. 2011. p. 1-5.

PRADHAN, D. K. **Fault-tolerant computer system design**. 1st. ed. Englewood Cliffs: Prentice Hall Publisher, 1996.

PRATT, B.; CAFFREY, M.; GRAHAM, P.; MORGAN, K. et al. Improving FPGA Design Robustness with Partial TMR. IEEE INTERNATIONAL RELIABILITY PHYSICS SYMPOSIUM. **Proceedings...** Piscataway: IEEE Press. 2006. p. 226-232.

PSARAKIS, M.; APOSTOLAKIS, A. Fault Tolerant FPGA Processor Based on Runtime Reconfigurable Modules. 2012 17th IEEE EUROPEAN TEST SYMPOSIUM (ETS). **Proceedings...** Los Alamitos: IEEE CS Press. 2012. p. 38-43.

RAMAKRISHNAN, K.; SURESH, S.; VIJAYKRISHNAN, N.; IRWIN, M. J. et al. Impact of NBTI on FPGAs. 20TH INTERNATIONAL CONFERENCE ON VLSI DESIGN. **Proceedings...** Los Alamitos: IEEE CS Press. 2007. p. 717-722.

SCHRIMPF, R. D. Radiation Effects in Microelectronics. In: VELAZCO, R.; FOUILLAT, P.; REIS, R. **Radiation Effects on Embedded Systems**. 1st. ed. Dordrecht: Springer, 2007. p. 11-29.

SEXTON, F. W. Destructive Single-Event Effects in Semiconductor Devices and ICs. **IEEE Transactions on Nuclear Science**, Piscataway, v. 50, n. 3, p. 603-621, June 2003.

SHE, X.; SAMUDRALA, P. K. Selective Triple Modular Redundancy for Single Event Upset (SEU) Mitigation. NASA/ESA CONFERENCE ON ADAPTIVE HARDWARE AND SYSTEMS (AHS). **Proceedings...** Los Alamitos: IEEE CS Press. 2009. p. 344-350.

SHNIDMAN, N. R.; MANGIONE-SMITH, W. H.; POTKONJAK, M. Fault Scanner for Reconfigurable Logic. Advanced Research in VLSI. **Proceedings...** Los Alamitos: IEEE CS Press. 1997. p. 238-255.

SONZA REORDA, M.; STERPONE, L.; ULLAH, A. An Error-Detection and Self-Repairing Method for Dynamically and Partially Reconfigurable Systems. 18th IEEE EUROPEAN TEST SYMPOSIUM (ETS). **Proceedings...** Los Alamitos: IEEE CS. 2013. p. 149-155.

STERPONE, L.; AGUIRRE, M.; TOMBS, J.; GUZMAN-MIRANDA, H. On the design of tunable fault tolerant circuits on SRAM-based FPGAs for safety critical applications. DESIGN AUTOMATION AND TEST IN EUROPE (DATE). **Proceedings...** New York: ACM. 2008. p. 336-341.

STERPONE, L.; VIOLANTE, M. A New Reliability-Oriented Place and Route Algorithm for SRAM-Based FPGAs. **IEEE Transactions on Computers**, Los Alamitos, v. 55, n. 6, p. 732-744, April 2006.

STERPONE, L.; VIOLANTE, M. A New Partial Reconfiguration-Based Fault-Injection System to Evaluate SEU Effects in SRAM-Based FPGAs. **IEEE Transactions on Nuclear Science**, Piscataway, v. 54, n. 2, p. 965-970, August 2007.

STRAKA, M.; KASTIL, J.; KOTASEK, Z. Generic partial dynamic reconfiguration controller for fault tolerant designs based on FPGA. NORCHIP. **Proceedings...** Piscataway: IEEE. 2010. p. 1-4.

VIOLANTE, M.; STERPONE, L.; MANUZZATO, A.; GERARDIN, S. et al. A New Hardware/Software Platform and a New 1/E Neutron Source for Soft Error Studies: Testing FPGAs at the ISIS Facility. **IEEE Transaction on Nuclear Science**, Piscataway, v. 54, n. 4, p. 1184-1189, August 2007.

WIRTHLIN, M.; JOHNSON, E.; ROLLINS, N.; CAFFREY, M. et al. The reliability of FPGA circuit designs in the presence of radiation induced configuration upsets. ANNUAL IEEE SYMPOSIUM ON FIELD-PROGRAMMABLE CUSTOM COMPUTING MACHINES (FCCM). **Proceedings...** Los Alamitos: IEEE CS Press. 2003. p. 133-142.

XILINX, INC. **Virtex-4 FPGA Configuration User Guide**. Xilinx, Inc. San Jose, 114 p. 2009a.

XILINX, INC. **Virtex-5 Family Overview**. Xilinx, Inc. San Jose, 13 p. 2009b.

XILINX, INC. **Virtex 5 FPGA User Guide**. Xilinx Inc. San Jose, 385 p. 2010.

XILINX, INC. **Virtex-5 FPGA Configuration User Guide**. Xilinx, Inc. San Jose, 166 p. 2011a.

XILINX, INC. **Virtex-II Pro and Virtex-II Pro X Platform FPGAs**. Xilinx, Inc. San Jose, 302 p. 2011b.

XILINX, INC. **Command Line Tools User Guide**. Xilinx, Inc. San Jose, 413 p. 2011c.

XILINX, INC. **Constraints Guide**. Xilinx, Inc. San Jose, 325 p. 2011d.

XILINX, INC. **7 Series FPGAs Overview**. Xilinx, Inc. San Jose, 15 p. 2012a.

XILINX, INC. Applications. **Xilinx**, 2012b. Available at: <http://www.xilinx.com/applications/index.htm>. Accessed in: 17 October 2012.

XILINX, INC. **Device Reliability Report**. Xilinx, Inc. San Jose, 118 p. 2012c.

XILINX, INC. ultrascale. **Xilinx**, 2013. Available at: <http://www.xilinx.com/products/technology/ultrascale/index.htm>. Accessed in: 22 July 2013.

ZEH, C. **Incremental Design Reuse with Partitions**. Xilinx, Inc. San Jose, 17 p. 2007.

APPENDIX A – TAXONOMY OF DEPENDABLE SYSTEMS

The taxonomy found in the field of dependable systems is vast and may vary from one work to another. Therefore, it is important to establish a common use of the definitions. In this appendix, we present the basic concepts related to dependable systems and discuss the nomenclature adopted in this work, based mainly on (AVIZIENIS, LAPRIE, RANDELL, *et al.*, 2004) and (PRADHAN, 1996), which are good sources for further reading on this topic.

A.1 Fault, Error and Failure

The most basic definitions are those of fault, error and failure, which follow cause-effect relations. A *fault* is defined as a cause of a possible error. It is, therefore, frequently associated with a physical phenomenon that may corrupt the system activity. Faults can also be human-made, such as mistakes during system design. Such faults, however, lie outside the scope of this work. An *error*, in turn, is defined as a divergence in the system state from the expected one, which may or may not lead to a service failure. Finally, a *service failure* (or simply *failure*) is defined as a deviation in the service provided by the system, as expected by a user or another system. This implies in the definition of *system boundaries*, which determine where the system being analyzed or developed begins and where it ends. If an error remains internal to the system boundaries and does not cause the system service to deviate, then no failure occurs. Similarly, if a fault never leads to an erroneous system state (it occurs in a component not in use, for example), then no error occurs.

An example can be used to better explain these concepts. Let us assume that an energetic particle hits a processor's arithmetic and logic unit (ALU) and temporarily changes the value of an internal wire, characterizing a fault. If that signal is in the shifter unit, for example, and this unit is not used, then no error occurs. Conversely, if a shift instruction is in execution when the fault occurs and it causes a register to receive an erroneous value, then an error takes place. Finally, if this error does not cause the service delivered by this program to deviate, then the system does not present failure. If the service differs, a failure occurs. Note that the placement of the system boundaries plays an important role at this point. If we consider the system as being strictly the processor, then the writing of an erroneous value to an external memory is considered a failure. If we place the off-chip memory within system boundaries, then a failure will only occur when there is a divergence in the service observed by external entities, e.g. another processor connected via network or a human user.

Faults, errors and failures can be classified into many different categories, according to several and frequently orthogonal properties. A comprehensive discussion on the

matter is presented in (AVIZIENIS, LAPRIE, RANDELL, *et al.*, 2004). Here, we focus on the aspects that are most relevant for the remainder of this work.

One of the most important aspects of faults regards its duration or persistence. *Transient faults* are those whose presence is bounded in time. Thus, it may be possible to completely remove them from the system. In other words, transient faults are those that do not damage the component in a permanent manner, and that disturb its operation for a limited time. The errors due to transient faults are called *soft errors*. Conversely, *permanent faults* are those with continuous or unbounded duration. They are usually due to irreversible damage to a component. The errors caused by permanent faults are called *hard errors*. Finally, some faults may lie in between permanent and transient. Although the term intermittent is used with another purpose in (AVIZIENIS, LAPRIE, RANDELL, *et al.*, 2004), we follow the taxonomy of (PRADHAN, 1996) on this matter. Thus, we refer to faults that appear and disappear repeatedly over time as *intermittent faults*.

Service failures are also classified into a variety of categories. For instance, they can be separated into *content* and *timing failures*, with the former referring to when the delivered value differs from the correct one, whereas the latter refers to when the time in which the information is delivered does not follow specification. Timing failures are, thus, very relevant for real-time systems. Failures can also be classified as *signaled*, when the system raises a warning signal informing that a failure occurred, and *unsignaled* when it does not.

A.2 Dependability and its features

With the definitions of fault, error and failure at hand, *dependability* can be defined. In (AVIZIENIS, LAPRIE, RANDELL, *et al.*, 2004) two definitions are presented. A dependable system can be considered as a system where trust can be justifiably placed. Alternatively, one can consider that a system is dependable when it can avoid failures that are more frequent or severe than is acceptable. The definition of *acceptable* is highly application-dependent. While a standard cell phone may acceptably fail once a year, an airplane engine cannot. Dependability envelops several other concepts:

- *Availability*: “readiness for correct service”. Also defined as the probability that the system will be functional at a given time t.
- *Reliability*: “continuity of correct service”. Also defined as the probability that the system will be functional during an interval $[t_0, t]$, provided it was functional at t_0.
- *Safety*: “absence of catastrophic consequences”. A failure may be *catastrophic* when it harms human lives, the environment or due to economical reasons.
- *Integrity*: “absence of improper system alterations”. This means that the system will not be modified in a way that harms its overall dependability.
- *Maintainability*: “ability to undergo modification and repairs”. In other words, how efficient is the system’s return to a functional state after a service failure.

The concepts listed above are those presented in (AVIZIENIS, LAPRIE, RANDELL, *et al.*, 2004). Other works include different sets of system features as part of dependability. In (PRADHAN, 1996), the concept of integrity is omitted, while two other features are included:

- *Performability*: the probability that the system will present a specific performance level at a given time instant.

- *Testability*: how simple it is to test the system, where testing is an attempt to identify specific problems within the system.

Just as the definition of an acceptable failure rate or severity is application-dependent, so is the relevance of each of the concepts encompassed by dependability. For example, data servers are typically concerned with high availability: the likelihood of a user finding the service unavailable must be as low as possible. Maintainability is also crucial for a high availability, as it is directly related to how long the system remains offline after a failure. Alternatively, for a system that is used during a mission time, such as those used in an aircraft, high reliability is the greatest concern. For these applications, it is crucial that the system does not fail during a given period of time, namely the mission, and failures during off-mission time are not nearly as severe. Performability, on the other hand, is highly relevant for real-time systems, where the system is required to produce an output or take an action within a restricted timeframe in order to avoid timing failures.

A.3 MTTF, MTBF, MTTR and FIT

Other relevant metrics are frequently used to evaluate dependable systems, or populations of such systems, especially over long periods of operation. The *mean time to failure* (MTTF) is the average time required for a system to present a service failure. Therefore, being an average metric, it requires a population of systems in order to be accurately estimated. Let N be the amount of identical systems in the population and tf_i the time that the i-th system took to present a service failure. The MTTF is defined in (A.1).

$$MTTF = \sum_{i=1}^{N} \frac{tf_i}{N} \tag{A.1}$$

Note that the MTTF is related to the first failure presented by the system. It is a very relevant metric when no repair is possible, i.e., once a failure occurs, the entire system must be replaced or removed from use. A slightly different metric, which is frequently used interchangeably with the MTTF, is the *mean time between failures* (MTBF). It is defined as the average time between two consecutive failures of a system. Assume that N instances of a system run for a time period T, with each system presenting, on average, n_{avg} failures. Equation (A.2) presents the definition of MTBF.

$$MTBF = \frac{T}{n_{avg}} \tag{A.2}$$

Let n_i denote the amount of failures presented by the i-th system during the period T. The average amount of failures n_{avg} used in (A.2) is defined in (A.3).

$$n_{avg} = \sum_{i=1}^{N} \frac{n_i}{N} \tag{A.3}$$

The MTBF is frequently reported with a slightly different metric called *failures in time* (FIT), which expresses the expected amount of failures per 10^9 device-hours of operation. It can be calculated using (A.4), provided the MTBF is expressed in hours.

$$FIT = \frac{1}{MTBF}.10^9 \tag{A.4}$$

Another related metric is the *mean time to repair* (MTTR). It represents the average time required to take the system from a failure state back a functional one. It is, hence, tightly related to the concept of maintainability and severely constrained for high availability systems. Let M denote the amount of failures presented by a population of systems and tr_i denote the time required to repair the i-th failure. The MTTR is defined in (A.5).

$$MTTR = \sum_{i=1}^{M} \frac{tr_i}{M} \quad \text{(A.5)}$$

A.4 Failure rate function, cross-section and the bathtub curve

The failure rate function $z(t)$, also called hazard function, represents the expected rate of failures of a population of systems at a given time t. In a population of N identical components, let $N_o(t)$ denote the amount of components operating correctly at time t and $N_f(t)$ the amount of components that have failed at time t. The derivative of $N_f(t)$, $dN_f(t)/dt$ represents the instantaneous rate of failing components. The failure rate function is defined in (A.6).

$$z(t) = \frac{1}{N_o(t)} \frac{dN_f(t)}{dt} \quad \text{(A.6)}$$

When evaluating $z(t)$ over electronic devices' lifetime, a general trend is found. Figure A.1 shows the bathtub curve, which depicts the typical behavior of $z(t)$. Shortly after manufacture, the failure rate is high due to "substandard" or "weak" components (PRADHAN, 1996). Manufacture faults which were not identified during testing may also contribute to this behavior. This period, called infant mortality phase, can be skipped by means of a burn-in process. Burn-in consists in operating the system, often under extreme conditions, in order to identify the weak components and to repair them or remove them from the population. Thus, when the components begin their actual service, they are already at the beginning of the useful life phase.

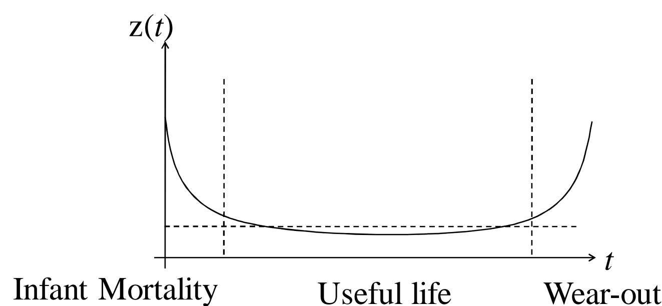


Figure A.1: The bathtub curve

The useful life is the period where the system presents its lowest failure rate and its most predictable behavior. Failures during this period are usually attributed to "random" effects, such as energetic particles or electromagnetic noise. Particularly regarding the effects of radiation, the sensitivity of a component is measured by its *cross-section*, which has the dimension of area (usually cm^2). It is defined as the area of the circuit that can lead to a given event (such as an error or a failure) if struck by a particle of a given energy. A good source for further reading on cross-section measurements is

(KARLSSON, LIDEN, DAHLGREN, *et al.*, 1994). In this work, we calculate the cross-section *CS* using (A.7). The *fluence* is the total amount of particles (e.g., neutrons, protons, etc.) per unit of area (cm^2 most frequently) that went through the device during a given period of time. *E* is the total amount of events (such as errors or failures, depending of the type of measurement) that was observed during the same period of time.

$$CS = \frac{E}{fluence} \tag{A.7}$$

After operating during its useful life, the wear-out phase begins. The components start to face *aging* effects that change their operating properties and that lead to an increase in the failure rate. It is, thus, very important to identify when a component is entering this phase, in order for it to be replaced.

A.5 Fault model and coverage

When developing dependable systems, or evaluating fault tolerance techniques to be used in such systems, one of the first questions that arise is: "what are the possible threats this system will face?" For example, a system operating in high altitudes, such as in space applications, needs to consider the impact of energetic particles on its operation, as it is not shielded by the atmosphere. Similarly, when a system is expected to be used for a long time, the effects of aging may have to be considered. In order to evaluate the resilience of a system against a given physical phenomenon, its impact on the system's operation needs to be accurately understood and modeled. For a *fault model* to be relevant, thus, it must closely represent the effects of one or more physical phenomena on the system's behavior. For example, transient and permanent faults will have different models and using one model to represent the other fault type is highly likely to lead to inaccurate results.

Furthermore, if one intends to use such fault model in fault injection campaigns, it is important to maintain the model's simplicity. As a statistically significant fault injection campaign for a complex system may take a long time, a complex fault model is likely to bring an undesirable computational burden to this task. Frequently used fault models include: single bit flip, in which one of the bits in the system's storage has its value changed; multiple bit flip, which is similar to single bit flip, but applied to more than one bit at the same time; single stuck-at, in which a net of the circuit receives permanently a given logic value, among others.

Once the relevant fault model(s) for the system at hand is defined, one can proceed to evaluate the fault coverage of the fault mitigation techniques available at the system. A fault is said to be covered depending on what the evaluated technique attempts to do. For example, all faults detected by a fault detection technique are considered covered, as are all faults masked by a fault masking technique. The fault coverage represents the probability that a fault of the evaluated model will be covered by the fault mitigation techniques. Parts of the system in which faults are not covered and may lead to a system failure are referred to as *single points of failure* (SPOFs). Let F_T denote the total amount of considered faults in the system, under the assumed fault model, and F_C denote the amount of covered faults. The fault coverage *C* is defined in (A.8).

$$C = \frac{F_C}{F_T} \tag{A.8}$$

APPENDIX B – USING NON-RANDOM INPUT VECTORS

In the experimental results reported in chapters 5 and 6, we have made use of pseudo-random input vectors in order to stimulate the operation of circuits. This was done to emulate a scenario in which little information was available to designers regarding input distribution. However, as will be shown in here, some of the discussed metrics can be affected by a highly correlated set of input vectors. Correlated inputs are observed naturally on many applications, such as stages of a pipelined processor which repeatedly execute the same small set of instructions.

The *alu_32b* circuit was used as a case study to evaluate some possible outcomes of changes in the properties of input vectors. For that purpose, we use vectors extracted from the execution of two pieces of software with the MIPS instruction set architecture, namely *CRC32* and *ins_sort*. As their names suggest, *CRC32* calculates the 32-bit cyclic redundancy check and *ins_sort* computes the insertion sort algorithm. These two algorithms stimulate the ALU very differently. *CRC32* makes use of many different instructions, since it requires numerous shifting and logic operations for the CRC calculation itself, and also additions and subtractions for loop control. On the other hand, *ins_sort* performs mostly additions and subtractions to compare elements of the vector and for loop control as well. Therefore, *CRC32* makes a much broader use of the ALU capabilities, selecting most of the operations available in the circuit.

Both algorithms were executed with two input instances, leading to different execution times (deemed *short* and *long* in the remainder of this appendix). For *ins_sort* a string with 8 characters and one with 43 were used, which led to execution times of approximately 2,500 and 28,300 cycles, respectively. For *CRC32*, a string with 43 characters and one with 430 were used, which led to execution times of approximately 3,900 and 29,800 cycles, respectively.

B.1 Impact on detection latency

As was discussed previously, an error can only be detected when its effects are stimulated and propagated to an observation point, i.e., a comparator. Therefore, input vectors heavily affect the observed detection latency. Figure B.1(a) shows the average required cycles for coarse- and fine-grained redundancies to detect errors. Figure B.1(b) shows the required time, assuming each circuit runs at its maximum frequency. Again, it is important to keep in mind that many errors affected the circuit but simply could not be detected at all, especially for coarse-grained redundancy, due to the limited amount of input vectors. Such errors could eventually be detected with a much longer latency, when appropriate vectors finally cause propagation to the primary outputs. However, since these latencies depend on what the ALU will compute afterwards, they cannot be estimated with a restricted set of vectors. Therefore, latencies for faults that only FG-

DMR could detect with the chosen vectors are not taken into consideration in the results, as was done in chapter 5.

Figure B.1 shows the results for the pseudo-random stimuli used in chapter 5 as well, labeled *rand*. It becomes clear that a highly correlated set of input vectors increases the average detection latency, since repeated (or similar) inputs do not aid in detection. This property therefore increases the relevance of having accelerated detection mechanisms for circuits to be used with highly correlated inputs. The *short* stimulus sets showed naturally reduced latencies compared to their *long* counterparts, as many errors remained silent during these limited testing scenarios but could be detected by the extended input sets.

FG-DMR was able to accelerate detection for all input sets, but with diverse ratios. *CRC32* showed more pronounced gains (31.4% and 35.9% for short and long executions, respectively) than *ins_sort* (19.2% and 18.8% for short and long executions, respectively). This is due to the poor stimulation provided by *ins_sort*, which makes no use of shift or logic functions. Since the ALU's adder/subtracter presents a relatively easy propagation compared to more complex modules, the latencies observed for *ins_sort* are shorter than those for *CRC32*, making the two approaches more similar and leaving reduced room for improvements from fine granularities. Proportional reductions were less pronounced than with pseudo-random inputs, as these were very efficient to stimulate the FG-DMR circuit and led to very significant gains (71.9%). On the other hand, the absolute time reduction obtained for the most critical case (*CRC32 long*) was the most expressive (4.5 µs).

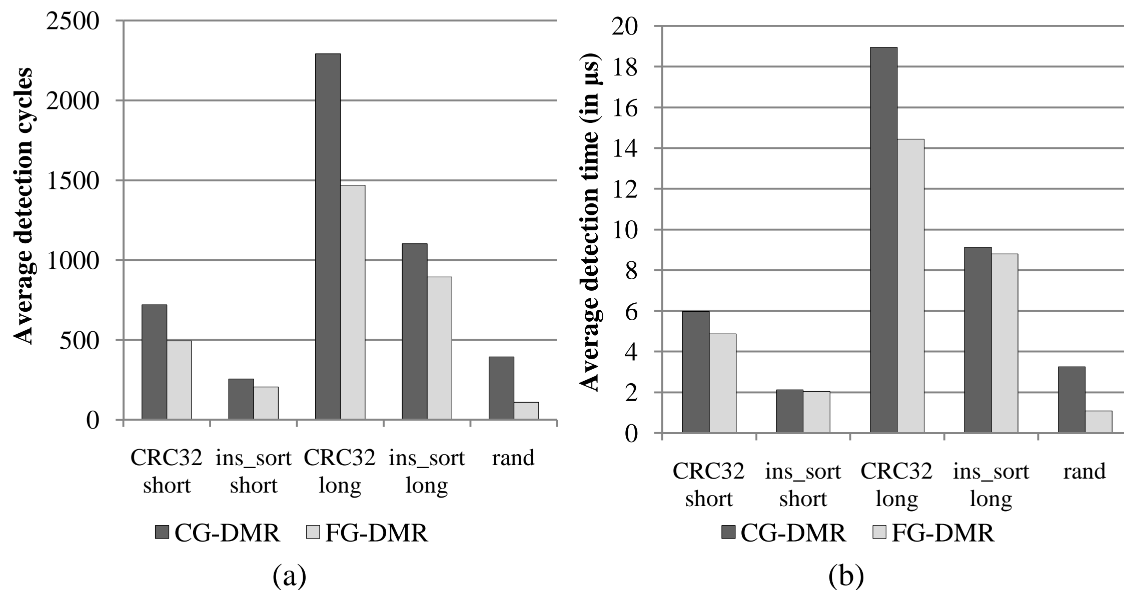


Figure B.1: Average detection cycles (a) and detection time (b) for different input sets

B.2 Impact on SURFER repair time

The experimental setup described in section 6.3 uses fault injection campaigns (and therefore also input vectors) to generate error signatures that are in turn used to build the SURFER translation tables. Thus, different input stimuli sets can generate different signature sets with varying distributions, leading to different translation tables with potentially different MTTR. Since the HST mechanism favors generating a translation table that is precise for signatures that are more frequent, it is important to validate that a short MTTR is maintained for input sets that differ from those used for the HST

algorithm. The approach used in chapter 6 to evaluate this aspect was to divide the generated signatures in two sets in order to *train* the table with a different set from that used to *test* it. Overall, very small variations were observed, as discussed in section 6.5.3, showing that the generated table was applicable not only to the signatures in the train set. In this section, we further evaluate this property by generating signatures with the input vectors used in section B.1. With these varied signature distributions we generate HST tables (with *maxSize* = 7) and then test them with the signatures generated with other input sets, as shown in Figure B.2. Thus, tables are tested not only with signature lists not available during training but also with lists that were obtained with different input vector distributions.

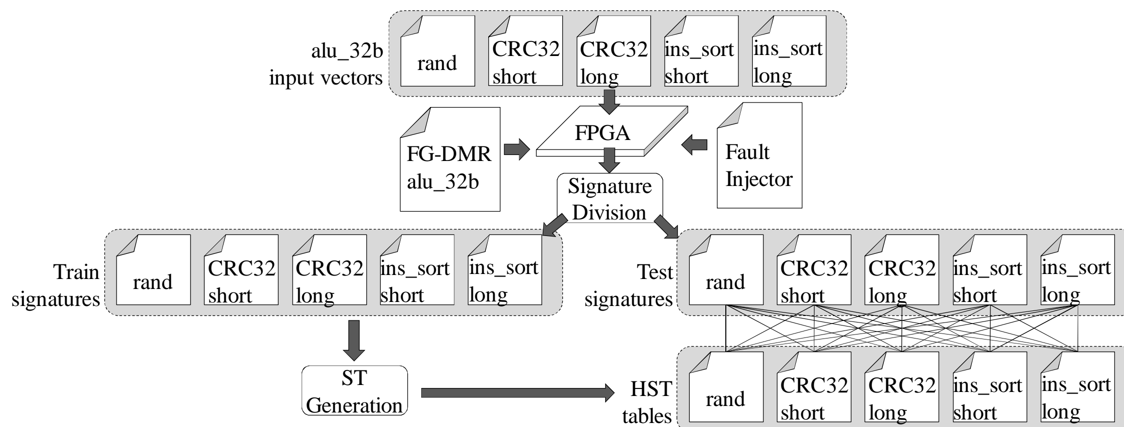


Figure B.2: Experimental flow for testing SURFER with varied input vectors

Figure B.3 shows the measured MTTR. Each entry in the *x* axis stands for one HST table generated with one training signature list while the data series (i.e., bar color) indicates the used test list. Overall, it can be seen that the input vectors had little effect, even when tables generated with one input set were used with signature distributions observed with others. Since the internal comparators assess the correctness of intermediate signals and not only of those that propagate to a primary output, they are able to detect errors even in modules not extensively used by the current input set (such as the shifter, which is not used in the *ins_sort* instances). Therefore, the generated translation tables captured approximately the same error-to-signature relations, showing little difference in terms of MTTR. The average over all results is 54.35 µs and the standard deviation is 2.13 µs, which results in a coefficient of variation of 0.04.

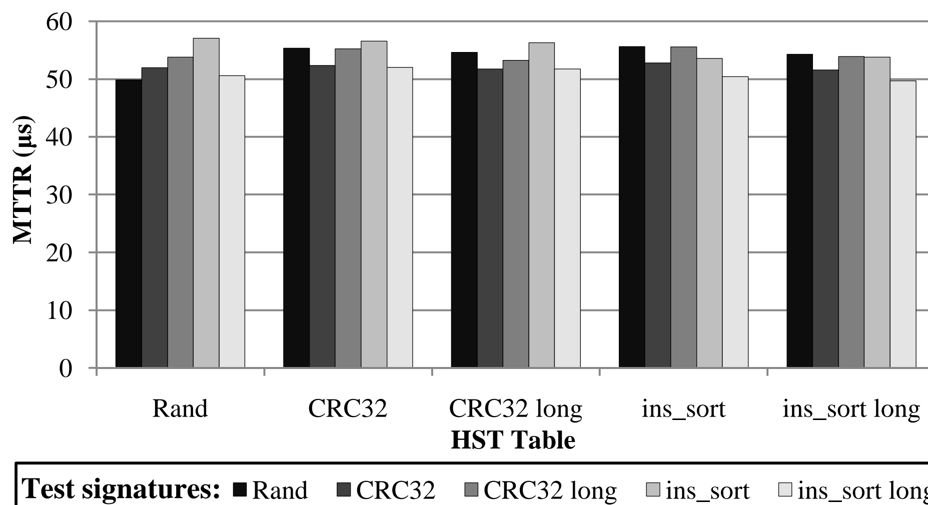


Figure B.3: MTTR with different translation tables and signature sets

APPENDIX C – MAXSIZE EVALUATION RESULTS

In this appendix, the impact of the *maxSize* parameter is measured for all benchmark circuits, regarding area and MTTR results. Results are presented for each iteration of the compression loop (shown in Figure 6.6). Each iteration stands for successively smaller *maxSize* values. Due to limitations in the synthesis tool, which is unable to handle tables with very large signatures, area results are reported only after a number of iterations of the compression loop for most circuits.

The 22 benchmark circuits are divided into three sets, and for each set MTTR (in Table C.1, Table C.3 and Table C.5) and area (in Table C.2, Table C.4 and Table C.6) results are reported. Entries in boldface are the ones associated with *maxSize*=7, i.e., the one used in the experimental results in sections 6.5.2 and 6.5.3.

Table C.1: MTTR (in µs) for the first set of circuits

Iteration	*alu4*	*alu_32b*	*alu_64b*	*apex2*	*apex4*	*bigkey*	*clma*
0	33.59	30.07	45.52	47.02	40.85	38.42	50.92
1	34.50	31.08	47.60	48.27	43.22	39.18	54.87
2	35.66	32.47	49.85	49.99	45.45	39.82	59.02
3	37.34	34.47	53.97	53.24	50.49	41.67	64.49
4	45.81	37.95	59.10	56.46	57.38	45.56	70.64
5	**57.06**	**49.84**	68.57	71.06	74.90	50.39	85.55
6	79.54	58.57	**86.28**	**94.39**	**89.37**	**60.06**	101.93
7	97.47	60.56	109.27	113.97	120.96	68.80	**138.74**
8	136.65	86.96	128.15	126.67	134.18	84.82	161.59
9			189.96	171.71	160.58	117.38	195.52
10							282.89
11							282.89

Table C.2: Area (in LUTs) for the first set of circuits

Iteration	*alu4*	*alu_32b*	*alu_64b*	*apex2*	*apex4*	*bigkey*	*clma*
0	N/A	N/A	N/A	N/A	N/A	N/A	N/A
1	N/A	N/A	N/A	N/A	N/A	N/A	N/A
2	N/A	N/A	N/A	N/A	N/A	N/A	N/A
3	12825	12259	N/A	N/A	45315	N/A	N/A
4	1622	401	16011	14609	8647	13186	22869
5	**54**	**49**	2147	3703	1198	1909	5169
6	44	38	**75**	**94**	**69**	**72**	702
7	39	36	73	79	67	69	**116**
8	32	28	67	71	61	64	114
9			60	66	56	59	110
10							103
11							103

Table C.3: MTTR (in µs) for the second set of circuits

Iteration	*des*	*diffeq*	*dsip*	*elliptic*	*ex1010*	*ex5p*	*frisc*
0	31.07	36.81	55.85	24.04	42.93	18.69	82.68
1	32.18	38.80	56.23	26.60	43.83	19.54	86.58
2	33.78	41.44	57.52	31.11	46.44	21.48	92.64
3	35.41	47.56	59.96	37.14	48.60	25.06	100.18
4	37.83	54.96	68.21	**50.60**	54.98	**33.85**	112.74
5	44.24	65.78	77.92	67.39	**60.99**	41.25	130.16
6	**51.87**	**80.74**	**105.69**	80.37	77.24	54.85	147.23
7	67.24	91.48	162.54	93.48	104.52	63.10	170.65
8	89.17	154.06	188.64		135.43		**190.37**
9	111.00		266.22				262.43
10							312.59

Table C.4: Area (in LUTs) for the second set of circuits

Iteration	*des*	*diffeq*	*dsip*	*elliptic*	*ex1010*	*ex5p*	*frisc*
0	N/A	N/A	N/A	1831	N/A	25290	N/A
1	N/A	N/A	N/A	1221	N/A	5049	N/A
2	N/A	12022	N/A	713	N/A	2126	N/A
3	N/A	4905	N/A	260	23113	563	N/A
4	14968	1530	10530	**25**	4239	**25**	N/A
5	1527	94	1979	24	**69**	28	21452
6	**74**	**48**	**76**	17	49	22	3948
7	71	45	73	13	40	14	193
8	67	39	71		36		**162**
9	60		62				153
10							150

Table C.5: MTTR (in μs) for the third set of circuits

Iteration	*misex3*	*pdc*	*s298*	*s38417*	*s38584.1*	*seq*	*spla*	*tseng*
0	47.16	62.00	16.88	76.34	85.95	63.36	31.90	30.85
1	50.10	63.45	**17.56**	81.79	87.77	65.81	32.86	32.39
2	52.63	64.80	20.08	90.21	89.77	68.78	34.56	34.06
3	57.46	68.46	21.53	97.45	92.71	73.36	39.72	36.97
4	67.88	75.46	24.05	112.71	96.41	82.86	50.54	39.51
5	83.59	94.05		131.58	110.31	99.58	**62.27**	46.14
6	**105.61**	110.31		156.53	123.36	**128.02**	85.25	**51.05**
7	136.53	**134.53**		189.47	152.68	144.76	119.68	59.70
8	170.15	156.38		**207.35**	**174.31**	181.55		72.70
9	197.37	247.90		268.75	238.68	266.38		97.68
10		296.78		388.22	371.55			
11				388.49	386.91			
12				388.55	386.91			
13					386.91			

Table C.6: Area (in LUTs) for the third set of circuits

Iteration	*misex3*	*pdc*	*s298*	*s38417*	*s38584.1*	*seq*	*spla*	*tseng*
0	N/A	N/A	119	N/A	N/A	N/A	N/A	N/A
1	N/A	N/A	**14**	N/A	N/A	N/A	19466	N/A
2	N/A	N/A	9	N/A	N/A	N/A	5112	N/A
3	57015	N/A	5	N/A	N/A	N/A	1701	11652
4	15799	N/A	2	N/A	N/A	15292	64	2433
5	458	17355		13960	26621	4030	**35**	582
6	**75**	299		2777	5944	**103**	24	**71**
7	69	**117**		313	758	85	19	66
8	64	114		**161**	**220**	76		63
9	59	106		159	194	72		57
10		102		157	193			
11				154	189			
12				149	185			
13					181			

APPENDIX D – RESUMO EM PORTUGUÊS

D.1 Introdução

Field Programmable Gate Arrays (FPGAs) são circuitos integrados reconfiguráveis que podem desempenhar diferentes funções uma vez que apropriadamente programados. Trazem um conjunto relevante de vantagens para sistemas críticos, o que inclui alta performance, flexibilidade e a programabilidade pós-implantação, permitindo a alteração de funcionalidades dos sistema, ou mesmo o acréscimo de novas capacidades. Com os avanços oferecidos pela Lei de Moore, se tornam cada vez mais eficientes, rápidos e com maior capacidade lógica.

Esse mesmo avanço nas técnicas de manufatura, entretanto, introduz um conjunto novo de desafios de confiabilidade a serem resolvidos. Em especial, destacamos a suscetibilidade da memória de configuração, responsável por armazenar a descrição do circuito desejado pelo usuário, a erros induzidos por partículas energéticas, como nêutrons, prótons e íons pesados. Essa tese versa sobre novas técnicas e mecanismos para prover confiabilidade a FPGAs, focando em falhas transitórias que afetam a memória de configuração, uma das principais ameaças a confiabilidade desses dispositivos (FULLER, CAFFREY, SALAZAR, *et al.*, 2000), (LESEA, DRIMER, FABULA, *et al.*, 2005).

D.2 Técnicas propostas

As técnicas aqui propostas têm por objetivo reduzir o tempo de reparo de FPGAs utilizados em aplicações críticas. Esse tempo é frequentemente bastante longo, pois a técnica mais amplamente usada, *scrubbing* (CARMICHAEL, CAFFREY and SALAZAR, 2000), acessa toda a memória de configuração de forma indiscriminada, o que se torna bastante lento à medida que essa se torna maior. Em especial, o foco é dado a técnicas de detecção de erro de grão fino baseadas em redundância modular dupla (FG-DMR). Essas técnicas intuitivamente reduzem a latência de erro, devido à maior quantidade de pontos de observação. Elas também proporcionam um diagnóstico mais detalhado, com o qual temos a possibilidade de identificar com maior precisão as possíveis localizações do erro.

D.2.1 Detecção de erros com comparadores de cadeia de propagação de vai-um

Uma das grandes desvantagens de técnicas de redundância de grão fino é a grande quantidade de comparadores que devem ser introduzidos. Propõe-se, visando a minimizar esses custos, uma forma de utilização alternativa dos circuitos propagadores de vai-um encontrados em profusão nos FPGAs modernos. Esse circuito, frequentemente subutilizado, pode ser empregado para comparar as saídas das LUTs. A técnica pode ser aplicada sempre que o propagador estiver disponível, juntamente com

entradas auxiliares do *slice* (bloco de elementos lógicos) necessárias para a aplicação da técnica.

D.2.2 Reparo rápido com diagnóstico de grão fino

Outro grande desafio encontrado ao se fazer uso de técnicas de diagnóstico de grão fino é como extrair, de forma eficiente, informações úteis para o reparo do sistema. Uma vez que temos uma grande quantidade de sinais de indicação de erro, precisamos de uma forma de mapeá-los para uma localização dentro da memória de configuração. Para esse propósito, é proposta a plataforma SURFER (*Scrubbing Unit Repositioning for Fast Error Repair*). Ela faz uso de um circuito que realiza a tradução das assinaturas de erros (ou seja, da concatenação de todos os sinais individuais de detecção de erro) em endereços de *frame*. É explorado ainda o conceito de que as operações de *scrubbing* não necessariamente iniciam na primeira posição da configuração. Assim, o endereço gerado pelo circuito de tradução indica o *frame* inicial das operações de reconfiguração, escolhido de forma a estatisticamente minimizar o tempo médio de reparo. Ainda foi proposta uma heurística para geração dos circuitos de tradução com custo reduzido, uma vez que, na sua forma mais precisa, os mesmos apresentavam custos muito altos em área ocupada.

D.3 Metodologia

As técnicas propostas foram desenvolvidas em ferramental de software integrado ao fluxo tradicional da Xilinx, fabricante dos FPGAs utilizados nessa tese. A partir de uma descrição do hardware sintetizado, já utilizando os componentes básicos do substrato do FPGA (LUTs, flip-flops, etc.), é criada uma versão do circuito que utiliza a variação de DMR proposta. A ferramenta identifica quais LUTs podem receber a comparação utilizando as cadeias de propagação de vai-um e, para as demais, instancia comparadores baseados em LUTs.

Sobre esses circuitos são conduzidas campanhas de injeção de falhas, visando a medir a cobertura atingida. Além disso, o ferramental provido pela Xilinx é utilizado para obtenção de dados referentes à área ocupada e ao atraso dos circuitos, medido aqui em termos do período mínimo de relógio dos circuitos (T_{Clk}). Todos os resultados obtidos são comparados com aqueles associados a uma técnica tradicional de DMR em grão grosso (CG-DMR). Campanhas de injeção de falhas também são utilizadas para extração das assinaturas de erro, que permitem a construção das tabelas de tradução propostas pela plataforma SURFER.

D.4 Resumo dos resultados

O uso de circuitos de propagação de vai-um conseguiu evitar o uso de LUTs para a criação de comparadores de grão fino. Assim, o custo em área foi de 118.8% sobre o circuito original, em média, enquanto que para CG-DMR foi de 111.6%. Ou seja, os circuitos com FG-DMR são apenas 3.57% maiores que os com CG-DMR. Foi observada uma redução de 99.62% para 99.58% na cobertura de falhas média com o uso de FG-DMR, além de um aumento médio de 48.7% em T_{Clk}. A quantidade de ciclos para detecção de erros, entretanto, foi reduzida em 66%, em média. Essa redução traduz-se em uma diminuição de 50% no tempo médio de detecção, se levarmos em conta os diferentes T_{Clk} observados para cada circuito.

Quando o circuito de tradução é utilizado, gerado através da heurística proposta, observou-se um custo total de 133.9% em área, o que representa um aumento de 10.5%

sobre CG-DMR. O tempo médio de reparo, entretanto, foi reduzido em 61.9%, em comparação com as abordagens tradicionais, ou seja, que iniciam o reparo sempre pela primeira posição da memória de configuração associada à partição com falha.

D.5 Conclusões

As técnicas propostas nesse trabalho permitiram o uso de redundância de grão fino de forma a acelerar o reparo de erros na memória de configuração de FPGAs com custos comparáveis aos de técnicas tradicionais de grão grosso. Portanto, os dois grandes objetivos desse trabalho foram atingidos.

Como trabalho futuro, prevê-se a criação de heurísticas de tradução de assinaturas aprimoradas para obtenção de pontos mais vantajosos no espaço de projeto. O uso de granularidades intermediárias e a extensão das técnicas propostas para que cubram falhas permanentes também são possíveis trabalhos futuros.