UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

New Fault Detection, Mitigation and Injection Strategies for Current and Forthcoming Challenges of HW Embedded Designs

Jaime Espinosa Garcia

Advisors:
David de Andrés Martínez, PhD
Juan Carlos Ruiz Garcia, PhD

Valencia, July 2016

"The problem with the world is that the intelligent people are full of doubts, while the stupid ones are full of confidence."

Charles Bukowski

Agraïments

Volguera començar agraïnt totes aquelles persones que han fet possible que esta tesi esdevinguera realitat. A totes aquelles persones i que m'han acompanyat durant tot este temps, amb el seu suport tècnic, lingüístic, físic, econòmic o psicològic.

Primerament, agrair els meus directors els doctors David de Andrés i Juan Carlos Ruiz per la seua dedicació, suport i ànim durant estos anys. Heu sigut de gran ajuda. També al professor Pedro Gil per confiar en mi i donar-me el recolzament econòmic i científic per portar a terme el projecte.

Així mateix, també volguera tenir unes paraules per als meus companys del grup de Sistemes Tolerants a Fallades (STF) i en especial als del laboratori: el ja doctor Jesus Friginal, Miquel Martinez, Héctor Marco, Javier Cancio i Antonio Bustos. Haveu sigut els millors, i junts hem passat molt bons moments. També agrair els companys de l'estada a Edinburgh Félix Casado i Ali Ebrahim, i molt especialment als del Barcelona Supercomputing Center Carles, Jaume, Mikel, Javi, Maria, David, Roberto, Milos, Mladen, Suzanna, Quixiao, Leonidas i la resta del grup CAOS per fer-me sentir com a casa.

Per descomptat que no només he rebut suport dels companys de recerca. Per això he d'esmentar la meua estimada Sara, per donar-me tot el suport i l'estima possibles i impossibles, i els meus pares i germans, perquè sempre han estat fent-me costat i m'han animat a seguir estudiant per comprendre millor el món que ens envolta.

Finalment a les persones de la Universitat en general, i del món científic en particular, a la vocació dels quals devem que el sistema continue funcionant.

Jaume.

Sumari

La rellevància que l'electrònica adquireix en la seguretat dels productes ha crescut inexorablement, puix cada volta més aquesta abasta una major influència en la funcionalitat dels mateixos. Però, per descomptat, aquest fet ve acompanyat d'un constant necessitat de majors prestacions per acomplir els requeriments funcionals, mentre es mantenen els costos i consums en uns nivells reduïts. Donat aquest escenari, la indústria està fent esforços per proveir una tecnologia que complisca amb totes les especificacions de potència, consum i preu, tot a costa d'un increment en la vulnerabilitat a diversos tipus de fallades conegudes, i a la introducció de nous tipus.

Per oferir una solució a les noves i creixents fallades als sistemes, els dissenyadors han recorregut a tècniques tradicionalment associades a sistemes crítics per a la seguretat, que en general oferixen resultats sub-òptims. De fet, les arquitectures empotrades modernes oferixen la possibilitat d'optimitzar les propietats de confiabilitat en habilitar la interacció dels nivells de hardware, firmware i software en el procés. Tot i això eixe punt no està resolt encara. Es necessiten avanços a tots els nivells en l'esmentada direcció per poder assolir els objectius d'una tolerància a fallades flexible, robusta, resilient i a baix cost. El treball ací presentat se centra en el nivell de hardware, amb la consideració de fons d'una potencial integració en una estratègia holística.

Els esforços d'esta tesi s'han centrat en els següents aspectes: (i) la introducció de models de fallada addicionals requerits per a la representació adequada d'efectes físics que apareixen en les tecnologies de fabricació actuals, (ii) la provisió de ferramentes i mètodes per a la injecció eficient del models proposats i dels clàssics, (iii) l'anàlisi del mètode òptim per estudiar la robustesa de sistemes mitjançant l'ús d'injecció de fallades extensiva, i la posterior correlació amb capes de més alt nivell en un esforç per retallar el temps i cost de desenvolupament, (iv) la provisió de nous mètodes de detecció per cobrir els reptes plantejats pels models de fallades proposats, (v) la proposta d'estratègies de mitigació enfocades cap al tractament dels esmentats escenaris d'amenaça i (vi) la introducció d'una metodologia au-

tomatitzada de desplegament de diversos mecanismes de tolerància a fallades de forma robusta i sistemàtica.

Els resultats de la present tesi constitueixen un conjunt de ferramentes i mètodes per ajudar el dissenyador de sistemes crítics en la seua tasca de desenvolupament de dissenys robustos, validats i a temps adaptats a la seua aplicació.

Sumario

La relevancia que la electrónica adquiere en la seguridad de los productos ha crecido inexorablemente, puesto que cada vez ésta copa una mayor influencia en la funcionalidad de los mismos. Pero, por supuesto, este hecho viene acompañado de una necesidad constante de mayores prestaciones para cumplir con los requerimientos funcionales, al tiempo que se mantienen los costes y el consumo en unos niveles reducidos. En este escenario, la industria está realizando esfuerzos para proveer una tecnología que cumpla con todas las especificaciones de potencia, consumo y precio, a costa de un incremento en la vulnerabilidad a múltiples tipos de fallos conocidos o la introducción de nuevos.

Para ofrecer una solución a los fallos nuevos y crecientes en los sistemas, los diseñadores han recurrido a técnicas tradicionalmente asociadas a sistemas críticos para la seguridad, que ofrecen en general resultados sub-óptimos. De hecho, las arquitecturas empotradas modernas ofrecen la posibilidad de optimizar las propiedades de confiabilidad al habilitar la interacción de los niveles de hardware, firmware y software en el proceso. No obstante, ese punto no está resulto todavía. Se necesitan avances en todos los niveles en la mencionada dirección para poder alcanzar los objetivos de una tolerancia a fallos flexible, robusta, resiliente y a bajo coste. El trabajo presentado aquí se centra en el nivel de hardware, con la consideración de fondo de una potencial integración en una estrategia holística.

Los esfuerzos de esta tesis se han centrado en los siguientes aspectos: (i) la introducción de modelos de fallo adicionales requeridos para la representación adecuada de efectos físicos surgentes en las tecnologías de manufactura actuales, (ii) la provisión de herramientas y métodos para la inyección eficiente de los modelos propuestos y de los clásicos, (iii) el análisis del método óptimo para estudiar la robustez de sistemas mediante el uso de inyección de fallos extensiva, y la posterior correlación con capas de más alto nivel en un esfuerzo por recortar el tiempo y coste de desarrollo, (iv) la provisión de nuevos métodos de detección para cubrir los retos planteados por los modelos de fallo propuestos, (v) la propuesta de estrategias de mitigación enfocadas hacia el tratamiento de dichos escenarios de amenaza

y (vi) la introducción de una metodología automatizada de despliegue de diversos mecanismos de tolerancia a fallos de forma robusta y sistemática.

Los resultados de la presente tesis constituyen un conjunto de herramientas y métodos para ayudar al diseñador de sistemas críticos en su tarea de desarrollo de diseños robustos, validados y en tiempo adaptados a su aplicación.

Abstract

Relevance of electronics towards safety of common devices has only been growing, as an ever growing stake of the functionality is assigned to them. But of course, this comes along the constant need for higher performances to fulfill such functionality requirements, while keeping power and budget low. In this scenario, industry is struggling to provide a technology which meets all the performance, power and price specifications, at the cost of an increased vulnerability to several types of known faults or the appearance of new ones.

To provide a solution for the new and growing faults in the systems, designers have been using traditional techniques from safety-critical applications, which offer in general suboptimal results. In fact, modern embedded architectures offer the possibility of optimizing the dependability properties by enabling the interaction of hardware, firmware and software levels in the process. However, that point is not yet successfully achieved. Advances in every level towards that direction are much needed if flexible, robust, resilient and cost effective fault tolerance is desired. The work presented here focuses on the hardware level, with the background consideration of a potential integration into a holistic approach.

The efforts in this thesis have focused several issues: (i) to introduce additional fault models as required for adequate representativity of physical effects blooming in modern manufacturing technologies, (ii) to provide tools and methods to efficiently inject both the proposed models and classical ones, (iii) to analyze the optimum method for assessing the robustness of the systems by using extensive fault injection and later correlation with higher level layers in an effort to cut development time and cost, (iv) to provide new detection methodologies to cope with challenges modeled by proposed fault models, (v) to propose mitigation strategies focused towards tackling such new threat scenarios and (vi) to devise an automated methodology for the deployment of many fault tolerance mechanisms in a systematic robust way.

The outcomes of the thesis constitute a suite of tools and methods to help the designer of critical systems in his task to develop robust, validated, and on-time designs tailored to his application.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A general introduction and background of the work is presented here alongside summarized points of motivation, goals and structure of the thesis.

1.1 Motivation

Embedded systems are becoming more and more ubiquitous in everyday life, just as computers did in recent years. For that reason, development of new foundation technology and applications for them has never stopped, what also brings a whole new set of challenges to tackle in several domains. In the present thesis the challenges covered are those related mainly to hardware aspects of dependability, with some additional hints on the security side these entail.

Dependability is defined as the justified confidence that can be placed in a service or function to be delivered correctly by a system to its user, be it another system or a human being. Security, while related to dependability, is defined as the capability to provide service to authorized requests while avoiding the provision of information to unauthorized ones and ensuring no undesired system or information alterations have occurred. Because not always correct service is delivered, the

causes and consequences of deviations from the expected function receive the name of factors of dependability: faults, errors and failures.

Faults are physical defects in hardware like a short-circuit or external events such as radiation (or mistakes in software functions) provoking errors, i.e. deviations from correct values in electrical nodes (which in turn affect data), and failures are the lack of delivery of expected outputs, or the delivery of incorrect ones as a consequence of errors. Figure 1.1 illustrates the process.

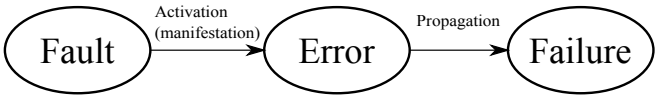


Figure 1.1: Factors of dependability: fault, error, failure chain

As long as errors are not perceived by users the system is considered as free of failures. Existence of failures is critical since they may lead to important harm to human life, monetary losses, leakage of information or damage to reputation. It is worth mentioning that the causal chain defined between faults, errors and failures is recursive, since a failure at a given system level can be seen as a fault from the perspective of another system level or user.

In the field of embedded systems, dependability is not a new topic. However, it is a quickly evolving topic where practitioners are continuously required to meet new and ever-growing demands arising from several sides, typically related to reduced failure rates, safe degradation modes or guaranteed lifetimes. To list some of the most powerful drivers of research in the field, it is possible to mention 3 main pillars: (i) new fabrication **technology** nodes, (ii) new safety and reliability **standards** for existing applications and (iii) completely new **applications** where never before had such systems been used. Security has likewise received a boost of attention in recent times, as educated individuals and organizations have demonstrated the potential harm malicious attacks can cause to insecure systems.

As fabrication technology progresses to keep the pace of rising demand for high performance, low power and a very low cost, dependability properties are likewise affected. Thus, high performance is achieved by adding more and more computing elements working in parallel, which increase the total cross section for charged particles arriving into the system. This means more radiation *events* will impact the system. Furthermore due to the reduction of single element cross section, an increase in the number of *multiple bit upsets* is also foreseeable, what poses an additional challenge to fault tolerance strategies. Thermal and power stress are also worsened as a consequence of more and more elements switching simultaneously. Low power computing has jumped from battery-powered devices to every kind of

system, as thermal and power considerations have been gaining importance exponentially. However, the achieved benefits come associated with important costs in several aspects. Because power budgets are so scarce, noise becomes a major concern, both the internally and externally originated. Moreover, an added source of unreliability comes from increased sensitivity to power supply variations. Additionally, to keep power low, feature dimensions are scaled accordingly, what greatly increases aging effects (NBTI, EM, HCI...) in deep sub-micron technologies. Finally low cost is a well known golden driver in industry. In that sense less and less fabs are willing to invest the quickly growing amounts of money required to develop new process nodes, what forces them to close in most cases due to insufficient workload. Those left in production face the challenge of coping with manufacturing defects, what reduces yield thus increasing price per unit. The use of fault tolerance techniques can increase yield to achieve sustainable productivity. With that panorama for the near future, insecurity can also turn an important issue. The reduced amount of manufacturers, located solely in far east countries, could increase the chances that any embedded systems designed in other countries may be tampered or illegally copied, beside the fact that any problem in the production line of a single fab could cause serious monetary losses and shortages. Although a number of fault tolerance techniques are already in use, they were designed with a set of specifications that is continuously changing. Therefore some gaps appear which are not correctly covered by existing solutions.

Safety and reliability standards have been evolving over the years alongside technology and applications. As a result, older standards are deprecated and new, more demanding ones appear as substitutes. Typical applications where such traditional standards apply are automotive, aerospace, railway, nuclear or process industries. In the new standards such as ISO26260 [81], DO-254 [40], IEC62278 [80], IEC61511 [79], etc. there is a trend to require for fault injection in the systems, in order to assess, at all stages of the development, the dependability of the systems. To that end, industrial sector demands optimized processes related to safety and robustness verification, in order to obtain approved certification at the lowest cost. The currently available set of injection tools needs to be completed with faster and more insightful choices to provide solid evidence towards certifications.

New exciting applications have found their way in the embedded systems realm. Many of those involve what is known as cyber-physical systems (CPS), where a computational system is tightly linked to a physical machine. These machines boast utterly useful capabilities for tackling all sorts of tasks, what makes them specially interesting under harsh environments which humans would not tolerate. Additionally, some are commanded to perform life, mission or business critical tasks. Under those premises application-specific integrated circuits (ASICs) but also special field-programmable gate arrays (FPGAs) have been in the market for some time, only to see commercial type SRAM based FPGAs get into that niche

in recent times due to their superior performance and value. For traditional and the newly introduced devices alike, detection and mitigation of potentially critical faults is a must if they are intended to operate correctly under critical environments. Moreover, powerful tools for simulating the mission conditions, such as injection of abnormalities, are another important requirement for the success of the whole program. Nevertheless, not only critical applications or very specific markets can benefit from proper detection, mitigation and/or assessment. General purpose recent applications like smart meters, cryptography blocks or even ubiquitous smartphones can also benefit from increased tolerance to spontaneous malfunctioning, premature aging or detection of an attempt of tampering, thus improving the quality perception of the brand and decreasing the negative impact accidental or malicious events may have on the economy or lives of human beings.

1.2 Objectives

The challenges presented in the previous motivation section cover a wide spectrum of areas. In the present dissertation, the aim is to contribute to such areas as a way to improve dependability of the future embedded systems, and by means of providing the community with new tools, methodologies and strategies to deal with present and upcoming challenges. The main objectives of this thesis can be established as the following:

- Study the current and upcoming accidental faults affecting embedded systems to better understand their origin, evolution and consequences and provide new representative models whenever required.

  With proper models it is possible to emulate physical events in laboratory conditions, testbeds or simulators, to quickly evaluate solutions before an actual device is built. The goal is to fix any dependability issues in the system as soon as possible to reduce cost of re-spins.

- Provide a fault injection and analysis tool capable of exploiting the current computing state of the art and focused on performance and flexibility, to accelerate the tasks.

  The tool will need to deal with HDL defined circuits at various levels, and support the utilization of newly defined models along classical ones. Flexibility in the analysis capabilities is a further objective for the tool.

- Develop a dependability assessment methodology which cuts development time for the industry and help optimize costs.

  By employing simulation-based fault injection in early development stages, it is desirable to propose and validate a methodology to obtain quick assessment

on the robustness characteristics as accurate as possible, thus incorporating dependability considerations at the beginning of the design process. Moreover, assessment helps to infer statistics of interest to design more robust circuits according to applicable fault models. This is indicated for ensuring smooth standards fulfillment.

- Propose new fault tolerance mechanisms better suited to the current technology and applications trends.

  As a means to fulfill this goal a study on available and currently used fault tolerance strategies will be performed where additional requirements are introduced. With the fault injection, detected weaknesses can be used to find a proposal for a better suited fault mitigation mechanism, helping to reach the required dependability levels.

1.3 Structure of the thesis

This section outlines the organization of the thesis document, with its different chapters. For each of them a brief explanation is given.

The present thesis is based in a collection of publications in the dependability area developed in the course of this PhD, and linked around a central challenge: the provision of dependable embedded devices for current and forthcoming demands in the industry. For the sake of explaining the reader what can be read hereafter, an introduction to each of the dealt areas comes first, situating the work topics in which advances have been made. After that, a wrap-up of the whole work is developed including conclusions and a summary of the production in the thesis.

Chapter 2: Faults modeling. This chapter focuses on the fault process, pathology, manifestation and modeling. It reviews the current state of the art in the topic and presents the contributions to it included in the present thesis, mainly based in some new faults to consider and how to model them.

Chapter 3: Fault injection. In this chapter a state of the art on different fault injectors precedes the justification for development of a new tool of the type, taking as the basis a preexisting tool in the group. It deals with the questions on what, when, how and where to inject faults, plus details in the different possible analysis to apply to the raw data. The prototype tool is called *F*ault injection and *A*nalysis for *L*ow *L*evel *E*valuation *S*uite.

Chapter 4: System dependability assessment. In this chapter new methodologies to perform assessment on the dependability levels of embedded devices are explored. A survey on current practice is included with the contributions of this thesis in the area of multi-level correlation for obtaining

(more) accurate early robustness verification results. The presented work studies the relationships between different levels of abstraction in terms of dependability assessment.

Chapter 5: Fault tolerance mechanisms. This chapter is devoted to fault detection, diagnosis, mitigation and recovery. It surveys alternatives available in those areas, and states the problems and benefits of them. Later it introduces the contributions presented in this work. These are mainly centered in the fault models presented in previous chapters. Likewise, the main technical implementation issues are discussed. Finally, testing of the mechanisms for proper detection of fault models is discussed.

Chapter 6: Discussion and conclusions. Contributions of this work are discussed in detail in this chapter, with a keen emphasis on the advances and limitations. The degree of accomplishment of the goals is discussed. Finally, some guidelines for future work are suggested to further enhance the presented solutions or tackle some of their current limitations.

Chapter 7: Contributions. The main scientific contributions derived from the work developed in this dissertation are listed in this section. The scientific framework in which this work has been involved is also described, including related research projects, international collaborations and awards.

Annexes: Papers

In the annexes each of the papers is reproduced, one per chapter. For a proper understanding they are organized in thematic areas, where contributed areas are mentioned.

Chapter A: Tolerating multiple faults with proximate manifestations in FPGA-based critical designs for harsh environments.. In this chapter an new model for faults is introduced where the increased rates observed in most recent devices is introduced in the problem. A new mechanism to deal with it in SRAM-based FPGAs is presented based in rewrite and relocation actions among others. Contributes to Fault Modeling and Tolerance.

Chapter B: The Challenge of Detection and Diagnosis of Fugacious Hardware Faults in VLSI Designs. In this chapter attention is focused towards those short duration faults which until now were ignored due to being time filtered, but which mean a good opportunity to improve dependability of systems. They are named *fugacious faults*. A tailored fault model is proposed, and the problems for effective detection and diagnosis are identified. Contributes to Fault Modeling and Tolerance.

Chapter C: Increasing the Dependability of VLSI Systems Through Early Detection of Fugacious Faults. In this chapter a complete fault

detection and diagnosis mechanism is presented, to tackle fault models introduced in the previous chapter. Alongside an implementation workflow and helping tool is presented. Finally application to a small design provides some measurements. Contributes to Fault Modeling and Tolerance.

Chapter D: An Aspect-oriented Approach to Hardware Fault Tolerance for Embedded Systems. The chapter explains how aspect oriented programming can be employed to automatically deploy fault tolerance mechanisms in IP cores. A few different examples are shown to demonstrate the feasibility. Contributes to Fault Tolerance.

Chapter E: Robust communications using automatic deployment of a CRC-generation technique in IP-blocks. The chapter extends the fault tolerance mechanisms presented as samples of possible automated applications to a communications targeted mechanism like custom CRC code generation. Contributes to Fault Tolerance.

Chapter F: Towards Certification-aware Fault Injection Methodologies Using Virtual Prototypes. This chapter presents a newly developed fault injector named FALLES, and provides a sketch of what could be done with virtual prototypes focused towards helping in safety certification. To do so extensive injections using FALLES can help improve the accuracy of the process. Contributes to Fault Injection and Dependability Assessment.

Chapter G: Analysis and RTL Correlation of Instruction Set Simulators for Automotive Microcontroller Robustness Verification. In this chapter a correlation is found for using instruction set simulators as virtual prototypes and the dependability metrics involved in safety of automotive microcontrollers. To do so permanent fault models have been injected using FALLES to study the outputs and compare them with the information available for the instruction set simulator. Contributes to Fault Injection and Dependability Assessment.

Chapter H: Characterizing Fault Propagation in Safety-Critical Processor Designs. In this paper a thorough analysis of propagation of faults through the pipeline of a Leon3 processors is presented. The purpose is to conclude whether injection at the architectural registers can capture the information conveyed by injection results obtained at RTL. Contributes to Fault Injection and Dependability Assessment.

Chapter 2

Faults Modeling

This chapter is devoted to the study of faults in their whole dimension: their pathology, propagation, manifestation and central to this work, modeling. A set of references and previous work is presented together with some views on the new trends and directions we can find in novel technologies coming out in the market.

2.1 Pathology

Advances in the semiconductor industry have been following over the last 50 years the well-known Moore's law [115]. It states that, for the same die size, transistor count will double every two years. The rule kept reasonably cheap to follow for many years but, with the advent of deep sub-micron technology nodes, the game changed completely. Alongside the benefits of doubled power and less consumption, came an important aggravation of the already-known problems of reliability, manufacturing defects, etc... which many practitioners have described [35, 3, 162, 154, 23]. The involved physical effects are Time Dependent Dielectric Breakdown (TDDB), ElectroMigration (EM), Negative Bias Thermal Instability (NBTI) or Hot Carrier Injection (HCI) to name a few. TDDB is a consequence of long term

operation beyond specified voltages; EM is due to high currents carrying material away from its original position; NBTI happens as a side consequence of dopants introduced to reduce leakage current and causes increase in threshold voltage, and HCI provokes charge carriers to get trapped in the gate dielectric hindering correct operation of the MOSFET. The most dangerous in terms of long term reliability can be the NBTI, but for high current devices EM can also pose severe design constraints. TDDB comes next in importance and HCI is the least severe of the mentioned effects.

While those are mainly intrinsic faults coming from inside the circuit or device, they have had a replica in the extrinsic faults, originated in the external side. Sadly enough, newer highly integrated technologies have turned more and more sensitive to the so-called *soft errors* [101, 17], which are caused by charged particles (α particles), neutrons reaching the silicon die and altering the logical value of an electrical node. The energy exchange process is similar to a bullet of a certain size hitting ever smaller targets. The critical charge required to 'damage' (upset) a single node is smaller and smaller, thus smaller bullets will have effect on that smaller target causing more upsets to appear. Additionally, sensitivity to power supply noise [62] or Electro Magnetic Interference (EMI) [124], because of reduced power budgets, has raised. To understand why, a quick look to energy consumption growth and heat density inside die explains the need to operate with low voltage power rails. Therefore logic levels are so close that fluctuations in the power supply and small interferences can cause a value to change. Finally, crosstalk [31] sensitivity has also raised dramatically, because an increase in switching speeds also involves an increase in generated noise to the vicinity.

As a summary of different physical processes involved in faults, 2 different figures in [66] were reproduced, one for permanent faults (Figure 2.1) and one for transient faults (Figure 2.2). In the figures, counterparts of the physical and electronic processes as logic/RTL level fault models are also shown for reference. The parameter used for separation -the different manifestation type- is developed in the next section.

2.2 Manifestation

All the previous section phenomena happen in different time spans. Some of them only take place during temporary periods and later disappear –transient faults– while others can only appear but never stop their unwelcome effect –permanent faults [13]. Moreover, there is a special type of faults which appear and disappear at random instants of time, i. e. they manifest themselves intermittently –the intermittent faults [34, 37]. These intermittent manifestations in the same location can be due to operational circumstances, which may eventually disappear, or be caused by wearout processes or manufacture defects. In the latter case they are

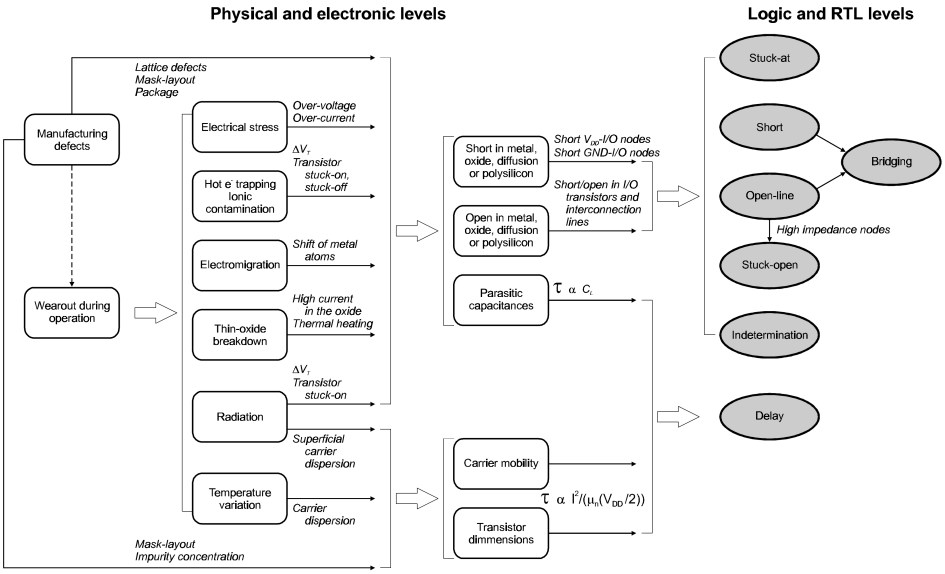


Figure 2.1: Pathology of permanent faults [66]

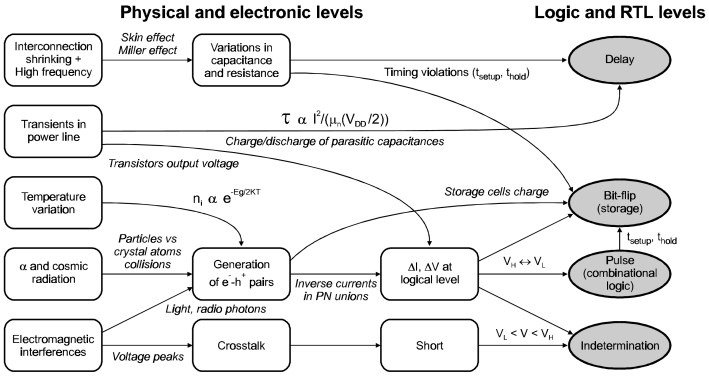


Figure 2.2: Pathology of transient faults [66]

preclude of permanent manifestations which follow a growing trend in time and intensity [159]. A table showing several of the physical processes which can lead to intermittent faults is shown in Figure 2.3.

There exist several different manifestations the community has identified which, up to day, are not well covered with current models, if covered at all. Such manifestations can happen in single or multiple fashion. In the latter case, for instance, the combination of an intermittent manifestation which has not been cleared from

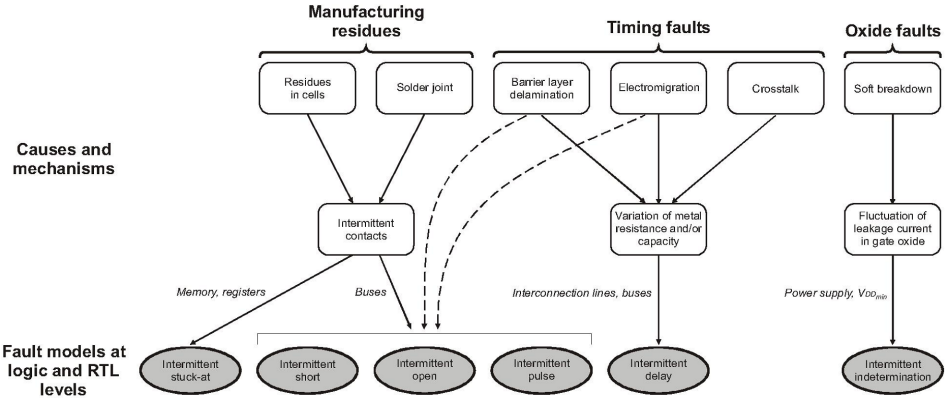


Figure 2.3: Pathologies of faults manifested as intermittent [69]

the system together with (e.g.) an additional transient or permanent, pose a new feasible threat to discuss. In fact, the same pathology can change its manifestation according to differing base technologies. Truly some of the fault processes which showed permanently under old ecosystems happen to manifest intermittently under modern scenarios, only to turn to permanent behavior much later. An example can be found in the ElectroMigration (EM) effect, which in time ends up breaking the physical continuity of a conductor but, in modern high frequency circuits, and due to skin effect, can cause increased delays which would randomly show an effect in the functionality. Only after an important lapse of time intermittent manifestation would lead to permanent disconnection. Likewise, processes which were responsible for transient faults, like crosstalk effects, have suffered a general increase in severity. Closer, more packed metal interconnects become the main reason after it. Therefore certain nodes can show now an intermittent manifestation, with a much higher potential of negatively impacting the system. The more complex manifestations faults have means new complex responses must be placed in order to adequately respond. Thus, mitigation techniques have to be revisited to continuously adapt to those complex fault footprints.

Returning to the consequences of size miniaturization from the dependability point of view, other notable facts must be underlined. Continuous tests show that, as channel length sizes have been reduced, so have been the fault manifestations when it comes to timespan. Measurements in laboratory conditions demonstrated that, for the same energy of the offending abnormality (charged particle, EMI interference, etc...), *shorter transient faults* were generated [60, 59, 43] and also *less energy* was required to cause an upset of the logical values of the circuit [41]. As narrowly shaped voltage variations have been traditionally filtered out, they have received little attention. However, with the most advanced sub-micron technologies

not only they become more frequent, but they can also reach propagation to further logic stages.

2.3 Propagation

After an undesired alteration of the voltage has affected an electrical node, it may or may not propagate through the circuit. Take for instance a fault causing a pulse in a combinational node. If propagation is not hindered in any of the ways explained next, the wrong value with traverse the logic path, reach into a sequential element and possibly have a negative impact in the provided service and therefore in the dependability level. However important filtering effects apply, avoiding every single manifested error to propagate and disrupt correct operation.

If the reached voltage is lower than the voltage threshold of the technology V_{th}, or the duration of the pulse is really small, it will not be able to switch the logical value of the next gate, since not enough energy will have been accumulated/discharged in the capacity associated to the affected node. Thus, an effect of *electrical filtering* will happen (Figure 2.4). In the literature there are plenty of studies related to the energy required to allow propagation but one of the most comprehensive can be found in [59].

When the upset has reached enough energy to propagate ahead in the base technology, other effects can stop it from doing so. One of the most relevant is *logic filtering*. The mechanism is simple: take for instance an AND gate, where one of its inputs is logical '0'. If any of the other inputs suffers an unexpected alteration of its value, there will be no impact at the output of the gate, and thus no propagation (Figure 2.5).

Once the upset has propagated all the way through the combinational logic down to the sequential elements, there is still a very important filtering mechanism to defeat: the *temporal filtering*. Any voltage value can be present at the input of a sequential element, but the only value to be captured will be that which is applied during the period in-between the set-up and hold associated to the capture clock edge (Figure 2.6). In [25] an experimentation with laser beams shows that, for a transient fault in combinational logic, there is great dependence on the frequency of the circuit w.r.t. the errors which will be propagated to the sequential elements. Conversely, upsets in the sequential elements will be totally independent from clock frequency.

When an upset has reached a sequential element, it can further propagate to the system or module outputs, (causing a failure) or not. But just as in the process from the manifestation of a failure until an upset hits the sequential element, filtering will apply. Hence, a complex function of system current state, inputs, instant of execution and workload will define the reach a fault will have in its

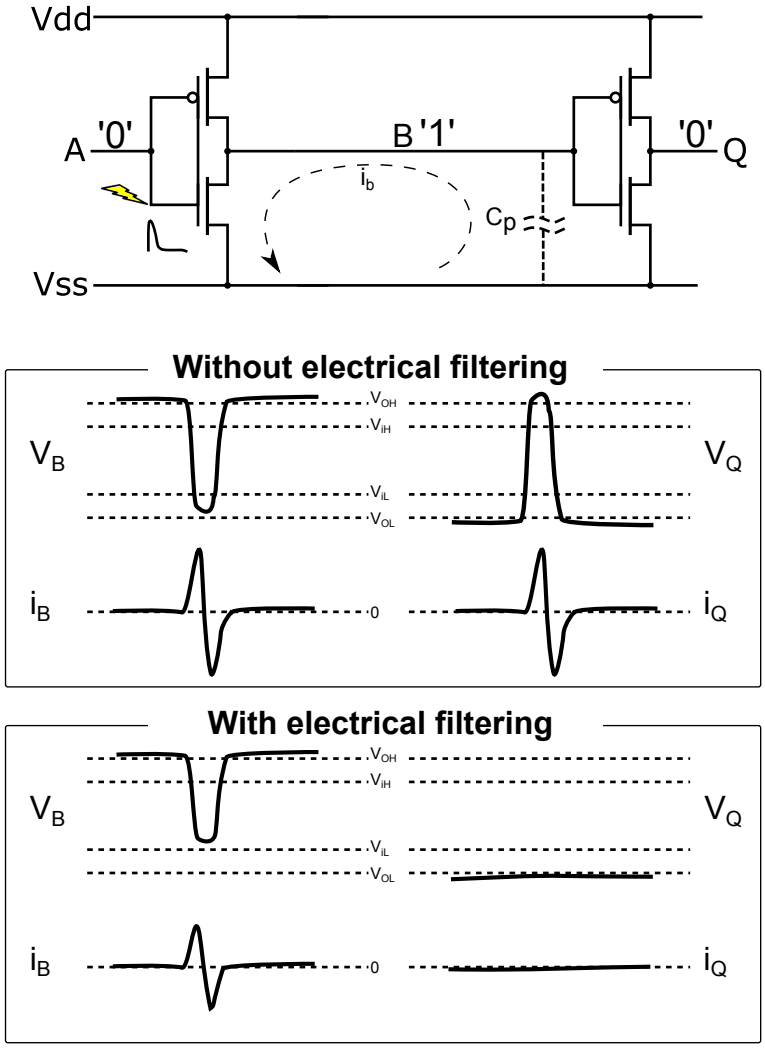


Figure 2.4: Electrical filtering effect

utmost consequences. Fault tolerance practitioners need tools to investigate those effects well in advance of the availability of the silicon, for economical and technical reasons, and the inputs to those tools will be models of the evaluated faults.

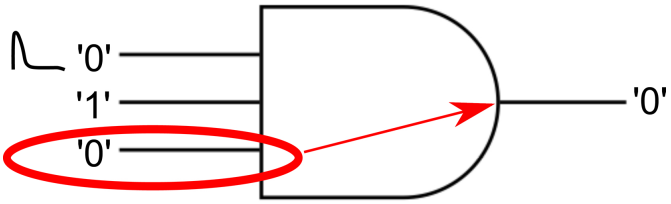


Figure 2.5: Logical filtering effect

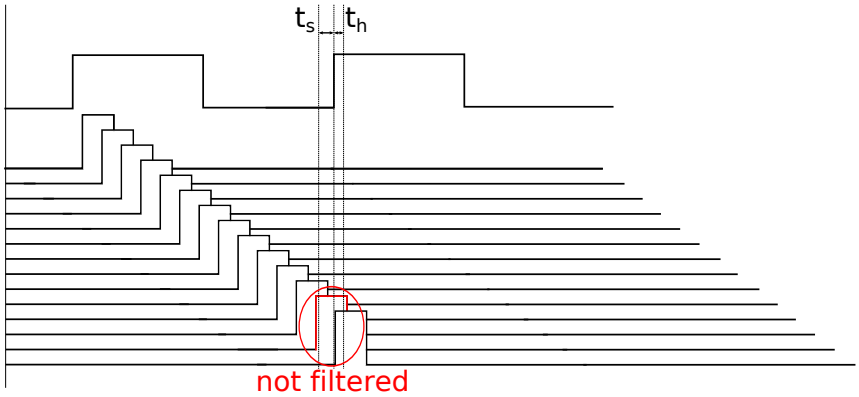


Figure 2.6: Temporal filtering effect

2.4 Modeling

In order to know how faults behave, i.e. their impact in circuits and systems, accurate models are required. With the purpose of providing such models, several works have been studying fault models for dependability analysis [66, 136]. In them, it is made clear how important models are to be able to study faults and their effects. In that sense a simplification, or an adaptation of what is present in the physical layer has to be done to suit the layer at which the model will be used, with a keen emphasis in retaining an accurate representation of the effects caused onto the target layer. It is a hot topic of research to explore the degree of precision achieved in the current representations at different levels, of a process occurring in the very physical level of a chip.

As the different levels of description entail different properties, those can make advantages or disadvantages from the dependability assessment point of view. In this section logic/RTL level is considered as the target level for a reason: it is the closest representation to physical where valuable injection properties of

repeatability, reproducibility, non-intrusiveness, representativity and reachability (explained in following chapters) are attained at high levels.

The usual models for soft error effects are *bit-flip* if targeting sequential logic [75, 17] and *pulse* if combinational logic is the target [156, 59]. The only difference is bit-flip keeps its erroneous value until a further rewrite operation takes place, while pulse recovers previous value after a constrained amount of time. In addition, they can also produce *indeterminations* if the threshold voltage value has not been fully reached. In essence the previous are considered transient models.

When other processes need to be emulated, such as wear-out or manufacture defects, permanent models are used. The usuals are *stuck-at-1* and *stuck-at-0*, with some other models in less occasions: *open line*, *indeterminations* or *delays*. Those are derived from the physical processes effects shown in Figures 2.1 and 2.2. Additionally, intermittent models can be injected to study wear-out effects in a premature stage, and some types of manufacturing defects only manifesting under certain circumstances. The most popular are *intermittent delay*, *intermittent indetermination* and *intermittent pulse* [64]. These are derived from processes shown in Figure 2.3. Furthermore, other models with lower impact are shown in the 3 figures to explain additional physical phenomena happening at the real devices.

Apparently all expectable fault types should be covered, but due to the new challenges brought by technology, additional models are required to effectively describe up to date reality, which otherwise could show unexpected behavior.

The usual fault hypothesis for long time has been a single fault in the system. However, as technology progresses and with new areas of application being introduced, it is no longer possible to consider just single faults as a safe hypothesis to work with (see Chapter A). With that in mind additional fault hypotheses need to be considered for detection, concocted out of measurements and observation. The procedure thus is to focus into real devices in their application fields and check out how faults/errors unveil to build an accurate model or hypothesis. In that sense this thesis has pinpointed 2 new scenarios.

First of all is the situation where an accumulation of faults can take place, when the system has not been able to clear a previous fault or it is in the middle of that operation. It is important to distinguish this from a well known situation of multi-bit upset (MBU) fault where a single radiation event can flip values of several closely located sequential nodes. In that case the techniques to deal with them vary from the techniques required for the proposed situation. Some intertwining or special arrangement of the information can diminish the effects of the MBU faults. Conversely, when faults of different nature pile-up the story goes differently. In such situations there is a special vulnerability; the existing potential mitigation measures might be defeated and so it needs a special consideration. In this work the name given to such fault models is ***faults with proximate mani-***

festations. Those faults refer to the affectation of a node of the circuit in which another node was already affected and not recovered, in such a way that the second hit can compromise the correct delivery of results even in a redundant configuration. The typical context of application is that of an SRAM-based FPGA where a reconfiguration engine can rewrite or relocate a subsection/replica of the design or *frame(s)*. Because that recovery operation takes some deterministic amount of time, another error can hit the circuits during that time causing a different situation from the starting one which must be dealt with successfully. A model of them is introduced plus a discussion on how to deal with their special characteristics in Chapter A. The model accounts for all possible combinations where **distance between incidence** is as stated **smaller than the recovery period**. In the target technology, an SRAM-based FPGA, 2 planes can be affected by faults: the configuration memory (CMEM) and the fabric of the device. Combinations of different single faults in both planes were included in the model, and Table 2.1 summarizes the applied set, suitable for any SRAM FPGA.

Table 2.1: Considered combinations of faults with proximate manifestation

1st fault		2nd fault	
Duration	Target	Duration	Target
Transient	Comb. (fabric)	Transient	Comb. (fabric)
Transient	Comb. (fabric)	Transient	CMEM [1]
Transient	Comb. (fabric)	Permanent	Comb. (fabric) or CMEM
Transient	CMEM [1]	Transient	Comb. (fabric)
Permanent	Comb. (fabric) or CMEM	Transient	Comb. (fabric)
Transient	CMEM [1]	Transient	CMEM [1]
Permanent	Comb. (fabric) or CMEM	Permanent	Comb. (fabric) or CMEM

[1] Transient faults in CMEM manifest as permanent ones in design logic (fabric) and can be assimilated to them from the logic point of view.

Secondly, as technology nodes enter the deep sub-micron realm, it has been mentioned a width reduction of the faults is noticed at the electronics level. For that reason and due to the heavy filtering these faults suffer (the higher the lower frequency of operation, which is not growing in recent years as steadily as before) new fault scenarios have appeared. To cope with the special nature of those faults, a specific set of models has been defined: the ***fugacious*** *faults*. The most characteristic feature is that their duration lasts less than a clock period. In the past short spikes in voltage values were already accounted as glitches. Those were randomly happening due to a reduced set of causes: occasional EMI, rebounds in ill-designed lines, couplings... Fugacious faults are meant to include the set of causes originating glitches, but broadening the scope to many additional pathologies which also present those symptoms in recent technology nodes. Furthermore, the notion of glitch does not take into account the nature of the fault which, as explained, can be transient, permanent or intermittent. While up to day those short length faults were ignored attending to their innocuousness, they bring an

opportunity to take profit of them as beacons or early indicators that the inner or outer situation around the system is changing. Modeling them allows to test for reaction of systems when operating conditions evolve.

Three different categories of fugacious fault have been established: *fugacious transient*, *fugacious intermittent* and *non-fugacious* fault. The framework to distinguish one or another is the duration of a clock period. A fugacious transient fault will activate only once in the clock period, and will remain active for less than a clock period. A fugacious intermittent fault will activate more than once in the clock period, and each activation will obviously remain active for less than a clock period. Not every activation needs to be active the same amount of time, nor the time between activations needs to be constant. A non-fugacious fault will be categorized as any of those which is active for more than a clock period. More information can be found in the appendices in Chapter B where focus is placed in the models, and ChapterC, where the focus is placed in detection and diagnosis of those faults.

2.5 Summary

In this chapter processes of fault generation, manifestation and propagation have been analyzed from a physical and electrical point of view. Afterwards, models for the physical/electrical phenomena to employ in logic/RTL abstraction levels of design flow have been presented in the case of classical types. As contributions 2 new categories of fault models at that level are proposed, based in the observations at the newest technology nodes: faults with proximate manifestations and fugacious faults. Fault models have a single purpose of existence: to be applied -injected- to system models or prototypes in order to be able to accelerate their occurrence rate (thus reducing experimental time) and study related reactions to them, without loosing much representativeness in the process. That injection is discussed in the next chapter.

Chapter 3

Fault Injection

Every aspect related to hardware fault injection is covered in this Chapter. With that purpose, the many available methodologies with their pros and cons and the existing tools supporting them are examined. After discussion on gaps to cover, a newly developed tool is introduced to the community, which enables the application of newly presented fault models in addition to classical ones.

3.1 Introduction

Fault injection is the process devoted to evaluate by means of testing a design, where the test vectors involve applying faults. The interest of fault injection resides in the fact that a deep knowledge of the behavior of the system in presence of faults can be attained by effective use of injection campaigns. In fact what is tested is the fault, error, failure chain, i.e. whether a fault will turn into an error and later cause a failure, somewhere in the midpoint or if it will have no effect at all.

The are 2 main categories of fault injection targets: software and hardware. Software fault injection mainly consists in altering some pieces of code in the program

or data prior to execution, hence mimicking errors in the coding. Although software fault injection falls out from the scope of this thesis interested readers can refer to [98] for further insights on this type of fault injection. Hardware fault injection is focused in the test of physical problems arising during the operation of the device(s). Again, focusing in hardware fault injection several methodologies can be used which will be commented in a further section.

In order to study and compare fault injection methodologies a first step is to explain the parameters and the properties involved in fault injection processes, to be evaluated later.[1]

3.1.1 Fault space: what, where, when

Faults are defined by a series of parameters that build a fault space: type, location and time. The fault type responds to the question of what is going to be injected, the fault model. Complete information on models has been provided in Section 2.4.

Regarding target locations, it is a topic widely studied since it encompasses with problems such as intrusiveness which can limit the level of representativeness of fault injection experiments. Different types of technology and devices suffer different types of faults. Hence, models are associated with distinct locations where physical phenomena take place, and not others. For instance, bit-flips only apply to standard sequential elements where no special structures have been added to maintain captured value under threat. Conversely pulses can only apply to combinational elements. When soft errors are studied, it is known sequential elements suffer greater susceptibility to them, so they must be included in the fault space of such a study. Combinational signals do not show so much cross-section for soft errors.

Time has an important influence in the fault space: the different injection instants. If a benchmark is studied, to fully test the benchmark an injection in every cycle and of every potential duration should be conducted for each location and model, what cannot be executed for any design with a minimal size.

Summarizing, a fault can be defined by those 3 key parameters as shown in the following function.

$$\mathbf{F}(model, time, location) \tag{3.1}$$

[1] It must be noted that a popular nomenclature to explain fault injection is the FARM model [11], where 'F' stands for *Fault set*, 'A' for *Activation*, 'R' for *Readouts* and 'M' for *Measures*. Fault set and their activation are described by the parameters, readouts refer to the results to be analyzed and finally measures refer to the evaluated coverage values.

It is very difficult to guarantee (if not impossible) that a fault space is *complete* enough, i.e. that every relevant model, location and instant have been included for testing. Obviously when testing a physical phenomenon best choice is to inject all the potentially affected elements at all potential instants to perform a proper analysis. But with complete designs doing so can take enormous amounts of time. Many times previous experience or literature references indicate some guidelines to follow. In any case, every fault space in literature employs the concept of sampling [150, 18, 110].

The key of sampling is to select a subset of faults which maximizes coverage. Generally speaking coverage (or *system* coverage) is defined as the percentage of potential events analyzed (or the representativeness of them) with respect to those the systems will face in its operational life [13]. As a matter of fact the perfect 100% coverage is practically impossible, since unexpected events may affect the operation of the system. However, a high enough value can be accepted as validation proof.

3.1.2 Properties of fault injection

The basic properties of a fault injection technique that can be highlighted are the following.

- **Reachability.** It refers to the capability to reach the desired injection locations of the device implementing the design.
- **Controllability.** The concept can refer to space or time. The capability to control the exact location for injection is the space controllability, whereas the precise instant of injection is selected when there is good time controllability.
- **Repeatability.** It means the capability to repeat experiments in a very similar way, i. e. with very well controlled time and space injections.
- **Reproducibility.** The meaning is the capability to obtain the same results statistically speaking for a constant set-up. It is possible to have reproducibility without repeatability, but the opposite is not true.
- **Non-intrusiveness.** This concept relates to the absence of effect of the fault injection process in the behavior of the targeted system.
- **Time measurability.** The concept refers to the capability to acquire timings related to the injection process, such as latencies.
- **Efficacy.** The property to produce relatively high amount of significant injections. This means few injection targets will remain unexcited or irrelevant to the operation of the design.

The whole set of properties is desirable in an optimal fault injection method. However, up to date there is no single methodology covering the whole spectrum. Nevertheless, academia and industry keep making efforts to find improved injection methodologies and tools.

3.2 Injection methodologies

Focusing on HW faults, different injection methods have been devised in literature throughout the years, each with different injection properties to study. Furthermore, and in close connection to the injection process, a subsequent analysis of results completes the functionality of the methods, providing valuable measurements to process afterwards.

3.2.1 Physical fault injection methods

In this category, it is possible to include those methods which directly inject by physical means in the target hardware. Three different types are commented.

First, heavy-ion radiation is a technique where a radiative source is used for attacking the device. A possible procedure is as follows: a golden device is kept functioning in a clean environment, while a device under test (DUT) is enclosed in a case together with the radiative source and a shutter. Then the experiment starts and the target device gets radiated while executing the workload [10]. The properties of such technique are commented: the reachability is great, since the device is attacked completely, the controllability is none in time of injection (it follows a disintegration law) and low in space (coarse shielding can be used), the repeatability is obviously none, reproducibility is good (it is statistical), the non-intrusiveness is low since a special location must be set for the irradiated device, time measurability would be very difficult since there is no controllability, and efficacy would be high for adequate levels of radiated energy.

Second, pin-level injection employs external interface of the device as a method to force values which are incorrect. The procedure is to enforce the pin to an incorrect voltage by means of another IC. The properties of such a technique are: medium reachability since only inputs/outputs are accessed, controllability is fairly good in space and depends on the synchronizing capability in time, repeatability is good provided synchronization was achieved, reproducibility is good, non-intrusiveness is not perfect due to pin loading with extra capacitance, time measurability is good and efficacy is very high.

Last but not least, electromagnetic interference (EMI) is applied using a probe or plate connected to a burst generator. Its properties are: medium reachability since coupling happens at the input/output pins or power pins, a low controllability both

in space and time since injections cannot be directed or easily synchronized with execution, accordingly a low repeatability, reproducibility and time measurability, a good non-intrusiveness and high efficacy.

Generally speaking, physical fault injection has the advantage of fidelity to the real target. However, there are important drawbacks. Reachability is not as good as desired, and to use heavy ion special facilities are needed, what is utterly inconvenient. Besides not all of the potential faults worth to study can be injected by physical methods.

3.2.2 Software-based fault injection methods

An additional method to inject faults in hardware is to use software as a means to represent the effects of hardware errors, the so called Software Implemented Fault Injection (SWIFI). The method consists in altering the values of register contents or memory in data or instruction space, according to the fault models. The alteration can be performed in compile-time or in run-time. When performed in run-time an injection software must be run in parallel with the target code.

The properties of such a method are low reachability due to just SW-used registers access, high controllability in space and time since the injection is previously inserted in the code, great repeatability and reproducibility, important intrusiveness for run-time methods but low for compile-time methods, good time measurability due to precise insertion point, and a low efficacy if no method is applied to reduce injection targets to used points [151].

It is a low cost method in economic terms and time. However, it is severely limited by the amount of different models to be injected. Its best advantage is to be able to cope with very complex systems at design and final stages alike.

3.2.3 Emulation-based fault injection methods

In the SWIFI methods, the limitation to inject and observe combinational nodes can be overcome by emulation methods. These consist in the use of hardware to inject tailored models using specific methodologies. A popular way is to employ a reconfigurable FPGA to perform injections by dynamically reconfiguring. Several efforts have been done in the past to demonstrate the effectiveness of the methodology with golden runs executed previously, such as in [8], or simultaneously in other device, as in [113]. Additionally it is possible to emulate faults by using the debug interface of processors [158] at the advantage of greater portability.

The properties of those methods are a good reachability for FPGA-based way and medium for debug interface, good controllability in space and time for both, great repeatability and reproducibility, big intrusiveness for both due to the need to stop

execution briefly to perform required changes, fairly good time measurability, and low efficacy if no previous analysis is performed to reduce injection to relevant points [165].

The main drawbacks of these methodologies can be recognized as the intrusiveness, and a lack of fine control for delay models.

3.2.4 Simulation-based fault injection methods

In simulation-based fault injection, the system description is simulated usually at Logic/RTL level (but others can be used) and the different fault models injected at runtime. Two techniques are differentiated to perform the process: simulation commands and saboteurs [65]. While the first one only requires issuing special commands to the simulator, the second needs the introduction of several modifications in the code. Therefore use of commands is simpler, faster and less intrusive. The counterpart is the capability to apply complex multi-node models only by means of saboteurs, though slowing down simulation.

The properties are as follows: reachability is excellent as every single element can be targeted, controllability is excellent as well because simulation is rich in detail, repeatability and reproducibility are also excellent as no variation should appear among runs, intrusiveness is null, time measurability is also excellent and finally, efficacy can be low if no previous restriction is performed in the number of target nodes.

The best advantages are low economic cost and great capabilities. On the other hand, the main disadvantage is the computational time required.

3.2.5 Analysis of injection results

The analysis of results can be performed in differing levels of detail. The possible general concepts to be included in an analysis of results could match the following classification, also known as failure modes:

- **Silent / Not activated.** The experiment has no impact in outputs or state elements.
- **Error.** The experiment has upset at least one of the state elements.

  Latent error. The experiment has upset at least one of the state elements, and it has not been detected.

  Detected error. The experiment has upset at least one of the state elements, and it has been detected.

- **Failure.** The experiment has upset at least one output or it has stopped execution prematurely.

  Silent Data Corruption (SDC). The experiment has upset at least one output or stopped execution prematurely and no detection mechanism (if any) has noticed the problem.

  Detected failure. The experiment has upset at least one output or stopped prematurely and it has been detected by a detection mechanism.

It is interesting to mention that, beside those main metrics, other derived or composed parameters can be provided.

Additionally, propagation latencies can also be measured and studied, what can be very useful for real-time systems.

3.2.6 Summary of methods

In Table 3.1 a summary of different methods of injection is shown concerning different qualitative properties of each.

Table 3.1: Evaluation of properties of different injection methodologies

Property	Heavy-ion	Pin-level	EMI	SWIFI	Emulation	Simulation
Reachability	high	medium	medium	low to medium	medium to high	high
Controllability (space)	low	high	low	high	high	high
Controllability (time)	none	low to medium	low	medium to high	high	high
Repeatability	none to low	medium to high	none to low	high	high	high
Reproducibility	medium to high	medium	low	high	high	high
Non-intrusiveness	low	medium	high	high	medium to high	high
Time measurability	low to high	medium	low	medium to high	medium to high	high
Efficacy	high	high	high	low	low	low

3.3 Injection tools

In order to implement the different injection methodologies, the scientific community has been providing a set of different tools which enabled a wide spectrum of injection possibilities.

3.3.1 Physical fault injection tools

In the realm of physical fault injection, several helping tools to perform pin-injection are available. A few examples include MESSALINE [11], RIFLE [107] or AFIT [111]. The supported fault models in all such tools are transient or permanent stuck-at values in single or multiple pins. AFIT also supports intermittent faults, and the other 2 can additionally inject open-line and bridging models. As stated previously the reachability is limited, but those pins can mimic the effects of internal faults.

Other physical techniques such as heavy ion radiation or EMI generation do not require of specifically designed tools, but instead use physical fixtures or commercial devices.

3.3.2 SWIFI tools

A plethora of SWIFI tools have been developed by many universities and companies alike. The following is a reduced representative subset.

Among software implemented fault injection tools, there are 2 variants: compile-time and run-time. Examples of the first type can be FIAT [153], or GOOFI-2 [158]. All these share many commonalities. In FIAT the task code is altered with a collection of faults generated from a library, and later results are collected and compared with a golden run. Validation of the inserted “bugs" w.r.t. the hardware faults is left unexplored. In GOOFI-2 this mode is called “instrumentation", and works by inserting special routines in the application code. In this case the temporal intrusiveness can be very high, depending on the workload.

Moving to run-time, there are proposals like Ferrari [91], Xception^{TM}[38], BOND [14], MAFALDA-RT [140], or GOOFI-2 again, in an exception based mode. Ferrari employs traps and system calls to inject and read results by using 2 concurrent processes. Xception is one of the few commercially available and supported tools. It works inserting exception calls to routines to inject bit-flips and analyze the consequences and latency. BOND intersects the communications between application and Windows OS to inject and read out the effects and also applies bit-flip models. MAFALDA-RT targets to reduce the temporal intrusion by stopping the hardware clock of Real Time (RT) systems, but it needs to use emulation for specific areas of the design. None of them injects permanent faults.

3.3.3 Emulation-based injection tools

A good amount of recent tools in the community rely on emulation to perform injection. Some of them modify the design to include HW injectors (instrumenting the design), such as the one from Politecnico di Torino [32] or the Autonomous Fault Emulation tool in [106]. Others take profit of dynamic partial reconfiguration capabilities of SRAM FPGAs to apply changes in the designs, which mean more types of faults can be injected. An incomplete list could include FLIPPER [4], FADES [7], FT-UNSHADES2 [113], or others. Finally some tools can use a debug interface such as Nexus to emulate faults, as is the case of GOOFI-2 or JTAG as in the tool from U. Carlos III [130].

The tools which instrument designs have a greater intrusiveness, but allow for easy controllability of the injection points and models regardless of the base technology. The tool from Torino can operate in 2 modes depending on the detail desired for analysis: one with error analysis and another without it, where analyzing take longer executions. It stores golden run executions at the host machine. The Autonomous Fault Emulation tool implements 3 different methods to instrument the design, which are executed at the HW speed. The variety covers methods to reduce area overhead of the golden circuit to only doubling the FFs, or to increase execution speed. The goal was to avoid the bottleneck of constantly communicating with the host.

For tools based in reconfiguration, there is a difference in the capabilities, since not only the target design is tested against single event effects (SEE) like in previous tools, but also the platform (SRAM FPGA) can be tested for weaknesses. In that sense, FLIPPER can performs alterations of bits in the configuration memory (CMEM) sequentially to check for effects at the design. Moreover, FADES adds additional control on the modifications to implement a huge catalog of injectable faults. It uses the package JBits [72] to gain access to precise bit positions coupled to the target element to inject. The drawback is that it depends on the availability of JBits to support any other base platform. FT-UNSHADES2 has been in use by ESA to validate several designs due to its powerful features. The system comprises 5 FPGAs within 3 PCBs (or 3 FPGAs and 2 target ASICs). With this big amount of HW, in can be target technology-independent, at the cost of supporting less fault models, and especially not being able to inject in combinational logic. The speed is very fast, reducing communication with host, unless latent errors analysis is required –what turns the system slower.

Tools based in debug interfaces like GOOFI-2 can access more positions than standard SWIFI, but cannot use as many fault modes as FPGA tools. Also, they depend on the availability of a Nexus (or similar) interface. The more common JTAG interface is employed by the tool from Carlos III.

As a bottom line, all these tools require a final implemented design which fits comfortably into an FPGA supported by the tool or a design running in real Nexus or JTAG enabled hardware.

3.3.4 Simulation-based injection tools

In the category of simulation-based tools, there are plenty of choices to mention. An incomplete list can be the following: MEFISTO [86], VERIFY [157] or VFIT [15] which operate at the register transfer level. Additionally, some variants operating at different levels include the tool used in [77], SWAT-Sim [103], or ASPHALT [177] to name a few.

MEFISTO used the VHDL description in which commands were issued to inject faults into variables and signals. It boasts the ability to use a network of workstations, but includes a limited set of fault models. Two variants were developed at CHALMERS (Sweden), MEFISTO-C, and at LAAS-CNRS (France), MEFISTO-L. VERIFY extends VHDL language to add descriptions of faults linked to a specific component in a multi-threaded strategy. Its greater speed is penalized with the need to modify the VHDL language itself. VFIT greatly extends the usable fault models. It also employs VHDL descriptions but implements 3 different techniques: simulation commands, saboteurs and mutants. The main limitations are the inability to handle medium-big complexity designs and the limited speed.

Among tools which do not operate at the RTL description, the one in [77] does it at the microarchitectural description. This provides an enhanced degree of speed at the cost of accuracy. However only the architectural elements are modeled what means anything else is not accessible for injection or analysis. SWAT-Sim tries to convey the best of both worlds by injecting at the gate level and then propagating the information to the microarchitectural level in a hierarchical way. While this works well for most types of faults, the issue is to find a proper boundary for the exchange of information without loss of propagation details. ASPHALT on its side mixes gate-level injections with RTL simulation, but with a tailor-made language (ASPHALT itself). It shows great coverage of gate-level faults at the RTL, though obviously requires the work of coding the design in its own language.

In summary there are tools to simulate gate-level, RTL, μarchitectural or a combination of those to end up employing the highest possible abstraction for speed. For an accurate and flexible evaluation of designs in the framework of this thesis, low level injection was required. No tool provided that with adequate capability to work with complex designs at an acceptable speed without loss of information. For that reason a new development was built on the grounds of previously available VFIT experience: the FALLES tool.

3.4 The FALLES Tool

3.4.1 Presentation

In the present thesis, a new simulation tool was developed to support fault injection and analysis of different designs with a specific set of requirements and constraints in mind. Its name is FALLES for **F**ault injection and **A**nalysis for **L**ow **L**evel **E**valuation **S**uite. It was designed to exploit the maximum performance available in the computing platforms of either a modern workstation or a cluster machine. As a simulation-based injector, it boasts all the properties from which non-intrusiveness, reachability and measurability can be highlighted. The infrastructure which supports the tool is a set of scripts developed in TCL and AWK, which interact with an HDL simulator from Mentor Graphics (Questa). The tool was presented in the paper in Chapter F of the appendix.

In terms of fault models, it supports the most common types of permanent, transient and intermittent categories plus the fugacious fault models, as shown in Table 3.2. The architecture is designed to ease the introduction of additional models, since only minor modifications would be required to a few scripts. **FALLES** was employed in Chapters G and H for conducting extensive injection campaigns where injected models were stuck-at-1, stuck-at-0, open line and transient indetermination. The current implementation is limited to independent (i.e. not short circuits, etc..) fault models as imposed by the use of simulation commands.

Table 3.2: Fault models currently supported by FALLES

Duration	Model
Transient	Pulse, bit-flip and transient indetermination
Permanent	Stuck-at, permanent indetermination, open line
Intermittent	Intermittent stuck-at, Intermittent pulse, intermittent indetermination, and intermittent open
Fugacious	Transient, permanent and intermittent set of previous models in the fugacious time frame

FALLES can work at RTL and gate level, conversely to previous tool VFIT which could not manage the complexity of gate level or medium-big sized RTL descriptions. When gate level is targeted the results can be as accurate as those obtained by an emulation tool such as FADES, with the versatility, compatibility and observability of the simulation techniques. For the bigger designs it has been applied

in detailed synthesizable RTL descriptions with success, and for medium designs gate level has also been targeted. Manageable designs do not have any theoretical limitation further than physical resources. The tool helps to accelerate and automate extensive simulations, not forcing the user to employ sampling in mid-sized designs. To reach that goal a cluster infrastructure is handled to support calculations, with different cluster management systems potentially supported (SGE based, extensible to others).

In relation to the timing, the instant of injection can be very important. In **FALLES**, it is possible to use a single instant of injection or, alternatively, choose a normal distribution between two time points. For the case of intermittent faults, different separation times between activations can be selected, from fixed to variable in a normal distribution thus allowing for a more complete set of possible models. To study different combinations of injection instant and targeted node, the former can be assigned randomly or sequentially to the later to provide extra flexibility.

3.4.2 Detailed operation

In Listing 3.1 a set of the main scripts participating in FALLES is shown, where .do and .tcl extensions feature TCL code and .awk feature AWK code. TCL refers to Tool Command Language and is widely used for engineering software scripting. In this case it was chosen because Modelsim supports that scripting language. AWK is a popular powerful text processing language for linux. It is installed in most linux distributions and is widely used in that environment.

Listing 3.1: Main scripts comprising FALLES

```
simConfig.do              simReadResults.awk
simFaultsMMP.tcl          simAnalyseMMP.tcl
simFaultsCMP.tcl          simAnalyseCMP.tcl
simFaultFree.do           simbgexec1.10.tcl
simFaultModel.do          simInjections5.do
simGenerateNodes.do       simInitModel.do
```

The names explain quite well the function of each of the files. The main files are simFaults* and simAnalyse*, which are followed by MMP or CMP depending on the variant for shared memory machine multi-processing or distributed memory machine multi-processing (cluster). The main reason behind producing 2 variants is the different management of the simulation tasks in those environments. The shared memory version was produced for use in workstations, where smaller designs can be perfectly managed. Once larger designs were considered, the need for more power drove the development of a cluster version to exploit the available computing resources of the university campus. Either way the procedure is the same, only varying the management routines to start and gather simulation tasks.

In this architecture it is possible to launch the injection followed by the analysis or alternatively perform each in a separate operation.

Its workflow can be described in the following points, which are also pictured in Figures 3.1 and 3.2:

1. Either the shared (MMP) or distributed memory version (CMP) is run from TCL shell depending on the available platform.
2. Configuration is read from the configuration input file. Includes number of cores and packets for injection, number of cores for analysis, choice to perform analysis immediately after injection, filters to apply to trace data, configuration of experiments campaign(s) such as fault model(s), duration, injection instants, etc.. and component / level of description to inject with the desired process corner(s) parameters.
3. A reader module builds the target nodes list if it was not already present (allows to manually tweak it).
4. Golden run is executed until first injection instant, then checkpoint is saved, then continues execution until end of workflow saving information of the fault free trace and filtering it accordingly as defined in configuration.
5. To execute golden run, content to save to trace is initialized by reading it in a specific input file.
6. Injections report is prepared
7. Distribution of injection experiments in multiple packets and processing cores using background processes, and launch of executions from previous checkpoint. Apply filtering as configured to saved traces.
8. When all cores are done, gather injection traces and reports into a centralized location and report file
9. Repeat 4-8 for opposite process if both process corners evaluation for gate-level is desired, i.e. if the first pass was performed using maximum delays then repeat using minimum delays and vice-versa.
10. Launch analysis if configured that way by user.

Injection is carried out using *force* commands, where *deposit* or *freeze* variations are applied depending on whether the target is a sequential or a combinational element. If deposit is used the value is held until a new source drives the node, whereas if freeze is used no driver will change the value of the node until the force

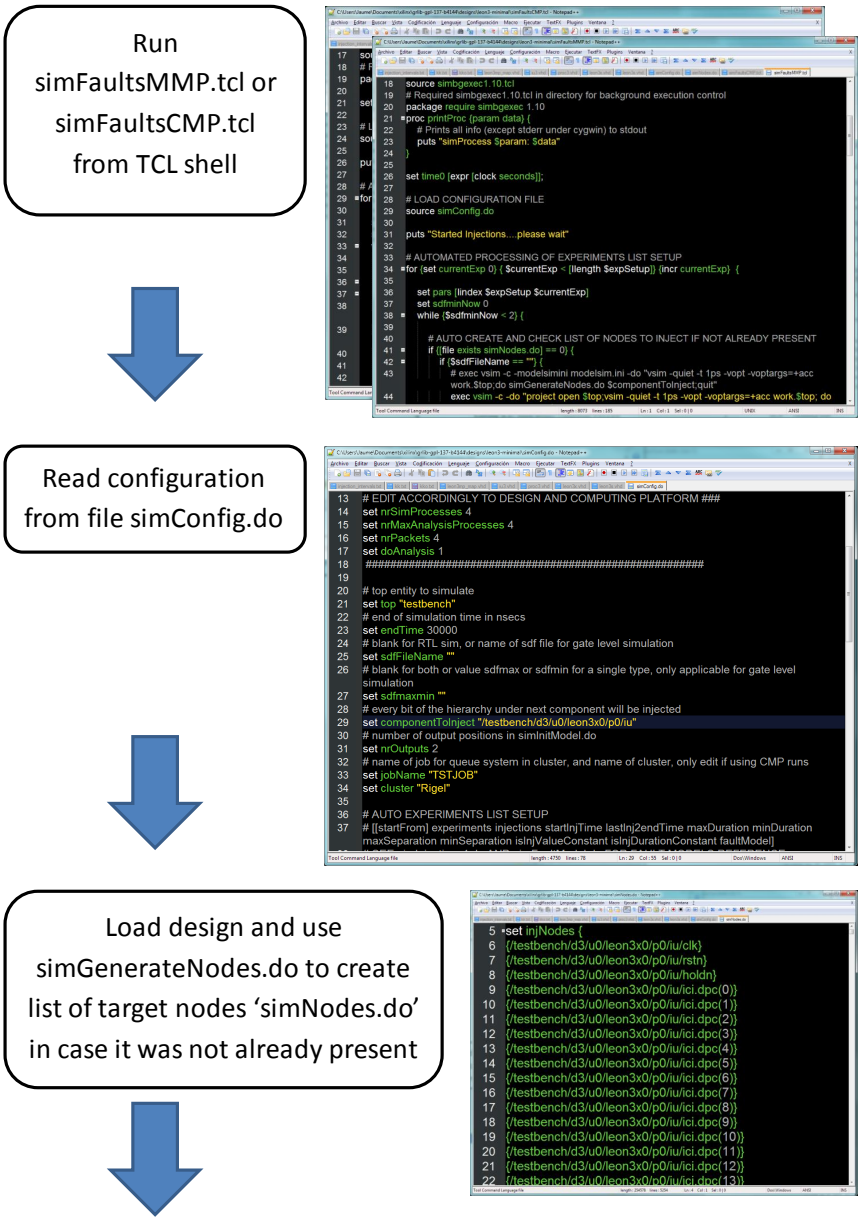


Figure 3.1: Workflow for FALLES, part1

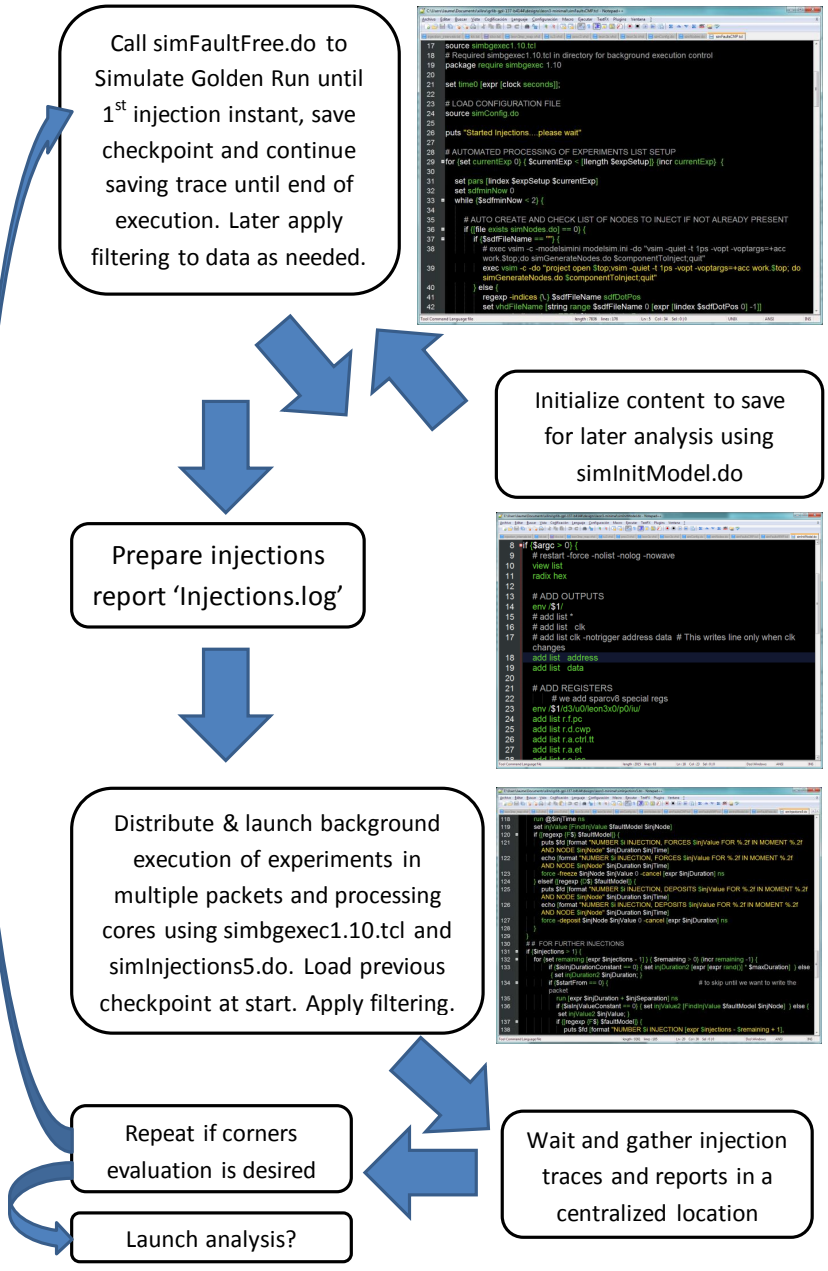


Figure 3.2: Workflow for FALLES, part2

command is revoked. The duration is calculated according to the configuration, and the forced values can be '1', '0', high impedance 'Z' or indetermination 'X'.

As stated there is an inherent parallelization of the execution of the injections and analysis, which has been implemented for both multi-core and cluster (grid) topologies for versatility. The total number of injections is divided into the maximum amount of cores and packets are created as convenient for later analysis. For instance, 100 cores can be used for injection but 500 are available for analysis (this can be a real situation provided simulator licenses are needed only for injection). Hence, each core creates 5 packets of trace data for later analysis and in the analysis phase each core will take care of one packet, accelerating the latter phase. Since code for management of parallel execution is located in a single file, it is possible to adapt it to new clusters/platforms.

A possibility to filter results on the fly is included to restrict or reduce the required data space. This enables, as an example, to observe only a subset of the bits of the positions under analysis, or to perform some combination operation to reduce the size of the saved traces. Storage has turned one of the most limiting factors found in our research.

Finally, additional features are the possibility to inject sequentially the list of nodes or randomly choose among them to create a more even distribution. Also option to continue from the last injection if additional experiments are desired is included. All the saved information is coded in plain text to ease the task of creating or debugging personalized analysis algorithms.

3.4.3 Analysis in FALLES

The analysis of results can be performed in several ways, depending on the desired measures to obtain. The needed step to achieve any measures is to get some readouts to analyze, which in the case of **FALLES** are the traces with the selected elements to capture. From there, if using the default analysis algorithms provided with FALLES, experiments are classified according to predefined categories. Those categories are related to those mentioned earlier in section 3.2. However, due to the practical use of FALLES performed in this Thesis, a slight variation can be appreciated. There was interest in a set of papers included in this Thesis, where detection of errors at the superstructure is only possible at the output boundary of the lower hierarchy block when a failure is present, to include the following categories:

- **Silent / Not activated.** The experiment has no impact in outputs or observed state elements.
- **(Latent) Error.** The experiment has upset at least one of the observed state elements, but not the outputs.

- **Failure.** The experiment has upset at least one output or it has stopped execution prematurely.

  Failure after error observed at the state elements. The experiment has upset at least one output after at least one error has appeared in the observed elements.

  Failure without any error observed at the state elements. The experiment has upset at least one output without showing error in the observed state elements first. This means that, while an error must exist in the system before failure, it was not reflected at the selected elements for observation, i. e. the error propagation did not impact the observed elements.

The analyzer can be adjusted to taste as per-case. For instance if a detection architecture is studied a new category 'detected' can be included, or if correction algorithms are present the same can apply for a 'corrected' category.

Additionally, in the default analyzer latency of propagation is also provided in maximum and average values, plus histogram of hits. It is depicted in Figure 3.3. It provides time elapsed between injection instant (A) and the moment an error or failure first appears (can be observed, points B or C), and from first error (B) to first failure instant (C). The value of these measures is most appreciated for real-time systems where reaction time is tightly constrained, but it can be employed in many ways. A last provided feature is the distribution of errors in different state elements throughout the whole campaign (number of 'x' in each S element). This delivers valuable information on which elements are most susceptible of suffering upsets, and thus perfect candidates for protection measures.

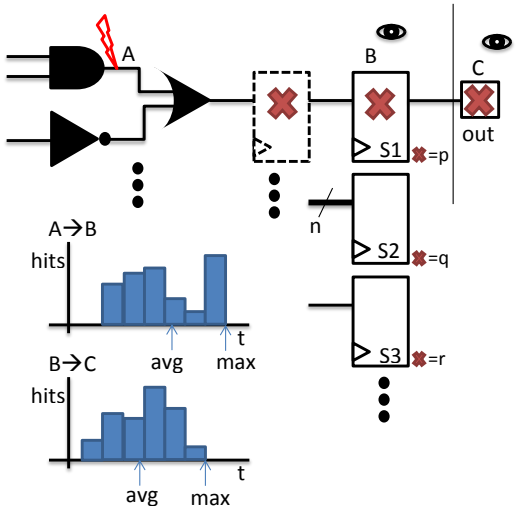


Figure 3.3: Latency analysis and error count in FALLES

3.5 Summary

In this chapter fault injection is dealt. Attention to several methodologies and techniques to inject faults has been paid, identifying strengths and weaknesses for each of them according to relevant properties. A study on a number of available tools of different types presents the advantages each of them provides, and underlines the lack for a customizable well suited tool to perform research on very fine time measurements or alternative fault models.

After study, a new fault injection tool, FALLES, is presented to fill the gaps identified in other solutions. Its capabilities and behavior are thoroughly explained. The availability of such an injector is unleashed in the area of dependability assessment, and more specifically in the robustness assessment, which is the target of the following chapter.

Chapter 4

Dependability Assessment

The following chapter covers the topics of how to perform proper dependability assessment in the context of an ever increasing demand of certifications compliance, and how to improve efficiency in this process to reduce development cost and redesigns.

4.1 Introduction

Out of the dependability attributes, as defined by Avizienis [13], it is possible to cite the 3 most relevant to embedded systems and their certification standards: reliability, availability and safety. In all 3 attributes the important concept is the justification that the intended characteristics are met, and how is that justification achieved.

In dependability assessment, there are 3 different techniques to verify required dependability attribute goals for standards certification have been met: *analytical study*, *field experimentation* and *fault injection testing*. Each of them cannot be applied at every design stage, but only at specific stages related to the technique, nevertheless some of the techniques can be used in several stages. Fault injection, explained in the previous chapter, is recommended for instance in the automotive

standard ISO26262 [81] Part 5 –safety requirements of hardware, where injection into models is deemed appropriate whenever hardware fault injection becomes specially difficult or ineffective (requiring irradiation tests, etc...).

The method requires realistic workloads to be applied, a non-interfering injection system and a targeted analysis to extract the measurements which will provide the cornerstone for obtaining useful dependability measures as required by standards. There is yet a further matter to solve: are specifications towards dependability correctly captured by the evaluated implementation or model? The following sections deal with the opened questions.

4.2 Analysis of injection results

The results of injection during testing for dependability assessment must be carefully studied to discover potential system weaknesses, or dependability bottlenecks. In the pursue of delivering accurate figures for reliability, safety and so on, a parameter which is considered fundamental is the percentage (or probability) of failures observed at the boundaries of the element under analysis, as provided by the tool presented in previous chapter 3.4. To get it an injection in all or a carefully selected representative set has been performed. Later, with the combination of that value and the expected raw rate of occurrence of faults in the system for the established mission profile, a failure rate is obtained. That is the standard unit to measure reliability, and is given in Failures in Time (FIT) where 1 FIT equals 1 failure every 10^{-9} hours.

The availability of a critical system plays an important role in the dependability studies. There are distinctions for systems which imply continuous operation and others which operate occasionally, mainly in the methodology for safety classification. It is noteworthy that for systems operating continuously, the reliability levels are remarkably more demanding.

In safety qualification, applicable standards such as the European IEC 61508 establish 4 different categories, or Safety Integrity Levels (SIL 1 - SIL 4) according to maximum expected risk. Though some others use the opposite ranking, in IEC 61508 level 4 means the safest. This is valid through all the field specific standards derived from it: ISO26262 for automotive, EN50126 / EN50129, for railway or IEC61511 for process industry, to name a few. In order to establish the taxonomy, there are separate tables for continuous operation systems and occasional operation ones. The tables refer to figures of dangerous failures in time, which relate to those figures obtained in reliability study, but with a special qualification (not all failures may be dangerous/catastrophic to safety). The tables are 4.1 for systems occasionally used, and 4.2 for systems continuously in use.

Table 4.1: SIL levels for systems occasionally used

SIL	PFD	D-FIT
1	0.1 - 0.01	$10^8 - 10^7$
2	0.01 - 0.001	$10^7 - 10^6$
3	0.001 - 0.0001	$10^6 - 10^5$
4	0.0001 - 0.00001	$10^5 - 10^4$

PFD: Probability of Failure on Demand
D-FIT: Dangerous Failures In Time

Table 4.2: SIL levels for systems in continuous use

SIL	PFH	D-FIT
1	0.00001 - 0.000001	$10^4 - 10^3$
2	0.000001 - 0.0000001	$10^3 - 10^2$
3	0.0000001 - 0.00000001	$10^2 - 10$
4	0.00000001 - 0.000000001	$10 - 1$

PFH: Probability of Failure per Hour
D-FIT: Dangerous Failures In Time

In the **FALLES** tool developed in the course of this thesis, it is possible to extract information which helps to determine the number of failures which cause the system to stop execution unexpectedly, what is presumably critical to safety. But not only that information is available to the user. With additional analysis of selected registers or components, the user can evaluate the effectiveness of deployed fault tolerance mechanisms, or study the propagation characteristics of faults inside the pipeline. In this sense, one of the types of analysis which is delivered by the tool is a quantification of different propagation delays for faults. This allows to know, for instance, how much execution time has passed from the moment some internal value is upset until an erroneous output is delivered. Very popular automotive microcontrollers, for instance, use lockstep architectures, where 2 replicas perform the same operations on identical data and compare results at the outputs. If they don't match, a rollback and re-execution is launched. The delay that takes a wrong value to reach any output can prove critical to real-time tasks to deliver correct results and on time [78].

Another study enabled by the tool is the propagation of faults to a certain preselected subset of elements, such as state elements. Using that capability, in Chapter H a paper is presented where architectural registers are selected as the subset, out of a real critical applications embedded processor pipeline. The focus is placed in

the results which would be obtained after injection in architectural level description of the design. When a comparison of the results considering injection at the architectural level and injection at the full detail of RTL is performed, it is clear some noticeable differences appear. The relevance of the result is that the estimated FIT value for the design when architectural level is used will differ importantly from the value obtained after RTL injections. This relates to the representativeness of fault injection results obtained at different abstraction levels. In essence, this reduces the usability of the injection results at the architectural level. Sadly enough, from a practical point of view the benefits of testing at a higher level of abstraction are substantial. Mainly, higher speed of test due to faster simulations, earlier availability in the design flow and hence the possibility to perform cheaper redesigns. Therefore efforts were focused to investigate a multi-level correlation in the testing process, in the sake of increasing confidence in the architectural level injection results.

4.3 Multi-level correlation

When injecting faults for dependability verification or assessment, it is notable that the closest to hardware the model to be injected, the more accurate the extracted information on the system behavior is going to be. However, the capability to test every fault model of interest in hardware is only reserved to the last stage of design if possible at all. Hence, and as explained in the previous section it is convenient in many ways to be able to perform accurate fault injection in the higher levels of abstraction, making an effort to reach an acceptable degree of representativity.

Microarchitectural simulators, or instruction set simulators, can model a virtual prototype with acceptable data and cycle accuracy, but neglecting most of the underlying infrastructure (usually mainly because it is not yet defined at that stage of design). They implement uniquely those elements that are defined in the architecture and constitute a minimal set of features. The benefits are a tremendous simplification in the simulations which means delivery of results is quick enough to iterate as required during design phase. Beside that, they come at zero cost, given the fact that software development also requires from that sort of simulators so they are already available for the software team.

In the multi-level correlation topic, this thesis introduces some efforts which were focused in a specific architecture but, without loss of validity, apply to any other. To begin with, Chapter F presents the original idea of injecting at the RTL and observing the propagation to those elements of the architectural description, and Chapter G demonstrates a method to correlate the injection information obtained at the RTL level with that available at the Instruction Set Simulator (ISS) for automotive environment. The key is to establish a parameter extracted at the ISS which conveys the failure probability obtained after injection at the RTL. The

mentioned parameter was the **instruction diversity**. The tested fault models were permanent, where existing correlation has been found with profiling of the benchmarks as extracted by ISS. A statistic analysis concluded that the behavior of the failure probability is logarithmically related to the instruction diversity value. That means for those fault models that after a quick profiling of the software to execute by ISS, to obtain the key instruction diversity parameter, and using the curves obtained in the paper or the corresponding equation, it is possible to provide a value for the probability of failure without even simulating. Such value is a fair approximation to the final value obtained by RTL injection, but at a much smaller time budget and without the need for an RTL description. The limitation is the method is only validated for permanent fault models.

Several correlation efforts had already been presented before in literature, though not providing the final figure for failures with such a direct approach. An interesting work by Maniatakos et alt. [109] linked instruction level effects of low level faults for permanent stuck at models and transient bit flip faults. It provided a rough classification of unexpected effects at the instructions side. Another effort by Li et alt. [103] focused in the results of injecting the same stuck-at and delay models at gate and μArchitecture levels, where poor matching was found.

When targeting multi-level correlation for any fault model, a possible methodology to ensure testing process at different levels represents the same physical reality is to find common points to check for concordance, as in Chapter H. In it, architectural registers were examined to determine, out of all the injections which had been performed, how many had impacted those registers, and how many had not. In that case the study focused in permanent fault models, but it has been further extended in upcoming publications. Up to day additional investigations to be published have demonstrated that, for permanent fault models, injection results can be even more accurately estimated by employing a parameter called *toggle coverage*. This relates to the utilization of each o the areas of the processor by the tested benchmarks. The amount of failures not reflected at the architectural elements of the system is important for permanent fault models, conversely to what happens for transient fault models. In the latter, estimation using profiling is rather inaccurate. Therefore, for transient fault models actual injection at the architectural level is utterly recommended since (i) it is more precise than for permanent fault models and (ii) the results cannot be estimated in advance by instructions profiling.

4.4 Summary

In this chapter the assessment on the dependability of systems is studied. After focusing in the fault injection testing methodology, an analysis of dependability attributes obtainable by FALLES is presented. Critical products development necessarily includes a testing stage to check those dependability attributes. Hence, with the purpose of accelerating the design process while maintaining requested dependability standards, multi-level correlation of injection results and profiling information was studied to understand the attainable accuracy by employing early (high level) fault injection testing methods.

The outcomes of such testing result often in the insufficient level of dependability as required by specification. That means one or many fault tolerance mechanisms need to be applied to increase the dependability level. Those need to be tailored to the fault assumptions considered for the mission. The next chapter offers a discussion on that topic.

Chapter 5

Fault Tolerance Mechanisms

In this chapter, fault tolerance mechanisms for embedded systems are covered. Within the concept are included the areas of fault and error detection, error handling, fault diagnosis and recovery. Though it is clear those are ample, cross-domain topics, the spectrum of this work is restricted to hardware in the current and upcoming technology generations. Previous work is mentioned along contributions to the field within this thesis.

5.1 Detection

When it comes to error detection architectures, there exists a plethora of different proposals to cope with the job. In the first place it is important to distinguish on-line from off-line techniques. Traditionally off-line detection has been performed as test vectors exercised right after manufacture, or as periodic checking routines in critical systems. The target was to either detect potential design or manufacture errors or to recover from errors in a gross time frame. Obviously in off-line techniques a latency of detection applies, which may be unacceptably high for that purpose. Consequently, on-line techniques are much more targeted towards

improving the robustness of system components against potential faults that may affect its behavior at runtime. They become the adequate solution to applications where no big detection latencies are acceptable. In essence, concurrent detection provides the mentioned low latency of detection, as it happens during normal operation of the system.They enable the system to react to errors, not originated by ill-design but operating conditions, while running and in an automated or assisted way. Different base techniques have been employed to perform detection concurrently: in space redundancy options are replicas comparison[20, 166] or tailored function checkers [167, 173, 152], in information redundancy the use of coding [74, 19, 174, 21, 61, 179], and in temporal redundancy the re-execution and comparison [93, 138, 47].

Some of the most widely used devices are SRAM based FPGAs, with an extraordinary recent growth in capabilities. It is notable that for critical applications like aerospace, defense, etc. special types of FPGA devices have been used over time. Nevertheless commercial or industrial grade SRAM FPGAs are now finding their way in those markets due to pricing and functionality. In this thesis attention was paid to the use of those FPGAs in critical contexts. Because of their partial, dynamic reconfiguration possibility, it is easy to foresee the situations where a reconfiguration process is engaged due to a previously detected fault and a new fault hits the system, or generally speaking the previous fault is being cleared when a new one arrives. That has been named in this thesis "faults with proximate manifestations" (see Chapter 2). Because when a reconfiguration process is in operation the outcomes of the affected section are not under control, so incorrect error masking could take place. Thus, it is important in order to avoid safety hazards to detect that situation and react connecting or disconnecting the erroneous section's outputs accordingly.

One of the contributions of this thesis was to present and test a mechanism **suitable to detect those complex situations**, as proposed in the work of Chapter A. By introducing a supporting finite state machine (FSM) combined with existing spatial redundancy strategies, such higher complexity scenarios can be adequately noticed. The mechanism is named TMR-MDR for *triple modular redundancy with module discard and repair*. With the sole use of previously existing mechanisms such as those earlier described, faults appearing in proximate instants of time could not have been accurately detected, thus probably causing unexpected hazardous situations. The working principle is to compare only those replicas which the FSM has established to be reliable, disregarding incoming information from the others. This qualified replicas keep changing with the recovery of correct operation in a dynamic fashion.

Moving on, after extensive characterization of faults affecting deep sub-micron technologies (see Chapter 2) the growingly important *fugacious faults* were tackled. Contributions towards **detection of such new fugacious fault models** was first proposed in Chapter B, where a schematic proposal was first presented. A further

contribution was to fully implement and test the proposal, as demonstrated in Chapter C. The basis of the detection lies in the use of an observation circuit which operates in the stability period of the set of signals in the bus. While this would not be a novelty ([39]), the addition of an infrastructure to effectively increase that stability period over which observations can be performed boosts its effectiveness sharply. Such infrastructure operates equalizing propagation times of combinational signals throughout the stage of the pipeline, so that switching period of the total clock period is reduced. A methodology to apply such detection method is also described, along some promising results of testing in small blocks. The latter solution is not restricted to FPGAs like the first, but can fit any VLSI design.

Beside the previous 2 contributions, where introduction of new applications and technology are the driving cause of the efforts for enhanced detection of new fault models, the commonplace for embedded systems' dependability is also reviewed for improvements. Truly, embedded systems can face changing operational conditions and can be usually deployed in harsh environments. Hence, one of the consequences is their communications can get corrupted fairly easily. To quickly explore the design space and deploy tailored solutions to each detection problem, a *metaprogram* -which defines a set of transformation rules from HDL code into a different HDL code in this case- was developed for automated CRC hardware block generation (Chapter E). That metaprogram matched an infrastructure called **CODESH** (for open COmpilation process for Design of dEpendable and Secure High-level HDL descriptions), created to enable the automated application of fault tolerance and security strategies, by means of open compilation (Chapter D). The advantages provided were ease of use and deployment and the lack of human intervention, what ensures high quality results with virtually no mistakes. Other detection strategies were developed as metaprograms to enable easy problem-free deployment, such as re-execution and comparison or Hamming codification. The novelty of these efforts was focused in the ability to **automate the addition of non-functional mechanisms to designs**, enabling a low-cost and effort method to adapt the system to operating conditions demands dynamically.

After effective detection of faults a proper recovery can result in improved availability, reduced cost of operation or increased reliability. For the recovery of detected errors proper handling is needed. Further on, if a correct diagnosis of the offending fault is available the recovery from it will be more efficient, durable and safe. These topics are discussed next.

5.2 Error handling

Once errors are detected, fault tolerant embedded systems require some kind of handling strategy to limit their unwanted effects. The concept of handling involves avoiding the process of errors turning into failures (incorrect outputs of the service under study).

In the sphere of error handling, there are 3 main known methodologies to apply:

- **Rollback.** To return to a previously saved correct state (checkpoint).
- **Rollforward.** To jump to a state without errors which should follow the last correct state.
- **Masking.** To correct the erroneous data using redundant information in the current state.

In terms of architectures to implement the methodologies, there are all sorts of proposals exploiting different principles. In this work software level solutions are out of the scope, hence placing the focus on hardware solutions. The most widespread method across different systems to handle the effect of errors is physical replication for masking. The simplest form, triplication and voting, has been around for some years and still provides one of the best error handling capabilities [20]. Some automated tools for its deployment have been developed for FPGAs [188], or for designs at the EDIF level [182]. In this work, an additional possibility is provided as a contribution: to apply physical replication at design entry level (Chapter D) by means of a specific metaprogram and easy-to-use configuration commands.

Other widely used technique (which is also extremely popular) is re-execution of instructions, pipeline stages, etc... It saves valuable amounts of area to provide error handling, but is ineffective for handling the errors caused by permanent faults [120, 47]. A novel method to quickly apply temporal replication for re-execution at entry level is featured in Chapter D, where a metaprogram is in charge of performing the required transformations and addition of infrastructure towards correct bug-free implementation of the architecture.

Error correction codes are a third way to handle errors. While these can be rather flexible [148, 155], they provide a remarkable timing overhead for encoding/decoding procedures and an extra area occupation. Their suitability will depend on the performance/sensitivity to errors relationship required. Again a novel flexible method to apply Hamming code correction at entry level is presented in Chapter D using metaprograms. Finally, combinations of different techniques can be found throughout literature, many times using replication with comparison and additional re-execution as needed, or codification or a combination of all of them [105]. In that sense a contribution in this thesis in Chapter A provides error handling

for specific degraded periods of operation in FPGAs, by means of combination of area and time redundancy.

5.3 Fault diagnosis

Once a fault is detected, irrespective whether errors are effectively handled or not, correct or mistaken diagnosis can make a huge difference in the job of a FT strategy. Take, for instance, a quick analysis based in the dependability attribute of reliability. If a transient fault appears in the system, and it is mistakenly diagnosed as a permanent fault, it could well force a reset or power-cycle to unwillingly recover from it. That causes a *downtime* which negatively impacts reliability. If, on the contrary, permanent faults are taken as transient ones, the increase in the probability of accumulation of faults in the system is noticeable. This yields diagnosis is as important as detection in order to provide a complete optimal solution for dependability.

Following the stated example, several previous works have provided good insight distinguishing permanent from transient faults in the domain of VLSI systems [39]. This makes a good leap towards better dependability and use of resources. In this thesis, the diagnosis method presented in Chapter A was based in the previous reference, with added support for combinations of faults which coincide in time. Nevertheless, intermittent faults were not included in the taxonomy considered in that piece of work.

Much effort has been later put in properly diagnosing intermittent faults, as differentiated from isolated transient faults [22, 37]. It is not a trivial problem, which is dealt at the lowest level hardware in some solutions and at higher level controllers in others. In any case adequate thresholds for the recurrence of faults in a certain location are key to perform correct distinction. That information will be given by experience, and always related to the base technology and operation environment. In the framework of this thesis, attention to intermittent diagnostics at the low level was placed in Chapter C. The contribution was to **enable diagnosis of intermittent faults at a very small time frame**, compared to the clock period. While it is referred to the clock cycle time frame, other combinations of recurrence threshold and time frame may be equally valid, when adjusted to specific technology and conditions. In the presented architecture, a repetition of 2 upsets in a clock period is tagged as intermittent fault. At that time frame, diagnosis information must be used as input to higher level managers which can consider additional inputs to perform a more accurate diagnostic.

5.4 Fault recovery

After proper diagnosis, the next stage is to recover from faults. Fault recovery refers to the application of changes in a faulty system devoted to return it to a non-faulty state.

Recovery of faulty systems is performed when the fault is not leaving the system on its own, i. e. it is not a transient fault which will disappear after some (short) period of time. In SRAM based FPGAs, where this thesis has placed focus, it is a powerful tool to increase dependability. For instance, when the fault affects the configuration memory, where logic functions implemented by the device are set, the problem will remain until a rewrite operation of that memory takes place. For that reason modern FPGAs usually implement a *scrubbing* engine, which continuously checks and rewrites as needed the different data frames to recover correct values when these have been altered [19]. Likewise, a fault can also affect an element in the fabric (logic) of the device. In any case, the faults can be usually corrected using the previous technique when their nature is transient. However, permanent faults will not be corrected by simply rewriting the configuration cells with the proper value. In such cases relocation of the implemented function to a different area in the device is the preferred choice [166]. To cope with both transient and permanent faults, a combined recovery scheme was integrated in the architecture presented in Chapter A. The contribution it makes is to use a reconfiguration engine to rewrite on-demand, as opposed to blind scrubbing, the affected block to save reaction time, and relocate that block when permanent faults have been diagnosed, while concurrently providing detection of errors.

In the fugacious faults area, recovery is left for a higher level of decision. The contribution of the presented work (see Chapter C) is to deliver new information on detection and diagnosis to the higher level management block. With that valuable input at hand, an adequate decision can be made on the recovery actions to take in order to keep the dependability properties of the system within the expected limits.

5.5 Summary

In this chapter the different fault tolerance aspects have been treated, from error and fault detection to handling, diagnosis and recovery. Along the way different well established techniques have been commented, with their known limitations. A set of advances in those areas, performed in the framework of this thesis, have been presented to tackle problems related to new technologies and applications.

The first effort was devoted to detect and recover from faults occurred proximate in time in a design, in which recovery from first fault had not been completed

when a second is detected. Those scenarios were simulated and checked to behave adequately by using a control FSM and specific partitioning of modified replicas, which in turn could be rewritten or relocated as needed. The novelty was to consider an new fault model: faults with proximate manifestations, to design an architecture which provides a better coverage of those scenarios.

A second effort targeted the other presented fault model of fugacious faults, which are expected to grow in number in modern technology nodes. As a contribution, a detection and further diagnosis architecture was unveiled, together with a methodology to implement it in combinational stages. This novel architecture provides a superior capability to capture those faults, as compared with what would be obtained with prior techniques. The information is delivered to a higher level FT manager which must take decisions accordingly.

Chapter 6

Discussion and Conclusions

In this chapter a global conclusion of the results given in this dissertation is presented. The chapter has been structured in three sections. First, the main findings of this thesis are summarized with the lessons learned and contributions to the field. Limitations of the methods presented in the previous sections are highlighted. Secondly, the main conclusions of this work are presented and it is summarized how they satisfy the objectives previously established. Finally, a third section is dedicated to future research lines.

6.1 Discussion

The 3 main problems motivating this work, as explained in the introduction, were the challenges posed by new complex fabrication technologies, the use of commercial products in critical applications and the ever increasing requirements of verification and validation mandated by standards. In light of the results presented in this thesis, a summary of the main contributions towards those problems in their research context can be explained as follows.

6.1.1 Fault models

After analysis of the literature and study on the current technology trends, a dissection of the different faults affecting deep sub-micron integrated circuits was performed. Practical measurements in the most recent devices compel researchers to face new reliability challenges, not only in those products deployed in critical markets but also in consumer grade products. In that context, a part of this thesis has explored the introduction of new fault models tailored to physical phenomena occurring in embedded devices.

Classical models divided in transient and permanent fault models were taken as the foundation for a new *proximate manifestation* fault model. Indeed, it refers to the phenomenon which takes place when a fault is affecting the system or it is undergoing recovery and another fault reaches or activates in the same operational context. The model refers to a problem of different nature than single faults or multiple faults as described in existing literature, since techniques well suited to the former may not be able to cope with the introduced fault model. Though it applies to both ASICs and FPGAs, impact is not the same in each of them. In the case of ASICs, fault tolerance strategies might be defeated and incorrect service provided for some time, whereas in (SRAM based) FPGAs the use of reconfiguration or relocation as tolerance strategies can turn the situation into a more frequent scenario, provided the time taken to partially reconfigure or relocate can be high, where global reset or total reconfiguration would be required. With the new model, currently deployed or newly designed fault detection and tolerance structures can finally be tested to check whether the response is satisfactory in terms of dependability or not.

A second model was introduced in the context of the thesis. After observation by the research community of the reduction in the duration of faults, together with a reduction of the maximum operating frequency in circuits due to power issues, a new scenario was to be considered. In this case, faults would last appreciably less than a clock period. The situation is distinct from a classic one where temporal filtering has not such high impact. A new set of fault models was introduced to the community: the *fugacious faults* models, with transient fugacious, intermittent fugacious and non-fugacious (or permanent-like) variants. With these models, it is possible to evaluate the suitability of fault tolerance techniques to properly deal with the new challenging scenarios previously presented. When compared to traditional single event transients (SET), the concept of fugacious fault encompasses faults of different nature and origin, which may well play the role of early indicator of potential reliability degradation. Additionally, other situations can be reflected in an increase of fugacious faults. It is known operational environment can change widely in the life of embedded systems, eg. traverse harsh areas with increased radiation levels, abnormal temperature, power instability etc... Those situations would mean an increase in the fugacious faults rate at first, and probably classic fault conditions later on. By considering fugacious faults in the dependability

strategy for the designs, a new tool comes at hand to gather information from the environment and how it can affect the circuits.

6.1.2 Fault injections

For the analysis of robustness of designs, fault detectors or FT architectures, the process of injection is mandatory. In the STF group, a preexisting tool named VFIT (VHDL Fault Injection Tool) provided some valuable experience, but strong limitations and poor performance invited to the development of a new flexible powerful tool. Taking that previous experience as a basis, the FALLES tool developed in this thesis delivers strong parallelization, performance optimizations and total flexibility to apply a wide range of fault models to HDL designs. Furthermore, it can analyze and extract useful information for dependability assessment in a quick and convenient way. Compared to other tools in the field, it is focused in comprehensive simulations, whereas others provide reduced detail or use emulation which does not allow the same degree of observability and controllability. With FALLES, the capacity to employ from compact multi-core shared memory machines to highly parallelized grid computers as computing base enables the simulation of real world problems within a manageable time frame for both single workstation and enterprise frameworks. With single workstation version simplicity and speed can be attained, for data movements between cluster and local machine are not required. It is therefore adequate for small and mid sized campaigns. With the grid version, big campaigns can benefit of an order of magnitude increase in speed, together with an architecture designed to adapt to different clusters as needed with minimal modifications of the code. This comes at the cost of the need of moving data to/from the cluster and a more complex management of the parallel processing.

During development, a series of lessons have been learned. First, memory management is important in simulation structures, so making the best use of it can provide a greater capability for bigger circuits. In fact, a bug in the simulator caused memory leaks only solved by distributing the task in extra parallel processes than the originally thought. Second, partitioning simulations must be done with a speed criterion, in order to accelerate potentially long campaigns. Third, a resuming capability is badly required to avoid costly re-simulations due to power losses, etc. Fourth, saved information must be trimmed to the minimum possible, otherwise it becomes unmanageable, for instance by only saving the significant bits of a word for later analysis. Fifth, for the analysis phase any optimization causes valuable time savings, but when properly debugged it takes much longer to simulate than to analyze the results. Sixth, partitioning the results helps tremendously to find and organize the experiments. Seventh, making room for adaptation to different hardware computing platform eases the deployability, what avoids extra problems when working with data in sparse machines. Finally, for a tool to be usable by many, it must be fully documented and the code easy enough to read and

understand. Therefore, complete documentation and the full code can be found in the public repository reachable at `https://bitbucket.org/jaiesgar/falles`.

6.1.3 Dependability assessment

In the dependability assessment area the thesis has contributed in a couple of topics.

First, the characterization of permanent faults propagation through the pipeline of a processor used in critical applications delivered information on how to inject in order to maximize representativity and reduce the cost of injection. Besides, knowledge was gained on the implications of injecting solely architectural registers, leaving aside the non-architectural or implementation-linked ones. For establishing an accurate safety integrity level, it was made clear an architectural injection is not sufficient to provide accurate reliable results.

Second, multilevel correlation of injection results has set a path to robustness verification at early stages of design, since a stronger confidence can be placed in the higher level injection. To do so, previous RTL injection was required for the architecture at a characterization stage but, once this has been performed, injection at the Instruction Set Simulator can be carried out for new applications employing previously gained information in a fast way. It can turn a very convenient method to perform early checks at a very reduced cost. However it was only checked to be true for permanent fault models, with transient fault models left for further work. In any case correlation for permanents was found, what is a good basis to think with some more research a better accuracy will be eventually reached, so as to provide rough figures at early stages of design or be able to compare different solutions in terms of dependability.

6.1.4 Fault Tolerance mechanisms

In terms of FT mechanisms, the proposals of the dissertation are focused to those fault models introduced in it. In the field of multiple faults with proximate manifestations, an architecture focused towards fault mitigation in the case of FPGAs has been presented. The assumptions were the utilization of commercial-grade SRAM based FPGAs in harsh environments, where fault rates can be certainly high. The proposed architecture *TMR-MDR* is capable of performing detection and diagnosis of multiple faults with proximate manifestations, by combining spatial and temporal redundancy. This maximizes reliability of the system since a lower probability of failure is achieved, and availability since accurate diagnosis allows to take relevant action avoiding extra hassle, as compared to other existing techniques. However, limitations apply in rare fault combinations where the system might still be defeated. Additional limiting factors are those blocks that

use hardwired elements of the FPGA which are not relocatable at will. The introduction of those complicates routing of the relocated blocks, with the added challenge of maintaining the system under timing specifications. In any case, the proposal will become a good solution for several building blocks of today's designs, and a starting point towards availability of mechanisms well suited to any type of structure.

Moving on to fugacious fault models, a tailored architecture for their detection and diagnosis was presented, together with a deployment workflow. The case study was applied to a 4 bit adder where injections showed the expected detection capabilities up to a limit imposed by the technology itself. Correct diagnosis was also demonstrated with the same technological limitations as the detection. An important fact of such a strategy is the enlargement or stretching of the observation period for detection of faults in the bus. Of course the longer the observation period can be made, the higher probability to detect and correctly diagnose a fugacious fault. What is important to mention is that the previous parameter depends on the implementation method and also on the target circuit to be monitored. Considering limitations, the advantage of such approach is that it can detect many more fugacious faults that previous architectures at a cost relatively high for smaller circuits, but with a sub-linear growth with circuit size. Besides, monitors can be strategically placed throughout a design to gather supervisory information, so their area cost is diluted among the covered region. With the monitoring information obtained by means of the presented structure, a higher level FT manager can take informed reaction to tackle any challenging situation to the dependability. Indeed, when a high-radiation area is approached in a satellite, when a high electromagnetic field area is entered in a vehicle, when power source fluctuations start to grow above normal operation and many other potentially challenging situations take place, early reaction can make a difference to maintain safety levels. With that goal, the FT manager could be a reconfiguration engine which, depending on the design and platform, increases the number of replicas in a fault-tolerant sphere of replication, applies time redundancy to recalculate results and compare with previous iterations, programs a watchdog which resets partially or totally the device after some threshold has been surpassed, sets an alarm which issues a warning to the user application, etc... Establishing an adequate faults threshold for different mission profiles is another research topic to investigate, depending on the type of reaction to take. What is clear is that multi-level fault tolerance structures which share information and collaborate –from the hardware level up to applications– will become much more effective mechanisms to produce optimum response to any challenge, improving availability, reliability, maintainability and safety alike.

6.1.5 Fault tolerance implementation

Besides the previous architectures, well known FT structures can be a burden to apply, or what is more dangerous can be applied incorrectly. If implementation is carried out by non-experts, or if errors are accidentally introduced the final result could present worse dependability properties than original structures. To reduce or minimize implementation mistakes an automated system was devised. The novelty was the use of pragmas in the HDL code to select and configure the metaprogram to be used, which is chosen from a preexisting library implementing different FT structures and applied by means of code transformations through an open compilation process. That process converts a code in a language into another code in the same or a different language. In this case the transformation goes from VHDL into a different VHDL code. A clear benefit for its use is the capability to completely transform or recreate the source code, enabling virtually every possible change to be applied.

As an example, a new metaprogram was developed to apply CRC protection scheme to a bus, where easy space exploration of structures allowed to select the optimal polynomial and configuration for each application. With such fast error-free technique to introduce FT into a circuit, design phase for FT is completely transformed. Previously a strategy had to be chosen, implemented and tested to validate its performance. When it was under specification, a new strategy had to be reimplemented with associated time and financial loss. With automation, the paradigm changes as several different implementations can be applied fast, which could even work in parallel to provide functional diversity to a circuit. In the CRC example, different blocks with hard-coded polynomials can be generated as quickly and conveniently as it could be done in software. Other classical metaprograms benefit from open compilation. In the encoder/decoder, different hardware codifications can be applied at will. In time redundancy, different hardware configurations allow for various levels of re-execution. Finally, additional FT strategies can be quickly introduced thanks to the open compilation interface.

6.2 Conclusion

Embedded systems are gaining terrain in today's world, as the number of smart systems grows exponentially. Consulting firms like IDC reveal an expected turnover of $1 trillion in 2019, with important increase in the manufacturing, transportation, smart homes and energy segments. Only year 2014 ended with a record revenue of $755 billion, but more importantly it unveiled the outrageous impact in human lifestyle they are going to have: Advanced Driver Assistance Systems (ADAS), electric motors control systems, all-connected vehicles and smart mass transportation systems in transportation sector, smart wearables, home appliances or intelligent lighting devices in consumer sector and wearable health monitors

and diagnosis devices in healthcare. All those areas will be heavily dependent on embedded systems, their continuity of service central to avoid catastrophic consequences in many cases, and severe inconveniences in others.

Reliability of embedded systems has received a good deal of attention up to date, and will require a much higher degree in forthcoming years. In fact, when failures to provide service take place in traditionally critical markets no doubts remain with respect to the negative consequences to life, business or mission it entails. However, the same event applied to consumer products does not raise so many concerns generally speaking. Beyond commercial damage to the brand supporting the product, a fully connected world in the Internet of Things (IoT) will link in a myriad of ways the devices around us, with potentially unexpected consequences. Furthermore, though advised not to, people will gradually build confidence in products which could not be designed with reliability as a primary concern.

Adding up to the presented issues, the need for extreme performance boost to support for those rich features, and the strict power consumption policies to provide an affordable $performance \times watt$ ratio, place higher pressure in the specifications side. To cope with it, fabrication technology has to evolve at a hectic pace. With every few technology nodes, a new set of dependability challenges appears, some are tackled and disappear, and most well known simply worsen. That is the reason why valid fault detection, mitigation, recovery, injection or analysis techniques need to be revisited, updated, upgraded or completed with up-to-date in-field gathered data.

The objectives in this thesis were focused to that end. In the next paragraphs they are summarized.

Objective: *Studying the current and upcoming faults affecting embedded systems to understand their origin, evolution and consequences and provide new representative models whenever required.*

After extensive study of literature and analysis of usual fault models, and cross-checking with technology reports and studies, a new set of fault models has been proposed. Faults with proximate manifestations model situations that are bound to happen at up-to-date devices with increased fault rates, while undergoing recovery. That is specially relevant for SRAM based FPGAs. An additional set of proposed fault models has been the fugacious faults, which are differentiated as transient fugacious, intermittent fugacious or non-fugacious variations. With the introduction of the latest, attention to forecasting and early detection of faults is given the deserved importance, provided the fact that embedded systems turn more and more adaptive over time.

Objective: *Propose new fault tolerance mechanisms better suited to the current technology and application trends*

For the most recent faults which show proximate manifestations, a tolerance architecture was devised, which mitigated faults and launched recovery, based on temporal and spatial redundancy. The proposed architecture was focused towards SRAM FPGAs, and it was demonstrated to improve availability and reliability of existing solutions under the proximate manifestations fault models.

Changing to fugacious fault models, a detection and diagnosis architecture was proposed, which exploited the idea of enlargement of the observation (stability of the bus) period. The architecture and workflow for deployment were presented and some testing demonstrated the feasibility of the approach. Though cost was still somehow high, it is expected to diminish as the size of the target circuit increases.

As fault tolerance mechanisms are more and more required not only to cover new fault paradigms but also classical ones, it is more interesting to employ automated deployment. This avoids implementation mistakes which would ruin the increase of reliability. In this area an automated deployment methodology of non-functional features was presented along with an example of fully configurable CRC deployment.

Objective: *Provide a fault injection and analysis tool capable of exploiting the current computing state of the art and focused on efficiency*

To fulfill this objective the tool **F**ault injection and **A**nalysis for **L**ow **L**evel **E**valuation **S**uite (FALLES) was developed. Multi-core and multi-node capabilities together with powerful analysis tools and flexibility are its main strengths. A complete set of fault models has been implemented to allow for different type of injections, where all the information is stored in plain text to enable custom trace analysis. The tool is well suited for both research and production environments.

Objective: *Develop a dependability assessment methodology which can cut development time for the industry*

For the optimization of dependability assessment procedure, a strong confidence must be built in the tests carried on to ensure reliability, safety and other parameters of certification standards. The work presented in this thesis means a step forward in the quest for providing trustworthy methods of testing for compliance which are performed at earlier, more convenient stages in the design flow. While this still needs to be validated along different architectures and implementations, the obtained results are promising for the possibility to have accurate enough assessment methodologies in those early descriptions, thus saving time and costs.

6.3 Future work

As future work there are several topics to continue working on, each one of those treated in the thesis.

First, in the fault injection tool FALLES, a good bunch of improvements and additional functions could be implemented. To name a few, a system to select the type of coding for enumerated types in VHDL would save time in preparation of design, a parser of EDIF format for instance could help distinguish sequential from combinational elements, in order to apply proper fault models to each element. New additional models could also be added, including delay or short (by adding a new *resolve* function).

Second, in the multi-level correlation further steps can continue the study of the relationship between injections at different levels. For instance by applying additional fault models for the correlation, or by studying the incidence of multi bit upsets or multiple register upsets. Other studies could perform a cross-architecture study to investigate to which extent the results for an architecture remain valid throughout other existing architectures in the market. Likewise, the same can be done for different implementations of the same architecture.

Third, in the architectures of tolerance for proximate manifestations, new state machine definitions and codifications could increase robustness and/or reduce area penalty. Further testing would also be interesting to check additional potentially hazardous situations.

Fourth, in the detection and diagnosis of fugacious faults, a natural step is to scale the solution to bigger blocks, analyzing the cost increase vs size increase of the target. Additionally, a finer delay insertion could be investigated to achieve better stability window enlargements. That would help overcome the process variation spread, as well.

Fifth and finally, referring to open compilation for the addition non-functional features, newer metaprograms could be implemented to provide additional functions which have not been yet automated. Additional support for Verilog input would also provide a wider possible application.

Chapter 7

Summary of contributions

In this chapter the full set of contributions in the framework of this thesis is gathered in short form, as a quick reference guide.

7.1 Publications

This thesis is supported by the publication of most of its content in reputed international conferences, a national conference ,a book chapter, and an article in review process of a high impact international journal.

7.1.1 Conferences

- **Robust communications using automatic deployment of a CRC-generation technique in IP-blocks** [50], which is published in the *Jornadas de Computación Reconfigurable y Aplicaciones, 2011.* This paper demonstrated an application of open compilation technology to dependability area by creating a tailored CRC-code infrastructure for transmissions.
- **Tolerating multiple faults with proximate manifestations in FPGA-based critical designs for harsh environments** [51], which is published

in the *Conference on Field Programmable Logic and Applications, 2012*. This work focused in providing coverage for faults occurred while the system is undergoing a recovery, avoiding failures and maximizing availability. It was demonstrated in a test case.

- **The Challenge of Detection and Diagnosis of Fugacious Hardware Faults in VLSI Designs** [56], which appeared in *European Workshop on Dependable Computing, 2013*. This publication modeled a new type of fault, the fugacious fault, and analyzed and evaluated the different challenges and difficulties to overcome in order to detect and diagnose such faults. Additionally it provided the outline of a methodology to achieve those goals.

- **Analysis and RTL Correlation of Instruction Set Simulators for Automotive Microcontroller Robustness Verification** [53], which is published in *Design Automation Conference, 2015*. This paper demonstrates a correlation between robustness data extracted after RTL injection and information available at the instruction level.

- **Characterizing fault propagation in safety-critical processor designs** [54], which is published in *International On Line Test Symposium, 2015*. In this paper, an in-depth analysis on how faults propagate through the pipeline of a safety-critical processor is performed. An idea on how much information is lost by injecting only at the architectural level is derived from the results obtained.

- **Increasing the Dependability of VLSI Systems Through Early Detection of Fugacious Faults** [52], which is published in *European Dependable Computing Conference, 2015*. This manuscript provided an architecture and implementation flow towards early detection of fugacious faults, which were modeled in previous work. The proof of concept was applied in an arithmetic circuit.

- **Towards Certification-aware Fault Injection Methodologies Using Virtual Prototypes** [57], which is published in *Forum on Specification and Design Languages, 2015*. This work suggested methods to effectively employ virtual prototypes for certification process. In it the FALLES tool was presented and explained thoroughly.

7.1.2 Journals

- **On the potentials of Robustness Verification using Architectural Registers-based Fault Injection.** [55], which is sent for publication in *IEEE Transactions on Computer-Aided Design*. This paper expands the concepts of correlation between injections at RTL level and effects at the microarchitectural level. It first shows propagation characteristics of faults. Later, it introduces the utilization factor of different processor resources to establish the potentiality to obtain accurate dependability information by injection at the microarchitectural level.

7.1.3 Book chapters

- **An Aspect-Oriented Approach to Hardware Fault Tolerance for Embedded Systems** [9], Publised in *Handbook of Research on Embedded Systems Design* by IGI Global Publishers. In the chapter full detail in the aspect oriented compilation process for non-functional features provision is explained.

7.2 Framework of the Dissertation

The present dissertation was developed in the Fault Tolerant Systems Group at the ITACA research institute (STF-ITACA), a part of *Universitat Politècnica de València*. The group, leaded by Prof. Pedro Gil, has long experience in fault injection, assessment and protection of critical systems.

7.2.1 Research projects

In the course of the thesis, there has been financial support from several public research projects:

- Ministerio de Ciencia e Innovación.

  Sistemas Empotrados seguros y confiables basados en componentes (SEMSE-CAP).

  From Jan. 1st, 2010 until Dec. 31st, 2012.

  Budget: 205.821,01 €.

- Ministerio de Economía y Competitividad

  Adaptive and resilient networked embedded systems (ARENES).

From Jan. 1st, 2013 until Dec. 1st, 2015.

Budget: 31.730,40 €.

- European Union and Ministerio de Economía y Competitividad

  Verification and Testing to Support Functional Safety Standards (VeTeSS).

  From May. 1st, 2012 until April. 31st, 2015.

  EU Budget: 3.06 M €. National Budget: 6.08 M €.
  Eligible Costs: 18.34 M €.

7.2.2 International research stays

The PhD. candidate has had the chance to work for 3 months in the University of Edinburgh, at the System Level Integration Group in 2013. While there Prof. Tughrul Arslan was his supervisor. The group is specialist in dynamic partial reconfiguration in FPGAs for reliability in real-time systems, what provided an invaluable academic experience. Important knowledge on the requirements and behavior of reconfiguration engines was gained while in there.

7.2.3 Collaborations

In 2014 after being contacted by Dr. Hernandez from the CAOS group, a 3 month internship was carried out by the candidate at Barcelona Supercomputing Center - Centro Nacional de Supercomputación (BSC-CNS). The goal was to provide knowledge in fault injection and assessment for a European FP7 Research Project called *VeTeSS*. From this collaboration a paper [53] was published in the world's most important conference in its field. The internship received support from HiPEAC association.

Later on, the BSC-CNS recruited the candidate for 4 additional months to continue research for VeTeSS (Verification and Testing to support functional Safety Standards) project. From this period other 2 publications were developed [54, 57]. A journal article was started as well.

7.3 Awards

After the publication in DAC 2015 of a paper of relevance to the HiPEAC community, a *HiPEAC Paper Award* was granted to the authors. The award distinguishes European contributions to very important international conferences. A reference can be found in:

`https://www.hipeac.net/research/paper-awards/2015/`

Appendices

Appendix A

Tolerating multiple faults with proximate manifestations in FPGA-based critical designs for harsh environments

Authors: Jaime Espinosa, David de Andrés, Juan Carlos Ruiz and Pedro Gil

---

Field-Programmable Gate Arrays (FPGA) have proven their value over time as final implementation targets. Their singular architecture renders them sensitive to a wide range of faults, specially to those causing multiple and non-simultaneous errors, that can result in silent data

corruption and also in structural changes in the hardware implementation. This papers presents and tests an approach to enable the confident use of conventional (low-cost) FPGAs in hostile environments. The design combines spatial and temporal redundancy with partial dynamic reconfiguration to increase the resilience of designs. The goal is to tolerate the occurrence of single and multiple faults, even during the reconfiguration process of FPGAs, while minimizing the impact of the recovery process on system availability. Fault injection techniques are used to experimentally evaluate various features of the approach. Results are very promising and lead us to state that, although many research is still required, the old idea of self-repairing HW designs is closer today.

A.1 Introduction

Recently FPGAs have been moving from the prototyping arena to the core of final products. The required evolvability of products has been well covered by low-cost Static RAM (SRAM) models, which add the chance to reconfigure their functionality while running. On top of that, higher speeds and lower power consumption, together with massive logic densities, foster the interest to expand their usage to critical systems in harsh environments, where currently expensive anti-fuse or rad-hard products are the only alternatives.

However, to achieve such powerful features SRAM FPGAs suffer higher fault proneness [36]. This fact is further aggravated in environments where extreme operational conditions may lead to the occurrence of multiple faults, whose related errors could manifest in so proximate instants of time that previous faults have not been cleared from the system [13]. To our best knowledge, no specific research has been conducted focusing on the impact of those combined effects on the system reliability. It is well-accepted today that multiple errors rarely manifest in the system at the same time, even when sharing a common fault origin. This context legitimises the following questions: How do today's FPGA-based fault tolerant (FT) designs cope with proximate manifestations of multiple faults? and, which are the implications of a new fault manifestation while undergoing recovery of a previous one? how to improve the resilience of conventional FT designs in these situations? and at which cost? These are the basic questions addressed in this paper.

The contribution is to attain an acceptable level of reliability to adopt the use of low-cost FPGAs in scenarios demanding high dependability. Detection, masking whenever possible and recovery triggering are the deployed functions. The introduced penalty in terms of area and time period is also quantified to evaluate the suitability of the approach.

The rest of the paper is structured as follows. Section A.2 presents basics on faults in SRAM FPGAs, and Section A.3 deals with existing approaches for FT in FPGAs. The proposed architecture to handle the occurrence of a set of representative faults, overcoming limitations derived from existing techniques, is detailed in Section A.4. A case study is used in Section A.5 to show the feasibility and usefulness of the proposed approach. Finally, Section A.6 presents and discusses the obtained results, and Section A.7 concludes the paper.

A.2 Faults in SRAM FPGAs

Attending to modern SRAM-FPGAs architecture, faults can happen either in the reconfigurable fabric or in the configuration memory (*fabric* and *CMEM*, from now on). Faults targeting the fabric could be assimilated to those typically affecting other VLSI systems, but the faults targeting the CMEM have further effects than those affecting common SRAM memories, as they will change the underlying hardware implementation of the considered design. For instance, when considering faults targeting the storage elements of FPGAs, FFs holding the design state only account for 0.42% of the total number of memory bits, whereas the remaining 99.58% is taken by the CMEM [88]. Hence, dealing with faults targeting the CMEM is much more relevant for improving the final resilience of the system. What is more, according to [70], faults targeting the CMEM affect the underlying fabric in the following proportion: 63.3% target multiplexers selection, 15% change the routing of the system, 10.6% affect control bits, 6.6% modify LUT's contents, 3.26% affect buffers, and just 1.3% remain unclassified. As can be seen, faults targeting the CMEM will mainly affect the routing and combinational functions of the implemented design as routing elements and combinational logic dominate the available area of modern FPGAs. Studying their behaviour in the presence of faults is essential for the dependable use of FPGAs in harsh environments.

Table A.1: Considered single fault models

Target	Duration	Manifestation on fabric
Combinational logic (fabric)	Transient	Pulse, indetermination, and delay
	Permanent	Stuck-at, stuck-open, indetermination, delay, short, open, and bridging
Sequential logic (CMEM & fabric)	Transient	*Transient* stuck-at, bit-flip, indetermination, delay, short, and open
	Permanent	*Permanent* stuck-at, indetermination, delay, short, and open

Table A.2: Considered multiple fault models

1st fault		2nd fault	
Duration	Target	Duration	Target
Transient	Comb. (fabric)	Transient	Comb. (fabric)
Transient	Comb. (fabric)	Transient	CMEM [1]
Transient	Comb. (fabric)	Permanent	Comb. (fabric) or CMEM
Transient	CMEM [1]	Transient	Comb. (fabric)
Permanent	Comb. (fabric) or CMEM	Transient	Comb. (fabric)
Transient	CMEM [1]	Transient	CMEM [1]
Permanent	Comb. (fabric) or CMEM	Permanent	Comb. (fabric) or CMEM

[1] Transient faults in CMEM manifest as permanent ones in design logic (fabric) and can be assimilated to them from the logic point of view.

Faults of interest in this work include both transient and permanent ones. Transient faults, due to cosmic radiation or capacitive coupling, for instance [101], temporarily alter the voltage value of a certain element, which may recover its right value after some time. Conversely, permanent faults relate to irreversible physical defects due to the manufacturing process (e.g. contamination of silicon or incorrect metallisation) or wear-out of the device (electromigration [3] or hot carrier injection [162], for instance). Recent studies even consider intermittent faults as a new type of fault modelling system malfunctions activated by environmental conditions. Those due to wear-out typically derive in permanent faults while those directly induced by operational conditions can be considered as transients [36]. This is why we limit the purpose of our study to transient and permanent faults. Table A.1 provides a comprehensive list of single transient and permanent fault models considered as representative in dependability studies [66]. However, recent evidences show that considering only single faults could be incomplete.

In modern devices, manufacturers provide accelerated life measurements [82], which show permanent fault rates (hard errors) of approximately 20 FIT (failures in time = $10^9 hours$) during the expected life of the device under normal conditions. While this might sound low, it has been demonstrated that under extreme conditions of temperature, voltage and use the increased rate can yield a lifespan as low as 1 year [163]. Adding to it the rates of transient faults caused by neutrons and alpha particles in the CMEM for current technology (105 $FIT/Mbit$ as published for ground level), we justify the importance of developing mechanisms to mitigate the effects of faults in both the CMEM and the fabric of FPGAs as they appear. Beside the possibility of coincidence, other phenomena can lead to multiple proximate manifestations. A fault which remains latent (inactive) for a period of time can later activate causing the mentioned situation. Also, a single harmful event can affect multiple signals [164] which are desynchronised at the output. Thus, for the sake of completeness, we also consider in our study that (i) several single faults can affect the system in proximate instants of time, and (ii)

one single fault may lead to several manifestations that can also eventually affect the system is proximate instants of time. Although differing in their causes, both situations lead to similar consequences that will be modelled for the purpose of this paper as listed in Table A.2. In this table, we can see that a multiple fault is limited to the occurrence of two faults. Though no field data on the frequency of impact is currently available, measurements show it grows dramatically in harsh environments [133].

A.3 Fault tolerance for FPGA-based designs

The kind of fault tolerance mechanisms deployed for SRAM FPGA-based designs pursue two different goals: i) detecting and masking the occurrence of faults, and ii) correcting and recovering from their effects, which may be transient or permanent.

In the first category, the most extended space redundancy approach is the popular *Triple Modular Redundancy* (TMR). It is capable to mask any faults affecting the output of only one replica at any time. Thus, complementary mechanisms are required to avoid faults build-up affecting other replicas. It is also the one requiring more area, but as an advantage it is a generic approach. Among others, an alternative method that exchanges area for performance is *Time Redundancy* [120]. It can only mask transient faults in fabric. Detection of permanent faults may be achieved by re-executing the operation with re-encoded operands but strong performance penalties are always applied and second fault may defeat it. A proposed combination of modular redundancy (duplication) with time redundancy [93] allows to reduce area and apply performance penalty only when needed. As a drawback, it cannot tackle the occurrence of multiple faults.

As far as the recovery of transient faults is concerned, it is worth nothing that though transient faults in fabric usually disappear after a short period of time, this is not the case of those targeting the CMEM of the FPGA, which remain until the affected area of CMEM is amended. Thus, *scrubbing* techniques [19] traverse the whole CMEM periodically refreshing its contents with the original configuration for the currently uploaded design. Accordingly, faults may remain for a whole scrubbing cycle in the worst case. This time could be reduced by using the technique of directly *rewriting* the affected location whenever detection mechanisms provide it. The capability of partial dynamic reconfiguration (PDR) of the FPGA is used to avoid stopping the system while immediately rewriting affected areas [20, 166]. Minimisation of multiple faults incidence is achieved by accelerating recovery in this way.

For the recovery of permanent faults, none of the presented mechanisms will be successful, being *relocation* the only viable option. This approach takes benefit

of the reconfigurable nature of FPGAs to move the affected component from the faulty area to a fault-free one. This is accomplished by rewriting the CMEM with a new configuration file reflecting the new distribution of the design on the device. Existing techniques [28] reserve internal spare resources and, upon the occurrence of a fault, move the faulty element onto an spare. So, faults may be tolerated as long as spare elements are available. Pre-compiled approaches are quick to reconfigure, but they require large FLASH memories to store configurations. Conversely computing a suitable new configuration on-the-fly may greatly affect the availability of the system. Nevertheless this approach is very flexible and a small FLASH memory should be enough.

Altogether, just on their own, reconfiguration approaches are useless. They require the combination of the previously presented detection and masking mechanisms to determine the location affected by the fault, mask it when possible and pass this information to the reconfiguration mechanisms in order to correct the problem, while providing multiple fault tolerance. Those are therefore the duties of our proposed approach.

A.4 A multiple fault tolerance approach

The proposed approach faces single and multiple proximate faults in FPGA-based designs while reducing the downtime of the system. From now on, we will refer to it as *TMR-MDR* for *TMR-Module Discard and Repair approach*. Next sections explain in detail the architecture.

A.4.1 Global architecture

An overview of the proposed TMR-MDR architecture is depicted in Figure A.1. It consists of two main hardware components: a regular FPGA with PDR capabilities, and a highly robust FLASH memory to store the different configurations to programme the FPGA with.

The design implemented in the FPGA includes space redundancy (TMR) to mask the occurrence of any kind of single fault in the system, either at CMEM or fabric level. The on-line fault detector (OFD) collects information provided by the design logic and manages correction signals. Its modus operandi is simple: the first fault detected is considered a transient and so proper output is selected on the fly. A repeated error in the same location is acknowledged as permanent, and will trigger the OFD notification for the reconfiguration circuit to restore the related portion of CMEM. If the problem persists, relocation will be applied to heal from hard errors. In case that a second fault occurs during the treatment of the first one, the OFD will issue a stall period to freeze the system while the situation prevents the system from obtaining a correct output.

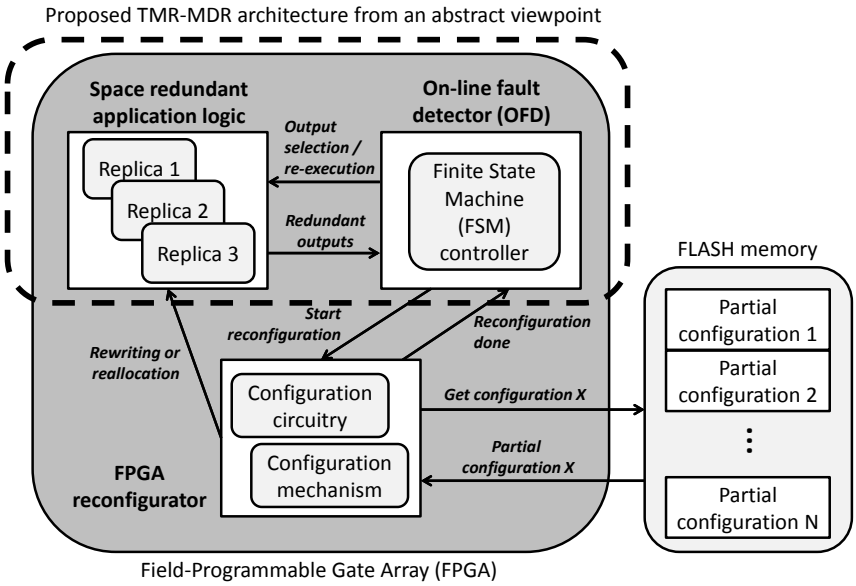


Figure A.1: Global architecture of the proposed fault masking and correction mechanism.

Reconfiguration employs built-in dedicated hardware in the FPGA [42]. To achieve a fast reconfiguration time, pre-compiled configurations will be employed. The number of configuration files required to deal with a set of m operative modules and k spare locations is $N = (m + k)(m + 1)$ plus an additional configuration file for the static part of the design. This large number could be reduced following an approach based on generic modules whose configuration file could be preprocessed prior to placement [114]. As it may slow down the reconfiguration process, care should be taken to the trade off between storage and speed. Next section describes the structure of the protected design logic and OFD.

A.4.2 Detailed description

Due to the nature of the set of faults considered representative for FPGA-based designs, this architecture is focused on protecting the internal routing and combinational logic of the implemented design. After the previous study, space redundancy appears as the most suitable approach to mask the effect of faults affecting these components. Furthermore, three replicas is the minimum number admissible to prevent the use of timing redundancy and then reduce its impact on performance. Hence, the target component (if not the whole design) will be replicated 3 times. However, instead of a majority voter the outputs of the replicas *R0, R1, R2* will be connected as depicted in Figure A.2 .

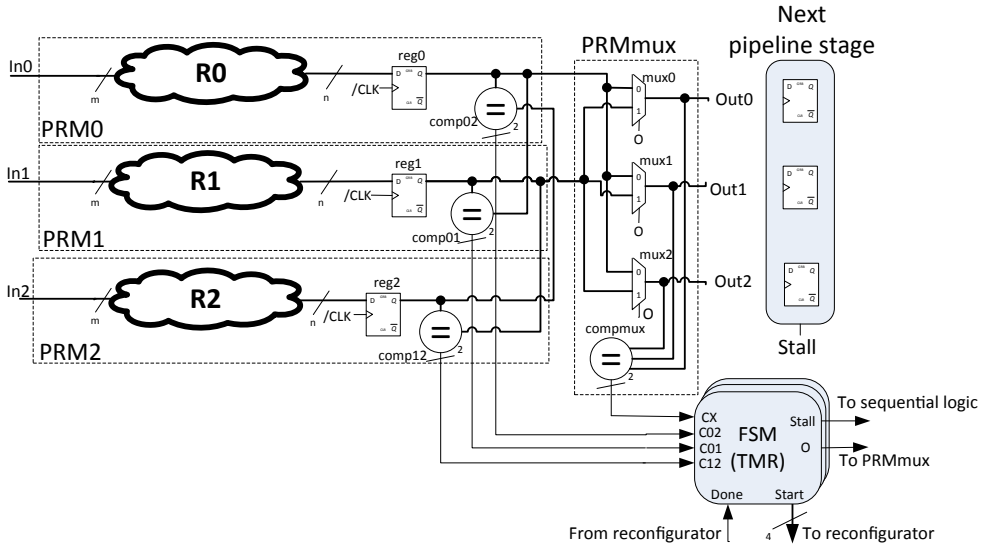


Figure A.2: Detailed architecture of the proposed approach.

First of all, as dealing with combinational logic, and as recommended by PDR flows, to avoid transient delays mismatching when checked intermediate registers *reg0, reg1* and *reg2*, sample data at the falling edge. So first half of the period is used for calculation.

Once the outputs are registered, they are compared in pairs to check whether there is none, one or more faulty outputs. The comparison is performed by dual-rail Totally Self-Checking (TSC) comparators, thus ensuring self-testing and fault secure properties. These comparators, named *comp02*, *comp01*, and *comp12* in Figure A.2, provide their result to a Finite State Machine (*FSM*), which manages detection and correction in the second half of the period. The different cases covered by the result of these comparisons are: i) all 3 comparators agree, so there is no fault, ii) one of them disagrees, thus that comparator is faulty and should be repaired, iii) only 1 comparator agrees, what determines that the unmatched output is erroneous and the related replica is faulty, and iv) all 3 comparators disagree, so a multiple fault has occurred either in the replicas and/or comparators.

As shown in Figure A.2, each replica and its associated register and comparator are integrated into the same Partially Reconfigurable Module (*PRM0*, *PRM1*, and *PRM2*). PRMs are the entities that could be reconfigured, either rewritten or relocated, and so they must contain all the logic that should be reconfigured together. A reconfiguration process affecting this PRMs will cause the system to work on a duplication with comparison (DWC) fashion using the 2 remaining replicas.

After the comparison process, it is necessary to select the right output to be provided to the system output or the next stage of the pipeline (depending on the target element being a whole combinational system or just a combinational part of the system). This selection is performed by the FSM, which controls three different multiplexers (*mux0*, *mux1*, and *mux2*) that will pass the selected output onto the final outputs, which are in turn compared in pairs to determine whether any multiplexer is faulty. This comparison is also performed by another dual-rail TSC comparator (*compmux*). The multiplexers and their comparators constitute another PRM, named *PRMmux*, so any reconfiguration in this module will prevent the system from providing an output and, hence, it should be stalled for the duration of the reconfiguration process.

Finally, the FSM has been also protected under a TMR approach. Its design is detailed in the next section.

A.4.3 Design of the FSM controller

According to the presented set of faults, eleven different situations have been identified deriving from the occurrence of single and two consecutive faults, taken as worst case.

Table A.3 summarizes the considered scenarios. It has been coded using 3-tuples (D, L, E) where i) $D = \{T, P\}$ states the duration type of the fault (P for permanent, and T for transient), ii) $L = \{F, M\}$ determines the location of the fault (F for fabric, and M for CMEM), and iii) $E = \{R, C, X\}$ states the affected element (R for replica, C for comparator, and X for multiplexers). Consecutive faults are separated by the '$\rightarrow$' symbol, and the symbol '$\star$' has been used to refer to any applicable value. Additionally, scenarios which have already been covered by other works have also been highlighted. Situations not comparable among different architectures are marked with the '—' symbol.

The occurrence of simultaneous faults has not been considered as there already exist different techniques that could help alleviating this problem, like special place and route processes [164] or bit scattering. Moreover, multiple faults simultaneously affecting any of the PRMs will manifest at their outputs and will be properly treated as either a transient or permanent one. The different scenarios are further described through illustrative examples.

A. No action: Whenever a transient fault targets the fabric affecting one of the replicas (R_i), proper output is simply selected.

B. Rewriting: The CMEM must be rewritten to remove a transient fault that may target i) the CMEM affecting any element of a PRM_i (R_i or $comp_i$), or ii) the fabric affecting a comparator ($comp_i$). The system could still provide the correct output during the reconfiguration process in a DWC fashion.

C. Rewriting and one stall cycle: In case of a transient fault in fabric affecting a given replica (R_i) and a second transient fault targeting either its comparator ($comp_i$) or another replica (R_j), the CMEM must be rewritten and a stall cycle should be issued to allow one transient to disappear. The occurrence of both faults in consecutive clock cycles causes the first fault to be treated as a transient fault in CMEM, thus requiring a rewriting. Furthermore, as both faults may simultaneously generate a wrong output, the stall signal is required.

D. Rewriting and multiple stall cycles: The occurrence of a transient fault affecting one the multiplexers (mux_i), will require rewriting the CMEM to restore the fault-free state for $PRMmux$ module. Accordingly, the system will be stalled meanwhile. This fault may either be single or preceded by any transient fault.

E. Relocation: A permanent fault targeting either fabric or CMEM, and affecting either a replica (R_i) or its comparator ($comp_i$) may only be corrected by relocating the PRM_i module into a fault-free area of the FPGA. This fault may either be single or preceded by a transient affecting the same module (R_i or $comp_i$). The system will provide the correct output during the reconfiguration process in DWC.

F. Relocation and one stall cycle: This time first a permanent fault, which targets a replica (R_i) or a comparator ($comp_i$), is followed by a transient one targeting the fabric and affecting either another replica (R_j), in the first case, or any replica or another comparator ($comp_j$) in the second case. As this second fault will shortly disappear, just one stall cycle is needed.

G. Relocation and multiple stall cycles: Any fault manifesting as permanent in $PRMmux$ (transient in CMEM or permanent in either fabric or CMEM) requires the relocation of this module. As in case D, it is necessary to stall the system until process finishes. This case also includes preceding transient faults affecting i) the fabric of any replica, comparator or multiplexer, or ii) the CMEM and also manifesting at the $PRMmux$.

H. Multiple consecutive rewriting and one stall cycle: This is a particular case of consecutive transient faults targeting the fabric and affecting different comparators ($comp_i$ and $comp_j$). Due to the proposed specific architecture, the system may provide the right output as long as the replicas and at least one comparator are fault-free. Accordingly, just 1 stall cycle is applied while providing the right output during rewriting. (PRM_i and PRM_j).

I. Multiple consecutive rewriting and multiple stall cycles: There exist different cases that prevent the system from providing a correct output and would require rewriting the CMEM consecutively. They include, among others, transient faults targeting the CMEM and affecting a replica (R_i) or comparator ($comp_i$), followed by another transient in CMEM and targeting another replica (R_j), the comparator of another module ($comp_j$) or the multiplexers ($PRMmux$), in the

Table A.3: Considered scenarios and combination of faults

Scenario	Single and multiple faults	Already covered in other works
A. No action	(T,F,R)	[26] [167][1][120] [93] [166]
B. Rewriting	(T,F,C)	—
	(T,M,⋆)	[26] [167][1][120] [93] [166]
	(T,F,R_i) → (T,F,R_i)	[26] [167][1][166]
	(T,F,C_i) → (T,F,C_i)	—
	(T,⋆,R_i) → (T,M,R_i)	[26] [167][1][93] [166]
	(T,⋆,C_i) → (T,M,C_i)	—
C. Rewriting and one stall cycle	(T,⋆,R_i) → (T,F,R_j)	[26] [167][1][93][1]
	(T,F,R_i) → (T,F,C_i)	—
	(T,M,C_i) → (T,F,R_i)	—
D. Rewriting and multiple stall cycles	(T,⋆,X)	—
	(T,F,X) → (T,M,X)	—
	(T,F,R_i) → (T,⋆,X)	—
E. Relocation	(P,⋆,{R,C})	[26] [167][1][120] [93] [166]
	(⋆,⋆,R_i) → (P,⋆,R_i)	[26][2][167][1][120][2][93][2][166][2]
	(⋆,⋆,C_i) → (P,⋆,C_i)	—
F. Relocation and one stall cycle	(P,⋆,R_i) → (T,F,R_j)	—
	(P,⋆,C_i) → (T,F,{R_j,C_j})	—
G. Relocation and multiple stall cycles	(P,⋆,X)	—
	(P,⋆,X) → (T,⋆,X)	—
	(P,⋆,X) → (T,F,{R,C})	—
H. Multiple consecutive rewriting and one stall cycle	(T,F,C_i) → (T,F,C_j)	—
I. Multiple consecutive rewriting and multiple stall cycles	(T,⋆,C) → (T,⋆,X)	
	(T,F,R_i) → (T,M,R_j)	[26][3][167][3][120][3][93][3][166][3]
	(T,M,R_i) → (T,M,{R_i,R_j})	[26][3][167][3][120][3][93][3][166][3]
	(T,M,R_i) → (T,⋆,C_j)	—
	(T,M,R) → (T,⋆,X)	—
	(T,⋆,C_i) → (T,M,C_j)	—
J. Multiple consecutive relocations and stall until finished	(P,⋆,R_i) → (P,⋆,{R_j,C_j})	[26][3][167][3][120][3][93][3][166][3]
	(P,⋆,{R,C}) → (P,⋆,X)	—
	(P,⋆,C_i) → (P,⋆,C_j)	—
K. Relocation and rewriting and multiple stall cycles	(T,⋆,R_i) → (P,⋆,{R_j,C_j})	[26][3][167][3][120][3][93][3][166][3]
	(T,F,C_i) → (P,⋆,R_j)	—
	(T,⋆,C_i) → (P,⋆,C_j)	—
	(T,⋆,C) → (P,⋆,X)	—
	(P,⋆,{R,C}) → (T,⋆,X)	—
	(T,M,R) → (P,⋆,X)	—
	(P,⋆,R_i) → (T,M,C_j)	—

[1] Detection only
[2] Mask only
[3] Do not cover faults targeting the same output on different replicas

first case, or another comparator ($comp_j$) or the multiplexers ($PRMmux$) in the second one. Stall cycles will be signalled for the first rewriting process. As faults affecting the multiplexers also impose a stall period, when the second fault targets the multiplexers the first rewriting will be interrupted and faults will be treated in the reverse order, thus reducing the downtime of the system.

J. Multiple consecutive relocations and stall until finished: This scenario covers all those cases that require relocating two different modules in a consecutive way, thus preventing the system from providing a correct output. They include permanent faults, either in fabric or CMEM, affecting a replica (R_i) or comparator ($comp_i$), followed by a similar fault affecting another replica (R_j), another comparator ($comp_j$) or the multiplexers (PRM). The same policy described in I will be applied here whenever the second fault targets the multiplexers, thus minimising the stall period.

K. Relocation and rewriting and multiple stall cycles: This case also groups a large number of possibilities, including permanent faults in fabric affecting a replica (R_i) or a comparator($comp_i$), combined with transient faults in CMEM targeting another replica (R_j), another comparator ($comp_j$) or the multiplexers ($PRMmux$). Like in scenarios I and J, faults in the multiplexers take precedence to minimise the impact on performance.

To cope with all these scenarios, the FSM is implemented as a Mealy machine, otherwise the control signals would be delayed one clock cycle. The FSM inputs are the two-rail results of comparing the outputs of the replicas and the outputs of the selection multiplexers, and the flag activated by the reconfiguration circuitry when process is correctly finished. Functional interrupts could affect the mentioned circuitry preventing a successful reconfiguration, so watchdogs to force a reset or any other recovery technique should be applied in the future. Outputs of the FSM drive the selection multiplexers, the stall signal, and the 4-bit signal pointing to reconfigure $PRMmux$, $PRM2$, $PRM1$ or $PRM0$.

A.4.4 Summary

As can be seen, the proposed architecture covers a wide range of both single an multiple faults. It is capable to provide correct output and, unlike the rest of existing approaches, tolerate the occurrence of one fault while it is under reconfiguration. Furthermore, the number of required stalls has been carefully minimised to reduce the downtime of the system whenever correct outputs cannot be provided.

A.5 Case study

A 32-bit floating point multiplier has been selected as the target design to show the feasibility of the proposed approach, as it makes extensive use of both combinational logic and internal routing, the most abundant elements on an FPGA and precisely those this mechanism has been designed for. To compare with other common techniques, a TMR mechanism enhanced with triple majority voter (eTMR) has also been considered.

Both approaches have been modelled using the VHDL hardware description language. These models have been implemented on a Virtex-6 (XC6VLX240T-1FFG1156) using the tools provided by the manufacturer (ISE). In this way, a twofold study can be conducted. On the one hand, the final implementation on a real FPGA provides information about i) resource utilisation, which relates to the silicon area, and ii) maximum clock frequency or longest combinational path, which estimates throughput. On the other hand, fault injection could be used to determine the behaviour of these systems in the presence of a representative set of faults, both in terms of i) coverage, estimated by means of the percentage of experiments leading to a failure, and ii) temporal intrusion, considering the percentage of experiments leading to single and multiple stalls.

The selected workload consisted of a large set of randomly generated input operands, thus allowing different data paths of the combinational multiplier to be activated.

Fault injection experiments were performed by means of VFIT [63], a VHDL-based Fault Injection Tool. The fault load injected consisted in 6 different configurations covering the models presented in Section A.2. Faults in CMEM were simulated taking into account their effect on the implemented circuit signals. *Delay* faults, however, were not injected as the models of the systems do not include the required timing information. Faults were uniformly distributed among all the possible target signals. Finally to reduce the number of experiments required, the worst possible case was applied for consecutive faults, i.e. they occur within one clock cycle delay between them. The total number of experiments was determined according to Eq. A.1 to achieve statistical confidence on results. P stands for the probability of a single element, for instance a FF or node, to be targeted by a fault (inverse to the number of elements), and Q is the desired level of confidence. Accordingly, in order to obtain a 95% of confidence that all the elements are targeted by at least one fault, 18237 and 23563 experiments were required for the eTMR and the TMR-MDR, respectively.

$$N = \frac{\ln(1-Q)}{\ln(1-P)} \tag{A.1}$$

A.6 Analysis of results

Interesting results are commented next, where the enhanced TMR is taken as comparison reference.

In terms of coverage and as expected, single faults still produce no failures since TMR-MDR is based in TMR. The best improvement comes when a permanent fault is followed by any other type of fault, where eTMR fails 17% of times, whereas TMR-MDR does so in just only 0.09% of the cases. When the first fault is transient a non-negligible improvement leads to just 0.01% of the experiments failing. In global, 6% of all eTMR experiments failed versus 0.03% of TMR-MDR, which clearly shows the great promise offered by this proposal for really harsh environments, rendering an advantage in every case. The small percentage of experiments leading to failure is related to i) two faults targeting different replicas of the control FSM, protected by TMR, and ii) faults targeting the same output of two different design replicas in the following fashion: the first fault remains silent and thus undetected, and after the occurrence of the second fault, both faults manifest and hence the system is unable to determine which is the right output or faulty module.

Table A.4: TMR-MDR approach coverage

Faultload[1]	Enhanced TMR			TMR-MDR approach		
	Number of experiments	Number of failures		Number of experiments	Number of failures	
T	18237	0	(0.00%)	23563	0	(0.00%)
P	18237	0	(0.00%)	23563	0	(0.00%)
T + T	18237	83	(0.46%)	23563	2	(0.01%)
T + P	18237	113	(0.62%)	23563	2	(0.01%)
P + T	18237	3271	(17.94%)	23563	22	(0.09%)
P + P	18237	3085	(16.92%)	23563	22	(0.09%)
Total	109422	6552	(5.99%)	141378	48	(0.03%)

[1] T stands for transient faults in fabric, whereas P stands for permanent faults both in fabric and CMEM, and transient faults in CMEM which manifest as permanent ones in design logic

Although the percentage of faults leading to a failure is very low, it is interesting to determine how this mechanism is affecting the uptime of the system as a highly unavailable system is equally useless. Table A.5 lists the number of experiments causing single and multiple stalls to be introduced.

Results show that single stalls are mainly used to mask transient faults in fabric. In 4.54% of all experiments just one stall applied. This figure reflects the expected benefits because, as described in Section A.3, existing mechanisms reconfigure the affected module as soon as a fault is detected. It must be noted that,

because 93.39% of transient faults generating a single stall targeted a replica and, after implementation, each replica takes up to 27.66% of the occupied silicon area, the uptime is increased avoiding unnecessary reconfigurations. Multiple stalls are related to the actions which prevent the output to be correctly delivered. Experiments affected by multiple stalls vary from 10% to a maximum of 62% in case of 2 permanent faults. Our reference eTMR is defeated in several of the experimented cases. It must be underlined that 32% of the faults resulting in multiple stalls targeted the multiplexers and that, after implementation, this module takes just 5% of the silicon area. This means i) the probability of a fault targeting this module is relatively low, because of its low occupied area (5%), compared to that of replicas (82.98%), and ii) anyway the reconfiguration file will be at most 5% of the total size of the original system, thus the reconfiguration will be fast and the number of stalls quite small. It is also worth mentioning that, according to the considered experimental conditions, which represent the worst case scenario where faults manifest in two consecutive cycles, reported results for experiments leading to multiple stalls represent an upper bound.

Table A.5: Temporal intrusion of the TMR-MDR approach

Faultload[1]	Number of experiments	Experiments leading to single stalls		Experiments leading to multiple stalls	
T	23563	0	(0.00%)	2402	(10.19%)
P	23563	0	(0.00%)	3002	(12.74%)
T + T	23563	2084	(8.84%)	4849	(20.58%)
T + P	23563	115	(0.49%)	7387	(31.35%)
P + T	23563	4214	(17.88%)	10416	(44.20%)
P + P	23563	12	(0.05%)	14623	(62.06%)
Total	141378	6425	(4.54%)	42679	(30.19%)

[1] T stands for transient faults in fabric, whereas P stands for permanent faults both in fabric and CMEM, and transient faults in CMEM which manifest as permanent ones in design logic

Finally, results obtained after the implementation of the system on the selected FPGA are listed in Table A.6. It can be seen that the additional area required with respect to the original non-protected system is relatively large. Likewise, the attainable clock period is rather longer. However, if we compare to the eTMR mechanism, the achievable benefits in terms of resilience greatly surpass the area and clock penalty of just 12 and 20 additional percentage points, respectively.

Table A.6: Area required and clock period attained by the original, the eTMR and the TMR-MDR versions of the target

System	Area	Overhead	Clock period	Overhead
Original	231 CBs	—	25.613 ns	—
Enhanced TMR	820 CBs	254%	30.864 ns	21%
TMR-MDR approach	846 CBS	266%	36.164 ns	41%

A.7 Conclusions

This paper has proposed an approach, combining space and time redundancy with PDR, as a first step to enable the use of low-cost FPGAs in harsh environments. The main benefit of this proposal is that, unlike existing approaches, the considered fault hypothesis takes into account the occurrence of not only single but also multiple proximate faults in time, even while under reconfiguration. The second innovation is the distinction when dealing with transient and permanent faults, which results in just one stall cycle instead of long periods of downtime due to unnecessary reconfigurations.

The feasibility of this approach has been experimentally shown through a case study consisting of a 32-bit floating point multiplier. Experiments were performed in the worst case scenario, when two faults take place in consecutive clock cycles. Even in such adverse scenario, results show the benefits of the TMR-MDR approach compared to other common approaches, like the considered eTMR.

Nevertheless, there are still some issues requiring further research. As previously stated there exist some combinations of faults that may result in a failure under certain circumstances. This is really unlikely, in particular compared to the considered eTMR strategy, although possible, so future work will focus on reducing even more the likelihood of this situation by improving the current resilience of the FSM. Furthermore, as FPGA manufacturers do not provide tools for modifying the state of sequential elements in fabric (FFs), restoring the state of a module or synchronizing the state of replicas after relocation is a challenge that should be faced for further work. Thus, the scalability of the system presents the same issues of a TMR plus the addition that, in case of collision, an arbiter is required for the reconfiguration circuit. A similar case happens with fixed logic blocks in FPGAs (BRAMs, DSPs) as they cannot be rewritten. A different strategy has to be devised to deal with this problem.

Appendix B

The Challenge of Detection and Diagnosis of Fugacious Hardware Faults in VLSI Designs

Authors: Jaime Espinosa, David de Andrés, Juan Carlos Ruiz and Pedro Gil

Current integration scales are increasing the number and types of faults that embedded systems must face. Traditional approaches focus on dealing with those transient and permanent faults that impact the state or output of systems, whereas little research has targeted those faults being logically, electrically or temporally masked -which we have named fugacious. A fast detection and precise diagnosis of faults occurrence, even if the provided service is unaffected, could be of invaluable help to determine, for instance, that systems are currently under the influence of environmental disturbances like radiation, suffering from wear-out, or being affected by an intermittent fault. Upon detection, systems may react to adapt the deployed fault tolerance mechanisms to

the diagnosed problem. This paper explores these ideas evaluating challenges and requirements involved, and provides an outline of potential techniques to be applied.

B.1 Introduction

Current embedded VLSI systems are widespread and operate in multitude of applications in different markets, ranging from life support, industrial control, or airborne electronics to consumer goods. It is unquestionable that the former require different degrees of fault tolerance, given the human lives or great investments at stake, but it is not so obvious to admit that unexpected failures in consumer products can undermine their success in the marketplace [118]. Hence, there is great interest in protecting equipment from eventual faults, which in turn involves providing a certain degree of service reliability over the whole lifetime. Specifically this relies on controlled operation of both software and hardware. While it is clear that potential programming bugs will affect the software behaviour, recent studies on complete systems highlight the disastrous impact which even transient faults happening in the hardware may have in the code execution of critical applications [75]. Therefore in order to achieve dependable devices it is no longer possible to preclude hardware implications from software design.

In the design stage of a product it is foreseeable an evolution in its operational state, from an ideal scenario to another posing several dependability threats. Since a set of specifications has to be met, a conservative approach is taken and security margins are applied to compensate for expected negative effects which may hinder correct service delivery. But that evolution can be no longer predicted accurately enough [23], leaving the only alternative to adapt the system to unexpected changes during its service lifetime. Hence, it is a growingly important requisite to create an information flow from environment to hardware and finally software. A major source of such sudden changes in a system is the occurrence of faults.

To explain why faults appear, there are a number of reasons to be mentioned. For instance, manufacturing capabilities have been evolving at a fast pace, bringing a new breadth of improvements to embedded systems in terms of logic density, processing speed and power consumption. However, those benefits become threats to the dependability of systems, causing higher temperatures, shorter timing budgets and lower noise margins which increase fault proneness. In addition, deep-submicron technologies have both decreased the probability of manufacturing defect-free devices, and increased the likelihood of problematic events originated by wear-out. Moreover, the susceptibility of extremely integrated electronics to α-particles and neutrons, arriving from outer space or radioactive materials grows steadily, yielding a non-negligible degree of so called *soft errors* [85], which affect temporarily the correctness of processing.

The mentioned faults can be classified in permanent or transient categories. Research in the field has focused mainly in tackling permanent faults, disregarding transient faults when their effect is not visible as errors in the captured data. For instance, transient faults with short activation times (percentage of time in which it is affecting the system relative to clock period), which have been shown difficult to detect by conventional means [68], may not produce incorrect outputs at once, but are a good indication of a problematic environment. We have named them *fugacious*. According to [34], out-of-range supply voltages, abnormal noise, temperature, etc. are triggers for such transient faults, which if repeated are called *intermittent* faults. Whether the final nature of the fault is transient or intermittent will depend on the wear-out conditions. For that reason we must make an effort to be able to detect and diagnose such types of faults, because these will provide valuable information when taking decisions for the evolution of the system. An example would be to change the data codification in a bus to a more robust scheme in the hardware, or to enable additional processing iterations or variable checks in the software, for redundancy purposes. Studies devoted to detection and diagnosis of fugacious faults are scarce or non-existent. However, certain known detection techniques could be applied to fugacious faults with limited success [39], since only a reduced period of time is monitored.

The contribution of this paper is based in 2 major points: (i) to identify and ponder the challenges of detection and diagnosis of fugacious faults in VLSI systems and (ii) to provide insight on methods and technologies to cover such challenges.

The rest of the paper is structured as follows. Section B.2 justifies the importance of on-line detection, and underlines the difficulties of detecting transient and intermittent faults with short activation times. Furthermore it presents the different fault models while providing an overview on diagnosis of such faults. A set of methods and technologies is presented in Section B.3 to cope with the task. Finally, Section B.4 indicates the following actions to be taken and related issues.

B.2 The problem of Fast Fault Detection and Diagnosis

According to Avizienis [13] the basic criterion to catalogue faults in permanent or transient type is the persistence. This can however be an incomplete information to comprise the whole picture and thus, *activation reproducibility* is the concept introduced to better describe the observed situations. For permanent faults, different activation patterns lead to solid, hard faults when these are systematically reproducible or to elusive, soft faults when they are not. Depending on circumstances those soft faults can be intermittent in time. For transient faults, elusive activation is the most common but certain circumstances can likewise make them manifest intermittently.

Such differentiated activation patterns require tailored fault tolerant techniques of detection and diagnosis for dependability threats caused by faults and errors. In several situations including high availability or high performance systems, a concurrent detection (on-line) becomes critical. Next the existing scenario related to such concepts is explained.

B.2.1 On-line detection of faults and errors

In order to test proper development of the systems several methods have been described. From post-manufacture checking by means of test vectors or burn-in testing used to discard flawed units, to assigning slots of regular service time for test, for instance, many off-line techniques are currently employed. But the advantage of on-line detection is clear. A loose detection or notification latency, can have disastrous consequences in certain situations [87]. Besides, the longer a fault is present in the system without detection the higher the probability of facing a multiple fault situation. Provided that the latter is a problem of increased complexity we find justified interest in early detection.

There is long tradition in the dependability community to develop on-line error detectors. Typically, they are based in the use of special data codification or in the replication and comparison of outputs or state variables. But the relationship between a fault occurred at the processing network and an error manifested in the outputs or state variables is a limiting factor known as *observability* [21]. When the observability in an output is null for a given fault, no matter which input data combination was applied the fault will not show at the output. Likewise if more than one output is observable for that fault, multiple alterations would be detectable at those outputs. This can be specially important when using encoded circuits, where several properties describe different types of such circuits according to the consequences of a set of faults [67].

- *Fault-Secure.* Circuits where in presence of a fault either the outputs are correct or they are not a valid code word.
- *Self-Checking.* If for every fault of the fault models and for every input either the output is correct or it is detected as error.
- *Self-Testing.* Circuits in which for every fault of the fault set there is at least one possible input which causes the error to be detected.
- *Totally Self-Checking.* Circuits that are simultaneously Fault-Secure and Self-Testing.
- *Code-disjoint.* Circuits where a non-code word input generates a non-code output. This allows detection of erroneous inputs or cascading of blocks.

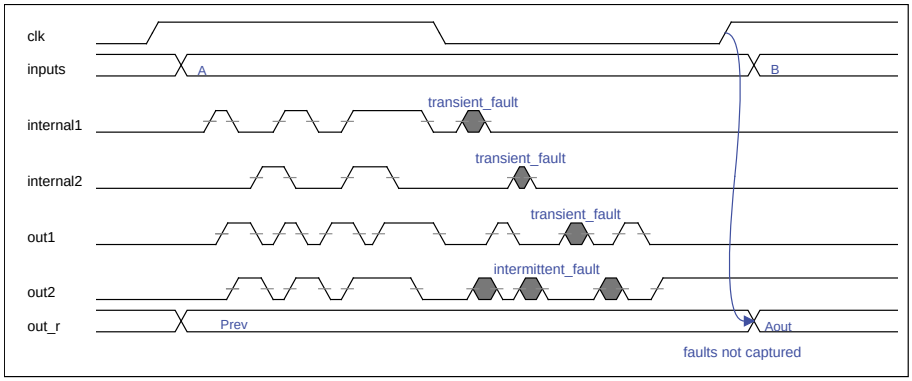


Figure B.1: Temporal filtering of fugacious faults

Therefore when employing codification it will be desirable to maximise the observability in order to achieve a good fault coverage.

There are 3 possible causes for fault filtering: electrical, logical or temporal. When dealing with permanent faults, temporal causes are discarded and due to the nature of hard faults, logical filtering can only be short in time. For those reasons any checking methodology will obtain positive results with hardly any misses meanwhile the observability is good enough.

Nevertheless, detection of transient or elusive and intermittent faults is not so straightforward. On the one hand, and according to field data from digital systems [84], transient faults have been shown to account for up to 80% of failures. These can be caused by several reasons as it is known. Among those reasons, the arrival of α-particles, protons or neutrons from radiation is one of the most studied and popular. If we pay attention to the evolution of transient fault duration produced by one of these particles impacting a CMOS node, the result is directly proportional to the feature size of the electronics [41]. However, the operational frequency of devices has not been following the historic monotonic growing trend, due to well known power dissipation issues. Consequently transients produced by radiated particles and charge build-ups are narrower and narrower compared to the clock periods. This paves the way to believe that although the number of faults affecting a system may be high, chances are these would not be easily captured by clock edges at the storage elements (heavy temporal filtering, see Figure B.1) . The derivatives of this are that the moment a fault is detected many more could have already happened and the available time for reaction could be too short. Therefore for self-awareness purposes it is desirable to detect them.

On the other hand, intermittent errors caused as studied by Nightingale [122] a total of 39% of all hardware errors which, according to reports by Microsoft from

950.000 computers, induced a crash in the operating system. This gives a hint on the number of intermittent faults that can be happening in the system if we consider that not all of them will end in an operating system crash. Other examples of can be found in certain cruise control modules for vehicles [96] which hit a return rate of 96% due to undetected intermittent failures. These figures can be uneven depending on the context of operation since, as distinguished by Savir [152], random originated intermittent faults appear and disappear in an unpredictable fashion whether systematic intermittent faults evolution can be numerically characterized [37]. This enables a proper decision on the best moment to apply recovery actions to maximise availability. Such systematic intermittent faults start by small fluctuations which grow in time and intensity until their effect is severe [159]. In order to set focus on the considered problem, a description of fault models is required.

B.2.2 Considered fault models

According to the presented concept of activation time, and the activation reproducibility described earlier, we have established the name of *fugacious* faults to refer to a set of 3 different types of faults. The fugacious transient faults are defined as those which remain active less than a clock cycle of the system. Likewise, fugacious intermittent faults are those transient faults which activate at least twice in a clock period. Finally, permanent faults are active the whole time span of the clock period, that means for us a fault lasting more than one clock period will be considered permanently active.

B.2.3 Fault diagnosis

Multiple efforts have been conducted towards an effective diagnosis of different types of faults based on their activation reproducibility. As demonstrated in [160] there are important benefits for the Mission Time Degradation and Mean Time To Failure (MTTF) Degradation associated to correctly discriminating transient from permanent faults. It is clear that no equal treatment has to be given to both of them. For instance, transient faults will require no corrective action at all when hardware redundancy provides a voted fault tolerance. Disregarding the affected element for a certain period of time will negatively affect the dependability of the system. Furthermore, given the nature of intermittent faults and their proneness to become permanent, a proper distinction provides insight on the convenience to isolate or recover the functional unit. Intermittent faults diagnosis is a hot-topic in the field. An analytical model for a fault controller was presented in [22], using a thresholds-based α count methodology to discriminate transient from intermittent faults. Its Stochastic Activity Networks (SAN) analysis is specifically based on the time step, where transient faults last for less than one step and intermittent faults repeat their appearance in subsequent steps. Its drawbacks are it requires

a long latency to discriminate, and infrastructure to detect and accumulate the respective faults. Other recent studies which also employ SAN with thresholds [136] applied to real systems only consider intermittent errors captured in state variables, which last more than one clock cycle.

In the case of fugacious faults, we take into account events of a quickly 'evanescent' nature where the capture and diagnosis procedure must have intrinsically low latency. It must be able to process two or more faults per cycle in order to discriminate an intermittent activation from a transient activation, avoiding a new constraint in the frequency requirements.

B.3 Solutions for detection and diagnosis

Our effort has been focused in two directions: determining an appropriate structure to detect and diagnose the set of faults we have previously presented and defining a procedure to apply such structure to the standard design flow.

B.3.1 Architecture of a faults detection and discrimination system

In every VLSI circuit we can find combinational stages separated by registers. Since the pursued goal is to have accurate and flawless computations, we will require 2 conditions:

- *To produce correct results.*
- *To sense any deviations in the datapath which may be out of reach by just checking registered values.*

The steps to take in order to reach these goals start by considering hardware replication and comparison. The large number of commercial systems utilizing such technique tells about the effectiveness of the approach at the expense of important amounts of hardware. The foremost advantage is quick on-line mitigation (when a voter is included), and usually there is no need to include voters in every stage but just in critical ones. Nevertheless, for detection and diagnosis a lighter, cheaper technique would enable the possibility to deploy detection to a larger number of partitions spread around the system. The use of codification may well fill the gap and combine with replication in a wise manner.

An interesting feature of codification is that systematic codes do not require to alter the original bits, thus alleviating the decoding of outputs in the datapath. In order to minimise speed penalty applied to outputs, this makes a great advantage [174]. Berger codes and parity groups are the most popular systematic codes and

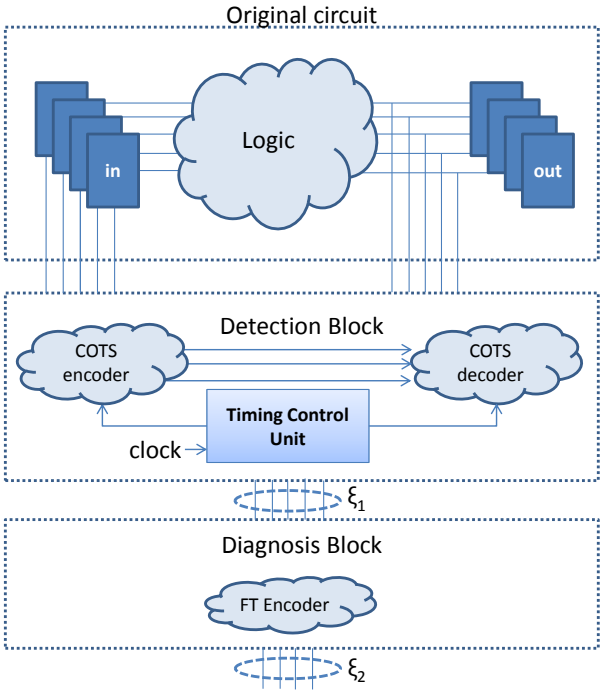


Figure B.2: Global scheme of the faults detection and diagnosis infrastructure. Timing Control Unit handles temporisation of Detection decoder

have been profoundly studied. In [121] conditions for fault secureness in parity predictors are derived. Furthermore [97] presents a generic optimisation technique for parity prediction functions, to achieve quick and small circuits.

The envisioned topology using codification would follow that in Figure B.2. In it, a *Detection Block* would receive inputs directly from partition input registers, and also from outputs prior to registering. A properly selected codification could reduce block area and optimise the speed. This block would include thus a set of Commercial Off-The-Shelf (COTS) encoder and decoder which can be a single bit parity prediction/decoding pair in its simplest form.

Is that enough to handle every type of fault? The answer is 'no'. Coding functions are effective techniques to detect permanent errors, or transient errors which are not time filtered. For effective detection of transient faults of a limited activation time (smaller than a clock period in general), additional elements are required. An example using triplication was presented in [39], where intermittent faults were not considered at all, and the sensing time was rather reduced. On-line detection of intermittent faults has been previously devised in different ways. One proposed

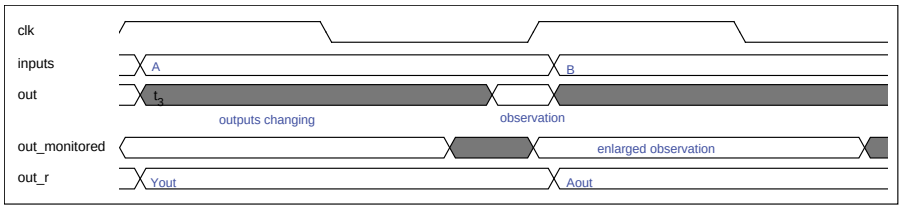


Figure B.3: Observation window enlarged by means of reducing period of signal switching

idea was to inject a carrier signal in the line under study and monitor the correct behaviour of it [95]. Again, the cost is rather high: an injector and receiver for the lines under analysis, plus extra wear-out due to increased switching of the lines. A cheaper detection can be achieved by monitoring those coded lines devoted to detection.

To avoid those shortcomings, an additional element included is the *Timing Control Unit* (TCU). Its function is to adjust the timings of detection elements with one goal in mind, i.e. to increase the *observation window*. The term refers to the percentage of the clock period when the lines under study are monitored for any potential faults. If we reduce the switching interval as opposed to the stability interval of the signals, we will have increased the observation window and thus the effectiveness of the detection (see Figure B.3). The reason for preferring this method to the observation of a reduced period of time assuming equiprobable distribution of faults is clear, i. e. to gain in speed of detection.

Finally once gathered, the detection information could be codified against faults using a code (ξ_1) and passed to a *Diagnosis Block*, where the same or a different code (ξ_2) can be used to notify the diagnosed output to a fault controller.

Inside the *Diagnosis Block*, inputs must be analysed and discriminated to offer 5 different output possibilities:

- *Transient fault.*
- *Permanent fault.*
- *Intermittent fault.*
- *No fault.*
- *Error in diagnosis infrastructure.*

To achieve the goal, the Diagnosis Block will be built using a fault-tolerant (FT) encoder designed to minimise resources taken. By providing all these different

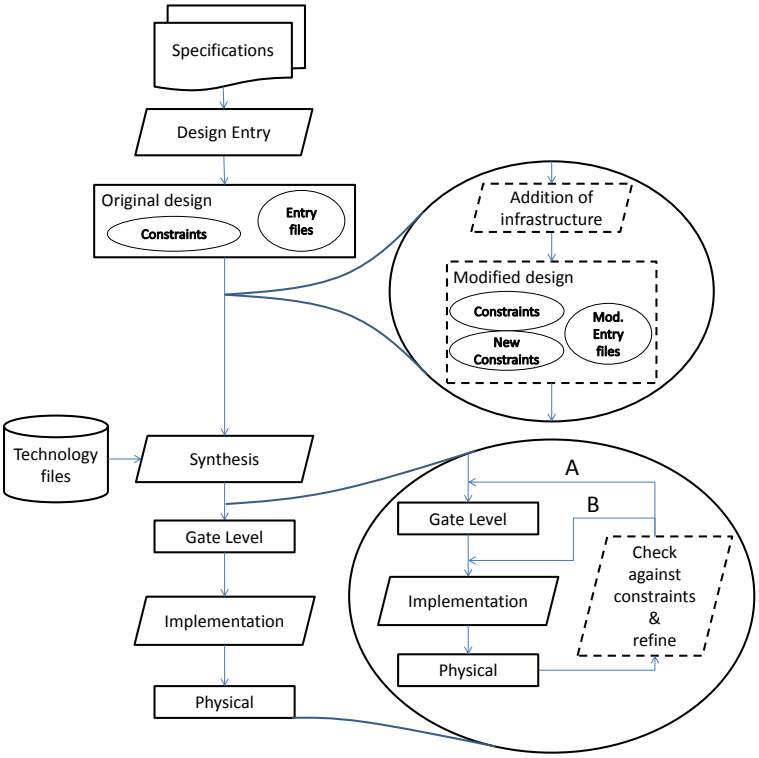


Figure B.4: Tools interaction

outputs and doing so in a fault tolerant codification, the most adequate decision will be enabled to be taken at the fault controller. Hence, smart reactions can be applied well in advance to an eventual collapse of fault tolerance infrastructure.

B.3.2 Workflow to apply in the proposed technique

In order to automate the process of deploying a detection and diagnosis infrastructure to a generic design block, a suggested procedure is shown in Figure B.4. What is depicted is a typical semi-custom design flow for VLSI products, where the standard steps are on the left hand side. *Technology files* can represent a silicon foundry design kit or an FPGA manufacturer primitives library. Likewise, *Physical* element can be a layout file or a programming bitstream for an FPGA. On the right we find detail of 2 interventions in the flow.

A first intervention comes before the *Synthesis* and after *Design Entry*. This step comprises an addition of required infrastructure in the Detection Block, i.e. the

COTS components and Timing Control Unit. Entry files are modified as required and new timing constraints are generated for the TCU, to drive the remainder of the design flow.

A second intervention happens in a loop between Gate-level and Physical stages of the design. The purpose is to check timings against new constraints, mainly affecting the TCU, and refine the implementation in a loop by tweaking in one of the 2 re-entry points A or B. If path B is selected, a faster process will be obtained as a result, but deep knowledge of the underlying technology will be required and we will find a side effect of loss of portability. With path A, a more general solution will be obtained at the cost of speed of implementation.

The challenge in the integration of processes is derived from the difficulty to achieve the optimum observation window for the whole range of process variability. Other difficulties can come from the capabilities and restricted information offered by technology suppliers.

B.4 Ongoing Work

An initial implementation is currently under development, where an FPGA-based design flow has been chosen to support initial testing. Following the presented ideas, we have been able to develop first modification point working models. To reach optimal performance, we need first is to maximise the detection capabilities of the structure, both in area and time. This means achieving a high degree of observability at the check lines.

As for the second modification our efforts are devoted to achieve low performance penalty results and at the same time maximising the period in which lines are under surveillance. We need the least possibly intrusive system in order not to give in too much in exchange for detection. This is vital when applied to extreme performance demanding systems.

Last but not least, keeping the additional area small can be complex in certain circuits, if a powerful logic optimisation is not wisely applied. The upper limit will be that imposed by pure replication but this should be perfectly reducible without loosing much of the observability. An associated parameter to area increase is the power drain due to new infrastructure. As usual in engineering, specifications and market constraints drive the balance between detection and diagnosis capability and power/area/performance penalty.

Appendix C

Increasing the Dependability of VLSI Systems Through Early Detection of Fugacious Faults

Authors: Jaime Espinosa, David de Andrés and Pedro Gil

Technology advances provide a myriad of advantages for VLSI systems, but also increase the sensitivity of the combinational logic to different fault profiles. Shorter and shorter faults which up to date had been filtered, named as fugacious faults, require new attention as they

are considered a feasible sign of warning prior to potential failures. Despite their increasing impact on modern VLSI systems, such faults are not largely considered today by the safety industry. Their early detection is however critical to enable an early evaluation of potential risks for the system and the subsequent deployment of suitable failure avoidance mechanisms. For instance, the early detection of fugacious faults will provide the necessary means to extend the mission time of a system thanks to the temporal avoidance of aging effects. Because classical detection mechanisms are not suited to cope with such fugacious faults, this paper proposes a method specifically designed to detect and diagnose them. Reported experiments will show the feasibility and interest of the proposal.

C.1 Introduction

In recent years, manufacturing capabilities have been evolving at a fast pace, bringing a new breadth of improvements to embedded systems in terms of logic density, processing speed and power consumption. However, those same benefits also become threats to the dependability of systems by causing higher temperatures, shorter timing budgets and lower noise margins, decreasing the probability of manufacturing defect-free devices, and increasing the likelihood of failures originated by wear-out.

Examples of such threats include, among others, i) the growing susceptibility to α-particles and neutrons arriving from outer space or radioactive materials, yielding a non-negligible degree of so called *soft errors* [85], ii) the noise affecting power supply lines which creates unexpected delays in critical paths [62], and iii) the Electromagnetic Interference (EMI), which can be inserted in the system over the air or through wires [76]. With increasing miniaturisation scales, a lesser amount of energy is required for an upset to reach the voltage threshold of the technology and generate a transient fault and, for the same amount of energy, those faults are shorter in duration [60]. New fabrication techniques such as silicon-on-insulator (SOI) follow this trend, with pulse widths decreasing from $250ps$ for $250nm$ feature sizes to $110ps$ for $100nm$ [41]. In the same way, the steady shrinking of voltage thresholds and the dramatic increase of speed have allowed for the propagation of shorter and shorter transient faults. For instance, the minimum pulse width required for propagating a transient fault in SOI has been reduced from $105ps$ for $350nm$ to a $40ps$ for $100nm$ sizes [41]. Finally, in high frequency systems the occurrence of transient upsets increases linearly with the frequency [25]. Altogether, these facts raise concern on the growing numbers and variety of faults next generation systems must face effectively.

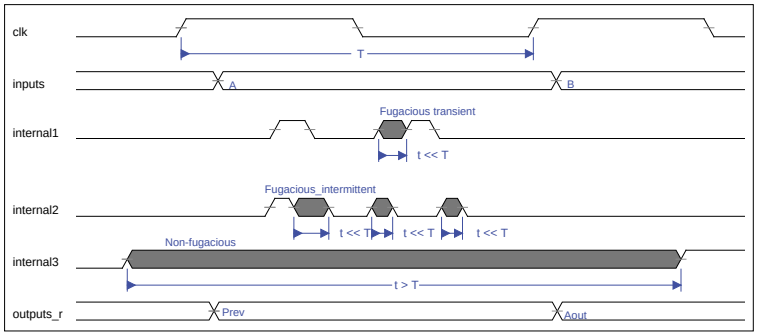


Figure C.1: Characterisation of fugacious faults

Research in the field has focused mainly on fault tolerance [13], i.e. preventing the system from failing via error detection (identifying the presence of an error) and system recovery (transforming the system state into one without errors and without faults that can be activated again). In order to be effective, existing techniques usually require errors to manifest in the state of the system, commonly represented by the contents of its sequential logic. However, deep integration scales are causing an increasing number of transient faults with shorter and shorter activation times (with regards to the clock period), known as *fugacious* faults [56], to be missed as they are not affecting the state of the system (they are not captured by the sequential logic). Although this is the expected behaviour of common fault tolerance mechanisms, the occurrence of fugacious faults is an excellent indicator of harsh environments, wear-out conditions, ageing, extreme temperatures, etc., which may finally lead to a system failure. If such faults are properly detected and diagnosed it could be possible for the system to face new operating conditions, effectively adapting its hardware and software towards new intrinsic or extrinsic demands, even enabling a forecast on future fault impact rates.

Some illustrative examples of the usefulness of an early detection and diagnosis of such fugacious faults include i) a satellite suddenly crossing a high radiation level area, which may trigger a reconfiguration process to increase the system redundancy before the radiation level is high enough to cause a failure, ii) an intentional EMI attack to break secrecy in an encryption core, allowing to change data codification or deploy any other countermeasures, or iii) an ageing problem in a braking control system of a train, which could well raise service alarms before total loss of control. Accordingly, the proper detection and diagnosis of fugacious faults may provide valuable information when taking decisions for the reconfiguration/evolution of the system to keep or improve its safety. If faults are missed, or even detected too late, they could well cause safety hazards, financial losses, or profit drop.

Previous studies devoted specifically to detect and diagnose fugacious faults are scarce or non-existent. Some works have mainly focused on characterising transient faults caused by radiation along different technologies [59]. However, they do not deal with their detection and diagnosis in working circuits. To this end, certain known detection techniques could be applied to fugacious faults with limited success [39], since only a reduced period of time (with respect to the clock period) is monitored. Hence, new detection capabilities are key to tackle the ever-changing profile of faults as technology advances.

A first step towards this goal was presented at [49], but this paper greatly extends that work by presenting a novel architecture and methodology for the detection and diagnosis of fugacious faults, which may be later used to trigger the system reconfiguration, to keep or improve its dependability, based on fault forecasting.

The rest of the paper is structured as follows. Section C.2 defines the proposed fault models for characterising fugacious faults. Section C.3 introduces the novel architecture and methodology to provide early detection and diagnosis of fugacious faults, and a suggested implementation flow is presented in Section C.4. The selected case study to show the feasibility of the proposed approach is detailed in Section C.5, whereas the obtained results are commented in Section C.6. Finally, Section C.7 concludes the paper and presents some ideas for future research.

C.2 Fugacious fault models

Faults are commonly classified according to their persistence [13] divided then into *permanent* faults, whose presence is continuous in time, and *transient* faults, whose presence is bounded in time. With new integration scales, the classification of transient faults was refined to characterise the particular behaviour of a new kind of faults, the so called *intermittent* faults. They account for transient faults that are repeated in the same area in a short period of time and due to the same cause [34]. Whether the final nature of a fault is transient or intermittent will depend on several factors, namely wear-out condition, ageing, extreme temperatures, etc.

This classical categorisation of faults required a further specialisation to account for the particular behaviour of more frequent and shorter faults induced by increasing integration scales. Accordingly, we propose a refinement of that classification for the characterisation of fugacious faults. It must be noted that the prime characteristic of a fugacious fault is its short duration, that may prevent the fault from being captured by sequential elements. That is why, the system clock cycle is taken as a reference for defining the models of fugacious faults. As such, faults with the same duration could be considered as fugacious or not depending on the system operation frequency. Figure C.1 illustrates the proposed fugacious fault models.

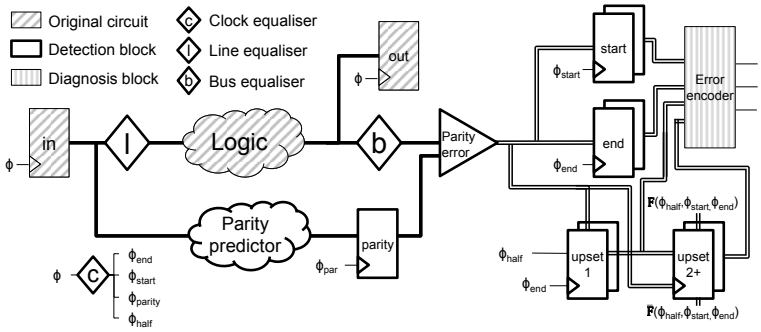


Figure C.2: Low level schematic implementation of the proposed detection and diagnosis architecture

Transient fugacious faults are those with a duration shorter than a system clock cycle and that occur just once during that clock cycle. Likewise, *intermittent fugacious* faults also had a duration shorter than a system clock cycle but, on the other hand, they appear more than once within the same clock cycle. As there could exist different activation patterns (number of pulses in a burst, duration of each pulse, and interval between pairs of pulses), this definition enables the characterisation of all the different possibilities under a common term.

Finally, those faults with a duration longer than a clock cycle are considered as *non-fugacious* from the perspective of fugacious faults. Being the scope of fugacious faults limited to the current clock cycle, it is not possible to determine the exact nature of the fault within that clock cycle, but it is necessary to observe the behaviour of the signal along consecutive clock cycles. That is why, from the perspective of fugacious faults, a very long pulse will be considered as a non-fugacious fault, which could be classified as a transient, intermittent or permanent fault from a higher level perspective.

Such differentiated activation patterns require tailored fault detection and diagnosis strategies for fugacious faults, as commonly used mechanisms rely on faults being captured by sequential logic to be detected. Due to the quickly 'evanescent' nature of fugacious faults, the latency of proposed mechanisms should be really low and, what is more, they should be able to detect two or more faults per clock cycle in order to discriminate intermittent from transient activations.

C.3 Novel architecture for detecting and diagnosing fugacious faults

Many commercial systems make use of hardware replication and comparison, at the expense of greatly increasing the area taken by the circuit and reducing its clock frequency, to mitigate the effect of faults in the system [105]. Nevertheless, for detection and diagnosis, lighter and cheaper techniques would enable the deployment of detection schemes to a larger number of partitions spread around the system, while relying on failure suspectors to trigger reconfiguration strategies to prevent further faults from affecting the circuit. Next sections will show a high- and low-level view of the proposed architecture.

A high-level perspective of the architecture could be roughly described as an error detection code system, which runs in a timing controlled framework to provide inputs to tailored diagnosis infrastructure. Careful selection of such code can provide a number of advantages and literature on the field has been analysed. Systematic codes, like Berger codes and parity groups, are very interesting as the original bits are not altered, which alleviates the decoding of outputs in the original datapath. Equally important, conditions for fault security in parity predictors (outputs are either correct or form an invalid code word) were derived in [121], and a generic optimisation technique for parity prediction functions to achieve fast and small circuits was presented in [97].

Accordingly, a detailed low level view of the proposed architecture is depicted in Figure C.2. The original circuit, comprising input and output registers and combinational logic, is highlighted in a diagonal patterned background. The *detection block*, comprising the encoder or parity predictor, the decoder or parity error, the detection auxiliary registers and the timing processing elements (for bus, line and clock), is shown in plain white background. Finally, the *diagnosis block* which consists of a fault tolerant encoder is depicted in vertical lines background. Interconnection double lines represent dual rail encoding to detect the occurrence of single faults in the detection block, as valid data should be either (‘0’, ‘1’) or (‘1’, ‘0’).

The *detection block* receives its inputs directly from partition input registers and from computed outputs just prior to registering. In first place, a parity predictor is in charge of generating the expected output parity code from registered inputs. This parity code is then stored in the *parity* auxiliary register on the ϕ_{parity} clock edge. Particularly tailored codifications could reduce block area and optimise speed, on a case per case basis.

In order to detect both transient (one pulse within one clock cycle) and intermittent (more than one pulse within one clock cycle) fugacious faults, it is necessary to continuously monitor computed outputs prior registering. However, it is not

Table C.1: Diagnosis of fugacious faults*

Fault	Start	End	Upset1	Upset2+
Non-fugacious	'1'	'1'	'1'	'1'
Transient fugacious	Any	'0'	'1'	'0'
	'0'	'1'	'1'	'0'
Intermittent fugacious	'1'	'0'	'1'	'1'
	'0'	Any	'1'	'1'
None	'0'	'0'	'0'	'0'
Diagnosis error	Any other combination			

* For simplicity, '1' here means "00" or "11", and '0' means "10" or "01".

possible to monitor those outputs during the whole clock cycle, since they intermittently switch as the inputs are propagated through logic elements. The monitoring process can take place once computed outputs are stable. Accordingly, just a small fraction of the whole clock cycle could be monitored for fault detection by using this approach.

To overcome those shortcomings, our proposal relies on including a novel element into this architecture, a *Timing Control Unit* (TCU), represented by the clock equaliser depicted in Figure C.2. Its function is simple: adjusting the timings of detection elements (ϕ_{parity}, ϕ_{start}, ϕ_{end}, ϕ_{half}) to stretch as much as possible the stability period of the signals (*observation window*), and thus maximise the probability of detecting any fault occurring during the clock cycle. Likewise, line and bus equalisers are also used to selectively delay the required signals with the same purpose.

The parity error decoder receives both the expected parity code and equalised circuit outputs to generate a dual-rail code which is then fed to the *diagnosis block*.

This code is then captured by different dual registers at different specific times. *Start* registers capture at the beginning of the stability period (within the observation window) using clock ϕ_{start}. *End* registers capture at the end of the stability period (within the observation window) using clock ϕ_{end}.

Meanwhile, two sets of dual registers, *upset1* and *upset2+*, are devoted to capture upsets occurred during the whole stability period. To do so, ϕ_{half} which is half the clock frequency keeps those registers switching each period. In the case of the *upset1* registers, each register of the pair is fed by ϕ_{half} and $not(\phi_{half})$, respectively, using the ϕ_{end} clock edge. Accordingly, their state alternates between ('0', '1') and ('1', '0') each clock cycle, ensuring they store a valid data code. The output of the parity error decoder is connected to the set input of both *upset1* registers, thus causing an invalid ('1', '1') data code whenever an upset is detected. The initialisation of the *upset2+* registers is performed through a combinational function that takes into account ϕ_{half}, ϕ_{start}, and ϕ_{end}. This function and its inverse

are respectively connected to the set and reset inputs of one of the registers, and to the reset and set inputs of the other register. This function will only activate those signals outside the observation window. It will also ensure that the registers will store a valid data word alternating between ('0', '1') and ('1', '0') each clock cycle. Upon the occurrence of an upset, the output of the parity error decoder will cause these registers to capture the current state of the *upset1* registers. In case of being the first upset during the clock cycle, they will receive a valid data code. When further upsets are detected, they will store ('1', '1'), an invalid data code signalling that two or more upsets have been detected.

All these data (the parity error at the beginning and end of the observation window, and the state of the *upset1* and *upset2+* registers) will be passed to the encoder in charge of diagnosing the kind of fault, if any, that have been detected. Table C.1 lists the different results of the diagnosis process according to the received inputs. The encoder output is a 3-bit signal that guarantees fault security, as only 4 different codes are required out of the 8 available. The result of the diagnosis process will be passed to a failure suspector that may trigger any corrective measures if required (like reconfiguring the affected component to increase its fault tolerance capabilities or relocating it to another fault free area of the device).

C.4 Proposed implementation flow

The typical semi-custom design flow for VLSI products should be adapted in order to automate the process of deploying the proposed detection and diagnosis infrastructure. The left hand side of Figure C.3 depicts the common implementation flow, where *Technology files* can represent a silicon foundry design kit or an FPGA manufacturer primitives library. Likewise, *Physical* element can be a layout file or a programming bitstream for an FPGA. The right side of the figure details the required addition to that flow to support this novel detection and diagnosis strategy.

The first intervention consists in introducing all the required infrastructure, as previously presented in section C.3, to support the desired detection and diagnosis. This is performed just before *Synthesis* and after *Design Entry*. Entry files are modified as required and new timing constraints are generated for the *Timing Control Unit* (TCU).

A second and more complex intervention takes place in a loop between *Gate-level* and *Physical* stages of the design. Its purpose is to stretch the observation window by checking timings against new constraints, mainly affecting the TCU, and successively refining the implementation by tweaking in one of the 2 possible re-entry points. Path *B* leads to a faster, more precise process, but requires a deep knowledge of the underlying technology and reduces the portability of the

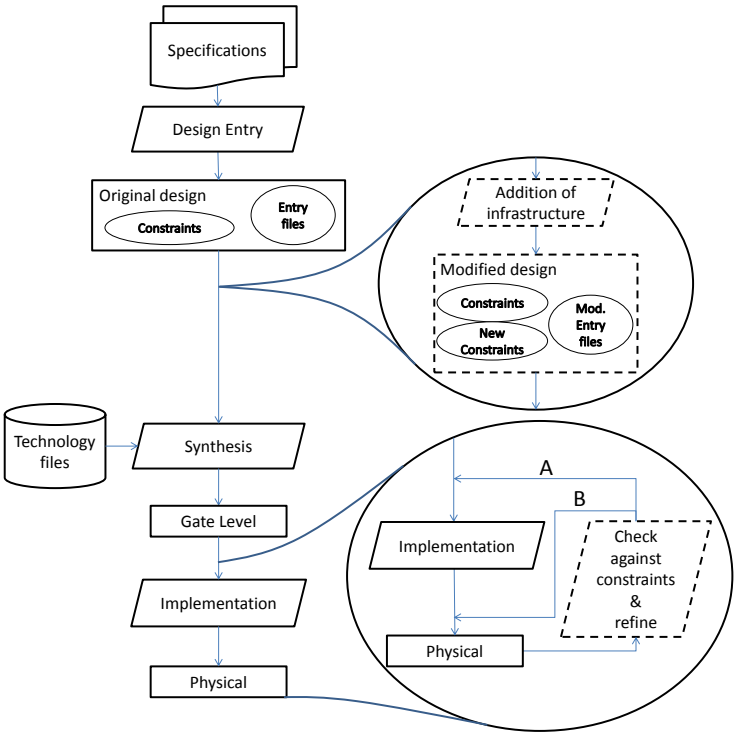


Figure C.3: Proposed implementation flow

approach. A more general solution will be obtained using path *A* at the cost of increasing the implementation time. This second intervention follows the flow shown in Figure C.4.

The initial step tries to adjust the datapath to stretch the observation window by selectively delaying the output and input lines. First, a bus delay equaliser is used to reduce the dispersion of output paths delays reaching the detection infrastructure (see Figure C.5). This is an iterative process in which the slowest output will be preserved as reference, and the fastest output will be delayed by inserting delay elements in its path. This process ends whenever the fastest output is also the slowest one. Once bus outputs have been equalised, a further compaction can be possibly achieved by applying a similar process to the inputs. This time a line equaliser delay is applied to those inputs which appear quicker at the first logic level (see Figure C.5), provided they do not violate the same conditions of the bus equaliser. This guarantees that the maximum clock frequency attainable is not affected.

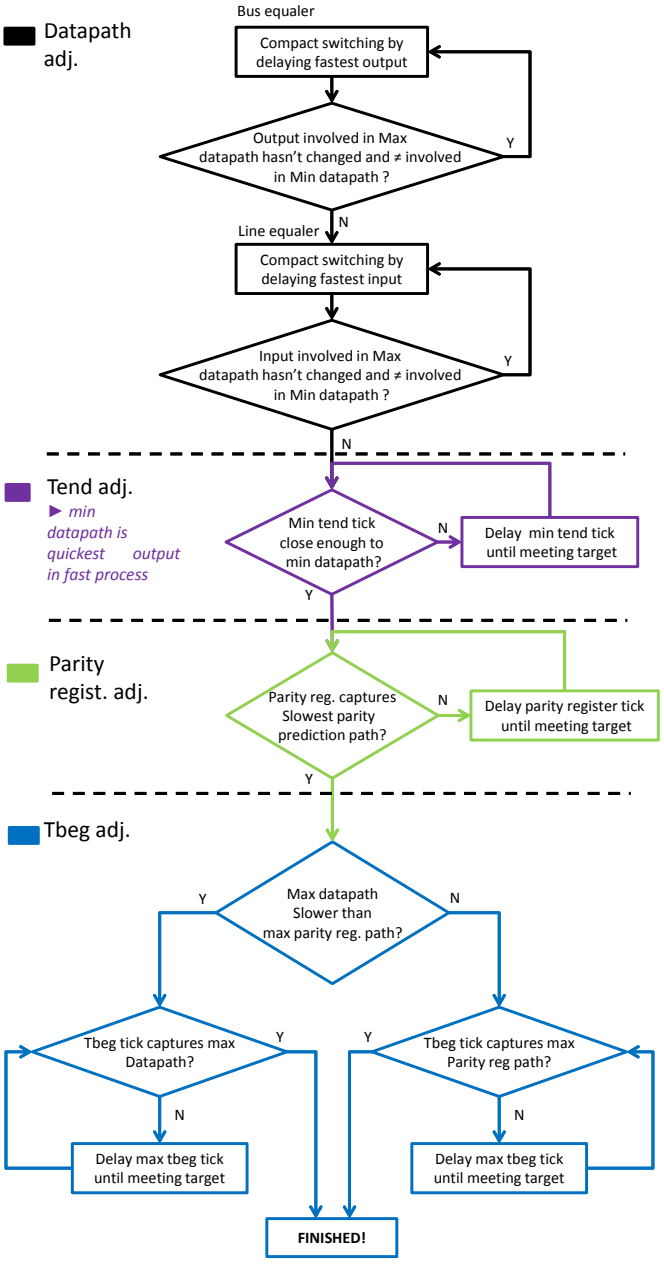


Figure C.4: Control flow for stretching the observation window

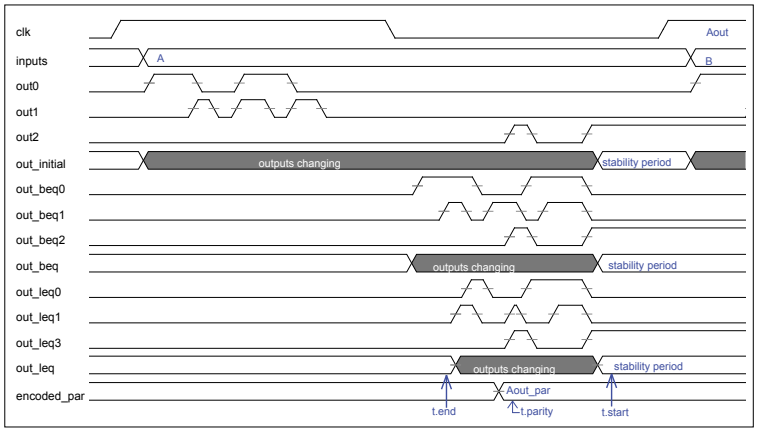


Figure C.5: Stretching the observation window step by step

The next step dephases ϕ_{end} so that the clock edge is in close vicinity to end of the stability period (the beginning of the switching period due to the quickest line of the datapath in the fast process corner). By forcing this condition iteratively the end of the observation window is pushed as late in time as possible.

The registers devoted to capture the predicted parity are tweaked in the third step. The ϕ_{parity} clock is delayed as necessary to capture the slowest parity prediction signal path. This is done iteratively to adjust the capture tick to the earliest possible moment.

Finally the fourth step focuses on dephasing ϕ_{start} so that the clock edge is located at the beginning of the stability period (observation window). First, it is necessary to determine whether the maximum propagation time of data feeding ϕ_{start} is longer than that of parity registers outputs feeding ϕ_{start}. This clock tick will be iteratively delayed to ensure that both data and parity registers outputs are correctly captured.

In this way the switching period has been shrunk as much as possible, thus enlarging the stability period, and the capture clock edge of all registers has been carefully dephased to observe this whole period.

C.5 First prototype and case study

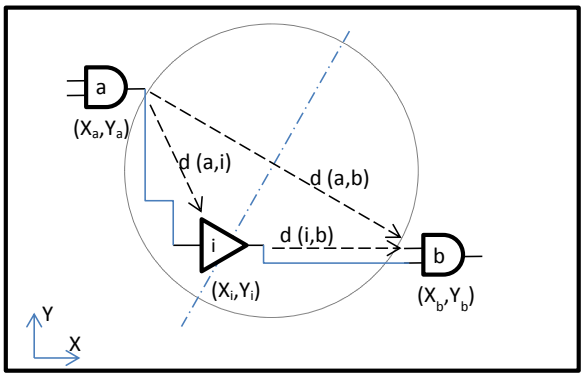
In order to test the functionality of the proposed architecture, a standard 4-bit adder has been selected as a suitable candidate. This is a state of the art combinational circuit, used in multitude of designs, which can suffer from undetected fugacious faults. Furthermore its simplicity eases the task of performing controlled experiments (only a small amount of eligible fault injection points for the placed and routed design) within a reasonable time frame.

As this circuit is modelled in Verilog, the required infrastructure to deploy the proposed architecture is also described in this HDL language. This infrastructure and the associated new timing constraints are manually inserted into the design before synthesis, although the automation of this process is currently under development. For its implementation, a Virtex-6 FPGA platform has been chosen due to the ease and speed of deployment, the availability of information related to its internal structure (XDL plain text files), and its partial dynamic reconfiguration capabilities that could be exploited after detecting and diagnosing the occurrence of faults. For the first prototype of a tool supporting this methodology, a publicly available toolset named TORC [161] has been used to interface with the manufacturer tools, which present certain limitations in the routing side hindering the required equalisation and dephasing. As the goal of this first implementation is just showing the feasibility of the proposed approach, it does not tune inserted delays in the finest possible way, so the observation window is not stretched to its physical limits.

The procedure for inserting delays in a target path is based on selecting an intermediate point between origin and destination, for the path to pass through. The location of that intermediate point is determined according to the algorithm shown in Figure C.6. This algorithm, based on geometrical principles takes into account different possible cases. Initially, if the delay to be introduced is very small, a position next to the origin or destination points is selected. For small and medium delays, the position is selected somewhere midway between origin and destination, modifying the total path distance according to the required delay. For bigger delays and also when origin and destination points are too close, the intermediate point is placed around a circumference, which includes both origin and destination, whose radius is modulated by the required delay.

This pass-through element can be a simple buffer in ASIC, or a Look-Up Table (LUT) implementing the identity function in an FPGA platform. In the latter case, when the chosen LUT is busy a spiral shaped search for a free close-by LUT is initiated until one is found or a new intermediate position is recalculated. Future work will focus on improving this implementation.

The controlled injection of fugacious hardware faults on an FPGA is also quite difficult. That is why, to test the detection and diagnosis capabilities of the pro-




```
If (Rdelay < 0.5) then
      (Xi,Yi) = (Xa, Ya-1)
else if (Rdelay < 1) then
      Xi = (Xa + dx(a,b)/2)
      Yi = (Ya + dy(a,b)/2)
else if (Rdelay < 2) and (d(a,b) > dmin) then
      α = Rdelay*π/2
      k = atan (dy(a,b)/dx(a,b))
      Xi = Xa + d(a,b)/2 *cos(α + k)
      Yi = Ya + d(a,b)/2 *sin(α + k)
else
      α = Rdelay*π/10
      k = atan (dy(a,b)/dx(a,b))
      Xi = Xa + (d(a,b)/2 +Rdelay*10) *cos(α + k)
      Yi = Ya + (d(a,b)/2 +Rdelay*10) *sin(α + k)
```

Figure C.6: Strategy for locating delay pass-through elements (X_i, Y_i), showing physical distances inside device. Delays are expressed in relative units.

posed architecture, fugacious faults will be injected in the post-place and route model of the system following a model-based fault injection technique. A number of Tcl scripts have been developed to ease the injection of different fugacious and non-fugacious faults using the Modelsim simulator [71]. Next section reports on the results obtained from experimentation.

C.6 Results and discussion

The first set of experiments is devoted to test the minimum pulse width that could be correctly detected as a transient fugacious fault. Table C.2 lists whether detection is achieved for shorter and shorter pulses in both fast and slow corners of the technology.

Table C.2: Minimum width of fugacious transient faults for correct detection

Pulse width T=10ns	Detected in Fast Process Corner	Detected in Slow Process Corner
0.5·T	✓	✓
0.1·T	✓	✓
0.05·T	✓	✓
0.01·T	✗(a)	✓
0.005·T	✗(a)	✓
0.001·T	✗(a)	✓

The narrowest pulses present a *pulse swallowing* effect, annotated as (a) in the table. In this cases, the physical limitation of the technology applies, as pulses are not wide enough to be propagated through the logic elements. This prevents those faults from being detected. Curiously enough, simulations do not feature this behaviour in the slow process corner, but common sense dictates it will happen at a certain moment. The proper characterisation of this behaviour requires further research.

The second set of experiments aims at determining the minimum separation (inactivity time) between consecutive pulses within a burst to properly detect it as an intermittent fugacious fault. In this case, two pulses with fixed width of $0.05 \cdot T$ are injected with decreasing inactivity time. Table C.3 lists whether faults are correctly detected, for fast and slow process corners, with decreasing inactivity time.

Table C.3: Minimum inactive time of intermittent fugacious faults for correct detection

Inactivity time width T=10ns	Detected in Fast Process Corner	Detected in Slow Process Corner
0.5·T	✓	✓
0.2·T	✓	✓
0.15·T	✓	✓
0.1·T	✓	✗(a)
0.05·T	✓	✗(a)
0.01·T	✗(a)	✗(a)
0.005·T	✗(a)	✗(a)

As focusing on detecting faults with very short duration, in case that the total length of the burst is longer than one clock cycle, then several transient faults could be reported instead (not shown in the table). On the other hand, technology

Table C.4: Check all diagnosis cases in all eligible fault injection points

Fault type	Injected value	Number of injections	Correctly detected	Incorrectly detected	Error in diagnosis	Not detected
Transient fugacious	Low pulse	9	6	0	0	3 (*b*)
	High pulse	9	2	0	0	7 (*b*)
Intermittent fugacious	Low pulse	9	6	0	0	3 (*b*)
	High pulse	9	2	0	0	7 (*b*)
Non-fugacious	Low pulse	9	2	5 (*b*)	0	2 (*b*)
	High pulse	9	0	6 (*b*)	0	3 (*b*)

limitations (*pulse swallowing*) appear again in those cases marked with (*a*). It is remarkable that for a slow process corner longer inactivity periods will be required as compared to fast process corner, where correct detection can be achieved for closer pulses within a burst.

Once the technological limitations imposed by the selected Virtex-6 platform are known, a new set of experiments is performed to check the different entries of the diagnosis table (see Table C.1) on all the 9 eligible fault injection points of the circuit selected as case study. Table C.4 details the results of this experimentation, reporting the number of faults correctly and incorrectly detected, or not detected at all. It must be noted that, as pulses width has been carefully selected according to the previous experiments, results are exactly the same for both fast and slow process corners, so they are just reported once on the table.

The first unexpected, but foreseeable result, is that non-fugacious faults are first detected as transient fugacious faults. This makes sense, as those faults last more than one clock cycle and there occurrence will be reported as transient faults during this first cycle. However, on the next clock cycle, they will be correctly diagnosed as non-fugacious faults. Accordingly, failure suspectors should take into account this phenomenon to wait for the next clock cycle before taking any decision and action.

Likewise, it can be noted that there exists a big number of not detected faults, marked as (*b*) on the table. After carefully analysing the simulations, we can conclude that this is the result of *logic filtering*. For instance, injecting a logic '0' in a node holding this same value, or injecting a logic '1' to the input of an AND logic gate when some other input holds a '0', prevents the fault from being propagated and thus detected. When this *logic filtering* is applied to non-fugacious faults, they will be probably detected as transient fugacious faults when the affected line should be switching along consecutive clock cycles. Obviously, the erroneous value should appear during two consecutive clock cycles at least to be correctly diagnosed. As in this case non-fugacious faults have been injected for $1.3 \cdot T$, they are mostly erroneously detected or not detected at all.

Finally, the overhead induced by the required detection and diagnosis infrastructure is shown in Table C.5.

Table C.5: Overhead induced in terms of area and clock period

Design	Area utilisation	Clock period
Original with I/O registers	6 Slices	1.83 ns
Protected with infrastructure	36 Slices	3.49 ns

Although the extra area for the required infrastructure could seem a bit too high in comparison with that required by the original circuit (about six times more), it must be noted that this is really a very small circuit. Furthermore, the Virtex-6 slice is huge in size (4 LUTs, 8 FFs, plus infrastructure) and those 30 additional slices are mostly using a small fraction of the available resources ($< 30\%$), so there is plenty of room for further circuit packing.

Finally, the attained clock period is not as close to that of the original circuit as desired. This is due to the tools used for the bus, line, and clock equalisation processes, which did not allow a fine control of the extra delay to be inserted. For the future it is intended to switch to a different more controlled method to add finely measured delays, taking into account manufacturing dispersions.

C.7 Conclusions

This paper presents a novel methodology to detect and diagnose a specific type of previously neglected faults, the so called fugacious faults. They are described as faults whose active period is smaller than the clock period of the system and they are not usually captured by the sequential logic of the system. Accordingly, they have been usually neglected, but their proper detection and diagnosis could be of great use to foresee the the proximity of harsh environments or the occurrence of wear-out mechanisms. The proposed approach appears as an effective method to quickly detect and diagnose such faults.

From the set of experiments performed to shown the feasibility of this approach, a number of lessons has been learned. First and foremost is that technology limits set a barrier against the quickest detectable upset in the circuit, which applies to each single pulse or the separation between pulses in the same burst. Another important fact is that logic filtering limits the observability of faults, so it makes sense to pay attention exclusively to output lines. Additionally, although its implementation in a modern FPGA may seem costly in terms of area penalty, if applied to large blocks or critical circuits cost versus benefit ratio is greatly improved.

Future research will focus on improving the equalisation tools to fine tune the insertion of controlled delays into the system, thus increasing the observation window and reducing the clock period penalty to the minimum possible. Likewise, the inclusion of a failure suspector in conjunction with the dynamic partial reconfig-

uration capabilities of FPGA will greatly increase the safety of the target system through its adaptation to face the unexpected events that may arise.

Appendix D

An Aspect-oriented Approach to Hardware Fault Tolerance for Embedded Systems

Authors: David de Andrés, Juan Carlos Ruiz, Jaime Espinosa and Pedro Gil

The steady reduction of transistors size has brought embedded solutions into everyday life. However, the same features of deep-submicron technologies that are increasing the application spectrum of these solutions are also negatively affecting their dependability. Current practices for the design and deployment of hardware fault tolerance and security strategies remain in practice specific (defined on a case-per-

case basis) and mostly manual and error prone. Aspect orientation, which already promotes a clear separation between functional and non-functional (dependability and security) concerns in software designs, is also an approach with a big potential at the hardware level. This chapter addresses the challenging problems of engineering such strategies in a generic way via metaprogramming, and supporting their subsequent instantiation and deployment on specific hardware designs through open compilation. This shows that promoting a clear separation of concerns in hardware designs, and producing a library of generic, but reusable, hardware fault and intrusion tolerance mechanisms is a feasible reality today.

D.1 Introduction

Current embedded VLSI (Very Large Scale Integration) systems are widespread and operate in multitude of applications in different markets, ranging from life support, industrial control, or avionics to consumer electronics. Benefits of current manufacturing capabilities, in terms of attainable logic density, processing speed and power consumption, become threats to systems dependability, causing higher temperatures, shorter timing budgets and lower noise margins [118]. In addition, deep-submicron technologies have both decreased the probability of manufacturing defect-free devices, and increased the likelihood of wear-out related problems and the susceptibility to radiated particles [36]. Likewise, communications among devices expose hardware embedded systems to a number of external threats, especially when they are manufactured as an aggregation of off-the-self (OTS) Intellectual Property (IP) cores developed by third, and sometimes untrusted, parties. Nonetheless, reusing these components offers a reduction in time-to-market costs and a rapid integration of technology innovations while minimizing the risk of designs that integrate millions of gates [1, 176]. It is unquestionable that critical systems require different degrees of fault and intrusion tolerance, given the human lives or great investments at stake. However, nowadays, the consideration of resilience in modern VLSI designs, understood as the ability of the system to ensure acceptable levels of dependability and security despite changes, is a requirement even in the industry of non-critical applications, as the occurrence of unexpected failures in consumer products may negatively affect the reputation of manufacturers and undermine the success of new products in the market. The dependability and security communities widely accept that involving unskilled designers in the development of non-functional strategies (such as fault- and intrusion-tolerance and security) may actually have a negative impact on the global resilience of the deployed solution [58]. There is therefore an emerging requirement for frameworks supporting the separate design of non-functional and system core (functional) mechanisms, and their subsequent integration. In other words, fault and intrusion tolerance mechanisms must be developed by experts, but hardware designers

with limited expertise in dependability and security must be able to integrate such mechanisms in their designs to make them resilient to faults and attacks. How to support such separation of concerns during the design of dependable VLSI solutions remains an open challenge today. Aspect orientation [94] provides interesting means to cope with this issue from the first steps of the system design flow, when integrated circuit models become available. The vast majority of modern solutions to digital circuit design revolve around the use of HDL (Hardware Description Language) models. Using such languages, hardware designers program circuits in a modular and hierarchical way. By modifying such models, related circuits can be accordingly adapted and evolved. The notion of metaprogramming, defining programs that automatically reason about and customize the structure of other programs, encompasses this idea. If this customization is specialized for fault tolerance [170], metaprogramming becomes a valuable technique to develop dependable strategies, which can be later (automatically and transparently) deployed onto HDL models following an open compilation process. This chapter explains how an open compilation process can be established to support i) the implementation of fault tolerance and security techniques as metaprograms, and ii) their subsequent application to HDL-based models of integrated circuits. Additionally, this process must be seamlessly integrated into the regular design flow typically followed for HDL-based hardware systems, thus offering a great potential to increase the productivity of designers and reduce their error proneness. Other asset is that it can be applied as soon as a model is ready to simulate, even if it is not synthesizable yet. Hence, this opens the chance to study the impact of the applied modifications in the early stages of the design cycle, thus reducing the costs associated to late corrections. By enabling the automated integration of non-functional mechanisms and system functional mechanisms, the old idea of providing libraries of dependability and security mechanisms that could be reused in different contexts and deployed on different components could become a reality. The rest of the chapter introduces first the basic concepts about aspect orientation and metaprogramming, and existing approaches to translate those concepts into hardware design in general, and the deployment of fault tolerance and security strategies in particular. After that, an already existing framework is used to describe a procedure for metaprogramming hardware fault tolerance and its integration into the hardware design flow. Taking this procedure as a guide, two examples of different metaprograms implementing time-redundant and symmetric encryption mechanisms are detailed in depth. The feasibility of the proposed approach is demonstrated by automatically deploying the implemented strategies onto a PIC microcontroller core and analyzing both the benefits obtained in terms of dependability and security, and the cost to pay in terms of silicon area, throughput and energy consumption overhead. Finally, different open challenges for further research are discussed.

D.2 Related Work

Although aspect orientation and metaprogramming have been applied to the development of fault-tolerant software during years, they have not been fully considered yet for their integration in the regular hardware design flow. Academia and industry research has mainly focused on metaprogramming a restricted number of commonly used fault tolerance strategies, but no framework supporting the metaprogramming of any required type of fault and intrusion tolerance mechanism has been considered yet. These concepts and current efforts towards provisioning metaprogramming and aspect orientation to hardware design are next presented.

D.2.1 Metaprogramming and aspect orientation

"Computational reflection is an activity performed by a system when doing computation about (and by that possibly affecting) its own computation" [108]. When reflection is applied to compilers, it is possible to modify the code analysis performed at compile-time and influence the code generation process, i.e. programs (named *metaprograms* or *aspects* in this context) can analyze and customize the structure of conventional programs as needed. This idea was initially applied to compilers by [29] and later by [94] as a mean to express and deploy transversal (non-functional) mechanisms on object-oriented programs. As a result, a metaprogram is associated to a compiler in order to customize its compilation process through a number of analysis and transformation rules. An exchange of information thus takes place at compile-time between the compiler and its metaprogram. The compiler reifies structural information about the input program, let's name it P, to the metaprogram. That metaprogram then uses intercession mechanisms to apply code modifications resulting from the analysis and customization process it carries out. This is how the input program P is transformed into a different output program, named P'. By default, the metaprogram applies an identity transformation to the input program P, i.e. the output program P' is equivalent to P. However, the user can modify such behavior by refining the implementation of the default metaprogram, i.e. by adapting existing or specifying new transformation rules. Such compilers are called *open compilers* (OC) after their ability to modify the default compilation rules by means of metaprograms. It is worth mentioning that in aspect-oriented programs, metaprograms are named aspects, since they express cross-cutting features of programs, and open compilers are called weavers, since they weave the functional code provided by programs with the non-functional one provided by aspects. Despite the difference of notation, the goal of both metaprogramming and aspect-oriented programming remains the same, i.e. promote a clear separation of mechanisms in programs. What is important in this chapter is that the underlying specialization process can be applied to generate a library of mechanisms that, if focused on providing fault and intrusion tolerance,

can constitute a library of resilience mechanisms like those created for software systems [5, 58].

D.2.2 Hardware fault and intrusion tolerance automation

Due to current stringent time-to-market constraints, the traditional manual development, adaptation, and deployment of the required mechanisms is no longer an option. Therefore, it seems just natural that both academia and industry had devoted great efforts towards defining methodologies and tools for the automatic generation and deployment of fault detection and tolerance mechanisms into hardware designs. One of such suitable methodologies relies on aspect-oriented programming (AOP) concepts that, although being applied to object-oriented high-level programming languages for years, have been seldom used for hardware design [45]. This survey constitutes a seminal work that identifies crosscutting concerns in hardware descriptions and provides a first definition of possible join-points and pointcuts for HDLs. Other preliminary works applying AOP concepts to the design of digital hardware focused on the design of a SystemC-based synthesizable resource scheduler at RTL [116], and the definition of an aspect-oriented extension of VHDL, named AspectVHDL [112]. As can be seen, the adoption of AOP concepts for hardware design is still in its infancy and, to the best of our knowledge, no work has focused yet on the application of AOP for the provision of fault detection and tolerance capabilities for hardware designs. Other methodology consists in providing separation of concerns implemented via metaprogramming for the design of customizable components [168]. This work discussed the use of just the target HDL or in combination with another metalanguage as metaprogramming paradigms for higher flexibility, reusability and customizability. These paradigms enabled the massive hardware replication through Triple Modular Redundancy (TMR) and error detection/correction codes for registers protection [46], and tolerating single bit-flips in a state register and multiple faults in the next state computation logic [102]. The automatic insertion of detection and fault tolerance mechanisms in high-level HDL descriptions, like Finite State Machines (FSM) or Register-Transfer Level (RTL), instead of at a lower (gate) level, allowed the early insertion and validation of the considered mechanisms at the cost of losing control over the generated hardware and increasing the overhead in terms of area, performance, and power consumption. Following this paradigm, a number of different tools have emerged to provide alternative solutions for improving the dependability and security of hardware designs. vMAGIC (VHDL Manipulation and Generation Interface) constitutes an automatic code generation tool, developed by the University of Paderborn, that makes use of a Java library to read, manipulate, and write VHDL code [128]. Although it has not been specifically designed with dependability in mind, the inclusion of this tool in the hardware design flow improves code reliability and reduces the development time. A similar approach, but dealing with EDIF (Electronic Design Interchange Format) netlists

instead of VHDL code, is implemented by the BYU EDIF Tools [182]. These tools work at a lower level, thus circumventing all the issues related to the optimizations performed by synthesis tools during the implementation process, although a high level of expertise is required to precisely define the required transformations to obtain the desired circuit. This set of tools includes an Automated TMR application to automatically deploy hardware replication. This same idea of providing a library of already defined fault detection and tolerance components and mechanisms is exploited by CODESH [146], an open compilation process for the design of dependable and secure high-level HDL descriptions. The default library provided by CODESH contains the N-Modular Redundant hardware replication strategy [144], and an error detection and correction Hamming approach for information stored on registers [143]. It is to note that this is the only recent tool from academia specifically developed to define and automatically deploy components and mechanisms for dependability and security. On the industry side and with TMR as its main dependability strategy too, XTMR Tool was the first commercial development tool to address the special requirements of programmable logic devices in high-radiation environments [188]. It automatically builds TMR into Xilinx FPGA (Field-Programmable Gate Arrays) designs, thus increasing designer productivity but limiting its applicability to just this particular strategy and only for Xilinx products. Mentor Graphics Precision Hi-Rel [119] offers a more generic approach by automatically adding TMR or safe FSM encoding at synthesis time, so targeting a wide variety of devices and manufacturers. However, it has been deemed International Traffic in Arms Regulation (ITAR) controlled by the US Department of State, so it can only be provided to United States Persons within the United States. This brief survey seems to support the notion that "AOP is a goal, for which reflection (metaprogramming) is a powerful tool" [94]. Indeed, Kiczales et al. (1996) started by developing simple metaobject protocols with which prototype imperative language aspect programs. Later, with a better knowledge of what the aspect programs should do, more explicit aspect language support was developed. This empirical process exactly describes how academia and industry are adopting the AOP concepts for the automatic development of hardware systems. Nowadays, and as previously presented, the stronger current still focuses on metaprogramming the required rules to customize components and systems to include predefined components and mechanisms for dependability and security. The next and more ambitious step of developing HDL extensions for AOP support, not just for dependability but also for any other non-functional concerns, still requires further research to reach a stage mature enough to transfer this approach to the industry. Accordingly, next sections will unfold the details of how to articulate a metaprogramming approach for the design of hardware systems as implemented by CODESH, the most up to date non-commercial tool that specifically focuses on the definition and deployment of fault detection and tolerance components and mechanisms.

D.3 Metaprogramming the design of dependable and secure HDL-based embedded systems

HDLs are description languages specifically designed to model the behavior of synchronous digital circuits in terms of the flow of information between hardware registers (Register Transfer Level), or specify the behavior of the circuit by means of logic equations or logic gates and their interconnections (Gate level). Similarly to other programming languages, HDL models are processed by specific design compilers, called synthesizers, in charge of transforming the HDL code listing into a physically realizable gate netlist. This netlist can take one of many forms, like a simulation-oriented netlist with gate delay information, or a standard EDIF format, which can be later used to either implement the design on reconfigurable hardware devices like FPGAs or manufacture silicon-based circuits. Hence, as hardware faults occur at the physical level, it is necessary to understand how these faults manifest at higher HDL levels (gate or RTL) to determine the kind of fault detection and tolerance mechanisms that could be considered and how they could be deployed into the HDL model of the system. As different types of faults can induce the same type of errors, it is enough that these faults induce similar behaviors (fault models). Transient faults appear during the normal operation of the circuit for a short period of time after which they disappear again. They usually result from the interference or interaction of the circuitry with its physical environment [92], such as transients in power supply, crosstalk, electromagnetic interferences, temperature variation, alpha and cosmic radiation, etc. The resulting fault models at RTL and gate levels are the *bit-flip* (reverses the logic state of a memory cell), *pulse* (reverses the logic state of a combinational logic element), *indetermination* (undetermined logic value between the high- and low-logic thresholds), and *delay* (increases propagation delay of a line). Permanent faults are due to irreversible physical defects in the circuit. They usually appear as a result of the manufacturing process or the normal operation of the system. In this latter case, sometimes they initially reveal as intermittent faults until some long-term wearout mechanisms cause the occurrence of a permanent fault [66]. The fault models derived at RTL and gate level include the *stuck-at* (fixes the logic value of a logic element), *stuck-open* (fixes the logic value of a logic element for a retention time, and to '0' afterwards), *short* (short-circuits two lines), *open-line* (splits a line into two parts), *bridging* (special combination of open-line and short), *indetermination* and *delay*. Accordingly, the challenge of metaprogramming the design of dependable HDL-based hardware systems is two-folded. On the one hand, it is necessary to articulate the metaprogramming approach to ease the definition and deployment of RTL and gate level fault tolerance mechanisms as metaprograms. On the other hand, this process should be engineered in such a way that it could be easily integrated in the common design flow for both programmable logic devices and standard cells. CODESH [146], a framework providing an open compilation

process for the design of dependable and secure HDL-based systems, will be taken as an example of how to meet these two requirements.

D.3.1 Open compilation to support the customization of hardware systems

The common open compilation flow has been customized by CODESH to ease the definition and deployment of dependable and secure components and mechanisms for HDL-based hardware designs. The resulting open compilation process is depicted in Figure D.1.

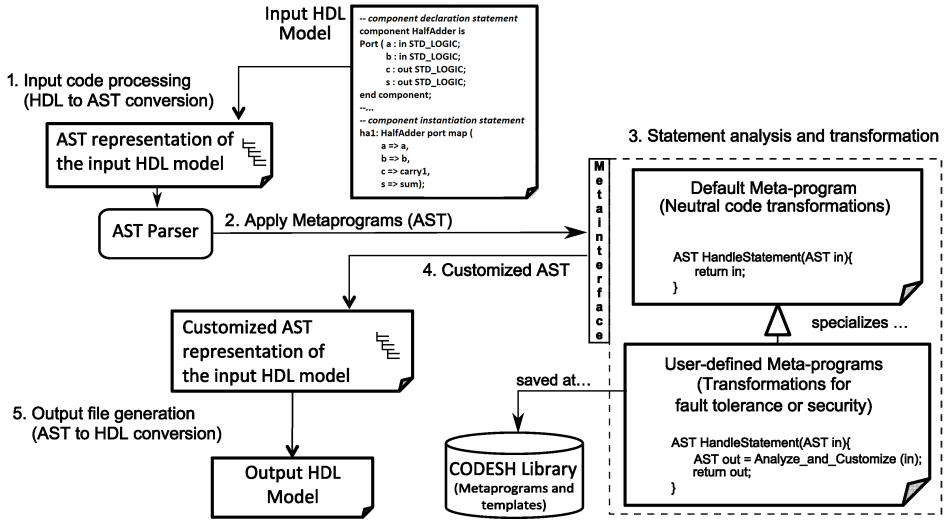


Figure D.1: Open compilation process defined by CODESH

The very first step consists in processing the input HDL model to obtain an abstract tree-form representation, or Abstract Syntax Tree (AST), of all the HDL constructs occurring in that model. Each node of this tree embeds the information related to the particular HDL construct it represents. For instance, an AST node for the component instantiation statement shown in the Input HDL Model of Figure D.1 will provide syntactical information about the instantiated unit (*HalfAdder*), its name (*ha1*), and the mapping established between its ports (*a, b, c,* and *s*) and the signals they connect to (*a, b, carry1,* and *sum*). In particular, CODESH has been developed using ANTLR [126] and thus all AST nodes inherit from the CommonNode class, which is specialized for each existing HDL construct. Once the AST generated, a parser is in charge of walking through the obtained data structure. For each identified construct (component instantiation statement, for instance), a metaprogram has the chance of analyzing and customizing the AST representation of such construct (step 2 in Figure D.1). The communication be-

tween the open compiler and the metaprogram is performed through a well-known interface implemented by the *Metainterface* class. Metaprograms must implement that part of the Metainterface related to those constructs they are willing to customize. The AST node representing the construct to be handled is systematically provided to the suitable metaprogram, which is thus activated (step 3 in Figure D.1). The variety of actions a metaprogram can carry out, defined by means of analysis and transformation rules, will obviously vary according to the considered resilience strategy. In fact, the implementation of the default metaprogram follows a neutral transformation approach, meaning that no actual transformation takes places onto the original HDL construct represented by the input AST. This behavior can be specialized through inheritance, so non-neutral metaprograms should overload those methods required for their specific purpose, such as removing constructs or modifying their internal structure, either by changing their internal elements (like identifiers or types) or by introducing new ones. It must be noted that the way in which analysis and transformation rules are applied to input HDL models also rely on the structure of such models. Accordingly, as each metaprogram could customize just a particular construct of the input HDL model, a whole set of metaprograms could be required to implement a given fault tolerance mechanism. In fact, these rules can be viewed as templates that are used to adapt the implementation of resilience mechanisms to each particular hardware component structure. Hence, each newly defined metaprogram, or set of metaprograms, integrates CODESH library of components and mechanisms for dependability and security. The customized AST node resulting from applying those analysis and transformation rules implemented by the metaprogram is finally returned back to the open compiler (step 4 in Figure D.1), thus replacing the original AST. Once this approach is recursively applied to all the nodes in the AST, the open compiler finally generates an output file reflecting all model transformations (step 5 in Figure D.1).

As an example, Figure D.2 details the process followed for replacing a non-fault-tolerant component by its Triple Modular Redundant (TMR) version. TMR is a well-known strategy consisting in physically replicating the hardware component to be protected, and obtaining the right output of the system by majority voting the outputs of all the replicas [12]. The leftmost column of the figure lists the structural definition of a FullAdder in terms of two internal HalfAdders. The AST obtained after parsing the VHDL model (step 1 in Figure D.2 is shown in the second column of the figure, with arrows mapping VHDL statements to AST nodes. Steps 2 to 4 apply a TMR strategy to *ha2* HalfAdder through transformations (a) and (b), thus affecting different nodes of the AST (shown in boldface in the third column of the figure). Finally, the rightmost column in Figure D.2 lists the TMR enhanced version of the original model after step 5 takes place. This schema tolerates any number of transient faults, as long as they affect just one replica at a time and disappear before any other fault occurs, and just one permanent fault affecting one replica. This approach is greatly favored when protecting hardware components

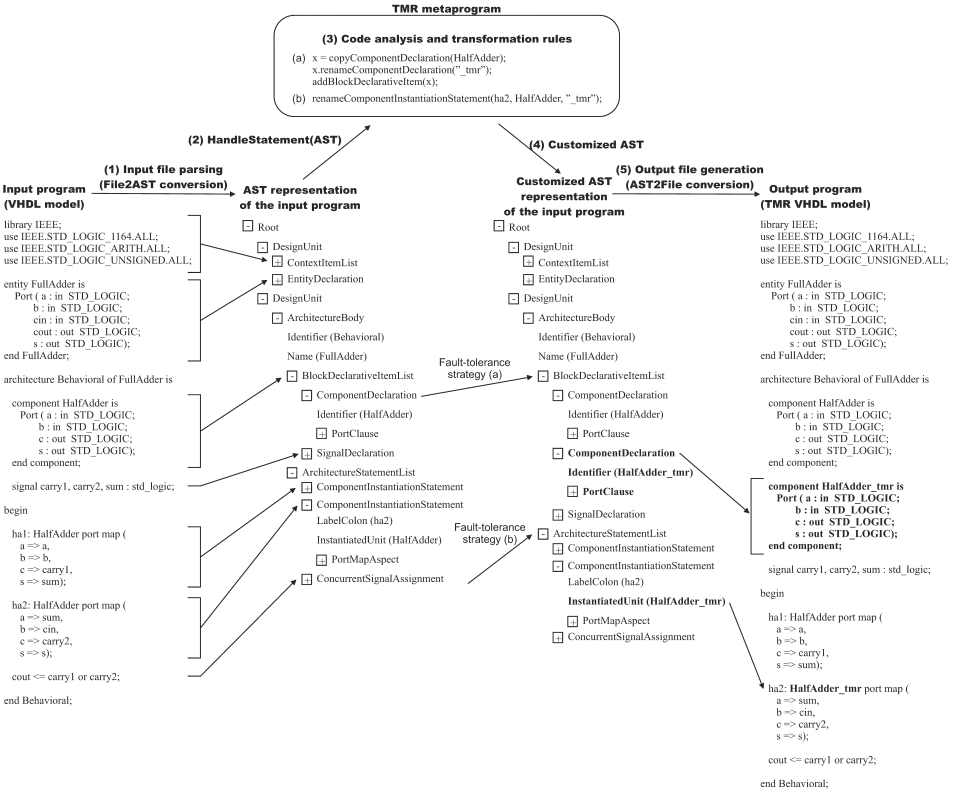


Figure D.2: CODESH Open compilation process in action: a TMR case

due to its simplicity and good tolerance capabilities, although it has a large cost in terms of silicon area, making it more suitable to critical-systems rather than consumer electronics.

D.3.2 Architecting hardware fault tolerance mechanisms as metaprograms

Currently, CODESH library of fault tolerance mechanisms contains a set of metaprograms enabling i) the deployment of spatial redundancy through N-Modular Redundancy [144], ii) the detection and correction of errors on registers via Hamming codes [143], and iii) the detection of errors on buses through Cyclic Redundant Checks [50]. After careful analysis, experience has demonstrated that fault tolerance strategies for hardware designs can be defined as metaprograms responsible for i) generating the required logic infrastructure, ii) encapsulating this logic with the original core within a new component, and iii) integrating this new compo-

nent into the original model. Any fault tolerance or security strategy for hardware systems needs to make use of multitude of different common components used in hardware design, like registers or multiplexers, and specific components to detect or tolerate faults, like comparators or majority voters. Particular metaprograms should be developed in order to generate this infrastructure according to the requirements of the given strategy. Once developed, these metaprograms will integrate the library of predefined components and strategies for fault tolerance, so they could be reused by any other strategy requiring them. It must be noted that metaprogramming the whole HDL model of any new component (a majority voter, for instance) is not a simple task. That is why it is recommendable to reuse the proposed open compilation approach to define some HDL *templates*. These templates (input HDL models in this case) should contain as much as possible of the required structure and only those parts that must be adapted to the particular strategy or hardware under consideration will be dynamically generated by means of smaller and much simpler metaprograms (like the size of the input/output ports, for instance). After the required infrastructure has been generated, it is usually encapsulated with the original target component (or system), which retains its original functional capabilities, to introduce the new non-functional capabilities (fault detection and tolerance in this case). This can be seen as a kind of *wrapper* producing a fault-tolerant and/or secure version of the original component. Metaprograms will be in charge of i) configuring and instantiating as many components as required, and ii) interconnecting all these elements to implement the selected strategy. To achieve this goal it is essential to parameterize the whole process according to the particular strategy and, in many cases, the target component. When those components are delivered in the form of HDL models (*soft IP cores*), or when reusing components developed in-house, it is possible to access their internal structure (white box approach) and precisely determine the required customizations. However, third party cores are usually delivered as *hard IP cores*, which are already implemented (synthesized, placed and routed) and ready for manufacturing. This black box approach prevents designers from getting any knowledge about the internals of the component, thus limiting the available information to that provided by the manufacturer. Newer approaches enabling and easing the reflection on hard IP cores are then in need. Finally, once the fault-tolerant and/or secure version of the target component has been obtained, another metaprogram will replace the original component by this customized version. In case the target component was the top-level component of the system, there is no replacement, but the new component is the one to be used from now on in the rest of the design process.

D.3.3 Integration within the regular hardware design flow

The common digital hardware design process, depicted in Figure D.3, involves several steps to obtain a final product from initial specifications. Verification is usually performed after each step of the design (considering a respectable design size) to prevent any undetected error from causing further delays. This general design flow has been enriched to include the open compilation process required to introduce metaprogramming as a technique to generate and deploy fault detection and tolerance mechanisms in the model of the system under development. The newly added steps shown in Figure D.3 are highlighted in light grey to ease their identification.

The first step takes a set of functional and non-functional specifications to act as an input for the design of a system meeting these requirements. From these specifications, an HDL model describing the behavior and/or structure of the final system is defined. The design entry, although mostly done using HDL files, could also take into account acquired off-the-shelf components, which are integrated with other hand-coded components in a hierarchical style. Step 2 in Figure D.3 performs functional simulations of the HDL model of the system to verify its proper behavior in terms of service delivery. It must be noted that, as non-functional mechanisms have not been deployed yet, this simulation just verifies that the correct functionality has been implemented. At this point (step 3 in Figure D.3), fault injection could be a useful mean to determine the sensitivity of each component to the fault models considered representative of each particular system. The aim of this process is to determine which mechanisms should be applied, and where they should be deployed, to meet dependability and security requirements. Fault injection may take place at this stage, using a simulation-based tool like VFIT [65]. Any problem reported during simulation or fault injection will feedback the design process, thus modifying the HDL model to correct any deviation from the specification. Provided that end users of CODESH will be HDL designers, and assuming that this community is not intended to know about the subtleties of open compilation and its implementation, an easy to use approach has been implemented to help them to specify where and how to deploy a given metaprogram (dependability and security mechanism) into the system model. This approach consists in inserting a number of HDL comment lines with a special format that has a particular meaning to the CODESH parser (step 4 in Figure D.3). The specific comment lines and related parameters for the customization of the deployed components and mechanisms are documented for each metaprogram available in the library. When running the open compiler, comment lines are first parsed to determine which metaprograms must be applied for each HDL construct. The main benefit of this approach is that hardware designers are already familiar with it to specify synthesis directives in commercial products, like Synopsys' Design Compiler [187]. Furthermore, as it makes use of actual HDL comments, any Electronic Design Automation (EDA) tool may implement the original model of the

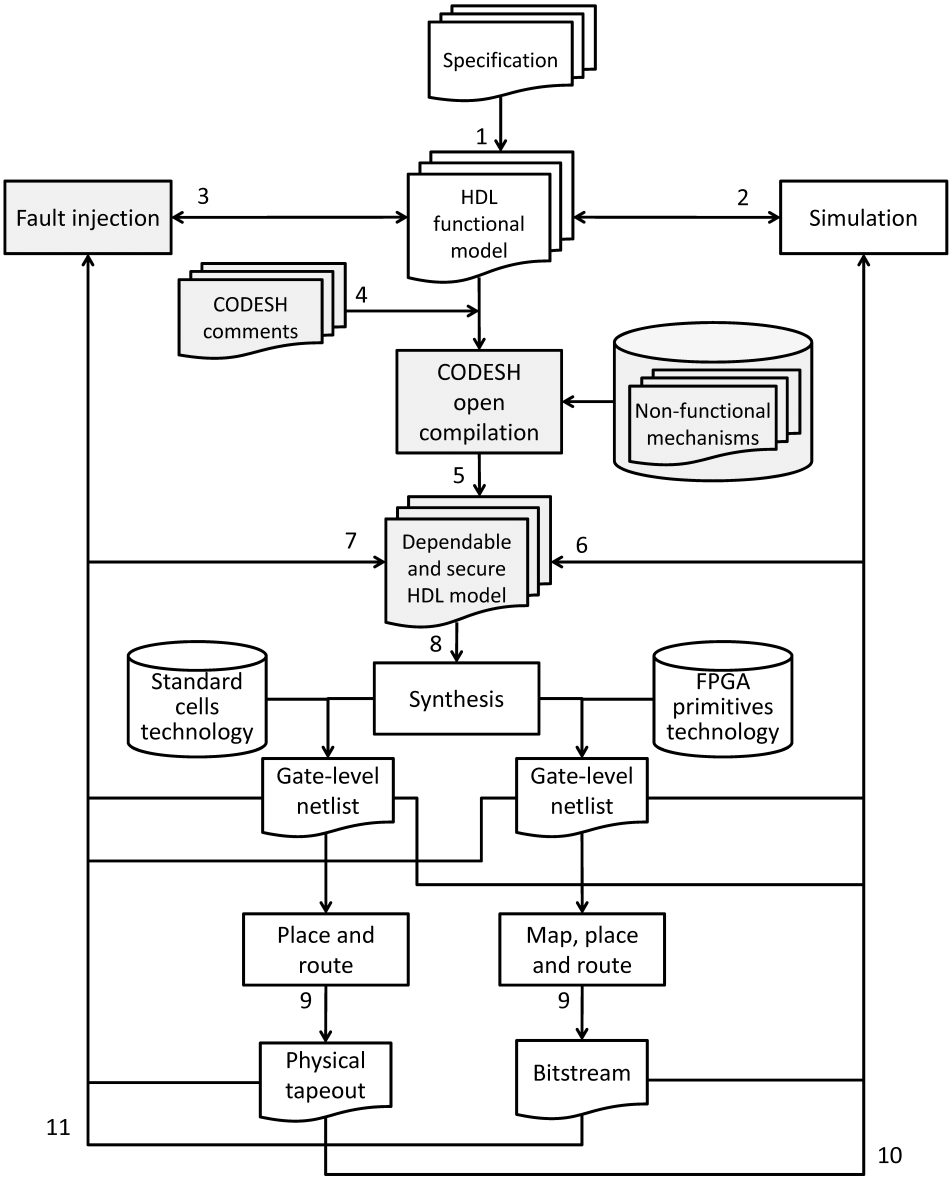


Figure D.3: Integrating the proposed open compilation process into the regular hardware design flow

system as it is, ignoring those comments that are only meaningful for CODESH. It is at this point that CODESH open compiler takes the incoming HDL model, which includes all the special comments specifying the parameterization of the different fault tolerance and intrusion mechanisms to be deployed, and applies all the required metaprograms to generate the output HDL model following the previously explained procedure (step 5 in Figure D.3). This new model can be then simulated and subjected to fault injection as in the previous step to confirm that all the (functional and non-functional) requirements are still met (steps 6 and 7 in Figure D.3). This output model is the one re-injected into the regular design flow for its implementation. The common hardware design flow typically continues with synthesis (step 8 in Figure D.3), which uses specific libraries for the final implementation technology selected (like FPGAs or Standard Cells) and generates a gate-level design adapted to the end target. Following stages (step 9 in Figure D.3) include map and place and route to achieve a physical tape-out or configuration file. Finally, the implemented design is simulated again (step 10 in Figure D.3), including extracted timings, to check that the functional requirements are still met. Likewise, emulation-/prototype-based fault injection (step 11 in Figure D.3) may be also considered to validate the non-functional requirements [8, 90, 127]. Any deviations from the specifications should be corrected by either parameterizing the implementation (synthesis, mapping, and place and route) process or modifying the HDL model. This may lead to iterate through the design flow until the requirements are finally met. As shown, the metaprogramming approach followed by CODESH can be seamlessly integrated into the regular hardware design flow. Next section will describe, by means of two case studies, how far more ambitious fault tolerance mechanisms than those commonly considered can be engineered while facing the problem of dealing with white and black box approaches.

D.4 Dealing with white and black box IP cores as case studies

As previously presented, third party components could be troublesome, as their degree of openness could limit the visibility of its internal structure and thus the flexibility metaprograms will have to deal with them. To illustrate both cases and show the feasibility of metaprogramming to deploy fault tolerance and security mechanisms, two different cases studies have been considered. The first metaprogram will define a fault tolerance strategy to tolerate transient faults via temporal redundancy for white box combinational components. Later on, a second metaprogram will enable the integration of a qualified black box encryption core into an embedded component to secure its communications with other components.

D.4.1 White box IP cores: tolerating transient faults via temporal redundancy

Temporal redundancy involves the use of additional time to perform tasks related to both software and hardware fault tolerance. It usually requires the repetition of a failed execution using the same resources involved in the initial one. This simple approach may tolerate the occurrence of timing or transient faults, as long as the causes of that fault are already absent prior to the re-execution of the failed task. Otherwise, the re-execution will lead to a failure again. Hence, temporal redundancy could be a suitable solution for those systems that cannot afford the extra cost of including redundant logic but may easily neglect longer execution times, like mission-critical systems [132]. Although latency-critical applications, like many of those that can be found in nowadays mobile devices, are not so eager to trade timing for area, the inclusion of a hardware core implementing this mechanism in the integrated circuited will alleviate the temporal cost of applying a software-based approach. Figure D.4 shows the generation and customization processes that should be applied to the original model of the system in order to deploy a temporal redundancy mechanism. Solid black lines represent data lines, solid gray lines depict control lines, whereas dotted lines are used for the clock signal distribution. The different generation and customization rules devised for the automatic deployment of such mechanism are described in the following sections.

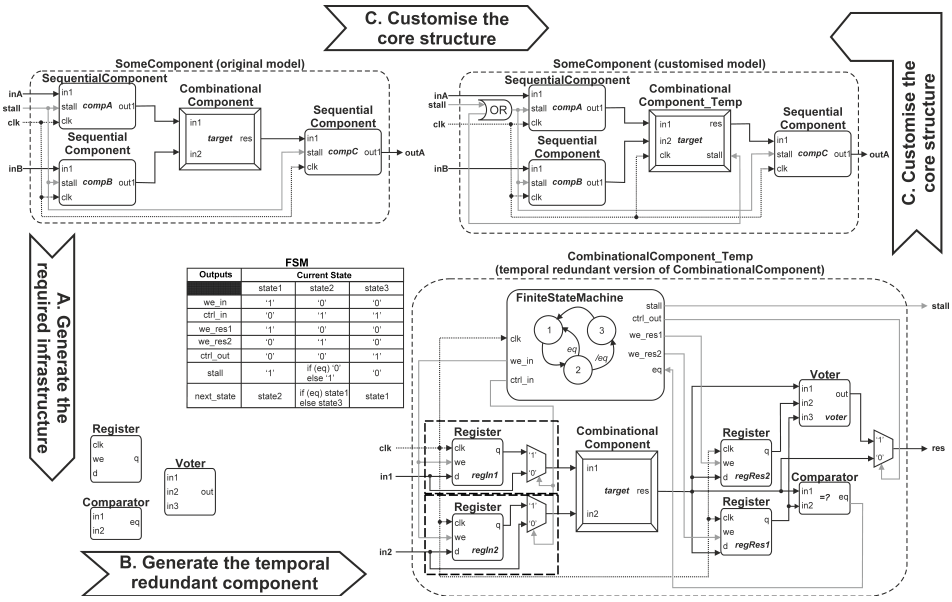


Figure D.4: Metaprogramming temporal redundancy

Generating the required infrastructure

Adapting an existing component to exhibit temporal redundancy capabilities involves the generation of some generic infrastructure to store and compare intermediate results (see Figure D.4a). As demonstrated in [144], the generation of voting infrastructures (*voters*) can be accomplished via metaprogramming. Accordingly, and following a similar approach, the generation of generic *comparators* can be easily accomplished. Finally, [143] showed how generic pre-designed *registers* can serve as temporal data storage. All these elements are kept in the CODESH library once generated. Hence, as many different instances of these components as required can be used to enhance the selected target component with temporal redundancy capabilities.

Encapsulating the temporal redundancy mechanism

Providing temporal redundancy relies on processing the incoming inputs a given number of times in order to compare the different outcomes obtained and determine the correct one when discrepancies appear. Figure D.4b depicts all the logic required and its interconnection to a given component, in order to expand the non-functional aspects of that component with temporal redundancy capabilities. It is to note the importance of the component's nature when applying that kind of approach. The outputs provided by a *combinational* component only depend on its current inputs. Accordingly, it is necessary to feed the combinational component with the same inputs for each required re-execution. However, when dealing with a *sequential* component, the obtained outputs not only depend on its current inputs, but also on its current state. Hence, when in a black box approach, sequential components should provide *intercession* capabilities to enable the system to obtain its current state (*getState*) and restore it later (*setState*) for each re-execution. In case those intercession capabilities are absent from the component's interface, it will not be possible to deploy the desired temporal redundancy mechanism. Then, this constraint must be taken into account when designing sequential components that could be considered as candidates for temporal redundancy techniques. If a white box approach is used, intercession could be implemented by inserting in the output of each register a structure as the one surrounded by thick dashed lines (register + multiplexer) in Figure D.4b. As this technique can always be applied to combinational components, the description of the proposed approach will consider that kind of component as target. Following the approach commonly used in the design of digital circuits, all the logic required to control the previously described infrastructure and the different re-execution stages is implemented by means of a FSM. This FSM will comprise a number of states equal to the number of re-executions (usually an odd number). In this example, the FSM consists of *three* (3) different states, and the activation of the signals it controls for each state is listed in the table depicted in Figure D.4b. As FSM are synchronous elements that

require a clock signal in order to activate the transition from one state to another, when dealing with combinational components it will be required to include a *clock input port* to the interface of the new customized component. During the first state incoming inputs are registered, so the following re-executions may use these very same data for their operation. Hence, each input of the target component is associated to a *register*. In this first state, the target component directly operates with these incoming data, not the registered ones, to avoid delaying the operation one clock cycle. The outcome of the computation is also stored in another *register* for later use. As several re-executions will take place after this first one, the rest of the system should suspend its normal operation to wait for the right outcome to be provided. Hence, a new *stall output port* should be included to the interface of the customized component. This signal is now activated by the FSM, which proceeds to its second state. From the second state and on, the target component must operate with the data stored in the incoming registers. Hence, the FSM activates the control signal of those multiplexers in charge of feeding the target component. The outcome of this operation is also stored on a dedicated *register*. Once more than half of the required re-execution have already been performed (3 for 5 executions, 4 for 7 executions, etc.), it is possible to optimize the performance of this component: if all operations have provided the very same result, it is not necessary to continue with the rest of re-executions, since this will be the right outcome for the considered inputs. Accordingly, a *comparator* is used to determine whether all the previously stored outcomes and the current one are equal. If this is the case, the current outcome is passed onto the outputs of the component, the *stall* signal is deactivated so the system may resume its normal operation, and the FSM proceeds to its first state again. Otherwise, the FSM proceeds to its next state (third state in the figure). Upon reaching the last (third in the example) state, the last re-execution is performed. As it is not necessary to store the outcome of this operation, a *register* is not required. This outcome, along with those previously stored, are fed to a *voter* to determine the correct output of the component. Finally, the FSM deactivates the *stall* signal to allow the system to resume its operation, and proceeds to its first state. In summary, and following this reasoning, to re-execute n times the operation of a given component with k inputs, a FSM with n states will be in charge of controlling: k registers to store the original inputs, *n-1* registers to store the computed outputs, *k+1* multiplexors, an *(n+1)/2* comparators to optimize the performance of the component, and an *n-inputs* voter. In addition, a *stall* signal will control the execution flow of the rest of the system to wait for the correct outcome. A metaprogram is in charge of deploying and interconnecting all these elements to generate the temporal redundant version of the target component.

Inserting the new component into the original model

The inclusion of the new (customized) component into the original model is usually quite straightforward, just replacing the original component by an instantiation of this new component. However, several other considerations must be taken into account to deploy a temporal redundancy strategy (see Figure D.4c). In first place, the clock signal must also be connected to the *clock input port* of the customized component. This should not pose any problem, except when the original component (SomeComponent in the figure) is combinational (asynchronous). In this case, an input clock port must also be added to the interface of that component, which should be connected to the clock signal of the component on the next (higher) level of the hierarchy. In case the whole system exclusively consists of asynchronous logic, this fault tolerance mechanism cannot be applied. The second problem appears when considering that the new component may suspend the execution of the rest of components to synchronize them with the computation of the correct output. The best situation is faced when all components have been designed with a *stall input port*. In this case, the current connection of each stall input port may be OR-ed in order to be activated by either the normal flow of the execution or the customized component. This is the approach shown in Figure D.4c. If this stall port is missing in any of the components, things get a bit more complex. One possible solution is determined by the existence of a *write enable input port* in these components. If they are designed in such a way that disabling that signal prevents the contents of *all* their registers from being updated, then this write enable port may be also AND-ed with the *NOT stall* signal. Otherwise, the last resort is to apply *clock gating* techniques [44]. In this case, *clk AND NOT stall* is added to the clock tree to prevent those components from receiving a clock edge, thus keeping their state until the stall signal is deactivated. This is the less desirable method because of possible race conditions generated in the path. As an example of how all these considerations are taken into account to customize the original model of the system, Table D.1 specifies, in a Java-like pseudo code, the metaprogram interface that implements this approach (Table D.1a) and all the required transformation rules (Table D.1b).

D.4.2 Black box IP cores: integrating third party cores for symmetric encryption

An IP core providing symmetric encryption [16] has been selected to show the feasibility of using metaprogramming to enhance the non-functional capabilities (security) of a given design and ease the integration of the core in further designs. Let us assume that the outgoing data of very same component previously studied should be encrypted in order to ensure its privacy. Figure D.5 shows the generation and customization processes that should be applied to the original model to deploy

Table D.1: Metaprogram interface (a) and transformation rules (b) required to insert the new component into the original model (customize the core structure)

A. Model customisation metaprogram

```
interface Model_Customisation {

  //Initialisation of local variables

  ComponentDeclaration handleComponenDeclaration(
    ComponentDeclaration cd
  ){
    if (cd.getName().equals(this.targetComponentName) {
      // Generate the new customised component declaration
      ComponentDeclaration newCD =
                Generation_Rule_g1(cd);
      //Add clk and stall ports to the new component
      Customisation_Rule_c1(
        newCD, "clk", Mode.IN, Subtype.STD_LOGIC);
      Customisation_Rule_c1(
        newCD, "stall", Mode.OUT, Subtype.STD_LOGIC);
      // Append the customised component to the
        list of components
      bdiList.append(newCD);
    }
    return(cd);
  }

  ArchitectureDeclarativePart
    handleArchitectureDeclarativePart(
    ArchitectureDeclarativePart adp
  ){
    BlockDeclarativeItem[] bdiList = adp.getList();
    // Generate the auxiliary stall signals declaration
    SignalDeclaration newSD =
      Generation_Rule_g2({"stall_temp", "stall_ored"});
    // Append the auxiliary signals to the list of signals
    bdiList.append(newSD);
    return(adp);
  }

  ComponentInstantiationStatement
    handleComponentInstantiationStatement(
    ComponentInstantiationStatement cis
  ){
    if (cis.getLabel().equals(this.targetComponentLabel)) {
      // Map the new ports in the target instantiation list
      Customisation_Rule_c3(cis, "clk", "clk");
      Customisation_Rule_c3(cis, "stall", "stall_temp");
      // Modify target instance for new component
      Customisation_Rule_c4(cis);
    }
    else {
      // Modify the stall mapping
      Customisation_Rule_c2(cis, "stall", "stall_ored");
    }
    return(cis);
  }

  ArchitectureStatementPart
    handleArchitectureStatementPart(
    ArchitectureStatementPart asp
  ){
    // Generate the stall signal assignment statement
    SignalAssignmentStatement newSAS =
      Generation_Rule_g3(
        "stall_ored", "stall", "stall_temp", LogicOperator.OR);
    // Append the new assignment to the concurrent
      assignments list
    asp.getConcurrentStatementList().append(newSAS);
    return(asp);
  }
}
```

B. Required metaprogram rules

```
ComponentDeclaration Generation_Rule_g1 (
  ComponentDeclaration cd,
) {
  //Generate a clone of the target component
  ComponentDeclaration newCD =
    new ComponentDeclaration(cd);
  newCD.setID(newCD.getID() + "_" + "Temp");
  return(newCD);
}

void Customisation_Rule_c1 (
  ComponentDeclaration cd,
  string pn, // Port name
  Mode pm, // Port mode
  Subtype ps, // Port subtype
) {
  cd.getPortClause().AddPort(
    new InterfaceSignalDeclaration(new Identifier(pn,pm,ps));
}

SignalDeclaration Generation_Rule_g2 (
  string[] snList, // Signal name list
) {
  //Generate a new signal declaration
  SignalDeclaration newSD = new SignalDeclaration(
    new IdentifierList(snList), Subtype.STD_LOGIC);
  return(newSD);
}

void Customisation_Rule_c2 (
  ComponentInstantiationStatement cis,
  string fp // formal part name
  string ap // actual part name
) {
  AssociationElement ae =
    cis.getPortMap().
      FindAssociationElementByFormalPartName(fp);
  ae.getActualPart().setName(ap);
  }

void Customisation_Rule_c3 (
  ComponentInstantiationStatement cis,
  string fpn // formal part name
  string apn // actual part name
) {
  cis.getPortMap().Add(
    new AssociationElement(fpn, apn));
  }

void Customisation_Rule_c4 (
  ComponentInstantiationStatement cis,
) {
  cis.getInstantiatedUnit().setName(
    cis.getInstantiatedUnit().getName() + "_Temp");
  }

SignalAssignmentStatement Generation_Rule_g3 (
  string tn, // Target name
  string se1, // Simple expression name
  string se2, // Simple expression name
  LogicOperator lo,
) {
  //Generate a new signal assignment statement
  SignalAssignmentStatement newSAS =
    new SignalAssignmentStatement(
      tn, new Expression(se1, se2, lo));
  return(newSAS);
}
```

the requested data encryption mechanism provided by the third party IP core. All the rules devised for the automatic integration of such core are next described.

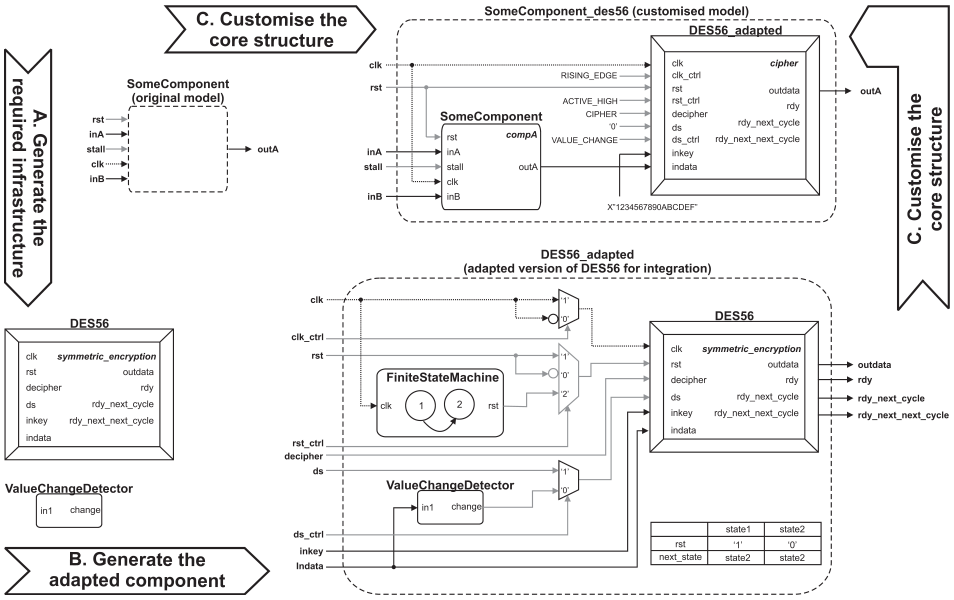


Figure D.5: Metaprogramming the integration of a third party core providing symmetric data encryption into a given model

Generating the required infrastructure

As Figure D.5a shows, the main infrastructure required is just the IP core to integrate into the system. This is somewhat different from what happens when adding redundancy mechanisms to a target core, which usually require lots of other elements to implement the desired non-functional aspect. In this case, the symmetric encryption mechanism is already implemented by the selected IP core and, hence, very little additional infrastructure is needed.

Encapsulating the third party core

The usefulness of a third party core, in addition to its intended functionality, relies on its ability to be easily introduced into any kind of systems. Therefore, analyzing and enhancing the interface of a given IP core may contribute to ease its reusability and extend its context of use. For instance, the following assertion may be found in the documentation of the selected symmetric encryption core: "The reset signal is used to set all internal signals to a known state and prepare the

core for operation. It should be strobed high at least once after power on and before attempting the first cryptographic operation." [16]. Accordingly, the system integrator should take it into account to feed the reset signal of that component with the expected high logic level. However, it could happen that the original reset signal of the system is active low, or even worse, that it consists of pure combinational logic and so no reset signal is available. In any case, it seems highly convenient to provide the third party core with a parameter (*rst_ctrl*) that can be used to control whether this component will receive an active high or low reset signal, or it should internally generate a reset to initialize the core. That is the purpose of the 3-to-1 multiplexer and the finite state machine depicted in Figure D.5b. This new control signal will prevent system integrators from forgetting to properly feed the reset signal of the third party core, as no input maybe left unconnected and only meaningful values like *ACTIVE_HIGH*, *ACTIVE_LOW*, and *RST_NOT_AVAILABLE*, are accepted. Something similar can be said about the clock signal. In this case, no information can be found on the documentation stating whether the symmetric encryption core operates on rising or falling edges. After checking the core's behavior (*black box* approach), and determining that it operates on rising edges, a 2-to-1 multiplexer has been included to take care of passing the right edge to the core. As in the previous example, it will be mandatory to activate the related control signal (*clk_ctrl*) with one of the two eligible values (*RISING_EDGE, FALLING_EDGE*). Another interesting case is related to the *ds* signal. According to the documentation: "the DS signal is the data strobe. When momentarily strobed high, it indicates the input data set is valid, and signals the core to start a cryptographic operation. Only the rising edge of this signal has meaning: all other states are ignored." [16]. A singular problem arises from the fact that, probably, the system integrating this core does not present any *ready* signal in charge of notifying when new data is available on the output port. One possible solution consists in making use of another component (*ValueChangeDetector* on Figure D.5b) that provides a rising edge whenever the data to be cyphered/deciphered changes. Accordingly, the *ds_ctrl* signal will control whether the *ds* input (*DS*) or the *ValueChangeDetector* (*VALUE_CHANGE*) will trigger the encryption/decryption operation. The development of metaprograms to wrap third party cores in the required additional infrastructure may not only help system integrators, but will enable other metaprograms to automatically deploy these cores into the target model.

Inserting the new component into the original model

Once encapsulated, this customized core may be easily integrated into the original model. As this component was not present in the model, it is necessary to include its declaration (interface) and insert a new instantiation to establish the connections among its input/output ports and the rest of signals and ports of the system. In the example depicted in Figure D.5c this is quite simple, since the data to be ciphered comes from the target component, and the actual output of the system is now computed by the symmetric encryption core. The clock and reset signals are directly connected to the corresponding core ports. As the original component do not provide any output ready signal, all the rdy output ports of the core are left unconnected (open). The last step is properly parameterizing the control signals that have been added to the customized third party core to ease its interconnection. In this example, the clk and rst signals of the system follow the same specification as noted in the documentation of the encryption/decryption core and, thus, its corresponding control signals are set accordingly (RISING_EDGE and ACTIVE_HIGH, respectively). As the component must encrypt the incoming data, the cipher port is set to CIPHER. The target component does not provide any information related to new data being available on its output and, hence, the ds activation signal must be internally generated by checking the incoming data (ds is set to '0' and ds_ctrl to VALUE_CHANGE). Finally, the key used to encrypt/decrypt the incoming data must be passed to the *inkey* port (X"1234567890ABCDEF" in the figure). The model resulting from the application of the generation rules defined in the proposed metaprogram to automatically integrate the symmetric encryption mechanism is depicted in Figure D.6. In this way, the privacy and confidentiality of outgoing data is easily ensured by reusing existing qualified components.

D.5 Analysis of Results and Discussion

A structural HDL model of a PIC (Programmable Intelligent Computer) 16C5X microcontroller [142], whose family is representative of those commonly used in embedded applications, has been selected to show the feasibility of the proposed approach. The generated metaprograms will be applied to that microcontroller to automatically deploy i) a temporal redundancy mechanism to tolerate transient faults in combinational logic, and ii) a symmetric encryption mechanism to ensure the privacy of data sent by its output ports. The rest of this section presents the considered experimental setup, the proposed solution to be automatically applied by the metaprograms, and the analysis of computed results.

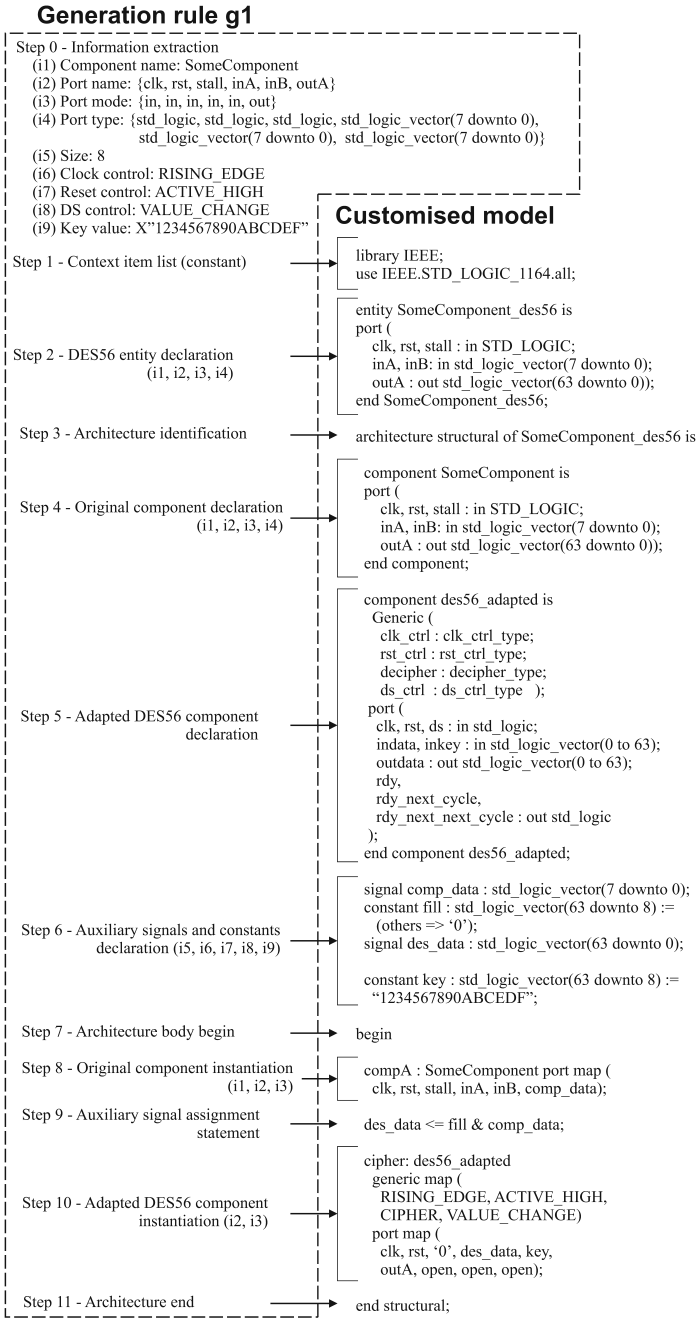


Figure D.6: Metaprogram generation rule required to integrate the symmetric data encryption third party component.

D.5.1 Experimental setup

In order to perform a fair comparison between the original model of the system and the automatically generated fault-tolerant and secure models, the same implementation technology should be used for all of them. Field-Programmable Gate Arrays (FPGAs), which are very useful for prototyping and testing HDL models in the field, seem a good implementation choice since all these models follow a synthesizable VHDL description. FADES [8], an FPGA-based Framework for the Analysis of the Dependability of Embedded Systems, appears as a suitable framework for implementing the models on a programmable device and injecting faults during the experiments execution to analyze the sensitivity and final robustness of the resulting system. The selected workload consists in a *quicksort algorithm*, which could be representative of those control algorithms typically executed, for instance, in automotive applications [33]. Temporal redundancy can deal with transient faults and, as previously stated, this case study specifically focuses on those faults affecting the combinational logic of the system. Hence, the considered faultload comprises *pulses, transient indeterminations*, and *transient delays* [66], with just a single fault being injected in each experiment. The number of experiments is determined according to the silicon area devoted to implement the combinational logic of the target component and, thus, the relative area exposed to the considered faults. The set of considered measures to estimate the robustness of the system includes the percentage of faults being *masked*, remaining *latent* or leading to a *failure*. Other measures related to cost or performance are the *silicon area, clock frequency* and *energy consumption*. The detailed process of how to extract all these measures from the observed measurements is described in [6]. In a first stage, the design is implemented and prototyped, and faults are injected using the aforementioned fault models to obtain a sensitivity analysis of each available component. This initial study revealed that the most suitable candidate to improve its dependability via temporal redundancy is the *control unit*, whereas communications from *input output ports* should be encrypted. Therefore, these are the target components for the proposed metaprograms. Modified designs are subjected to a new fault injection campaign to assess the improvements, proper functioning and overhead of the inserted strategies. The results from this experimentation are reported in next section.

D.5.2 Analysis of results

For each fault model, a number of 330 experiments were injected during the sensitivity analysis of the original control unit, and 850 more were required after its customization. As the goal of the encryption mechanism is to ensure the privacy of the generated data, it makes no sense to inject accidental faults into that version of the system. Dependability/security improvements usually come at a price, in terms of increased silicon area, reduced throughput and increased power consumption. The comparison of these results for the original IP core (*PIC*), the

version enhanced with a temporally redundant control unit (*TR*), and the version encrypting the outputs (*DES*) is presented in Table D.2.

Table D.2: Comparison of the original (PIC), temporally redundant (TR), and secured (DES) cores in terms of failures, area, throughput, and energy consumption.

A. Impact of faults					
IP core	Pulses leading to failure	Transient indeterminations leading to failure		Transient delays leading to failure	
PIC	9.7%	6.6%		0.9%	
TR	3.7%	2.8%		1.3%	
B. Area estimation					
IP core	Number of Flip-Flops	Number of Look-up tables	Number of slices	Equivalent logic count	Area increment
PIC	752	457	629	18362	–
TR	880	720	772	20979	+14%
DES	1006	1002	984	24051	+31%
C. Throughput estimation					
IP core	Clock period (ns)	Number of clock cycles	Execution time (us)	Throughput (executions/s)	Throughput reduction
PIC	33.232	1722	57.225	57.225	–
TR	43.691	3344	146.102	146.102	-61%
DES	33.922	1722	58.413	58.413	-2%
D. Energy consumption estimation					
IP core		Power consumption (mW)	Execution time (us)	Energy consumption (mW·s)	Energy consumption increment
PIC		366	57.225	0.0209	–
TR		257	146.102	0.0375	+79%
DES		399	58.413	0.0233	+11%

As can be expected, the impact of transient faults on the combinational logic of the system (see Table D.2a) is quite low since they can be electrically, temporally and logically masked. Nevertheless, nearly a 10% of pulses and a 7% of indeterminations lead the system to a failure. This rate is reduced to just a 1% in the case of transient delays. After protecting the system with the temporal redundancy mechanism, the occurrence of any transient fault within the control unit is completely tolerated. However, the whole set of logic that has been added in order to deploy this mechanism is not protected against these faults and, thus, the percentage of faults leading to a failure (on the whole) is decreased to just near a 4% and 3% for pulses and indeterminations, respectively. The case of transient delays is somewhat special. As the impact of delays is so low, the large amount of additional unprotected logic included is negatively affecting the robustness of the system, and no benefit can be obtained from this strategy. Moreover, the registers introduced to hold input data may suffer other faults that have not been tested but could add a minor impact in the system. That shows the importance of performing a previous sensitivity analysis to accurately determine which strategies to deploy and which components to target. However, these benefits in terms of dependability and security do not come for free. As can be seen in Table D.2b, and as can be expected, the original core is the one requiring the smallest amount of silicon area for its final implementation. This area, estimated by the number of logic gates required to build and equivalent circuit, increases a 14% when considering the fault tolerance mechanism (additional logic required to store inputs and intermediate results, and to control the re-execution of the operations) and a 31% for the security mechanism (additional logic required to keep tables with information for the next encryption round and to execute the operation in 16 rounds). This somehow estimates the increased cost related to the larger quantity of resources (flip-flops and look-up tables to implement the sequential and combi-

national logic of the system, respectively) required for implementing any of these mechanisms. The number of times the system can execute the selected workload per second (throughput) is also estimated in Table D.2c. The original core presents the highest clock frequency and the lowest execution time, leading to the highest throughput. As could be expected, deploying the fault tolerance mechanism onto the control unit is highly impacting the critical path of the core and, hence, increasing its clock period. Furthermore, as each instruction of the workload must be executed at least twice, the finally obtained throughput is reduced a 61% with respect to the original core. On the other hand, as the third party encryption core has been designed with a 16-stages pipeline structure, its insertion in the original model barely affects the finally obtained throughput (-2%). This variation is not significant and maybe attributed to the non-deterministic implementation process. The energy consumed by the three considered versions is reported in Table D.2d. Once again, the original core obtains the best results, since it requires the smallest quantity of resources to be implemented, and presents the lowest clock frequency and the shortest execution time for the selected workload. The inclusion of the temporal redundancy mechanism reduces the *power consumption* of the core, as half of the time most of its components are *stalled* to recompute the output signals of the control unit. However, the so long execution time counterbalances this result, leading to an increase in the *energy consumption* of a 79%. Although the original core and the version improved with a security mechanism present very similar clock frequency and execution times, the latter consumes an 11% more energy due to the larger amount of physical resources required for its implementation. It must be noted that all these overheads are inherent to the insertion of the different fault tolerance and security mechanisms considered, and are not due to their definition and deployment via metaprogramming. These overheads are very similar to those that can be expected when implementing the very same mechanisms by hand. Finally, experimental results validate the metaprogram-generated implementation and deployment of both mechanisms. Temporal redundancy increases the robustness of the system against transient faults targeting its combinational logic at the cost of greatly reducing the expected throughput and increasing its energy consumption. Symmetric encryption enhances the security (privacy) of the system by using a large number of physical resources for its implementation and slightly increasing its energy consumption. This hinders malicious attacks based on eavesdropping.

D.6 Conclusions and Open Challenges

Increasing integration scales, time to market pressure and the use and re-use of third party cores are greatly increasing the likelihood of occurrence of faults in hardware embedded systems. Although once reserved for safety-, mission-, and business-critical systems, fault tolerance and security strategies are nowadays a requirement even to consumer electronics. Accordingly, both academia and industry are currently moving towards the provision of tools for automating the implementation of fault-tolerant and secure components and their subsequent deployment in hardware systems. This chapter has shown that aspect orientation concepts, which have been successfully used for software development, can also be applied to hardware development. The most common approach to support the separation between functional and non-functional concerns in hardware design is based on metaprogramming and open compilation. In concrete, CODESH, an open compilation process for the design of dependable and secure high-level HDL descriptions, has been used to illustrate how metaprograms could support the design of fault-tolerant and secure hardware by i) developing the required basic infrastructure, ii) encapsulating these elements to define a new fault-tolerant or secure component, and iii) integrating it into the original HDL model. The proposed open compilation approach is seamlessly integrated into the regular hardware design flow, thus enabling hardware designers to apply different fault tolerance and security strategies to any HDL-based design within minutes and avoiding error prone procedures. The feasibility of the approach has been proved through two different case studies, which also showed the importance of properly analyzing the weaknesses of the system so as not to incur in large overheads with negligible benefits. Despite metaprogramming and open compilation provide a highly flexible approach for the generic development and automatic deployment of fault tolerance and security mechanisms, its use requires a deep technical knowledge of HDL, the metalanguage (Java in the case of CODESH), and the API reflecting the structure of the input HDL model. Latest research in this domain is focusing on instantiating all the concepts related to aspect orientation, like join points, advices and pointcuts, in the domain of HDL. In this way, common HDLs could be extended to support the definition of fault tolerance and security mechanisms as aspects using the same language hardware designers usually employ. The applicability of this methodology may also be hindered by the openness (white box and black box) of the considered models. *Soft cores*, with a white box approach, can usually be analyzed and modified as desired and, thus, it is possible to add the logic needed to implement the intended functionality. However, performing this analysis to understand the implemented functionality, and modify it accordingly, is not always as straightforward as it could seem. The definition of precise design guidelines to help open compilers and metaprograms to reason about the input model and locate the critical infrastructure necessary to implement the desired interface is necessary. This is very similar to hardware design guidelines to help synthesizers to obtain the right circuit from the input HDL model. When dealing

with *hard cores*, with a black box approach, and even under *grey box* approaches, all the implementation details are hidden from the metaprogram and, thus, the responsibility for implementing and properly documenting the required interfaces falls again upon the core designer. This problem could be alleviated by designers adhering to standards supporting the definition of buses for easing the interconnection of cores, and thus their reuse, such as the Advanced Microcontroller Bus Architecture [183], the Wishbone architecture [181], and the OpenCore Protocol architecture [186]. Thus, components could be provided with a common interface allowing their easy interconnection and interrogation about their configuration parameters and capabilities. Another interesting point is related to the composition of mechanisms. Up to now, research has mainly focused on showing the feasibility of using open compilers and metaprogramming to automatically deploy fault tolerance and security mechanisms into a given hardware system but, however, the implications derived from its composition have not been considered yet. Obviously, faults will affect differently a *spatially redundant and secure* component and a *secure and spatially redundant* component. So, it could be very interesting to study the composition of available mechanisms, how to deploy them in an efficient and automatic way, the effect of faults and attacks on these combinations and, also, how they impact the area, throughput and energy consumption of the final system. From this study, and by analyzing all the possible combinations with HDL models considered representative of different application contexts (like automotive, aircraft, or consumer electronics), it could be possible to obtain a rough estimation of the benefits and drawbacks of each combination in each application context. This could assist the open compiler user when deciding which mechanisms to deploy on a given system according to cost, performance, and dependability requirements. Finally, this kind of approaches may be of great interest for the development of adaptive hardware systems. Let us assume, for instance, that to protect a system against faults a *Triple Modular Redundancy* mechanism is deployed, thus increasing the required area for a simple (non-protected) system by around 200% (note again that this overhead is intrinsic to the mechanism and is not due to its metaprogramming). In case a permanent fault occurs, the faulty module is removed from the system and a *Dual System with Comparison* is used instead. Now, when results do not match, they are re-executed once more. This system would increase, with respect to the simple system, the required area by a 100% and the execution time by 100%, but only in presence of faults. In case another permanent fault is detected, the remaining fault-free component can be encapsulated into a *Temporal Redundancy* mechanism, which will increase the execution time by a 200%. If another permanent fault is detected, the system will fail. Although a metaprogram could be developed to handle the deployment of all three mechanisms at once, with the logic required to switch from one mechanism to another, the cost in terms of area, performance and energy consumption will be enormous. However, reconfigurable devices like FPGAs could be used not only as prototyping platforms, but also as the implementation technology for the final system. In this way, the hardware system could change from one defined config-

uration to another as faults occur, but also as dependability, area, performance, or energy consumption requirements change. For instance, if the reconfigurable device is required to implement a given function, the system could move to a reduced area implementation (like the temporal redundancy mechanism) sacrificing performance and dependability in favor of area. Once this additional function is no longer needed, the system can change into a more dependable but otherwise conservative configuration, such as the comparison with detection. Finally, when the device detects that the likelihood of occurrence of faults is high, it may switch into a fully dependable configuration in spite of the area taken. As shown, the powerful automation capabilities provided by metaprogramming and open compilation may pave the way towards the actual use of adaptive resilience mechanisms, whose performance and dependability capabilities could evolve depending on faults, attacks and changes in the operation environment.

Appendix E

Robust communications using automatic deployment of a CRC-generation technique in IP-blocks

Authors: Jaime Espinosa, David de Andrés, Juan Carlos Ruiz and Pedro Gil

Electronic systems manage their complexity and reduce their time-to-market throughout designs integrating off-the-shelf IP cores. As fabrication scales are reduced, interactions between such cores become more sensitive to accidental faults. A typical strategy to cope with such problem is the use of Cyclic Redundancy Checks (CRC). In most of the cases, CRC are manually designed for each system by engineers with limited dependability skills. Consequently, the reusability of resulting mechanisms is limited and their proneness to design bugs is not negligible. This paper investigates the use of metaprogramming techniques to design generic and reusable CRC that can be automatically specialised

through open compilation attending to the particular features of each IP core design. The various parametrisation decisions that must be considered to obtain CRC instances with a good balance between performance and dependability are also reported and experimentally assessed.

E.1 Introduction

Miniaturisation in electronics has enabled the production of more powerful and tiny HW designs whose complexity and time-to-market are handled through the use of off-the-shelf building blocks, named IP cores. However, the barrier of one micron sized gates in integrated transistors was long trespassed and associated negative dependability side-effects have come into play [36]. Today, not only harsh environment or mission critical elements can benefit from fault-tolerant techniques but also consumer electronics which are in the need of robustness to maintain a high quality standard and brand reputation [118].

One important aspect relating to CMOS shrinking is the number and potential impact of accidental faults, such as single/multiple bit-flips and burst errors, in inter IP core digital data communications. Cyclic redundancy checks (CRC) [139, 178] are privileged means to tolerate such faults. Although other fault-tolerant mechanisms such as error-correcting codes can also solve the problem, the information and computational overhead they induce is bigger than the one related to CRC [171].

The problem handled in this paper is how to develop and deploy a CRC strategy to promote their reusability in IP core-based HW designs. The idea is to enable the use of such mechanisms even to designers with limited dependability skills, which are those that are more prone to introduce a bug in the design of fault tolerance mechanisms. This goal yields to consider design mechanisms promoting a clear separation between the functional and non-functional (in our case CRC-based fault tolerance) features of each IP core [145].

To cope with such separation of concerns, the considered CRC strategy has been designed in a generic way as a metaprogram, applying on IP cores' HDL (Hardware Description Language) models a set of analysis and transformation rules to produce specific CRC instances suitable for each IP-core. The reusability of such metaprograms is promoted through the use of open compilation tools [146, 144, 143], which enables the automatic deployment of the metaprograms on concrete IP core models. Finally, fault injection techniques are used to assess the level of fault tolerance attained by resulting HW designs.

The rest of the paper structures as follows. Section E.2 presents the context of this research. Section E.3 reports on the design of a CRC metaprogram, which

is later deployed on real case study in Section E.4. Results and conclusions are finally detailed in Sections E.5 and E.6.

E.2 Research context

This section presents the basic notions about CRC mechanisms, metaprogramming of dependability mechanisms and their automatic deployment using open compilation.

E.2.1 CRCs and fault tolerance

CRC fitted transmitters send original data with extra bits calculated from applying a generator polynomial. In the receiver an identical calculation is performed to the original+extra bits received to obtain a match signal indicating no bit has been altered. If the result is negative, the packet has to be retransmitted.

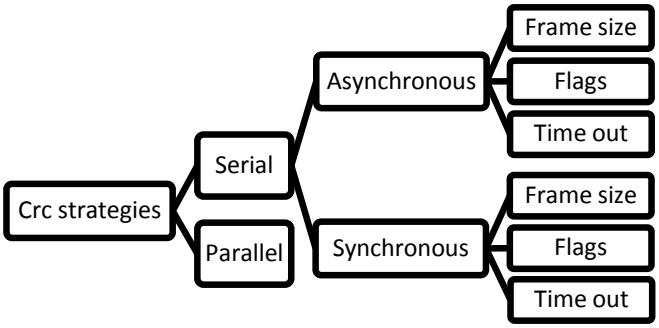


Figure E.1: Transmission CRC strategies

Different transmission strategies are candidate to include CRCs (see Figure E.1). They are classified into serial or parallel i/o structures. Further on, serial structures comprise synchronous or asynchronous modes, depending on whether a clock line is transmitted alongside the data line. Likewise, control blocks can command checksums to be sent following, among others, i) a frame size strategy, which transmits the checksum after a fixed number of data words, ii) a flags strategy, which transmits the checksum after an end-of-frame flag, or iii) a time-out strategy, which transmits the checksum after a fixed period of inactivity.

The degree of resilience provided by the mechanism will depend on the generator polynomial [171, 89]. Koopman studies the theoretical properties of a huge set of polynomials in [99], and highlights the importance of parameters like *Hamming Distance* (HD) -minimum amount of bits to be changed to miss detection for a given message size-, and *Hamming Weight* (HW) -number of combinations for a

specific amount of wrong bits in a fixed message size which would go undetected. The first $HW \neq 0$ is the HD for that message.

Balancing the error detection capabilities and data overhead of a CRC, in different contexts of use, will require the parametrisation of the polynomial in use and the message size.

In the root of this paper, our research is focused on deploying CRC detection mechanisms on serial asynchronous frame size based components. There are some noteworthy constraints to be taken into account: i) the serial asynchronous frame size strategy has been chosen due to its simplicity and common use in UART/USART protocols, but an extension to include other strategies is present in our roadmap, ii) only the transmitter is implemented, iii) an enable input signal must be activated every clock cycle data has to be sent, and iv) no start/end bits are supported yet. The aforementioned mechanism will be defined as a metaprogram.

E.2.2 Metaprograms and open compilation

Open compilation approaches enable the application of program source to source transformations. Those programs driving such transformation process are called *metaprograms*.

For the sake of illustrating the concepts of open compilation and metaprogramming with a concrete example, let us consider the *CODESH* tool, which provides *an open COmpilation process for the design of DEpendable and SEcure Hdl-based systems*.

The workflow of this tool is depicted in Figure E.2. First, the comments grammar parses the input file extracting any CODESH commands and parameters it may contain. After that, the syntax grammar parses the code and creates a structured representation -*abstract syntax tree (AST)*- of that file. The AST is then walked through and the identified language statements are sent to the selected metaprogram, which analyses and transforms them according to its parametrisation and code. If no specific transformation is defined, then the statement is not modified. CODESH embeds by default a neutral metaprogram providing no transformations to input HDL models. Those metaprograms in the *Metaprogram Library* change such default behaviour, by instrumenting HDL models to deploy non-functional (in our case fault tolerance and security) strategies. For each statement, the metaprogram applies the required transformations and returns the modified AST version of the statement to the open compiler, which translates it back to the considered model. This is basically how the metaprogram is able to produce a fault-tolerant/secure version of an input HDL model using open compilation techniques. Next section describes how to metaprogram a CRC strategy.

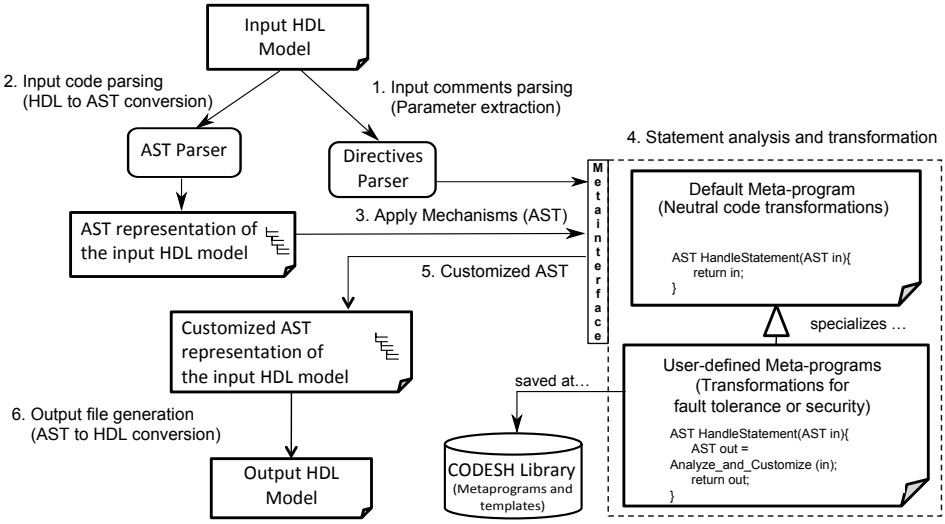


Figure E.2: CODESH workflow

E.3 CRC as a metaprogram

The general procedure designers should follow to develop new fault tolerance/security strategies as metaprograms for CODESH comprises three different phases: phase 1) generating the required infrastructure, phase 2) generating the new protected component by encapsulating the original component and the generated infrastructure, and phase 3) integrating the protected component into the given design. Figure E.3 details the model of the considered input component (see *Original component*).

E.3.1 Phase 1: Infrastructure generation

Following a bottom-up process, the deployment of a CRC mechanism first requires the generation of some basic infrastructure. As depicted in Figure E.3, the creation of a CRC-protected component involves the development of a suitable CRC generator block, a custom control block, and some combinational logic.

Reuse of tested blocks is a convenient and robust design technique and, thus, mature serial and parallel structures have been picked for the CRC generator. These blocks have been downloaded from the open source OpenCores website [125] and included in the CODESH library, however any valid implementation can be used.

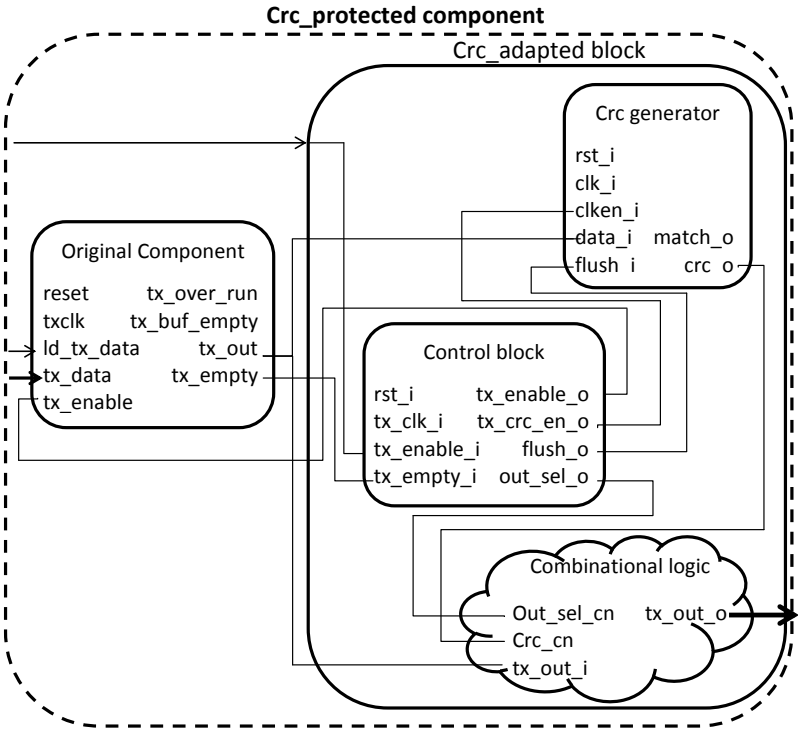


Figure E.3: A CRC-protected block structure showing relevant interconnections

Conversely the control block, which is in charge of generating control signals to drive the CRC generator block and the output combinational logic, greatly depends on the kind of transmitter being protected. This is why it must be custom generated during the deployment of the CRC strategy. CODESH provides two different paths for metaprograms to generate custom blocks: i) starting them from scratch, or the chosen ii) using pre-designed templates and customise them attending to the input commands. New metaprogram rules may later adapt the control block to accept start/stop bits or different transmit enable schemes. Combinational logic to control output is also custom generated.

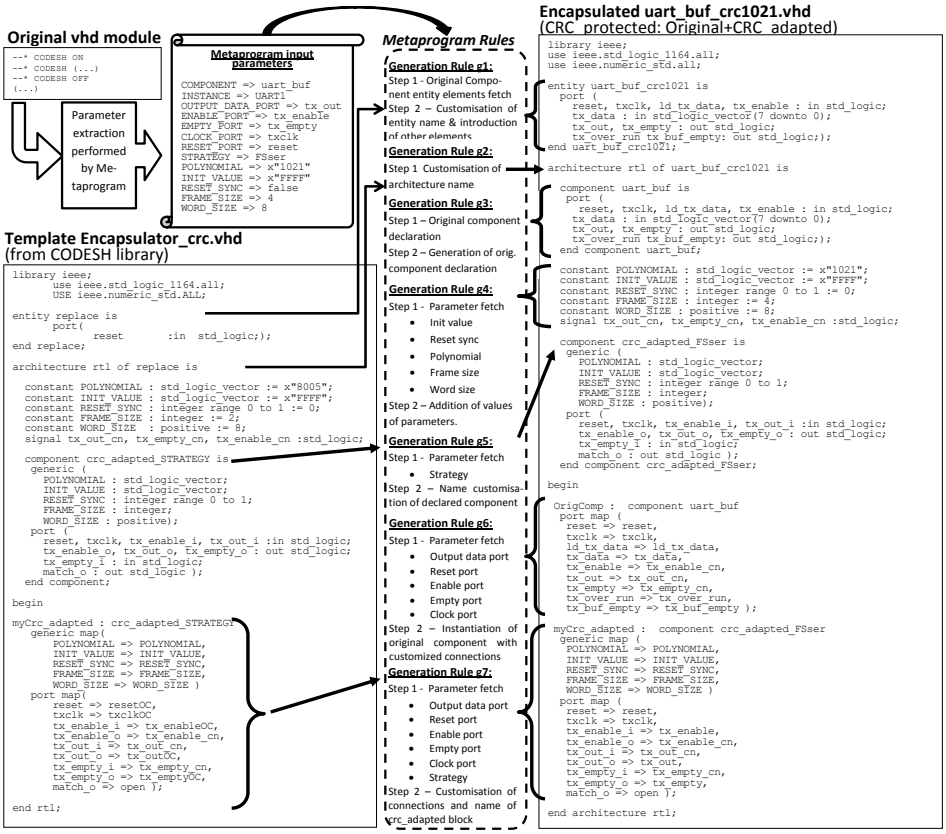


Figure E.4: Phase 2, encapsulation of the new CRC-protected component using template

E.3.2 Phase 2: Component encapsulation

After generating the required infrastructure, it must be encapsulated with the original component into a new CRC-protected component following the schema depicted in Figure E.3.

The CRC generator, custom control block and combinational logic are first joined within a new *CRC_adapted* block. Then, a new *CRC_protected* component is generated, for the *CRC_adapted* block to process the output of the original component. The encapsulation of these elements is performed using a set of rules as shown in the example in Figure E.4.

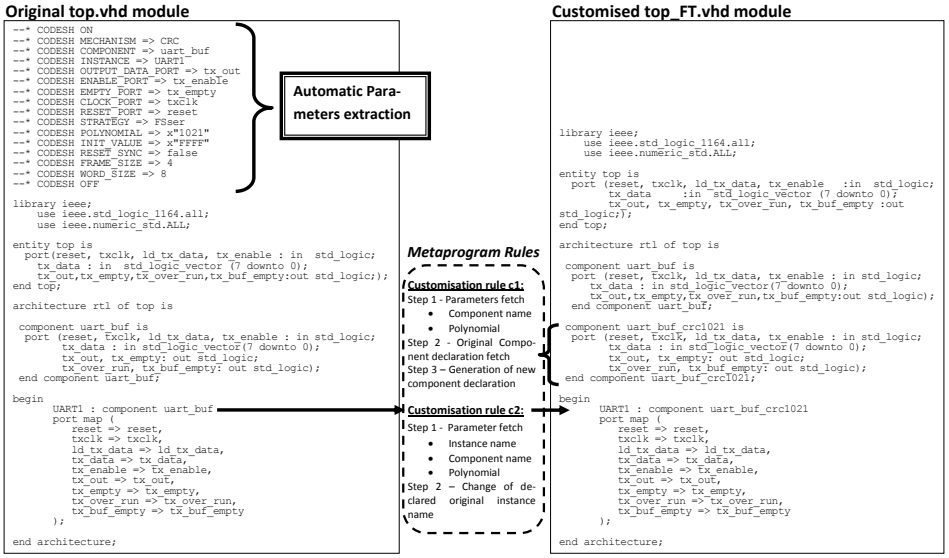


Figure E.5: Phase 3, integration of the CRC-protected component into the target system.

E.3.3 Phase 3: Component integration

Finally those instances of the original component to be protected must be replaced by instances of the *CRC_protected* component obtained in the previous phase. Likewise, the new component declaration must be included in the affected component. This procedure is represented in Figure E.5.

E.3.4 Bridging mechanism deployment and VHDL coding

In order to simplify the deployment of the presented analysis and transformation processes, the behaviour of CODESH is parametrised by inserting specific tags into the HDL code to provide CODESH with all the required information. So as not to interfere with synthesizers, these tags are formatted as special HDL comments (see Figure E.5, Automatic parameter extraction). The required information is the name of relevant ports, strategy of implementation, generator polynomial, initialisation value, type of reset, and additional strategy-dependent parameters.

An example of application is shown in Figure E.5, where the original HDL module includes all the information required to make the output of a given component CRC-protected.

E.4 Case study

This section shows how the proposed approach applies on the transmitter section of a simple UART providing serial asynchronous communication capabilities. Next sections describe the cores under study, the selected test bed and its parametrisation.

E.4.1 CRC-protected UART transmitter

The UART transmitter, downloaded from OpenCores [125], was modified to i) introduce a buffer acting like a FIFO in full-featured UARTs to avoid wasted clock cycles, and ii) disable any start/stop bits until they can be handled by the mechanism. It fits the model presented in Figure E.3 as *Original component*.

Thanks to CODESH, and once the metaprogram was developed, the generation and deployment of the CRC mechanisms for protecting the transmitter was accomplished in no time with negligible effort. Hence, different CRC-protected UARTs were automatically generated to check the correctness and robustness of resulting solutions.

Selected data sizes were those of commercial standards like XMODEM or CAN, CRC sizes were those of XMODEM, ISDN or USB header packets, and polynomials were chosen after Koopman [99]. The considered configurations are listed in Table E.1.

The target system was a protected component transmitting a given frame to a receiver which featured the very same CRC to compute the check once the frame was received.

E.4.2 Faultload

The resilience capabilities provided by the deployed mechanism can be determined by observing the behaviour of the system in the presence of a representative set of faults. The number of experiments was determined to obtain an statistical quality of a 95% as stated in [172]. Table E.1 list the number of experiments carried out and the considered fault models.

Single and multiple bit and burst faults are considered as fault models. Single bit faults represent value switches in the serial output that may affect any bit in a packet. Burst faults were emulated by: i) flipping all burst bits, and ii) fixing all burst bits to a given value. Burst length was fixed to 16-bits for the purpose of the present experimentation. Finally, multiple bit faults were injected to check i) the HD claimed by the polynomials and ii) the announced HW. For the former all possible combinations of *HD - 1* bits of the packet should be considered, thus

Table E.1: Number of experiments for the selected configurations

Data size (bits)	CRC size (bits)	Polynomial (hexadecimal	Fault model		
			Single bit	Burst	Multiple bit
1024	16	x"A7D3"	3115	3067	-----
	8	x"D5" | x"39"	3091	3043	10000 |1000
	5	x"05"	3082	3034	-----
64	16	x"90D9"	239	191	-----
	8	x"9B"	215	167	-----
	5	x"05"	206	158	-----
16	16	x"2D17"	95	47	-----
	8	x"2F" | x"39"	71	-----	7888 | 1826
	5	x"05"	62	-----	-----

yielding a sheer amount of experiments to be computed. Hence just 16-bits data / 8-bits CRC configurations, requiring a smaller number of experiments, were chosen to demonstrate the importance of selecting a suitable polynomial. For the latter, 1024-bits data / 8-bits CRC configuration were selected, as polynomials present equal HD with considerably different HW. In this case, a limited number of runs was carried out due to computational limitations.

E.4.3 Experimental procedure

For each configuration, a fault-free experiment were carried out and results were used as comparison reference. Then, fault injection experiments were performed using the VFIT tool [65] the number of times established in table E.1 following a uniform random distribution in time.

Results provided by fault injection experiments where classified attending to three different failure modes: i) *data corruption*, if the CRC mechanism correctly detects that the received data do not match the original payload, ii) *checksum corruption*, if the mechanism detects a problem in the received packet and asks for a retransmission (data was valid but the CRC was not), and iii) *missed corruption*, if the CRC mechanism incorrectly signals a right check after receiving corrupted data.

E.5 Results and discussion

As shown in Table E.2, and as could be expected, 100% of single fault were correctly detected.

Table E.2: Results for single bit faults

Data size (bits)	CRC Size (bits)	Data corruption	CRC corruption	Missed corruption
1024	16	98,39%	1,61%	0,00%
	8	99,13%	0,87%	0,00%
	5	99,35%	0,65%	0,00%
64	16	82,01%	17,99%	0,00%
	8	90,23%	9,77%	0,00%
	5	94,66%	5,34%	0,00%
16	16	63,16%	36,84%	0,00%
	8	76,06%	23,94%	0,00%
	5	88,71%	11,29%	0,00%

Although all selected CRC sizes were fit to detect single bit faults, bigger sizes increase the probability of faults affecting the CRC and thus provoking spurious retransmissions (CRC corruption causes right data to be discarded). This increases the transmission delay in addition to the overhead introduced by the CRC section of the packet. A further analysis reveals that the percentage of CRC corruptions is strongly dependent on the ratio between data and CRC sizes. Figure E.6 illustrates that, for reasonable spurious retransmission rates, this ratio has to be kept moderately low.

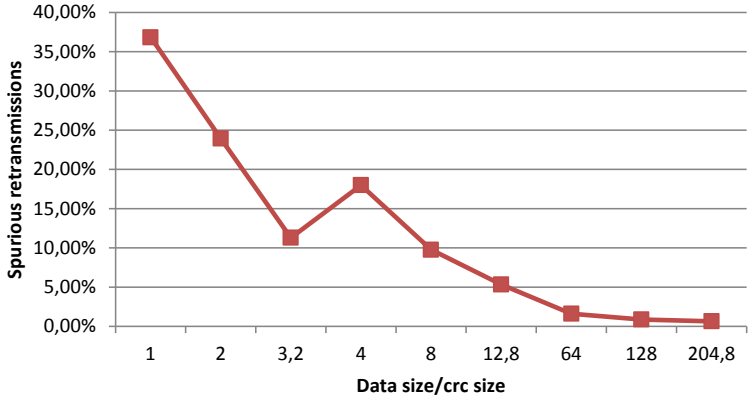


Figure E.6: Relation between the spurious retransmission rate and the data/CRC size ratio

Burst faults are fully detected by CRCs with the same length as the burst. For longer bursts, the probability of missed detection is 2^{-g}, where g is the degree of the polynomial. So, as shown in Table E.3, it is not easy to find a burst combination which misses detection.

Table E.3: Results for 16-bits burst faults

Data size (bits)	CRC Size (bits)	Data corruption	CRC corruption	Missed corruption
1024	16	99,84%	0,16%	0,00%
	8	100,00%	0,00%	0,00%
	5	99,57%	0,00%	0,43%
64	16	96,86%	3,14%	0,00%
	8	100,00%	0,00%	0,00%
	5	96,84%	0,00%	3,16%
16	16	91,49%	8,51%	0,00%

Although no 16-bits burst led to a missed corruption for 8- and 16-bits CRC, both burst fault models caused missed corruptions for 5-bits CRC. Hence, as the probability of undetected bursts is only related to the CRC size, if frequent long burst faults are expected the use of longer CRCs together with shorter data is strongly recommended.

Two different polynomials have been considered to analyse the HD for sparse multiple bit faults. Results listed in Table E.4 demonstrate that 0x2F correctly detects any combination of up to 3 erroneous bits, whereas only single bit faults are detected by 0x39 (*good* and *bad* polynomials according to [99]).

Table E.4: Results for multiple bit faults: HD

Data/CRC size (bits)	Polynomial (hexadecimal)	Detected faults		
		2-bits	3-bits	4-bits
16 / 8	x"2F" (HD=4)	100,00%	100,00%	99,80%
	x"39" (HD=2)	99,52%	100,00%	-----

Table E.5: Results for multiple bit faults: HW

Data/CRC size (bits)	Polynomial (hexadecimal)	Hamming weight	Missed 2-bits faults	
			Expected	Experimental
1024 / 8	x"D5" (HD=2)	5214	0,995%	0,96%
	x"39" (HD=2)	30810	5,882%	6,00%

The HW importance has been studied using two polynomials with a poor HD = 2, but with a 5.9 ratio between their HW. This translates to a missed detections ratio of nearly 6. Experimental results listed in Table E.5 present a fairly good match, although only a 10% of the required number of experiments were performed due to computational constraints.

Experiments have demonstrated i) the correctness of the transformations applied to obtain a CRC-protected component, and ii) the importance of choosing a good polynomial to improve detection capabilities at no extra cost.

E.6 Conclusions

This paper proposes a new approach to develop generic and reusable CRC protection mechanisms as metaprograms. Open compilation techniques were required to automate the deployment and adaptation of such mechanisms attending to the particular features of each component. Fault injection techniques are finally necessary to check the correctness of, and assess the level of dependability finally attained by metaprogram deployments.

As discussed, the reuse of metaprograms eases their debugging and testing, thus making generated mechanisms less prone to incorporate design bugs. Likewise, automating the generation of CRC instances has a very positive impact on i) the time-to-market and cost of resulting products, ii) the exploitation of the approach by non-skilled engineers, and iii) the reduction of the effort required to produce and evaluate different alternatives attending, for instance, to existing parameters and their impact on the final system performance, dependability, power consumption and cost.

A subset of serial asynchronous communication schemes has been chosen to show the feasibility of the proposal. The fault tolerance capabilities of the automatically generated CRC instances, and the impact of data and CRC sizes, and generator polynomials, has been studied using a real design in the presence of different types of faults. This opens the door to design adaptive schemes of protection, which may select the right configuration on the fly among those previously generated by the tool, in response to environmental changes. It is also important to extend the coverage of the strategy to support more transmissions schemes, such as those with other serial asynchronous control protocols and those with synchronous and parallel transmitters.

Appendix F

Towards Certification-aware Fault Injection Methodologies Using Virtual Prototypes

Authors: Jaime Espinosa, David de Andrés, Juan Carlos Ruiz, Carles Hernández and Jaume Abella

Safety-critical applications are required today to meet more and more stringent standards than ever. In the need of reducing the costs associated with the certification step, early robustness evaluation can provide valuable information, as long as it is fast and accurate enough. Microarchitectural simulators have been employed for testing reliability properties in several domains in the past, but their use in the process of robustness verification of safety critical systems has not been validated yet, as opposed to RTL or gate-level simulations. In the present work, we propose a methodology to improve the accuracy of fault-injection re-

sults when targeting robustness verification, by using microarchitectural simulators and virtual prototypes for an early estimation of deviations with respect to the certification standards.

F.1 Introduction

To cope with the increasing functionality demands coming from the safety-critical systems industry processor designs offering more computation power are required. However, the growing complexity together with the requirement for adhering to certification standards causes processor designs targeting safety critical markets rarely achieve reduced time to market. In that context, new verification and test methodologies and tools have to be devised for a quick and cost-effective way to check whether robustness properties are achieved throughout the whole design flow on top of complex processor designs.

Simulation-based fault injection is regarded as a suitable methodology for the robustness verification process, as quick and cheap corrections on misbehavior can be made. Unfortunately, fault injection experiments are often carried out at gate level, and so the testing process becomes excruciatingly slow. This problem is alleviated when fault-injection is applied at a higher level of abstraction like the Register Transfer Level (RTL). Verification at the RTL level reduces the burden but still results can get too slow for repeated use in the context of complex designs.

Software-based fault injection techniques and in particular fault-injection experiments using virtual prototypes (or microarchitectural simulators) are a potential candidate to reduce the costs associated with the robustness verification process [123][175]. The main benefits of these approaches with respect to RTL-based fault injections are the simulation speed and the possibility to anticipate deviations w.r.t. the safety requirements before having an actual implementation of the system. Note that the effort required to have a virtual prototype [169][104] of the system is much lower than the one required to have an RTL processor description[1]. However, for these approaches to be employed in the robustness verification process of safety critical processors their accuracy must be proven. It is important to mention that fault-injections using microarchitectural simulators are typically restricted to the register file [77][150] and the different memory structures [149][2]. The reason is that the majority of the potential injection nodes that are present at more detailed abstraction levels like RTL or gate-level are missing at this level of abstraction.

In this context, estimating accurate failure rate metrics using virtual prototypes is challenging. On one hand, implementation details of virtual prototypes provided

[1] In fact virtual prototypes are already needed to enable software developers to test their software before the actual processor is shipped.

with commercial tools [169] are typically protected. On the other hand, even if implementation details were available, the existing information in virtual prototypes implementation is reduced in comparison to other detailed implementations like RTL. Taking these facts into account, obtaining failure metrics as the ones required by safety critical systems certification standards [81][134] using virtual prototypes is a complex task. Our hypothesis is that for virtual prototypes to be considered for the robustness verification process, results obtained at this level must be correlated with the ones obtained at lower levels of abstraction like the RTL or gate-level as fault-injection at these lower abstraction levels has already been shown to provide accurate enough results [135].

In this paper we propose a methodology that increases the confidence on fault injection experiments into virtual prototypes by using some key information extracted from more detailed levels of abstraction (say for instance RTL or Gate Level). In particular, we are interested in knowing the likelihood that a fault in any [169] possible processor net, gate, or flip/flop, propagates to the register file, system registers or the different memory structures existing in the CPU virtual prototype. Note that this architectural information represents the minimum implementation details that need to be available in any virtual prototype and visible to the user.

The rest of the paper is organized as follows. Section F.2 reviews the state-of-the art in fault injection related to the certification of safety-critical systems. Section F.3 presents the problem in detail and the proposed methodology towards enabling the use of Virtual Prototypes for injection. The employed tool is presented in Section F.4. Section F.5 shows the preliminary evaluation data and finally, in Section F.6 some conclusions are drawn.

F.2 Related Work

Several techniques exist to perform RTL level fault injection. A widely-used method is the injection in the HDL through simulator commands [86], which works well for most of the fault models described in the literature. In fact, some unconventional fault models such as those involving several injection points –short-circuit, multi-bit injection– can be applied if the more intrusive technique of saboteurs is used [15], but an instrumentation of the model –and the consequent decrease in simulation speed– is entailed.

Fault models representativeness has been validated for logic/RTL levels [66]. On the contrary, for higher abstraction levels like the microarchitectural one, some works have pointed out the difficulties of correlating these results with the ones obtained at the physical level [103]. In [30], a quantitative analysis on the divergences was presented, though restricted to bit-flip models injected in flip-flops.

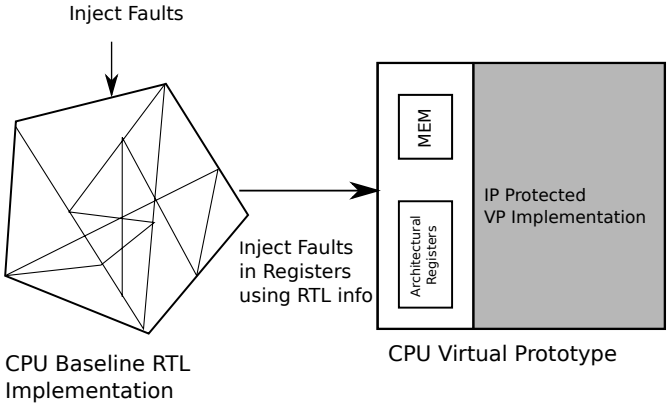


Figure F.1: Proposed methodology

Further analysis on the impact at instruction level of low-level (RTL and gate level) faults was presented in [109]. In that work, stuck-at and bit-flip models are injected to profile higher-level implication, but injections are limited to the control logic and a reduced set of it in the case of gate level.

F.3 Certification-Aware Fault Injection in Virtual prototypes

In this paper we present a methodology that targets performing meaningful fault-injection experiments using virtual prototypes. The proposed methodology consists of two separate steps: (1) a characterization of the fault propagation in an RTL description of the processor, and (2) the actual injection in the virtual prototypes.

F.3.1 Characterizing Fault behaviour at RTL level

We are interested in analyzing the influence of faults in the system towards the incorrect delivery of results, i.e. the appearance of failures, and the moment at which those failures may appear. To characterize fault propagation we propose a methodology that consists of injecting faults in all possible nets of an RTL processor description, using the tool FALLES described in Section F.4.

From the injectable nets we have excluded the register file and cache memory structures due to the following reason: errors occurring within these structures are effectively detected and/or corrected by employing redundancy mechanisms (e.g., error correction codes) and this is the case in most of the processors targeting

safety-critical applications [185][83]. Moreover, available nets in these structures do not realistically represent their area. Memory structures are typically implemented using SRAM cells to minimize area and power and the RTL includes only an instantiation of these components as a black box and/or its behavioral description.

For every fault injection where a net has been forced to a given value, we compare the outputs of architectural registers (general purpose and control registers) and the data and address buses of the core at the on-chip boundaries to the ones obtained with a fault-free simulation. Furthermore, we account for the time elapsed between a mismatch appearing in any of those architectural registers and the subsequent mismatch in the buses which appears in some cases. Gathering this information enables, for instance, detecting the most sensitive registers, which may be ideal candidates for mitigation, or the average time it takes to show erroneous state to the buses, which can be used to determine the maximum detection timespan in lockstep systems [77]. More importantly, we can match it with the information available at the architectural level to increase the accuracy of the injections at such level [53]. In depth, if the propagation information –the number of injected faulty nodes which reflect the wrong value in any architectural register or memory position– is used to inject precisely those nodes of the virtual prototype (see Figure F.1), an increase in the accuracy of the high-level injection should be attained.

F.3.2 Fault injection at Virtual prototypes

Typically, a CPU virtual prototype consists of two differentiated parts: the functional emulator and the timing simulator. Depending on the tool and the targeted CPU, the details of the implementation for both the timing simulator and the functional emulator might be not accessible. A functional emulator is able to run application code that has been compiled for a particular architecture and to perform its execution in such a way that the memory data and architectural registers contain an exact representation of the real processor state. Precisely, the information of the processor state has to be disclosed to allow compilers and system software to use the modeled CPU. In that respect, we propose a fault-injection approach that uses only architectural registers and memory contents to minimize the intrusiveness of the fault injection in the virtual prototype. Additionally, as mentioned before, to accurately capture the behavior of the faults affecting the different processor nets, fault-injections at the virtual prototype must be enriched with meaningful information from the RTL. For example, in line with the results shown in [54], if fault injections target architectural registers only it is important to know the percentage of total processor faults that are represented by such fault-injection strategy. We let as future work the definition of the suitable injection strategies in the virtual prototype.

F.4 FALLES: Fault injection and Analysis for Low Level Evaluation Suite

The tool FALLES is an injection and analysis tool comprising a set of TCL and AWK scripts. It is designed to operate with low level descriptions of a system, mainly RTL and Gate Levels, by using the technique of simulation commands [86]. To do so, it currently supports Modelsim [71] tool to perform the simulation.The reason of its existence is accelerating the costly process of simulating a complex system running a realistic workload. The methods to achieve the improvement are quite straightforward. First of all, it trims all the simulation generated outputs to the minimum required amount of data, offering the possibility to apply any type of post-processing to them. Second it exploits massive parallelism of multi-core or grid computers to run up to several thousand concurrent threads. Third, it uses highly optimized text parser AWK to process the results.

When using a tool like FALLES, maintainability, extendability, ease of use and versatility are paramount. It can perform cycle-accurate simulations or, for an implemented design in Gate level, optionally accept Standard Delay Files (SDF) as inputs to study process corners. If such type of simulation is performed clusters or grids are the preferred option to exploit a huge amount of cores. Up to now Oracle Sun Grid Engine (SGE) management system [180] is supported, but it is very easy to add support for other grid managers.

Regarding the injectable fault models, stuck at 1 or 0, open line, indetermination, pulse or bit-flip are supported, in the variations of transient, intermittent or permanent duration. New fault models can be added at will just editing some TCL code.

F.5 Experimental Results

In this section we present some preliminary results of the characterization of how faults in any of the available nets in the processor propagate to the architectural registers. The results presented in this section correspond with fault injections performed in a CPU RTL description and focus on the propagation of faults in every processor net to the processor architectural registers. In fact, this information is the one that will be used in the future to feed fault injections in the virtual prototype.

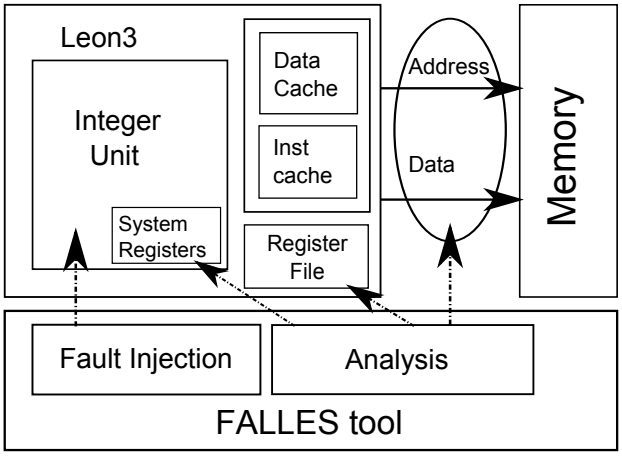


Figure F.2: RTL robustness verification framework

F.5.1 Experimental Setup

For the experimentation a 32-bit LEON3 SparcV8 microcontroller is selected, mainly because an RTL model is available along a microarchitectural description, and it is a processor used in safety-critical systems [184]. The LEON3 comprises mainly a 7-stage pipeline for integer operations (integer unit, *IU*) plus data and instruction caches. Since a minimal configuration of the processor has been chosen to limit the cost of RTL simulations, all instructions use all pipeline stages, and there is no floating point unit. The style of RTL description is homogeneous in the integer unit, which is described in a structural synthesizable VHDL. Such *IU* unit has been chosen as the target of injections, in a test framework as described by Figure F.2. Injection and analysis points have been selected according to Section F.3. To enable analysis of register faults in a timely manner LEON3 has been configured to use a flat register file[2] as this reduces the number of total registers that need to be studied in every simulation.

The workload chosen for investigation includes programs from 2 different benchmark suites: the Mälardalen WCET group suite [73], suitable to test real-time system properties and the EEMBC Autobench suite [129], which reflects realistic tasks of some automotive safety-critical systems. The selected programs are: a finite impulse response filter over a 700 items long sample (*fir*), a matrix multiplication of 4x4 size (*matmult*), a road speed calculator (*rspeed*), a CAN bus reader (*canrdr*) and a tooth-to-spark task, which locates the engine's cog when the spark is ignited (*ttsprk*).

[2]Note that a typical SPARC configuration uses a windowed register file configuration with around 144 32-bit registers. Tracking the contents of 144 32-bit registers even for relatively small benchmarks is currently unfeasible.

Regarding the faultload, several permanent hardware fault models have been chosen, specifically single stuck-at-1, stuck-at-0 and open line. These have been injected using simulator commands as in [86]. The campaign for each fault model and workload has consisted of one experiment per injection node (since permanent faults are applied), totaling 5,246 nodes. As the focus of the experiments is to classify fault propagation, each experiment applies a single injection in a fixed instant: just before the execution of the main procedure, after the initialization.

F.5.2 Results

After injections, Figure F.3 shows the distribution of different errors in the register file ('user registers') and control registers ('system registers') of the integer unit. The axis shows the percentage of total injected faults which propagated to the registers to become errors, for the specified fault models. Only one of the different benchmarks, ttsprk, is shown for space reasons. As observed, with the FALLES tool we have determined the information regarding how are faults propagated throughout the circuits to reach when applicable the analyzed registers, where the critical positions of the register file appear to be numbers 15 and 129 for such benchmark.

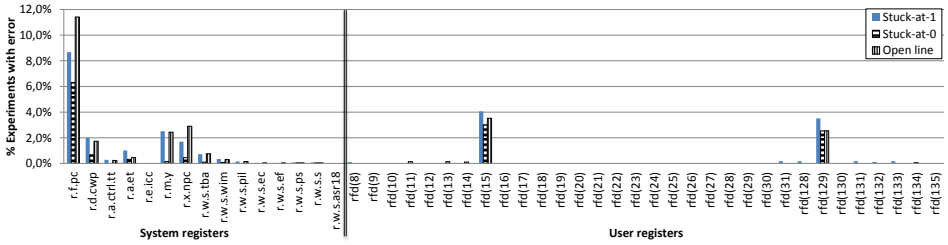


Figure F.3: Errors distribution in system and user registers, ttsprk

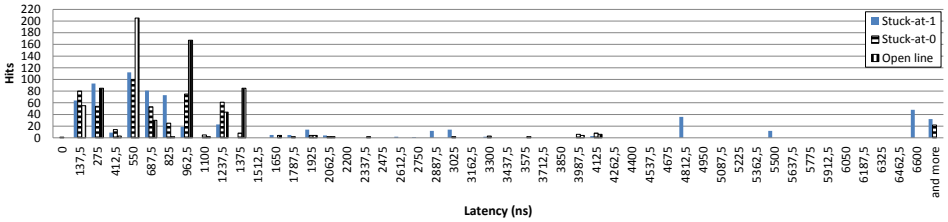


Figure F.4: Histogram of propagation latencies from error to failure, ttsprk

In addition, Figure F.4 shows the histogram of the distribution of latencies for the same ttsprk benchmark for the 3 fault models. The shown latencies account for the time spent since the first moment an architectural register is altered until a mismatch is detected at any of the peripheral buses (data or address). Taking into

account that the clock cycle of the considered system was 10 ns, we can tell the number of cycles it takes to propagate an error to a failure (considering the buses as outputs) is mainly under 138, with some sparse cases taking longer time. This latency information is helpful when assessing the behavior of real time systems such as those meant to be certified.

F.6 Conclusions

The use of virtual prototypes has recently arised as a promising approach to reduce the costs associated with the robustness verification of safety critical processors. However, for this low-cost simulation approach to be adopted its accuracy must be validated. In this paper we presented a methodology to increase the confidence on the fault injection experiments using virtual prototypes. The proposed methodology uses some meaningful information that is extracted from fault injection experiments at the RTL to enrich the fault injections at the virtual prototype. In the proposed methodology fault injections in the virtual prototype target only memory and architectural registers of the CPU to minimize its intrusiveness. The actual definition of the fault injection strategies in the virtual prototypes is let as future work.

Appendix G

Analysis and RTL Correlation of Instruction Set Simulators for Automotive Microcontroller Robustness Verification

Authors: Jaime Espinosa, Carles Hernández, Jaume Abella, David de Andrés and Juan Carlos Ruiz

Increasingly complex microcontroller designs for safety-rele–vant automotive systems require the adoption of new methods and tools to enable a cost-effective verification of their robustness. In particular, costs associated to the certification against the ISO26262 safety standard must be kept low for economical reasons. In this context, simulation-based verification using instruction set simulators (ISS) arises as a promising approach to partially cope with the increasing cost of the verification process as it allows taking design decisions in early de-

sign stages when modifications can be performed quickly and with low cost. However, it remains to be proven that verification in those stages provides accurate enough information to be used in the context of automotive microcontrollers. In this paper we analyze the existing correlation between fault injection experiments in an RTL microcontroller description and the information available at the ISS to enable accurate ISS-based fault injection.

G.1 Introduction

An increasing number of complex functionalities in automobiles rely on electronic components such as airbag modules, electronic parking brakes, etc [100, 141]. Thus, modern cars may include up to 100 million lines of code that need to be integrated into the least number of Electronic Control Units (ECUs) for cost contention [27]. Moreover, the amount of software in cars is expected to further increase in the future. Hence, more powerful and complex microcontrollers implemented with more integrated and less reliable technology are needed to respond to this increasing performance demand. However, hardware complexity challenges V&V processes to adhere to safety standards.

Complex and error-prone microcontrollers require the adoption of new methods and tools to enable a cost-effective robustness verification of safety-relevant systems. With the adoption of safety-related certification standards like ISO-26262 [81] in the automotive domain robustness verification has become one of the fundamental stages in the certification process for any new design. Robustness verification is carried out at different stage levels by performing intensive fault injection experiments [18]. Complex microcontroller verification challenges product design cycles, what can lead to financial loss and severe delays especially if left for the final production stages (i. e. hardware prototypes). Hence, designers have been striding to move this procedure towards the early stages of design, in order to detect design flaws or safety threats in a timely (and low-cost) manner.

Simulation-based verification has been shown to reduce costs associated with the robustness verification process as any misbehavior or defect can be corrected early. Unfortunately, simulation-based verification is often carried out at the gate level, and so the testing process is extremely time-consuming. With a higher level of abstraction such as RTL, the burden is reduced but it is still overwhelming for repeated use. This fact renders impractical fault injection after each design modification. Thus, a sheer increase in simulation speed is needed while still obtaining acceptably accurate results. Simulation-based verification using Instruction-Set Simulators (ISS) arises as one of the most promising approaches to partially cope with the increasing complexity of the verification and test process of complex systems. The main benefits of this low-cost verification step are (1) the reduction of

the verification time and (2) the ability to start the verification process long before having the RTL description of the processor, thus saving costs.

However, performing meaningful fault injection experiments using an ISS simulator is challenging as the modeled processor lacks most of the information required for accurately injecting faults. In fact, the majority of the potential injection nodes that are present at more detailed abstraction levels like RTL or gate-level are missing. For example, typical ISS-based fault injection experiments that rely on injecting faults in the register file [77][150] cannot be used to estimate failure rate metrics as required by certification standards if it is not possible to determine the probability that a given fault present at any possible microcontroller net or gate propagates to the register file.

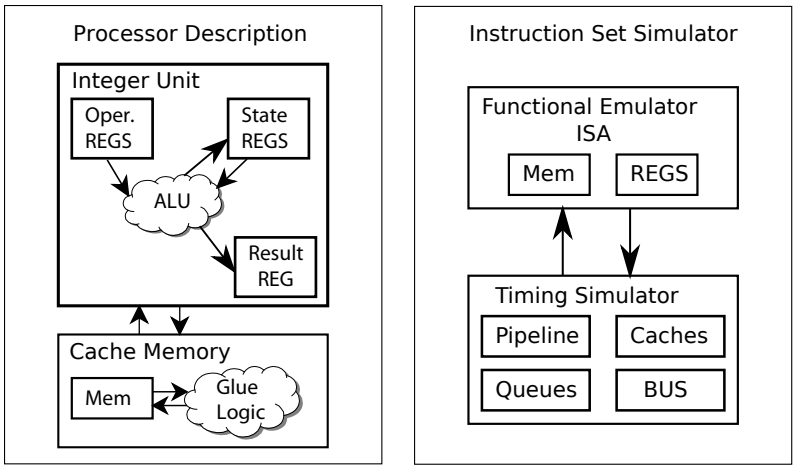
In this paper we increase the confidence in the fault injection experiments performed with an ISS by carrying out a thorough correlation of the fault injection experiments in an RTL microcontroller description with the information available at the ISS. In particular, we propose instruction's *diversity* as a metric to enable a coarse-grain correlation of the probability that faults injected in the RTL propagate to the system outputs (i.e. the probability that a fault becomes a failure). Instruction's *diversity* is computed as the number of unique instruction types (opcodes) used by the application and represents the area the application exercises by assuming all instructions make a uniform use of microcontroller resources. Furthermore, for permanent fault models – the scope of this work – it is independent of the particular order in which instructions within this application are executed. This information is crucial to perform efficient fault injection campaigns that simulate programs exercising only the hardware components that have been modified[1] so that impact of faults can be understood with a limited number of short simulations. While data reported does not include latent errors not manifested at off-core boundaries, mechanisms such as LiVe [77] can be used in the context of lockstep processors for safety-critical systems to enforce latent errors to manifest at off-core boundaries, where errors are detected by lockstep execution (and reported as failures in our work).

G.2 Towards Simulation-based Robustness Verification

In the safety-related hardware development process, fault injection is a valuable method for the verification of hardware design in the automotive domain as indicated in ISO26262 Part 5 clause 7.4.4.1 [81] for ASIL B, C and D[2]. During the development phase, simulation-based fault injection methods are typically em-

[1] Input data triggering injected faults depends on the programs used. Devising software-based tests [131] with specific coverage for the particular processor evaluated is beyond the scope of this work, so we use performance benchmarks.

[2] ASIL stands for Automotive Safety Integrity Level. There are four levels, from A to D, being D the highest one.



(a) (b)

Figure G.1: (a) RTL processor description (b) Microarchitectural processor description

ployed instead of physical-based methods –such as injecting disturbance in power lines, electromagnetic interference (EMI), etc.– due to their repeatability, controllability and cost. Fault injection using simulation can be performed using different levels of abstraction like functional, RTL, or gate-level. The current state of practice uses RTL and gate-level experiments to test hardware robustness as these methodologies have been shown to provide good accuracy [123]. A commonality of every simulation methodology is that it has to be related with the techniques used at silicon level for validation. For proper use of ISS to that end, these must be qualified in the same way.

G.2.1 Fault injection at the RTL

A circuit described at the functional level does not provide information on the internal components, but only an method to obtain outputs from inputs. Conversely, RTL description of a circuit comprises contents of registers and combinational logic, expressed in terms of logic functions and connections as shown in Figure G.1(a). Specifically, the detail on the intermediate steps in terms of internal signals and operands, which allows for later synthesis of the design, renders it an ideal candidate for fault injection. Two are the main benefits. First, it is the lowest level –most detailed– and closest to the level where faults happen in the real system –the physical level– which, without loss of generality, achieves a good degree of representativity. Second, since the next level in detail –the gate level– does include the *implementation technology* in the description of the system,

results of injection in RTL stay valid across different implementations, platforms, etc.

G.2.2 Fault injection at the ISS Level

Typically, an ISS consists of two differentiated parts: the functional emulator and the timing simulator (see Figure G.1(b)). The functional emulator contains the full description of the instruction set architecture (ISA) and keeps the architectural state of the processor (i.e. architectural registers and memory data). A functional emulator is able to run application code that has been compiled for a particular architecture and to perform its execution in such a way that the memory data and architectural registers contain an exact representation of the real processor state. In other words, the functional emulator is the interpreter. The timing simulator interacts with the functional emulator and mimics with some degree of accuracy the timing behavior of the different instructions during their execution. To do so, the timing simulator models the cache memories, the processor pipeline, the register file structure, and several other queues and structures depending on the target degree of accuracy. Thus, it allows computing information like the number of execution cycles, cache hits/misses and the like. Some implementations of an ISS may have functional and timing simulation integrated, although this typically challenges their flexibility.

In this paper we focus on the functional part of the ISS given that it is the highest (and so the cheapest) abstraction level. This is a necessary step to validate the suitability of an ISS for the robustness verification of safety-relevant processors. We consider little timing information (basically instructions latency). Moreover, by working mostly with the functional part of the ISS results mainly depend on the actual ISA used and remain valid for any implementation of such ISA (or the method can be ported easily). Of course, this comes at the expense of trading off some accuracy. Still, as we show later, the functional part of an ISS already provides highly-valuable information to characterize the behavior of microcontrollers in presence of faults.

G.2.3 ISS-based Verification

Safety-relevant systems need to go through a certification process. In automotive systems the ISO26262 functional safety standard [81] specifies the safety requirements that the different system components need to fulfill in relation with the overall system's safety. Simulation-based fault injection is one of certification-friendly methodologies for the safety requirements verification when analytical methods are not considered to be sufficient as specified in ISO26262 Part 5 Table 3. Note that this is the case for complex hardware components verification like a microcontroller. Current practice on simulation-based verification is performed

at the RTL and gate-level descriptions of the circuit as these methodologies have been shown to provide good accuracy in automotive microcontrollers [175].

The use of an ISS for verification in the context of ISO26262 is challenging as the correlation of the experiments at this abstraction level with the physical level tests is not a straightforward task [103]. In this sense, a first step in that direction is to correlate with a closer level such as RTL.

Robustness verification using ISS brings several benefits that can significantly contribute to the cost and complexity reduction of the verification process. We target the achievement of the following three main benefits of using ISS-based robustness verification: *(B1)* Fast simulation time, *(B2)* Detection of safety misbehavior at very early design stages of product development and *(B3)* Improvement of the hardware/software integration.

B1 speaks about the need for reducing simulation time to be able to perform the verification of increasingly complex circuits. Furthermore, increasing the simulation speed also allows the validation of more significant workloads where not only functional deviations related to safety can be detected, but also timing-related deviations [77]. Speeding up this process helps microcontroller designers evaluate the impact on safety of modifications quickly (e.g., adding new instructions). Differently, *B2* refers to the economical gain associated to the early detection of design malfunctions which is specially significant in the case of ISS-based simulation, as it does not require the actual microcontroller to be fully described. Instead, a complete definition of the ISA (or the subset of the ISA to be analyzed) suffices to perform this step. Finally, *B3* talks about the benefits of enabling ISS-based simulations in the safety-related software development. On one hand, ISS-based fault injection will help improving the modeling of the hardware/software interactions with respect to the system's safety as defined in [81]. On the other hand, as system software development relies mainly on the information available at the ISS (e.g., architectural and system registers), being able to perform meaningful fault injection experiments at the ISS level also opens the door to meaningful reliability analysis of the software components and layers long before the actual microcontroller has been deployed. Thus, software and hardware development and verification can occur in parallel to some degree, hence reducing the time-to-market, which is a key metric in the automotive domain.

G.3 Correlating RTL with ISS fault injection

In this paper we consider the probability of failure P_f as the probability that a fault is propagated to off-core boundaries. We have selected off-core boundaries as the point of failure manifestation as this is the exact point at which light-lockstep cores outputs are compared for error detection purposes. Microcontrollers implementing light-lockstep compare any off-core activity (i.e., memory read/write, I/O read/write), but cannot detect faults that do not propagate outside cores (e.g., latent faults in registers or cache memories). Microcontrollers implementing light-lockstep like the Infineon AURIX [83] and the STMicroelectronics SPC56XL60/54 family [147] are widely used for safety-relevant applications in the automotive domain.

To correlate RTL fault injection experiments with the ISS we analyze the information from the applications that are executed in the microcontroller that can be used to approximate failure manifestation probability. As the ISS decodes all instructions of the executed applications, information is available at the granularity of instructions. In this regard, we make the following hypothesis: the probability that a fault present in the microcontroller becomes a failure when executing a given set of instructions I_s is a function of the actual executed instructions I_s, their input data, and the temporal behavior of the executed instructions. Thus, $P_f = f(I_s, inputs, time)$. I_s temporal behavior includes the instruction dependences and their latency, as well as the exact point in time at which faults are present in the microcontroller. Note that our initial hypothesis about the fact that the failure probability depends on the microcontroller's *spatial* and *temporal* vulnerability, is in line with the traditional analysis of processor vulnerability factors [117] in the high-performance domain, where processors are indeed more complex than microcontrollers used in the automotive domain. In the previous P_f expression, I_s and input data determine the processor's *spatial vulnerability*, whereas the I_s temporal behavior defines the microcontroller's *temporal vulnerability*.

However, expressing P_f as a function of I_s, its input data, and I_s temporal behavior is still an overly complex function due to the value space for input data (e.g. 2^{32} different values for a 32-bit input). To reduce the problem space we consider that the data's universe can be restricted and/or upper bounded if, either we are able to introduce enough data variability, or we use corner cases for the applications' input data. Instructions temporal behavior can be captured using ISS by annotating the exact cycle at which the different instructions in a given I_s enter and leave a given microcontroller unit. However, in this paper we remove the dependence on the temporal utilization of the failure probability by focusing on permanent fault models, e.g. stuck-at-1, stuck-at-0 and open-line. We focus on permanent faults not only to remove the temporal variable but also because the number of injections to perform in every node in order to obtain significant results for transient faults

is extremely high. For example, the determination of single-point fault and latent fault metrics as required by ISO26262 [81] hardware certification typically relies on the use of software-based tests (SBT) [131] and stuck-at fault models. Note that the huge execution time SBT require to achieve high coverage precludes the use of fault-models requiring very large number of fault injections.

With the assumptions above P_f can be reformulated as $P_f = f(I_s)$. This simple definition of P_f implies that the probability of an injected fault to become a failure depends on the set of instructions exercised regardless of the order in which they are executed and the particular existing dependences across instructions. In other words, our hypothesis is that the probability that a failure is triggered by a given set of instructions I_s is proportional to the processor utilization (in terms of area). This hypothesis translates the problem of determining the failure manifestation probability in the problem of determining what is the processor utilization that a given set of instructions makes.

Determining Microcontroller's Utilization. We introduce the *diversity* metric to determine the processor utilization for a given application. To relate instruction's diversity with the area exercised we consider these items:

1) The probability that a given instruction triggers a failure depends on the number of functional units a given instruction exercises. For example, all instructions have the same probability of triggering a failure at decode and fetch stages as these stages are used by every instruction [137]. On the contrary, different type of instructions, like logical and arithmetic instructions, do not necessarily use the same functional units.

2) Different functional units have different area occupation. From an RTL perspective this means that the number of fault injection points in a given functional unit is not the same for all of them and that the number of fault injection points is not necessarily proportional to the occupied area. The first concept speaks about the fact that in homogeneously detailed RTL representations of functional units the number of fault injection nodes is closely related to the area of a given component. The latter concept is related to the fact that heterogeneously detailed HDL descriptions of functional units lead to a decoupled relationship between injectable nodes and area occupancy.

To be able to deal with the heterogeneous processor utilization originated due to (1) and (2), *diversity* is computed for the the different functional units. Instruction's *diversity* of the m^{th} functional unit, D_m, can be computed using the ISS by dumping instructions information and finding the number of accesses to any of the available functional units for each instruction. Finally, D_m has to be related with the failure probabilities for the different processor functional units. The probability of failure of the m^{th} processor unit P_f^m can be computed using the following equation:

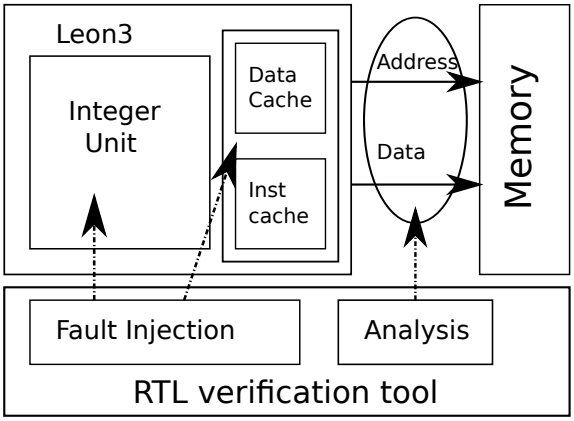


Figure G.2: RTL robustness verification framework

$$P_f = \sum_{m=1}^{N_{mod}} \alpha^m * P_f^m \tag{G.1}$$

In this equation N_{mod} is the number of processor components and α^m is used to ponderate the effect of the heterogeneity in detail. α^m is in the range $[0, 1]$ and represents the fraction of the total area occupied by the processor unit m.

It is important to remark that the *diversity* metric inherently assumes that the utilization of resources within a given functional unit that instructions make is uniform.

Note that the area exercised by different instructions can be partially overlapped. Hence, executing different instruction types when few of them have been executed is likely to increase P_f, whereas executing them when many of them have been executed is less likely to increase P_f because the units accessed have been probably accessed by previous instruction types.

G.4 Experimental Validation

G.4.1 Experimental Setup

In this section we analyze how accurately RTL fault injection experiments can be reproduced using a microcontroller ISS. To do so, we inject faults in the RTL microcontroller model and measure the percentage of injected faults propagating to failures. Any mismatch detected when writing to memory is considered a sys-

	Benchmarks					
	Automotive				Synthetic	
Instructions	puwmod	canrdr	ttsprk	rspeed	membench	intbench
Total	111866	96492	96053	75058	19908	2621
Integer Unit	111862	96488	96049	75054	19908	2621
Memory	40613	33766	34905	25155	4385	19
Diversity	47	48	47	47	18	20

Table G.1: Benchmarks characterization

tem failure. Figure G.2 illustrates the fault injection methodology followed in this paper. For the analysis and correlation we use the 32-bit Leon3 sparcv8 microcontroller as both the ISS and RTL description of this circuit are available [184]. This microcontroller consists of a 7-stage pipeline for integer operations (IU). In this microcontroller all instructions use all pipeline stages. The RTL processor description follows the structural VHDL design guidelines and it models the IU and the cache memory ($CMEM$) as separate components.

In this study we inject faults using simulation commands as described in [86]. The choice of injected faultload is single hardware faults of permanent type, targeted to VHDL signals, ports and variables which appear at a fixed injection instant and cause either stuck-at-1, stuck-at-0 or an open line. It has been applied to all available points from the IU and CMEM microcontroller units.

For the workload in this study we use the EEMBC Autobench suite [129] which reflects current real-world demand of some automotive CRTES and 2 synthetic benchmarks, which have been designed to use intensively memory instructions or integer instructions, and provide additional diversity values. Table G.1 shows the benchmarks analyzed.

G.4.2 Experimental Results

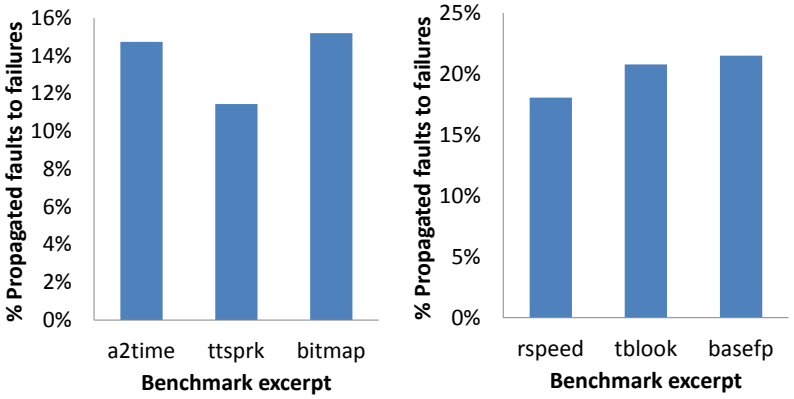
In this section we proceed incrementally to validate the hypothesis made in the previous section. First, we show that the impact of inputs data variability in the probability of failure is captured for applications executing a large number of instructions. Later, we analyze the effect of the instructions temporal behavior. Finally, we show the existing correlation between the processor's utilization and the probability of failure.

Application's data. We analyze the impact of input data variation on the probability of failure making two different experiments. For the first experiment we have injected faults in short excerpts of 2 different subsets (consisting of 3 different applications each) of EEMBC benchmarks. The selected excerpts represent the initialization phase of the benchmarks where the data to be used in the experiment are read and allocated in memory. All three applications within a subset have identical code and the only difference among them comes from the different input data they require. Each subset of applications consists of a different I_s. Figure G.3 shows the effect of input data variability in the probability of failure (as I_s is fixed). Differences across benchmarks are meaningful, up to 4 percentage points (pp), so in principle the impact of data variability cannot be neglected for short applications.

We have performed a second experiment to show that input data effect can be removed when benchmarks execute a significant number of instructions. To do so, we have injected faults in the microcontroller's *IU* and run benchmarks with different number of iterations (2, 4 and 10 iterations). Figure G.4 shows results for the *rspeed* application. As shown, P_f remains constant regardless of the executed iterations meaning that the effects of new realistic data exercised in the subsequent iterations are already included in the data space covered with 2 iterations. Further, P_f is exactly the same for the other benchmarks that use the same type of instructions. Therefore, we can conclude that for sufficiently long benchmarks, *inputs* is no longer needed in $P_f = f(I_s, inputs, time)$. Regarding fault detection latency, maximum latency grows with the number of iterations (see plot (b)) due to those faults affecting data that is not used until the last part of the program, after the iterations, in line with the observations in [77]. Thus, 2 iterations provide the same information as 10, but allow reducing fault injection and analysis time.

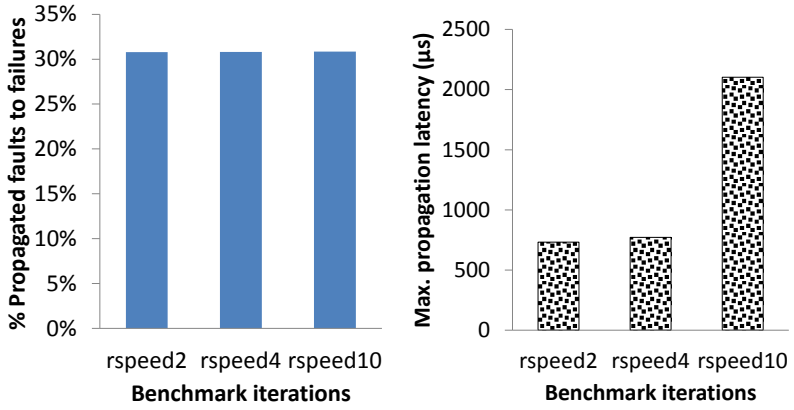
Temporal Behavior. The next independent variable to clear from equation $P_f = f(I_s, inputs, time)$ is *time*. In the case of permanent faults one expects a fault to become a failure regardless of when it is triggered. In order to prove this, we have evaluated *ttsprk* and *puwmod* benchmarks that have exactly the same **diversity**, so they execute the same type of instructions, but they execute them in different order. As shown in Figure G.5, the percentage of propagated faults for both benchmarks is roughly identical for different types of permanent faults. A different case would happen with *transient* faults, as their impact can vary greatly depending on the instructions being executed at the moment faults hit the system. We let the analysis of the impact of transient faults as future work.

Microcontroller Utilization. Finally, we check the hypothesis that the probability of failure mainly depends on the instruction set (I_s) used in the benchmark. To do so, we study the correlation between utilization of the different instructions – which relates to the spatial utilization of the microcontroller – and P_f. Furthermore, we also check that the correlation holds when applied to the IU and CMEM



(a) 8 types of instructions (b) 11 types of instructions

Figure G.3: Input data variation in 2 sets of benchmark excerpts with uniform instruction types and numbers, using stuck-at-1 injections at integer unit



(a) (b)

Figure G.4: Input data variation impact analyzed with 2, 4, and 10 full iterations of benchmark rspeed using stuck-at-1 injections at integer unit

modules separately. We identify instruction **diversity** as the appropriate metric to determine the processor's spatial utilization.

Figures G.5 and G.6 present RTL injection results for the IU and CMEM, respectively for stuck-at-0, stuck-at-1, and open-line fault models. The first observation is that, for the automotive benchmarks, P_f is almost constant despite the fact that the executed benchmarks present different number and distribution of the executed instructions (see Table G.1). However, if we pay attention to the instruction **diversity** we realize that these 4 benchmarks use almost the same number of

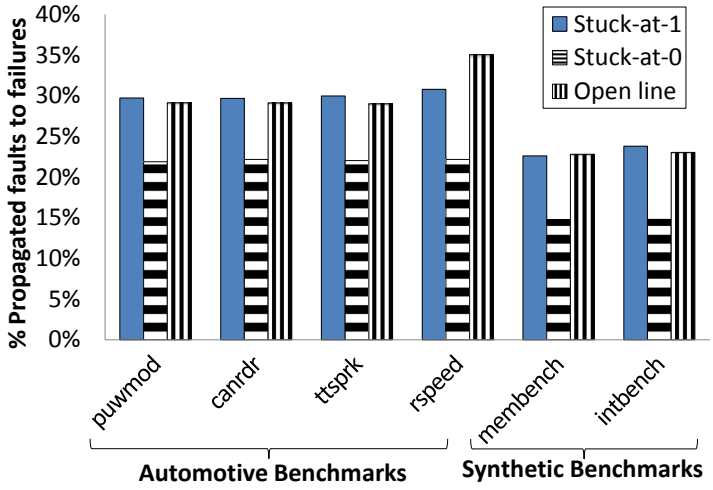


Figure G.5: Fault injection experiments for different benchmarks and fault models at IU nodes.

different instructions as given by the **diversity** factor. To prove that P_f is coupled with the instruction **diversity** we also used two different synthetic benchmarks. As these benchmarks are designed to used different I_s we observe some variability in the P_f.

Finally, Figure G.7 correlates P_f for the different benchmarks used in this study with the instruction **diversity**. To increase the number of points in the plot we also consider the benchmarks excerpts shown before. In these benchmarks the effect of input data variability is minimized by including the P_f value of all 3 benchmarks of each subset.

Simulation time.In order to obtain the fault injection data for the complete benchmark executions, up to 25,478 hours of computing time have been employed, distributed in 2 massively-parallel clusters and 2 powerful workstations.In contrast, less than 300 computing hours on a single workstation is enough for performing the same number of experiments with an ISS. This illustrates the importance of qualifying low-cost methods of achieving accurate results.

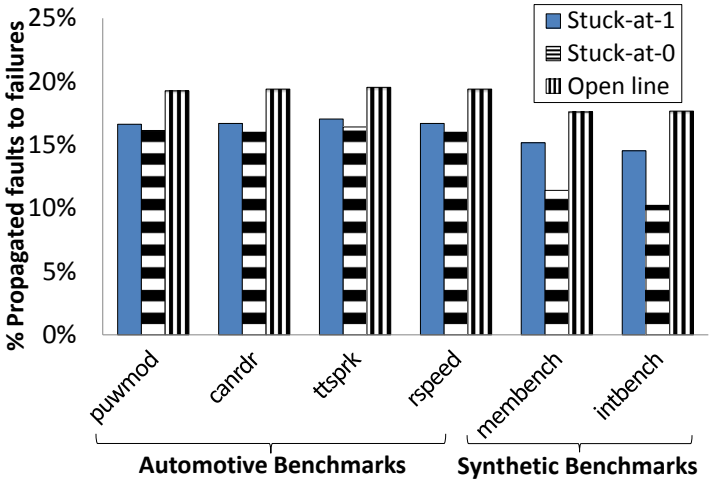


Figure G.6: Fault injection experiments for different benchmarks and fault models at CMEM nodes.

G.5 Related Work

Fault injection methodologies are widely employed for the microcontrollers robustness verification in the automotive domain [123]. Fault injection experiments can be performed at several abstraction levels to exploit the existing accuracy cost trade-off [175]. RTL and gate-level fault injection experiments are the most adopted approaches to perform the certification of hardware products against certification standards [81]. Practitioners have performed fault injection at the logic and RTL levels using different techniques. A widely-used method is the injection in the HDL through simulator commands [86], which works well for most of the fault models described in the literature. Furthermore, some additional fault models, such as those involving several injection points – short-circuit, multi-bit injection – can be applied if the more intrusive technique of saboteurs is used [15] where an instrumentation of the model – and the consequent decrease in simulation speed – is required.

Fault models representativeness was validated for logic/RTL levels [66]. For higher abstraction levels like the ISS previous work pointed out the difficulties of correlating the results with experiments at the physical level [103]. The majority of works at the ISS level focus on processor's reliability estimation, which is obtained by the determination of the architectural vulnerability factor (AVF) [117]. The AVF is determined by the fraction of the architectural bits contributing to the processor's reliability. A similar approach is the one in [24] where the concept of instruction vulnerability factor (IVF) is proposed to evaluate how faults in every instruction affect the final application output. Likewise, in [137] the IVF is used to define a

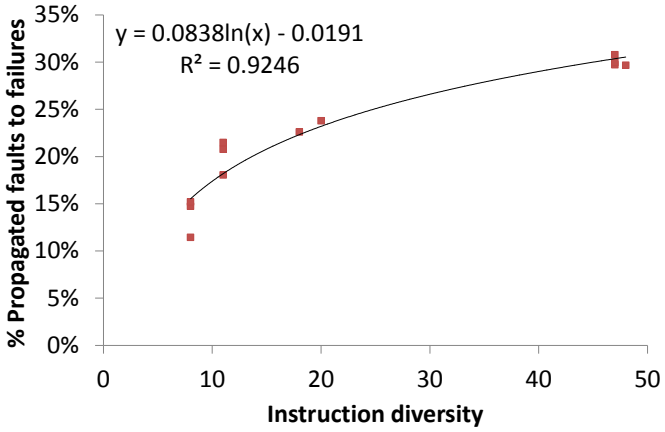


Figure G.7: Propagated faults in terms of instruction diversity for the stuck-at-1 model in IU.

compilation process taking into account ISS reliability information. An attempt of correlating ISS and logic/RTL was done in [109] focusing on the correspondence between instruction and low-level fault models. In this paper we focus on showing the correlation of results of RTL fault injection and the data available at the ISS level.

G.6 Conclusions

Microcontroller verification based on fault-injection is a key approach in the automotive domain, specially for the most critical functionalities as detailed in ISO26262. However, early detection of design flaws is incompatible with having a detailed description of the microcontroller such as RTL or gate-level ones. Moreover, fault injection in RTL or gate-level designs is painfully slow. Therefore, there is a need for having low-cost models of hardware that can be had at early stages of the design and provide accurate-enough information. The ISS is one of those as it is needed to allow software providers to start their developments before the hardware is ready.

In this paper we apply correlation between fault injection in the ISS and in the RTL showing that highly accurate results can be had for different permanent fault models. In the study we prove that the order of instructions in the execution and their input data are roughly irrelevant for permanent faults. Instead, the different types of instructions exercised by the benchmarks run in the ISS are the key difference towards measuring fault propagation.

Appendix H

Characterizing Fault Propagation in Safety-Critical Processor Designs

Authors: Jaime Espinosa, Carles Hernández, Jaume Abella

Achieving reduced time-to-market in modern electronic designs targeting safety critical applications is becoming very challenging, as these designs need to go through a certification step that introduces a non-negligible overhead in the verification and validation process. To cope with this challenge, safety-critical systems industry is demanding new tools and methodologies allowing quick and cost-effective means for robustness verification. Microarchitectural simulators have been widely used to test reliability properties in different domains but their use in the process of robustness verification remains yet to be validated against other accepted methods such as RTL or gate-level simulation. In this paper we perform fault injections in an RTL model of a processor to characterize fault propagation. The results and conclusions of this characterization will serve to devise to what extent fault injec-

tion methodologies for robustness verification using microarchitectural simulators can be employed.

H.1 Introduction

Increasingly complex modern electronic designs for safety-critical markets must adhere to the strict requirements of functional safety standards. Thus, those designs have to undergo an expensive and time-consuming certification process, which is against the always stringent need to reduce the time to market. In that context, new tools and procedures have to be devised for a quick and cost-effective way to test whether robustness properties are achieved throughout the whole design flow.

Simulation-based fault injection is considered a suitable methodology for the robustness verification process, as quick and cheap corrections on misbehavior can be made. Unfortunately, such fault injection is often carried out at the gate level, and so the testing process can be excruciatingly slow and requires unduly high computing resources. When applied at a higher level of abstraction such as Register Transfer Level (RTL), the burden is reduced but it is still overwhelming for repeated use. This fact renders impractical fault injection after each design modification. If designers are intended to verify each modification, a sheer increase in simulation speed is needed while still obtaining acceptably accurate results. Considering the constraints mentioned above, microarchitectural simulators arise as one of the most promising approaches to partially cope with the increasing complexity of the verification and test process of complex systems. The main benefits of this low-cost verification step reside on the reduction of the verification time and on the ability to start the verification process long before having the RTL description of the processor, thus saving costs.

The use of microarchitectural simulators to estimate failure rate metrics is challenging as the modelled processor lacks most of the information required for accurately injecting faults. In fact, the majority of the potential injection nodes that are present at more detailed abstraction levels like RTL or gate-level are missing. Typically, fault injection experiments using microarchitectural simulators focus on the register file [77][150] and on the different memory structures [149][2]. However, these fault injection experiments, while useful to test the effectiveness of fault-tolerant capabilities and the like, are not suitable to estimate failure rate metrics as required by certification standards. For this to happen it is required to quantify how likely is that a fault present at any possible processor net, gate, or flip/flop propagates to the register file or the different memory structures.

In this paper we focus on the characterization of processor fault's behavior as a first step towards increasing the confidence on failure rate estimates given by microarchitectural simulations. In particular, we focus on how faults propagate

to the different system's outputs and how many of these faults propagate to the processor register file and/or control/status registers. For the characterization of fault propagation, faults have been injected in an RTL description of the LEON3 processor [184] using simulation commands as described in [86].

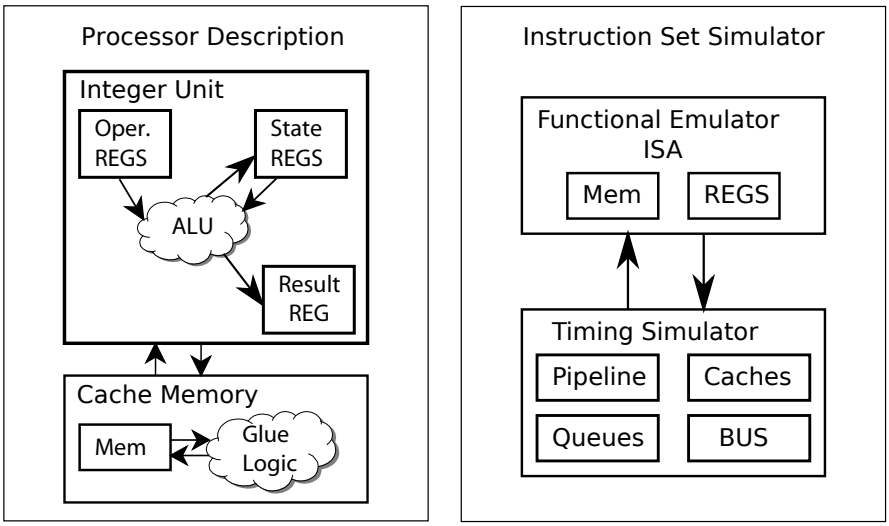
Results in this paper show that even though a significant fraction of the faults originated in the core show up in the register file and/or control/status registers, injecting faults only in such registers is not enough to accurately model the impact of core faults, since a considerable amount of system failures are not preceded by any error manifestation in the mentioned registers.

The rest of the paper is organized as follows. Section H.2 reviews the state-of-the art in fault injection in the RTL and microarchitectural simulators. Sections H.3 and H.4 present the methodology used in this paper for characterizing processor's fault propagation and the results of such analysis, respectively. Finally, in Section H.5 some conclusions are drawn.

H.2 Background on Simulation-based Robustness Verification

Safety-relevant systems need to go through a certification process. For instance, in automotive systems the ISO26262 functional safety standard [81] specifies the safety requirements that the different system components need to fulfil in relation with the overall system's safety. In the case of avionics systems the standard that defines the methods and tools to certify electronic products is the DO-178B [134]. Regardless of the application domain, simulation-based fault injection is a certification-friendly methodology for the safety requirements verification when analytical methods are not considered to be sufficient. This is, for instance, specified in the case of automotive systems in ISO26262 Part 5 Table 3. Note that this is the case for complex hardware components verification like a microcontroller. Fault injection through simulation can be performed using different levels of abstraction like functional, RTL, or gate-level. The current state of practice uses RTL and gate-level experiments to test hardware robustness as these methodologies have been shown to provide enough accuracy when related to the silicon level [123]. In the same way, if microarchitectural simulators are to be considered for the robustness verification process, results obtained at this level must be correlated with the ones obtained at lower levels of abstraction like the RTL or gate-level. In this paper we focus on relating RTL fault injection to experiments carried out at the microarchitectural level.

Several techniques exist to perform RTL level fault injection. A widely-used method is the injection in the HDL through simulator commands [86], which works well for most of the fault models described in the literature. In fact, some addi-



(a) (b)

Figure H.1: (a) RTL processor description (b) Microarchitectural processor description

tional fault models such as those involving several injection points –short-circuit, multi-bit injection– can be applied if the more intrusive technique of saboteurs is used [15], but an instrumentation of the model –and the consequent decrease in simulation speed– is entailed.

Fault models representativeness has been validated for logic/RTL levels [66]. On the contrary, for higher abstraction levels like the microarchitectural one some works have pointed out the difficulties of correlating these results with the ones obtained at the physical level [103]. The majority of works carried out with microarchitectural simulators focus on processor's reliability estimation. Processor's reliability is estimated by the determination of the architectural vulnerability factor (AVF) [117]. The AVF is determined by the fraction of the architectural bits contributing to the processor's reliability. A similar approach is the one in [24] where the concept of instruction vulnerability factor (IVF) is proposed to evaluate how faults in every instruction affect the final application output. Likewise, in [137] the IVF is used to define a compilation process taking into account ISS reliability information. Finally, a truly existing correlation between fault injections experiments performed in an RTL processor description and the information available at the ISS was shown in [48] for the case of permanent fault models, though limited to final number of failures. Other detailed studies targeted to soft error models did not find such correlation by using single bit-flip injections in registers, suggesting other fault models should be devised for high-level injections [30].

H.2.1 Fault injection at the RTL

A circuit described at the functional level does not provide information on the internal components, but only a method to obtain outputs from inputs. Conversely, RTL description of a circuit comprises contents of registers and combinational logic, expressed in terms of logic functions and connections as shown in Figure H.1(a). Specifically, the detail on the intermediate steps in terms of internal signals and operands, which allows for later synthesis of the design, renders it an ideal candidate for fault injection. Two are the main benefits:

- First, it is the lowest level –most detailed– and closest to the level where faults happen in the real system –the physical level– which, without loss of generality, achieves a good degree of representativity.
- Second, since the next level in detail –the gate level– does include the *implementation technology* in the description of the system, results of injection in RTL stay valid across different implementations, platforms, etc.

H.2.2 Fault injection at the ISS Level

Typically, a microarchitectural simulator consists of two differentiated parts: the functional emulator and the timing simulator (see Figure H.1(b)). The functional emulator contains the full description of the instruction set architecture (ISA) and keeps the architectural state of the processor (i.e. architectural registers and memory data). A functional emulator is able to run application code that has been compiled for a particular architecture and to perform its execution in such a way that the memory data and architectural registers contain an exact representation of the real processor state. In other words, the functional emulator is the interpreter. The timing simulator interacts with the functional emulator and mimics with some degree of accuracy the timing behavior of the different instructions during their execution. To do so, the timing simulator models the cache memories, the processor pipeline, the register file structure, and several other queues and structures depending on the target degree of accuracy. Thus, it allows computing information like the number of execution cycles, cache hits/misses and the like. Some implementations of an ISS may have functional and timing simulation integrated, although this typically challenges their flexibility.

Therefore, fault injection in a ISS needs to be typically performed in registers and memory in the emulator, and propagation information can be obtained, including its timing, based on the event modeled by the timing simulator.

H.3 Characterizing Fault Propagation

As mentioned before, in this paper we want to show to what extent microarchitectural simulators can be employed in the robustness verification flow of safety-critical systems. In particular, we elaborate on the potentials of injecting faults in the processor architectural registers. The emulator part of a microarchitectural simulator only includes registers (inside the cores) and memory (outside the cores). Therefore, our view is that by characterizing which faults that reach core boundaries are reflected, either in the general purpose registers and/or in the control/status registers, we will know to what extent core faults behavior can be captured using the emulator part only.

Throughout the paper we use the common nomenclature to distinguish between faults, errors, and failures. We consider *faults* those upsets that can take place at any point of the design and are actually what is being injected; *errors*, the mismatches in the values of user registers (stored in the register file) or system registers in this case, and *failures*, the mismatches at the considered outputs. In our case, address and data buses are taken as system outputs. Note that this is the exact point at which light-lockstep cores outputs are compared for error detection purposes. Microcontrollers implementing light-lockstep compare any off-core activity (i.e., memory read/write, I/O read/write), but cannot detect faults that do not propagate outside cores . Processors implementing light-lockstep like the Infineon AURIX [83] and the STMicroelectronics SPC56XL60/54 family [147] are widely used for safety-relevant applications in the automotive domain.

We are interested in analyzing the influence of faults in the system towards the incorrect delivery of results, i.e. the appearance of failures. As studied earlier in literature, determining how faults in a system propagate through logic paths is not a straightforward task. The relationship between faults and errors depends on several factors. First of all, the actual implementation of the processor determines heavily which nodes are connected with which others, so that a path for propagation exists. Second, the system architecture according to the executed instructions and data determines the paths that are exercised and thus can propagate faults in nets to other structures and/or system outputs. Finally, in case faults are transient, the exact point in time when the fault occurs is also very relevant to have an error captured that can later potentially become a failure. In fact, the shorter the fault duration the lower the probability that it will be captured at a sequential element due to *time filtering*. In this study to facilitate the analysis we consider only permanent faults. However, we strongly believe that main conclusions drawn for permanent faults will remain also valid in the case of transient faults but the confirmation of this hypothesis is let as future work.

To characterize fault propagation we follow a methodology that consists of injecting faults in all possible nets of an RTL processor description. From the injectable

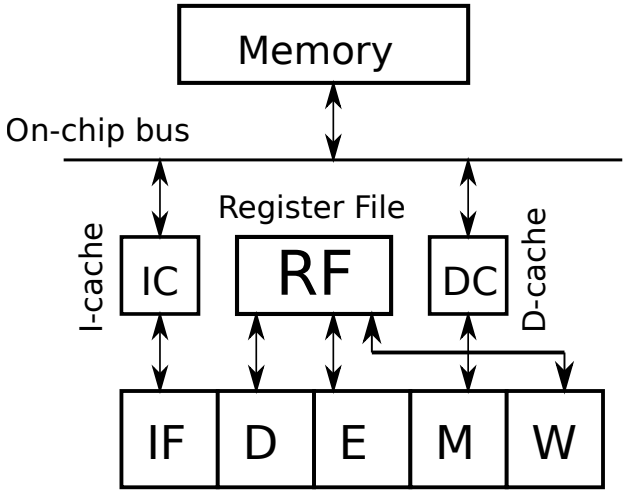


Figure H.2: Generic processor pipeline scheme. IF (instruction fetch), D (decode), E (execution), M (Memory), W (Write-back).

nets we have excluded the register file and cache memory structures due to the following reason: errors occurring within these structures are effectively detected and/or corrected by employing redundacy mechanims (e.g., error correction codes) and this is the case in most of the processors targeting safety-critical applications [185][83]. Moreover, available nets in these structures do not realistically represent their area. Memory structures are typically implemented using SRAMs cells to miminimize area and power and the RTL includes only an instantiation of these components as a black box and/or its behavioural description.

For every fault injection where a net has been forced to a given value, we compare the outputs of architectural registers (general purpose and control registers) and the data and address signals of the core at the on-chip boundaries to the ones obtained with a fault-free simulation.

In typical processor architectures memory operations are performed reading or writting architectural registers. Based on this, an inmediate hypothesis that can be drawn is that roughly all errors that will be visible at the on-chip bus boundaries will also be reflected in the register file and/or control/status registers. If this hypothesis is confirmed it would mean that faults in the core can be easily mimicked using microarchitectural simulators or even functional simulators as both types of simulator tools have access to the architectural register file of the processor. However, if we pay attention to a typical core pipeline implementation we realise that correlating fault injections performed at RTL nets using only the architectural registers might not be a straightforward task and, in fact, it might even be unfeasible. Figure H.2 shows a schematic of a generic processor pipeline

and its interface with the on-chip bus to reach main memory. Note that, while in a typical processor pipeline all memory operations are performed through writing and reading general-purpose registers, the actual implementation makes on-chip bus communication to occur through D-cache and I-cache modules in the fetch and memory stages, respectively. The previous implementation view illustrates that not all faults affecting core nets will reach the architectural registers as some of these nets have logic paths that go directly to the on-chip bus. In the next section we analyze and evaluate in detail fault propagation for the faults occuring within the core and show the exact fraction of faults that can be covered by performing error injection at the architectural registers.

H.4 Experimental Results

In this section we present results characterizing fault propagation in order to confirm the hypothesis presented in the previous section.

H.4.1 Experimental Setup

For our experiments we use an RTL model of a 32-bit LEON3 SparcV8 microcontroller, since it is used in the context of safety-relevant systems and both the microarchitectural simulator and the RTL description of this processor are available [184]. The LEON3 processor consists mainly of a 7-stage pipeline for integer operations (integer unit, *IU*) plus data and instruction caches. In this processor all instructions use all pipeline stages, since we use a minimal configuration where a floating point unit is not present. The RTL processor description follows the structural VHDL design guidelines and it models our target of injection (*IU*) as an entity. The test framework used in the paper is the one shown in Figure H.3. Injection and analysis points in this framework are consistent with the methodology explained in Section H.3. To make analysis costs of register faults affordable we have used the LEON3 with a flat register file configuration[1] as this reduces the number of total registers that need to be tracked in every simulation.

The workload chosen for investigation includes programs from 2 different benchmark suites: the Mälardalen WCET group suite [73], suitable to test real-time system properties and the EEMBC Autobench suite [129], which reflects current real-world demand of some automotive CRTES. The selected programs are: a finite impulse response filter over a 700 items long sample (*fir*), a matrix multiplication of 4x4 size (*matmult*), a matrix initialization of 20 elements (*initmat*), a vehicle speed calculator (*rspeed*) and a CAN bus reader (*canrdr*).

[1]Note that a typical SPARC configuration uses a windowed register file configuration with around 144 32-bit registers. Tracking the contents of 144 32-bit registers even for relatively small benchmarks is unfeasible.

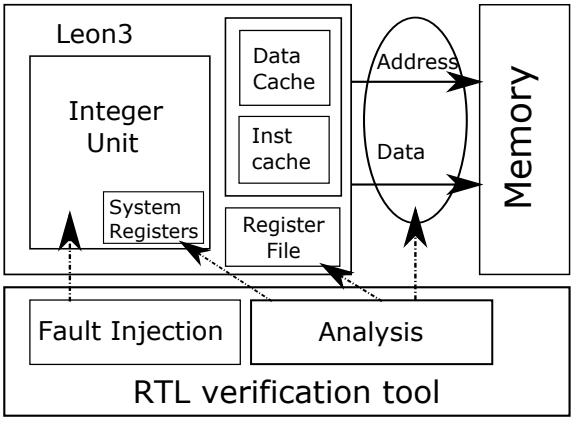


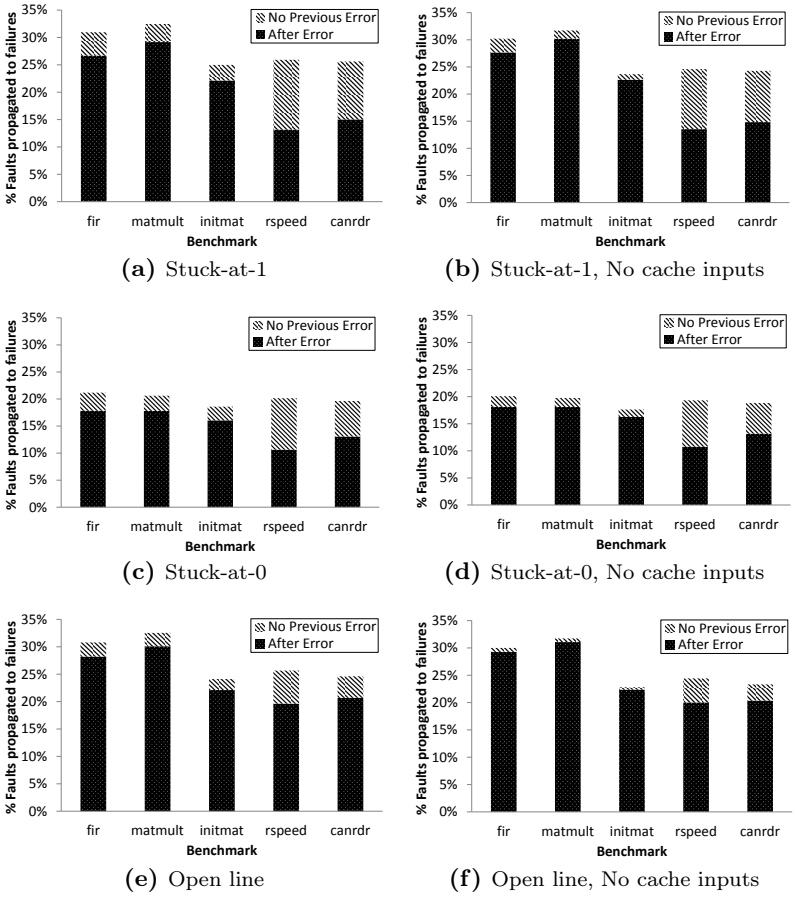
Figure H.3: RTL robustness verification framework

Regarding the faultload, several permanent hardware fault models have been selected, specifically single stuck-at-1, stuck-at-0 and open line. These have been injected using simulator commands as in [86]. The campaign for each fault model and workload has consisted of one experiment per injection node in the *IU* (since permanent faults are applied), totaling 5,246 nodes. As the focus of the experiments is to characterize fault propagation, each experiment applies a single injection in a fixed instant: just before the execution of the main procedure, after the initialization.

H.4.2 Results

Capturing failure probability. Figure H.4 shows the percentage of experiments ending in failure, broken down into those that showed a previous error –in the register file or system registers– and those that did not. This result is specially important as it provides relevant information about how accurately we can capture the behavior of faults occurring within the core pipeline by injecting errors in the architectural registers of a microarchitectural simulator. Plots in the first row (so (a), (b) and (c)) show the percentages of experiments ending in failure when faults are injected in all the nets available in the LEON3 *IU*. As shown in these plots a non-negligible number of failures (dashed lines) were not preceded by an error. The percentage of experiments not showing an error that ended in a failure ranges from 13% (stuck-at-1 faults in *rspeed*) to just 2% (open line faults in *matmult*).

This indicates that the effect of faults within the core cannot be captured with register-based error injection solely. Furthermore, we note that in all core nets in the *IU* also the inputs to the Data cache and Instruction cache modules are being injected directly. To weight the impact this fact causes we remove these nets from



(a) Stuck-at-1 **(b)** Stuck-at-1, No cache inputs

(c) Stuck-at-0 **(d)** Stuck-at-0, No cache inputs

(e) Open line **(f)** Open line, No cache inputs

Figure H.4: Percentage of failures in the experiments according to whether they caused a prior error or not.

the injection in the plots shown in the second row ((d), (e) and (f)). As expected, the number of failures not reflected in architectural register errors decreases.

In particular, the fraction of these failures decreases by around 1 to 3 percentage points with respect to the case when all nets are injected. Thus, the number of remaining failures without error is still significant. Even more important, the fraction of failures without error changes across benchmark and does not correlate with the number of failures with error. For instance, the fraction of failures without error for *rspeed* w.r.t. the failures with error or w.r.t. the total number of injections is much higher than for *initmat* for all fault types. This indicates that injecting faults only in the register file with a simulator does not provide information about failures without error and such information cannot be inferred indirectly.

Error's profile. After injection and analysis we find the percentage of faults that are propagated to errors for each campaign. Figure H.5 shows the percentage of faults that cause one or more errors for the 3 fault models considered in this study. As shown in the figure, the percentage of faults that propagate to errors is slightly superior to the percentage of faults that propagate to failures through errors – black columns in Figure H.4 (a), (c) and (e) – which agrees with the expected fact that some error manifestations do not end up causing a system failure. In any case, the fraction of experiments with errors not causing system failure is always below 7%, meaning that for the selected fault models most of the bits in the exercised registers become critical.

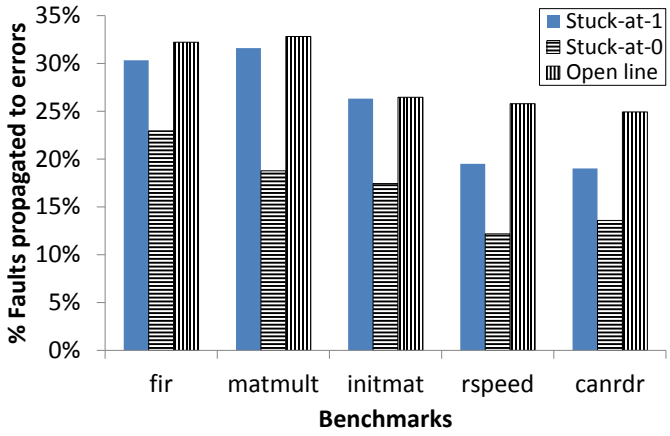
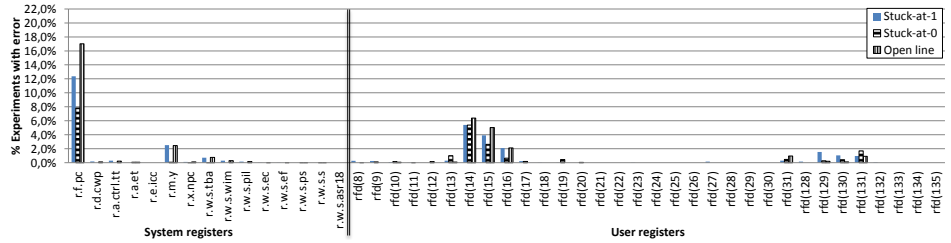


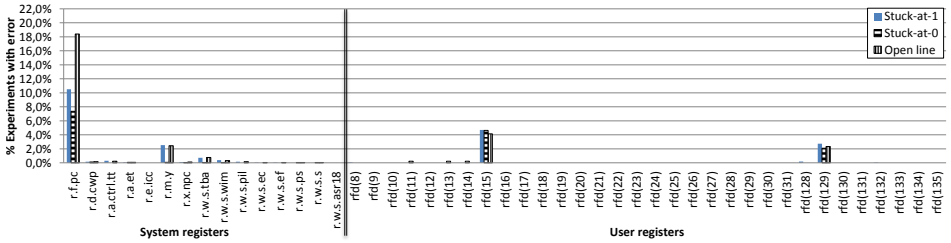
Figure H.5: Percentage of experiments which cause 1 or more errors in the registers

Figure H.6 shows the error distribution across the processor architectural registers for two of the benchmarks analyzed. As expected the program counter (r.f.pc) is one of the registers that accumulates more errors. This is a consequence of two factors: (1) regardless of the benchmark executed the program counter is always severely exercised and (2) a significant number of logic paths exist between *IU* nets and the program counter. On the contrary, in the case of general purpose registers the benchmark exercised determines the exact registers to which faults are propagated. In the example of the plot *canrdr* concentrates a large fraction of the errors in two registers while in *initmat* those (several) registers frequently written during the execution accumulate a significant fraction of errors.

Another important conclusion that holds for the case of permanent faults is that the probability of failure does not depend on how errors reach architectural registers, i.e. how likely registers are affected by an error, but only on the type and amount of instructions that are exercised. In fact, this is in line with the work in [48] that showed that the probability of failure for permanent faults can be



(a) initmat



(b) canrdr

Figure H.6: Errors distribution in system and user registers for different benchmarks

approached by knowing how *diverse* the set of instructions exercised by a given benchmark is.

H.5 Conclusions

The use of microarchitectural simulators has recently arised as a promising approach to reduce the costs associated with the robustness verification of safety critical processors. However, for this low-cost simulation approach to be adopted its accuracy must be validated. In this paper we characterize fault propagation for those faults occurring within the core in order to understand to a what extent microarchitectural simulators can be used in the robustness verification process. To do so, we have injected faults in an RTL processor description and analyze the percentage of faults propagating to errors and failures.

Results in this paper show that while a significant number of faults originated within the core can be covered with verification methodologies focusing on error injection in architectural registers, the achieved coverage is not enough to provide very accurate results. A potential candidate to increase the confidence on verification methodologies using microarchitectural simulators is the use of a combined approach and perform error injection in both cache modules and architectural registers. The immediate practical consequence of this is that functional emula-

tors only are not sufficient to mimic the behavior of all potential core faults and thus, more detailed simulation tools like microarchitectural (timing) simulators are required. We let as future work the validation of the combined error injection approach.

While the results in this work have been obtained for a specific architecture, it includes similar features to the bulk of architectures in the domain, so conclusions can be easily extrapolated to other safety-critical architectures.

Bibliography

[1] In: *Integr. VLSI J.* 37.4 (2004). ISSN: 0167-9260 (cit. on p. 116).

[2] J. Abella et al. "RVC-based time-predictable faulty caches for safety-critical systems". In: *On-Line Testing Symposium (IOLTS), 2011 IEEE 17th International*. 2011, pp. 25–30. DOI: 10.1109/IOLTS.2011.5993806 (cit. on pp. 160, 186).

[3] S.M. Alam et al. "Reliability computer-aided design tool for full-chip electromigration analysis and comparison with different interconnect metallizations". In: *Microelectronics Journal* 38.4-5 (2007), pp. 463–473 (cit. on pp. 9, 72).

[4] M. Alderighi et al. "Evaluation of Single Event Upset Mitigation Schemes for SRAM based FPGAs using the FLIPPER Fault Injection Platform". In: *Defect and Fault-Tolerance in VLSI Systems, 2007. DFT '07. 22nd IEEE International Symposium on*. 2007, pp. 105–113. DOI: 10.1109/DFT.2007.45 (cit. on p. 27).

[5] R. Alexandersson and P. Öhman. "On Hardware Resource Consumption for Aspect-Oriented Implementation of Fault Tolerance". In: *Dependable Computing Conference (EDCC), 2010 European*. 2010, pp. 61–66. DOI: 10.1109/EDCC.2010.17 (cit. on p. 119).

[6] D. de Andres, J.-C. Ruiz, and P. Gil. "Using Dependability, Performance, Area and Energy Consumption Experimental Measures to Benchmark IP Cores". In: *Dependable Computing, 2009. LADC '09. Fourth Latin-American Symposium on*. 2009, pp. 49–56. DOI: 10.1109/LADC.2009.17 (cit. on p. 138).

[7] D. de Andres et al. "FADES: a fault emulation tool for fast dependability assessment". In: *Field Programmable Technology, 2006. FPT 2006. IEEE*

International Conference on. 2006, pp. 221–228. DOI: 10.1109/FPT.2006.270315 (cit. on p. 27).

[8] D. de Andres et al. "Fault Emulation for Dependability Evaluation of VLSI Systems". In: *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on* 16.4 (2008), pp. 422–431. ISSN: 1063-8210. DOI: 10.1109/TVLSI.2008.917428 (cit. on pp. 23, 128, 138).

[9] David de Andrés et al. "An Aspect-Oriented Approach to Hardware Fault Tolerance for Embedded Systems". In: *Handbook of Research on Embedded Systems Design* (2014), p. 123 (cit. on p. 63).

[10] J. Arlat et al. "Comparison of physical and software-implemented fault injection techniques". In: *Computers, IEEE Transactions on* 52.9 (2003), pp. 1115–1133. ISSN: 0018-9340. DOI: 10.1109/TC.2003.1228509 (cit. on p. 22).

[11] J. Arlat et al. "Fault injection for dependability validation: a methodology and some applications". In: *Software Engineering, IEEE Transactions on* 16.2 (1990), pp. 166–182. ISSN: 0098-5589. DOI: 10.1109/32.44380 (cit. on pp. 20, 26).

[12] Algirdas Avizienis. "Design Methods for Fault-Tolerant Navigation Computers". In: *Jet Propulsion Laboratory* (1969) (cit. on p. 123).

[13] Algirdas Avizienis et al. "Basic Concepts and Taxonomy of Dependable and Secure Computing". In: *IEEE Trans. Dependable Secur. Comput.* 1 (1 2004), pp. 11–33. ISSN: 1545-5971 (cit. on pp. 10, 21, 37, 70, 87, 99, 100).

[14] A. Baldini et al. "ldquo;BOND rdquo;: An interposition agents based fault injector for Windows NT". In: *Defect and Fault Tolerance in VLSI Systems, 2000. Proceedings. IEEE International Symposium on*. 2000, pp. 387–395. DOI: 10.1109/DFTVS.2000.887179 (cit. on p. 26).

[15] J.-C. Baraza et al. "Enhancement of Fault Injection Techniques Based on the Modification of VHDL Code". In: *IEEE Transactions on VLSI* 16.6 (2008), pp. 693–706. ISSN: 1063-8210. DOI: 10.1109/TVLSI.2008.2000254 (cit. on pp. 28, 161, 182, 188).

[16] *Basic DES Cryptography Core*. [Online] Opencores. 2010 (cit. on pp. 132, 135).

[17] R. Baumann. "Soft errors in advanced computer systems". In: *Design Test of Computers, IEEE* 22.3 (2005), pp. 258–266. ISSN: 0740-7475. DOI: 10.1109/MDT.2005.69 (cit. on pp. 10, 16).

[18] Alfredo Benso and Paolo Prinetto. *Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation*. Kluwer Academic Publishers, 2003. ISBN: 1402075898 (cit. on pp. 21, 170).

[19] M. Berg et al. "Effectiveness of Internal Versus External SEU Scrubbing Mitigation Strategies in a Xilinx FPGA: Design, Test, and Analysis". In: *IEEE Transactions on Nuclear Science* 55 (4 2008), pp. 2259–2266 (cit. on pp. 44, 48, 73).

[20] C. Bolchini, A. Miele, and M.D. Santambrogio. "TMR and Partial Dynamic Reconfiguration to mitigate SEU faults in FPGAs". In: *IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems*. 2007, pp. 87–95 (cit. on pp. 44, 46, 73).

[21] Cristiana Bolchini, Fabio Salice, and Donatella Sciuto. "Fault Analysis for Networks with Concurrent Error Detection". In: *IEEE Des. Test* 15.4 (1998), pp. 66–74. ISSN: 0740-7475. DOI: 10.1109/54.735929 (cit. on pp. 44, 88).

[22] Andrea Bondavalli et al. "Threshold-Based Mechanisms to Discriminate Transient from Intermittent Faults". In: *IEEE Trans. Comput.* 49.3 (2000), pp. 230–245. ISSN: 0018-9340. DOI: 10.1109/12.841127 (cit. on pp. 47, 90).

[23] S. Borkar. "Designing reliable systems from unreliable components: the challenges of transistor variability and degradation". In: *Micro, IEEE* 25.6 (2005), pp. 10 –16. ISSN: 0272-1732. DOI: 10.1109/MM.2005.110 (cit. on pp. 9, 86).

[24] Demid Borodin and Ben H.H. Juurlink. "Protective Redundancy Overhead Reduction Using Instruction Vulnerability Factor". In: *CF*. 2010 (cit. on pp. 182, 188).

[25] S. Buchner et al. "Comparison of error rates in combinational and sequential logic". In: *Nuclear Science, IEEE Transactions on* 44.6 (1997), pp. 2209–2216. ISSN: 0018-9499. DOI: 10.1109/23.659037 (cit. on pp. 13, 98).

[26] Carl Carmichael. *Triple Module Redundancy Design Techniques for Virtex FPGAs*. Tech. rep. Xilinx Corporation, 2006 (cit. on p. 79).

[27] R.N. Charette. “This Car Runs on Code”. In: *IEEE Spectrum online*. 2009 (cit. on p. 170).

[28] J.A. Cheatham and J.M. Emmert. “A Survey of Fault Tolerant Methodologies for FPGAs”. In: *ACM Transactions on Design Automation of Electronic Systems* 11.2 (2006) (cit. on p. 74).

[29] Shigeru Chiba. “A Metaobject Protocol for C++”. In: *Proceedings of the Tenth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*. OOPSLA ’95. Austin, Texas, USA: ACM, 1995, pp. 285–299. ISBN: 0-89791-703-0. DOI: 10.1145/217838.217868 (cit. on p. 118).

[30] Hyungmin Cho et al. “Quantitative evaluation of soft error injection techniques for robust system design”. In: *Design Automation Conference (DAC), 2013 50th ACM / IEEE*. 2013, pp. 1–10 (cit. on pp. 161, 188).

[31] P Civera, L Macchiarulo, and M Violante. “A simplified gate-level fault model for crosstalk effects analysis”. In: *17th IEEE International Symposium On Defect And Fault Tolerance In Vlsi Systems, Proceedings*. 2002, pp. 31–39. ISBN: 0-7695-1831-1 (cit. on p. 10).

[32] Pierluigi Civera et al. “FPGA-based fault injection techniques for fast evaluation of fault tolerance in VLSI circuits”. In: *Field-Programmable Logic and Applications*. Springer. 2001, pp. 493–502 (cit. on p. 27).

[33] The Embedded Microprocessor Benchmark Consortium. *AutoBench?1.1 Benchmark Software*. 2013 (cit. on p. 138).

[34] C. Constantinescu. “Intermittent faults and effects on reliability of integrated circuits”. In: *Proceedings of the 2008 Annual Reliability and Maintainability Symposium*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 370–374. ISBN: 978-1-4244-1460-4. DOI: 10.1109/RAMS.2008.4925824 (cit. on pp. 10, 87, 100).

[35] Cristian Constantinescu. “Impact of Deep Submicron Technology on Dependability of VLSI Circuits”. In: *Proceedings of the 2002 International Conference on Dependable Systems and Networks*. DSN ’02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 205–209. ISBN: 0-7695-1597-5 (cit. on p. 9).

[36] Cristian Constantinescu. “Trends and Challenges in VLSI Circuit Reliability”. In: *IEEE Micro* 4.23 (2003), pp. 14–19 (cit. on pp. 70, 72, 116, 146).

[37] A. Correcher et al. “Intermittent Failure Dynamics Characterization”. In: *Reliability, IEEE Transactions on* 61.3 (2012), pp. 649 –658. ISSN: 0018-9529. DOI: 10.1109/TR.2012.2208300 (cit. on pp. 10, 47, 90).

[38] Diamantino Costa et al. “Xception^{TM}: A Software Implemented Fault Injection Tool”. English. In: *Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation*. Ed. by Alfredo Benso and Paolo Prinetto. Vol. 23. Frontiers in Electronic Testing. Springer US, 2003, pp. 125–139. ISBN: 978-1-4020-7589-6. DOI: 10.1007/0-306-48711-X_8 (cit. on p. 26).

[39] Sergio D’Angelo, Giacomo R. Sechi, and Cecilia Metra. “Transient and Permanent Fault Diagnosis for FPGA-Based TMR Systems”. In: *Proceedings of the 14th International Symposium on Defect and Fault-Tolerance in VLSI Systems*. DFT ’99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 330–338. ISBN: 0-7695-0325-X (cit. on pp. 45, 47, 87, 92, 100).

[40] *DO-254 Design Assurance Guidance for Airborne Electronic Hardware*. RTCA, 2000 (cit. on p. 3).

[41] P. E. Dodd et al. “Production and Propagation of Single-Event Transients in High-Speed Digital Logic ICs”. In: *IEEE Transactions on Nuclear Science* 51 (2004), pp. 3278–3284. DOI: 10.1109/TNS.2004.839172 (cit. on pp. 12, 89, 98).

[42] David Dye. *Partial Reconfiguration of Xilinx FPGAs Using ISE Design Suite*. Tech. rep. WP374 (v1.1), 2011 (cit. on p. 75).

[43] P. Eaton et al. “Single event transient pulsewidth measurements using a variable temporal latch technique”. In: *Nuclear Science, IEEE Transactions on* 51.6 (2004), pp. 3365–3368. ISSN: 0018-9499. DOI: 10.1109/TNS.2004.840020 (cit. on p. 12).

[44] Frank Emnett and Mark Biegel. *Power Reduction Through RTL Clock Gating By*. 2000 (cit. on p. 132).

[45] Michael Engel and Olaf Spinczyk. “Aspects in Hardware: What Do They Look Like?” In: *Proceedings of the 2008 AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*. ACP4IS ’08. Brussels, Belgium: ACM, 2008, 5:1–5:6. ISBN: 978-1-60558-142-2. DOI: 10.1145/1404891.1404896 (cit. on p. 119).

[46] Luis Entrena, Celia L?z, and Emilio Ol? "Automatic generation of fault tolerant VHDL designs". In: *in RTL? in: Forum on Design Languages (FDL?001.* 2001 (cit. on p. 119).

[47] Dan Ernst et al. "Razor: circuit-level correction of timing errors for low-power operation". In: *IEEE Micro* 6 (2004), pp. 10–20 (cit. on pp. 44, 46).

[48] J. Espinosa et al. "Analysis and RTL Correlation of Instruction Set Simulators for Automotive Microcontroller Robustness Verification". In: *DAC.* `http://people.ac.upc.edu/jabella/DAC2015BSC.pdf`. 2015 (cit. on pp. 188, 195).

[49] J. Espinosa et al. "Ideas Towards Early Detection of Fugacious Faults for Increased Safety of VLSI Systems". In: *ITACA WIICT - Workshop on Innovation on Information and Communication Technologies.* Valencia, Spain, 2014, pp. 25–34 (cit. on p. 100).

[50] J. Espinosa et al. "Robust communications using automatic deployment of a CRC-generation technique in IP-blocks". In: *XI Jornadas de Computaci?Reconfigurable y Aplicaciones (JCRA).* 2011. ISBN: 978-8-46148-8-148 (cit. on pp. 61, 124).

[51] J. Espinosa et al. "Tolerating multiple faults with proximate manifestations in FPGA-based critical designs for harsh environments". In: *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on.* 2012, pp. 292–299. DOI: `10.1109/FPL.2012.6339195` (cit. on p. 61).

[52] Jaime Espinosa, David de Andrés, and Pedro Gil. "Increasing the Dependability of VLSI Systems Through Early Detection of Fugacious Faults". In: *Proceedings of the 11th European Dependable Computing Conference (EDCC11) Paris, France. 7-11 September 2015.* 2015 (cit. on p. 62).

[53] Jaime Espinosa et al. "Analysis and RTL Correlation of Instruction Set Simulators for Automotive Microcontroller Robustness Verification". In: *Proceedings of the 52Nd Annual Design Automation Conference.* DAC '15. San Francisco, California: ACM, 2015, 40:1–40:6. ISBN: 978-1-4503-3520-1. DOI: `10.1145/2744769.2744798` (cit. on pp. 62, 64, 163).

[54] Jaime Espinosa et al. "Characterizing fault propagation in safety-critical processor designs". In: *Proceedings of the 21st International Online Testing Symposium (IOLTS15) pages=144–149, Elia, Halkidiki, Greece. 6-8 July 2015.* IEEE. 2015 (cit. on pp. 62, 64, 163).

[55] Jaime Espinosa et al. "On the potentials of Robustness Verification using Architectural Registers-based Fault Injection". In: *Design Test, IEEE* TBA.TBA (2015), TBA (cit. on p. 63).

[56] Jaime Espinosa et al. "The Challenge of Detection and Diagnosis of Fugacious Hardware Faults in VLSI Designs". In: *Dependable Computing*. Ed. by Marco Vieira and Jo?oCarlos Cunha. Vol. 7869. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 76–87. ISBN: 978-3-642-38788-3. DOI: 10.1007/978-3-642-38789-0_7 (cit. on pp. 62, 99).

[57] Jaime Espinosa et al. "Towards Certification-aware Fault Injection Methodologies Using Virtual Prototypes". In: *Proceedings of the Forum on specification & Design Languages (FDL2015) Barcelona,Spain. 14-16 September 2015*. 2015 (cit. on pp. 62, 64).

[58] J.-C. Fabre and T. Perennou. "A metaobject architecture for fault-tolerant distributed systems: the FRIENDS approach". In: *Computers, IEEE Transactions on* 47.1 (1998), pp. 78–95. ISSN: 0018-9340. DOI: 10.1109/12.656088 (cit. on pp. 116, 119).

[59] V. Ferlet-Cavrois, L.W. Massengill, and P. Gouker. "Single Event Transients in Digital CMOS -A Review". In: *Nuclear Science, IEEE Transactions on* 60.3 (2013), pp. 1767–1790. ISSN: 0018-9499. DOI: 10.1109/TNS.2013.2255624 (cit. on pp. 12, 13, 16, 100).

[60] V. Ferlet-Cavrois et al. "Direct measurement of transient pulses induced by laser and heavy ion irradiation in deca-nanometer devices". In: *Nuclear Science, IEEE Transactions on* 52.6 (2005), pp. 2104–2113. ISSN: 0018-9499. DOI: 10.1109/TNS.2005.860682 (cit. on pp. 12, 98).

[61] P. Fiser, P. Kubalik, and H. Kubatova. "An Efficient Multiple-Parity Generator Design for On-Line Testing on FPGA". In: *Digital System Design Architectures, Methods and Tools, 2008. DSD '08. 11th EUROMICRO Conference on*. 2008, pp. 96–99. DOI: 10.1109/DSD.2008.46 (cit. on p. 44).

[62] J. Freijedo et al. "Impact of Power Supply Voltage Variations on FPGA-Based Digital Systems Performance". In: *Journal of Low Power Electronics* 6.2 (2010), pp. 339–349. DOI: doi:10.1166/jolpe.2010.1076 (cit. on pp. 10, 98).

[63] D. Gil et al. "Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation". In: ed. by Alfredo Benso and Paolo Prinetto. Kluwer Academic Publishers, 2003. Chap. 4, pp. 159–176 (cit. on p. 81).

[64] D. Gil et al. “Injecting intermittent faults for the dependability validation of commercial microcontrollers”. In: *High Level Design Validation and Test Workshop, 2008. HLDVT ’08. IEEE International*. 2008, pp. 177–184. DOI: 10.1109/HLDVT.2008.4695899 (cit. on p. 16).

[65] D. Gil et al. “VHDL Simulation-Based Fault Injection Techniques”. In: *Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation*. Ed. by A. Benso and P. Prinetto. Vol. 23. Springer US, 2004, pp. 159–176 (cit. on pp. 24, 126, 154).

[66] Pedro Gil et al. *Fault Representativeness*. Tech. rep. DBench project, IST 2000-25425 [Online]. Available: http://www.laas.fr/DBench, 2002 (cit. on pp. 10, 11, 15, 72, 121, 138, 161, 182, 188).

[67] Michael Goessel et al. *New Methods of Concurrent Checking (Frontiers in Electronic Testing)*. 1st ed. Springer Publishing Company, Incorporated, 2008. ISBN: 1402084196, 9781402084195 (cit. on p. 88).

[68] Joaquin Gracia-Moran et al. “Experimental validation of a fault tolerant microcomputer system against intermittent faults”. In: *DSN*. 2010, pp. 413–418 (cit. on p. 87).

[69] J. Gracia et al. “Analysis of the influence of intermittent faults in a microcontroller”. In: *Proceedings of the 2008 11th IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 1–6. ISBN: 978-1-4244-2276-0 (cit. on p. 12).

[70] P. Graham et al. “Consequences and Categories of SRAM FPGA Configuration SEUs”. In: *Military and Aerospace Programmable Logic Devices International Conference*. 2003, pp. 1–9 (cit. on p. 71).

[71] Mentor Graphics. *Modelsim*. 2014 (cit. on pp. 109, 164).

[72] Steve Guccione, Delon Levi, and Prasanna Sundararajan. “JBits: Java based interface for reconfigurable computing”. In: 1999 (cit. on p. 27).

[73] Jan Gustafsson et al. “The Mälardalen WCET Benchmarks – Past, Present and Future”. In: *WCET2010*. Ed. by Björn Lisper. Brussels, Belgium: OCG, 2010, pp. 137–147 (cit. on pp. 165, 192).

[74] R.W. Hamming. “Error detecting and error correcting codes”. In: *Bell System Technical Journal* 29 (1950), pp. 147–160 (cit. on p. 44).

[75] Olof Hannius and Johan Karlsson. "Impact of Soft Errors in a Jet Engine Controller". In: *Computer Safety, Reliability, and Security*. Ed. by Frank Ortmeier and Peter Daniel. Vol. 7612. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 223–234. ISBN: 978-3-642-33677-5. DOI: 10.1007/978-3-642-33678-2_19 (cit. on pp. 16, 86).

[76] Y. Hayashi et al. "Intentional electromagnetic interference for fault analysis on AES block cipher IC". In: *Electromagnetic Compatibility of Integrated Circuits (EMC Compo), 2011 8th Workshop on*. 2011, pp. 235–240 (cit. on p. 98).

[77] C. Hernandez and J. Abella. "LiVe: Timely error detection in light-lockstep safety critical systems". In: *DAC*. 2014 (cit. on pp. 28, 160, 163, 171, 174, 179, 186).

[78] C. Hernandez and J. Abella. "Timely Error Detection for Effective Recovery in Light-Lockstep Automotive Systems". In: *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 34.11 (2015), pp. 1718–1729. ISSN: 0278-0070. DOI: 10.1109/TCAD.2015.2434958 (cit. on p. 39).

[79] *IEC 61511 Functional safety - Safety instrumented systems for the process industry sector*. IEC, 2004 (cit. on p. 3).

[80] *IEC 62278 Railway applications - Specification and demonstration of reliability, availability, maintainability and safety (RAMS)*. IEC, 2002 (cit. on p. 3).

[81] *ISO 26262 Functional Safety - Road Vehicles*. ISO (cit. on pp. 3, 38, 161, 170, 171, 173, 174, 176, 182, 187).

[82] Xilinx Inc. *Device Reliability Report*. Tech. rep. UG116 (v8.1), 2011 (cit. on p. 72).

[83] Infineon. *AURIX - TriCore datasheet. Highly Integrated and Performance Optimized 32-bit Microcontrollers for Automotive and Industrial Applications*. https://www.infineon.com/dgdl?folderId=db3a304412b407950112b409ae660 fileId=db3a30431f848401011fc664882a7648. 2012 (cit. on pp. 163, 175, 190, 191).

[84] R. K. Iyer and D. J. Rossetti. "A statistical load dependency model for cpu errors at SLAC". In: *Fault-Tolerant Computing, 1995, ' Highlights*

from Twenty-Five Years'., Twenty-Fifth International Symposium on. 1995, p. 373 (cit. on p. 89).

[85] JEDEC. "Measurement and reporting of alpha particle and terrestrial cosmic ray-induced soft errors in semiconductor devices". In: *JEDEC Standard JESD89A*. JEDEC, 2006 (cit. on pp. 86, 98).

[86] E. Jenn et al. "Fault injection into VHDL models: the MEFISTO tool". In: *FTCS*. 1994 (cit. on pp. 28, 161, 164, 166, 178, 182, 187, 193).

[87] C.W. Johnson and C.M. Holloway. "The Dangers of Failure Masking in Fault-Tolerant Software: Aspects of a Recent In-Flight Upset Event". In: *System Safety, 2007 2nd Institution of Engineering and Technology International Conference on*. 2007, pp. 60 –65 (cit. on p. 88).

[88] E. Johnson et al. "Accelerator validation of an FPGA SEU simulator". In: *Nuclear Science, IEEE Transactions on* 50.6 (2003), pp. 2147 –2157. ISSN: 0018-9499. DOI: 10.1109/TNS.2003.821791 (cit. on p. 71).

[89] H. S. Warren Jr. "Hacker's Delight". In: Addison-Wesley Professional, 2003. Chap. 14. ISBN: 0201914654 (cit. on p. 147).

[90] L. Kafka, M. Danek, and O. Novak. "A Novel Emulation Technique that Preserves Circuit Structure and Timing". In: *System-on-Chip, 2007 International Symposium on*. 2007, pp. 1–4. DOI: 10.1109/ISSOC.2007.4427437 (cit. on p. 128).

[91] G.A. Kanawati, N.A. Kanawati, and J.A. Abraham. "FERRARI: a flexible software-based fault and error injection system". In: *Computers, IEEE Transactions on* 44.2 (1995), pp. 248–260. ISSN: 0018-9340. DOI: 10.1109/12.364536 (cit. on p. 26).

[92] T. Karnik and P. Hazucha. "Characterization of soft errors caused by single event upsets in CMOS processes". In: *Dependable and Secure Computing, IEEE Transactions on* 1.2 (2004), pp. 128–143. ISSN: 1545-5971. DOI: 10.1109/TDSC.2004.14 (cit. on p. 121).

[93] Fernanda Gusmao de Lima Kastensmidt et al. "Designing Fault-Tolerant Techniques for SRAM-Based FPGAs". In: *IEEE Des. Test* 21 (6 2004), pp. 552–562. ISSN: 0740-7475 (cit. on pp. 44, 73, 79).

[94] G. Kiczales. "Aspect-oriented Programming". In: *ACM Comput. Surv.* 28.4es (1996). ISSN: 0360-0300. DOI: 10.1145/242224.242420 (cit. on pp. 117, 118, 120).

[95] C. Kim. "Detection and location of intermittent faults by monitoring carrier signal channel behavior of electrical interconnection system". In: *Electric Ship Technologies Symposium, 2009. ESTS 2009. IEEE*. 2009, pp. 449 – 455. DOI: 10.1109/ESTS.2009.4906550 (cit. on p. 93).

[96] K Kimseng et al. "Physics-of-failure assessment of a cruise control module". In: *Microelectronics Reliability* 39.10 (1999), pp. 1423 –1444. ISSN: 0026-2714. DOI: 10.1016/S0026-2714(99)00018-9 (cit. on p. 90).

[97] Seok-Bum Ko and Jien-Chung Lo. "Efficient Realization of Parity Prediction Functions in FPGAs". In: *J. Electron. Test.* 20.5 (2004), pp. 489–499. ISSN: 0923-8174. DOI: 10.1023/B:JETT.0000042513.15382.e7 (cit. on pp. 92, 102).

[98] M. Kooli et al. "Software testing and software fault injection". In: *Design Technology of Integrated Systems in Nanoscale Era (DTIS), 2015 10th International Conference on*. 2015, pp. 1–6. DOI: 10.1109/DTIS.2015.7127370 (cit. on p. 20).

[99] P. Koopman and T. Chakravarty. "Cyclic Redundancy Code (CRC) Polynomial Selection For Embedded Networks". In: *IEEE International Conference on Dependable Systems and Networks*. 2004, pp. 145–154 (cit. on pp. 147, 153, 156).

[100] G. Leen and D. Heffernan. "Expanding Automotive Electronic Systems". In: *IEEE Computer* 35.1 (2002) (cit. on p. 170).

[101] A. Lesea et al. "The rosetta experiment: Atmospheric soft error rate testing in differing technology FPGAs". In: *IEEE Transactions on Device and Materials Reliability* 5.3 (2005), pp. 317–328 (cit. on pp. 10, 72).

[102] R. Leveugle. "Automatic modifications of high level VHDL descriptions for fault detection or tolerance". In: *Design, Automation and Test in Europe Conference and Exhibition, 2002. Proceedings*. 2002, pp. 837–841. DOI: 10.1109/DATE.2002.998396 (cit. on p. 119).

[103] Man-Lap Li et al. "Accurate microarchitecture-level fault modeling for studying hardware faults". In: *HPCA*. 2009 (cit. on pp. 28, 41, 161, 174, 182, 188).

[104] LiP6. *SoCLib*. http://www.soclib.fr/trac/dev. 2003-2012 (cit. on p. 160).

[105] Fernanda de Lima Kastensmidt, Luigi Carro, and Ricardo Reis. *Fault-Tolerance Techniques for SRAM-based FPGAs*. Vol. 32. Frontiers in Electronic Testing. Springer, 2006 (cit. on pp. 46, 102).

[106] C. Lopez-Ongil et al. "Autonomous Fault Emulation: A New FPGA-Based Acceleration System for Hardness Evaluation". In: *Nuclear Science, IEEE Transactions on* 54.1 (2007), pp. 252–261. ISSN: 0018-9499. DOI: 10.1109/TNS.2006.889115 (cit. on p. 27).

[107] Henrique Madeira et al. "RIFLE: A General Purpose Pin-level Fault Injector". In: *Proceedings of the First European Dependable Computing Conference on Dependable Computing*. EDCC-1. London, UK, UK: Springer-Verlag, 1994, pp. 199–216. ISBN: 3-540-58426-9 (cit. on p. 26).

[108] Pattie Maes. "Concepts and Experiments in Computational Reflection". In: *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*. OOPSLA '87. Orlando, Florida, USA: ACM, 1987, pp. 147–155. ISBN: 0-89791-247-0. DOI: 10.1145/38765.38821 (cit. on p. 118).

[109] M. Maniatakos et al. "Instruction-Level Impact Analysis of Low-Level Faults in a Modern Microprocessor Controller". In: *Computers, IEEE Transactions on* 60.9 (2011), pp. 1260–1273. ISSN: 0018-9340. DOI: 10.1109/TC.2010.60 (cit. on pp. 41, 162, 183).

[110] Michail Maniatakos et al. "AVF Analysis Acceleration via Hierarchical Fault Pruning". In: *16th European Test Symposium, ETS 2011, Trondheim, Norway, May 23-27, 2011*. 2011, pp. 87–92. DOI: 10.1109/ETS.2011.42 (cit. on p. 21).

[111] R. J. Mart?z et al. "Dependable computing for critical applications". In: ed. by A. Avizienis, H. Kopetz, and J. C. Laprie. Vol. 12. Dependable computing and Fault-Tolerant Systems 7. ISBN : 0-7695-0284-9. IEEE Computer Society Press, 1999. Chap. Experimental Validation of High-Speed Fault-Tolerant Systems Using Physical Fault Injection, pp. 249–265 (cit. on p. 26).

[112] Matthias Meier, Stefan Hanenberg, and Olaf Spinczyk. "AspectVHDL Stage 1: The Prototype of an Aspect-oriented Hardware Description Language". In: *Proceedings of the 2012 Workshop on Modularity in Systems Software*. MISS '12. Potsdam, Germany: ACM, 2012, pp. 3–8. ISBN: 978-1-4503-1217-2. DOI: 10.1145/2162024.2162028 (cit. on p. 119).

[113] J.M. Mogollon et al. “FTUNSHADES2: A novel platform for early evaluation of robustness against SEE”. In: *Radiation and Its Effects on Components and Systems (RADECS), 2011 12th European Conference on*. 2011, pp. 169–174. DOI: 10.1109/RADECS.2011.6131392 (cit. on pp. 23, 27).

[114] David P. Montminy et al. “Using Relocatable Bitstreams for Fault Tolerance”. In: *Proceedings of the Second NASA/ESA Conference on Adaptive Hardware and Systems*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 701–708. ISBN: 0-7695-2866-X (cit. on p. 75).

[115] Gordon Moore. “Cramming more components onto integrated circuits”. In: *Electronics* 38.8 (1965), p. 114. DOI: 10.1109/JPROC.1998.658762 (cit. on p. 9).

[116] T.R. Muck et al. “A Case Study of AOP and OOP Applied to Digital Hardware Design”. In: *Computing System Engineering (SBESC), 2011 Brazilian Symposium on*. 2011, pp. 66–71. DOI: 10.1109/SBESC.2011.23 (cit. on p. 119).

[117] S.S. Mukherjee et al. “A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor”. In: *MICRO*. 2003 (cit. on pp. 175, 182, 188).

[118] V. Narayanan and Y. Xie. “Reliability Concerns in Embedded Systems Design”. In: *IEEE Computer* 1.39 (2006), pp. 118–120 (cit. on pp. 86, 116, 146).

[119] *New tool for FPGA designers mitigates soft errors within synthesis*. [Online] Available at DSP-FPGA.com Magazine. 2011 (cit. on p. 120).

[120] M. Nicolaidis. “Time Redundancy Based Soft-Error Tolerance to Rescue Nanometer Technologies”. In: *IEEE VLSI Test Symposium*. 1999, pp. 86–94 (cit. on pp. 46, 73, 79).

[121] M. Nicolaidis, S. Manich, and J. Figueras. “Achieving Fault Secureness in Parity Prediction Arithmetic Operators: General Conditions and Implementations”. In: *Proceedings of the 1996 European conference on Design and Test*. EDTC ’96. Washington, DC, USA: IEEE Computer Society, 1996, pp. 186–. ISBN: 0-8186-7423-7 (cit. on pp. 92, 102).

[122] Edmund B. Nightingale, John R. Douceur, and Vince Orgovan. “Cycles, cells and platters: an empirical analysisof hardware failures on a million consumer PCs”. In: *Proceedings of the sixth conference on Computer sys-*

tems. EuroSys '11. Salzburg, Austria: ACM, 2011, pp. 343–356. ISBN: 978-1-4503-0634-8. DOI: 10.1145/1966445.1966477 (cit. on p. 89).

[123] J.-H. Oetjens et al. "Safety Evaluation of Automotive Electronics Using Virtual Prototypes: State of the Art and Research Challenges". In: *DAC*. 2014 (cit. on pp. 160, 172, 182, 187).

[124] C. Oliveira et al. "Reliability analysis of an on-chip watchdog for embedded systems exposed to radiation and EMI". In: *Electromagnetic Compatibility of Integrated Circuits (EMC Compo), 2013 9th Intl Workshop on*. 2013, pp. 89–94. DOI: 10.1109/EMCCompo.2013.6735179 (cit. on p. 10).

[125] *OpenCores*. [Online] Available: www.opencores.org. 2011 (cit. on pp. 149, 153).

[126] Terence Parr. *The Definitive ANTLR 4 Reference*. The Pragmatic Programmers, 2013 (cit. on p. 122).

[127] Ab?o Parreira, J. P. Teixeira, and Marcelino Santos. "A Novel Approach to FPGA-Based Hardware Fault Modeling and Simulation". In: *Proc. of the Design and Diagnostics of Electronic Circuits and Syst. Workshop*. 2003, pp. 17–24 (cit. on p. 128).

[128] Christopher Pohl, Carlos Paiz, and Mario Porrmann. "vMAGIC - Automatic Code Generation for VHDL". English. In: *International Journal of Reconfigurable Computing, Hindawi Publishing Corporation,* 2009.Article ID 205149 (2009), pp. 1–9. ISSN: 1687-7195. DOI: 10.1155/2009/205149 (cit. on p. 119).

[129] J. Poovey. *Characterization of the EEMBC Benchmark Suite*. North Carolina State University. 2007 (cit. on pp. 165, 178, 192).

[130] M. Portela-Garcia et al. "Fault Injection in Modern Microprocessors Using On-Chip Debugging Infrastructures". In: *Dependable and Secure Computing, IEEE Transactions on* 8.2 (2011), pp. 308–314. ISSN: 1545-5971. DOI: 10.1109/TDSC.2010.50 (cit. on p. 27).

[131] M. Psarakis et al. "Microprocessor Software-Based Self-Testing". In: *Design Test of Computers, IEEE* 27.3 (2010), pp. 4–19. ISSN: 0740-7475. DOI: 10.1109/MDT.2010.5 (cit. on pp. 171, 176).

[132] Laura L. Pullum. *Software Fault Tolerance Techniques and Implementation*. Norwood, MA, USA: Artech House, Inc., 2001. ISBN: 1-58053-137-7 (cit. on p. 129).

[133] Heather Quinn et al. "Radiation-Induced Multi-Bit Upsets in SRAM-Based FPGAs". In: *IEEE Transactions on Nuclear Science* 52.6 (2005), pp. 2455–2461 (cit. on p. 73).

[134] RTCA and EUROCAE. *DO-178B / ED-12B, Software Considerations in Airborne Systems and Equipment Certification*. 1992 (cit. on pp. 161, 187).

[135] P. Ramachandran et al. "Statistical Fault Injection". In: *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*. 2008, pp. 122–127. DOI: 10.1109/DSN.2008.4630080 (cit. on p. 161).

[136] Layali Rashid, Karthik Pattabiraman, and Sathish Gopalakrishnan. "Intermittent hardware errors and recovery: modelling and evaluation". In: *International Conference on Quantitative Evaluation of Systems (QEST)*. 2012 (cit. on pp. 15, 91).

[137] S. Rehman et al. "Reliable software for unreliable hardware: Embedded code generation aiming at reliability". In: *CODES+ISSS*. 2011 (cit. on pp. 176, 182, 188).

[138] D. A. Reynolds and G. Metze. "Fault Detection Capabilities of Alternating Logic". In: *IEEE Trans. Comput.* 27 (12 1978), pp. 1093–1098. ISSN: 0018-9340 (cit. on p. 44).

[139] GmbH Robert Bosch. *CAN Specification 2.0*. Robert Bosch, GmbH, 1991 (cit. on p. 146).

[140] M. Rodriguez, A. Albinet, and J. Arlat. "MAFALDA-RT: a tool for dependability assessment of real-time systems". In: *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*. 2002, pp. 267–272. DOI: 10.1109/DSN.2002.1028909 (cit. on p. 26).

[141] S. Rohr et al. "An Integrated Approach to Automotive Safety Systems". In: *SAE Automotive Engineering International magazine* (2000) (cit. on p. 170).

[142] E. Romani. *Structural PIC165X microcontroller, The Hamburg VHDL Archive*. 2007 (cit. on p. 136).

[143] J.-C. Ruiz, D. de Andrés, and P. Gil. "Design and Deployment of a Generic ECC-based Fault Tolerance Mechanism for Embedded HW Cores". In: *IEEE International Conference on Emerging Technologies and Factory Automation*. Mallorca, Spain, 2009, pp. 3956–3964 (cit. on pp. 120, 124, 130, 146).

[144] J.-C. Ruiz et al. "Generic Design and Automatic Deployment of NMR Strategies on HW Cores". In: *IEEE Pacific Rim Int. Symp. on Dependable Computing*. Taipei, Taiwan, 2008, pp. 265–272 (cit. on pp. 120, 124, 130, 146).

[145] J.-C. Ruiz et al. "Reflective Fault-Tolerant Systems: From Experience to Challenges". In: *IEEE Transactions on Computers* 52.2 (2003), pp. 237–254. ISSN: 0018-9340 (cit. on p. 146).

[146] J.-C. Ruiz et al. "Using Open Compilation to Simplify the Design of Fault-Tolerant VLSI Systems". In: *Workshop on Compiler and Architectural Techniques for Application Reliability and Security*. 2008, B14–B19. ISBN: 978-1-4244-2398-9 (cit. on pp. 120, 121, 146).

[147] STMicroelectronics. *32-bit Power Architecture microcontroller for automotive SIL3/ASILD chassis and safety applications*. 2014 (cit. on pp. 175, 190).

[148] Luis-J. Saiz-Adalid et al. "Flexible Unequal Error Control Codes with Selectable Error Detection and Correction Levels". In: *Proceedings of the 32Nd International Conference on Computer Safety, Reliability, and Security - Volume 8153*. SAFECOMP 2013. Toulouse, France: Springer-Verlag New York, Inc., 2013, pp. 178–189. ISBN: 978-3-642-40792-5. DOI: 10.1007/978-3-642-40793-2_17 (cit. on p. 46).

[149] Daniel Sánchez et al. "Modeling the Impact of Permanent Faults in Caches". In: *ACM Trans. Archit. Code Optim.* 10.4 (2013), 29:1–29:23. ISSN: 1544-3566. DOI: 10.1145/2541228.2541236 (cit. on pp. 160, 186).

[150] B. Sangchoolie et al. "A Study of the Impact of Bit-Flip Errors on Programs Compiled with Different Optimization Levels". In: *EDCC*. 2014 (cit. on pp. 21, 160, 171, 186).

[151] Behrooz Sangchoolie et al. "A Comparison of Inject-on-Read and Inject-on-Write in ISA-Level Fault Injection". In: *Dependable Computing Conference (EDCC), 2015 Eleventh European*. 2015, pp. 178–189. DOI: 10.1109/EDCC.2015.24 (cit. on p. 23).

[152] J. Savir. "Detection of Single Intermittent Faults in Sequential Circuits". In: *IEEE Trans. Comput.* 29.7 (1980), pp. 673–678. ISSN: 0018-9340. DOI: 10.1109/TC.1980.1675642 (cit. on pp. 44, 90).

[153] Zary Segall et al. "FIAT-fault injection based automated testing environment." In: *FTCS*. IEEE Computer Society, 1988, pp. 102–107. ISBN: 0-8186-0867-6 (cit. on p. 26).

[154] Krishna Seshan, Timothy J. Maloney, and Kenneth J. Wu. "The Quality and Reliability of Intel's Quarter Micron Process". In: *Intel technology Journal* (1998) (cit. on p. 9).

[155] S. Shamshiri and Kwang-Ting Cheng. "Error-locality-aware linear coding to correct multi-bit upsets in SRAMs". In: *Test Conference (ITC), 2010 IEEE International*. 2010, pp. 1–10. DOI: 10.1109/TEST.2010.5699220 (cit. on p. 46).

[156] P. Shivakumar et al. "Modeling the effect of technology trends on the soft error rate of combinational logic". In: *Proceedings of the 2002 International Conference on Dependable Systems and Networks*. 2002, pp. 389–398 (cit. on p. 16).

[157] V. Sieh, O. Tschache, and F. Balbach. "VERIFY: evaluation of reliability using VHDL-models with embedded fault descriptions". In: *Fault-Tolerant Computing, 1997. FTCS-27. Digest of Papers., Twenty-Seventh Annual International Symposium on*. 1997, pp. 32–36. DOI: 10.1109/FTCS.1997.614074 (cit. on p. 28).

[158] D. Skarin, R. Barbosa, and J. Karlsson. "GOOFI-2: A tool for experimental dependability assessment". In: *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*. 2010, pp. 557–562. DOI: 10.1109/DSN.2010.5544265 (cit. on pp. 23, 26).

[159] B.A. Sorensen et al. "An analyzer for detecting intermittent faults in electronic devices". In: *AUTOTESTCON '94. IEEE Systems Readiness Technology Conference. 'Cost Effective Support Into the Next Century', Conference Proceedings.* 1994, pp. 417 –421. DOI: 10.1109/AUTEST.1994.381590 (cit. on pp. 11, 90).

[160] Janusz Sosnowski. "Transient Fault Tolerance in Digital Systems". In: *IEEE Micro* 14.1 (1994), pp. 24–35. ISSN: 0272-1732. DOI: 10.1109/40.259897 (cit. on p. 90).

[161] University of South California. *Tools for Open Reconfigurable Computing*. 2014 (cit. on p. 108).

[162] J Srinivasan et al. "The impact of technology scaling on lifetime reliability". In: *2004 International Conference On Dependable Systems And Networks, Proceedings*. 2004, pp. 177–186. ISBN: 0-7695-2052-9 (cit. on pp. 9, 72).

[163] Suresh Srinivasan et al. "FLAW: FPGA lifetime awareness". In: *Proceedings of the 43rd annual Design Automation Conference*. DAC '06. San Francisco, CA, USA: ACM, 2006, pp. 630–635. ISBN: 1-59593-381-6 (cit. on p. 72).

[164] L. Sterpone and M. Violante. "A New Reliability-Oriented Place and Route Algorithm for SRAM-Based FPGAs". In: *IEEE Transactions on Computers* 55 (6 2006), pp. 732–744 (cit. on pp. 72, 77).

[165] L. Sterpone and M. Violante. "A new analytical approach to estimate the effects of SEUs in TMR architectures implemented through SRAM-based FPGAs". In: *Nuclear Science, IEEE Transactions on* 52.6 (2005), pp. 2217–2223. ISSN: 0018-9499. DOI: 10.1109/TNS.2005.860745 (cit. on p. 24).

[166] M. Straka, J. Kastil, and Z. Kotasek. "Fault Tolerant Structure for SRAM-Based FPGA via Partial Dynamic Reconfiguration". In: *Euromicro Conference on Digital System Design: Architectures, Methods and Tools*. 2010, pp. 365–372 (cit. on pp. 44, 48, 73, 79).

[167] M. Straka, Z. Kotasek, and J. Winter. "Digital Systems Architectures Based on On-line Checkers". In: *EUROMICRO Conference on Digital System Design Architectures, Methods and Tools*. 2008, pp. 81–87 (cit. on pp. 44, 79).

[168] Vytautas Štuikys et al. "Soft IP Design Framework Using Metaprogramming Techniques". In: *In*. Kluwer Academic Publishers, 2002, pp. 257–266 (cit. on p. 119).

[169] Synopsys. *Platform Architect*. *http://www.synopsys.com/Prototyping/ArchitectureDesign/Pages/platform-architect.aspx* (cit. on pp. 160, 161).

[170] F. Taïani, J.-C. Fabre, and M.-O. Killijian. "A multi-level meta-object protocol for fault-tolerance in complex architectures". In: *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on*. 2005, pp. 270–279. DOI: 10.1109/DSN.2005.10 (cit. on p. 117).

[171] A. S. Tanenbaum. *Computer Networks*. Prentice Hall, 2002. Chap. 3.2. ISBN: 9780130661029 (cit. on pp. 146, 147).

[172] Pascale Thévenod-Fosse and Hélène Waeselynck. "An Investigation of Statistical Software Testing". In: *Softw. Test., Verif. Reliab.* 1.2 (1991), pp. 5–25 (cit. on p. 153).

[173] J. Tobola et al. "Online Protocol Testing for FPGA Based Fault Tolerant Systems". In: *Digital System Design Architectures, Methods and Tools, 2007. DSD 2007. 10th Euromicro Conference on*. 2007, pp. 676–679. DOI: 10.1109/DSD.2007.4341541 (cit. on p. 44).

[174] Nur A. Touba and Edward J. McCluskey. "Logic Synthesis of Multilevel Circuits with Concurrent Error Detection". In: *IEEE TRANS. CAD* 16.7 (1997), pp. 783–789 (cit. on pp. 44, 91).

[175] *VeTeSS project: www.vetess.eu*. ARTEMIS Joint Undertaking (cit. on pp. 160, 174, 182).

[176] A. Vörg. *http://toolip.fzi.de*. ToolIP- Tools and Methods for IP (cit. on p. 116).

[177] C.R. Yount and D.P. Siewiorek. "A methodology for the rapid injection of transient hardware errors". In: *Computers, IEEE Transactions on* 45.8 (1996), pp. 881–891. ISSN: 0018-9340. DOI: 10.1109/12.536231 (cit. on p. 28).

[178] A. Youssef et al. "Communication Integrity in Networks for Critical Control Systems". In: *European Dependable Computing Conference*. 2006, pp. 23–34 (cit. on p. 146).

[179] Chaohuang Zeng, Nirmal Saxena, and E.J. McCluskey. "Finite State Machine Synthesis with Concurrent Error Detection". In: *International Test Conference*. 1999, pp. 672–679 (cit. on p. 44).

[180] http://www.oracle.com/us/products/tools/oracle-grid-engine-075549.html. *Sun Grid Engine*. Oracle (cit. on p. 164).

[181] http://opencores.org/opencores,wishbone. *Wishbone System-on-Chip (SoC) Interconnect Architecture for Portable IP Cores*. OpenCores (cit. on p. 142).

[182] `http://reliability.ee.byu.edu/edif/`. *BYU EDIF Tools Home Page*. Brigham Young University, FPGA Reliability Studies (cit. on pp. 46, 120).

[183] `http://www.arm.com/products/system-ip/amba`. *CoreLink System IP and Design Tools for AMBA*. ARM Ltd. (cit. on p. 142).

[184] `http://www.gaisler.com/cms/index.php?option=com_content&task=view&id=13&Itemid=53`. *Leon3 Processor*. Aeroflex Gaisler (cit. on pp. 165, 178, 187, 192).

[185] `http://www.gaisler.com/index.php/products/processors/leon3ft`. *Leon3 fault-tolerant Processor*. Aeroflex Gaisler (cit. on pp. 163, 191).

[186] `http://www.ocpip.org`. *Open Core Protocol International Partnership (OCP-IP)*. OpenCores (cit. on p. 142).

[187] `http://www.synopsys.com/Tools/Implementation/RTLSynthesis/Pages/default.aspx`. *Accelerate Design Innovation with Design Compiler*. Synopsys Inc. (cit. on p. 126).

[188] `http://www.xilinx.com/ise/optional_prod/tmrtool.htm`. *XTMR Tool*. Xilinx, Inc. (cit. on pp. 46, 120).