UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

Dependability-driven strategies to improve the design and verification of safety-critical HDL-based embedded systems

Author: Ilya Tuzov

Advisors: Dr. David de Andrés Martínez
Dr. Juan Carlos Ruiz García

Valencia, November 2020

Resumen

La utilización de sistemas empotrados en cada vez más ámbitos de aplicación está llevando a que su diseño deba enfrentarse a mayores requisitos de rendimiento, consumo de energía y área de silicio ocupada (PPA). Asimismo, su utilización en aplicaciones críticas provoca que deban cumplir con estrictos requisitos de confiabilidad para garantizar su correcto funcionamiento durante períodos prolongados de tiempo. En particular, el uso de dispositivos lógicos programables de tipo FPGA como tecnología de implementación final resulta un gran desafío desde la perspectiva de la confiabilidad, ya que la memoria de configuración de estos dispositivos es muy sensible a la radiación. Por todo ello, la confiabilidad debe considerarse como uno de los criterios principales para la toma de decisiones a lo largo del todo flujo de diseño, que debe complementarse con diversos procesos que soporten y permitan alcanzar estrictos requisitos de confiabilidad.

Primero, la evaluación de la robustez del diseño frente a los fallos permite identificar sus puntos débiles, guiando así la definición de mecanismos de tolerancia a fallos. Segundo, la eficacia de los mecanismos definidos debe validarse experimentalmente. Tercero, la evaluación comparativa de la confiabilidad (dependability benchmarking) permite a los diseñadores seleccionar los componentes prediseñados (IP), las tecnologías de implementación y las herramientas de diseño (EDA) más adecuadas (desde la perspectiva de la confiabilidad) entre aquellas alternativas existentes. Por último, la exploración del espacio de diseño (DSE) puede desplegarse para configurar de manera óptima los parámetros de los componentes y las herramientas seleccionados, mejorando así la confiabilidad y las métricas PPA de la implementación resultante.

Todos los procesos anteriormente mencionados se basan en técnicas de inyección de fallos para poder evaluar la robustez del sistema diseñado. A pesar de que existe una amplia variedad de técnicas y herramientas de inyección de fallos, ninguna de ellas permite cubrir completamente las necesidades planteadas en el flujo de diseño semicustom. Aquellas soluciones basadas en simulación (SBFI) normalmente están limitadas a trabajar con modelos hardware de alto nivel, proporcionando estimaciones de robustez imprecisas, siendo altamente intrusivas y/o específicas para alguna tecnología de implementación particular. Las técnicas de inyección de fallos basadas en FPGAs (FFI) deben abordar problemas relacionados con la granularidad del análisis, no permitiendo la localización precisa de los puntos débiles del diseño y considerando puntos de inyección innecesarios (no esenciales).

Otro desafío es la reducción del coste temporal de los experimentos de inyección de fallos. Teniendo en cuenta la alta complejidad de los diseños actuales, el tiempo experimental dedicado a la evaluación de la confiabilidad puede ser excesivo incluso en aquellos escenarios más simples, mientras que puede ser simplemente inviable en aquellos procesos relacionados con la evaluación de múltiples configuraciones alternativas del diseño (benchmarking y DSE).

Por último, estos procesos orientados a la confiabilidad carecen de un soporte instrumental (herramientas) que permita cubrir el flujo de diseño con toda su variedad de lenguajes de descripción de hardware, tecnologías de implementación y herramientas de diseño.

Esta tesis aborda los retos anteriormente mencionados, con el fin de integrar de manera eficaz estos procesos orientados a la confiabilidad en el flujo de diseño. Primeramente, se proponen nuevos métodos de inyección de fallos que permiten una evaluación de la confiabilidad precisa y detallada en diferentes niveles del flujo de diseño. Segundo, se definen nuevas técnicas para la aceleración de los experimentos de inyección que mejoran su coste temporal. Tercero, se define dos estrategias DSE que permiten configurar de manera óptima (desde la perspectiva de la confiabilidad) los componentes IP y las herramientas EDA, con un coste experimental mínimo. Cuarto, se propone un kit de herramientas (DAVOS) que automatiza e incorpora con eficacia los procesos orientados a la confiabilidad en el flujo de diseño semicustom. Finalmente, se demuestra la utilidad y eficacia de las propuestas mediante un caso de estudio en el que se implementan tres procesadores empotrados en un FPGA de Xilinx serie 7.

Resum

La utilització de sistemes encastats en cada vegada més àmbits d'aplicació està portant al fet que el seu disseny haja d'enfrontar-se a majors requisits de rendiment, consum d'energia i àrea de silici ocupada (PPA). Així mateix, la seua utilització en aplicacions crítiques provoca que hagen de complir amb estrictes requisits de confiabilitat per a garantir el seu correcte funcionament durant períodes prolongats de temps. En particular, l'ús de dispositius lògics programables de tipus FPGA com a tecnologia d'implementació final resulta un gran desafiament des de la perspectiva de la confiabilitat, ja que la memòria de configuració d'aquests dispositius és molt sensible a la radiació. Per tot això, la confiabilitat ha de considerar-se com un dels criteris principals per a la presa de decisions al llarg del tot flux de disseny, que ha de complementar-se amb diversos processos que suporten i permeten aconseguir estrictes requisits de confiabilitat.

Primer, l'avaluació de la robustesa del disseny enfront de les fallades permet identificar els seus punts febles, guiant així la definició de mecanismes de tolerància a fallades. Segon, l'eficàcia dels mecanismes definits ha de validar-se experimentalment. Tercer, l'avaluació comparativa de la confiabilitat (dependability benchmarking) permet als dissenyadors seleccionar els components predissenyats (IP), les tecnologies d'implementació i les eines de disseny (EDA) més adequades (des de la perspectiva de la confiabilitat) entre aquelles alternatives existents. Finalment, l'exploració de l'espai de disseny (DSE) pot desplegar-se per a configurar de manera òptima els paràmetres dels components i les eines seleccionats, millorant així la confiabilitat i les mètriques PPA de la implementació resultant.

Tots els processos anteriorment esmentats es basen en tècniques d'injecció de fallades per a poder avaluar la robustesa del sistema dissenyat. A pesar que existeix una àmplia varietat de tècniques i eines d'injecció de fallades, cap d'elles permet cobrir completament les necessitats plantejades en el flux de disseny semicustom. Aquelles solucions basades en simulació (SBFI) normalment estan limitades a treballar amb models maquinari d'alt nivell, proporcionant estimacions de robustesa imprecises, sent altament intrusives i/o específiques per a alguna tecnologia d'implementació particular. Les tècniques d'injecció de fallades basades en FPGAs (FFI) han d'abordar problemes relacionats amb la granularitat de l'anàlisi, no permetent la localització precisa dels punts febles del disseny i considerant punts d'injecció innecessaris (no essencials).

Un altre desafiament és la reducció del cost temporal dels experiments d'injecció de fallades. Tenint en compte l'alta complexitat dels dissenys actuals, el temps experimental dedicat a l'avaluació de la confiabilitat pot ser excessiu fins i tot en aquells escenaris més simples, mentre que pot ser simplement inviable en aquells processos relacionats amb l'avaluació de múltiples configuracions alternatives del disseny (benchmarking i DSE).

Finalment, aquests processos orientats a la confiabilitat manquen d'un suport instrumental (eines) que permeta cobrir el flux de disseny amb tota la seua varietat de llenguatges de descripció de maquinari, tecnologies d'implementació i eines de disseny.

Aquesta tesi aborda els reptes anteriorment esmentats, amb la finalitat d'integrar de manera eficaç aquests processos orientats a la confiabilitat en el flux de disseny. Primerament, es proposen nous mètodes d'injecció de fallades que permeten una avaluació de la confiabilitat precisa i detallada en diferents nivells del flux de disseny. Segon, es defineixen noves tècniques per a l'acceleració dels experiments d'injecció que milloren el seu cost temporal. Tercer, es defineix dues estratègies DSE que permeten configurar de manera òptima (des de la perspectiva de la confiabilitat) els components IP i les eines EDA, amb un cost experimental mínim. Quart, es proposa un kit d'eines (DAVOS) que automatitza i incorpora amb eficàcia els processos orientats a la confiabilitat en el flux de disseny semicustom. Finalment, es demostra la utilitat i eficàcia de les propostes mitjançant un cas d'estudi en el qual s'implementen tres processadors encastats en un FPGA de Xilinx sèrie 7.

Abstract

Embedded systems steadily extend their application areas, dealing with increasing requirements to their performance, power consumption and area (PPA). Whenever embedded systems are used in safety-critical applications, they must also meet rigorous dependability requirements, thus to guarantee their correct service during an extended period of time. It becomes especially challenging to meet the dependability requirements for those systems that use SRAM-based Field Programmable Gate Arrays (FPGAs) as the target implementation technology, since they are very susceptible to Single Event Upsets (SEUs) in their configuration memory. This leads to increased dependability threats, especially in harsh environments. In such a way, dependability should be considered as one of the primary design goals for embedded systems, driving the design decisions throughout the whole design flow.

To meet the rigorous dependability requirements, the common semicustom design flow should be accompanied by several dependability-driven processes. First, dependability assessment should quantify the robustness of the design against faults, and identify its weak points, thus supporting the definition of fault mitigation mechanisms. Second, dependability-driven verification ensures the correctness and efficiency of fault mitigation mechanisms. Third, dependability benchmarking allows designers to select the most suitable (from a dependability perspective) IP cores, implementation technologies, and electronic design automation (EDA) tools from available alternatives. Finally, dependability-aware design space exploration (DSE) can be deployed to optimally configure the parameters of the selected IP cores and EDA tools, to improve as much as possible the dependability and PPA features of resulting implementation.

The aforementioned dependability-driven processes rely on fault injection testing to quantify the robustness of the designed systems. Despite nowadays there exists a wide variety of fault injection solutions, several important problems still should be addressed to completely cover the needs of a dependability-driven design flow. In particular, simulation-based fault injection (SBFI) methodologies must be adapted to implementation-level HDL models, to enable the accurate and low-intrusive simulation of logic faults in technology-specific macrocells. Likewise, the assessment of FPGA-based designs requires to refine the granularity of FPGA-based fault injection (FFI) to accurately locate the relevant fault targets within the configuration memory, as well as to map detected weak points (critical bits) onto the source HDL design.

Another important challenge, that should be addressed to efficiently integrate dependability-driven processes into the design flow, is the reduction of SBFI and FFI experimental effort. Considering the high complexity of modern hardware designs, the required fault injection effort may exceed the experimental time budget even in simple dependability assessment scenarios. Thus, in presence of alternative design configurations (especially in case of DSE dealing with thousands of alternatives) the fault injection experimental effort becomes infeasibly high.

Finally, the aforementioned dependability-driven processes lack an instrumental support (tools) covering the semicustom design flow in all its variety of description languages, implementation technologies, and EDA tools. Existing fault injection tools only partially cover the individual stages of the design flow, being usually specific to a particular level of the design representation and implementation technology.

This work addresses the aforementioned challenges, in order to efficiently integrate dependability-driven processes into the design flow. First, it proposes new SBFI and FFI approaches that enable an accurate and detailed dependability assessment at different levels of the design flow. Second, it improves the performance of dependability-driven processes by defining new techniques for accelerating SBFI and FFI experiments. Third, it defines two DSE strategies, that enable optimal dependability-aware tuning of IP cores and EDA tools, while reducing as much as possible the robustness evaluation effort. Fourth, it proposes a new toolkit (DAVOS), that automates and seamlessly integrates the aforementioned dependability-driven processes into the semicustom design flow. Finally, it illustrates the usefulness and efficiency of these proposals through a case study consisting of three soft-core embedded processors implemented on a Xilinx 7-series SoC FPGA.

Contents

List of Figures

List of Tables

List of Acronyms

ASIC	Application-specific integrated circuit
BEL	Basic element of logic
BRAM	Block RAM
CLB	Configurable logic block
CM	Configuration memory
CSV	Comma-separated values (file)
DSE	Design space exploration
DSP	Digital Signal Processing block
DUT	Design under test
DVF	Device vulnerability factor
DoE	Design of experiments
ECC	Error correction code
EDA	Electronic design automation
FAR	Frame address register
FFI	FPGA-based fault injection
FF	Flip-Flop
FIT	Failure in time unit (one failure each one billion of device hours)
FPGA	Field Programmable Gate Array
FSM	Finite state machine

GA Genetic algorithm
GPIO General-purpose input/output
GSR Global set-reset
HDL Hardware description language
HW Hardware
ICAP Internal configuration access port
IC Integrated circuit
IP Intellectual property
MCDM Multi-criteria decision making
MTTF Mean time to failure
NSGA Non-dominated sorting genetic algorithm
PCAP Processor configuration access port
PPAD Performance, power, area, and dependability attributes
RTL Register transfer level
SBFI Simulation-based fault injection
SDC Silent data corruption
SDF Standard delay format
SEU Single event upset
SGE Sun grid engine
SoC System on chip
TMR Triple modular redundancy
UDP User-defined primitive
VITAL VHDL initiative towards ASIC libraries
WSM Weighted sum model
XML Extensible markup language

Chapter 1

Introduction

1.1 Motivation

Embedded systems have become increasingly widespread across diverse application domains, ranging from consumer electronics to aerospace systems. The design of embedded systems supposes that they must perform their dedicated (predefined) functions within the enclosing product in a most efficient way. On the one hand, they should minimize the usage of hardware (and software) resources to keep reasonable the cost of the resulting product. On the other hand, when embedded into portable (autonomous) devices, they must be also energy-efficient. These properties must be achieved without compromising the ability of the system to confidently perform the required functionality, often under hard real-time constraints [108]. At the same time, embedded system must meet the dependability requirements of the target application domain. *Dependability* is understood as the ability of the system to deliver a service that can justifiably be trusted [13]. In non-critical applications (consumer electronics) the lack of dependability may lead to financial losses and risks for the vendor's reputation. In safety-critical applications (automotive systems, power plants, medical equipment) an invalid or absent system service is completely unacceptable, as it may lead to catastrophic consequences on humans and the environment.

The causes and consequences of incorrect system behaviour are described by the chain of dependability threats [13], depicted in Fig.1.1, which comprises three main concepts: faults, errors, and failures. *Faults* are defects introduced in the system during development, manufacturing, or operation. A fault may remain dormant (or masked) without any effects on the system behaviour, but once activated, causes a deviation from the correct logical values of system's internal nodes; this deviation is referred to as *error*. Errors may cause an incorrect behaviour of the system, and after propagating to the system interface, cause a *failure*, which is perceived by the user (external system) as the deviation of the delivered service from the correct one. The failure of a system service causes a permanent or transient external fault for other systems that receive that service. Errors may be also detected by the system, and corrected, and/or reported to the user (higher-level system) in order to alert her that the delivered service cannot be trusted.

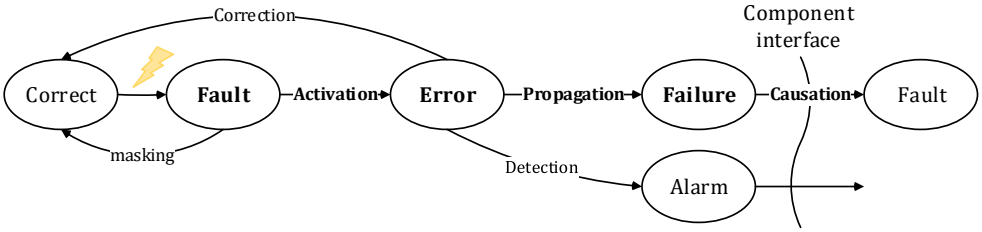


Figure 1.1: Causality relationship between faults, errors, and failures

Various faults in the system originate from development flaws, manufacturing defects, and external effects [13]. Development faults are due to imperfections in the designed software and hardware. Despite this work focuses on hardware dependability, it is worth noting that software faults (incorrect type casting, unhandled exception, memory leakage, etc.) may pose an important dependability threat, as it has been reported for the widely-known Ariane 5 flight failure [102]. Hardware development faults may result from design flaws and bugs in the source HDL model of the system, or in the used third-party IP cores. Indeed, in order to reduce the development cost and time to market, the industry often makes use of third-party IP cores (soft-core processors, communication controllers, etc.) within the automated design flow. The side effect of integrating such IPs is that they allow a very limited inspectability and, being often poorly verified by the designer, may seriously compromise the reliability of the resulting system.

The steadily scaling of manufacturing technologies for VLSI Integrated Circuits (IC) brings another set of benefits and challenges for the designers of embedded system. On the upside, it favours higher packaging densities, higher clock speeds,

and lower voltages, resulting in improved overall performance and lower power consumption. On the downside, shrinking technologies also imply an increasing vulnerability to hardware faults [120]. Even though the manufacturing *open*, *short* and *bridging* defects [173] are detected at post-production testing (being less critical for the consumer), the increased sensitivity of ICs to external (environmental) effects become more critical. First, the higher cross-section increases the rate of soft errors (single and multiple-bit upsets) in registers and memory cells, due to the impact of ionizing particles [121]. Second, lower voltages make the circuit more sensitive to electromagnetic (EM) noise, resulting in higher rates of transient logic faults, while high-power EM pulses may also permanently damage the IC [91]. Third, shrinking technology also intensifies the wear-out process due to accumulated environmental effects; this increases even more the rate of soft errors, and makes intermittent faults more pronounced until they finally manifest as permanent faults [152].

The problem of IC sensitivity to external effects becomes particularly challenging in the context of the increasing usage of Field Programmable Gate Arrays (FPGA) in embedded systems. Programmable devices have always been considered as a suitable alternative to ASICs, because of (i) quick development cycle, (ii) cost-efficiency when the system is produced in small volumes or in unique samples, and (iii) reconfiguration capabilities allowing to deliver post-production updates to the designed systems, thus to extend their lifetime. This trend has intensified with the advances of SRAM-based FPGA technologies, revealing superior performance and ability to allocate the required functionality within the FPGA chip on demand at run-time. At the same time, SRAM-based FPGAs are known to be very sensitive to single event upsets (SEUs) in their configuration memory [70]. This problem is especially challenging for mission-critical and safety-critical systems used in harsh environments [127].

To avoid service failures in presence of faults, the designed systems must be protected by fault tolerance mechanisms, which perform error detection and recovery. Most tolerance mechanisms exploit some kind of redundancy: hardware redundancy (N-modular replication), data redundancy (error detection and correction codes), temporal redundancy (repeated execution, checkpointing/rollback). Each mechanism has its pros and cons in terms of error correction capabilities, and performance/power/area penalties. Hardware designs often combine several mechanisms attending to their specific features. For instance, as it is pointed in [69], modular redundancy is more appropriate for the protection of blocks with pipelines and individual registers, because of its low delay penalty; while ECC, introducing much lower resource overhead, is more appropriate for register files and memories. Some design-hardening techniques specifically target FPGA-based

designs. For instance, scrubbing of configuration memory (CM) takes advantage of FPGA partial reconfiguration capabilities to locate the corrupted CM frames, and recover them from the reference bitstream [68] [77]. In case of permanent and intermittent faults, error handling may also be accompanied by fault handling, to prevent the faults from being activated again [13]. For instance, FPGA-based systems may dynamically relocate the failing modules into other (non-corrupted) areas of the chip [115].

Many commercial IP cores are supplied with integrated fault tolerance mechanisms, for instance Gaisler LEON3-FT [40] and Xilinx Microblaze [181] soft-core processors are protected against SEUs in registers, caches and on-chip RAM by means of error correction codes. Leading EDA tool manufacturers, trying to gain momentum in the aerospace and automotive industry, also offer specific products to instrument the resulting designs (at the netlist level) with fault tolerance mechanisms (TMR, safe state machines, ECC), like Xilinx's XTMR tool [183], Mentor Graphics' Precision RTL Plus [111], and Synopsys' Synplify Premium [156]. There also exist third-party tools, automating the design hardening, like the one proposed by Brigham Young University [32], which deploys the Triple Modular Redundancy (TMR) in EDIF netlists. Some FPGA vendors also offer IP cores for CM scrubbing, like for instance Xilinx's Single Error Mitigation controller [182]. Nevertheless, not all of these tools are easily available, and most of them lack customization capabilities, so designers often prefer to integrate their custom fault tolerance mechanisms into the designed systems.

The design process must be accompanied by the verification of integrated fault tolerance mechanisms, and by the assessment of dependability features of the resulting system. Verification should be started as soon as it becomes possible in the design flow, in order to reduce the cost of fixing any weak points in the design. At the same time, it should also be accomplished at each subsequent step of the design flow to take into account the impact of involved implementation processes and implementation technologies on dependability. Contemporary design and certification standards in different domains, such as automotive (ISO-26262), aerospace (DO-254), and railway (IEC-62279), require to verify the designed systems in presence of faults throughout the whole design flow and recommend the use of fault injection for that purpose. However, the fault injection process itself is not standardized. Despite nowadays there exist a wide assortment of different fault injection solutions in the hardware domain, none of them is generic enough to cover the complete hardware design flow in all its variety of Hardware Description Languages (HDL), modelling levels, EDA tools, and implementation technologies. Furthermore, the integration of fault injection into the design flow

still faces some important challenges related to the accuracy of involved injection procedures and to the required experimentation effort.

Because of their early availability in the design flow, simulation-based (SBFI) and emulation-based (FFI) fault injection are two useful dependability evaluation techniques along the design flow. The former targets HDL models of the designed systems at different levels of abstraction. The latter targets the FPGA prototype, being used for both acceleration of model-based injection and for the assessment of final implementations.

Despite SBFI can potentially support the dependability assessment at all levels of HDL representation, designers often limit its application to the Register Transfer Level (source) model, thus considering a very limited set of fault models and neglecting the impact of EDA optimizations and implementation technologies on dependability. On the one hand, this is explained by the lack of generic low-intrusive SBFI techniques that could accurately reproduce fault effects within the diverse technology-specific (implementation-level) libraries. On the other hand, this is due to the prohibitive experimentation effort at lower HDL representation levels, resulting from significantly slower simulation speed and much wider fault space.

FFI solutions enable a much higher experimentation performance, but must handle even a wider fault space to evaluate the effects of upsets in configuration memory (CM), which in modern FPGAs may amount to hundreds of megabits. In addition to that, FFI faces accuracy challenges related to the granularity of the deployed analysis, resulting from the lack of information and tools which would allow to map the netlist logic components onto the underlying CM layer.

The existence of a wide variety of alternative IP cores, EDA tools, and implementation technologies, requires designers to select the solution that best meets the design goals. The comparison and selection of alternatives from the dependability perspective constitutes the dependability benchmarking process. Beyond the challenges raised by dependability assessment, benchmarking must address the problem of making decisions with multiple conflicting design goals, including performance, power consumption, area/cost, and dependability (PPAD). At the same time, since multiple alternative implementations must be evaluated, benchmarking makes the problem of fault injection performance even more pronounced.

Finally, IP cores and EDA tools nowadays provide multitude of configuration parameters that may significantly impact the attainable PPAD results. On the one hand, the impact of most parameters on dependability is a priori unknown. For that reason, designers often prefer to keep them at their default levels. On the

other hand, improperly configuring these parameters may negatively impact the quality of resulting implementations. Tuning selected parameters towards better PPAD results constitutes an optimization problem in the space of alternative configurations, and is referred to as design space exploration (DSE). Since the design space growth exponentially with increasing number of available parameters, DSE becomes a very resource-intensive problem even with respect to the simplest design goals, such as silicon area or power consumption. Thus, straightforward DSE approaches become practically infeasible in dependability-aware contexts, due to the very high cost of evaluation of alternatives through fault injection.

1.2 Objectives

The primary goal of this work is to contribute as much as possible to the efficient integration of dependability-driven fault injection-based processes into the hardware design flow. In particular, the following objectives are established:

- Studying the capabilities and limitations of existing fault injection solutions with respect to the dependability assessment at different stages of the hardware design flow, and whenever required, defining additional techniques for an accurate fault simulation/emulation.

  Defined simulation procedures should be low-intrusive and generic enough to properly reproduce the effects of common logic faults in HDL models defined on the basis of an arbitrary technology-specific libraries. Defined fault emulation procedures should be able to selectively target CM cells pertaining to any selected design scope (module) and the major types of FPGA logic primitives.

- Defining new fault injection speed-up techniques and refining existing ones to improve, as much a possible, the performance (reduce the experimentation effort) of dependability assessment at different design representation levels.

- Defining a design space exploration (DSE) methodology for the dependability-aware tuning of EDA tools and IP cores. The DSE methodology should allow multiobjective PPAD optimizations, while reducing as much as possible the experimentation (evaluation) effort. It should also take into account properties of the design space, such as the number of levels adopted by EDA/IP parameters, possible interactions of parameters, and the design space regularity (existence of incompatible configurations).

- Providing an instrumental support for the efficient integration of dependability-driven processes into the semicustom/FPGA-based design flow. Developed tools should be generic enough to support different hardware description languages and abstraction levels, fault models, EDA tools, and implementation technologies.
- Evaluating the effectiveness of proposed techniques and analysis tools in application to benchmark circuits considered representative of embedded systems.

1.3 Structure of the thesis

In addition to the present introduction, the rest of this thesis is structured into nine chapters:

- **Chapter 2:** It studies the background on dependability-driven processes within the semicustom and FPGA-based design flow. First, it provides an overview of the processes, technologies, and standards of the baseline design flow. After that, it presents the terminology and existing solutions in the domain of dependability-assessment, dependability benchmarking, and dependability-aware design space exploration.
- **Chapter 3:** This chapter studies the background on fault injection methodologies, which are at the base of aforementioned dependability-driven processes. First, it introduces the basic concepts, requirements, and approaches of fault injection. Second, it studies in detail existing techniques and accuracy-related challenges in the domain of simulation-based (SBFI) and FPGA-based (FFI) fault injection. Third, it studies existing approaches for improving fault injection performance. Finally, it studies the capabilities and limitations of existing SBFI and FFI tools with respect to their integration into the semicustom design flow.
- **Chapter 4:** It proposes a low-intrusive fault injection approach that allows the accurate simulation of common logic faults in VITAL-based and Verilog-based implementation-level HDL models. After that, it unifies the application of defined fault injection procedures to the diverse macrocells and fault models, through a flexible tool-independent fault dictionary format.
- **Chapter 5:** This chapter improves the accuracy of FFI experiments by (i) establishing a bit-accurate mapping between several of the most important

types of Xilinx netlist primitives and the configuration memory, and (ii) by optimizing the localisation of essential bits within the FPGA configuration memory to deploy bit-accurate FFI experiments for any selected design scope and type of logic primitives.

- **Chapter 6:** This chapter presents the contributions for improving the fault injection performance. The proposed approach reduces the number of injection runs by filtering and prioritizing essential bits through the profiling of the switching activity, and by sampling the fault space in an iterative way until reaching a given confidence interval for derived metrics. In addition to that, each individual experiment is also accelerated by developing the ideas of multi-level fault injection and checkpointing in the context of SBFI and FFI.

- **Chapter 7:** It proposes three DSE techniques that aim at the dependability-aware tuning of EDA tools and IP cores with a minimal experimental effort. The first two techniques rely on i) Design of Experiments (DoE) to representatively sample the design space with the smallest possible number of configurations, ii) statistical methods to quantify the impact of considered parameters on PPAD results, and iii) MCDM techniques to determine the best suitable configuration of parameters. The third technique explores the design space by means of genetic algorithms, optimizing the DSE performance through an iterative dependability-driven selection. Finally, this chapter discusses the advantages and limitations of each technique with respect to the properties of the design space under study.

- **Chapter 8:** This chapter describes the DAVOS toolkit, developed to provide an instrumental support for the seamless integration of considered dependability-driven processes into the semicustom design flow.

- **Chapter 9:** The efficiency of proposed techniques and tools is evaluated in application to three soft-core processors (MC8051, AVR, and Microblaze) considered to be suitable benchmarks in the context of embedded systems.

- **Chapter 10:** This chapter summarizes the conclusions drawn from this work, discusses the advantages and limitations of proposed techniques, and outlines the ways for future research based on proposed approaches.

Chapter 2

Dependability-aware Hardware Design Flow

The semicustom design methodology supplies designers with a wide range of interoperable EDA tools, customizable IP cores, and implementation technologies that significantly accelerate the development cycle and help to meet the required performance, power and area (PPA) requirements. Being applied to the development of critical systems, the baseline design flow should be complemented by several dependability-driven processes, namely: (i) dependability assessment to evaluate the dependability features of the designed system against the requirements of target applications, (ii) dependability benchmarking to select the most suitable IP cores, EDA tools, and implementation technologies attending to dependability criteria, and (iii) dependability-aware design space exploration to optimally configure the EDA tools and IP cores from the viewpoint of the robustness of resulting implementations. This chapter provides the necessary background on dependability-aware design flow. Section 2.1 first introduces the basic processes of semicustom design flow. After that, Sections 2.2, 2.3, and 2.4 detail the terminology and analyse existing solutions in the domain of dependability assessment, benchmarking, and design space exploration, respectively. Finally, Section 2.5 concludes this chapter.

2.1 Semicustom and FPGA-based design flow

Nowadays, the hardware design flow is automated enough to translate highly abstract HW models directly into their final implementation technology with minimum level of designer's effort. The designer's task in this flow is to define the source high-level model of the system and to verify the intermediate implementation results through simulation. This section provides an overview of the model-based HW design flow and its application to SRAM-based FPGAs.

2.1.1 Model-based design

The modern semicustom hardware design flows rely on the use of Hardware Description Languages (HDL) to model the hardware, and on Electronic Design Automation (EDA) tools to translate these models into the target implementation technology. HDL models can be defined at either the implementation level, the logic level, or the Register Transfer Level (RTL). In practice, the source designs are usually defined by high-level RTL models, which describe the circuit in terms of registers and how information flows among them with just clock cycle accuracy. Synthesis tools translate RTL models into functional netlists, comprising a set of inferred interconnected sequential and combinational logic components [165]. The technology mapping process realizes these generic netlists in a specific technology, using a library of primitive components available from different technology vendors. Technology-dependent netlists are then placed (assigning physical locations for each element) and routed (adding connection, power, and clock lines) to obtain a physical netlist that can be translated into a tapeout (for standard cells) or bitstream (for Field-Programmable Gate Arrays (FPGAs)) file. Fig.2.1 illustrates the generic design flow for the case of FPGAs as the target implementation technology.

Due to design reuse concerns [171], source designs can be also represented by a hierarchical amalgamation of third party intellectual property (IP) cores described in different HDLs at different abstraction levels. The high complexity of these heterogeneous hardware designs requires the common semi-custom design flow to rely on the interoperability of highly flexible EDA tools [84]. In such a way, different third-party synthesis tools (e.g. Synopsis' Synplify and MentorGraphics' Precision Synthesis) may interchange the generated netlists with the vendor-specific place/route tools, using the so-called Electronic Design Interchange Format (EDIF).

A set of constraints can be supplied to EDA tools to guide them towards meeting the design requirements. Synthesis constraints usually customize the logic infer-

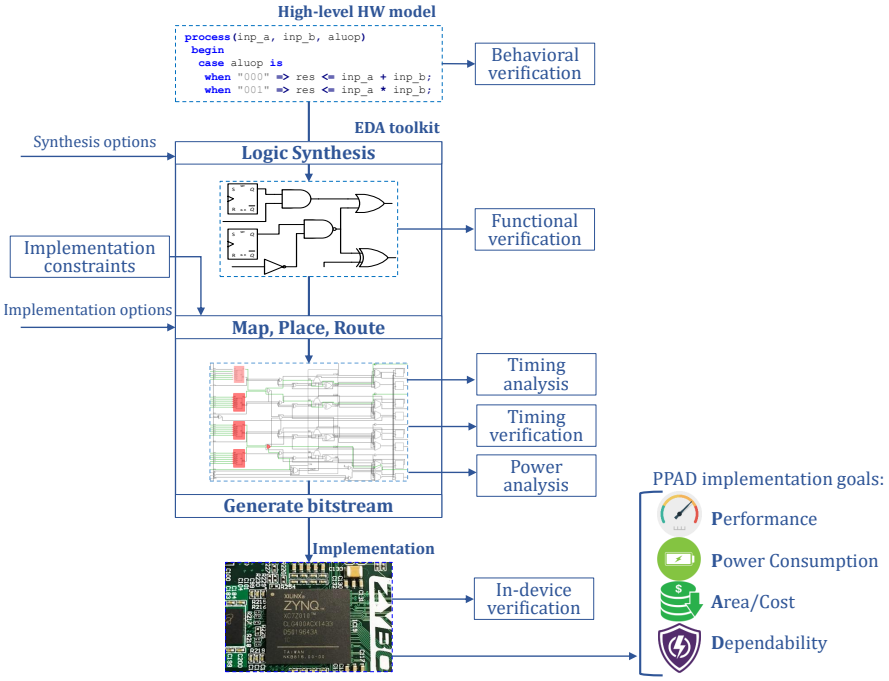


Figure 2.1: FPGA-based design flow

ence. For instance, designers may explicitly specify in the RTL code (by means of synthesis attributes), how a particular memory array should be inferred in an FPGA: using block memory (BRAM), and/or distributed memory (LUTRAM). Timing constraints commonly specify the required clock frequencies, maximum critical path, setup times for input pins, etc. Finally, physical constraints assign the IO pins to the design interface and define the area on the chip layout for the placement of design modules. Additionally, each process in the design flow can be customized by means of EDA tools parameters. These parameters globally tune the logic inference, logic optimizations, enable/disable the usage of certain types of technology components, tune the place-route strategies, and configure the processing effort.

Intermediate implementation results should be verified at each step of the design flow through the simulation of HDL models, exported by EDA tools from the generated netlists. The behavioral simulation verifies the functional correctness of the source design. The post-synthesis simulation verifies the functionality of inferred and optimized gate-level netlist. The post-place-route simulation verifies

the resulting implementation from both functional and timing perspectives. The verification process is usually automated by means of testbenches, which supply a set of input tests to the model and check the responses against the precomputed reference results. More elaborated verification methodologies [20] develop layered coverage-driven testbenches in SystemVerilog, which generate constrained-random input tests, while monitoring the reached functional and structural coverage. The responses of the design under test (DUT) in the layered testbenches are usually verified against the reference results, computed at run-time by the abstract algorithmic model (not necessarily synthesizable). This can be accompanied by a run-time checking of formal properties, defined by designers using Open Vera Assertions (OVA) or SystemVerilog Assertions (SVA) [37], which are particularly useful for the verification of communication protocols and Finite State Machines (FSMs). Some works [28] also propose the synthesis of hardware assertion checkers, which can be used for in-device verification and on-line monitoring.

It must be noted that recent advances in High-level synthesis (HLS) allow definition of source designs at higher levels than RTL – in form of untimed or partially timed System-C models, or even by ANSI C/C++ programs [42]. Such HLS tools as Mentor Graphics' Catapult [26] translate these highly-abstract descriptions into the RTL code suitable for either ASIC or FPGA logic synthesis. Likewise, FPGA vendors provide support for HLS-based flow directly within their EDA suites, like for instance Xilinx' Vivado HLS. High-level models may significantly improve the productivity of the design process. They are especially useful in the context of FPGA-based applications, allowing quick implementations and run-time reconfigurations. However, designers may lack control over the micro-architecture of resulting HDL models. Therefore, RTL models still remain at the core semicustom design flow, since they provide a reasonable trade-off between the development efforts and the level of details that designers would like to specify manually.

The quality of the resulting implementation is usually computed attending to whether it meets the design requirements in terms of performance, power, and area (PPA). Performance estimations are usually tightly related to the results of a timing analysis, which determines the critical paths through the circuit along with the maximum operating frequency. The analysis of power consumption relies on switching activity estimations obtained by simulating the implementation-level model. The area metric may refer to both physical area of silicon chip (standard cells) and the utilization rate of FPGA resources. When dependability is an expected design feature, common PPA design goals should be complemented with dependability metrics, thus leading to the PPAD notation. Existing commercial tools provide no direct support for estimating dependability attributes [52]. Therefore, custom dependability assessment processes should be integrated

into the generic design flow to provide support for dependability-driven design strategies.

2.1.2 SRAM-based FPGA as target implementation technology

SRAM-based FPGA is a popular implementation technology, featuring quick development process and run-time reconfiguration among other benefits. An FPGA chip comprises two layers: the configurable FPGA fabric and the configuration memory. The configurable fabric is a two-dimensional grid of configurable logic blocks (CLB), digital signal processing blocks (DSP), memory blocks (BRAM), and IO transceivers, interconnected by means of a configurable routing network. The functionality of these components and their interconnection is determined by the content of the underlying configuration memory (CM). The CM content is loaded on FPGA start-up from the bitstream file stored in an external non-volatile memory. Later, the CM content can be modified at run-time to alter the functionality of some modules or completely replace them; this is achieved by loading a full or partial bitstream.

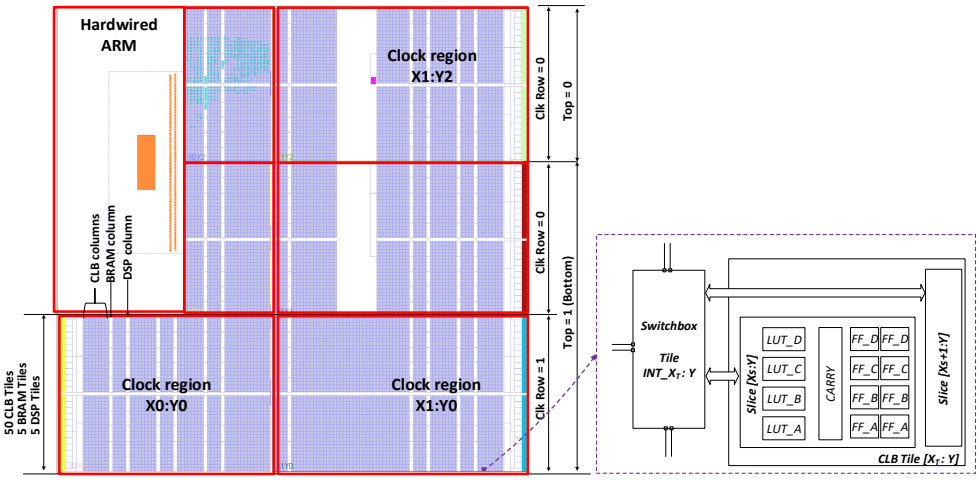


Figure 2.2: Coarse-grained PL architecture, illustrating the relation between BEL, Slice, Tile, clock region, clock row, Top/Bottom for the case of Xilinx 7-series (Zynq SoC) device

Following the Xilinx's terminology regarding the design flow, the sequential and combinational logic inferred at logic synthesis is mapped onto technology-specific logic *cells*. At device level these cells are placed into the *basic elements of logic* (BEL), which are the smallest components of FPGA fabric. Each BEL supports the placement of some class of logic cells: Flip-Flops, LUTs, MUXes, Carry chains

(being all of them part of a CLB,) DSPs, RAM blocks, etc. BEL can be seen as placement of cell. A group of related BELs is referred to as a *Slice* (site). Slices are arranged on the FPGA layout as a two-dimensional grid, and can be located by (X_S, Y_S) coordinates, counted from the bottom-left corner of the device layout. In Xilinx 7-series devices each type of slice has its own independent grid coordinates, except Slice_L and Slice_M, which share the same space of grid coordinates.

A group of slices form a *Tile*, which are arranged into columns on the device layout. Each column of tiles corresponds to the same type of FPGA resource (CLB, DSP and BRAM). Tiles can be located by (X_T, Y_T) coordinates, which are shared among tiles of all types, except switch box tiles, whose coordinates coincide with the tile of linked resources (CLB, DSP and BRAM). A two-dimensional array of tiles forms a *clock region*, which is crossed in the middle by the clock lane. In 7-series devices, for instance, the clock region has a height of 50 CLB tiles, or 5 DSP blocks, or 5 RAMB36 tiles. A horizontally aligned group of clock regions forms a clock row. Finally, FPGA devices are divided into Top and Bottom parts with one or more clock rows in each part, as depicted in Fig.2.2

The physical placement of design modules can be constrained by defining a rectangular area on the FPGA layout (X_T, Y_T coordinates of bottom-left and top-right Tiles), referred by Xilinx as *Pblock*. Each Pblock should include enough Tiles of each type (CLB, DSP and BRAM) as to allow the placement and routing of all logic cells within the associated design module.

The sequential logic in Xilinx 7-Series FPGAs is inferred on CLB Flip-Flops (eight FFs per CLB slice), memory blocks (BRAM), and distributed memory (LUTs of type-M). Combinational logic in Xilinx FPGAs is implemented by means of LUTs, MUXes, carry chains, and DSPs. LUT cells implement an arbitrary function of up to six variables; larger functions are implemented by combining these cells by means of slice multiplexers and carry chains. Each CLB Tile comprises two CLB slices. Each CLB slice includes four LUTs (labelled as A, B, C, D bottom-to-top). CLB slices pertain to one of two types: type L (logic) or M (Memory). The distinguishing features of the latter (M) is that it allows the implementation of distributed memories and shift registers on its LUT BELs. Further details on Xilinx's CLBs can be found in [175].

2.1.3 Technology-specific libraries

Semicustom and FPGA-based design flows rely on libraries of basic technology elements to implement the logic described in source HDL models. *Macrocell* is a term used to describe the basic technology elements in the semicustom design flow, being usually considered as the generalization of standard cells [48]. FPGA-based flow commonly uses by the term *logic primitive* to refer to the basic elements of logic (BELs), which is a native term in the architecture of the selected FPGA. Likewise FPGA vendors define macros, used to instantiate complex elements, which are expanded by EDA tools to their underlying primitives [184].

The libraries of technology primitives are distributed by vendors for their devices. They cover implementation and simulation aspects. From the implementation viewpoint these libraries provide the information required by EDA tools for technology-specific implementation processes. Primitives can be instantiated directly in the HDL code or inferred at logic synthesis. From the simulation perspective these primitives define the BEL's HDL models, which are used as the basis for functional and timing verification of intermediate implementation results.

This subsection provides an overview of the technologies and standards used to define the libraries of primitives (macrocells) from the simulation perspective.

2.1.3.1 Libraries for the functional and timing simulation

The models exported by EDA suites for post-synthesis verification are usually defined in the basis of functional simulation libraries, e.g. Xilinx's *Unisim*. These primitives must accurately reflect the functional behaviour of modelled BELs. At the same time, since the post-synthesis model may include thousands or millions of such primitives, they should be very efficient in simulation. The Verilog HDL provides a set of built-in generic logic primitives for gate-level modelling and supports user-defined primitives (UDPs). UDPs can be sequential (both edge-sensitive and level-sensitive) or combinational. UDPs models should comply with a set of requirements [36] for the definition of their interface and functionality. The functionality of both sequential and combinational UDPs can be modelled by truth tables, defining the primitive's output as a function of its inputs and internal state. The main advantage of using UDPs for component modelling is that simulators are able to apply efficient compile-time and run-time optimizations, thus improving the overall simulation performance. Further details on UDP rules and syntax can be found in [109].

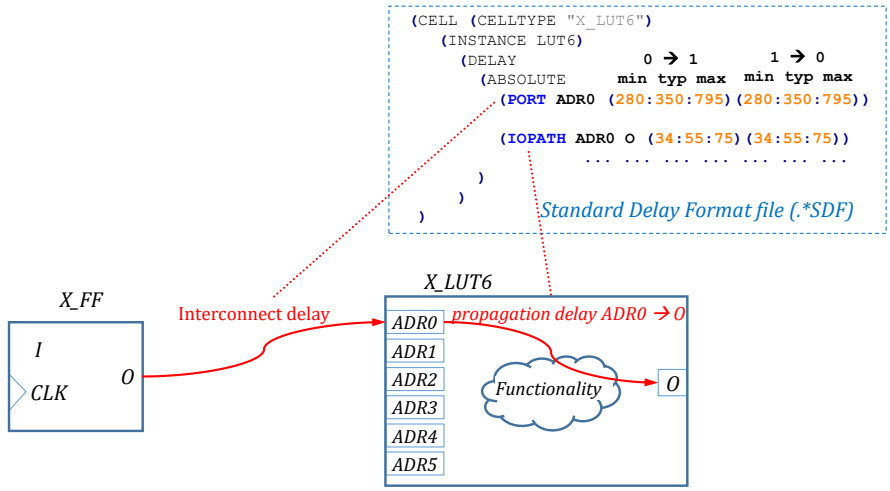


Figure 2.3: Annotation of interconnect and propagation delays from standard delay format (SDF) file

The post-place-route verification relies on timing simulation libraries, e.g. Xilinx's *SimPrim*. These primitives must accurately reflect both the functional and timing behaviour of modelled BELs. The models, exported by EDA tools, are annotated from the timing properties of the resulting circuit, specified using Standard Delay Format (SDF) files. SDF has been established by the Open Verilog International organization, as a tool- and language-independent format for representation and interpretation of timing data at any stage of the electronic design process [75]. Currently it is standardized as IEEE-1497.

Each macrocell in the netlist SDF file specifies a set of timing properties and timing checks supported by that macrocell. Two commonly modelled timing properties are the interconnect and propagation delays. The interconnect delay refers to the net connected to a given input port of a macrocell. It can be modelled on a pin-to-pin basis between two macrocells (Verilog), or by delaying the signal directly on the input port (VHDL-VITAL). In the first case the delay value is annotated from the *INTERCONNECT* tags of SDF file, in the latter case delay is annotated from the *PORT* tag. The propagation delays refer to the different paths that can be activated through the macrocell. In the SDF file path delays are specified by the *IOPATH* property followed by the names of input and output ports. As it is exemplified in Fig.2.3, all SDF delays are specified separately for different transitions: '0→1', '1→0', '0→Z', 'Z→0', '1→Z', 'Z→1'. At the same time, the minimum, typical, and maximum values are specified for each transition,

which correspond to the best-case, expected, and worst-case operating conditions of the circuit respectively.

Verilog natively supports the modelling of delays in simulation primitives through *specify* blocks, as well as back-annotation of timing properties from SDF files by means of the *$sdf_annotate* system task. Specify blocks define a (i) set of timing parameters (*specparam*), (ii) a set of paths, which model the interconnect and propagation delays on a pin-to-pin basis, and (iii) system timing checks. *Specparam* are special Verilog parameters, accessible only within the *specify block* (which declares the timing properties of Verilog macrocells), and they are used to store and calculate the actual (or default) delay values. Path declarations can use these parameters or constant literals instead. As detailed in [168], paths specifications may be continuous, edge-sensitive, and state-dependent. Timing parameters are annotated from a SDF file by looking-up the paths definitions with matching source/destination pins and path activation events. Timing checks monitor different timing properties that should be satisfied for a correct macrocell operation, such as setup and hold times annotated from *SETUPHOLD* tag of SDF file.

2.1.3.2 The VITAL standard

In its early years, there was no uniform and efficient method for handling timing in VHDL, which resulted in a lack of ASIC libraries for modelling and implementing digital systems. The VHDL Initiative Towards ASIC Libraries (VITAL) standard [76] was the result of an agreement among ASIC vendors, EDA tool vendors, and ASIC designers about the requirements (timing accuracy, model maintainability, and simulation performance) for the effective modelling of ASIC primitives, or macrocells, in VHDL.

The VITAL specification contains four main elements: i) the *Model Development Specification document* defines how to specify ASIC libraries in VITAL-compliant VHDL to be used in simulators; ii) the *Vital_Timing package* provides a standard set of procedures for checking timing constraints defined in a SDF file; iii) the *Vital_Primitives package* models all gate-level primitives already used by simulation tools vendors, so they could be optimised for a faster simulation of VHDL; and iv) the *VITAL SDF map* which maps SDF files to VHDL generic values.

The basic architecture of a VITAL-compliant macrocell is depicted in Fig. 2.4 and Listing 2.1 specifies an inverter that follows this architecture.

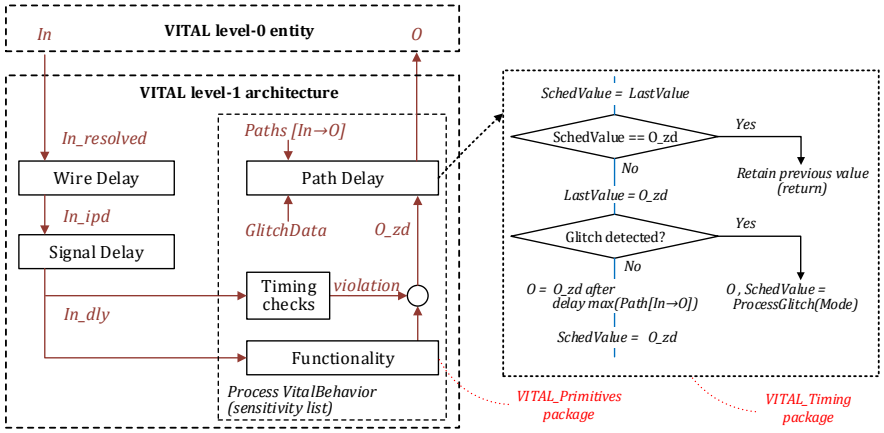


Figure 2.4: VITAL-compliant macro-cell model

VITAL defines two levels of support. *VITAL level 0* requires the definition of a level 0 attribute (line 21 of Listing 2.1), using ports of type *std_ulogic* and *std_logic_vector* with no underscores in the port names (lines 18–19), and special naming convention for timing generics (lines 8–11, with ports names prefixed as *tpid_*—interconnect path delay that represents the delay between components—and *tpd_*—propagation delay that represents the pin-to-pin delay within a component). Compliance with VITAL level 0 provides SDF back annotation and negative timing constraints. *VITAL level 1* requires the definition of a level 1 attribute (line 26), no use of shared variables, use of those operators defined in the *Standard* and *std_logic_1164* packages (lines 1–3 ensure that only those packages and the VITAL packages are used), and all outputs must be driven by a *VitalPathDelay* or a *Vital* primitive (lines 53–63 make use of a *VitalPathDelay01* primitive to drive the *YNeg* output—line 54). Compliance with VITAL level 1, as the inverter described in Listing 2.1, provides accelerated simulation of primitives and tables.

Any VITAL-compliant macrocell must include a *Wire Delay* block, a *Signal Delay* block, and a *VITALBehavior process*, as depicted in Fig. 2.4.

Listing 2.1: VITAL-compliant inverter gate (std04.vhd) [137]

```
1  LIBRARY IEEE; USE IEEE.std_logic_1164.ALL;
2                USE IEEE.VITAL_timing.ALL;
3                USE IEEE.VITAL_primitives.ALL;
4
5  -- ENTITY DECLARATION
6  ENTITY std04 IS
7    GENERIC (
8      -- tipd delays: interconnect path delays
9      tipd_A       : VitalDelayType01 := VitalZeroDelay01;
10     -- tpd delays
11     tpd_A_YNeg   : VitalDelayType01 := UnitDelay01;
12     -- generic control parameters
13     MsgOn        : BOOLEAN          := DefaultMsgOn;
14     XOn          : BOOLEAN          := DefaultXOn;
15     InstancePath : STRING           := DefaultInstancePath
16   );
17   PORT (
18     A            : IN  std_ulogic   := 'U';
19     YNeg         : OUT std_ulogic   := 'U'
20   );
21   ATTRIBUTE VITAL_LEVEL0 of std04 : ENTITY IS TRUE;
22 END std04;
23
24 -- ARCHITECTURE DECLARATION
25 ARCHITECTURE vhdl_behavioral of std04 IS
26   ATTRIBUTE VITAL_LEVEL1 of vhdl_behavioral : ARCHITECTURE IS TRUE;
27   SIGNAL    A_ipd : std_ulogic := 'U';
28
29 BEGIN
30   -- Wire Delays
31   WireDelay : BLOCK
32   BEGIN
33     w_1: VitalWireDelay (A_ipd, A, tipd_A);
34   END BLOCK;
35
36   -- No Signal Delay block
37
38   -- VITALBehavior Process
39   VITALBehavior : PROCESS(A_ipd)
40
41     -- Functionality Results Variables
42     VARIABLE YNeg_zd      : std_ulogic          := 'U';
43     -- Output Glitch Detection Variables
44     VARIABLE Y_GlitchData : VitalGlitchDataType;
45
46   BEGIN
47     -- No Timing Checks section
48
49     -- Functionality Section
50     YNeg_zd := VitalINV (A_ipd);
51
52     -- Path Delay Section
53     VitalPathDelay01 (
54       OutSignal     => YNeg,
55       OutSignalName => "YNeg",
56       OutTemp       => YNeg_zd,
57       XOn           => XOn,
58       MsgOn         => MsgOn,
59       Paths         => (
60         0 => (InputChangeTime => A_ipd'LAST_EVENT,
61               PathDelay       => tpd_A_Yneg,
62               PathCondition   => TRUE)),
63       GlitchData    => Y_GlitchData );
64
65   END PROCESS;
66
67 END vhdl_behavioral;
```

Interconnect delays represent the time it takes a signal to traverse the circuit from one component to another, which depends on various factors, like the length of the wire, its resistance, and fan-out, among others. The *Wire Delay* block (lines 30–34 in Listing 2.1) is in charge of delaying incoming signals by the time specified in the associated timing generic (*tpid* using the *VitalWireDelay* routine (line 37 in Listing 2.1). This routine can only be called once per input port, which cannot be used anywhere else in the model afterwards, and its output must be an internal signal (*A_ipd* in the example).

Components may present negative timing constraints (either setup or hold times, not both) only if they present some kind of internal delay. The *Signal Delay* block takes charge of delaying the internal signals of the component (suffix *_ipd*), if needed, in a similar fashion to the *Wire Delay* block, but using the *VitalSignalDelay* routing instead. The sample model has no negative timing constraints, so this block is not defined (line 36).

The actual functionality of the component is defined within a process labelled as *VITALBehaviour* (lines 38–63 in Listing 2.1). Any signal read within the process (*A_ipd* is read in line 46) must be included in its sensitivity list (line 39), so the process will be triggered whenever the value of any of these signals changes. First of all, the process may perform timing constraint checks for possible violations if the *TimingChecksOn* generic parameter is active. Those checks make use of predefine routines, like *VitalSetupHoldCheck*, from the *VITAL_Timings* package. No timing checks are performed in the sample model (line 47). After that, the functionality section computes the actual logic function of the component without any delay (*YNeg_zd*). Lines 49-50 of Listing 2.1 define the behaviour of the inverter by means of the predefined *VitalINV* function to comply with VITAL level 1. Finally, the output values computed are delayed after applying the appropriate delays using *VitalPathDelay* procedures (lines 52-63). These procedures enable different optimisations of the simulation like, for instance, checking whether the output has changed with respect to the previously scheduled value to prevent further processing.

Beyond meeting their functional and timing constraints, many hardware designs remain useless if they are not able to comply with their functional requirements while providing acceptable levels of dependability. Next section introduces the process of dependability assessment that deals with quantification and analysis of dependability attributes.

2.2 Dependability assessment

One of the primary goals of a dependability-aware design flow is to ensure that the designed system meets the dependability requirements of the target application domain. Dependability assessment, in a broad sense, deals with the quantification of dependability attributes that, following the terminology in [13] and [191], are defined as follows.

- *Reliability* is the probability that the system will operate correctly throughout the interval $[t_0, t]$, given that system was operating correctly at time t_0.
- *Availability* is the probability that the system is operating correctly and is available to perform its functions at a given time instant t.
- *Safety* is the probability that a system will either perform its functions correctly or will discontinue its functions in a manner that does not disrupt the operation of other system or compromise the safety of any people associated with the system.
- *Integrity* is the measure of absence of improper system alterations;
- *Maintainability* is a measure of the ease with which a system can be repaired, once it has failed; in other words, the aptitude to undergo repairs and evolution.

Above attributes are defined by [13] as primary ones. By specializing them to a specific class of faults, secondary dependability attributes can be defined. For instance, *robustness* is defined as dependability with respect to externals faults.

In practice dependability assessment often focuses on the estimation of attributes based on *reliability function* $R(t)$. It expresses the probability R that the system will survive up to a specified time t [101]. The arithmetic mean value of the reliability function is known as the *mean time to failure* ($MTTF$) for non-repairable systems, or *mean time between failures* ($MTBF$) for repairable ones, and can be computed by Equation 2.1.

$$MTTF = \int_0^\infty R(t) \cdot dt \tag{2.1}$$

The *failure probability function* $F(t)$ is the metric complementary to $R(t)$, expressing the probability that an element will fail before a certain time instant. The *failure density* $f(t)$ is the mathematical derivative of failure probability, expressing how the latter changes over time. On the basis of these functions the *failure rate* metric is defined as $\lambda(t) = \frac{f(t)}{R(t)}$. It quantifies the probability that a

module that has not failed up to time t will fail up to time $t + dt$ (within the following small period dt). As detailed in [101], failure rate can be estimated in practice from the observations of the failure mode of a large number of similar devices, being λ interpreted as the number of elements (devices) which fail on average in a time unit.

The failure rate plotted with respect to time is commonly following a bathtub curve. It comprises three intervals: early failures, random failures, and wear-out failures. The former describes the rate of failures resulting from manufacturing defects. The failure rate typically drops during this time interval. The middle interval represents random failures, which are usually unforeseeable due to the superposition of the wide range of independent factors. The failure rate during this interval becomes stable (constant). Finally, the latter interval represents the failure rate resulting from circuit aging effects, characterized by an increasing failure rate. As detailed in [101], for constant failure rate $\lambda(t) = \lambda$ (middle interval), the $MTTF$ can be quantified as follows: $MTTF = \frac{1}{\lambda}$.

$MTTF$ is one of the most commonly used metrics, allowing to compare different design alternatives from the reliability perspective. At the same time, MTTF of simplex designs is known to be greater than MTTF of redundant (K-out-of-N) systems [47]. This is explained by the reliability function of redundant systems, which is very high at the beginning, as replicas tolerate faults, but quickly falls down at the end of the lifetime, as the redundancy is exhausted. Nevertheless, at the beginning of the life time the reliability of redundant (e.g. TMR) systems is much higher than that of simplex versions. Therefore, MTTF may be misleading when comparing simplex designs with their redundant versions. Instead, the reliability of critical replicated systems can be compared on a fixed time interval called *mission time*,as opposed to MTTF computed for $t \to \infty$. The *mission time* metric is defined as the time in which the reliability of the system remains above a given threshold. Assuming an exponential reliability function and a constant failure rate, the mission time can be computed by solving the Equation 2.2 for a given reliability threshold R_T. This metric allows to take into account the reliability benefits of redundancy for critical systems.

$$R_T = e^{-\lambda \times t} \tag{2.2}$$

The estimation of dependability attributes is related to the *fault forecasting* process, as depicted in Fig.2.5. Fault forecasting aims at estimating the present number, the future incidence, and the likely consequences of faults [13]. Fault forecasting is conducted by evaluating the systems behaviour with respect to fault occurrence and activation. This process has qualitative and quantitative

aspects. The qualitative forecasting aims at identifying, classifying and ranking failure modes, or event combinations, leading to systems failure. The quantitative forecasting evaluates in terms of probabilities the extent to which some of the attributes are satisfied. Quantitative forecasting comprises modelling and testing. As explained in [13] these approaches are complementary since modelling requires the data on the basic modelled processes, that may be obtained by testing.

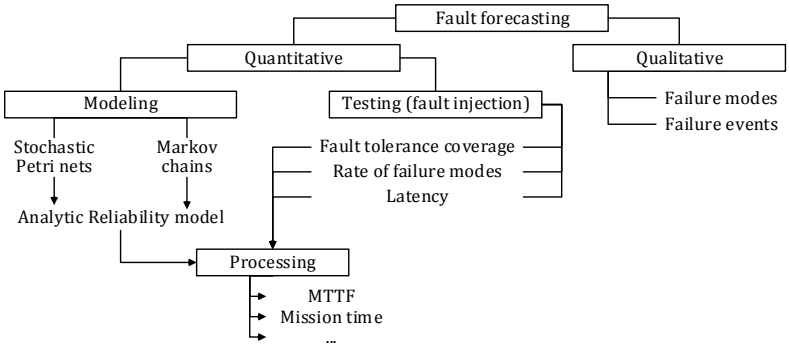


Figure 2.5: Estimation of reliability attributes in the context of fault forecasting

Modelling starts with the definition of the model of the system from the elementary stochastic processes describing the behaviour of the component and their interactions. Discrete Markov chains (DTMCs) and Petri nets [148] are two commonly used methods to define the model of a system. The defined model subsequently is processed to obtain the expressions and dependability measures. The testing process refines the dependability metrics obtained at model processing by quantifying the functional ability of HW designs to tolerate faults. First of all, HW implementations may present some intrinsic robustness, since not every fault leads to an error, and not every error is propagated to a failure. Thus, testing should measure the percentage of faults leading to a failure (functional failure rate). Furthermore, as pointed in [13], when evaluating fault-tolerant systems, the computation of dependability measures from the model should take into account the measure of efficiency of fault handling mechanisms, known as *fault tolerance coverage*.

For instance, the work in [64] defined the reliability models of FPGA-based system in presence of N-modular redundancy, in order to compare alternative implementations from the reliability viewpoint. The defined models take into account the utilization of different types of FPGA resources and their upset rate, reported by Xilinx in [179]. Configuration memory (CM) and changeable memory (BRAM

and LUTRAM) are two main types of resources considered of a higher priority for reliability estimations of FPGA designs [72].

The design-specific set of CM cells determining the actual circuitry within the FPGA fabric is known as *essential bits*. In such a way, the failure rate of FPGA design with respect to CM upsets is commonly computed by multiplying the amount of essential bits by the CM upset rate. This may lead, however, to significantly overestimate the resulting failure rate. Indeed, as it is pointed in [179], any design implemented in FPGAs has unused and non-critical bits, which do not lead to a failure when upset. Likewise [97] proposes to distinguish the CM upsets leading to a failure from the rest of upsets. The subset of essential bits that in case of upset lead a system to a failure are referred to as *critical bits*. When the bit-accurate mapping between the FPGA logic/routing resources and the CM cells is known (which is very rare in case of modern FPGA devices), the essential and critical bits can be located by means of static analysis of the routed design, as it is proposed in [154]. Otherwise, critical bits can be identified by means of fault injection testing.

Hence, for more realistic estimations a percentage of critical bits should be taken into account, commonly referred to as *device vulnerability factor* (DVF), which according to [179] rarely exceeds 10%. Considering the DVF in reliability models allows to refine the derived reliability estimates. In such a way, the failure rate taking into account the DVF can be quantified as follows:

$$\lambda_{design} = \lambda_F \times K \times N_{EB} \times DVF, \tag{2.3}$$

where:

λ_F is a constant failure rate at sea-level taken from device reliability reports provided by FPGA manufacturers. For instance, it can be found in [179], expressed in FIT units per megabit of configuration memory, for the Xilinx 7-series FPGAs. One FIT unit corresponds to one upset per 10^9 device hours.

K is a derating factor scaling the upset rate according to the altitude. It is based on Rosetta test results [72] [136], being 1.0 at sea-level, 21.3 at high-terrestrial altitudes, and 327.8 for space applications.

N_{EB} is the amount of essential bits (in megabits) in a given implementation.

DVF is the device vulnerability factor (percentage of critical bits within the set of essential bits), obtained by fault injection testing.

It is worth noting, that system failures may have different severity. The design of safety-critical systems must estimate the risk of dangerous failures, i.e. those which may lead to catastrophic consequences. Depending on its functional reliability, the safety-instrumented system may qualify for one of Safety Integrity Level (SIL). The international safety standard IEC-61508 defines four safety levels, abbreviated SIL-1 to SIL-4, being the former attributed to a highest malfunction risk (lowest reliability), and the latter attributed to a lowest risk of malfunction (highest reliability) [169]. In the safety of automotive systems, regulated by standard ISO-26262, these levels are mapped into Automotive Safety Integrity Levels (ASIL), as listed in Table 2.1. To qualify for a given SIL, a hardware device must meet a maximum probability of dangerous failure P_{DF}, listed in Table 2.1, or similarly ensure a given risk reduction factor $RFF = {}^{1}/_{P_{DF}}$. These values are established separately for continuously used systems (e.g. motor car brakes), and for systems used on demand (e.g. car air bag). The former considers the probability of failure per hour, while the latter estimates the probability of failure on demand [151].

Table 2.1: Safety Integrity Levels (SIL) for continuously used systems and for systems used on demand

Safety integrity level (SIL) (IEC615087)	ASIL (ISO26262)	Continuously used system (probability of dangerous failure per hour)	Low-demand system (probability of dangerous failure on demand)
1	A	$10^{-9} - 10^{-8}$	$10^{-5} - 10^{-4}$
2	B/C	$10^{-8} - 10^{-7}$	$10^{-4} - 10^{-3}$
3	C/D	$10^{-7} - 10^{-6}$	$10^{-3} - 10^{-2}$
4	—	$10^{-6} - 10^{-5}$	$10^{-2} - 10^{-1}$

Fault injection is considered one of the most valuable testing techniques in dependability assessment. First of all, it is not always feasible (or very costly) to observe the system in the field to get statistical data. Second, unlike other testing techniques, fault injection can be employed at different phases of the design flow, starting from the high level model and up to the final prototype. Fault injection is widely recognized by the industry and explicitly mentioned in safety standards. Particularly, the safety standard in automotive domain ISO-26262 recommends the usage of fault injection to verify the designed system in presence of faults throughout the whole design flow, starting from early design stages. As discussed in [133], fault injection may efficiently complement traditional dependability analysis processes in automotive development processes (like FMEA [135]). Nevertheless, as it will be detailed in Chapter 3, its integration into the design flow still faces important challenges related to the representativeness of involved fault models, accuracy of injection procedures, and experimentation performance.

When several alternatives are available for selection, the process of assessing and comparing their dependability metrics for choosing the most suitable one is known as dependability benchmarking. Next section introduces this concept.

2.3 Dependability benchmarking

The semicustom design flow supplies designers with a wide assortment of alternative IP cores, EDA tools, and implementation technologies. The selection of one or another alternative may significantly impact the PPAD features of resulting implementations, thus being crucial for meeting the design goals. In the context of HW designs, *dependability benchmarking* can be understood as an experimental approach aiming at comparing and selecting alternative solutions attending to their PPAD features.

As detailed in [86], to be useful in practice, benchmarks should rely on measures that are representative for a given application domain. At the same time, to provide justifiable results, the involved evaluation and analysis procedures should be reproducible. These requirements are well addressed by numerous conventional (performance) benchmarks. On the one hand, they develop representative workloads for a given application domain, like for instance, *LINPACK* targeting high-performance computers (HPC), or any of the EEMBC (Embedded Microprocessor Benchmark Consortium) benchmarks for embedded systems. On the other hand, they obtain a relatively small set of widely recognized performance measures like, for instance, the number of floating-point operations executed per second (*FLOPS*).

Addressing the aforementioned requirements in the context of dependability is not straightforward. On the one hand, fault injection is a widely-recognized instrument for evaluating the dependability attributes of alternative solutions. On the other hand, there is no standardized uniform way for carrying-out such experiments. Existing benchmarking proposals rely on diverse fault injection approaches (software-based, hardware-based, model-based, etc.), apply diverse faultloads and estimate different dependability attributes that they consider representative for their particular application scenario. In addition to that, benchmarking EDA tools and implementation technologies requires to consider the implementation-level models. However, existing benchmarking solutions usually relying on custom fault injection tools, rarely handle implementation-level HDL models, or they do so in a highly intrusive way which degrades the credibility of results.

Finally, the comparison and selection processes are not trivial, since the relevant dependability attributes may be application-dependent, and often several of them

must be considered at the same time along with the rest of (PPA) attributes. Multiple-criteria decision making (MCDM) methods provide a way for structuring complex problems and considering multiple, and usually conflicting, criteria to make more informed and better decisions [79]. As it is the case when implementing designs on reconfigurable devices, there is usually no unique optimal solution for problems involving multiple criteria, so it is necessary to take into account the preferences of the decision maker (usually by weighting the relative importance of each criterion) to differentiate between available solutions. Different methods, each one with its own mathematical foundations, have been developed along the years, such as the Weight Sum Model (WSM) and the Weight Power Model (WPM) [160], the Analytic Hierarchy Process (AHP) [141], the VIseKriterijumska Optimizacija I Kompromisno Resenje (VIKOR) [125], or the Technique for Order of Preference by Similarity to Ideal Solution (TOPSIS) [189]. For instance, as [59] has shown, the uncertainty of the analysis process can be effectively addressed by weighting all the different PPAD metrics in a hierarchical way (according to the requirements of application scenarios) and, subsequently, by ranking alternatives on the basis of a score-based model (such as WSM/WPM, for instance).

A challenge of paramount importance when assessing the dependability of a hardware design or when benchmarking different design alternatives from a dependability viewpoint, is the number of fault locations and fault models to consider during the evaluation process, which may lead to unaffordable number of experiments to carry out. The problem augments with the number of parametrization options in the IP cores or EDA tools under use. The dependability-aware design space exploration introduced in next section addresses this problem.

2.4 Dependability-aware design space exploration

The development of complex digital systems poses a great design optimization problem, especially for dependability-related applications. Engineered solutions must not only meet the required functionality, but they must also meet a number of (sometimes conflicting) implementation goals, such as minimizing power consumption and occupied area, while maximizing the attainable clock frequency and exhibited robustness [170].

Due to its critical role in meeting the implementation goals, FPGA manufacturers and third party companies providing EDA toolkits include a wide range of different optimization options suitable to different kind of devices, architectures, and scenarios. Nevertheless, far from alleviating the task of designers, the myriad of available options makes it very difficult to know the precise contribution of

each option to a particular goal. Some options may have a greater impact in the implementation goals than others, some of them may have opposite effects, and most of them are never used because it is not clear enough what could rightly be expect from them [89]. The side effect is that improperly tuning EDA parameters negatively impacts the quality of resulting design implementations.

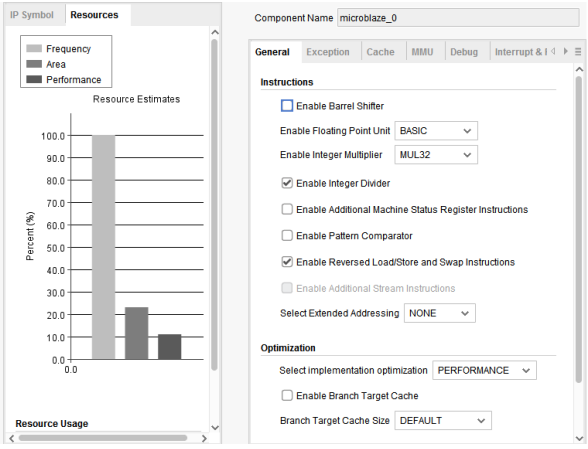


Figure 2.6: Impact of Microblaze IP parameters on expected area and performance (Vivado 2018.3)

Likewise, complex IP cores often supply designers with dozens of parameters, allowing their customization for a given application scenario. Some IP cores, distributed with off-the-shelf EDA suites, allow designers to get an idea about how setting each parameter to one level or another may impact PPA results. For instance, the customization guide of Xilinxs' Microblaze IP, illustrated in Fig. 2.6 allows designers to tune the architectural (enabling FPU, MMU, caches, etc.) and functional (exceptions, stack/memory protection, etc.) parameters, and to choose from predefined optimization presets. For each selected setting of IP parameters, it provides a relative estimation ($0\% \rightarrow 100\%$) of expected frequency, area, and performance. However, estimations in absolute units are not available at this abstraction level, since PPA results are strongly conditioned by synthesis/placement/routing optimizations. For instance, as Fig.2.7 shows, enabling just one 'resource sharing' optimization in Mentor Graphics' Precision Synthesis, allows to reduce up to three times the utilization of FPGA resources (Area) for implementing arithmetic operations. It is not surprising that the expected dependability features are completely omitted by customization guides, since the impact of all these parameters on dependability is a priori unknown.

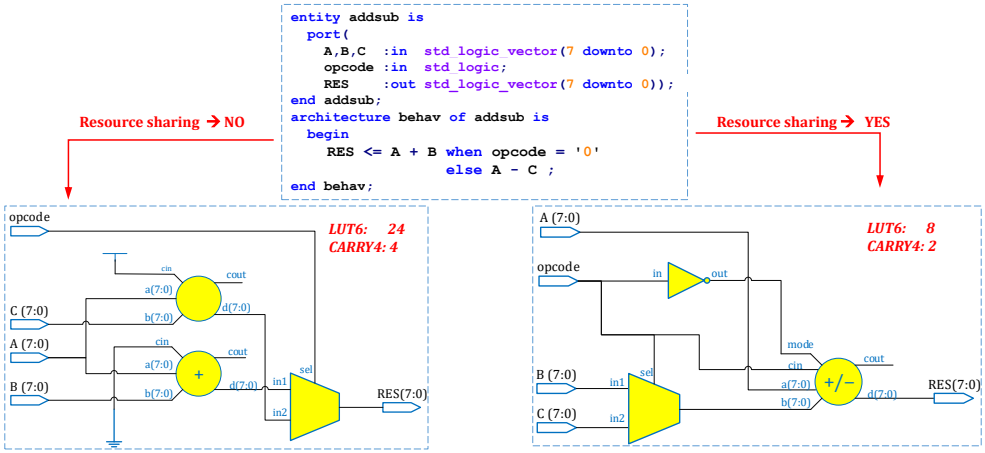


Figure 2.7: Impact of resource sharing optimization (Mentor Graphics' Precision Synthesis) on resulting area (Virtex-6 FPGA)

Determining the best configuration of available EDA/IP parameters for a given implementation goal would require exploring the whole design space (all possible combinations of parameters at different levels). FPGA manufacturers provide different tools for design space exploration, like Xilinx's SmartXplorer [178] or Intel Design Space Explorer II [3], which perform several different implementations (changing the synthesis options) of the same design looking for the configuration that best meets the required PPA goals. Nevertheless, even when many-core machines or computer clusters can be used to perform the exploration in parallel, the time required to sweep the whole design space is prohibitive. For instance, considering that the 34 different synthesis options from Xilinx's XST synthesizer could be set at just two possible levels, and the implementation and evaluation of the design takes just 1 minute (very optimistic), exploring the whole design space (2^{34} configurations) will take roughly 32000 years running on a single-core machine. Some modern EDA suites like Xilinx Vivado present an even bigger design space due to a considerable number of parameters which should be treated at more than two levels (e.g. synthesis, placement, routing strategies).

It is worth noting, however, that industrial designs are often characterized by implementation constraints, that could limit the free adjustment of some synthesis, placement and routing parameters offered by EDA tools.

Several works have dealt with the problem of design space exploration from different perspectives. For instance, a given architectural configuration for a multiprocessor may required a couple of weeks to be simulated, so statistical simulation [65]

or predictive modeling [61] are some of the proposed approaches to reduce that design space. Focusing on reconfigurable devices, evolutionary approaches were proposed in [132] to find a good solution in High-Level Synthesis with conflicting design objectives [147] focused on the parametrization of soft-core processors through a greedy search method. A calibration free algorithm for automatically optimizing design parameters was proposed in [95]. None of these works specifically focused on the parametrization of the synthesis, placement, and routing processes, but on the architectural features of the designs to be implemented onto the reconfigurable device.

In the particular context of FPGAs, an approach based on machine-learning autotuning was presented in [105] to sample the parameter space and thus reduce the time devoted to the configuration search process. Nevertheless, although this approach, and those provided by FPGA manufacturers, may find a suitable configuration for the requested goals, the particular contribution of each selected option and their interactions are not accounted. Accordingly, designers are at a loss when deciding how to configure the synthesis tool for each given design.

A very preliminary first step towards achieving this goal was taken at [110], which estimated the impact of different Xilinx's ISE optimization options on the power consumption of different security algorithms. However, that study considered just 4 different options, and focused on just one primary goal (power consumption). In addition to that, the contribution of each particular parameter to that goal was not determined, just the difference between configurations.

In fact, most of existing DSE proposals are tailored to either increase the maximum clock frequency of the final system or decrease the number of resources required for its implementation, usually at the expense of the rest of goals. This is especially true in the case of robustness, which traditionally falls behind industrial traditional needs in terms of costs and computing power [52].

Existing approaches targeting the enhancement of robustness of HW implementations present two main limitations. First, those approaches acting at the design entry [140] [27] [4] (relying on high-level model of the system), do not, or only partially, consider the side effects of improving the dependability in the rest of PPA goals, which limits their usefulness in practice. Second, those approaches optimizing the synthesis [51], mapping [130], placement [33], and routing [73], are extremely difficult to apply, as they usually involve the modification of processes embedded within off-the-shelf tools that can be very rarely accessed.

2.5 Conclusions

The development of critical systems must meet not only the required functionality, performance, power, and cost budget, but also the strict dependability requirements of the target application domain. The conventional semicustom design flow should be thus complemented by several dependability-driven processes. First of all, dependability assessment must be carried out at each step of the design flow, to ensure the efficiency of any deployed fault tolerance mechanism and to evaluate resulting dependability attributes against the requirements of the target application. Second, dependability benchmarking should guide the selection of most suitable IP cores, EDA tools, and implementation technologies that best satisfy PPAD goals. Third, dependability-aware design space exploration should be carried out to optimally configure selected EDA tools and IP cores to obtain the best possible PPAD results.

All three dependability-driven processes rely on fault injection to evaluate the dependability features of designed systems. To seamlessly integrate these processes into the design flow, the fault injection methodology must meet a set of requirements. First of all, to support the dependability assessment (as the safety-aware design standards require), it should cover all the stages of the design flow from high-level HDL models to final prototypes, and ensure the accuracy of devised analysis at each of these levels. Second, to support benchmarking, it should be generic enough to handle any HDL design and implementation technology. Finally, it should provide a very high experimentation performance, in order to prevent the approach from becoming a bottleneck for the deployment of these processes.

Particularly, the fault injection effort is one of the major challenges of dependability-aware DSE, requiring the evaluation of a large number of alternatives in the design space. This could be a reason why (as the analysis of the background shows), no solution has been proposed so far for optimally tuning EDA/IP parameters to simultaneously improve both PPA and dependability features of resulting implementations.

The following chapters of this thesis address the aforementioned dependability-driven processes throughout the semicustom design flow. Chapter 3 studies the capabilities and limitations of existing fault injection solutions with respect to the dependability assessment at different levels of the design flow. Chapters 4 and 5 propose simulation-based and FPGA-based fault injection approaches that address the limitations of existing fault injection approaches, enabling accurate dependability assessment at each stage of the design flow. Chapter 6 proposes a set of techniques to improve the performance of fault injection experiments.

After that, time-efficient DSE strategies for optimal EDA/IP tuning are defined in chapter 7. All the defined strategies are seamlessly integrated into the semicustom design flow by means of a new toolkit described in chapter 8. Finally, chapter 9 illustrates the usefulness and efficiency of these proposals through a case study of three embedded soft-core processors.

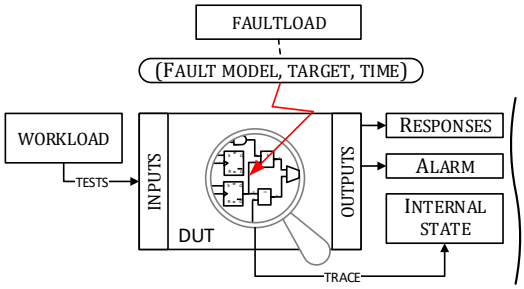
Chapter 3

Fault Injection for Dependability Assessment of HW Designs

This chapter studies existing advances and challenges of fault injection with respect to its integration into a dependability-aware semicustom design flow. Section 3.1 introduces the background (fault injection concepts, techniques and requirements) required to understand the rest of the chapter. Section 3.2 presents the commonly considered fault models at the logic level. Section 3.3 discusses the challenges existing today when relying on simulation-based fault injection (SBFI) for dependability assessment. It also analyses the capabilities of existing fault injection tools with respect to their integration into the semicustom design flow. Likewise, Section 3.4 discusses the techniques, tools, and challenges of FPGA-based fault injection. Section 3.5 presents the currently existing proposals for improving the performance of fault injection experiments. Finally, Section 3.6 summarizes the main conclusions of this chapter.

3.1 Introduction

Fault injection is a verification and dependability assessment methodology consisting in the analysis of system behaviour under the deliberate introduction of faults into that system. The verification dimension of the concept encompasses the qualitative analysis, referred to as fault removal. It aims at discovering the potential weak points (fault tolerance deficiencies) of the target system, and at determining the most appropriate means to design, develop, and deploy the necessary fault tolerance mechanisms [14]. The dependability assessment aspect is related to the quantitative analysis of the targets systems, being a part of a fault forecasting or robustness characterization process, as it has been discussed in Section 2.2.



(a) components of fault injection experiment

COMPARE TO REFERENCE (GOLDENRUN)

RESPONSES	ALARM	INTERNAL STATE	FAILURE MODE
MATCH	NO	MATCH	MASKED
MATCH	*	MISMATCH	LATENT
MISMATCH	YES	*	SIGNALED FAILURE
MISMATCH	NO	*	SILENT DATA CORRUPTION

(b) Failure modes for robustness characterization

Figure 3.1: Basic concepts of fault injection

A fault injection experiment within the semicustom design flow involves a set of generic components, depicted in Fig.3.1, namely:

- Targeted system (design under test, DUT), represented by its HDL model or its physical implementation (e.g. FPGA prototype);
- Workload, executed by the DUT, comprising a set of input test vectors and executable code (in case of processor designs);
- Faultload, specifying a set of faults to be injected into the DUT during the experimentation; each fault configuration is commonly described by three attributes: fault model (bit-flip, stuck-at, pulse, delay, etc), targeted design node, and injection time;
- Responses, characterizing the service delivered by the DUT in response to the input workload;
- Alarm, which can be set by the DUT to notify the higher-level system regarding internally detected errors;

– Trace of DUT's internal state (registers and memories), captured during the workload execution;

With respect to the popular *FARM* model, established to describe the fault injection experiments in [10], the aforementioned components can be mapped as: 'F' (*Fault set*) →faultload, 'A' (*Activation*) →workload, 'R' (*Readouts*) →tuple (Responses, Alarm, Internal state), 'M' (*Measures*) →Failure modes, latency and derived metrics.

Failure modes characterize the DUT behaviour in the presence of injected faults. Failure modes are determined by comparing the readouts obtained from each injection run to the readouts from a fault-free run (the so-called *GoldenRun*), as depicted in Fig.3.1. Particularly, when the responses and the DUT internal state match the GoldenRun, the failure mode is classified as *masked fault* (also referred to as silent fault). When the DUT responses are correct, but the internal state at the end of the injection run does not match the GoldenRun, the failure mode is reported as *latent error*. This means that even if the DUT was able to deliver a correct service on the considered workload, its internal state is corrupted by a fault, which may result in a future DUT failure. In case of incorrect DUT responses, the failure mode is classified as either a *signalled failure* when the alarm is raised, or as a *silent data corruption* (SDC) otherwise. The latter represents the most dangerous fault effect from a safety perspective. The rate (percentage) of each failure mode within the complete set of executed injection runs quantifies the relative robustness of the DUT and the coverage features of its fault tolerance mechanisms.

To obtain credible measurements by means of fault injection, several features must be taken into account in the design of fault injection environments [103]:

– *Representativeness* – the fault injector should properly reproduce those faults (fault models and distribution) that are representative of the real operational conditions.
– *Low intrusiveness* – the fault injector should not affect, or should minimize its effect on, the DUT behaviour in any way other than introducing a selected fault into the designated design node (fault target). An important challenge is to reach all relevant fault targets within the DUT while minimizing the DUT modifications and preventing the interference between the injector and the DUT. It must be noted that increasing the level of intrusiveness may degrade the credibility of observations and, thus, the usefulness of derived results and conclusions.

- *Observability* – the fault injector should be able to observe all relevant readouts (responses and DUT state) to properly characterize the DUT behaviour with respect to the injected faults.
- *Inspectability* – injection runs and related results (mainly reported traces) must be provided with enough level of detail to enable any ulterior audit.
- *Reproducibility* of experiments – repetition of the same experiment should provide statistically similar results. This is tightly related to the *repeatablity* feature of experiments, meaning that injection experiments must be repeated in case of need to verify the correctness of results. This requires a very fine control on applied injections from the perspective of both time and space.

Depending on the level of description of the DUT, the existing injection solutions can be classified as prototype-based and model-based. Prototype-based solutions target the physical prototype of the system either at hardware or software level, thus referring to *Hardware implemented fault injection* (HWIFI) and *Software implemented fault injection* (SWIFI), respectively.

HWIFI solutions rely on a specially designed test equipment to introduce physical faults. Some of them operate at the pin level, by forcing (the voltages) and monitoring the values on the DUT interface. Known pin-level HWIFI solutions are: AFIT [107], MESSALINE [10], and RIFLE [104] tools. Other HWIFI solutions rely on the generation of controlled electromagnetic interference (EMI) to induce the faults into the DUT installed between the conducting places [87] [38] [166]. Finally, some HWIFI solutions place the DUT into the vacuum chamber and introduce faults through heavy-ion radiation [88]. Two common limitations of HWIFI approaches is that they are applicable only at the final stages of design flow (once the prototype becomes available) and require expensive specialized equipment. Most of them feature low reproducibility and poor observability of experiments. Furthermore, they pose a high risk of damaging the device under test itself.

SWIFI solutions inject faults through the software layer by altering data and instructions in memory and registers. Some well-known SWIFI implementations are: MAFALDA [9], FERRARI [85], and XCEPTION [35] tools. On the one hand, SWIFI tools can be seen as a low-cost alternative to HWIFI that provide a good reproducibility of experiments. On the other hand, their applicability is limited to those fault targets which are accessible by means of software (registers and memories). Nevertheless, SWIFI is a well-suited approach for debugging and testing the software from the dependability perspective, e.g. operating systems, drivers, applications.

As previously stated, the alternative to prototype-based fault injection (HWIFI and SWIFI) is Model-based fault injection. It exercises a model of the system either through simulation (*Simulation-based fault injection*, SBFI) or emulation in FPGA (*FPGA-based fault injection*, FFI); some examples of such emulation-based tools are FT-Unshades [153], and FLIPPER [2]. They allow a very high control on the experimentation process, featuring much higher observability and reproducibility. Unlike HWIFI and SWIFI, model-based fault injection can be applied much earlier in the design flow, thus significantly reducing the cost of fixing existing design problems. At the same time, they also cover most design phases up to the prototype level (FPGA-based). This makes model-based fault injection techniques practically indispensable for supporting the dependability-driven strategies throughout the semicustom design flow. Nowadays there exist a wide range of SBFI and FFI solutions. The rest of this chapter is devoted to the analysis of their advantages, challenges, and practical implementations (tools).

3.2 Fault models

Defining representative faultloads is one of the major issues in fault injection experiments, in particular when faults are simulated or emulated. On the one hand, real faults in the system may have very different origin – from physical defects at the hardware level up to software bugs at the application layer. On the other hand, as it is pointed in [8], to characterize the behaviour of computing systems in presence of faults it is not necessary to inject real faults, it is sufficient to inject faults that induce a similar behaviour of the DUTs in terms of activated errors. Similar errors can be induced by different types of faults like, for instance, an incorrect value in a memory cell may be caused by the impact of an ionizing particle or by electromagnetic interference. Hence, it is important not to establish an equivalence in the fault domain, but rather in the error domain. Despite fault injection still should consider different representation levels of the system throughout the design flow (to account for the impact of involved processes and technologies on dependability), it may rely on common fault models, which abstract the fault causes, while reproducing the fault effects in a similar way at the considered abstraction levels.

Numerous works in the domain have dealt with the definition of representative hardware fault models by studying the manifestation at logic and RT levels of physical CMOS defects [173], including those induced at operation [63]. Common logic fault models devised by these studies are commonly classified in three categories according to their persistence:

- *transient faults* [23], appearing for a short period of time, being usually induced by the interaction of the circuit with the physical environment, such as electromagnetic interference (EMI), crosstalk, impact of ionizing particles, etc.
- *permanent faults* [152], appearing due to irreversible circuit defects both due to manufacturing imperfections and environmental effects, and remaining active for a long period of time (until fault handling actions take place).
- *intermittent faults* [43], appearing and disappearing periodically, usually tending to become more notable in a long period of time due to wear-out mechanisms, and finally to manifest as permanent faults.

Table 3.1: Common logic fault models

Fault model	Description	Persistence	Target Logic
Bit-flip	Models the occurrence of Single Event Upsets (SEU) that inverts the logic state of a register, latch or memory cell. Does not have any associated persistence time, as it remains in the system until the affected state element is rewritten by normal circuit operation.	Transient	Sequential
Pulse	Models the occurrence of a Single Event Transient (SET) (voltage spike) in the combinational logic. The logic state of the affected node remains inverted for a short period of time (usually shorter that clock cycle), retaining its proper logic level afterwards.	Transient	Combinational
Delay	Models the violation of timing properties of circuit elements from the expected values. Usually models the increase of propagation time though the circuit and its individual logic gates. When affecting the registers, it causes the violations of setup/hold times.	Transient/ Permanent/ Intermittent	Sequential/ Combinational
Indetermination	Models undermined logic state of logic node, caused by unstable voltage between high and low thresholds.	Transient/ Permanent/ Intermittent	Sequential/ Combinational
Stuck-at	Models the binding of a logic node (output of sequential or combinational primitive) to a determined logic value (1/0), and the retaining of this value independently of the inputs of affected primitive.	Permanent	Sequential/ Combinational
Short/Bridging	Models interconnection (routing) defect, resulting in the short-circuit of two circuit lines. The resulting effect depends on the logic value and relative strength of each signal.	Permanent	Combinational
Open	Models the break of the line into two independent segments, resulting in the loss of driving signal for affected nodes.	Permanent	Combinational

Most commonly used models of permanent, transient, and intermittent logic faults are described in Table 3.1. It is important to note that Single Event Upsets (SEU) in configuration memory (CM) are currently considered as prevalent dependability threats for FPGA-based designs [167]. Being modelled as soft-errors (bit-flips) at the CM layer, SEUs may cause a wide range of diverse effects in the underlying FPGA fabric, corrupting both the logic and the routing for long periods of operation (until FPGA reconfiguration or CM scrubbing take place).

As it is explained in [22], CM upsets affecting the LUT content corrupt the logic function implemented by the LUT, in such way that it produces an incorrect output only when its input value is the one associated with the faulty CM bit; for that reason, this effect cannot be simulated by the stuck-at fault on the LUT interface. CM upsets affecting routing resources may activate or deactivate programmable interconnection points (PIP) within switchboxes, causing several possible topological modifications [21] (see Fig. 3.2): (i) disconnection of net segment (*open fault*) due to PIP deactivation (deletion of routing segment), (ii) *antenna fault* when a new routing segment is instantiated between an unused input node and a used output node, (iii) *conflict fault* when new routing segment is instantiated between used input node and used output node, and (iv) *bridging fault* when an existing routing segment is deleted and a new routing segment is created between a used input and the output of the deleted routing segment.

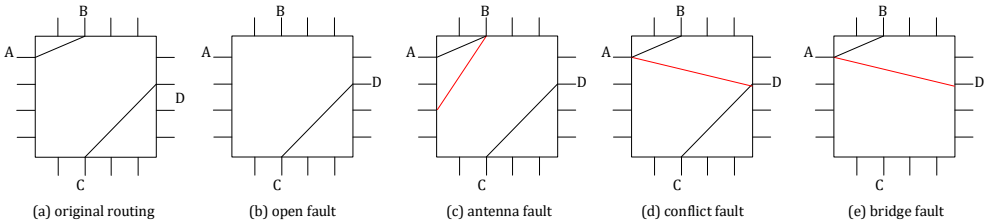


Figure 3.2: Routing faults within the FPGA switchbox (according to [21])

By comparing the results of electrical fault injection with behavioural simulations, the works in [22] [21] have reported that aforementioned topological modifications within the switchbox can be mapped onto following logic fault models at the netlist level: (i) *stuck-at 1/0*, which binds a net (that is connected to an affected switchbox) to a constant logic 1/0, (ii) *logical bridge* between two nets, which causes that their logic values are exchanged, (iii) *wired-AND* (*wired-OR*) between two independent net segments B and D, originally connected to the nets A and C respectively, which drives both of them to $A \wedge C$ ($A \vee C$), (iv) *wired-MIX* between two independent net segments B and D, connected to the nets A and C respectively, which drives B to logic 1, and D to logic 0 when $A \neq C$, and keeps

them unaltered otherwise. As it has been previously mentioned, the analysis of such SEU effects requires the knowledge of bit-accurate mapping between the FPGA fabric resources and the underlying CM. Despite this mapping is rarely available in case of modern FPGAs, several existing tools proposed in [131] [31] are able to establish such mapping internally for certain types of logic resources (though not detailing the mapping algorithms).

3.3 Simulation-based fault injection

Simulation-based fault injection analyzes the effects of faults in HDL models, which can be described at different abstraction levels. The faults are induced by altering the logic values and timing parameters of DUT elements during the simulation.

3.3.1 SBFI techniques

SBFI approaches rely on the modification/instrumentation of the HDL model and/or on simulator commands [62]. Those that instrument the model use *saboteurs* and *mutants*. Saboteurs are special components inserted within signals that alter the value or timing characteristics of those signals when activated. Mutants are variations of existing components that can behave like original components and reproduce their behaviour in presence of faults upon activation of the injection signal.

Several existing instrumentation-based solutions can be noted. First of all, [62] proposes generic procedures for the definition of mutants and saboteurs at the behavioural level. It considers a wide set of fault models including bit-flip, stuck-at, pulse, indetermination, delay, etc. The VERIFY [149] tool, makes the library provider responsible for taking into account all possible faults in the macrocells and requires a non-standard HDL simulator to support an extension of the VHDL language. The proposal in [106] instruments the original library for post-synthesis simulation to enable the injection of SEUs in FPGA-based designs. A saboteur-based technique [18] takes into account the setup/hold time window to properly simulate bit-flips in clock-gated ASICs. Although these methods enable the injection of sophisticated fault models [16], they are highly intrusive, as they introduce/replace components into/from the original model. Moreover, they cause a significant overhead in the experimental process. If the model is instrumented for each experiment, which may take a couple of minutes for a medium-sized netlist (100K gates), this means that the experimentation will take 2 additional weeks to complete for a campaign consisting of only 10000 experiments. If the model

is instrumented to support the injection of several faults, then the size of the model may increase so much that it will slow down the simulation of each single experiment.

Table 3.2: ModelSim commands to simulate the fault effects at RT level (as proposed in [15])

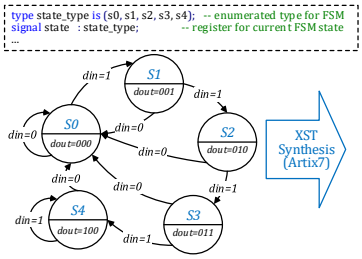
Fault model	Simulator commands (RTL injection approach)
Stuck-at(Signal, value)	*force –freeze Signal Value*
Indetermination(Signal)	*force –freeze Signal 'X'*
Pulse(Signal, Duration)	*set Value [examine Signal]* *force –freeze Signal [expr $Value^1] –cancel Duration*
Bit-Flip(Signal)	*set Value [examine Signal]* *force –deposit Signal [expr $Value^1]*
Delay(TimeVar, Duration)	*change TimeVar Duration*

Simulator commands avoid instrumenting the model by changing the value of signals/variables at run-time to simulate the effect of faults [19]. Particularly, the ModelSim/QuestaSim [112] environment can use the following commands to induce faults into the DUT at run-time: (i) *[examine Signal]* – it returns the current value of a signal/variable, (ii) *[force -freeze Signal Value -cancel T]* – it assigns a new value to a signal and retains it for T time units or until it is cancelled by a *noforce* command, (iii) *[force -deposit Signal value]* – it assigns a new value to a signal and releases it immediately, so the signal remains driven by the model, (iv) *[change Var Value]* – it assigns a new value to the VHDL Variable (similar to *force -deposit* for signals). Table 3.2 lists the RT-level injection procedures for most commonly used logic fault models.

Most notable tools supporting SBFI by simulator commands are MEFISTO [80] and VFIT [15] tools. An alternative command-based approach [44] injects stuck-at faults into Verilog models by including Verilog instructions in the testbench. The approach proposed in [122] injects logic faults at gate-level with a rate dependent on the switching activity of the netlist. Even if the set of logic fault models that can be injected using simulator commands is smaller than instrumenting the model, this is the preferred technique to reduce the time overhead induced on simulations. However, the same approaches used for RTL models cannot be applied to implementation-level ones, as simulator commands sequences do not take into account the particular architecture of macrocells (see Fig.2.4 for details), which may result in an unexpected behaviour of the target component in presence of faults.

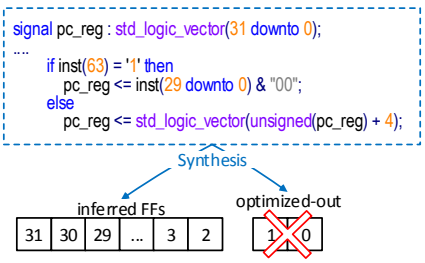
3.3.2 Insufficiency of RT-level fault injection

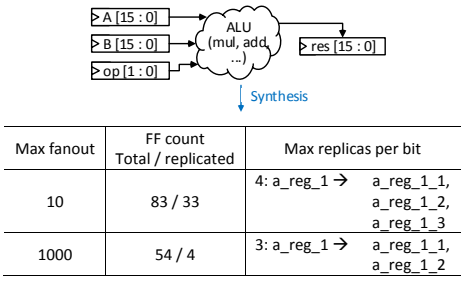
SBFI experiments are commonly conducted using the source RTL model, which describes the design in terms of registers and data flow between them. Despite some works [157] establish the correlation between RT-level and gate-level SBFI results, RTL models still provide rather limited capabilities with respect to dependability assessment. On the one hand, RTL models do not support the simulation of faults in combinational logic, since its actual structure only becomes known after logic synthesis. On the other hand, the representativeness of injection experiments targeting the sequential logic (registers) may be affected by the various optimizations that EDA tools can introduce during the implementation of HW designs. Some EDA optimizations such as duplication, elimination, and balancing of registers (among others) [90] may prevent the systematic use of RTL models for taking decisions regarding the dependability features of the design.



FSM Encoding Algorithm	Resulting encoding	FF
Compact	S0: 000 S1: 001 S2: 010 S3: 110 S4: 100	3
One-hot	S0: 00001 S1: 00010 S2: 00100 S3: 01000 S4: 10000	5
Speed	S0: 1000000 S1: 0100100 S2: 0010010 S3: 0000111 S4: 0001001	7

Figure 3.3: Impact of Finite-State Machine (FSM) encoding option on sequential logic (FF) produced by synthesis tool





Max fanout	FF count Total / replicated	Max replicas per bit
10	83 / 33	4: a_reg_1 → a_reg_1_1, a_reg_1_2, a_reg_1_3
1000	54 / 4	3: a_reg_1 → a_reg_1_1, a_reg_1_2

(b)

Figure 3.4: Optimization of the sequential logic during logic synthesis: removal of redundant registers (a), and register duplication (b)

Let us consider the case of a bit-flip injected in the finite state machine (FSM) displayed in Fig. 3.3. The *state* register of this 5-state FSM is defined as a signal

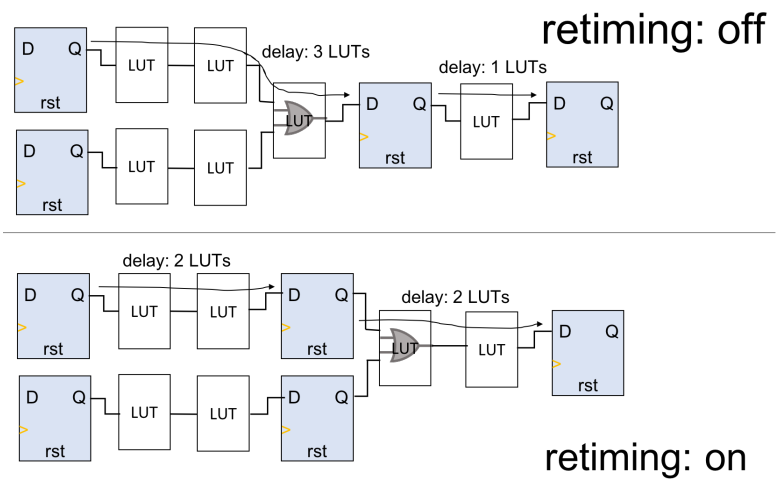


Figure 3.5: Impact of retiming in HW implementations

of type *state_type*. At RTL level, the *state* register can only adopt one of the 5 possible values (*s0*, *s1*, *s2*, *s3* and *s4*) in the enumeration *state_type*. This register can be encoded in the implementation in one way or another depending on the value of certain options of the synthesis tool. In the case of the XST tool, the *FSM encoding* option can adopt, among others, the values *compact*, *one-hot* or *speed* (see Fig. 3.3). As a result, the *state_type* can be encoded, and the *state* register implemented, using 3, 5, or 7 bits. From a fault injection perspective, the RTL representation of the *state* register does not allow the injection of all possible value-faults of such flip-flops, whereas the binary representation obtained after the model synthesis does. As a result, the effect of bit-flips cannot be accurately reproduced at RTL.

Other synthesis optimizations affecting the structure of the inferred sequential logic may also condition the level at which faults can be injected. Consider now the example of the register removal (Fig .3.4-a) and the register replication (Fig. 3.4-b) synthesis optimizations. The former prevents the inference of redundant or unused Flip-Flops, while the latter, on the contrary, replicates them to reduce the fan-out, attending to constrains that can be either explicitly specified or implicitly required by the implementation technology. From the perspective of fault injection this means that, on one hand, optimized-out register bits in implementations should be excluded from SBFI experiments carried out at RTL in order to get comparable results. On the other hand, since duplicated registers

do not appear at RTL descriptions, the effect of faults affecting such registers can only be studied by considering lower level (post-synthesis) models.

The re-timing approach, also known as register balancing, is another widely used synthesis technique to increase the frequency of implemented circuits by reducing the delay of their critical paths. Fig. 3.5 shows a circuit with a delay of 3 LUTs (LookUp Tables) between the first and second D flip-flop and a delay of 1 LUT between the second and the third flip-flop. By activating the retiming optimization, the design becomes more balanced in terms of delays, which results in a critical path with a delay of 2 LUTs. Whenever this technique is applied, the configuration of the circuit changes, although from a functional viewpoint, it behaves the same. As a result, an RTL model is not appropriate to estimate the impact that this type of optimizations may have in the behavior of circuits in the presence of faults.

As it can be seen, there are many situations leading to the need of considering very detailed models during SBFI. This subsection has only presented some of them, but they are sufficient to show the type of changes and optimizations introduced in the structure of circuits that will be neglected when implementation models are not considered. Nevertheless, RTL models are useful to accelerate the SBFI experiments by following a mixed-level and multi-level fault injection approaches, as it will be shown in Section 6.3.1. Particularly, RTL models can be used for the study of the impact of faults in those elements that can be mapped on lower level (and more detailed) models.

3.3.3 Performance and accuracy challenges of implementation-level SBFI

Simulation-based fault injection is commonly applied to RTL models due to the uniformity of involved procedures, and the low computational cost of the simulation process. At the same time, as it has been previously discussed, the dependability assessment of HW designs cannot rely solely on RTL-level SBFI. Very detailed implementation-level models should be considered to take into account the impact of the target technology and EDA optimizations on resulting implementations. However, SBFI at implementation level faces two important challenges.

The first challenge is related to the very high computational complexity of implementation-level models. First, simulation components provided by device vendors (known as macrocells, as explained in Section 2.1.3) describe very accurately the timing behaviour of components. An inverter which is defined as 1 line in RTL (a

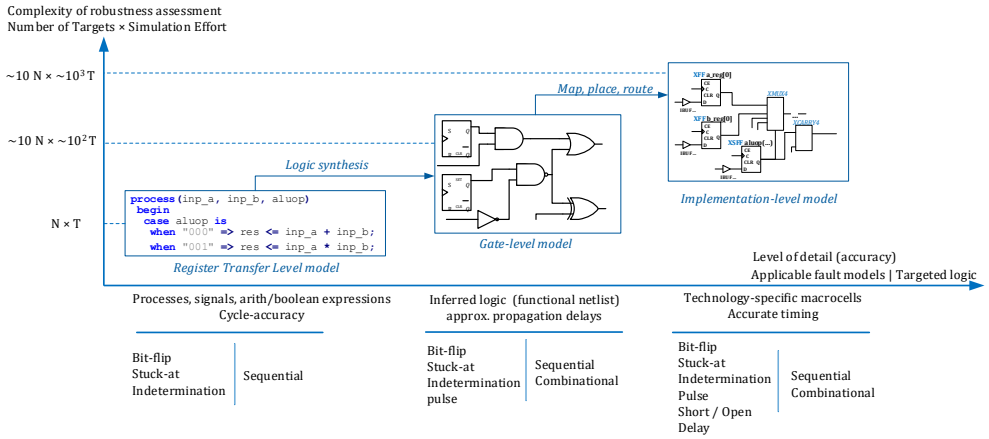


Figure 3.6: Relative accuracy and complexity of fault injection experiments at different levels of HDL description

$<= not\ b$) is translated into the macrocell described in Listing 2.1 with 40 lines of code (removing comments and blank lines), without considering the code of required libraries. Second, a simple combinational component like an 8-bit adder which is defined in 1 line in RTL ($result <= a + b$) is translated into a set of 34 interconnected macrocells (16 input buffers, 8 output buffers, 8 look-up tables and 2 carry chains). This level of detail implies a drastic increase of the simulation effort, as the model approaches to implementation level, slowing down from 2 to 4 orders of magnitude with respect to a similar RTL model [150], Fig. 3.6 summarises this discussion. In practice this may lead to very high (even unfeasible) SBFI effort, especially under complex experimentation scenarios considering multiple (alternative) implementations.

Another challenge concerns injection the procedures themselves, as they are not as straightforward as at RTL. Indeed, macrocells should comply with the standards and guidelines established for a selected description language. This is to ensure the compatibility of defined models with off-the-shelf simulators and also improve the simulation performance. Particularly the basis for VHDL models is established by the VITAL standard, discussed in Section 2.1.3.2. The complex structure and performance-aware optimizations of VITAL-compliant macrocells prevents the usage of the same injection procedures as at RTL.

For instance, Fig.3.7 illustrates the result of applying the same bit-flip injection script to the RT-level and the implementation-level model of a 3-bit counter. The implementation-level model is built by Xilinx ISE suite in the basis of Xilinx's

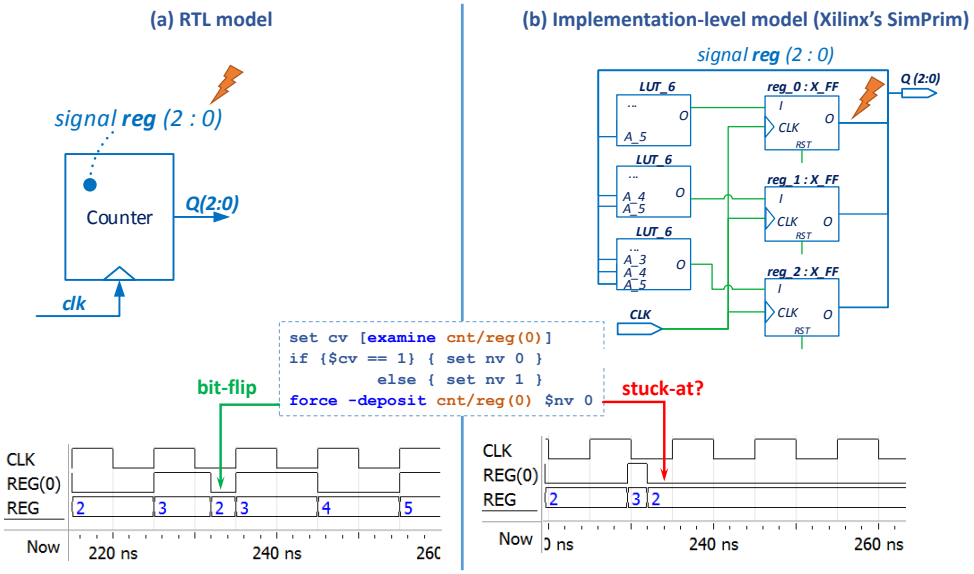


Figure 3.7: Applying the same injection script to the same node within the counter model at two description levels: implementation-level model fails to reproduce the expected bit-flip effect

VITAL-compliant SimPrim library. The injection script is applied at both levels to the same node *'reg[0]'* at the same time instant 232 ns using ModelSim as simulator. As it can be seen, at RTL this injection procedure produces the expected effect: after flipping the *'reg[0]'* signal, it is recovered at the following clock cycle; so the counter continues its operation merely missing one clock cycle. While at implementation level the *'reg[0]'* signals hangs in a faulty state, being this behaviour generally not representative of a bit-flip effect. The explanation of this unexpected behaviour relies on the detailed study of *X_FF* macrocell's internal structure and underlying VITAL semantics, which will be provided in Section 4.2.3.

Although, comprehensive guidelines for VITAL-based component modelling have been presented in [118] and [41], the fault injection problem has not been covered. Furthermore, existing proposals dealing with fault injection in VITAL descriptions lack practical details and/or are highly intrusive. In such a way, a very global view on this problem has been presented in [53] without taking into account the representativeness and accuracy of considered fault models. A 'mutant-based' SBFI solution, specifically targeting ASIC standard cell libraries has been presented in [145]. It covers some aspects of VITAL level-0 and level-1 compliance.

However, it presents at the same time a rather high-intrusiveness and does not address the problems arising when carrying out SBFI experiments by means of simulator commands.

3.3.4 SBFI tools

Fault injection procedures enabling the dependability assessment of final implementations and the verification of deployed non-functional strategies, must be at the core of any dependability-driven approach within the semi-custom design flow. Likewise, all these approaches should seamlessly integrate into this flow, enabling the use of any standardized HDL, off-the-shelf EDA tools, manufacturer libraries, and implementation technologies in a transparent way.

Nowadays, there exists a wide range of SBFI tools that could be considered for their integration into the semi-custom design flow under the stated conditions. The MEFISTO tool [80], for instance, targets RTL and technology-independent implementation-level HDL models. It supports the use of simulator commands and saboteurs. It speeds up the execution of fault injection campaigns by scheduling the simulations on a network of workstations. However, it only works on models defined in the VHDL language. The VFIT [15] tool supports a wider set of fault models, but also handles only RTL models specified using VHDL. Likewise, ALIEN [139] targets RTL models, but relies on highly-intrusive (mutant-based) injection approach. The VERIFY [149] tool supports both RTL and generic netlists, while keeping simulation time manageable thanks to the use of multi-threaded fault injection. The limitation in this case is related to the need for instrumenting vendor-specific libraries using an extension of the VHDL language, as well as for custom simulators supporting these extensions. The ASPHALT tool [190] develops the fault-models by abstracting the effects of low-level (in fact gate-level) faults to RTL. Through this process the simulation remains independent from the implementation details, despite obtaining accurate results and introducing a low overhead. However, ASPHALT only targets processor models.

Therefore, as it is summarized in Table 3.3, none of these tools completely meets the accuracy and performance requirements to support the dependability-driven processes identified within the semicustom design flow. Among other limitations, they cannot be easily extended to support new fault models, provide very basic analysis and reporting facilities, and provide very limited measures to speed-up the remarkably slow simulation of implementation-level models.

Table 3.3: Characterization of some well-known SBFI tools

	HDL				Abstraction level				Fault models			Intrusiveness			Speed-up strategies			Analysis[1]	
						netlists						Low	High						
Tool	VHDL	Verilog	SystemVerilog	SystemC	RTL	Technology-independent	Technology-dependent	Fully routed	Value	Timing	Routing	Simulator commands	Mutants	Saboteurs	Faultload optimization	Multiprocessing	Others	Static report	Interactive web-based
MEFISTO [80]	✓				✓	✓			✓		✓[3]	✓		✓				✓	
ASPHALT [190]	✓				✓				✓								✓	✓	
VERIFY [149]	✓				✓	✓			✓							✓		✓	
VFIT [15]	✓				✓				✓	✓[2]	✓[3]	✓	✓	✓		✓		✓	
ALIEN [139]	✓				✓				✓				✓			✓		✓	

[1] All tools provide failure mode analysis and latencies estimations.
[2] Synthesisable RTL models usually consider timing just at a clock-cycle granularity.
[3] Saboteurs enable the emulation of routing-related problems at RTL models from a high level perspective.

3.4 FPGA-based fault injection

FPGA-based fault injection (FFI) has notably evolved during the last two decades. Early solutions [98] [7] [6] treated FFI as an efficient means to speed-up model-based fault injection experiments, considering a wide range of faults models. At the same time, the steady trend towards employing SRAM-based FPGAs in critical applications requires to use FFI for assessing the dependability of final systems. In this case FFI usually focuses on emulating the occurrence of SEUs in FPGA configuration memories, as they pose the highest dependability threat. On the one hand, modern FPGAs allow to deploy the complete FFI flow on chip, thus eliminating most performance bottlenecks of earlier solutions. On the other hand, the lack of instrumental support to relate the utilized FPGA fabric resources with the underlying CM cells complicates the deployment of fine-grained FFI experiments. Following subsections study existing FFI techniques (Section 3.4.1), the approaches they use to identify relevant fault targets in FPGA configuration memories (Section 3.4.2), and the tools supporting those techniques and approaches (Section 3.4.3).

3.4.1 FFI techniques

The intensive adoption of SRAM-based FPGAs as target implementation technology in critical applications makes FFI an indispensable technique for the assessment of final implementations, rather than just a faster alternative to SBFI. There exist two major approaches for emulating faults in FPGAs: (i) instrumenting the implementation-level model, and (ii) using the runtime reconfiguration capabilities of modern FPGAs.

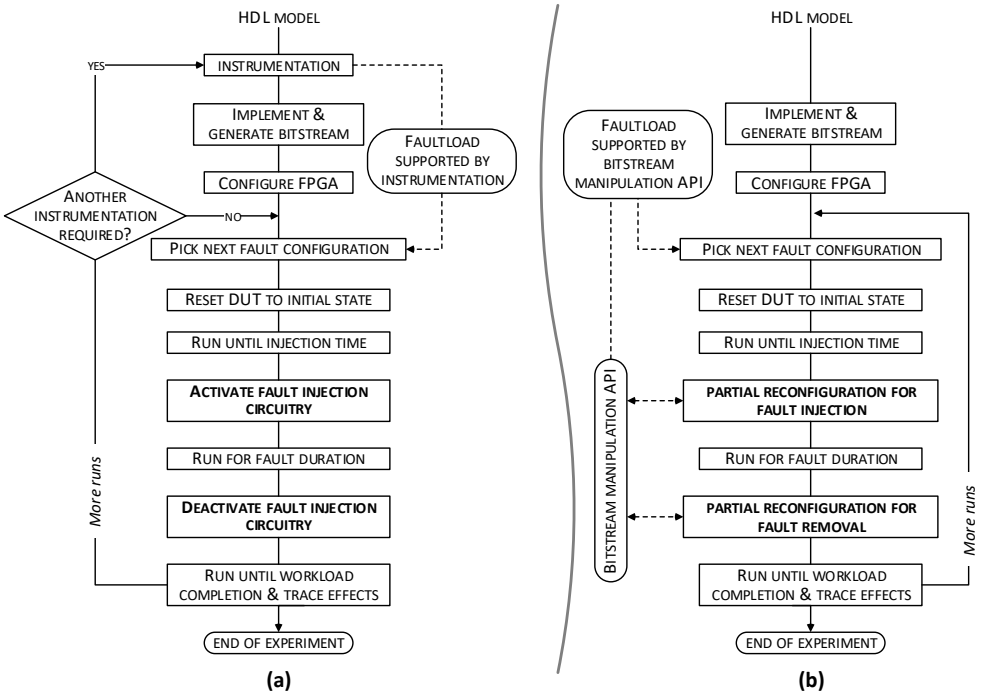


Figure 3.8: FPGA-based fault injection flow: (a) based on model instrumentation, (b) based on runtime reconfiguration

Instrumentation-based approaches integrate into the original HDL design an additional control logic that, similarly to saboteur-based SBFI, allows to inject faults into selected design nodes. Instrumentation can be accomplished at RT level before synthesis, as proposed in [39], or at a lower (netlist) level, as proposed in [162]. The bitstream of the resulting instrumented design is generated and downloaded to the FPGA. During the execution of the Golden-run the fault control circuitry remains inactive. In subsequent runs, the fault control signals are activated in a way that is determined for each fault configuration to emulate the

occurrence of a given fault. The supported faultload is completely determined by the given instrumentation. Therefore, the FPGA utilization may limit the instrumentation of all fault targets at once, thus requiring several instrumentation steps until the whole set of fault targets is covered. The instrumentation-based FFI flow is depicted in Fig.3.8-A.

The main advantage of instrumentation-based FFI is the low time overhead of the injection process. The fault injection can be accomplished while the DUT keeps running, not requiring to pause the workload execution. The main challenge usually addressed by instrumentation-based techniques is the reduction of the area overhead imposed by the fault control circuitry. Another important limitation of this approach is the high degree of intrusiveness it induces into the considered DUT. Indeed, it must be ensured that the integrated fault injection control logic does not interfere with the DUT. But even in this case, any modifications committed at RTL or netlist level, may impact the synthesis/placement/routing results, in such a way that the resulting implementation may significantly differ from the one that would be produced when fault injection is not considered. This may be acceptable when the goal of using FFI is to speed up RTL-level SBFI, but it will be inappropriate when using FFI for assessing the dependability of the final implementation of a particular design (unless all deployed instrumentation logic remains in the final system).

An alternative FFI approach, which has received more attention during the last two decades, relies on the use of the runtime reconfiguration (RTR) capabilities of modern FPGAs. Earlier proposals emulated faults in FPGA fabric by altering the circuitry implemented in the FPGA using vendor-specific APIs. Particularly, the Jbits library supported by Xilinx devices of earlier Virtex series, allowed to transparently update the CM content in accordance with the modifications applied at the netlist level. In such a way, a proposal in [5] defined a set of operations, which can be applied to reconfigurable elements (LUTs, FFs, MUXes, pass transistors) through the Jbits API, to emulate the effects of a wide range of transient and permanent faults affecting the FPGA fabric. Particularly, by monitoring the current state of logic gates, manipulating their interconnections, and allocating the additional logic primitives at runtime, it emulated the bit-flip, stuck-at, pulse and short faults. By altering the routing and allocating the additional logic cells, it increased the fan-out of selected nets, thus emulating the delay faults. Fig.3.8-B illustrates the generic workflow of RTR-based FFI.

The main advantage of RTR-based FFI is its low intrusiveness, since faults are emulated in the original circuit merely by manipulating the FPGA configuration memory (CM). At the same time, the applicable faultload in this approach is determined by the capabilities of APIs available for the bitstream manipulation.

The Jbits library is currently obsolete without any equal replacement. Hence, RTR-based FFI tools relying on modern Xilinx devices should develop their own custom solutions for CM manipulation.

Another challenge addressed by early RTR-based FFI was the reduction of time overheads introduced into the experimentation process by runtime reconfiguration procedures. One of the major performance bottlenecks of earlier proposals was that the injection process was controlled from the host PC, accessing the CM through the relatively slow JTAG Test Access Port (TAP).

The lack of instrumental support for bit-accurate bitstream manipulations complicates the application of these earlier fault emulation procedures to modern FPGA devices.

On the one hand, these procedures can be adapted by relying on the Xilinx's RapidWright framework [96], which automates the manipulation of low-level netlists similarly to the Jbits API. For instance, a work in [94] relied on the first generation of this framework (called RapidSmith) to inject faults into FPGAs by applying modifications at the netlist level. This framework, however, is not capable of reflecting the netlist changes directly into the bistream. Instead, the updated netlist should be supplied to the Vivado/ISE suite to load and process the design and generate the partial bitstream. The bitstream can be then loaded to the device to emulate the fault effect. Hence, as this adapted procedure requires the invocation of the Vivado-based software stack at FFI runtime, it significantly limits the experimentation performance. Despite this approach can be useful for very targeted experiments to test FPGA-specific fault tolerance mechanisms (like the one demonstrated in [94]), its application to the dependability assessment of complex designs leads to rather unfeasible experimentation time.

On the other hand, FFI does not necessary deal with sophisticated fault models that require alterations at the netlist level of the targeted system. Indeed, when FFI is used for dependability assessment of FPGA-based designs, it usually focuses on the emulation of upsets in FPGA configuration memory, as these faults pose the primary threat for SRAM-based FPGAs. Furthermore, modern FPGAs are able to internally access and manipulate their own configuration memory (CM). Hence the complete fault injection process can be deployed directly on chip, eliminating most of the aforementioned performance bottlenecks.

In such a way, most recent works in this field focus on emulating SEUs in using the Xilinx's Internal Configuration Access Port (ICAP) [184]. Some of these works [138] rely on the use of proprietary IP soft-cores, like Xilinx's Soft Error Mitigation (SEM) core [182], to emulate SEUs in CM cells. Others [124] [142]

propose custom fault injection infrastructures, usually based on the Xilinx's Microblaze soft-core processor. In such a way, they have a more precise control over the fault injection process and they are able to define more elaborated injection scenarios than those that can be defined using proprietary IPs. As [34] showed, a complete reimplementation of ICAP libraries can notably accelerate the access to CM. Despite their benefits, common disadvantages of on-chip fault injectors operating through ICAP are that i) they share the reconfigurable fabric with the DUT, reducing the resources available for the target implementation, and ii) they require additional constraints to isolate the injector from the DUT to prevent occasional interferences.

Finally, alternative proposals [163] rely on the use of the Xilinx's Processor Configuration Access Port (PCAP) [188], instead of ICAP, to access the CM from the hardwired processor core embedded into Xilinx's Zynq devices. This significantly reduces the interference of the injector with the DUT and accelerates the injection process, specially when PCAP drivers are also reimplemented [164].

Accordingly, the approach based on runtime reconfiguration can be considered as the most suitable one for the dependability assessment of FPGA-based implementations. On the one hand, its low intrusiveness is favourable for reducing the perturbation induced by the FFI process on the target system under analysis. On the other hand, by eliminating the host-to-FPGA communication bottleneck (using ICAP or PCAP interfaces), the FFI performance can be improved.

3.4.2 Locating the fault targets in the FPGA configuration memory

Dependability assessment through FFI usually deals with the emulation of SEUs in configuration memory (CM), as these faults pose the primary dependability threat for SRAM-based FPGAs. The general procedure for emulating CM upsets is quite straightforward: after pausing the DUT clocking at the injection time, a certain CM cell (target) is selected, its value is read from the device, inverted, and written back to the device; after that the DUT clocking is resumed to analyse the effect of the injected upset. At the device level all CM cells are arranged into *Frames*. A *Frame* is the smallest addressable unit of the configuration memory, which can be independently read/written from/to the CM. In Xilinx 7-Series FPGAs, it comprises 101 memory words of 32 bits each. The frame address (FAR) comprises five fields, listed in Table 3.4. It is worth noting that the frame addresses are sparse and strongly related to the alignment of fabric resources within the given device. The procedures for accessing (reading and writing) the configuration memory frames through different interfaces (defined on the basis of

publicly available Xilinx's documentation and related works), are formalized in Section A.1.

Table 3.4: Frame address composition for Xilinx 7-series FPGA family

FAR filed	Description
FAR [25:23]	block type; CLB configuration/LUT content is located in the Frames of type-0 (000); BRAM content is located in Frames of type-1 (001)
FAR [22:22]	Top (0) or bottom (1) part of the device
FAR [21:17]	clock row counted from the center to top/bottom (as depicted in Fig.2.2)
FAR [16:07]	major Frame address corresponds to the Tile column, i.e. X_{Tile} coordinate
FAR [06:00]	minor Frame address within the column; e.g. CLB columns comprise 36 minor frames

The selection of relevant fault targets is one of the major challenges at FFI. The simplest strategy is to treat the FPGA design as a blackbox and target every CM cell available in FPGA. However, given that modern FPGAs have tens to hundreds of megabits of configuration memory, this strategy leads to an unaffordable experimentation effort. Even if the test is not exhaustive, but restricted to a sample of all available CM cells, the FFI process will remain very costly and inefficient from a time perspective.

One of the ways to address this problem is to restrict the injection of faults to the so-called Xilinx essential bits [97]. These essential bits are those CM cells which determine the circuit functionality and integrity, i. e. those that are used by the design under evaluation. Xilinx EDA tools (ISE and Vivado) export this information in the form of a text-formatted bitmask file (*.ebc). This bitmask was originally intended to be used by the Xilinx Single Error Mitigation IP [182], which corrects the SEUs in the FPGA's CM by scrubbing those bits included in the bitmask. Nevertheless, it can be also used for FFI purposes, allowing to make the fault injection process much more selective in comparison to a completely blind strategy.

However, the use of the Xilinx's essential bits does not completely support the fine-grained FFI. First, the related bitmask only considers non-changeable CM, thus leaving registers and memories out of consideration. Second, it does not allow to selectively locate the CM cells pertaining to the particular design modules and/or types of logic primitives. Though earlier Xilinx tools (ISE suite) considered the possibility of exporting essential bits in a hierarchical way [97], the documentation of Vivado (IDE supporting modern Xilinx devices) does not mention this feature. Finally, the bitmask file (a file with an ebc extension or EBC file for short) itself is not self-descriptive, since the mask is not annotated with frame addresses. Hence to make use of this mask, it should be first mapped onto a list of valid frame addresses for a given device.

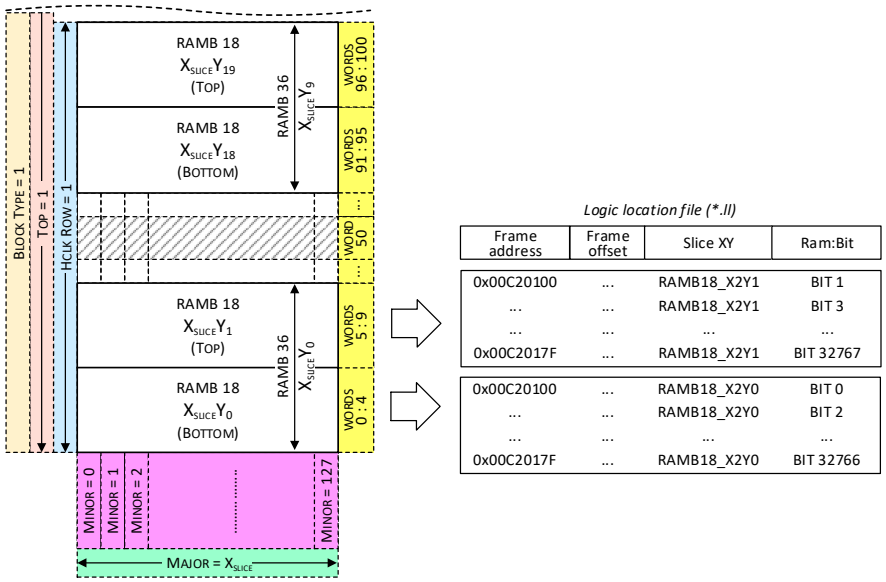


Figure 3.9: Mapping of RAM blocks onto the FPGA configuration memory: logic allocation (LL) file locates all 18K/36K bits of used BRAM18/BRAM36 slices without distinguishing between essential and non-essential bits

The first mentioned limitation of the Xilinx's bitmask can be addressed by combining it with the Xilinx's logic allocation (LL) file. This file locates the registers and memories within the CM. The location of registers is quite straightforward, as for each FF/Latch, the LL-file lists its corresponding readback cell in the CM. For the memory blocks (BRAM), the LL file lists the location of each separate bit of each used BRAM cell. Xilinx 7-series netlists include two kind of BRAM cells: BRAM36 and BRAM18.

The analysis of LL files shows that the content of both types of BRAM spans across 128 Minor frames of type-1, within a given major frame (coinciding with the X_{slice} coordinate). In the case of BRAM36, its content corresponds to 10 consecutive words within each of these 128 frames, as depicted in Fig.3.9. The data bits corresponding to BRAM36 cell are listed in LL file in the range [0:32767], where bits are marked as *Ram=B:BIT*. Each 64 data bits have 8 corresponding parity bits, which are used in simple dual port mode (SPD) under a 512x64 alignment pattern [177]. In the LL file parity bits are marked by *Ram=B:PARBIT*. Each BRAM36 can be seen as a tuple of BRAM18 cells, whose content is interleaved as follows: BRAM18 on the bottom (even Y coordinate) includes even

data bits, while BRAM18 on the top (odd Y coordinate) includes odd data bits. Unlike FF/latch mapping, the LL file does not list any information regarding the source design nodes for the BRAM cells. All entries are identified by the tuple (Slice XY, Ram:Bit), being those listed for the complete BRAM BEL, even if it is used just partially to implement the small RT-level memory. This obfuscated and redundant BRAM mapping style significantly complicates the use of LL files in the context of FFI campaigns requiring a bit-accurate fault injection.

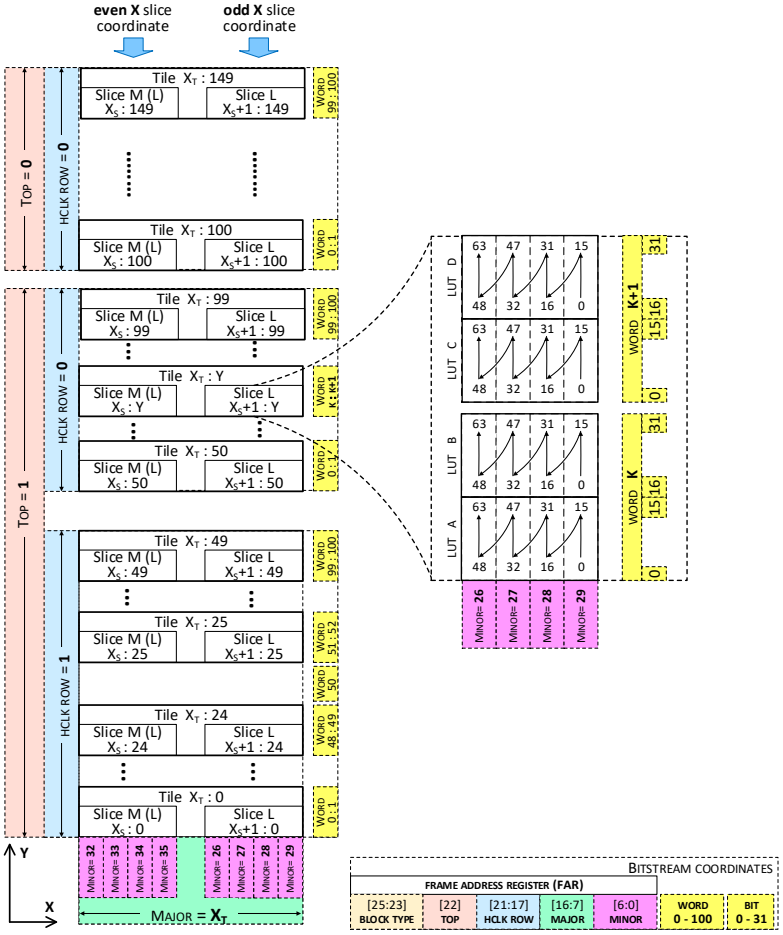


Figure 3.10: Coarse-grained mapping of Look-up tables onto the configuration memory

For the mapping of the rest of fabric resources a very scarce information is currently available. Several notable works have dealt with the mapping problem

for modern FPGA devices. First, the COMET tool [30] allows to visualize and manipulate the Xilinx bitstreams, though not detailing a bit-accurate mapping between the CM cells and the underlying logic and routing resources. The PyXEL framework [31] allows to manipulate the bitstreams of 7-series Xilinx FPGAs, as well as to analyse the effects of SEUs in FPGA routing resources. PyXEL internally maps the routing resources (Programmable interconnection points, PIP) onto the CM in order to enable or disable the specific PIPs. However, it does not detail the mapping algorithms and, to the best of authors knowledge, it does not list essential CM cells for individual netlist primitives (macrocells and routed nets). Likewise, the BitMan tool [131] allows low-level bitstream manipulations for Xilinx FPGAs. Particularly, it allows to extract and update the LUT content, however with rather coarse granularity – it is unknown how each of the 64 bitstream bits (extracted for a given LUT BEL) is mapped to the LUT logic function (INIT). The work in [34] describes the LUT-to-bitstream mapping for Xilinx 6-series FPGAs, and provides a low-level ICAP API to manipulate the LUT content on the device, but again with a coarse granularity. Despite that bit-accurate mapping algorithms remain unavailable, useful information regarding the coarse-grained LUT mapping can be extracted from these works.

Each LUT cell in the netlist has an associated INIT value, which determines its logic function (truth table, LUT content). In the simulated netlist, the LUT content is modelled as a 2^N-bits register, where N is the number of LUT inputs. At the device level, LUT cells are mapped to LUT6 or Bels, which have 64 and 32 bits of CM, respectively. Any LUT cell can be located by the coordinates of the LUT BEL where it is placed. This includes the grid XY coordinates of a CLB slice and the LUT Bel label within the slice (A6/A5, B6/B5, C6/C5, D6/D5).

Fig. 3.10 illustrates the relationship between LUT BEL coordinates and the fragment of configuration memory storing its INIT content. Each major frame configures two columns of CLB slices (each CLB Tile comprises two slices). The LUT content for slices with an even coordinate X_S is located in four consecutive minor frames 32-35, whereas the LUT content for slices with an odd X_S is located in minor frames 26-29. Two consecutive words in each of these four frames store the content of all four LUTs in a slice (LUTs A, B, C, D bottom to top). Hence, the bitstream fragment corresponding to a LUT Bel comprises four half-words of four consecutive frames as depicted in Fig. 3.10. It must be noted that word 50 in each frame is reserved for the configuration of clock lanes.

Finally, the work in [81] detailed a bit-accurate LUT-bistream mapping, in order to reconstruct the LUT logic function (LUT cell INIT) from the bistream content. However, this mapping is valid only for a certain type of LUTs (type L), and only when LUT cell inputs are mapped onto the LUT BEL pins in a direct way, i.e.

whenever $A0$ corresponds to $I0$, $A1$ to $I1$ and so forth.. Note that Vivado decides the mapping of LUT cell inputs $I0 - I5$ to LUT BEL pins $A1 - A6$ to reduce the critical path.

Combining the coarse-grained LUT mapping with Xilinx's essential bits could enable the location of LUT-specific essential bits within a selected design scope. However, as it will be evidenced in Section 5.3, the bitmask reported by Xilinx Vivado is often redundant, since all LUT bits are always marked as essential even when a LUT is used just partially.

3.4.3 FFI tools

A major problem of FFI tools is that they are tightly related to the vendor-specific technologies, thus quickly becoming obsolete as vendors release new FPGA architectures, EDA tools, and APIs. Despite the big assortment of FFI solutions available in the literature, only few of them can be applied to modern FPGAs. Some solutions, like those proposed in [54] [55], rely on circuit instrumentation, being highly-intrusive. Their main usefulness consists in speeding-up the injection of faults in models providing a faster alternative than the ones proposed by conventional instrumentation-based SBFI. Other solutions, like the one introduced by [94], rely on netlist manipulations through the Xilinx RapidSmith/RapidWright frameworks. This provides very limited performance due to the involvement of Vivado/ISE stack and due to the existence of host-to-FPGA communication bottlenecks.

Most FFI tools aiming at dependability assessment, deploy the fault injection flow directly on chip (through ICAP or PCAP) and focus on the injection of single or multiple bit upsets in the configuration memory. Some of them [138] rely on the use of proprietary injection IP cores, provided by Xilinx [182] and Intel [78] for their devices. Others develop their own injection cores operating through ICAP, like in the case of the FIRED tool in [124], or through PCAP, like the solution in [163]. The main limitation of these solutions is that they do not relate the targeted CM bits with the hierarchy of the DUT scope and/or logic primitives. So, basically, they propose a blind fault injection process either targeting all CM configuration bits or at best relying on Xilinx essential bits mask. Furthermore, custom FFI tools are rarely publicly available, which prevents a detailed analysis of their capabilities and limits the reproducibility of experiments.

3.5 Existing strategies for improving fault injection performance

Improving the accuracy and reducing the experimentation effort are two conflicting goals in the dependability assessment of HDL designs. On the one hand, accuracy requires the consideration of very detailed implementation-level models, which offers a great potential for providing evaluation metrics with a higher statistical significance than those that can be reported when relying on more simple RTL models for experimentation. On the other hand, the use of implementation-level models increases the experimentation effort potentially beyond the affordable time limits. This problem becomes especially challenging under complex experimentation scenarios where, for instance, the dependability of multiple alternatives must be assessed. Thus, one challenge of utmost importance when dealing with implementation-level fault injection is the ability to conduct as many injection runs as possible in the shortest possible time frame.

State of the art solutions address the aforementioned challenge from two different perspectives: by reducing the total number of injection runs to be carried out, and by speeding-up the execution of individual injection runs. The number of injection runs is reduced by two main techniques: *fault collapsing* and *statistical fault injection*. The former identifies equivalent faults in the fault space. The latter reduces the number of faults to inject by focusing the campaigns only on a sample of all the potentially injectable faults. At the same time, there exists a wide variety of techniques to accelerate each individual injection run. This sections details the main advances in the field of improving the performance of fault injection campaigns.

3.5.1 Optimizing the fault space through fault collapsing

Fault injection experiments characterize the DUT behaviour with respect to a set of possible fault configurations defined in a multidimensional fault space. These dimensions commonly include the type of fault, the location of the fault target within the DUT, the time of fault occurrence, and its duration [191]. The more complex the HDL design and its workload, the larger the fault space to explore.

Fault collapsing is a common approach for the reduction of the fault space. It consists in identifying equivalence and dominance relations between the generated fault configurations. In the Automatic Test Pattern Generation (ATPG) domain, a fault F_i is said to dominate fault F_j if every test that detects F_j also detects F_i [134]. Thus if F_i dominates F_j, and F_j is detectable, only F_j needs to be considered during test generation, since F_i will be detected by the same test.

When several faults are detected by the same tests (dominate each other), they are referred to as functionally equivalent, and only one of them needs to be considered at test generation.

The adaptation of the fault collapsing approach to the RT-level fault injection has been proposed by [24]. It defined three different fault collapsing techniques. The first one, referred to as *workload independent fault collapsing*, analyses the topology of the RT-level model to determine the equivalent SEUs in FFs. For instance, faults in any register R_i connected directly and exclusively to another register R_j belong to the same category as faults of R_j. The experimental gain of this optimization is reported to reduce the fault list by nearly 10% on average [24].

The second technique, referred to as *workload dependent fault collapsing*, relies on simulation to trace read and write operations on memory elements. SEUs are then collapsed using two rules: (i) all SEUs between any operation and any write operation can be omitted since their effects are masked, (ii) all SEUs between any operation and read operation are equivalent. This optimization is reported to reduce up to 80% of fault configurations [24].

Finally, the third technique, referred to as *dynamic fault collapsing*, analyses the DUT state $C(t)$ at several simulation time instants t. If a SEU F_i injected at time t_i is not activated until a certain time $t_e > t_i$, then another SEU F_e injected into the same memory element but at time t_e, can be omitted from the fault list since it is considered equivalent to F_i. Conversely, if during the simulation, some fault F_i is identified as equivalent to some previously simulated fault F_e (with injection time $t_e > t_i$), the simulation can be stopped, reporting the same fault effect as the one observed for F_e.

Despite a quite notable reduction of the fault space (up to an order of magnitude on the whole), this proposal only is only applicable to RT-level models and, to the best of author's knowledge, it has not been adapted so far to the implementation-level models based on technology-specific macrocells that this thesis focuses on.

3.5.2 Statistical fault injection

Statistical sampling is the process of drawing conclusions for an entire population after conducting a study on a sample taken from that population [82]. In statistics, a population denotes a large group consisting of individuals having at least one common feature, while a sample is nothing but a part of the population that is selected in order to represent the entire group. Representativeness is a primary concern in statistical sampling, since non-representative samples of the population

may lead to results that will be different from those generated by considering the entire population.

More formally, statistical sampling [159] consists in selecting a subset of individuals from a population to estimate the characteristics of the whole population (N). Assuming that these characteristics follow a normal distribution and that individuals are randomly selected following a uniform distribution, then the *margin of error* (e) represents the maximum difference between the estimation provided by the sample and the actual value of the characteristic for the whole population, with a given confidence interval (represented by t). If these assumptions hold, the minimum sample size (n) required to achieve a desired error margin with a given confidence interval can be computed by Equation 3.1. A common assumption in statistics is considering a confidence interval of a 95% ($t = 1.96$). This means that the difference between the estimation provided by the sample and the actual value for the whole population will be no greater than the specified margin of error in 95% of the cases.

$$n = \frac{N}{1 + e^2 \times \frac{N-1}{t^2 \times p \times (1-p)}} \tag{3.1}$$

As shown in Equation 3.1, the sample size is also affected by p, which represents the probability of an individual to exhibit the characteristic being estimated. As it is unknown before SBFI/FFI experiment, then the worst case scenario is assumed (the individual may exhibit the characteristic or not with the same probability, $p = 0.5$) and a larger sample size is hence required. Likewise, it is possible to estimate the margin of error for a given sample size through Equation 3.2.

$$e = t \times \sqrt{\frac{p \times (1-p)}{n} \times \frac{N-n}{N-1}} \tag{3.2}$$

The analysis of these equations shows that the increase of population size (N) leads to the increase of required sample size (n) only until reaching a certain population size threshold. As it is depicted in Fig. 3.11, such threshold equals roughly 10^4 individuals for $e = 5\%$, 10^6 individuals for $e = 1\%$, and 10^8 individuals for $e = 0.1\%$. Larger populations (even infinite ones) can be representatively

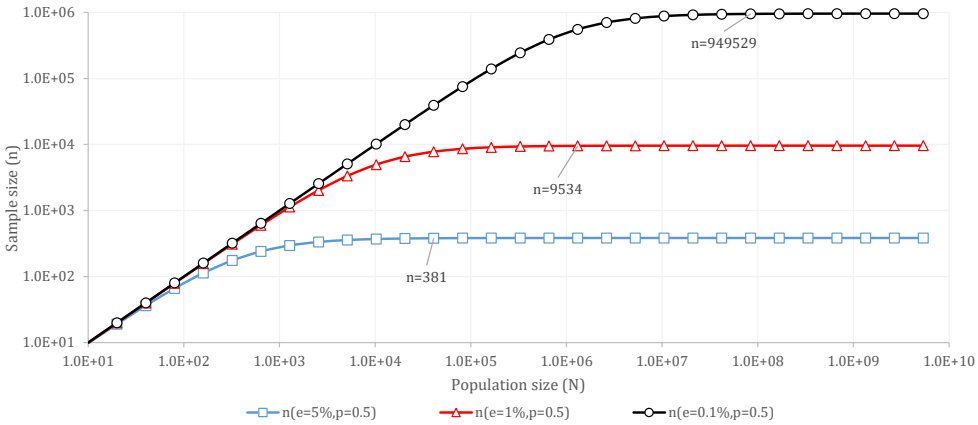


Figure 3.11: Sensitivity of sample size n to the increasing population size N

sampled using simplified Equation 3.3, which is obtained from Equation 3.1 by ignoring the finite-population correction, as it is explained in [159].

$$n = \frac{t^2 \times p \times (1-p)}{e^2} \tag{3.3}$$

Once these concepts are integrated into the fault injection domain, they should be understood as listed in Table 3.5.

Table 3.5: Statistical sampling concepts mapped to the fault injection domain

Sampling	Fault injection
Population (N)	Any fault that can potentially affect each possible circuit (combinational and/or sequential) element or memory cell in each possible clock cycle.
Sample size (n)	Number of faults to consider (faultload), assuming the injeciton of one fault per experiment.
Characteristic (P)	Failure modes rate
Margin of error (e)	Margin of error
Confidence interval (t)	Confidence interval

The research carried out in [100] established the basis for applying the notion of statistical sampling to the injection of HW faults in integrated circuits. In this study, the considered population was defined as all the possible faults (soft errors in fact) that can potentially affect an integrated circuit. As a result, an individual is a fault, i.e. a type of fault (fault model) that may affect a particular

circuit location during a concrete execution cycle. The common feature shared by all individuals was defined in terms of their ability to provoke failures, i.e. the probability of a fault to result in one of the considered failure modes. It is also explained that under the assumption that such features follow a normal distribution and each circuit location has the same probability being affected by a fault, a representative sample can be defined following a uniform distribution for the selection of individuals during statistical (random) sampling. Under such hypotheses, authors showed that it was possible to obtain very precise results about the robustness of a circuit while injecting only a very small portion (sample) of all the possible considered faults. As an example, only 384 experiments were enough to achieve a 5% margin of error with a 95% of confidence in the estimation of the various failure modes that can affect a circuit! Although this level of precision may seem enough for regular designs, it will not be enough in the case of safety-critical systems, where the robustness of designs under study must be estimated as precisely as possible. Keeping the confidence in the reported estimations to 95%, but decreasing the expected error from 5% to 0.1%, the number of experiments will increase from only 384 to nearly 790000.

3.5.3 Speeding-up fault injection runs

Even after reducing and sampling the fault space, the required experimentation effort may still remain very high. Particularly, when targeting implementation-level models, the simulation effort raises up to three orders of magnitude with respect to RTL. This requires additional speed-up measures at the level of the scheduled injection runs. State of the art solutions address this problem from different perspectives.

First of all, exploiting run-time partial reconfiguration capabilities of modern FPGAs for fault emulation has attracted very high attention over the last decade [5] [46]. The most promising proposals focus on exploiting the ability of modern FPGAs to internally access and manipulate their own configuration memory (CM) to emulate the occurrence of Single Event Upsets (SEUs). Thus, the fault injection process is internally managed on chip, eliminating communication bottlenecks with the host. As discussed in section 3.4, FFI solutions relying on the use of SoC platforms (e.g. Zynq) are especially useful, since they avoid the interference between the injector and its target [163].

FPGA-based methods drastically accelerate fault injection experiments - several orders of magnitude with respect to simulation-based techniques. However, their applicability for robustness assessment is limited to FPGA-based designs, since considered measurements to derive target DUT robustness metrics are often spe-

cific to the particular implementation technology. Moreover, they usually rely on vendor-specific technologies and device-specific architectures that quickly become obsolete when FPGA changes. Simulation-based solutions, on the other hand, allow fault simulation for any implementation technology defined in a library of macrocells, thus being much more generic and flexible.

The proposal in [24], in addition to previously described fault collapsing approaches, proposed three run-time optimization techniques. The first one called *early stop* identifies as early as possible the effect of faults during simulation to stop its execution as soon as possible. The second optimization, referred to as *hyperactivity*, proposes to disable the tracing of intermediate fault effects in those cases when the injected fault causes an excessive number of mismatches during the simulation. The fault effect is thus determined by analysing the results at the end of the workload. Finally, the optimization referred to as *smart resume* proposes to reuse the launched simulation session to simulate subsequent faults when the effects of previously injected fault disappeared from the circuit. The speed-up potential is workload-dependent. This approach, however, is described in a very abstract way, not covering neither its practical aspects, nor its application to implementation-level SBFI.

The *multi-level fault injection* is another known concept in model-based SBFI, which aims at simulating faults effects at the highest possible (less costly) level of the design flow. The main challenge here is the identification of representative fault targets at the high abstraction level. However, one should not forget that some faults in state elements (registers/memories) cannot always be properly simulated at RT level. This limitation is related to the impact of implementation technology and EDA optimizations, as discussed in Section 3.3.2. The work in [99] partially addresses this problem. It considered the state registers as the most critical fault targets and demonstrated the feasibility of carrying out a representative simulation of some FSM faults at RT level. To this end it generated RT-level FSM mutants, which take into account the partial post-synthesis information. The reported results showed good correlation between different description levels. However, the scope of this proposal is limited to FSMs, while the mutant-based injection approach is generally considered as highly intrusive.

A closely related approach is presented in [190], which develops fault-models by abstracting the effects of low-level (in fact gate-level) faults to RTL. It reports that the simulation remains independent from the implementation details, despite obtaining accurate results and a low overhead. However, the approach only targets processor models.

Checkpointing is a generic technique which can be also exploited for speeding-up the experimentation process. The idea behind it is to save the intermediate simulation states during the golden run, and to recover them in subsequent injection runs to reduce the simulation time. Despite being a well-known technique, the practical application of checkpointing to SBFI/FFI platforms is rarely discussed. The most notable contribution, exploiting the checkponting in SBFI, is described in [129]. The use of checkpointing in FPGAs is often discussed from the fault tolerance viewpoint [92], but not for speeding-up the FFI experiments. Likewise, the attainable speed-up gain has not been widely discussed by existing proposals.

Finally, *multiprocessing* allows running several fault injection experiments in parallel. The main challenge here is to exploit the available computing resources in the most efficient way. The resulting speed-up gain is directly proportional to the amount of available processing resources (PC cores, grid nodes, FPGA boards). For instance, the proposal in [80] speeds up the SBFI campaigns by scheduling the simulations on a network of workstations and combining such approach with checkpointing in order to reduce the warm-up time of each simulation to the strict minimum. The proposal in [56] relies on a stack of FPGA boards to execute FFI experiments in parallel, thus the reduction of experimental time is related to the number of FPGA boards available for experimentation.

Some works also propose the development of specialized fault simulators, like the one proposed in [119] for IcarusVerilog models. Others rely on the high performance offered by today's GPGPUs to accelerate the injection of faults on simulated models [93]. Although these GPU-powered techniques seem quite promising (speeding up fault injection in up to two orders of magnitude), they encounter problems of flexibility and scalability. To the best of authors knowledge, no GPU-powered solution has been proposed so far that would completely support the standard Verilog/VHDL-VITAL models.

3.6 Conclusions

Simulation-based fault injection (SBFI) and FPGA-based fault injection (FFI) are two valuable techniques supporting the dependability assessment and verification along the semicustom design flow. The SBFI technique covers several major phases of the design flow – from the high-level (RT-level) HDL model, up to the technology-specific post-place-route model (implementation-level). Being applied at RT-level, SBFI allows the early identification of dependability bottlenecks in the design, but operates with a very limited set of fault models (those affecting the sequential logic), and does not take into account neither the synthesis

optimizations nor the impact of the implementation technology. Implementation-level SBFI, on the other hand, can potentially provide accurate technology-specific dependability estimates, taking into account the impact of synthesis, placement, and routing optimizations, but becomes available much later in the design flow and requires much higher experimental effort. In case of FPGA-based design flows, the FFI technique can be used at the later design stages (when an FPGA prototype becomes available) for the evaluation of FPGA-specific faults, such as SEUs in configuration memory, which can only partially be evaluated by means of implementation-level SBFI.

The practical application of these techniques, however, requires to address several accuracy-related and performance-related challenges. First of all, to obtain credible level of dependability measures, the fault injection process should reduce as much as possible the level of intrusion into the DUT. This increases the interest in those solutions which do not rely on DUT instrumentation, such as SBFI based on the use of simulator commands, and FFI based on the runtime reconfiguration capabilities of FPGAs. Despite that several existing SBFI solutions make use of simulator commands, they mostly focus on RT-level models, being in many cases inapplicable to the technology-specific implementation-level models. In fact, to properly simulate the fault effects at the implementation-level, existing fault injection procedures should be revisited to take into account the internal structure and semantics of underlying macrocell libraries used for the circuit implementation.

With respect to FFI, the accuracy challenge concern is related to the fine-grained location of the essential CM cells that should be targeted during experimentation. Despite the existence of the Xilinx essential bits information and several third-party bitstream manipulation tools, the provided information is not enough to allow the identification of CM cells corresponding to a selected design scope or a particular type of logic primitives in FPGA fabric. Inaccurate (or redundant) location of essential bits may degrade the accuracy of dependability estimations and it may also slow-down the experimentation. Therefore, a particular problem to be addressed with respect to fine-grained FFI is the bit-accurate mapping of CM cells to the primitives available in the netlist.

Another identified challenge relates to the reduction of the experimentation effort, while ensuring acceptable levels of confidence in resulting dependability metrics. Existing approaches in this domain focus on (i) reducing the number of experiments to carry out through fault collapsing and statistical sampling, (ii) speeding-up the individual injection runs by abstracting the fault to inject to a higher level, using FPGA-based emulation, checkpointing, and so on, and (iii) parallelizing the execution of experiments. These approaches provide an important performance gain. Particularly, statistical injection may provide an impressive speed-up gain

in case of rough robustness estimations, but might be insufficient when accurate estimations are required (narrow error margin). The multi-level injection and checkpointing require to address many practical aspects of their application to SBFI and FFI environments.

Finally, the analysis of existing SBFI and FFI tools has shown that they do not support their seamless integration into the semicustom design flow. Available tools are usually very specific (limited) to a particular HDL representation level, set of fault models, simulation tools, etc. A more generic and customizable injection tool is thus required to completely cover the diverse dependability-related requirements existing along the semicustom design flow.

Chapter 4

Enabling Low-intrusive Simulation-based Fault Injection for Implementation-level Models

This chapter proposes a new fault injection approach that enables the accurate simulation of logic faults in implementation-level HDL models, while reducing to the minimum the intrusion in targeted models. Section 4.1 describes the challenges addressed by the proposed approach. Section 4.2 defines low-intrusive fault injection procedures for VHDL/VITAL-compliant models. Likewise, Section 4.3 defines fault injection procedures for the Verilog-based models. Section 4.4 proposes a tool-independent fault dictionary format that formalizes the definition of fault injection procedures for the diverse libraries of macrocells. Finally, Section 4.5 concludes this chapter.

4.1 Introduction

As it has been discussed in Section 3.3, the behavioural (RT-level) fault injection provides a very limited insight on the dependability of targeted designs. On the one hand, it allows to simulate only faults affecting the sequential logic. On other hand, it does not take into account neither the synthesis, placement, and routing optimizations, nor the impact of the target implementation technology on dependability. A more elaborated dependability assessment requires to employ implementation-level models. Being the most detailed HDL models in the design flow, they can be used for the accurate simulation of a wide variety of faults, including technology-specific and timing-specific ones.

Implementation-level models are represented by the so-called *netlists* of technology-specific macrocells. As it has been discussed in Section 2.1.3, these macrocells are defined on the basis of comprehensive rules and instruments (libraries) established by VHDL/VITAL and Verilog standards, in order to make them compatible with different simulators and to improve the simulation speed. These standards, however, do not take into account the fault injection capabilities, limiting the ability to properly reproduce the effects of logic faults in implementation-level models.

As it has been explained in Section 3.3.3, existing simulation-based fault injection (SBFI) solutions do not (or only partially) handle implementation-level models. Those SBFI solutions, that are based on simulator commands, are unable to properly simulate the fault effects in implementation-level models, since they do not take into account the structure and semantics of underlying macrocells. Those few solutions, that handle the implementation-level models, are highly intrusive, since they modify the original HDL model (netlist). As it has been previously discussed, this may lead to unforeseen side-effects on the circuit behaviour, degrading the credibility of derived robustness estimates.

This chapter defines a new SBFI approach that addresses these limitations. By studying the structure and semantics of VITAL-based and Verilog-based macrocells, Sections 4.2 and 4.3 respectively define low-intrusive fault injection procedures for most common logic faults. Whenever possible these procedures rely on the use of simulator commands. However, some faults can be injected into VITAL-compliant models only after instrumenting the targeted macrocells. Nevertheless, this approach can be considered less intrusive, since the netlist itself is kept untouched. Furthermore, the absence of side effects can be easier verified at the macrocell level than at the netlist level. After defining the fault injection procedures for the common logic faults, Section 4.4 describes a new fault dictionary format, that generalizes these procedures for macrocells of any complexity, in a compact, flexible, and tool-independent way.

4.2 Fault simulation in VITAL-compliant models

The complex architecture of VITAL-compliant macrocells (previously discussed in Section 2.1.3.2) makes that fault injection procedures, commonly used at RTL, cannot be directly applied to implementation level models. Injecting some (transient) faults requires to take into account the optimizations implemented in underlying VITAL packages, whereas injecting some other faults may require to upgrade (instrument) the macrocells. This section first defines the generic operations that might be required to instrument VITAL-compliant macrocells and to support subsequent fault injections by means of simulator commands. After that, it describes how these operations are applied to the generic and technology-specific macrocells in order to properly reproduce the effects of common logic faults.

4.2.1 Definition of generic operations to support fault injection

The netlist obtained after the synthesis of an RTL model, described using VHDL, consists of a set of interconnected VITAL-compliant macrocells. As these macrocells are also defined in VHDL, it is possible to use the *simulator commands* approach [19] to modify the state of its internal signals and variables to reproduce the effect of a given fault model and observe the behaviour of the system in presence of such fault. Table 4.1 lists the operations commonly used by state of the art simulators to retrieve and modify the contents of signals and variables of the model. These commands have been defined after Mentor Graphics' Modelsim/Questasim commands [112]. For instance, taking Listing 2.1 as a reference, the current state of signal *A_ipd* can be obtained by the *examine(A_ipd)* operation, and the state of the *YNeg_zd* variable can be set to a low logic level by the *change(YNeg_zd, 0)* operation. Nevertheless, not all possible elements of a macrocell can be directly modified by following this approach, thus requiring the definition of additional generic operations to support the required transformations in the VITAL-compliant model.

Sometimes it would be required to modify the value of model parameters that are defined as *generic* attributes, like the timing properties (delays), the truth table of a look-up table, or the initial contents of a memory block. However, as it is noted in Questasim commands reference manual [112], changes in generic parameters cannot take place if the design is optimised for high simulation speed and, even so, such changes may not be propagated to the dependent expressions. This would prevent the injection of delays in any macrocell and SEUs in LUT and BRAM macrocells. Thus, a new operation called *generic2signal* has been defined to include supplementary signals in the macrocell that can capture the

Table 4.1: Operations on VITAL-compliant macrocells to support fault injection

Operation	Type	Description
examine(target)	simulator command	gets the current value of the *target* signal or variable
force(target, mode, value, duration)	simulator command	changes the state of the *target* signal to *value* for *duration* (*freeze mode*) or until overwritten (*deposit mode*)
change(target, value)	simulator command	like the *force* command, but the *target* is a constant, generic, or variable
generic2signal(target)	instrumentation rule	initialises an internal signal with the value of the generic and feeds that signal wherever the generic is used
constant2signal(target)	instrumentation rule	like the *generic2signal* operation, but the target is a constant
addToList(target, signal)	instrumentation rule	adds the *signal* signal to the sensitivity list of the process identified by the *target* label
encloseInProcess(target)	instrumentation rule	encloses the procedure identified by the *target* label into a process activated by all the incoming parameters of the enclosed procedure

value of generic parameters and feed that signals to those elements using the associated generics. Thus, to make injectable the *tipd_A* generic from Listing 2.1, the *generic2signal(tpid_A)* operation should be executed. It will take charge of: i) creating a new signal of the same type as the generic in the declarative part of the model architecture (lines 26–28)—*SIGNAL v_tpd_A_YNeg : VitalDelayType01 := UnitDelay01;* ii) initialising that signal after back-annotation by including the following assignment—*v_tpd_A_YNeg <= tpd_A_YNeg*—outside any other block defined in the body of the model (lines 30–66); and iii) using that signal instead of the generic wherever required, like in line 61—*PathDelay =>v_tpd_A_YNeg.*

A similar procedure but handling constants is defined by *constant2signal.*

Processes are only activated upon changes on any of the signals listed in their sensitivity list. Thus, the transformation of any generic or constant into an internal signal that is used within a process requires also this signal to be included into the process sensitivity list. The *addToList* operation takes care of this transformation. Following the previous example, the *addToList(VITALBehavior, v_tpd_A_YNeg)* operation will include the signal *v_tpd_A_YNeg* in the sensitivity list of the *VITALBehavior* process in line 39—*VITALBehavior : PROCESS(A_ipd, v_tpd_A_YNeg).*

VITAL level 1 enables the deployment of optimisations to speed up the simulation of the model. For instance, when procedures have input parameters taken from generics and constants they are not rechecked during simulation, as they are not supposed to change dynamically. Thus, any fault injected in these elements will not propagate through the model. A new operation called *encloseInProcess* has been defined to insert the desired procedure within a process that will be activated whenever an input parameter changes its value. This will ensure that the new state of generics or constants (previously transformed by *generic2signal*

Listing 4.1: Enabling procedures to recompute incoming generics and constants

```
1    encloseInProcess_w_1 : PROCESS(A, v_tipd_A)
2    BEGIN
3      w_1: VitalWireDelay (A_ipd, A, v_tipd_A);
4    END PROCESS;
```

or *constant2signal* operations) will be recomputed within the process. So, for the *VitalWireDelay* procedure call in line 33 to be aware of a change in its input *v_tipd_A* (resulting from the transformation of *tipd_A* generic into an internal signal), the operation *encloseInProcess(w_1)* will transform it into the code listed in Listing 4.1. It must be noted that, although this process describes exactly the same functionality as the original procedure, it does not comply with the requirements for VITAL level 1, as a *Wire Delay* block can only contain concurrent procedure calls. The instrumented model will be still compliant with VITAL level 0 and it will maintain most of the optimisations available for VITAL level 1, although this particular one (preventing the recomputation of expressions based on generics and constants), will be forfeited.

4.2.2 Stuck-at, pulse, and indetermination faults

The permanent persistence of stuck-at faults makes that their injection into implementation level models could follow exactly the same procedure used for RTL models. In this case, it is just a matter of targeting the signal connected to the output of the selected macrocell instead of dealing with the internal complexities of the component. As the signal will be permanently set to the injected value, it does not matter whether the internal state of the macrocell is really stuck or not. Thus, causing a stuck-at-1 to a flip-flop driving a signal named *ff_o* will be accomplished by using the operation *force(ff_o, freeze, 1)*.

As pulse fault models affect combinational components, and these macrocells do not store any logic value, it is also possible to just target the signal driven by the macrocell. For instance, a pulse can be injected for 10 ns to a look-up-table driving a signal named *lut_o* by means of these operations *newValue = (examine(lut_o) == 0) ? 1 : 0; force(lut_o, freeze, newValue, 10 ns)*. Fig. 4.1 displays the injection of two consecutive pulses in the output of an *X_LUT6* macrocell from the Xilinx's SIMPRIM library [174].

Permanent indetermination faults can be treated as stuck-at faults, and transient indetermination targeting combinational elements can follow the same approach as pulses, but setting the target signal to an 'X' value.

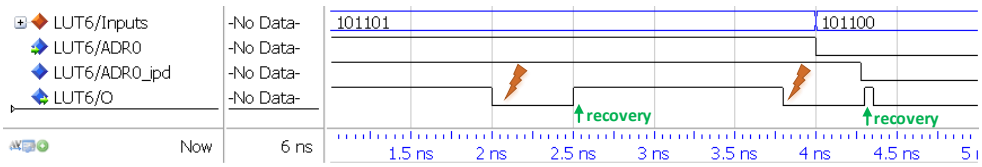


Figure 4.1: Injecting two consecutive pulses into a combinational component

4.2.3 Bit-flip faults in registers

Bit-flip faults can be injected by inverting the logic value of RTL signals that represent sequential elements, like flip-flops. However, when applying that same procedure to the signal driven by the output of the sequential macrocell, the observed behaviour is not the one expected. As depicted in Fig. 4.2a, once the fault has been injected, the flip-flop's state is not restored on the following rising clock edge. On the next simulation event after the fault injection, the *VITALBehavior* process is activated (see Fig. 2.4). As this is a rising edge active flip-flop, the *Functionality* section recomputes the 'zero-delay' output. However, as *O_zd* equals the previously scheduled value (*Path Delay* checks), the output retains its current (faulty) value to optimize the simulation.

To bypass this check, the fault injection procedure should also invert the value of the *O_zd* variable. The simulated fault effect in this case is illustrated on Fig. 4.2b – on the falling clock edge the *schedValue* is changed, thus on the next rising clock edge it does not coincide with the *O_zd* anymore; therefore the Path Delay procedure continues its execution, recovering the flip-flops' output value.

However, if no event occurs after the injection, the next rising clock edge updates the logic value on *O_zd* to '1'. As this is exactly the value that was previously scheduled, the output will retain its faulty logic value. This is a case of injection after the falling clock edge, illustrated in Fig. 4.2c. Thus, the scheduled value should also be targeted to prevent this erroneous behaviour. However, the first action within the *Path Delay* section updates this value according to the last value stored to prevent glitches. Hence, the *O_GitchData.LastValue* variable should also be modified.

Now, as depicted in Fig. 4.2d, the flip-flop recovers from the fault as expected. On the rising clock edge, the scheduled value is updated with the last value and, as being different from that captured on the clock edge, *O_zd* is propagated to the scheduled and last values and the output of the flip-flop. The proposed sequence of simulator commands-based operations is listed in Listing 4.2.

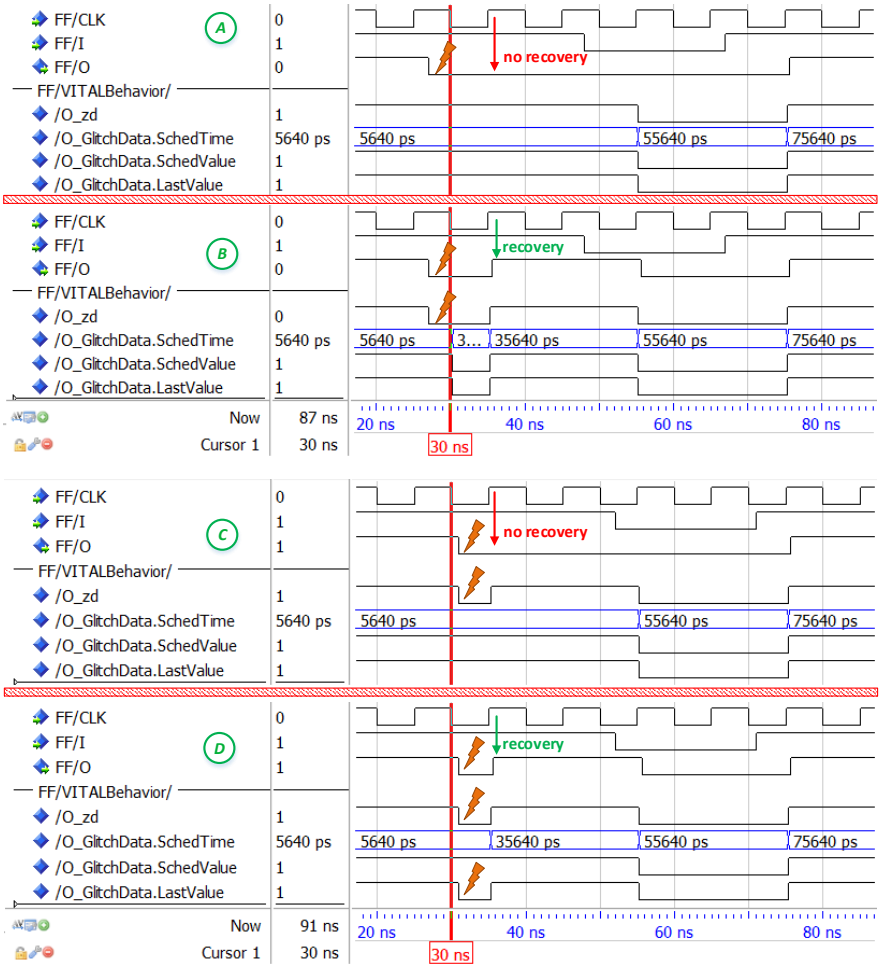


Figure 4.2: Injection of bit-flip into a flip-flop at implementation level: by following the same procedure used at RTL (A), taking into account zero-delay output (B - before rising clock edge , C - after rising clock edge), taking into account zero-delay output and last scheduled value (D - resulting fault injection procedure)

4.2.4 Delay faults

Although possible in theory, delays faults are rarely considered in RTL models, even when implementation level models present a very accurate timing description that enables the injection of these faults.

Listing 4.2: Proposed procedure to inject a bit-flip in the *target* macrocell

```
1  bitflip(target) {
2     newValue = (examine(target/O) == 0) ? 1 : 0;
3     force(target/O, deposit, newValue, 0);
4     change(target/VITALBehavior/O_zd, newValue);
5     change(target/VITALBehavior/O_GlitchData.LastValue, newValue);
6  }
```

Listing 4.3: Enabling the injection of delays in the *target* macrocell

```
1  delayInstrumentation(target) {
2     foreach(genericInput in target) {
3        if (prefix(genericInput, "tpd_") {
4           internalSignal = generic2signal(genericInput);
5           addToList("VITALBehaviour", internalSignal);
6        }
7        else if (prefix(genericInput, "tipd_") {
8           blockLabel = generic2signal(genericInput);
9           encloseInProcess(blockLabel);
10       }
11    }
12 }
```

All VITAL-compliant macrocells must define two main timing generic parameters: interconnect path delays and propagation delays. Being generic parameters, it will be necessary to include some internal signals to be able to modify their values and pass these signals to the elements that use the original generics. In the case of generics used in the *Path Delay* block, this signal must be added to the sensitivity list of the *VITALBehavior* process. If the generics are used in the *Wire Delay* block, then it must be enclosed into a process activated by all its input signals. The proposed procedure for instrumenting a macrocell to support both types of delays is listed in Listing 4.3.

Once the macrocell is instrumented, the available operations based on simulator commands can be used to change the state of those internal signals holding the delay values. For instance, after instrumenting the inverter modelled in Listing 2.1 by means of *delayInstrumentation(std04)*, the propagation delay can be increased in 2 ns by calling *force(v_tpd_A_YNeg, freeze, 2 ns, 0)*.

Fig. 4.3a depicts a warning issued by the simulator due to a setup time violation after increasing the interconnect delay of the data input port *I* of a flip-flop from the Xilinx's SIMPRIM library [174] in 3 ns. Likewise, Fig. 4.3b shows the case

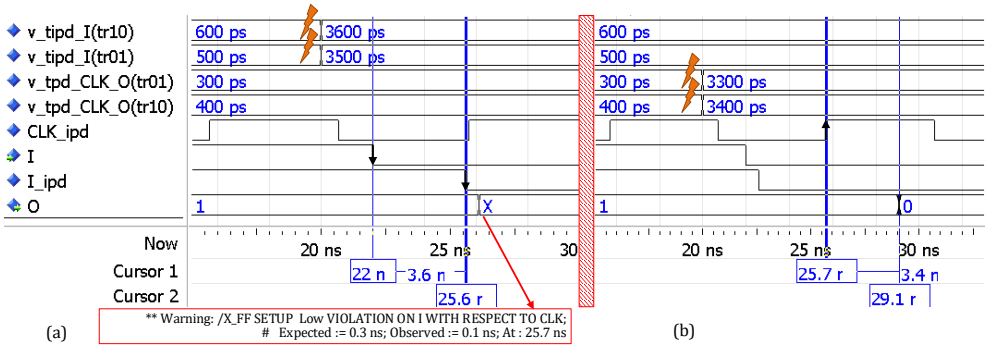


Figure 4.3: Injecting delay faults into a flip-flop: a) interconnect delay of I input, and b) propagation delay from CLK to O path

of increasing the propagation delay from clock input *CLK* to data output *O* in 3 ns.

4.2.5 Considering FPGA-specific components: bit-flips in configuration memory of LUTs

The proposed procedures are generic enough to be applied to any VITAL-compliant macrocell and support the injection of any logic fault not related to the interconnection of components. To show its generality, these operations will be used to define a platform-specific fault injection approach to enable the injection of upsets into the configuration memory of look-up tables.

The combinational logic in Field-Programmable Gate Arrays (FPGAs) is mostly implemented by means of Look-Up Tables (LUTs). Accordingly, arithmetic and boolean expressions of arbitrary complexity at RTL are mapped onto a set of interconnected LUTs at the implementation-level. As depicted in Fig. 4.4, each LUT consists of a tree of multiplexers controlled by the input address, which selects the output from the configuration memory cells.

From a robustness assessment perspective, this means that only input/output signals of arithmetic/boolean expressions are available for fault injection at RTL, whereas input/output ports of all LUTs can also be targeted by faults at the implementation level. This can be accomplished by the previously presented injection procedure for stuck-at, pulse, and indetermination faults. However, by considering the particular implementation of each macrocell, it is also possible to define specific fault injection approaches like, for instance, to study the sensitivity of configuration memory cells to bit-flip faults.

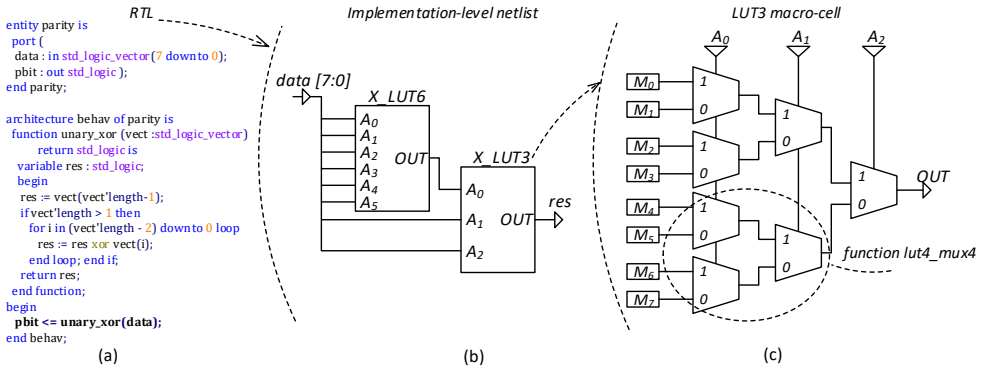


Figure 4.4: Combinational logic: a) described at RTL, b) implemented by LUTs, and c) macrocell's internal structure

The logic function (truth table) implemented by the *X_LUT6* macrocell, from Xilinx's SIMPRIM library [174], is defined as a generic parameter (*INIT*). This parameter initialises an internal constant named *INIT_reg*, which represents the memory cells of the macrocell. This constant is used in the *Functionality* section of the *VITALBehavior* process to compute the expected output of the macrocell. Accordingly, the macrocell must be instrumented to use a signal instead of a constant and activate the process whenever this signal changes. The required instrumentation is deployed by the following operations: *constant2signal(INIT_reg); addToList(VITALBehavior, v_INIT_reg)*. After that, any bit of this truth table can be modified. For instance, a bit-flip targeting the bit 0 can be injected by means of *value=examine(v_INIT_reg); force(v_INIT_reg, freeze, value xor 0x00000001, 0)*.

4.3 Fault simulation in Verilog-based models

Verilog HDL provides embedded features for the definition of high-performance and portable macrocell libraries, such as logic primitives and timing *specify* blocks. Verilog-based macrocells, though not restricted in their structure, may face fault injection problems similar to those of VITAL-compliant libraries. The injection of faults into macrocell nodes modelled by constants (parameters) can follow the same procedure as defined for VITAL (*constant2signal* and *addToList* operations). However the injection of bit-flips and timing faults requires to take into account some Verilog-specific model aspects.

4.3.1 Bit-flip faults

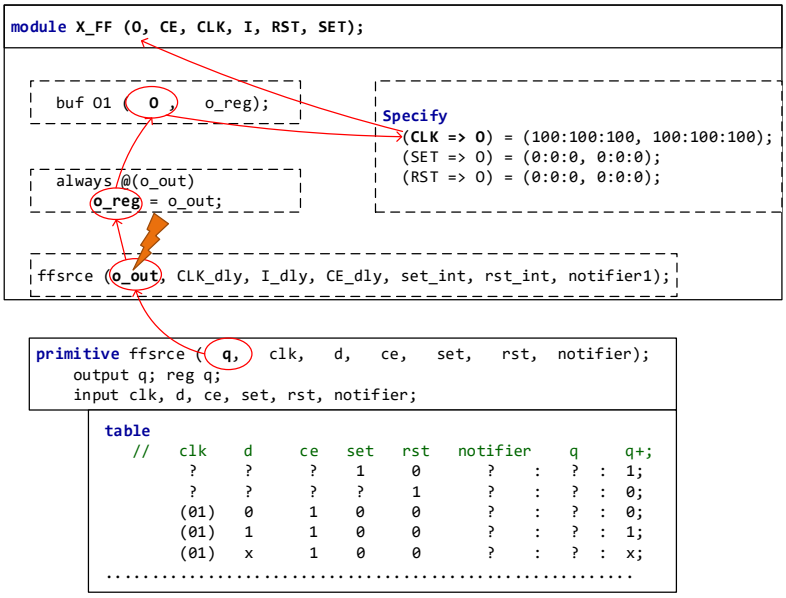


Figure 4.5: Structure of Verilog-based Flip-Flop macrocell (Xilinx simprim library)

Verilog User-Defined Primitives (UDPs) are commonly used instruments for the definition of sequential and combinational macrocells. On the one hand, a UDP describes the macrocell logic in the form of truth tables, which are quickly evaluated at runtime, thus providing very high simulation performance. On the other hand, UPD's internal scope is hidden in simulation, so it cannot be traced nor altered by simulator commands. Fig.4.5 illustrates the structure of the Verilog-based X_FF macrocell from the Xilinx simprim library. It can be seen that the FF state is recomputed by the *ffsrce* primitive and stored in *o_reg* node, passing through an intermediate signal *o_out*. The recomputed value is driven to the output through the buffer primitive, being first delayed in the specify section according to the timing parameters annotated from SDF.

To simulate a bit-flip the o_out signal could be toggled, as it is the source node in this chain, using simulator commands. After flipping the *o_out*, it is expected to be recomputed (recovered) on the following rising clock edge by the edge sensitive truth table within the *ffsrce* primitive. However, as it can be seen from simulation results in ModelSim, the expected recovery does not take place - Fig.4.6a. The FF is only recovered when its input changes. This behaviour has been observed

under all ModelSim versions (10.3 to 10.7), and independently from compilation options.

The possible explanation of this behaviour is that UDP does not update the output when the newly computed value remains the same as the previous one. Indeed, the *reg q* node (within *ffsrce*) equals '1' both before and after the rising clock edge. The event scheduling mechanism re-evaluates the driven signal in response to events on the driver. Thus, the signal *o_out* will not be rescheduled in the absence of switching events on *q*. Accordingly, the actual node that should be targeted at fault injection is the *q* signal. However, it pertains to the UPD's internal scope, which is not accessible for simulator commands.

Listing 4.4: Proposed procedure to inject a bit-flip into Verilog *target* macrocell

```
1  bitflip(target) {
2     newValue = (examine(target/O_out) == 0) ? 1 : 0;
3     force(target/O_out, deposit, newValue, 0);
4     when {expected_update_event(target/O_out)} {
5                   noforce target/O_out;
6           }
7  }
```

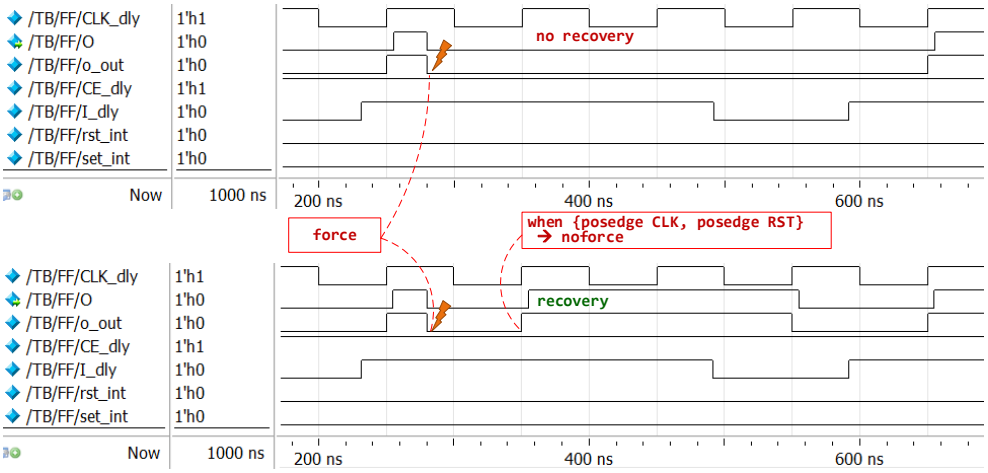


Figure 4.6: Simulation of bit-flip fault in Verilog macrocell: (a) RTL procedure, (b) implementation-level (resulting) procedure

To overcome this problem and mimic the expected bit-flip effect, the *o_out* can be forcibly recovered by means of simulator commands, as it is shown in Listing 4.4. It relies on *when* clause to detect the recovery time instant (clock edge or reset), and on *noforce* command to recover the targeted signal to its original state (the

one it had before executing the *force* command). The resulting behaviour is illustrated in Fig.4.6b. It can be seen that the state of the FF, after being flipped, is properly recovered at the next rising clock edge.

4.3.2 Delay faults

Timing delays are modelled in Verilog macrocells by means the of so-called *specify* section, which defines a set of propagation paths and timing checks. The nominal delays in each path can be defined by literals or by specify parameters. At back-annotation these nominal delays are replaced by actual values from SDF, using port names in each path as a key for path lookup. Thus the annotation of SDF timing does not require an explicit declaration of specify parameters. Even if these parameters are declared, they are not accessible (hidden) in simulation. This implicit back-annotation mechanism prevents the usage of the same procedure for injecting timing faults as in the VITAL-compliant macrocells. Delays in Verilog models can be changed at runtime only by annotating a modified SDF file by means of the Verilog function *sdf_annotate(sdf_file, [scope])*.

The annotation of a modified SDF for the complete design may introduce a significant time overhead during simulation. Fortunately, Verilog allows annotating timing properties in a partial way, i.e. from separate SDF files with the granularity of individual macrocells. Moreover, the SDF annotation function can be invoked directly at simulation time by means of a ModelSim command *call $sdf_annotate {sdf_file}*.

Thus, timing faults can be injected into Verilog macrocells according to the following procedure. First, the model is annotated with the reference SDF file and simulated until the fault injection time - Fig.4.7A. At this point, a partial SDF file is created, which specifies the increased delays for the targeted macrocell. This partial SDF is annotated by a simulator command, as depicted in Fig.4.7B, and the simulation continues until the end of the experiment of until the recovery time. Macrocell delays can be recovered by following the same partial SDF approach.

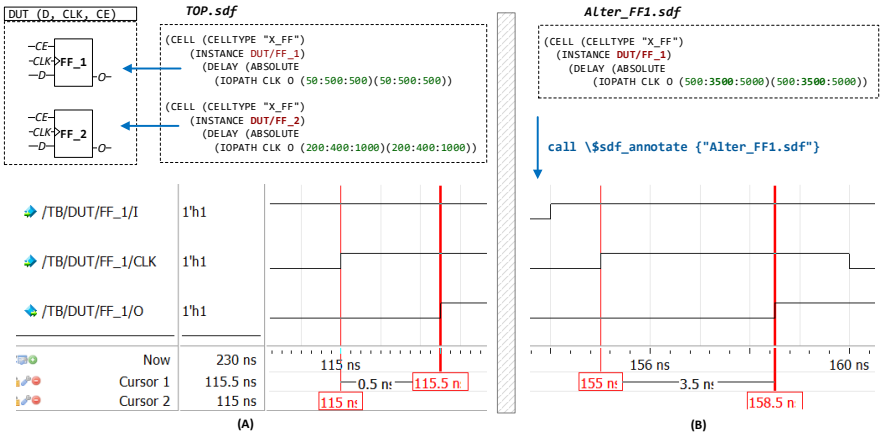


Figure 4.7: Simulation of timing faults in Verilog macrocells: (A) DUT annotated with reference complete SDF, (B) annotation of partial SDF (for the selected macrocell) with increased delays at run-time

4.4 Unified fault dictionary

As fault injection procedures at RTL are uniform (same sequence of simulator commands applied to each target signal), these procedures are usually defined directly in the SBFI tool. However, this approach is rather complex and inefficient at the implementation level, since each macrocell requires its own fault injection procedure for each fault model, according to its internal structure and semantics.

As VITAL-compliant macrocells share the same architecture and naming conventions, this can be exploited to automatically build custom fault injection scripts at runtime. These scripts should define a set of rules to instrument the selected macrocells and generate the proper sequence of simulator commands to inject a given fault. A fault dictionary format illustrated in Fig 4.8, allows to unify the definition of fault models and decouple them from the code of SBFI tools. The fault dictionary model comprises a set of fault descriptors that aggregate instrumentation and injection rules that can be applied to a set of macrocells in a uniform way.

First, for each type of macrocell it should be checked whether it satisfies the instrumentation rules defined in the fault descriptor. If any rule is missing, it must be applied and the macrocell recompiled. Logic fault models studied so far have led to the definition of four different basic instrumentation rules: i) *Generic2Signal*, which defines supplementary signals for generic parameters to be

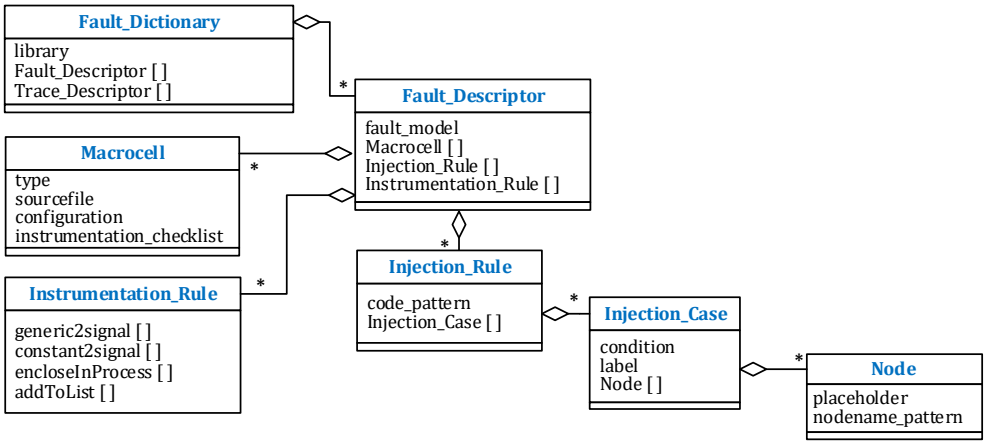


Figure 4.8: Fault dictionary model

injectable and appends the required initialization code after back-annotation, ii) *Constant2Signal*, which redefines the constant as a signal, iii) *EncloseInProcess*, which forces timing routines to capture the newly injected timing values by enclosing them in a process, and iv) *AddToList*, which makes a process sensitive to changes of signals.

Once all instrumentation rules are satisfied, injection rules can be applied. Each injection rule defines a parametrized pattern of simulator commands that is applied to the set of injection cases. Each injection case represents a particular injection point within a macrocell (I/O ports, internal registers, propagation paths, etc.), an optional *iterator* attribute to handle vectors and arrays, and an optional *condition* for the fault to be injected.

Macrocells with similar architecture may share the same fault descriptor in some cases. For instance, bit-flip injection rules are identical for different Xilinx Flip-Flops (*X_FF*, *X_SFF*, and *X_FDD* macrocells), so just one fault descriptor is required. However, the delay fault model requires to define individual descriptors for each macrocell due to a different set of inputs and propagation paths. By following this specification model, it is possible to define fault models in a relatively compact way for huge libraries and complex macrocells (containing thousands of internal injection points).

The parametrization of code patterns and injection cases enables fine-tuning fault-loads from a single configuration file. Fig. 4.9 illustrates an example of an XML file describing the procedure required to inject delay faults into the *X_FF* macro-

Figure 4.9: Excerpt from the fault dictionary file and configuration file describing the delay fault model for a Xilinx's X_FF macrocells and a delay faultload, respectively

cells. By following this specification, the SBFI tool first checks the instrumentation rules, particularly that timing generics are injectable and that delay processes are sensitive to the change of timing parameters. After that, injection scripts are generated from the injection rule by applying the code pattern to each injection case within the macrocell (different ports and propagation paths).

The faultload is also defined by means of another XML configuration file. The *<fault_model>* section defines a number of attributes that the *faultload builder* shall use to generate the injection scripts. The *model* and *target_logic* attributes state which fault description rules will be used for generating these scripts. Likewise, the *condition* attribute determines which injection cases will be applied. Finally, the rest of the attributes determine the values for specific placeholders, like forced value or fault duration. The condition attribute allows to filter the injection cases for fine-tuning the faultload. For instance, the faultload depicted in Fig. 4.9 will generate the scripts required to inject interconnect delay faults

(not propagation delays) into all the X_FF, X_SFF, X_LUT5, and X_LUT6 components of the implemented design.

4.5 Conclusions

VITAL and Verilog standards establish a comprehensive set of rules to unify the design of macrocells libraries and EDA tools, enabling the implementation of efficient optimizations for simulation speed-up. However, their rigorous requirements make that common SBFI techniques cannot be easily applied to implementation level HDL models defined on the basis of such macrocells libraries.

This chapter has carefully studied the architecture of VITAL-compliant macrocells to define generic operations that can be deployed to enable the injection of most common logic faults into implementation-level models. Sequences of generic simulator commands have been defined to conduct the injection of faults whenever possible and to reduce to the minimum the intrusion and the overhead in the simulation time. However, some faults can only be injected after instrumenting the target macrocells. In such cases, the defined operations keep the functionality and timing behaviour of the target macrocell while following VITAL requirements. Only in the case of interconnection path delays, the VITAL level of support will be degraded from 1 to 0. By instrumenting the macrocells, the original implementation-level model and the VITAL libraries are not modified by any means, reducing the intrusiveness of the proposed approach. Likewise, once the macrocell is instrumented and recomplied, no further recompilations are required, thus reducing also the experimental overhead with respect to other common approaches. The defined operations are generic enough to be technology independent, so they can be applied to the VITAL-compliant macrocells of any vendor and the commands can be supported by most common industry standard simulators.

After that, it has been shown that despite Verilog-based models face similar fault injection problems as the VITAL-based ones, they require somewhat different fault injection procedures. These procedures are completely non-intrusive, relying solely on the use of simulator commands and native Verilog functions.

Finally, the definition of fault injection procedures for diverse macrocells and fault models has been unified by means of a new tool-independent fault dictionary format. Each fault descriptor in such dictionary defines a set of instrumentation and injection rules, that should be followed by any third-party fault injector, to properly simulate any given fault within any given macrocell.

Chapter 5

Improving the Accuracy of FPGA-based Fault Injection

This chapter proposes an approach for bit-accurate FPGA-based fault injection (FFI). Section 5.1 describes the problems that should be addressed to deploy FFI experiments with a bit-accuracy. Section 5.2 describes the algorithms for bit-accurate location of LUT and BRAM content within the configuration memory of FPGAs. Section 5.3 describes how the bit-accurate mapping is used to generate an optimized essential bits file that allows FFI tools to locate the relevant fault targets (CM cells) within any given design scope. Section 5.4 describes an FFI flow that relies on optimized essential bits, and takes into account the kind of logic primitive configured by the targeted CM cells, in order to properly emulate upsets in registers, changeable CM, and the non-changeable CM. Section 5.5 concludes this chapter.

5.1 Introduction

Single Event Upsets (SEU) in configuration memory (CM) pose one of the primary threats for the dependability of FPGA-based designs. Designers thus should carefully evaluate the robustness of such designs against SEUs and analyse the efficiency of integrated SEU mitigation mechanisms. FPGA-based fault injection (FFI) is one of the main instruments allowing this kind of analysis.

An important problem that should be addressed when deploying FFI experiments is to accurately locate the relevant fault targets within the CM, i.e. those CM cells that configure the circuit and its constituent nodes (macrocells). The finer is the granularity with which design nodes are related to the underlying CM cells, the more detailed robustness estimates can be derived from FFI experiments and the more time-efficient such experiments become. For instance, the most straightforward FFI strategy, which blindly targets all CM cells in FPGAs, is able to quantify only the robustness of the design as a whole, without any insights on the robustness of its individual modules. In addition it leads to lots of useless FFI runs, since even complex FPGA designs are very unlikely to use all available FPGA resources.

A known approach to refine the location of relevant fault targets (CM cells) is to rely on the Xilinx *essential bits* information. It provides a bitmask file (EBC file generated by Vivado design suite) that highlights those CM cells that actually configure the circuitry in FPGAs. On the one hand, it significantly reduces the number of useless fault injections, since non-essential bits do not impact in any way the behaviour nor the integrity of the circuit and, thus, can be safely omitted from FFI experiments.

On the other hand, as it has been discussed in Section 3.4.2, Xilinx essential bits has several limitations with respect to FFI. First of all, it considers only non-changeable CM cells, leaving registers, LUTRAM and BRAM content out of consideration. Second, the essential bit mask file does not relate CM cells to individual design units and/or types of netlist primitives (macrocells). This again leads to rather blind FFI experiments (although with a reduced injection scope) that estimate the robustness of the implementation as a whole but do not allow to locate the weak points of the design. Third, essential bits reported by Xilinx may be redundant for partially used LUTs (as it will be shown later in this chapter). Therefore, many FFI runs still could remain ineffective, wasting experimental time and affecting the accuracy of derived robustness estimates. In addition to that, this mask file is not self-descriptive, i.e. to make use of it, it should be first mapped onto the configuration memory of the target FPGA. This complicates its practical application in FFI.

Another useful aid provided by Xilinx for the location of CM cells is the logic allocation (LL) file. This file only considers changeable CM cells (FFs, LUTRAM, and BRAM), so it can be potentially combined with an EBC file to cover the entire set of CM cells that are essential for the design. However, as it has been discussed in Section 3.4.2, LL file reports the BRAM-specific and LUTRAM-specific CM cells in an obfuscated and redundant way, which does not relate the CM cells with the hierarchy of the DUT and leads to considering unused CM bits.

This chapter describes a bit-accurate FFI approach that addresses the aforementioned problems. First, Section 5.2 studies the bit-accurate mapping between some major types of netlist primitives and the CM, and proposes algorithms for identifying LUT-specific and BRAM-specific essential bits within the CM. On the basis of bit-accurate mapping, Section 5.3 then defines a custom essential bit mask file that allows FFI tools to selectively target any given design unit and any type of netlist primitives among those that can be mapped with a bit-accuracy. So as to consider the whole set of CM bits (not only those related to the mapped primitives) it also incorporates the information from Xilinx essential bits file, resulting in an *optimized essential bits* file that eliminates the redundancy existing in the EBC file and also takes into consideration changeable CM cells. Finally, Section 5.4 describes an FFI flow that relies on the proposed optimized essential bits file and takes into account the type of logic primitive behind the targeted CM bits in order to properly emulate upsets in registers, changeable memories (BRAM/LUTRAM), and in the non-changeable CM with a bit-accuracy.

5.2 Towards bit-accurate mapping of macrocells onto the configuration memory

The objective of CM mapping is to establish a relationship between the netlist primitives (macrocells) and the CM cells that configure these macrocells. FPGA-based netlists comprise a wide variety of macrocells. This makes their complete mapping onto the configuration memory quite a challenging problem. Nevertheless, if prioritizing the different macrocells according to their relative weights in the netlists, it can be concluded that three components are of the utmost interest for FFI experiments: (i) LUTs as the main building block of combinational logic, (ii) CLB Flip-Flops implementing the DUT registers, and (iii) BRAMs that implement on-chip user memories as well as the control memories of inferred FSM.

The mapping of Flip-Flops can be extracted from the logic allocation (LL) file generated by Vivado. The mapping of BRAMs is also listed in the LL file, but in an obfuscated and redundant way that does not provide any clear relation with

the source (RTL) design nodes and often leads to considering unused (dummy) BRAM bits. The mapping of LUTs is not reported by Xilinx tools in any way. Likewise, the mapping of non-changeable CM in general is not reported by Xilinx tools.

It should be noted that routing resources, represented by programmable interconnection points (PiPs) within the switchboxes are not reflected at the netlist level (implementation-level HDL model). Hence, the mapping of routing resources is not considered in this work.

This section, thus, studies how the LUT and BRAM macrocells can be mapped onto the CM with a bit-accuracy and how to determine which of their corresponding CM cells are actually essential.

It is worth noting that this section operates by such terms as *macrocell*, *BEL*, and *Slice*. Term macrocell is used to refer to the logic primitives constituting the netlist. For instance, the netlists generated by Vivado suite for Xilinx 7-series FPGAs may include six types of LUT macrocells, attending to the number their inputs: *LUT_1* to *LUT_6*. The term BEL refers to the basic element of logic, into which one or more macrocells are placed. A group of BELs is referred to as Slice. For instance, Xilinx 7-series FPGAs have two types of CLB Slices – Type L, and Type M – the former implements only the combinational logic, whereas the latter also supports the implementation of distributed memories on LUTs. Each CLB slice includes four pairs of LUT BELs, labelled *A5/A6, B5/B6, C5/C6, D5/D6*. Each pair of LUT BELs supports the placement of two LUT macrocells (when the LUT combining is enabled). Refer to the Section 2.1.2 for further details regarding the terminology used in Xilinx design flow.

5.2.1 Mapping of Look-Up tables

Mapping LUT cells onto the configuration memory comprises coarse-grained and fine-grained levels, illustrated in Fig.5.1. The former locates the bitstream fragment (BF) that corresponds to the placement of a LUT macrocell. The latter locates each individual bit of a LUT's truth table (INIT attribute) within the BF. The global bit-accurate LUT mapping is derived by combining these two mapping levels, so each LUT bit is related to the coordinates of corresponding CM cells (designated by the tuple *Frame, word, bit*).

The bitstream fragment, located by the coarse-grained mapping, comprises four half-words from four consecutive frames designated by the structure *(Block, Top, Row, Major, Minor, word, bits)* - as depicted in Fig.5.1. These coordinates are calculated from the LUT placement on the design layout (LUT BEL). The LUT

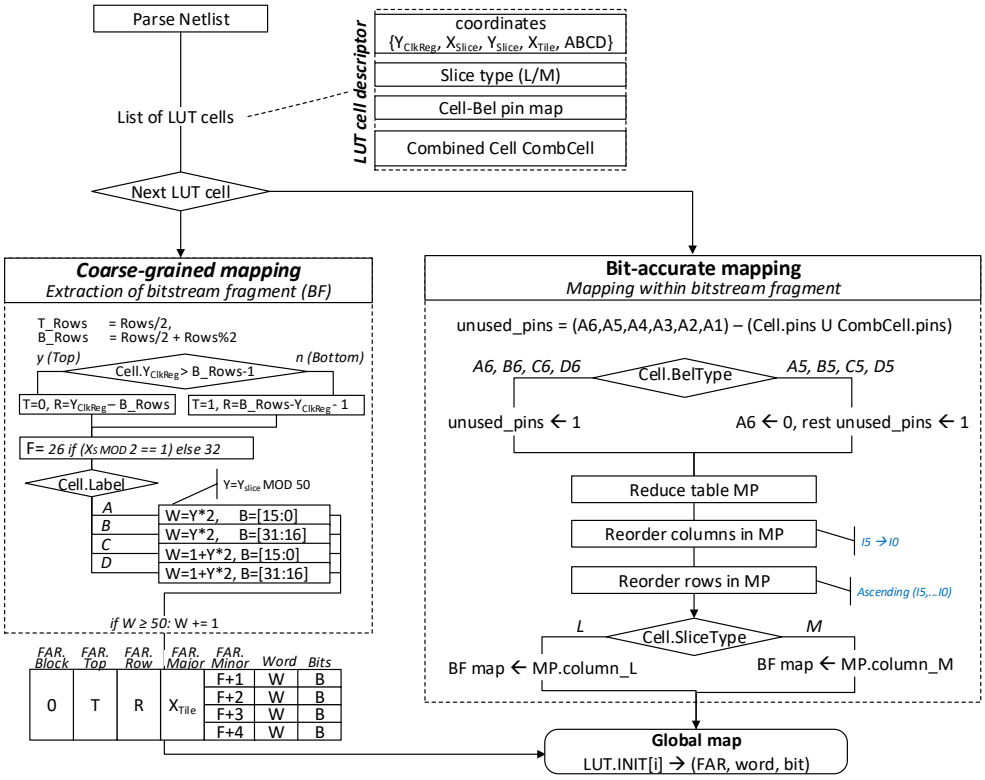


Figure 5.1: Algorithm for the bit-accurate mapping of LUT cells onto the configuration memory (bitstream)

content is located in CM frames of Type-0. The top/bottom part T and the clock row R are calculated from the vertical coordinate of the clock region Y_{ClkReg}. The major frame coincides with the index of the CLB column, i.e. X_{Tile} coordinate. The four minor frames are 26–29 for the odd X_{Slice}, and 32–35 for the even X_{Slice}. Finally, the word index and its part (high or low 16 bits) are calculated from the vertical slice coordinate Y_{Slice} and from the LUT label within the slice (A,B,C,D), according to the coarse-grained LUT mapping depicted in Fig.5.1.

The bits of the extracted BF are scrambled, so they should be properly reordered to obtain the INIT (truth table) of the LUT macrocell. The proper order of BF bits for the complete (6-input) LUT has been derived experimentally according to the procedure detailed in Section A.2. Table 5.1 lists the resulting order of BF bits in which they should be concatenated to obtain the INIT of a LUT6 macrocell

under the direct mapping of LUT inputs to BEL pins (*I0:A1, I1:A2, I2:A3, I3:A4, I4:A5, I5:A6*). This order of bits is further referred to as LUT BEL mapping, designated as table MP in Fig.5.1. Experiments detailed in Section A.2 have shown that LUTs located in the CLB slices of type L and M have different CM mapping. For that reason Table 5.1 has two different mapping columns (column-L and column-M respectively). At the same time, experiments have also shown that this mapping does not depend on the BEL location. In other words, Table 5.1 is valid for any clock region, CLB tile coordinates, and LUT label within the CLB slice (A,B,C,D). Fig.5.2 illustrates the derived bit-accurate mapping of LUT BELs onto the bistream for both types of slices (L and M).

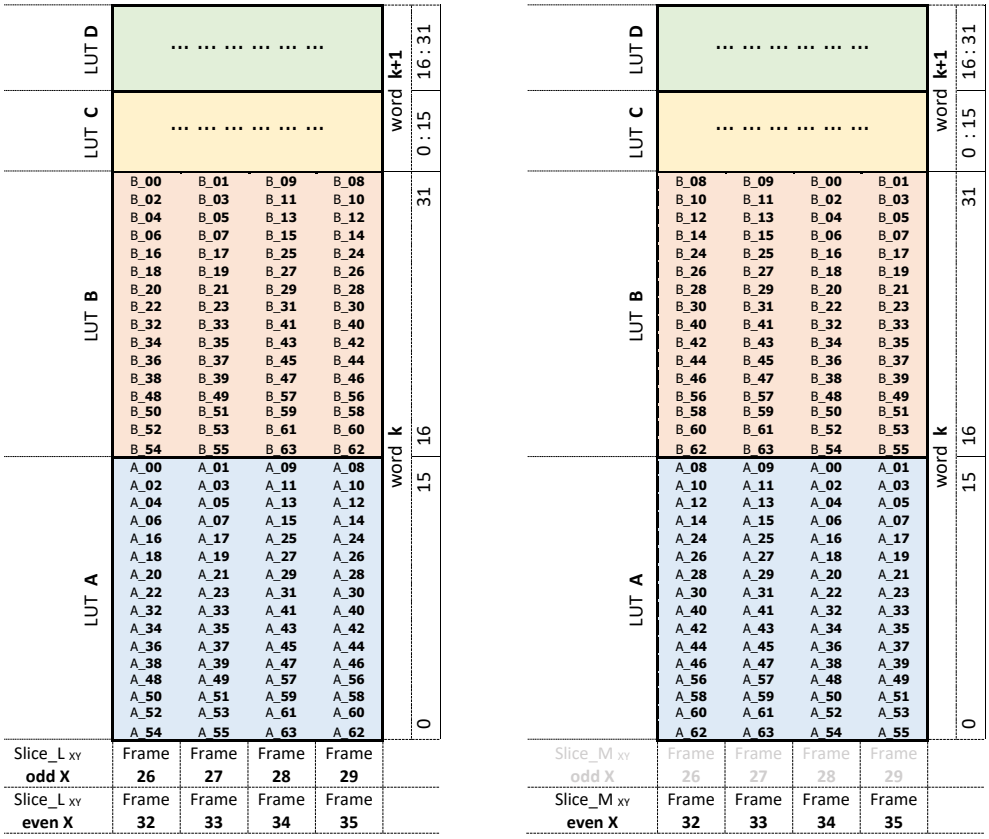


Figure 5.2: Location of LUT content within the configuration memory of 7-series FPGA

Table 5.1: Bit-accurate mapping of LUT6 content onto the bitstream fragment under the direct assignment of LUT inputs to the BEL pins (experimentally obtained for Xilinx 7-series FPGA)

LUT BEL inputs						LUT6.INIT bit_index	Matching bit of bitstream fragment	
A6	A5	A4	A3	A2	A1		Slice L	Slice M
0	0	0	0	0	0	0	**63**	**31**
0	0	0	0	0	1	1	**47**	**15**
0	0	0	0	1	0	2	**62**	**30**
0	0	0	0	1	1	3	**46**	**14**
0	0	0	1	0	0	4	**61**	**29**
0	0	0	1	0	1	5	**45**	**13**
0	0	0	1	1	0	6	**60**	**28**
0	0	0	1	1	1	7	**44**	**12**
0	0	1	0	0	0	8	**15**	**63**
0	0	1	0	0	1	9	**31**	**47**
0	0	1	0	1	0	10	**14**	**62**
0	0	1	0	1	1	11	**30**	**46**
0	0	1	1	0	0	12	**13**	**61**
0	0	1	1	0	1	13	**29**	**45**
0	0	1	1	1	0	14	**12**	**60**
0	0	1	1	1	1	15	**28**	**44**
0	1	0	0	0	0	16	**59**	**27**
0	1	0	0	0	1	17	**43**	**11**
0	1	0	0	1	0	18	**58**	**26**
0	1	0	0	1	1	19	**42**	**10**
0	1	0	1	0	0	20	**57**	**25**
0	1	0	1	0	1	21	**41**	**9**
0	1	0	1	1	0	22	**56**	**24**
0	1	0	1	1	1	23	**40**	**8**
0	1	1	0	0	0	24	**11**	**59**
0	1	1	0	0	1	25	**27**	**43**
0	1	1	0	1	0	26	**10**	**58**
0	1	1	0	1	1	27	**26**	**42**
0	1	1	1	0	0	28	**9**	**57**
0	1	1	1	0	1	29	**25**	**41**
0	1	1	1	1	0	30	**8**	**56**
0	1	1	1	1	1	31	**24**	**40**
1	0	0	0	0	0	32	**55**	**23**
1	0	0	0	0	1	33	**39**	**7**
1	0	0	0	1	0	34	**54**	**22**
1	0	0	0	1	1	35	**38**	**6**
1	0	0	1	0	0	36	**53**	**21**
1	0	0	1	0	1	37	**37**	**5**
1	0	0	1	1	0	38	**52**	**20**
1	0	0	1	1	1	39	**36**	**4**
1	0	1	0	0	0	40	**7**	**55**
1	0	1	0	0	1	41	**23**	**39**
1	0	1	0	1	0	42	**6**	**54**
1	0	1	0	1	1	43	**22**	**38**
1	0	1	1	0	0	44	**5**	**53**
1	0	1	1	0	1	45	**21**	**37**
1	0	1	1	1	0	46	**4**	**52**
1	0	1	1	1	1	47	**20**	**36**
1	1	0	0	0	0	48	**51**	**19**
1	1	0	0	0	1	49	**35**	**3**
1	1	0	0	1	0	50	**50**	**18**
1	1	0	0	1	1	51	**34**	**2**
1	1	0	1	0	0	52	**49**	**17**
1	1	0	1	0	1	53	**33**	**1**
1	1	0	1	1	0	54	**48**	**16**
1	1	0	1	1	1	55	**32**	**0**
1	1	1	0	0	0	56	**3**	**51**
1	1	1	0	0	1	57	**19**	**35**
1	1	1	0	1	0	58	**2**	**50**
1	1	1	0	1	1	59	**18**	**34**
1	1	1	1	0	0	60	**1**	**49**
1	1	1	1	0	1	61	**17**	**33**
1	1	1	1	1	0	62	**0**	**48**
1	1	1	1	1	1	63	**16**	**32**

In practice, however, LUTs are not constrained to the direct mapping of LUT inputs to LUT BEL pins, so that Xilinx Vivado suite can optimize it in such a way as to reduce the critical path. In fact, there exist 720 possible pin mappings for the LUT6 macrocell (number of permutations of LUT inputs). Furthermore,

LUT macrocells may have less than 6 inputs, leaving some BEL pins unused. Finally, in the case of LUT combining, some BEL pins can be used by the adjacent (combined) LUT macrocell. All these factors must be taken into account to properly locate the LUT content within the bitstream fragment BF.

It is important to note that, in the case of LUT combining, the A_6 Bel pin is explicitly driven to logic '1', thus splitting the 64 LUT bits into two independent parts: the top 32 bits determine the O_6 output and the bottom 32 bits determine the O_5 output. However, the state of unused BEL pins is not specified explicitly. LUT mapping experiments in Section A.2 have shown that, in the case of underutilized LUTs (LUT macrocells with less that 6 inputs), each of its bits maps onto several bits of the extracted bitstream fragment BF. The multiplicity of this mapping equals 2^N, where N is the number of unused BEL pins. For instance, each bit of a *LUT_4* macrocell maps onto four CM cells, as it is shown in Section A.2. In fact, the LUT content is replicated in the bitstream in such a way as to make the LUT output independent of unused BEL pins, i.e. unused pins are conservatively assumed do not care levels. However, BEL pins in FPGAs are expected to be driven to a certain predetermined logic level. By following the experimental procedure in Section A.3 it has been found that unused pins of LUT BELs are actually driven to the logic '1'.

The fine-grained mapping algorithm, depicted in Fig.5.1, takes into account all the aforementioned considerations. For each LUT macrocell in the netlist it determines the location of its INIT bits within the BF by consecutively applying the *reduce*, *reorder columns*, and *reorder rows* operations to the LUT mapping table MP. The *reduce* operation first renames the columns labelled by BEL pins (A) according to the inputs (I) of the mapped LUT macrocell. Unused BEL pins are assumed a constant '1', except for A_6, which equals '0' when the LUT output is mapped into the output O_5. Afterwards, all the entries in the table MP that do not match this assumption are filtered-out. Likewise, the columns corresponding to the unused pins are left out of consideration. In the reduced table MP the columns I are reordered by descending indexes. Afterwards, the rows are reordered by ascending values of LUT input vector I. As a result, the columns L and M (corresponding to the Slices of type L and M) in the table MP contain the bit indexes within the BF that directly correspond to the bits of the mapped LUT macrocells in ascending order. Finally, the determined BF indexes are used to look-up the global coordinates of CM cells for each bit of the mapped LUT macrocell.

LUT descriptors, supplied to the mapping procedure, can be extracted from the netlist in Vivado by means of the TCL script provided in listing A.2 (Annex-A.4). It extracts the placement of each LUT macrocell in the netlist, i.e. XY

coordinates of the corresponding slice, tile, and clock region, as well as the type (L or M) and the label (A/B/C/D) of LUT BEL. For the given macrocell placement it queries a name (path in the design hierarchy) of an adjacent LUT macrocell (if any) that is combined with a mapped LUT within the LUT5/LUT6 BEL pair. Finally, for each input of the LUT macrocell it queries an assigned BEL pin ($CellBelPinMap$ attribute). The collected LUT descriptors are exported into a csv-formatted table - one row per LUT cell. Fig.5.3 illustrates an example of resulting LUT descriptors. The cell path uniquely identifies the LUT macrocell within the netlist (design hierarchy). In the case of LUT combining, the $CombLut$ attribute links the combined macrocells by their $CellPath$ attribute.

It is worth noting that some LUT BELs can be utilized without any placed LUT cell, being a part of net routing (pass-through LUTs) or driving a constant value to the LUT output. Despite these LUT BELs are not reflected in the netlist (not mapped), they utilize the BEL pins and, thus, impact the mapping of the LUT macrocells when they are combined with such constant/pass-through LUTs within the LUT5/LUT6 BEL pair. The pass-through and constant LUT BELs are marked as unused in Vivado. The former has an output pin driven by some of its input pins, whereas the latter has an empty pin list, but it has an assigned equation $O6/O5 = 1/0$. Nevertheless, all CM cells corresponding to BF of such LUTs are marked by Vivado as essential in the EBC file. Hence, the descriptors exported for such LUTs by the script in listing A.2 do not include any macrocell-related properties, but only indicate the BEL coordinates and the utilized BEL pins.

CellPath	CellType	SliceXY	TileXY	ClkRegXY	Slice.BelLabel	CellINIT	CombLut	CellBelPinmap
.../mul_o1	LUT3	Slice_X45Y104	CLBL_X30Y104	X0Y2	SliceL.A6LUT	8'h...		{O:O6}{I0:A3}{I1:A2}{I2:A4}
.../add_o9	LUT2	Slice_X45Y102	CLBL_X30Y102	X0Y2	SliceL.C6LUT	4'h...	.../sub_o9	{O:O6}{I0:A3}{I1:A2}
...	...	...	...	...	...	...	...	...
.../sub_o9	LUT2	Slice_X45Y102	CLBL_X30Y102	X0Y2	SliceL.C5LUT	4'h...	.../add_o9	{O:O5}{I0:A4}{I1:A5}

Figure 5.3: Example LUT descriptors extracted from the netlist in Vivado

Fig.5.4 illustrates an example of mapping a LUT3 macrocell by the proposed algorithm. The LUT descriptor on the input (*mul_o1* in Fig.5.3) has the following attributes: slice coordinates $X_{45}Y_{104}$, tile coordinates $X_{30}Y_{104}$, BEL type L, BEL label $A6$, pin mapping $I_0 : A_3, I_1 : A_2, I_2 : A_4$, and vertical coordinate of the clock region Y_2.

First, the bitstream fragment (BF) is located by means of the coarse-grained mapping. The given FPGA (XC7Z020) has three clock rows: one in its top part ($T\_Rows = 1$) and two in the bottom part ($B\_Rows = 2$). The clock region coordinate Y_2 corresponds to the first row of the top part $T = 0, R = 0$, as

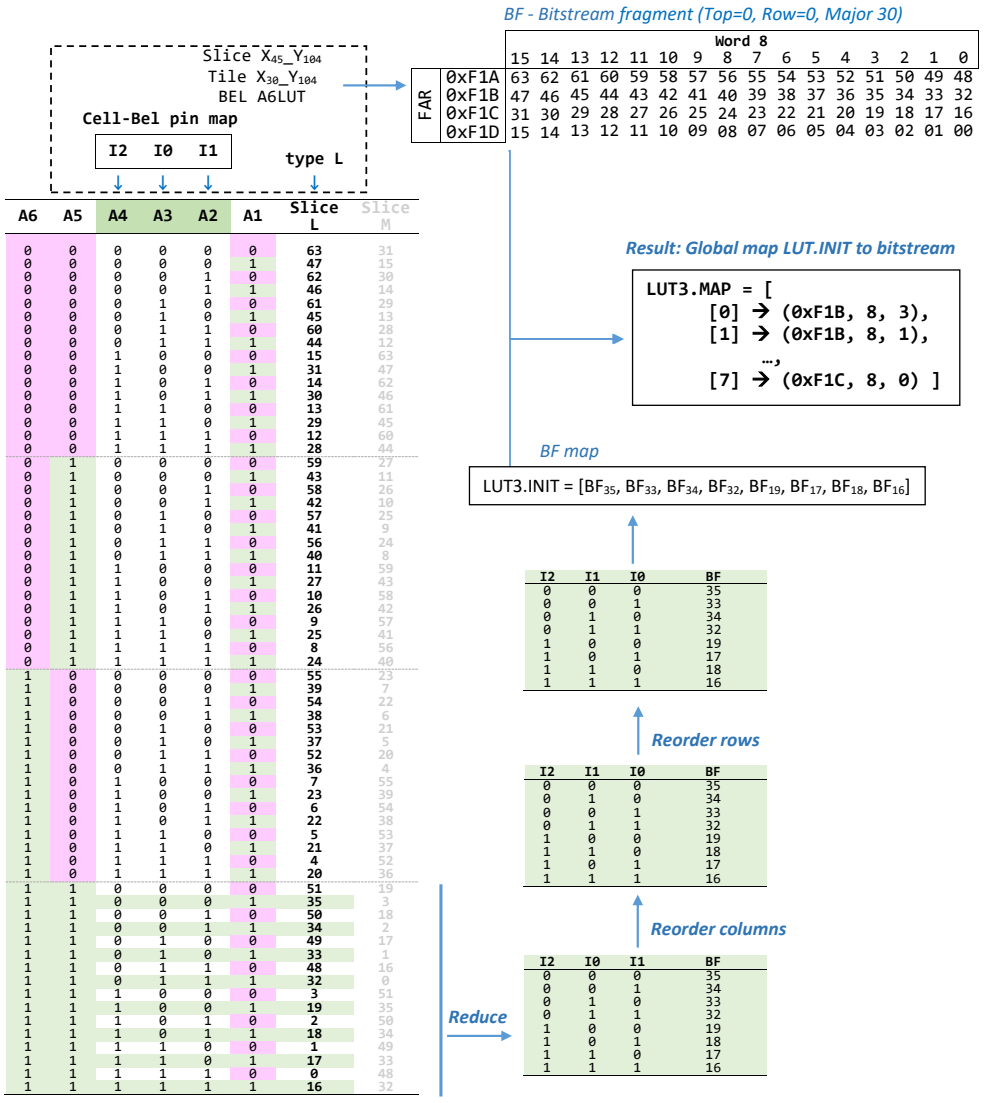


Figure 5.4: Example of bit-accurate LUT mapping

calculated by the expressions in Fig.5.1. The major frame equals the horizontal tile coordinate ($X_{Tile} = 30$). The four minor frames are $26-29$, since it is an odd CLB column ($SliceX = 45$). Finally, the word corresponding to the LUT with label A is computed as $Y \times 2 = 8$ (where $Y = Y_{slice} mod 50 = 4$), bits $[15:0]$. The

resulting bitstream fragment is encoded as *FAR=[0xF1A,0xF1B,0xF1C,0xF1D], word=8, bits=[15:0].*

Second, the fine-grained mapping locates the LUT bits within the BF. The A columns of the table MP are renamed according to the assigned inputs of the mapped LUT: $A_4 \rightarrow I_2, A_3 \rightarrow I_0, A_2 \rightarrow I_1$. Since the BEL $A6$ uses the output $O6$ and there are no combined LUTs, the rest of unused BEL pins are assumed a value of '1'. The table MP is now reduced by filtering-out those rows which does not match this assumption. The columns are reordered by descending indexes of used LUT inputs (I_2, I_1, I_0). Afterwards, the rows are reordered by ascending values of LUT input vector $(I_2 I_1 I_0 = 000 \rightarrow 111)$. The L column of the resulting table now contains the indexes of BF bits, directly corresponding to the INIT bits of the mapped LUT macrocell – *BF map* depicted in Fig.5.4. Finally, this *BF map* is used to look-up the global CM coordinates of each LUT bit in the BF, e.g. $INIT_0 \rightarrow BF_{35} \rightarrow (0xF1B, 8, 3)$, etc.

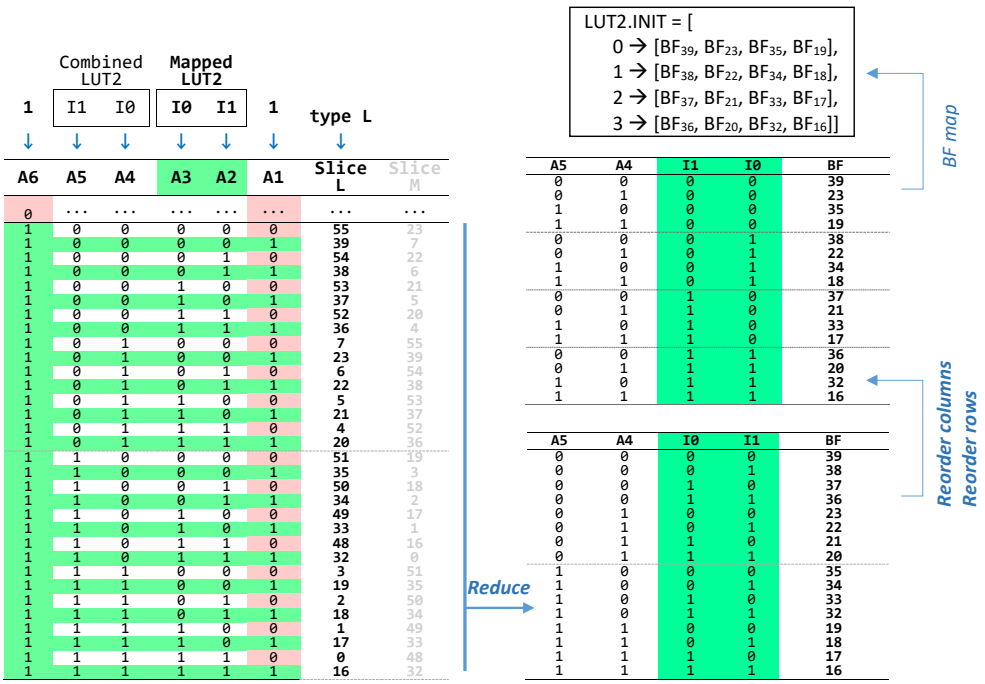


Figure 5.5: Mapping of LUT content in case of LUT combining

It is worth noting that in the case of LUT combining, all BEL pins exclusively used by an adjacent LUT macrocell should be assumed do not care levels. Fig.5.5

illustrates an example of mapping a LUT2 macrocell which is combined with another LUT2 macrocell within a single LUT6/LUT5 BEL pair (descriptor *add_o*9 in Fig.5.3). The mapped LUT is placed into the LUT6 BEL (output O_6), thus unused input pin A_6 should be assumed a logic level '1'. The rest of unused pins are assumed '1' as their default state. In the reduced table MP the columns corresponding to the inputs of the mapped LUT are reordered in descending order. After that, the rows are reordered by ascending input values of the mapped LUT. Each input value of the mapped LUT is linked to four table entries (rows), corresponding to the input values of the combined LUT. Hence, in the resulting mapping, each INIT bit of the mapped LUT macrocell has four corresponding BF bits (CM cells).

In general, the mapping multiplicity becomes $1 \rightarrow 2^N$, where N is the number of BEL pins that are used by an adjacent LUT but are not shared with the mapped LUT. Therefore, these non-shared inputs of an adjacent LUT will determine which of the mapped CM cells will drive the LUT output at any given time. It is worth noting that post-place-route models exported by Vivado for simulation do not reflect the LUT combining. With respect to the fault injection this means that, in case of LUT combining, one fault target (LUT bit) in the simulator may correspond to several fault targets (CM cells) in the FPGA. This may lead to discrepancies in dependability estimates between SBFI and FFI experiments, such as the underestimation of the number of critical bits in case of SBFI.

5.2.2 Mapping of Block RAMs

The mapping between the BRAM content and the CM can be extracted from the logic allocation (LL) file generated by Xilinx Vivado suite. However, the mapping provided by the LL file has two limitations. First, the BRAM macrocells are only referenced by their slice coordinates $Slice_{XY}$, without any relation with the source tree. Second, it lists all 16Kb/32Kb of BRAM18/BRAM36 content, even when these BRAM macrocells are used only partially. For instance, a BRAM may implement a small RTL memory array of WD bits wide and 2^{WA} words depth, as depicted in Fig.5.6. The resulting LL file will include all bits in the range [0-32767] in the case of RAMB36, or their even/odd subset in the case of RAMB18 with even/odd Y coordinate, as it has been explained in Section 3.4.2. In order to determine which of these bits are indeed essential for the DUT, it is necessary to establish a bit-accurate mapping between the source data structure and the CM cells corresponding to the inferred BRAM.

The procedure depicted in Fig.5.6 accomplishes such mapping for a BRAM configured in the most common TDP (true dual port) mode; only one port is used in

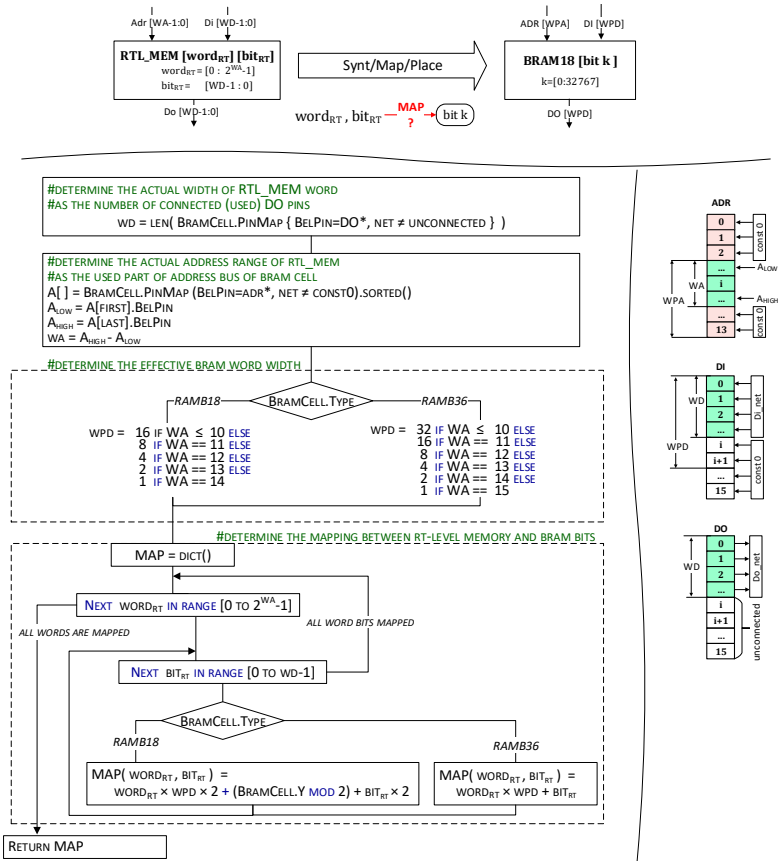


Figure 5.6: Procedure for locating the RT-level memory content within the inferred RAMB18 cell

the considered example. It takes into account such BRAM cell properties as slice type (RAMB18 or RAMB36), Y slice coordinate (in case of BRAM18), and an aspect ratio of the BRAM port. First, the actual word width *WD* of the source data structure is determined as the number of used pins of the data output port *DO*. In Vivado this can be accomplished by a TCL command: *llength [get_nets -of_objects [get_pins -of_objects $cell -filter {BUS_NAME=~DO*}]]*. Likewise, the actual address width *WA* is determined as the number of used pins of the address bus. It should be noted that unused address pins are tied by Vivado to the logic 0 (*const0* net).

The width *WPD* of internal BRAM words does not necessarily match the actual width of output data. It rather determines the data alignment pattern within the BRAM, and is dependent on the predefined port aspect ratios (listed in [177] for different RAM types and operating modes [TDP/SDP]). For instance, RAMB18 in TDP mode can be configured as $2^{10} \times 16$, $2^{11} \times 8$, $2^{12} \times 4$, $2^{13} \times 2$, $2^{14} \times 1$. The corresponding range of address pins *WPA* always occupies the topmost part of address bus: [13 : 4], [13 : 3], [13 : 2], [13 : 1], [13 : 0] respectively. Accordingly, the data alignment pattern (*WPD*) can be calculated from the actual address width WA, as it is depicted in Fig.5.6.

Based on this alignment pattern it is possible to identify the location of the actual data structure (RT-level array) within the BRAM, this location is designated by the mapping $MAP(word_{RT}, bit_{RT})$. In the case of RAMB36 each $word_{RT}$ is mapped onto the first *WD* bits of BRAM macrocell, located at the offset $word_{RT} \times WPD$. In the case of BRAM18 it is also necessary to take into account the data interleaving between the adjacent bottom and top BRAM macrocells, as it is depicted in Fig.5.6.

The BRAM bits in the resulting *MAP* dictionary are functionally essential (represent the source data structure), and should be considered as fault injection targets. Each entry in this dictionary is annotated with the CM coordinates from the LL file. The rest of bits listed in the LL file are non-essential and can be omitted for fault injection.

Fig.5.7 illustrates an example of locating the content of RT-level memory within the inferred BRAM18 by means of the proposed algorithm. The source data structure, described by *gpram* signal in the behavioural VHDL model, comprises 128 words of 8 bits. It is initialized according to the pattern $word_i = 0xFF - i$, i.e. $word_0 = 0xFF, word_1 = 0xFE, ..., word_{127} = 0x80$. This allows to highlight this data structure within the initial content (INIT) of the inferred BRAM, in order to verify the mapping algorithm. The *'block'* attribute, attached to the data structure, instructs Vivado to synthesize this memory using BRAM whenever possible. After synthesizing and implementing this RTL model by Vivado 2018.3 suite onto the XC7Z020 device (Xilinx 7-series family), the resulting data structure is placed into the BEL RAMB18_X2Y50. The inferred BRAM is configured in TDP mode, in which only port A is used. The resulting LL file lists all 16383 bits of the inferred BRAM, in such a way that it includes only even bit indexes (*BIT0, BIT2,...,BIT32766*), since the BRAM macrocell is placed into the BRAM18 slice with an even Y coordinate.

The mapping starts by determining the actual address word width *WD* and address range *WA*. The data output bus of port A (*DOA*) has 8 leftmost pins used,

```
entity RAM is
  port( clk  : in  std_logic;
        reset  : in  std_logic;
        data_i : in  std_logic_vector(7 downto 0);
        data_o : out std_logic_vector(7 downto 0);
        adr_i  : in  std_logic_vector(6 downto 0);
        wr_i   : in  std_logic;
        en_i   : in  std_logic);
    attribute bram_map: string;
    attribute bram_map of RAM: entity is "yes";
end RAM;

architecture behav of RAM is
   type   ram_type is array (127 downto 0) of unsigned(7 downto 0);
   signal gpram: ram_type := (
          0   => X"FF",
          1   => X"FE",
              ...
          126 => X"81",
          127 => X"80" );

   attribute ram_style        : string;
   attribute ram_style of gpram : signal is "block";

begin
    -- behavioral RAM description below ...
```

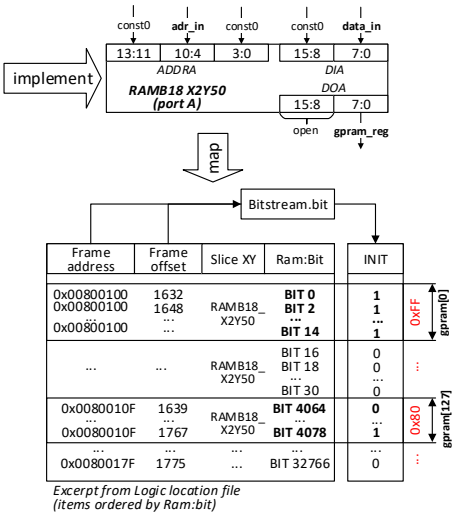


Figure 5.7: Example of locating the RT-level memory content (128x8 bits) within the inferred RAM block (18K bits listed in LL file)

i.e. *WD=8*. In the input address bus of port A only the pins [10:4] have an assigned net, while the rest of them are tied to logic 0, thus the actual address width is *WA=7*. It is worth noting that despite *WD* and *WA* parameters are known a-priory from the source model in this example, it may not be the case when using a third party IP. Since the type of inferred macrocell is RAMB18 and *WA<10*, the internal data alignment pattern should be $2^{10} \times 16$, hence the width of internal data words *WPD=16*. It can be seen that internal data words are two times wider than source data words. Accordingly, the low half of internal words stores the actual data, while their topmost half corresponds to the dummy data (constant 0). That is why the topmost half of data input port (DIA[15:8]) is tied by Vivado to *const0*.

The resulting mapping of the source data structure to the BRAM content can be now determined by applying the mapping procedure in Fig.5.6:

- $gpram_0 \rightarrow BIT\{14, 12, 10, 8, 6, 4, 2, 0\}$
- $gpram_1 \rightarrow BIT\{48, 46, 42, 40, 38, 36, 34, 32\}$
- ...
- $gpram_{127} \rightarrow BIT\{4078, 4076, 4074, 4072, 4070, 4068, 4066, 4064\}$

By looking-up the CM coordinates corresponding to the selected BRAM bits in the LL file, it is possible to extract the initial content of the mapped (RTL) mem-

ory. In this example it is $gpram_0 = 0xFF$,..., $gpram_{127} = 0x80$, that matches the initial RTL model of this data structure.

In the considered example only 1024 mapped BRAM bits are functionally essential out of the total 16383 bits listed in the LL file. Regarding fault injection, this allows to deploy much more selective/precise fault injection experiments to explain the impact of SEUs affecting the mapped memory on the design behaviour with much finer granularity than by blindly targeting all 16383 bits listed in the LL file for that BRAM. Additionally, this significantly reduces the experimentation effort, as only $^1/_{16}$ of BRAM bits are in fact essential.

5.3 Optimized essential bits

Bit-accurate mapping of netlist macrocells can be exploited to optimize the location of essential bits of the DUT and to address the limitations of Xilinx EBC mask. The complete procedure for generating an optimized essential bits file is depicted in Fig.5.8. In essence, this procedure builds a binary file comprising a set of mask records for each CM frame. Each mask record is annotated with its corresponding frame address (*FAR*) and comprises *FrameSize* words of 32 bits each. Each mask *word* denotes the corresponding *bits* of the configuration memory that should be targeted at fault injection, i.e. *mask[FAR][word][bit] = 1* for those CM cells that are essential.

This procedure masks only those CM bits that correspond to the selected type of logic cells and/or to the selected design scope, provided as input parameters *LogicType* and *Scope*, respectively. These two parameters are internally used to lookup the CM (bitstream) coordinates of matching CM bits within the MAP dictionary (previously obtained by the bit-accurate mapping), as it is depicted in Fig.5.8. It is thus possible to deploy the selective bit-accurate FFI experiments for those CM cells that correspond to the mapped macrocells (Flip-Flops, BRAMs and LUTs). CM cells corresponding to the rest of fabric resources (routing, MUXes and DSPs) can be also incorporated into this essential bits file from the Xilinx EBC file, although with a coarse granularity (without relating them with the DUT hierarchy).

The generation of an optimized essential bit file starts by allocating the empty mask for a complete list of valid FAR entries *FarList*. In case that the CM cells corresponding to all type of resources (all type-0 CM frames) should be considered, the content of an EBC file is parsed (mapped onto the list of valid frame addresses *FarList* for a given FPGA) and copied to the mask. After that, the mask is updated by highlighting (setting to 1) those mask bits that match the

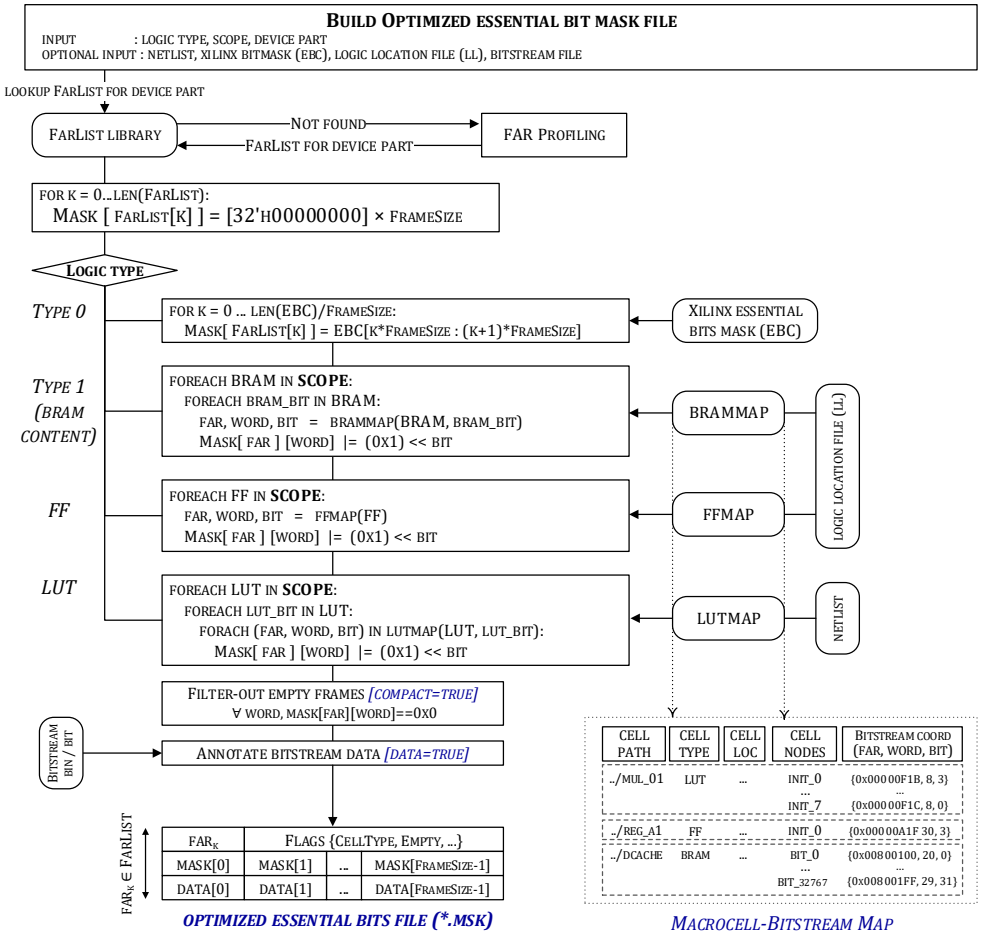


Figure 5.8: Procedure for generating an optimized essential bit mask file

selected design scope and that correspond to the essential BRAM bits (extracted from the BRAM Map) and to the FF readback CM cells. The inclusion of BRAM and FF is an extension to the Xilinx essential bits, since the latter includes only non-changeable CM cells. Finally, the mask is set for those LUT-related CM cells that match the selected design scope, taking into account that each LUT bit may have several matching CM cells (as it is explained in Section 5.2.1).

It is worth noting that prior to generating a custom LUT mask, the corresponding mask frames are cleared, since they are already highlighted in the EBC file, but

in a redundant way. Thus, replacing them with a custom LUT mask improves the precision of resulting essential bits. Fig.5.9 exemplifies a potential benefit of considering the custom essential bit mask for LUTs instead of the mask provided by the EBC file. It illustrates a sample utilization of LUTs of one CLB slice, along with the mask (custom and EBC) and corresponding bitstream fragments. As it can be seen, EBC always masks all 64 LUT bits even when LUT is used partially, whereas the custom mask highlights only those bits that are actually essential. For each individual LUT with k unused input pins the mask is reduced by $(1-1/2^k)\times 100\%$. A potential benefit of considering a custom BRAM mapping instead of using the raw LL file has been discussed in Section 5.2.2.

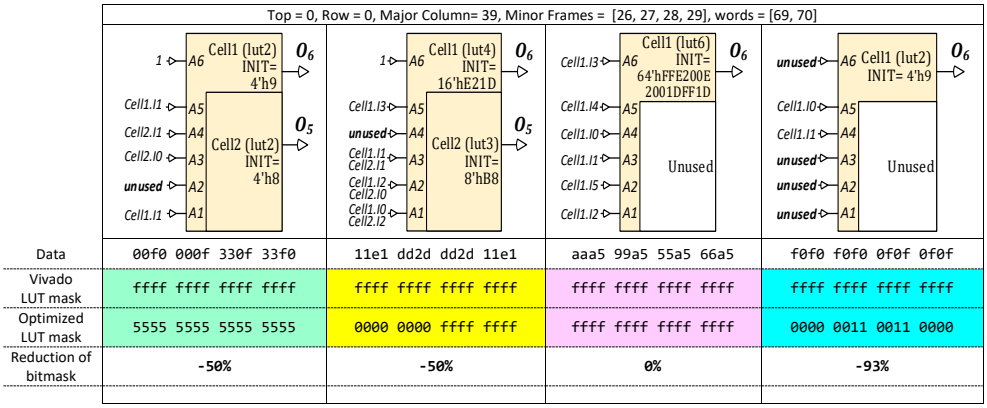


Figure 5.9: Example result of optimizing essential bit mask for LUTs of one CLB slice

Unlike the Xilinx EBC mask, the proposed optimized bitmask is annotated with frame addresses. This makes it not only more manageable at FFI, but also more compact, since unmasked frames can be removed from the mask file. A compact file, depicted in Fig.5.8, is especially useful when the FFI tool uses on-chip memory (very limited in size). Furthermore this accelerates the selection of fault targets when essential bits are sparse.

Since the list of frame addresses does not appear neither in the Xilinx mask (EBC) nor in the bitstream (bit/bin) files, it should be somehow obtained before proceeding to generating the optimized bitmask. The proposed approach to obtain the precise list of valid FAR entries is to profile the FAR register by means of the algorithm depicted in Fig.5.10. It is based on a frame-by-frame readback of configuration memory frames in the FAR auto-increment mode. The starting valid FAR is *32'h00000000*. Once the readback transaction of each frame is completed, the FAR register is changed automatically by an FPGA configuration controller

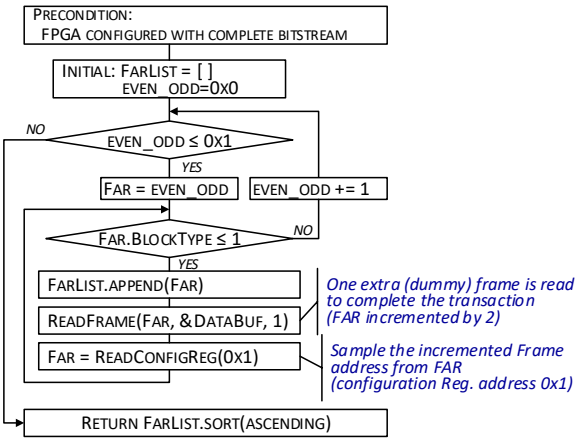


Figure 5.10: FAR profiling procedure used to extract the list of valid frame addresses for any given device part

to the next valid frame address. At this point the FAR is sampled by reading back the configuration register (register ID=0x1). This sequence is executed in a loop until FAR is not incrementing any more or until its BlockType field exceeds 1 (out of scope of valid configuration data). Since a valid read transaction should output one extra dummy frame (to flush the buffers as explained in [176]), the FAR is actually incremented by 2. Thus the profiling procedure is executed twice: first to sample the even frames (FAR starting at 0x0), afterwards to sample the odd frames (FAR starting at 0x1). The collected list of valid FAR entries (sorted ascending) for a given device can be cached by an FFI tool and quickly retrieved on the next run of mask generation (or any bitstream manipulation) procedure.

Mask frames can be complemented by configuration data frames extracted from the bitstream file (*.bit or *.bin). When injecting the faults into the non-changeable memory (all CM cells except BRAM, LUTRAM, and FF), this allows to skip the *Readback* step in the standard *Readback-Modify-Write* sequence, thus to additionally speed-up the fault injection process. The procedure to extract the configuration data frames from the bitstream is depicted in Fig.5.11. It is based on the interpretation of configuration commands, searching for a particular sequence that introduces the configuration data packets: *write FAR command* →*Write config data command* →*write FDRI command* →*number of words* →*data packet.*

Finally, the mask record for each frame is complemented with a *CellType* descriptor, which denotes the type of fabric macrocell being configured by the given frame. This allows to select the proper injection procedure for any given target

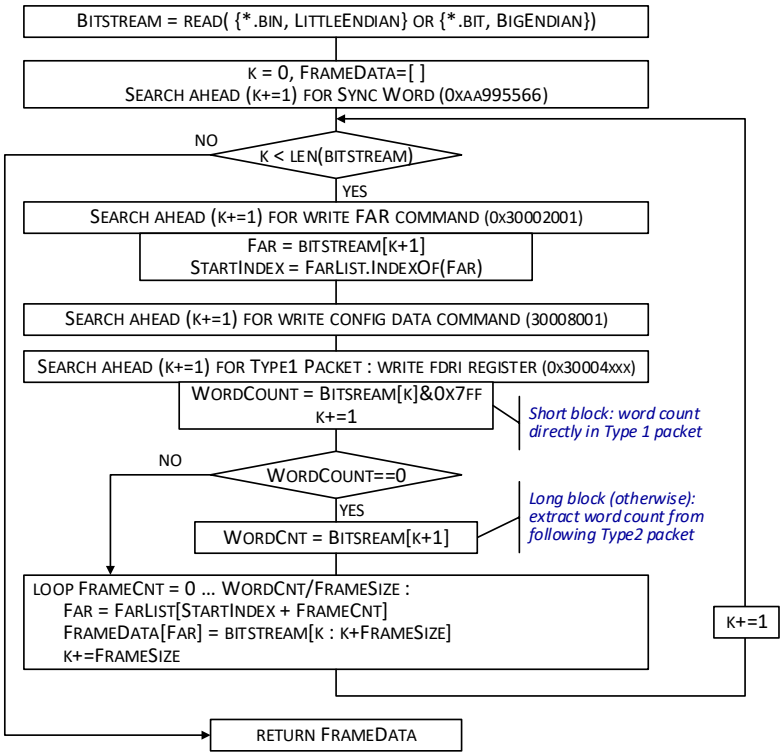


Figure 5.11: Procedure for extraction of configuration data from the bitstream file and their annotation with frame addresses

selected from the essential bit mask, as it will be detailed in the next Section 5.4. Most frames are labelled by the corresponding type of fabric resources, while those frames configuring several types of macrocells (if any) are marked as mixed-type.

5.4 Exploiting optimized essential bits for the bit-accurate emulation of SEUs

Optimized essential bits file (MSK) can be used by FFI tools to locate the relevant fault targets (CM cells) within the DUT (or its constituent modules) with bit-accuracy. The FFI tool itself can be implemented on the same FPGA device where the targeted DUT is implemented. This reduces the communication with the host PC only to uploading the MSK file to the memory of the FFI tool and to reporting the FFI results upon the completion of experiments.

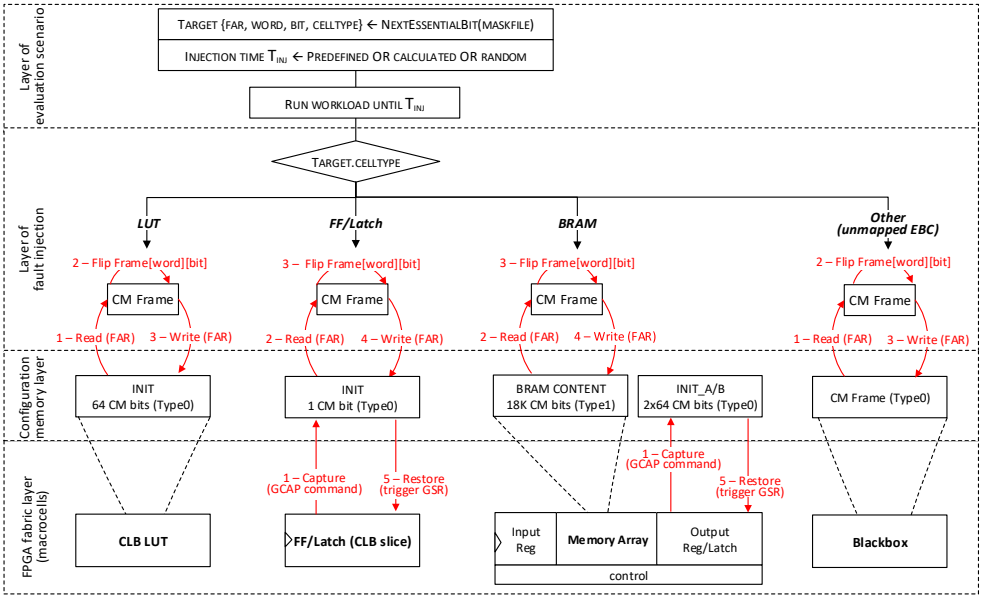


Figure 5.12: SEU injection procedure based on optimized essential bits

The FFI application running on the device can be split into two layers: the evaluation scenario layer and the fault injection procedures layer, as depicted in Fig.5.12. The scenario layer is DUT-specific, being in charge of (i) generating a workload for the DUT, (ii) checking the responses from the DUT (determining the failure modes), and (iii) generating the faultload, i.e. the fault configurations comprising the fault targets and the injection time. For instance, the verification of SEU mitigation mechanisms may require to exhaustively test the effects of single-bit upsets in all essential bits within a certain design scope. Another experimentation scenario, estimating the aggregated dependability metrics of the DUT, may require to emulate single-bit and multiple-bit upsets randomly sampled in time and space.

The injection layer itself is DUT-independent, being in charge of properly emulating the occurrence of SEUs in the selected fault target. To this end, it takes into account the type of logic primitive behind the targeted CM cell. For instance, an upset in the LUT-specific CM cell located at the coordinates *FAR, word, bit* can be injected by: (i) reading the configuration data frame from the device (at the address FAR), (ii) inverting the value of the selected bit (*Frame[word][bit]*), and (iii) writing back the modified data frame. After that, changes made in the

CM are directly reflected in the LUT content. The procedures for reading and writing the CM of Xilinx FPGAs through the ICAP and PCAP interfaces are detailed in the Annex A.1. It is worth noting that the configuration data, being incorporated into the bitmask file, can be used to quickly retrieve the reference value for the targeted CM cell instead of reading it back from the device. But this is only applicable for non-changeable CM cells, i.e. those corresponding to type-0 frames except for FF readback cells and LUTRAM.

Injecting bit-flips into registers requires to first capture their current state in the associated readback CM cells (selected from the MSK file). The captured value is modified in the associated CM cell using the same procedure: *Readback(FAR) →Flip(Frame[word][bit]) →Write(Frame, FAR)*. Finally, the Global-Set-Reset (GSR) signal is triggered to reinitialize the registers to their associated readback CM cells and, thus, to set the targeted register to a new (inverted) logic level previously written to its corresponding CM cell. As it is explained in [67], triggering the GSR through the configuration command *'GRESTORE'* may have no effect. Thus, to ensure the *restore* operation one should trigger the GSR line through the STARTUP primitive.

It must be noted that the clock in the targeted FPGA region should be paused before performing any CM operation in order to prevent the unexpected corruption of the CM content.

The BRAM content is directly stored within the type-1 CM frames. Hence, the injection of bit-flips into the BRAM content can be performed by applying the same Read-Modify-Write procedure to the corresponding CM cell. However, the FPGA configuration controller accesses the BRAM content through the IO ports of the BRAM, thus sharing them with the DUT. For that reason, any read or write operation of BRAM content alters the state of BRAM's output register. As a result, when resuming the DUT clocking after the fault injection, the targeted BRAM gets desynchronized from the DUT due to the corrupted value of its output register. This may drive the DUT to a faulty state even on simple readback operations.

Accordingly, the injector would require to save the state of BRAM output registers before reading or writing the BRAM content and to recover the saved register state afterwards. The readback CM cells corresponding to the BRAM output registers are not easily located, since they are not reported by Vivado in the EBC file nor in the LL file. Nevertheless, there is no need to actually read these cells back from the CM. It is enough to capture the register state in the CM by issuing a global capture command (*GCAP*), before performing the read/write of BRAM frames,

and to restore it afterwards by triggering the GSR line. As a result, the BRAM registers will be kept synchronized with the DUT after the fault injection.

It is worth noting that the global capture and restore commands can be masked on a per-column basis. By writing a special frame with the *FAR.BlockType=2* each column can be protected/unprotected from the capture/restore operations [116]. By-default the entire device is unprotected. When the DUT is paused and there is no other logic running in the target FPGA, the capture/restore operations can be safely performed in unprotected mode. However, when performing these operations on a partial region, the rest of FPGA (which keeps running) should be protected. This protection mechanism is particularly required when the fault injector is also implemented by the configurable logic.

Finally, the standard Read-Modify-Write procedure can be followed to inject upsets into the unmapped CM cells retrieved from the Xilinx EBC file, since it reports only the non-changeable essential bits which do not require any capture/restore operations.

Once the fault is injected, the DUT clock is activated again to run the rest of the workload. At this point, the failure mode is determined by tracing the DUT outputs and DUT internal state. A wrong DUT output (that does not match the golden run) is reported as a failure, either signalled or silent. Otherwise, the DUT internal state is traced to detect latent errors. This is achieved by issuing a capture command, followed by the readback of all those frames that contain the state elements (registers and changeable memories). The frame addresses to trace are extracted from the MAP dictionary (its FF and BRAM entries). Any mismatch with the reference trace is reported as latent error. The targeted CM cells whose upsets have led to a failure are traced through the macrocell MAP dictionary back to the netlist level, and subsequently to the source HDL model in order to locate the weak points of the DUT. Backtracing those CM cells whose upsets lead to latent errors reveals the fault propagation paths within the DUT.

5.5 Conclusions

The dependability assessment of FPGA-based designs by means of FFI experiments requires to accurately locate the relevant fault targets within the configuration memory (CM). On the one hand, targeted CM cells should be related with the DUT hierarchy in order to identify its weak points. On the other hand, optimizing the experimental effort requires to reduce the injection scope to only those CM cells that are essential for a given DUT and for its individual units. The Xilinx essential bits (EBC) and the logic allocation (LL) files are two known

aids for the location of essential cells of changeable and non-changeable CM respectively. These aids, however, do not allow to deploy the FFI experiments in a hierarchical way and lead to the consideration of redundant (non-essential) CM cells when FPGA BELs are utilized partially.

To address these limitation this chapter has studied the bit-accurate mapping of LUT and BRAM macrocells onto the CM and, on its basis, it has optimized the location of essential bits. The proposed LUT mapping algorithm locates each individual bit of the LUT's truth table within the CM, taking into account all placement-routing optimizations, such as LUT combining and optimized assignment of BEL pins. The BRAM mapping algorithm discovers the location of each individual bit of the source data structure within the inferred BRAM. The optimized essential bits file, defined on the basis of mapping procedures, is generated in a hierarchical way for the mapped macrocells, allowing to deploy much more fine-grained FFI experiments and to obtain the robustness estimates for each individual module in the DUT hierarchy. At the same time, the proposed mapping algorithm eliminates the redundancy existing in Xilinx EBC and LL files in the location of LUT-specific and BRAM-specific essential bits, which may notably speed-up the experimentation. Additionally, the generated essential bits file is self-descriptive and compact, thus being more manageable for FFI experiments than the Xilinx EBC file.

The proposed SEU injection procedure, based on an optimized essential bit mask, takes into account the type of logic primitives behind the targeted CM cells, allowing to emulate SEUs in registers, changeable memories, and non-changeable CM.

To consider the rest of FPGA resources, whose mapping has not been covered by the proposal (routing, internal slice configurations and DSPs), the Xilinx EBC mask has been also incorporated into the optimized essential bits. Accordingly, FFI experiments for such unmapped CM cells still remain coarse-grained (not related to the DUT hierarchy). It is also worth noting that the proposed mapping algorithms are limited to the Xilinx 7-series FPGA. Thus, the future research should study the mapping of the rest of FPGA resources and consider other FPGA families.

Chapter 6

Contributions in Improvement of Fault Injection Performance

This chapter proposes a set of techniques to improve the performance of SBFI and FFI experiments from a practical perspective. Section 6.1 introduces the proposals. Section 6.2 describes two complementary techniques to reduce the number of experimental runs. The first presented technique filters and prioritizes the essential bits of FPGA implementation through the profiling of the switching activity. The second technique relies on the use of statistical fault injection to reduce the number of experiments to carry out beyond what conventional statistical fault injection does. Section 6.3 pushes the performance improvement one step forward by proposing an additional technique for speeding-up the fault injection runs through the use of mixed-level models, multi-level fault injection, and simulation and FPGA-based checkpointing. Each proposal is introduced and discussed from a quantitative viewpoint, i.e. by computing the speed-up that one can obtain from its use. Section 6.4 discusses the various proposals in the context of existing techniques for speeding-up the fault injection. Section 6.5 summarizes the advantages and limitations of proposed techniques and concludes this chapter.

6.1 Introduction

The fault injection performance is commonly optimized from two perspectives: the reduction of the number of considered fault configurations, and the acceleration of the injection runs. Some existing proposals in this field (discussed in Section 3.5), such as checkpointing and multilevel fault injection, are defined in a very generic way, without any insights on how they can be applied to SBFI and FFI experiments, and what speed-up gain can be expected from them in a each particular case. On the other hand, under the complex experimentation scenarios, the speed-up provided by the existing techniques still may be insufficient. This chapter addresses these problems in two ways: (i) by defining additional techniques for the further improvement of fault injection performance, and (ii) by studying the practical aspects of applying the defined and existing techniques to SBFI and FFI, and providing the models for estimation of their attainable speed-up gain.

The number of experiments can be reduced following a two-way strategy involving the optimization and sampling of the considered fault space. Particularly, the bit-accurate FFI, defined in chapter 5, can be also seen as a technique looking for the optimization of the fault space. Indeed, it reduces the set of relevant fault targets by being more selective in the location of essential bits than the Xilinx's essential bits file. A more specific optimization, proposed in Section 6.2 of this chapter, reduces the number of relevant fault targets even further by analysing the switching activity of DUT macrocells. The idea is to locate those CM cells, that despite belonging to essential bits, remain inactive at the given workload, and may be safely omitted from the faultload.

Even after deploying the platform-specific optimizations of the fault space, it may still remain infeasible to conduct an exhaustive fault injection campaign. In this case, the statistical fault injection approach formalized in [100], can be exploited to reduce the faultload to an affordable size, at the cost of assuming a certain confidence level and error margin in derived robustness metrics. This approach, however, relies on a conservative assumption regarding the estimated robustness metrics, that leas to overestimation of required sample size. Section 6.2 proposes an iterative statistical fault injection approach, which significantly reduces the sample size when the actual robustness metrics significantly differ from the conservative assumption.

After having reduced the faultload, the focus is placed on the acceleration of the remaining fault injection experiments. Section 6.3 proposes two solutions to this end. The first solution exploits the high-level (RTL) models to speed-up the implementation-level SBFI in two different ways, referred to as mixed-level and

multi-level injection. The former generates an HDL model combining the targeted unit at implementation level with the rest of the design simulated at RTL, thus speeding-up the simulation of each injection run. The latter establishes a mapping between the implementation-level macrocells and the source structures at RTL, in order to identify the fault configurations that can be representatively simulated at the faster RT level. Another speed-up solution relies on the ckeckpointing of the DUT state in SBFI and FFI experiments. Despite the multi-level injection and checkpointing are known concepts, their practical aspects and attainable speed-up gain so far have been covered rather superficially. Accordingly, section 6.3 studies how these concepts can be efficiently integrated into a simulation-based and FPGA-based fault injection flow, and how their attainable speed-up gain can be estimated.

6.2 Strategies to reduce the number of fault injection runs

This section proposes two different techniques to reduce the faultload. The first technique analyses the switching activity of DUT primitives to identify the inactive CM cells that can be omitted from the faultload. The second technique deploys the statistical fault injection in a iterative way. By taking into account the actual estimations of robustness metrics the proposed technique can significantly reduce the number of sampled faults with respect to the conventional statistical approach that basically makes a priori conservative assumption.

6.2.1 Filtering and prioritization of essential bits through the profiling of the target switching activity

Any FPGA-based implementation has its specific set of essential bits, which determine the circuit functionality and integrity. Fault injection experiments targeting these essential bits, allow to identify which of them are critical i.e. lead to a failure if toggled. The set of critical bits is determined attending to the circuit and the workload, which activates a certain set of control and data paths within that circuit. Advanced simulation tools allow to trace these paths by means of coverage graphs. For instance, code coverage tool of ModelSim/QuestaSim simulator keeps track of how many times each statement, branch or expression within the design gets executed during the simulation. These metrics are commonly used to improve the quality of tests. Conversely under the given workload they could indicate which parts of the design remain inactive and thus can be omitted at fault injection. However, the configuration memory (CM) layer is not directly reflected in coverage graphs, even by using implementation-level models. Indeed, the netlist of macrocells must accurately model the functionality and tim-

ing behaviour of technology primitives, but not necessarily detail their internal structure. Nevertheless, for some macrocells with known bitstream mapping (as detailed in Section 5.2), it becomes possible to determine which CM bits are active under the given workload, by profiling the switching activity on the macrocells interface.

This kind of analysis can be particularly accomplished for the Look-up tables (LUT), which are the basic building blocks of combinational logic in FPGAs. Structurally LUT comprises (i) an array of configuration memory cells M_i storing a truth table (*INIT*) of the LUT, and (ii) a tree of multiplexers which selects one of the M_i cells by the combination of its inputs *ADR*, and drives it to the LUT's output. Likewise LUT_X can be seen as an element of read-only memory, which asynchronously reads one of its 2^X bits to the output. For instance, as depicted in Fig.6.1, the 3 inputs (ADR_2, ADR_1, ADR_0) of a LUT_3 primitive address 8 CM cells, each one containing the 1-bit output for each combination of inputs. Under combination *ADR='000'* the topmost path is selected through the multiplexer tree, driving to the output the content of cell M_0; switching the ADR_2 to *'1'* (combination *'100'*) changes the selected cell to M_4, and so on.

A circuit running a workload not necessarily will access all CM cells in each LUT. In fact, many of them may be rarely or never accessed. A CM cell can be denoted as active at certain time instant if its value is driven (read-out) to the LUT output. Thus the total time which CM cell determines the LUT output during the workload execution can be denoted as its *activity time*.

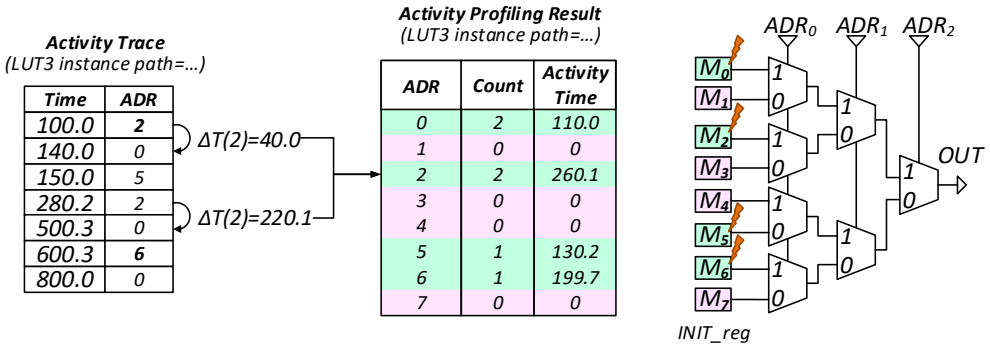


Figure 6.1: Profiling of switching activity on LUT inputs to determine inactive cells of configuration memory

The activity time of LUT-related CM cells can be estimated by profiling the switching activity on the LUT interface. Profiling comprises two steps: (i) simulation of implementation-level model of the circuit to monitor and log the switching

events on the LUT's input *ADR*, and (ii) analysis of the collected logs to compute the activity time of CM cells of each LUT.

The simulation step starts by configuring the simulation environment for tracing of switching activity. In the ModelSim simulator activity traces can be captured by means of the script in listing 6.1. First, the inputs of each LUT_i are concatenated in descending order into a virtual signal ADR_i, e.g. I_5 to I_0 for LUT_6. Created signal is appended for monitoring to a new list window. Dedicating a separate list to each LUT_i makes logging of their switching events independent on each other, reducing the total footprint of resulting logs and speeding-up their subsequent analysis. Once the traces for all LUTs are configured, the simulation is run for the workload duration. Each time when ADR_i switches, its value and a corresponding switching time are logged by the simulator to the activity trace in the format *(Time, ADR)*, as it is depicted in Fig.6.1. Finally, each collected trace is exported to a tabular file for the analysis.

There are two important observations regarding the simulation. First of all, the model must run exactly the same workload that will be used in FFI experiments, since activity profile is workload-specific. Second, the simulated model must accurately reflect the timing behaviour of the circuit, which means that a post-place-route model (in the basis of Xilinx's SimPrim library) with annotated SDF file should be used.

```
1  For each LUTi in netlist:
2      #concatenate separate LUT inputs into a single virtual signal
3      quietly virtual signal -env LUTi.Path -install LUTi.Path \
4             { ((concat_range (LUTi.size-1 downto 0)) (LUTi/I5 & ... & LUTi/I0) )} ADRi
5      #create a dedicated list for each LUT cell
6      set Labeli [view list -new -title Labeli]
7      #binary format allows fine-graned resolution of X states
8      radix bin
9      #append created signal to the list
10     add list LUTi.Path/ADRi -window $Labeli

11 run $WorkloadDuration

12 For each LUTi in netlist:
13     #save each trace to its dedicated file (tabular list)
14     view  list -window $Labeli
15     write list -window $Labeli  /Traces/Labeli.lst
```

Listing 6.1: ModelSim script to collect the activity traces for LUT cells (ModelSim commands highlighted in bold red; pseudocode in italics)

During the analysis step, each collected activity trace is processed to calculate the total activity time of each CM cell, as depicted in Fig.6.1. For instance, cell M_2 (*ADR=2*) has been activated twice during the simulation: at time instant 100.0 ns for 40.0 ns, and at time 280.2 ns for 220.1 ns, accounting for a total activity

time of 260.1 ns. Likewise the total number of activation events can be calculated - see column *'Count'* in Fig.6.1.

It is worth noting that Xilinx's simulation primitives support indetermination (*X-state*) on the inputs, in the case when resolving X to either 0 or 1 doesn't impact the result, i.e. when both alternative branches of corresponding multiplexer drive the same value to the LUT output. Despite the X-state is just a simulation abstraction, the analysis of activity traces should account the activity time interval with indetermination to both alternative *ADR* values, since either of two path may be activated in the hardware.

After completion of this analysis, all addresses in the profiling table with a zero activity time (or with a *Count* value of 0) indicate those LUT bits that have remained inactive during the simulation of the HDL model. However, being more strict, computation of activity metrics may be not so straightforward, since glitches also could be filtered out. These glitches are very short (multiple) transitions that may appear in signals before they become stable. Consider that cell 000 is accessed and next access goes to cell 111. In practice, changing the value of these 3 bits may suppose to transition from 000 to 100, then to 110 until reaching the stable address 111. This is why instead of registering all switching events, it may be worth to take into account only effective switches, i.e. those not filtered by the electrical characteristics of the circuit. Under the inertial delay model the glitches are filtered, while transport delay model [118] propagates such glitches.

The profiling results can be exploited to improve the dependability assessment in two ways. First, all CM cells that remained inactive at profiling can be skipped at fault injection experiments, as they are non-critical. The attainable speed-up gain equals the percentage of inactive bits within the complete set of essential bits. Second, a hypothesis for the rest of cells is that the higher is the activity time of a certain CM cell, the higher is the probability that its fault will not be masked, and thus the higher is its criticality (probability to lead to a DUT failure if toggled). Considering a limited experimentation time, it is interesting to prioritize fault injection experiments according to the criticality of the bits they target.

The application of obtained activity metrics to the optimization of SBFI experiments is quite straightforward: each inactive bit of LUT content (INIT) is removed from the list of fault targets. To use the activity metrics for the optimization of FFI experiments, it is necessary to first map each bit of LUT content onto the corresponding CM coordinates. This is accomplished by means of the LUT mapping procedure, proposed in Section 5.2. It should be noted that Xilinx Vivado tookit may combine two LUT primitives within one LUT BEL, in

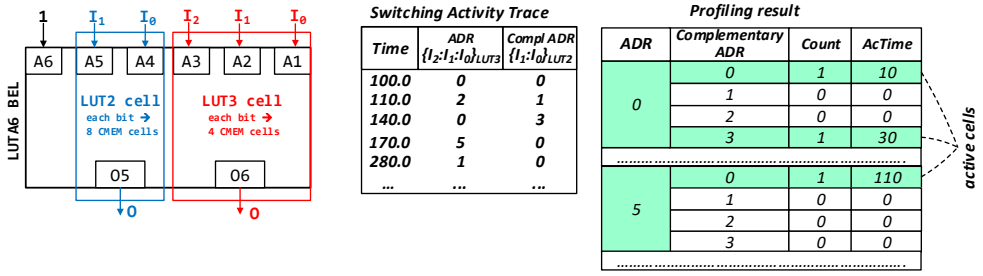


Figure 6.2: Profiling of switching activity in case of LUT combining with non-shared inputs

such a way that the combined LUT primitives may have one or more non-shared inputs. In this case, each INIT bit of mapped LUT may correspond to several CM cells. Hence, to determine, which CM cell is active at each time instant, it is necessary to consider the switching activity of both combined LUT primitives. Such activity trace includes two *ADR* values (depicted in Fig.6.2): inputs of the profiled LUT primitive (*ADR*), and those complementary inputs of adjacent LUT primitive which are not shared with the profiled LUT. Monitoring the switching events of this complementary *ADR* allows to resolve this one-to-many relation between the INIT bits of the profiled LUT primitive and corresponding CM cells, thus to annotate the proper activity time to each CM cell even in the case of LUT combining.

In the example illustrated in Fig.6.2 each bit of the profiled *LUT_3* has 4 corresponding CM cells, since the adjacent *LUT_2* occupies two other inputs *A5* and *A4* of the same LUT BEL without sharing them with the profiled *LUT_3*. The profiling monitors the switching events of the tuple comprising the *ADR* of the *LUT_3* and the *ADR* of complementary (adjacent) *LUT_2*. In the resulting trace the tuple (0,0) is active for 10 ns, the tuples (0,1) and (0,2) don't appear (they are inactive), and tuple (0,3) is active for 30 ns. Accordingly, only the 1st and the 4th CM cells are active, out of four CM cells mapped to the bit 0 of *LUT_3*. Similarly the rest of tuples are analysed.

Fig.6.3 illustrates the complete optimized faultload generation flow, which integrates the optimization of essential bits proposed in Section 5.3 and the profiling of switching activity introduced in this section. The faultload generator first obtains a list of LUT primitives cells from the netlist parser. These primitives are subsequently mapped onto the configuration memory by the bitstream parser, which implements the bit-accurate mapping algorithm proposed in Section 5.2. For each mapped LUT primitive it creates a descriptor, illustrated in Fig.6.4. The netlist parser annotates this descriptor with the properties of LUT primi-

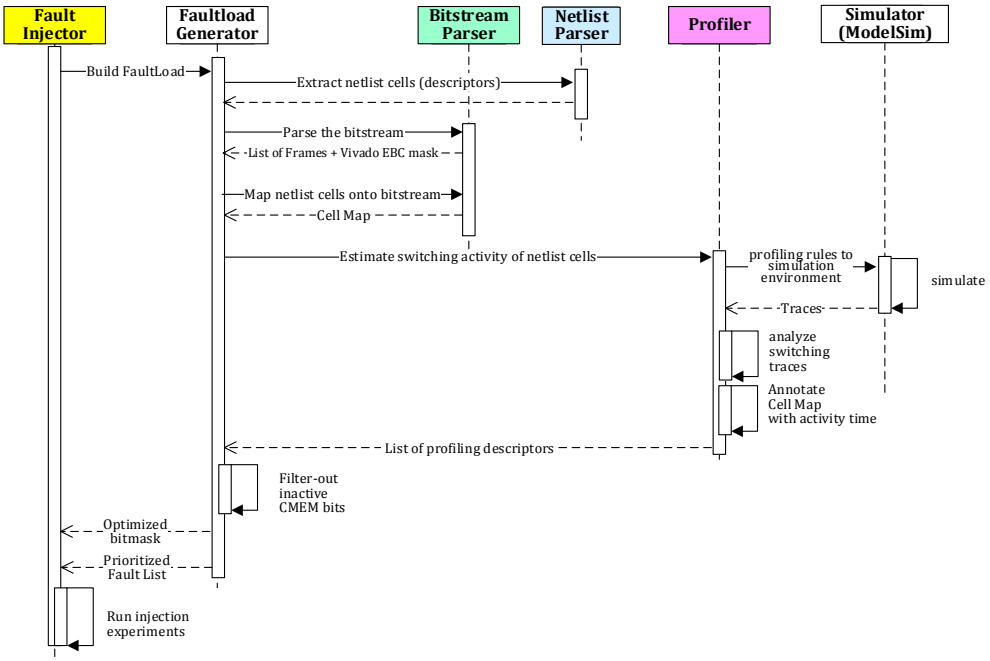


Figure 6.3: Interactions among entities involved in the generation of an optimized faultload for an FPGA-based fault injection campaign

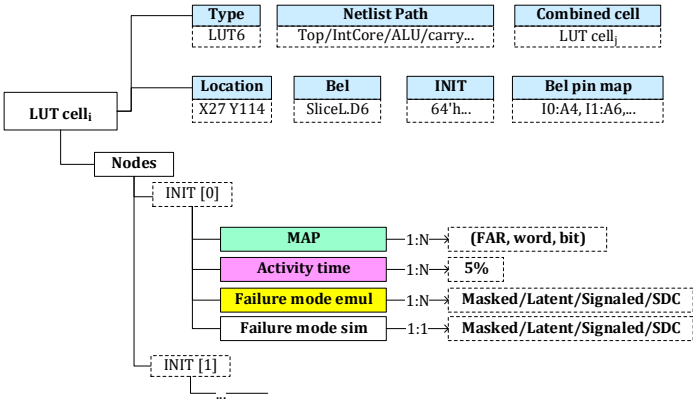


Figure 6.4: Structure of resulting LUT cell descriptors after mapping, profiling and fault injection

tive extracted from Vivado, such as the path, type, location, and pin mapping (highlighted in blue). The bitstream parser annotates the descriptor with the coordinates of CM cells corresponding to the LUT primitive.

The list of LUT cell descriptors is passed to the Profiler, which builds the profiling script for a given set of LUT cells (following the pattern in Listing 6.1), and launches it within the simulation environment. The simulator collects the switching activity traces of each LUT cell. Collected traces are analysed by the profiler to calculate the activity time and to count the activation events for each bit of LUT content. Resulting activity metrics are appended to the LUT descriptor (highlighted in red in Fig.6.4). The multiplicity of the relation between the LUT bit and its activity metrics is the same as the multiplicity of mapping between the LUT bit and its corresponding CM cells. The profiler annotates the LUT descriptors with activity information and returns them to the faultload generator.

Finally, the essential bit mask is build by the faultload generator, taking into account only active CM cells for the profiled LUTs. Under the constrained experimentation time, a prioritized fault list is generated, which can be used by the fault injector to schedule the experiments according to their activity time: CM cells with higher activity time are targeted with higher priority, since they are more likely to be critical. The filtered bit mask and prioritized fault list are returned to the fault injector, which executes the scheduled experiments.

6.2.2 Iterative statistical fault injection

The use of statistical sampling for robustness assessment typically follows a very conservative approach. As there is usually no estimation available of actual failure rates for the target system, the worst case scenario is considered: a fault is as likely to cause a given failure mode (let us say a Silent Data Corruption), as to not cause it. Thus the computation of sample size by the Equation 3.1 assumes $p = 0.5$, which means that we do not have any a priori knowledge on what will happen when a fault is injected. In other words, when p = 0.5, a fault can lead to a failure (probability of 50%) or not (the other 50%). At the same time, the analysis of this equation shows that such conservative approach leads to much larger sample size, than it would be required when the estimated failure rate is significantly lower (or higher) than $p = 0.5$. This difference becomes especially notable when a narrow margin of error is desired, what is common for dependability assessment of critical designs.

For instance, assuming an infinite population, an error margin of 0.1%, and a confidence interval of 95%, the conservative sample size ($p = 0.5$) would roughly raise a number of experiments to $n = 0.96 \times 10^6$. Now assuming an estimated failure rate of 1% ($p = 0.01$), the required sample size is reduced to just $n = 0.038 \times 10^6$. This means that the estimation can be obtained with the specified error margin and confidence interval by performing only $1/25$ of the experiments proposed by the conservative sampling approach.

So the problem is to use the knowledge we obtain during experimentation from the DUT in order to iteratively adjust the sample size to the strict minimum to guarantee the specified error margin and confidence interval in the finally provided estimations. Equation 3.2 could be used for this purpose. The conservative approach will lead to much narrower error margins than those required when estimated value is farther away from the worst case scenario ($p = 0.5$). This means that *an even smaller sample size would be actually necessary to obtain the desired margin of error*. This is an important observation for the dependability community, as the rate of failures (although unknown) is usually expected to be relatively low. This is because dependable systems are designed to keep the value of p (probability of failure), usually below 5%.

The proposal is to deploy an iterative strategy that dynamically schedules the fault injection experiments (samples the fault space), and that relies on a *realistic* (instead of a pessimistic) estimations of the actual sample size required for a given margin of error.

The basic experimentation strategy is driven by error margin, i.e. the goal is to estimate the failure modes exhibited by the considered target with the required margin of error with the minimum possible number of experiments (smallest possible sample). The algorithm that describes this proposal is depicted in Fig. 6.5.

Assuming there is no estimation available for any of the considered failure modes, it is necessary to start the process by taking into account the worst-case scenario (where each failure mode is estimated with a probability $p = 0.5$) but for a wide margin of error ($E = 5\%$). In such a way, an initial estimation for each failure mode rate will be obtained by running only 384 experiments.

In each iteration of the algorithm, the sample size, or number of fault injection experiments (N), is computed attending to the currently selected margin of error (E) and the current failure rate (P). Then the required number of fault injection experiments *beyond those already executed* are carried out. Results are used to re-estimate the rate (P) of each one of the considered failure modes, and these estimations are used to compute the new margin of error (E) associated to the

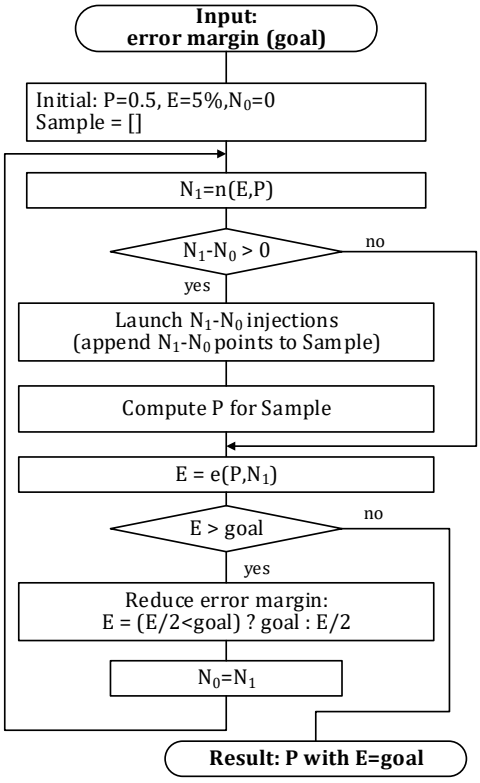


Figure 6.5: Algorithm to minimise the sample size for estimating a given failure mode with any given *goal* for the error margin

reported estimations. If this margin of error is greater than the required goal, then the currently selected error is reduced (let us say that it is halved) to slightly increase the considered sample size in the next iteration of the algorithm.

In such a way, the sample size is iteratively increased to determine when the required margin of error is reached. If the selected margin of error is not drastically reduced in each iteration, this algorithm ensures that the resulting sample size will be much closer to the minimum required to reach the margin of error than by following the conservative approach. As failure rates are usually expected to be very low, especially for critical systems, this proposal can greatly reduce the required sample size and the experimentation time.

The previously described strategy can also be adapted to consider a situation where it could be a hard limit in terms of time available for experimentation. In

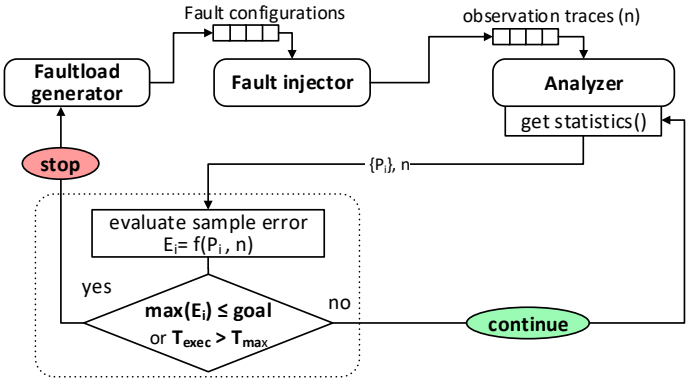


Figure 6.6: Fault injection process driven by both error margin and experimentation time

that case, the error margin could be iteratively reduced until the specified margin of error is reached or the maximum allowable experimentation time elapses.

As Fig.6.6 shows, this would require a fault injection manager that could be constantly monitoring (at a given rate) the estimated rate of failure modes to compute the current margin of error and schedule new fault injection experiments to iteratively increment the sample size. Although similar in concept to the previous proposal, in this case a much finer control over the sample size increments (probably constant) is required so as not to inadvertently exceed the allotted experimentation time. Thus, this procedure will either prevent unnecessary experimentation efforts when the error margin goal is reached, with the available experimentation time, or when this time is exceeded, it provides the means to quantify the error margin existing in reported estimations.

It is worth noting that if setting the error margin goal to zero, the proposed algorithm will stop either when reaching the experimentation time limit, or when sampling the complete population. In the latter case the true value of failure modes will be estimated (without any error). This requires to deploy the so-called sampling without replacement, which means that each individual (fault configuration) can be only chosen once. In practice this will require to keep track of already sampled individuals. In FFI based on the optimized essential bits (as proposed in Section 5.4), this can be achieved by clearing the mask (in the mask file) for already sampled targets.

$$S_{IS} = \frac{n(p = 0.5)}{n(p_{est})} = \frac{t^2 p_{est}(1 - p_{est}) + e^2(N - 1)}{p_{est}(1 - p_{est}) \times (t^2 + 4e^2(N - 1))} \tag{6.1}$$

$$S_{IS}(N \to \infty) = \frac{0.25}{p_{est} \times (1 - p_{est})} \tag{6.2}$$

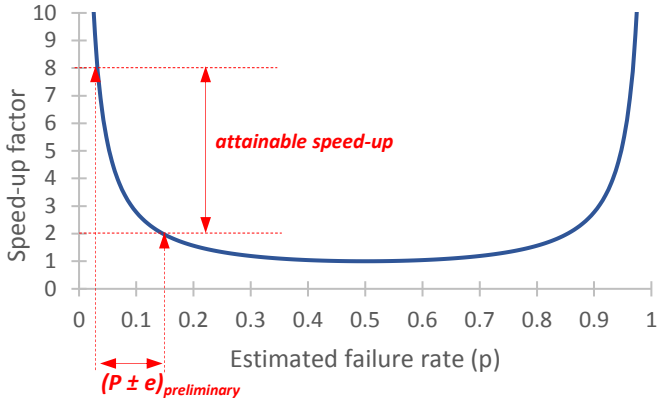


Figure 6.7: Speed-up attainable through iterative statistical sampling with respect to conservative approach; under infinite population and confidence interval of 95%

The experimentation speed-up achievable through the iterative approach can be quantified by dividing the sample size for the conservative scenario $n(p = 0.5)$ by the size of resulting sample collected by iterative approach $n(p_{est})$. For any given failure rate estimation p_{est}, error margin e and population size N, this speed-up can be computed by Equation 6.1, or by simplified Equation 6.2 for the infinite populations. The resulting speed-up can be computed only after the completion of experiments, when p_{est} becomes available. Nevertheless, one can estimate the magnitude of attainable speed-up when a preliminary confidence interval $(p \pm e)_{preliminary}$ for the estimated failure rate becomes available, as it is depicted in Fig.6.7.

6.3 Strategies to speed-up SBFI and FFI experiments

While the previous section has focused on reducing the number of fault injection experiments in a campaign, this section proposes different approaches to accelerate each one of the experiments to carry out. This goal can be attained by reducing the complexity of considered models and the duration of each simulation/emulation.

A high-level (RTL) model can be used to reduce the complexity of implementation-level SBFI experiments. This can be achieved through two different propos-

als. The first one is to generate a mixed-level model where the targeted design unit is described using an an implementation-level model, while the rest of the design is modelled at RTL. The resulting model is less costly to simulate, while allowing the same robustness estimations as those that can be obtained by using the pure implementation-level model. The second proposal relies on the fact that the structure of sequential logic (the set of registers in a hardware design) is usually maintained without major changes throughout the various steps (synthesis, mapping, placing and routing) of the semicustom design flow. This enables the mapping between the inferred technology-specific structures representing the sequential logic at the implementation level and the source registers specified at RTL. Thanks to this, fault injection experiments targeting a design's sequential logic can be carried out using a multi-level fault injection approach: faults injected into mapped registers will be carried out using the related RTL model (which is fast in simulation), while for the rest of registers (replicated, and/or affected by retiming) the implementation-level model will be considered.

In order to reduce the time required to cope with each simulation/emulation, the use of checkpointing techniques becomes essential. The idea is to capture the DUT execution context at several uniformly distributed time instants during the golden run. Each of the subsequent injection runs will start with the recovering of the checkpoint with the closest timestamp. This will reduce the total run time of each injection experiment.

The following subsections further explain to what extend each experiment in a fault injection campaign can be accelerated thanks to the use of mixed-level and multi-level models and the deployment of checkpointing techniques.

6.3.1 Mixed-level and multi-level fault injection

As it has been discussed in Section 3.3, RT-level models are relatively fast in simulation, but not detailed enough to draw the confident conclusions regarding the dependability of of a system. Implementation-level models, however, reflect most relevant implementation details of resulting circuits, including the target technology, timing, and EDA optimizations. This level of detail, makes implementation-level models very slow in simulation. Exploiting the advantages of both models within the same SBFI campaign may allow to speed-up the experimentation with respect to the pure implementation-level SBFI.

Implementation-level models consist of hundreds of thousands of different basic elements whose simulation, even for simple workloads, may take a very long time. Hence, the first proposed optimization focuses on reducing the complexity of that

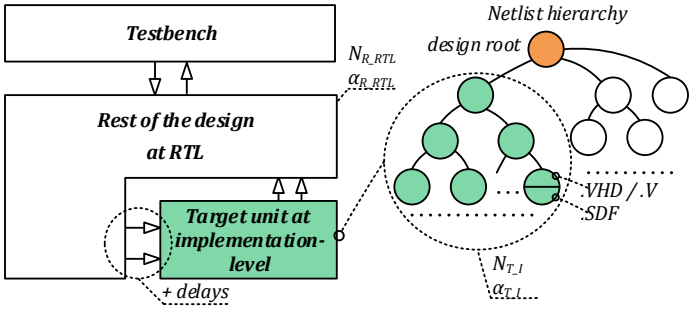


Figure 6.8: Mixed-level model comprising behavioural and implementation-level components

model by using a mixed-level HDL model. This optimization can be applied when dealing with a complex system, consisting of different modules or parts, so that experiments could be scheduled to inject faults in each particular module selectively. In such a way, the target module is simulated at the very detailed and slow, but highly accurate, implementation level, whereas the rest of the system is simulated at the fast RTL (see Figure 6.8).

The attainable simulation speed-up of using a mixed-level design directly depends on the weight of the implementation-level component within the entire DUT, as well as on the switching activity of its constituent elements, as it is expressed by an approximated speed-up model in Equation 6.3.

$$S_{MM} = \frac{N_{T\_I} \times \alpha_{T\_I} + N_{R\_I} \times \alpha_{R\_I}}{N_{T\_I} \times \alpha_{T\_I} + N_{R\_RTL} \times \alpha_{R\_RTL}} \tag{6.3}$$

where:

N_{T_I}, N_{R_I}, N_{R_RTL}, are the number of signals of the target unit (T) and the rest (R) of the design at the implementation level (I), and of the rest of the design at the RTL, respectively

α_{T_I}, α_{R_I}, α_{R_RTL}, are the switching activity rate of signals of the target unit and the rest of the design at the implementation-level, and the rest of the design at the RTL, respectively

Assuming that $\alpha_{T\_I} = \alpha_{R\_I} = \alpha_{R\_RTL}$ and $N_{R\_RTL} \ll N_{T\_I}$, then the speed-up factor of using a mixed-model will be $S_{MM} \approx 1 + {}^{N_{R\_I}}/_{N_{T\_I}}$. Accordingly,

the smaller is the size of the target unit at the implementation-level model with respect to the rest of the design, the higher is the attainable speed-up.

Building this mixed model is not as straightforward as it could seem. First, a hierarchical netlist should be obtained from the source RTL model by enabling the *Keep Hierarchy* synthesis option. Although the hierarchy tree is retained all interfaces are flattened (buses transformed into several individual signals), so the interfaces between the implementation-level model and the RTL model should be reconnected. Furthermore, the implementation model includes the actual timing behaviour of the target device described in a Standard Delay Format (SDF) file. As RTL includes no timing behaviour, the incoming signals from RTL to the implementation-level model should be delayed. To prevent the timing violations, these delays should at least satisfy the setup/hold times of input registers on the implementation side, which can be extracted from SDF of hierarchical netlist. Depending on the particular design hierarchy, this may also require to take into account the delays of combinational paths before and after the RTL-implementation connection point.

An alternative approach of exploiting RT-level models for speeding-up of SBFI experiments consists in distributing the faultload between RTL and implementation levels. In such a way, the logic that is represented at RT-level with enough detail, namely registers which have not been optimized during the synthesis and implementation, can be targeted at the RT-level, while the rest of logic is targeted at the implementation level.

In practice, this can be achieved through the inter-level model matching procedure, which automates the selection of proper fault targets at different levels of HDL description. It traces the macrocells from lower-level model, the implementation-level in our case, back to source logic structures at higher levels, the RTL in this case. The basic idea is to determine which fault targets can be covered at higher description levels, thus with a lower simulation (temporal) cost. Matching RTL and implementation-level descriptions means determining which RTL signals represent the actual set of registers in the implementation-level model generated by EDA tools. Fig. 6.9 illustrates this matching process.

First, both (RTL and implementation-level) models are parsed to extract the set of logic primitives existing in their respective (RTL and IMPL) design trees. This parsing process, defined as *Phase_1: Model Parsing* in the figure, is very simple and relies on the identification of signals of embedded and custom types at RTL, and macrocell's basic types specified by vendor-specific libraries (describing flip-flops, look-up tables, etc.) at implementation-level. It is worth noting that the signals represented by complex data types at RTL (arrays and structures)

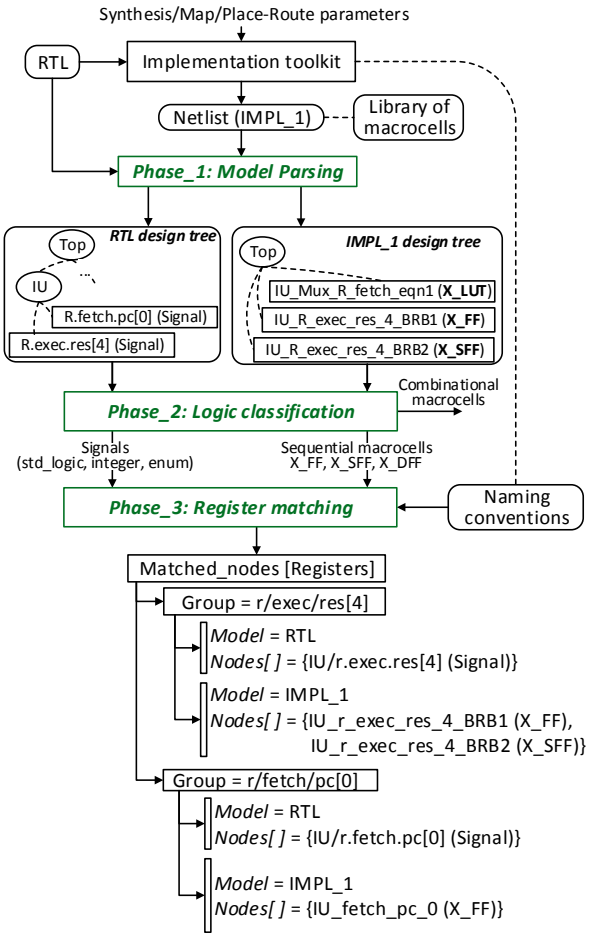


Figure 6.9: A method to match the sequential logic between the behavioural (RTL) and implementation levels of HDL description

are parsed up to their simplest primitives representing single bits, integers and enums. This parsing results in two different design trees, called the RTL and implementation design trees, respectively.

In the second phase of the approach, named *Logic classification*, the information contained in both trees is filtered in order to focus the matching process only on those RTL signals that can represent registers and those macrocells at implementation-level that are related to sequential logic. This is why only signals of types *std_logic, integer or enum* are considered from the RTL design tree,

while only sequential macrocells (representing flip-flops of any type, i.e. *X_FF*, *X_SFF*, *X_DFF*) remain after filtering in the implementation design tree.

Then, the approach only needs to establish a correspondence between the information remaining in the RTL and implementation design trees. In order to cope with such matching, the use of the naming conventions is essential to assign a generic internal name to each logic primitive, and subsequently to decide which RTL signal is the source of a certain group of implementation-level macrocells. The naming conventions are usually explicitly declared by EDA vendors. These conventions are essential in order to ensure the proper structure of the various intermediate models generated by each tool in the EDA toolkit under use. Thanks to such conventions, it is possible, for instance, to declare a hierarchy of components, or specify when a register has been duplicated. Consider the RTL example provided in Fig. 6.9 in the expression *IU/r.exec.res[4]*. This expression represents the bit 4 of register *r.exec.res* of the Instruction Unit (*IU*). However, this bit is duplicated in the implementation level, where bits *IU_r_exec_res_4_BRB1* and *IU_r_exec_res_4_BRB2* can be found. In addition, this encoding reflects the fact that such duplication has been carried out using a *Backward Register Balancing (BRB)* technique for retiming, as shown in Fig. 3.5. In this particular case, the naming conventions under consideration are those defined by Xilinx for its XST synthesizer, which are defined in the Chapter 16 of [187].

The result of all this process is a set of matched nodes. Each element of the set represents a group including two tuples (model, nodes). The name of the group reflects the name of the particular register it represents. As the tuple (model, nodes) is concerned, it reflects an RTL signal and locates the signal in the model. The same applies to the second tuple, but in this case, it reflects an implementation-level macrocell or a set of them. It is worth mentioning that, in most of the cases, the multiplicity of the relationship in a matching group is expected to be 1 (RTL) to 1 (implementation-level), but it may also be 1 (RTL) to N (implementation) in case of, for instance, register duplication. In case of elimination of registers (1 to 0 relation), the matching group will not be created, since those registers do not exist in final implementations and thus, they are not relevant from the perspective of fault injection.

This is how the proposed approach exploits the idea of multi-level fault injection: inject faults at RTL signals when targeting sequential macrocells with an entry in the resulting register matching; otherwise, inject faults at implementation-level. The match rate (percentage of matched registers) and the multiplicity of each relationship will determine the speed-up benefit in each practical case.

6.3.2 Simulation-based and FPGA-based checkpointing

The execution of each injection run encompasses several steps: i) DUT initialization, e.g. execution of a bootloader that initializes the register file, memory controllers, UARTs, timers, and the like; ii) workload initialization including the execution of an operating system or various applications before starting the workload execution; iii) workload execution, where the selected workload is actually run and fault injection takes place at a certain instant of time, and iv) readout of workload processing results. Although the two first initialization stages may not be required for many different designs, they are fairly common when considering microprocessors as target designs. All the stages preceding the workload execution may be attributed to warm-up overhead, required to bring the DUT into an initial state, as denoted by T_{init} in Fig.6.10. Likewise, the part of the workload execution interval preceding the fault injection instant T_{INJ}, can be attributed to the runtime overhead, since the actual observation of fault effects starts only after T_{INJ}.

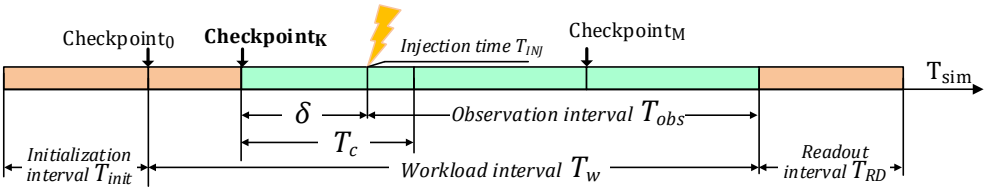


Figure 6.10: Restoring execution from clustering checkpoint to speed-up the fault injection experiments

Reducing the aforementioned warm-up and runtime overheads by means of checkpointing may greatly improve the performance of SBFI/FFI experiments. Checkpoint is a dataset containing the DUT execution context at a certain time instant, which can be captured and recovered on demand by the SBFI/FFI environment. During the execution of the GoldenRun a set of M checkpoints is stored. The first one is captured (C_0) just before starting the execution of the workload (at the end of the initialization stages). The rest $M-1$ checkpoints split the execution time (T_w) into M sub-intervals of duration T_c, as depicted in Fig.6.10. Accordingly, each subsequent injection run is started by restoring the checkpoint C_K corresponding to the state of the system just previous to its fault injection time T_{INJ}.

Since the checkpoints may require a considerable amount of RAM/storage space, it is usually not feasible to create a checkpoint for each time instant of the scheduled fault list. Hence, this approach groups the experiments into M clusters in

such a way that all the experiments whose injection time fall into the same cluster will use the same checkpoint. The total execution time of each injection run comprises the observation interval T_{obs}, and the time difference δ between the injection time T_{INJ} and the timestamp of the corresponding (closest) clustering checkpoint C_K, as depicted in Fig.6.10.

The execution time of each injection run should also take into account the time overheads for restoring the checkpoint, which is platform-dependent. In simulation-based fault injection the checkpoint management relies on the embedded features of the used simulation tool. For instance, in ModelSim the simulation state can be saved at any time point (when simulation is paused) by means of the simulator command *[checkpoint ChekpointFile.sim]*, and recovered by command *[restore ChekpointFile.sim]*. It is worth noting that ModeSim captures and recovers only the internal objects such as model state, observation lists, etc. The simulated design thus should not use any external resource (e.g. open file descriptors) to allow the use of checkpointing. Our experiments show, that checkpoint recovery in ModelSim is as fast as resetting the simulation to its initial state (time 0), and even faster than loading the simulation state from scratch. Hence, checkpoint recovery doesn't induce any time overhead in the SBFI flow. Accordingly, the total speed-up, attainable through the use of checkpointing in a simulation-based fault injection campaign of N runs, can be quantified by dividing the total time that should be simulated in a non-optimized flow by the time resulting from the use of the presented checkpoint-assisted flow, as it is expressed by Equation.6.4.

$$S_{CC} = \frac{N \times (T_{init} + T_w + T_{rd})}{T_{init} + \sum_{i=0}^{N-1}(T_{obs\_i} + \delta_i + T_{rd})} \tag{6.4}$$

To estimate the attainable speed-up in practice it is necessary to assume the particular workload and faultload parameters. For instance, the model in Listing 6.2 can be used to estimate the speed-up in the case when fault injection instants are uniformly distributed along the workload time. Assuming the sampled number of injection runs N, workload time T_w, result readout time T_{rd}, and initialization time T_{init} proportional to T_w, it estimates the attainable speed-up factor SCC and the mean distance to the checkpoint δ. Fig.6.11 illustrates the results obtained under $N = 10000$, $T_w = 100.0$ ms, $T_{rd} = 0$ ms, increasing initialization time $T_{init} = 0/T_w/5T_w$, and increasing number of checkpoints ($M = 1 \rightarrow 50$).

```
import random
#models linking of checkpoints to uniformly distributed injection runs
#returns the estimated speed-up factor and mean distance to the checkpoint (delta)
#inputs: N      - number of injection runs
#        Tinit - duration of initialization interval
#        Tw    - duration of workload interval
#        Trd   - duration of result readout interval
#        M     - number of checkpoints
def Speedup(N, Tinit, Tw, Trd, M):
        #allocate M checkpoints within workload interval Tw
        Tc = Tw/M
        checkpoints = [Tinit + i*Tc for i in range(M)]
        #generate N injection instants, uniformly distributed within the workload
        Tinj = [random.uniform(Tinit, Tinit+Tw) for i in range(N)]
        #link injection instants to the closest checkpoint
        linked_cp = [ max([c for c in checkpoints if (t-c)>=0]) for t in Tinj ]
        #compute distance to checkpoint for each injection run
        delta = [ Tinj[i] - linked_cp[i] for i in range(N)]
        #compute observation time for each injection run
        Tobs = [Tinit+Tw-Tinj[i] for i in range(N)]
        #compute the speed-up factor
        SCC = (N*(Tinit + Tw)) / (Tinit + sum([delta[i]+Tobs[i]+Trd for i in range(N)]))
        return(SCC, sum(delta)/len(delta))

N       = 10000
Tw      = 100.0
Trd     = 0.0
Tinit   = 5*Tw
for M in range(1,50,2):
    SCC, mean_delta = Speedup(N, Tinit, Tw, Trd, M)
    print('{0} ; {1:.2f} ; {2:.2f}'.format(M, SCC, mean_delta))
```

Listing 6.2: Model for estimating the checkpointing-related speed-up under uniformly distributed injection time instants (executed by python)

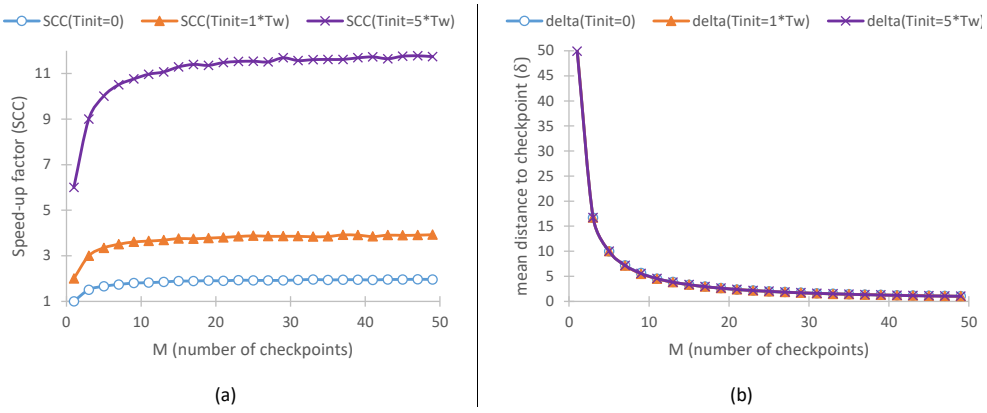


Figure 6.11: Speed-up factor (a) and mean distance to checkpoint (b), estimated under increasing number of checkpoints (1→50) and increasing duration of initialization interval

First of all, a significant speed-up is achieved by checkpointing of the initialization interval, especially under $T_{init} \gg T_w$. Analytically this can be seen from Equation 6.4 by assuming that all experiments start from the same checkpoint at the beginning of the workload interval ($M = 1$): $SCC = \frac{N \times (T_{init}+T_w)}{T_{init}+N \times T_w}$. Under the increasing number of clusters M the speed-up grows will slowdown, being

generally limited by a factor of two. Under $M = 20$ it reaches roughly 90% of its maximum value. Further increase of M provides a very small additional benefit, at a cost of significantly increasing number of checkpoints. As Fig.6.11-B shows, this is correlated with the mean distance to the closest checkpoint δ, which is rapidly reduced until M reaches a value of 20. Accordingly, in case of a uniform distribution of injection instants, splitting the workload interval into 20 clusters can be seen as a reasonable trade-off between attainable speed-up and the storage footprint.

FPGA-based frameworks usually lack of the instrumental support for checkpointing management. Thus, FFI tools themselves should be in charge of capturing and restoring the DUT state. Several works in the domain [67][92][143] provide basic instructions for capturing and recovering the DUT state in FPGA.

The general idea of FPGA checkpointing concerns with making a snapshot of state elements, such as CLB registers, user memories (BRAM and LUTRAM), and registers within the DSP and BRAM slices. Most state elements in the Xilinx design flow are attributed to changeable CM cells, that can be localized by means of the logic allocation file (*.ll). At the beginning of each time cluster within the workload interval the DUT clocking should be paused, and the capture command should be issued to the FPGA configuration controller, in order to copy the state of all FPGA registers into the corresponding INIT/readback cells [67]. All the data frames with state elements can be subsequently read back from the configuration memory. Finally, the captured frames, representing the DUT state at the time T_K, can be enclosed into the configuration sequence, and the latter can be annotated with a time stamp T_K.

The resulting partial bitstream can be directly supplied to the PCAP/ICAP reconfiguration interface, in order to recover the DUT state. An additional bitstream modification could be required to properly recover the BRAM content, as it is pointed out by [67]. This bitstream can be maintained as dataset in the RAM, or stored into a file on the external storage (e.g. SD card).

The collected set of checkpoints can be used to speed-up FFI experiments. However, it should be noted that checkpoint recovery time may exceed the time required to simply reset the DUT and re-run it from scratch until the fault injection time. Accordingly, before injecting faults into the target system, the checkpoint recovery time T_{REC} should be measured, and at the beginning of each injection run the injection time instant T_{INJ} should be compared to the checkpoint recov-

ery time, in order to determine whether checkpoint recovery could provide any speed-up benefit with respect to resetting and warming-up the DUT from scratch.

$$S_{CC\_FFI} = \frac{N \times (T_{init} + T_w + T_{rd})}{(N - N_{REC}) \times (T_{init} + T_w + T_{rd}) + \sum_{i=0}^{N_{REC}-1} (T_{obs\_i} + \delta_i + T_{REC} + T_{rd})} \quad (6.5)$$

Accordingly, if denoting the number of times the DUT is recovered from the checkpoint by N_{REC}, the Equation 6.5 can be used to estimate the FFI speed-up attainable by means of checkpointing. By measuring the T_{REC} for a particular DUT and taking it into account in the model in Listing 6.2, the attainable speed-up can be estimated similarly to the SBFI case.

It should be noted, however, that implementing FPGA checkpointing in practice should address some additional problems, like those related to the recovery of those registers that do not appear in the bitstream (e.g. DSP registers and BRAM pipeline registers), synchronization in case of multiple clock domains, as well as ensuring safe interruption of tasks and transactions (atomic IO transactions) [12]. In addition to all this, the main problem is recovering the state of elements that are outside the FPGA fabric, like shared RAM memory, ARM cores, etc. These problems are out of the scope of this thesis, thus this section should not be treated as a complete solution for FPGA checkpointing, but rather as a general model to analyse the potential FFI speed-up.

6.4 Discussion

The existence of various state of the art speed-up solutions (discussed in Section 3.5), may lead the reader to the question of how they are related and compared to the proposed techniques. Their comparison from the viewpoint of attainable speed-up benefit would require to consider particular benchmark design and injection platform. This analysis is accomplished in Chapter 9 (experimental evaluation). Nevertheless, it is worth mentioning several general remarks regarding the usefulness of the proposals within the context of existing techniques.

The profiling may be compared with the workload-dependent fault collapsing approach described in [24]. Both solutions allow the identification of ineffective faults, whose effects (masking) can be determined without fault injection. However, they solve the problem from different perspectives. The approach in [24] identifies ineffective faults along with fault equivalences by analysing the precedence of read and write operations in RT-level memories. Whereas the profiling approach proposed in this chapter identifies ineffective faults at the implemen-

tation level, specifically targeting the combinational logic implemented on LUTs. Despite a LUT can be seen as a read-only memory element, it is in fact an FPGA-specific combinational primitive. To detect the accesses to the LUT-related CM cells it takes into account a number of implementation and technology-specific factors, including (i) mapping of LUT cells to configuration memory, (ii) LUT combining, (iii) filtering or propagation of glitches. Furthermore, the profiling approach not only removes the ineffective fault configurations (inactive CM cells) from the fault list, but also prioritizes the criticality of the rest of faults through the measured activity time. Section 9.2 will provide the experimental support for the hypothesis that the criticality of CM cell increases as its activity time does.

The iterative statistical fault injection approach is, in fact, a further development of the standard statistical injection formalized in [100]. The main difference of these techniques is the paradigm of scheduling the injection runs. The classic approach makes a conservative assumption regarding the estimated metrics (i.e. the probability of failure that maximizes the sample size), and after samples this (overestimated) number of faults. The iterative approach on the contrary dynamically extends the sample until reducing the error margin to the required threshold. Replacing the standard statistical approach by the iterative one, would provide an additional speed-up as quantified by Equation 6.1.

The proposed mixed and multi-level injection approaches can be compared to the multi-level injection presented in [99]. The latter adapted the multi-level injection paradigm to simulate the effects of faults in the finite state machines (FSM). The proposed approach on the one hand, adapted the multi-level paradigm to any type of registers; on the other hand an alternative mixed-level optimization allows to speed-up the fault simulation for the rest of implementation primitives as well. Another important difference is that the former relies on the use of mutants to inject the faults, while the proposal is based on the use of simulator commands, being low-intrusive. Additionally, the proposed approach allows quantification of attainable speed-up on the basis of model properties.

The checkpointing proposal, although relying on the use of a well-known concept [129], remains interesting from a practical viewpoint for SBFI and FFI. Particularly, the proposal defines a model for estimating the attainable speed-up under different workload parameters, and can be used to find a trade-off between the number of clustering checkpoints (storage footprint) and the speed-up gain.

Finally, most speed-up techniques are rather complementary than alternative, i.e. they can be used all at the same time for obtaining a cumulative speed-up gain. For instance, in addition to the techniques proposed in this chapter, multiprocessing can be used to speed-up the experiments even further.

6.5 Conclusions

Enabling the dependability assessment of implementation-level models requires to improve as much as possible the performance of fault injection experiments. On the one hand, a high number of logic primitives, which can be targeted within the detailed models, combined with complex workloads, leads to huge fault lists. On the other hand, the high simulation effort of implementation-level models leads to very long run times. This chapter has proposed four techniques to improve in practice the performance of SBFI/FFI experiments from these two perspectives, namely: (i) reducing list of faults to consider while keeping the statistical representativeness of derived metrics, and (ii) accelerating each individual injection run. The benefits and limitations of proposed techniques are summarized below.

First, the fault list for implementation-level models can be reduced by profiling the switching activity of it's constituent macrocells. Particularly, when targeting the models in the basis of FPGA libraries, the profiling of LUTs allows to locate those cells of underlying configuration memory which remain inactive for a given workload. All such inactive CM cells can be omitted from the fault list, while the rest of them can be prioritized in the fault list according to their activity time. The attainable performance gain of this optimization is directly proportional to the percentage of inactive (filtered-out) CM bits within the whole set of essential bits. The main limitation of the proposed profiling technique is that it currently targets only LUT-specific CM cells. Despite LUTs cover the major part of the combinational logic in an FPGA, they account for less than 30% of CM cells. Thus a future work should extend this approach to a wider set of components (MUXes, DSPs, routing resources, etc.). Furthermore, to provide the actual SFBI/FFI performance gain, the profiling itself should be implemented in a very efficient way, so that it would require less time than the filtered-out injection runs.

After filtering the fault list through profiling, it may still remain too huge for exhaustive experimentation. In this case, the iterative statistical sampling is employed to reduce it to an affordable size, by assuming a certain error margin in the derived metrics. In comparison to the standard sampling approach, it leads to a much smaller sample size. The gain may be particularly high when the estimated metric is far from the conservative assumption, which is quite common for critical designs, whose estimated failure rates are expected to be low. The iterative statistical approach by itself has the same limitations as the standard statistical fault injection: it is applicable only for the estimation of aggregate dependability metrics (expressed as the sample proportion), such as failure rate. As already mentioned, the iterative sampling is expected to provide a significant

benefit with respect to the conservative approach only when the estimated metrics are very low or very high (far-distant from 50%).

Speeding-up of individual injection runs encompasses the optimization of model complexity and the reduction of simulation time overheads. The complexity of the model is reduced by mixed-level and multi-level injection techniques. Mixed-level technique integrates the targeted module at implementation level into RTL model of the rest of design, providing a speed-up gain roughly equal to the proportion of targeted unit in the entire design. The multi-level model traces the sequential logic from implementation level back to source RTL structures in order to distribute the faultload between different representation levels. In such a way, the simulation cost is significantly reduced, while most synthesis and implementation logic optimizations are taken into account. Both mixed-level and multi-level techniques are limited to SBFI experiments.

Finally, the clustering checkpointing technique provides a significant speed-up in the case of long DUT initialization time. In the absence of initialization time, the attainable speed-up is strongly conditioned by the faultload. Particularly, in case of uniform distribution of faults along the workload, the speed-up is limited by a factor of two, and 90% of this speed-up gain is achieved by splitting the workload time in 20 clusters. The rest 10% gain is achieved by a significant (generally unlimited) increase in the number of checkpoints (storage footprint). Furthermore, in case of FFI the checkpointing is efficient only under long workloads, when the checkpoint recovery becomes faster that reinitializing and rerunning the DUT from scratch.

Chapter 7

Contributions in Dependability-aware Design Space Exploration

Optimal tuning of design parameters (offered by EDA tools and IP cores) concerns with the design space exploration (DSE), which is a very resource-intensive process in the context of dependability. Its feasibility is determined by two main factors: i) effectiveness of the selected exploration strategy, measured as the total number of configurations that should be sampled from the design space, and ii) fault injection effort required to evaluate the dependability of each sampled configuration. This chapter proposed two solutions for the time-efficient dependability-aware DSE. Section 7.1 introduces the proposals in the context of exiting DSE approaches. After that, Section 7.2 describes a DSE approach that reduces the experimentation effort by minimizing the number of sampled configurations by means of design of experiments (DoE) methodology. Section 7.3 then proposes an iterative selection technique that allows to efficiently adapt the genetic algorithms (GA) for dependability-aware DSE. Finally, Section 7.4 summarizes the advantages and limitations of the proposed approaches.

7.1 Introduction

Design space exploration strategies aim at determining the (sub)optimal solutions in the design space by evaluating the minimum possible number of individuals (configurations of design parameters). According to [128], existing DSE strategies are based on i) heuristics, e.g. simulated annealing; ii) genetic algorithms (GA), iii) statistical techniques with domain knowledge, e.g. techniques relying on a Markov Decision Process, and iv) statistical techniques without domain knowledge, like those usually based on the design of experiments. The efficiency of each strategy is context-dependent.

Simulated annealing (SA) and genetic algorithms (GA) are two very popular optimization approaches, relying on iterative convergence to (sub)optimal solution from the initial random points in the design space. The former (SA) moves through the design space in the direction which improves the solution (hill-climbing), combining it with random probabilistic moves in order to prevent trapping in local optima [1]. The latter (GA) converges towards the optimal solution by iteratively applying the selection, crossover and mutation operators to the population of competing individuals (configurations). The more iterations are carried out, the closer to the global optimum become the selected individuals. The usage of conventional GA for the DSE problems within the HW design flow has been formalized in [49], which has focused on optimization of VLSI cell placement. DSE proposals dealing with multicriteria optimization often combine the conventional GA with Pareto optimization, in order to determine a set of non-dominated solutions, as for instance it has been proposed in [71] to study a trade-off between the performance and area results of high-level synthesis.

GA-based solutions often calibrate the selection, crossover and mutation processes in order accelerate the convergence towards the global optima. Nevertheless, their experimentation effort still may be prohibitive in dependability domain, since the required number of GA iterations (evaluations) is a-priori unknown due to the stochastic nature of involved processes. Even though the dependability evaluation of each individual can be accelerated by adopting the techniques in Chapter 6, the resulting performance still may be insufficient in the context of dependability-driven DSE.

It has been shown in [17] that the usage of domain knowledge to setup the DSE as a discrete-space Markov Decision Process reduces drastically the number of required evaluations. However, when considering the problem of tuning the EDA toolkits, those strategies requiring domain knowledge become less useful since the impact of available parameters on the dependability of resulting implementations is rarely known a-priori.

Statistical strategies without domain knowledge are much more flexible. They rely on the Design of Experiments (DoE) methodology to sample the design space in a statistically representative way, and on its basis quantify the contribution of each optimized parameter towards the implementation goals. Existing proposals in this field usually focus of PPA attributes [146], neglecting the impact of optimized parameters on dependability. Adaptation of these strategies to the dependability context requires to care about the pertinence of selected DoE technique [114]. The selected DoE should minimize as much as possible the number of sampled individuals, while maximizing the ability to observe all significant effects of tuned design parameters.

Being well-balanced and orthogonal, fractional factorial designs are the most rigorous DoE today. They allow to estimate the parameters' main effects, as well as their interactions without confounding the estimators. In practice the lowest-resolution factorial designs are used for *screening* of main effects, since observation of interactions significantly increases the design (sample) size: the higher is the order of interactions – the bigger the sample required. An important limitation of orthogonal designs is that they are not applicable to irregular design spaces, i.e. those containing incompatible configuration of parameters. Any configuration that is inaccessible for evaluation invalidates the design orthogonality and its statistical representativeness. Furthermore, orthogonal designs in practice are rarely inferred (by off-the shelf statistical tools like Matlab) for parameters with more than two treatment levels, while reasonable quantification of parameters at two levels becomes less feasible in modern design flow.

This chapter addresses the aforementioned limitations of DoE-based and GA-based DSE, which they expose in the dependability context. First of all, Section 7.2.2 defines a dependability-aware DSE approach based on fractional factorial designs, which is the most rigorous exploration approach for the regular design spaces with two-level factors. After that, Section 7.2.3 proposes a DoE-based approach which handles irregular designs spaces with multilevel parameters by means of more flexible (but less rigorous) *D-optimal designs*. Thanks to the augmentation capability of D-optimal deigns, the DSE is deployed as iterative process starting from the smallest possible design, which after checking the quality of derived results is refined until satisfying the predefined quality indicators. Finally, Section 7.3 proposes an iterative dependability-driven selection algorithm, which further develops the idea of iterative statistical fault injection in application to the GA-based DSE. By dynamically adapting the confidence intervals of dependability estimates to the actual diversity in population, it allows to confidently select the best (most robust) individuals from the population by smallest possible number of injection runs.

7.2 DSE based on the design of experiments

This section proposes two DoE-based DSE approaches aiming at dependability-aware optimization of EDA/IP parameters. The first approach relies on fractional factorial (orthogonal) designs, and handles regular design spaces with two-level factors (e.g. LUT combining: enable/disable, optimization effort: high/low, etc.). Another approach relies on the D-optimal designs (which can be dynamically repaired and augmented, while preserving their optimality), and allows to explore the irregular design spaces with multilevel parameters by minimum possible number of experimental trials. Both approaches are supported by the DAVOS toolkit, which is described in detail in the next Chapter 8. Before describing the approaches, this section summarizes the background on the used statistical techniques.

7.2.1 Background on design of experiments and its statistical analysis

A DSE experiment that exhaustively evaluates each individual in the design space is described by a *full factorial design*, in which every setting of every factor appears with every setting of every other factor. *Factors* (X_i) are those process inputs (in this case EDA/IP parameters) that are deliberately changed to observe their effect on the *response variables* (V_j) (resulting PPAD attributes). As previously discussed, estimating the response variables even for a single combination of factors settings is a very time-consuming problem in dependability domain, and the exploration of a full factorial design becomes unfeasible with increasing number of factors.

A selected subset of factors' settings form a *fractional factorial design*, which considerably reduces the design space and, thus, the time required to estimate the effect of those factors. The fractional factorial design of experiments exploits two main principles [172] [50] to reduce the design space without affecting the validity of drawn conclusions: i) *sparsity of effects*, which states that the number of relatively important effects and interactions in a factorial design is small, and ii) *hierarchical ordering*, which states that lower order effects are more likely to be important than higher order effects, main effects are more likely to be important than interactions, and effects of the same order are equally likely to be important. The combinations of factors settings should be carefully chosen so the design is both *balanced* (the combination of factors settings for any group of factors have the same number of observations) and *orthogonal* (the effects of any factor balance out (sum to zero) across the effects of the other factors) [123]. Furthermore, it is necessary to take into account the *resolution* of the design, which

describes the degree to which estimated main effects are aliased (or confounded) with estimated low-level interactions [83]. The resolution of a design is one more than the smallest order interaction that some main effect is confounded with. Accordingly, to precisely determine the contribution of each factor towards the implementation goals, a fractional factorial design with resolution IV should be considered, so main effects are not confounded among one another nor with any two-factor interaction.

Fractional factorial designs are commonly denoted as I_R^{K-P}, where I is the number of possible settings (*levels*) of each factor, K is the number of factors, R is the resolution of the design, and $1/I^P$ is the size the fraction with respect to a full factorial design. According to [113], 'two-level designs ($I = 2$) should be the cornerstone of industrial experimentation for product and process development, troubleshooting, and improvement.' In fact, many EDA parameters present a dual nature ('Yes/No', 'True/False', 'Enable/Disable'), whereas some others can be modelled as a maximum and minimum threshold ('0%/100%'). Nevertheless, there exist some situations in which it could be necessary to consider factors at more than two levels ('Normal/High/Fast', for instance), at the cost of an even larger design space. If all factors are quantitative, then two-level designs with centre points should be employed [113]. Otherwise, existing algorithms, like the one proposed in [57], can be used to generate the required mixed-level fractional factorial design. In practice, though, orthogonal multilevel designs are not generated by off-the-shelf statistical suites (like Matlab), but are subject of custom implementation for a particular use-case.

The *analysis of variance* (ANOVA) is used to determine whether there are significant differences between the means of two or more independent (unrelated) groups [123]. The null hypothesis to be tested is that there is no difference in the population means of the different levels of a given factor. This procedure splits the total variation in the response variable into a part due to random errors (*sum of squares of error*) and a part due to changes in the levels of the selected factor (*sum of squares of treatment*). The degrees of freedom are split in the same way, and are used to compute the mean square for treatments and errors. If the null hypothesis is true, then both mean squares estimate the same quantity and its ratio (*F-test*) should be close to 1. If the probability (*p-value*) of an F value being greater than or equal to the observed value is less than or equal to the significance level (α, usually 0.05), then the null hypothesis is rejected, meaning that setting a given factor to one level or another has a statistically significant impact in the response variable observed.

Regression analysis is a statistical technique that can be applied in this context to estimate the parameters of an equation relating a particular response variable

to a set of factors. In such a way, it is possible to predict the value of the response variables for the whole design space even when only just a fraction has been actually explored. A linear regression model assumes that the relationship between the response variable and the factors is linear ($V = \beta_0 + \beta_1 X_1 + \ldots + \beta_n X_n$) [58]. The *coefficient of determination* (R^2) explains the proportion of the variance in the response variable that can be explained by the factors, i.e. how well the model fits the data (values for the response variable obtained through the fractional factorial design). In case the linear regression model does not adequately fit the data, then a more complex generalized linear regression model can be used. The generalized linear regression can be used to predict responses for both variables with discrete distributions or which are not linearly related to the predictors.

7.2.2 Exploring regular design spaces by means of fractional factorial designs

All the previously described statistical techniques are the cornerstones of the proposed method, whose flowchart is depicted in Figure 7.1. It comprises four main steps: (i) sampling the configurations from the design space through DoE, (ii) evaluation of sampled configurations, (iii) statistical analysis, and (iv) multiple response variable optimization.

At the initial (DoE) step each EDA/IP parameter is denoted as a factor X_i with I treatment levels. As previously explained, this approach focuses on two level fractional factorial designs ($I = 2$) since they can be generated and analyzed by standard off-the-shelf tools, and present an affordable size. The factors along with their selected treatment levels are specified by the designer within the XML-formatted DAVOS configuration file (as detailed in the next chapter 8).

To estimate the precise contribution of each factor towards PPAD goals, the resolution of the experiment should be at least IV (the effects of the main factors are not confounded with any two-factor interactions). Resulting fractional factorial design for K two-level factors with resolution IV is denoted as 2_{IV}^{K-P}, and consists of a $K \times 2^{K-P}$ table where each cell ($L_{c,i}$) defines the setting of each factor (X_i) at the given treatment combination (c). Predefined designs for up to 11 factors can be found in [29] and [113], whereas MATLAB's Statistics and Machine Learning Toolbox [158] can be used to generate two-level fractional factorial designs with 2^{K-P} experiments and a given resolution using the Franklin-Bailey algorithm. For instance, a 2_{IV}^{12-7} design can be obtained by running the MATLAB code from Listing 7.1. The resulting table is stored in a csv-formatted file. This step of

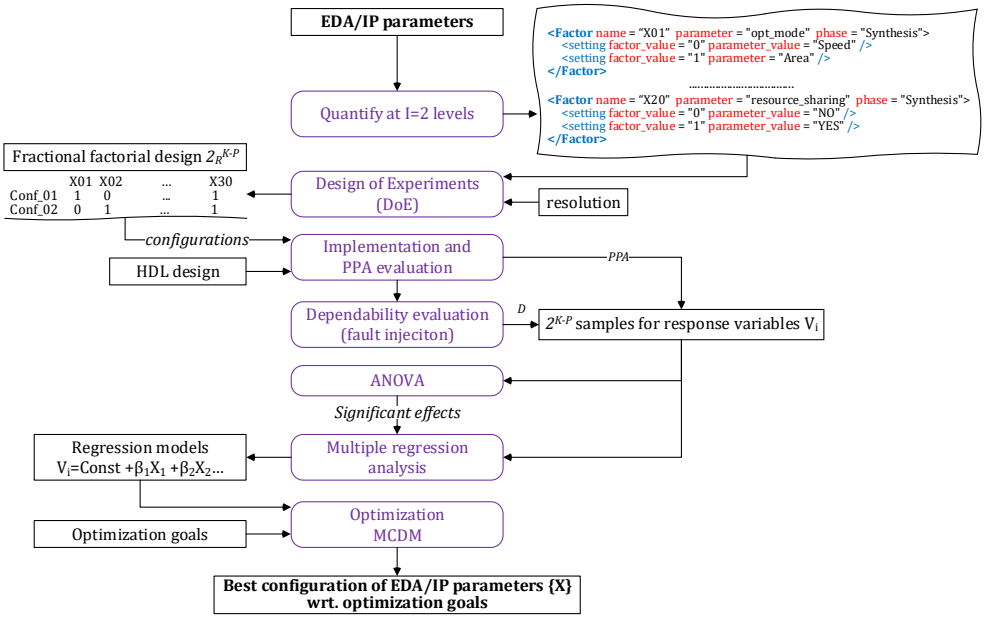


Figure 7.1: DSE flow to optimize EDA/IP parameters though the fractional factorial design of experiments

DSE flow is automated by the DoE tool, which forms a part of DAVOS decision support core (see Fig.8.1).

```
factors      = 'a   b   c   d   e   f   g   h   i   j   k   l';
aliases      = {'X1' 'X2' 'X3' 'X4' 'X5' 'X6' 'X7' 'X8' 'X9' 'X10' 'X11' 'X12'};
Resolution   = 4;   %main effect and two-factor interactions without confounding
FractionSize = 5;   %design of 32 configurations

generators = fracfactgen(factors, FractionSize, Resolution);
[dff, confounding] = fracfact(generators);

writetable(cell2table(num2cell(dff), 'VariableNames', aliases), ...
           'FracFactDesign.csv',     'WriteRowNames', true);
```

Listing 7.1: Fractional factorial design generation with MATLAB

On the second (evaluation) step each sampled configuration from the factorial design is used to configure the IP cores within the HDL design, and EDA tools (synthesis, map, place-route) used in the implementation flow. Each EDA tool executes a parameterized shell/TCL script, whose parameters are replaced by the actual values from the input configuration. The implementation process may

be iterative in case of optimizing maximum achievable clock frequency (performance attribute) through the gradual strengthening of timing constraints. After implementing each design configuration, its PPA attributes are evaluated using the vendor-specific EDA tools: (i) performance is estimated through static timing analysis, (ii) area is retrieved from utilization reports in terms of the number of used BELs, (iii) power consumption is estimated by the power analysis tools with the prior simulation of switching activity. Finally, each implemented configuration is supplied to either SBFI or FFI tool for evaluation of dependability attributes. This evaluation step is automated by the DAVOS PPAD evaluation engine, described in Section 8.4.2.

```
Factors = {'X01', 'X02', 'X03', 'X04', 'X05', 'X06', 'X07', 'X08', 'X09', 'X10', 'X11', 'X12'};
RespVar = 'FIT'
sample = readtable('Sample.csv');
Threshold = 0.9

%ANOVA
[prob_F, detail]= anovan(table2array(sample(:, RespVar)), table2array(sample(:,Factors)), ...
                         'model','linear',    'varnames', Factors);

%select factors with statistically significant effects
SignFactors = {}
for i = 1:(size(prob_F, 1))
    if(prob_F(i) < 0.05)
        SignFactors{end+1} = Factors{i}
    end
end

%infer generalized linear regression model (significant factors)
Model= fitglm(sample(:,[SignFactors, RespVar]), 'linear' , 'ResponseVar', RespVar, ...
              'CategoricalVars', SignFactors, 'Distribution', 'normal')

%try to improve the determination coefficient by including interactions (all factors)
if(Model.Rsquared.Adjusted < Threshold)
    Model= stepwiseglm(sample(:,[Factors, RespVar]), 'interactions' , 'ResponseVar', RespVar, ...
                       'CategoricalVars', Factors, 'Distribution', 'normal')
end

%export resulting model to csv
writetable(Model.Coefficients,'Result.csv','WriteRowNames',true);
```

Listing 7.2: N-way ANOVA to identify the significant effects and regression modeling to relate the response variables to the factors

On the third (analysis) step the collected sample is used for ANOVA analysis and for regression modeling. The ANOVA is used to determine whether each considered factor X_i statistically significantly contributes to each PPAD response variable V. As previously stated, this is the case when the obtained p-value is below the significance level (usually 0.05). Those p-values can be easily computed by the MATLAB's Statistics and Machine Learning Toolbox [158], by using the *anovan()* function. By comparing the p-value of each factor to the significance level (0.05), the statistically significant factors' effects can be selected, as exemplified in Listing 7.2.

Subsequently, a generalized linear regression model is inferred, which allows to predict the value of the response variables for any setting of EDA/IP parameters. The eligible distribution is *'Poisson'* for discrete response variables (utilization of BELs, number of critical bits), or *'Normal'/'gamma'/'inverse gaussian'* for continuous variables (frequency, consumption, failure rate). It is usually worth to consider only the significant factors (previously determined by ANOVA) at regression modeling. Otherwise (when all factors are considered), the statistical significance of the obtained model terms should also be assessed, by checking the p-value to be less than the significance level (usually 0.05). Likewise, the coefficient of determination R^2 denotes the proportion of the variance in the response variable that can be explained by the factors. When the quality of inferred model is considered insufficient to predict the responses (an R^2 threshold should be defined), a more complex model is computed instead, which takes into account the interaction terms (the model type is changed from 'linear' to 'interactions'). The used *stepwiseglm()* function iteratively evaluates the benefit of appending the terms to the model, and keeps them only when they improve the determination coefficient of resulting model.

Finally, on the optimization step, the inferred regression models are used to compute the expected PPAD attributes for the whole set of 2^K possible configuration of EDA parameters. This enables to determine the best configurations of EDA parameters attending to the given implementation goals. In the particular case of implementing a design on FPGA, those implementation goals are usually conflicting, as for instance increasing the clock frequency is likely to increase also the dynamic power consumption. In such cases, when there is no information about the particular preferences of the designer, it is sufficient to provide the Pareto optimal set (or Pareto frontier). It consists of all those configurations in which it is impossible to improve a response variable without making worse another one. As no subjective information is provided, all of those configurations are considered a good solution towards optimizing the implementation goals.

However, if the designer formalizes the preferences about which implementation goal is more important than other (usually in the form of weights), then it is possible to estimate the best configuration of EDA parameters according to different multi-criteria decision making (MCDM) methods. Although several different MCDM methods exists, the simplest Weighted Sum Method (WSM) [160] can be generally used to combine the different response variables into a single score, accounting for the quality of the implementation according to the designer's goal. It must be noted that, in case of dealing with response variables expressed in different units, it is necessary to normalize (usually between 0 and 1) the predicted values prior to computing the WSM. In this case, Equation 7.1 can be used to

normalize predicted response variables according to whether they should be interpreted as *the-higher-the-better* values, like clock frequency, or *the-lower-the-better* ones, like power consumption. Equation 7.2 shows how to compute the final score S according to the *weights* (ω) defined by the designer for each predicted response variable after normalization ($V^{*'}$).

$$V^{*'} = \begin{cases} V^*/V^*_{MAX}, & \text{if the-higher-the-better response variable} \\ V^*_{MIN}/V^*, & \text{otherwise} \end{cases} \tag{7.1}$$

where:

V^*_{MAX}**:** maximum V^* across all configurations

V^*_{MIN}**:** minimum V^* across all configurations

$$S = \sum_{i=1}^{M} \omega_i \times V_i^{*'} \tag{7.2}$$

where:

M**:** number of response variables

The configuration obtaining the highest score will be that optimizing the implementation goal according to the designer's preferences.

7.2.3 Exploring irregular design spaces through iterative refinement of D-optimal designs

The valuable feature of orthogonal experimental designs is that they allow estimation of main effects and factor's interactions without confounding. On the other hand, they also expose two limitations. First of all, the tuning of EDA/IP parameters often deals with irregular design spaces, that contain incompatible combinations of parameters (which are inaccessible for evaluation). Encountering these combinations within the fractional factorial designs invalidates its balance and orthogonality, degrading its statistical representativeness. Second, parameters in practice often can't be reasonably quantified at two levels. Despite the existence of proposals for generation of fractional factorial designs for multilevel parameters [57], these algorithms are non-standard and are not supported by off-the-shelf statistical suites. Furthermore, resulting designs still may be quite huge,

thus exceeding the experimentation time budgets (especially in dependability-aware context).

Optimal designs are suitable to handle these challenges. In general, this type of designs requires less number of experiments than non-optimal ones to estimate the same set of parameters with the same precision. From a statistical viewpoint, parameters are estimated without bias and with minimum variance with respect to a selected statistical criterion, which is always related to the variance-matrix of the estimator. For instance, *D-optimal designs* maximize the determinant of the variance-matrix defined by selected experiments. From a practical viewpoint, this means that D-optimal designs select experiments with the goal of maximizing the coverage of the design space under exploration. This is a feature of the utmost interest when the experimentation time is limited and the sample cannot be as big as one may want. In the particular case of EDA/IP tuning, *d-optimal designs* are well suited for i) the consideration of EDA toolkits offering multi-level parameters, ii) the replacement of invalid configurations by new (valid) ones without affecting the statistical significance of the ongoing experimentation and, when necessary, iii) the iterative improvement of the statistical representativeness of estimators on the basis of adding new experiments to already executed ones. Further details about the d-optimality of optimal designs can be found in [60].

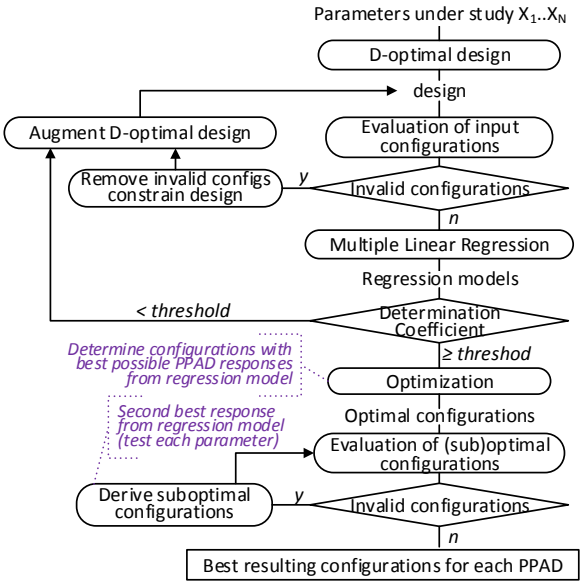


Figure 7.2: Iterative D-optimal design-based DSE

The flowchart of proposed iterative methodology is depicted in Fig.7.2. Similarly to the previous approach, the considered parameters are denoted as factors $X_1 ... X_N$ in the factorial design space, but this time they can be quantified at any number of treatment levels. The degrees of freedom of each parameter is one less that its number of treatment levels, e.g. a parameter with 4 treatment levels has 3 degrees of freedom.

The D-optimal design is generated attending to the parameters offered by the toolkit under evaluation and their degrees of freedom. The minimal size of a design is the sum of degrees of freedom of all the parameters, plus one. If the EDA toolkit offers, for instance, 4 parameters, two of them with 2 degrees of freedom and the other two with 3, then the design must have, at least, 11 experiments. There exist many statistical tools, like the Matlab's *Statisticals and Machine Learning Toolbox*^{TM}, to obtain a D-optimal design of experiments. Matlab's function *cordexch*(*Nfactors*, *TreatementLevels*, *Nruns*, *model*, ...) uses a coordinate-exchange algorithm where i) the factors to consider and their treatment levels are specified by the value *Nfactor* and the vector *TreatmentLevels*, respectively, ii) the number of experiments in the design is limited by *Nruns*, and iii) the type of regression model that will be later used for inference (purely linear, quadratic or it can consider interactions among parameters) is defined by *model*.

Similarly to the previous approach, the resulting set of design configurations is evaluated by the DAVOS PPAD Evaluation Engine. However, some of these configurations may contain specific interactions among parameters that may prevent the actual implementation of the design or lead to implementations that do not meet the required constraints. These invalid configurations must be excluded from the design, and a set of new configurations must be appended to *repair* it. Following with the Matlab example, the number of experiments of the D-optimal design can be augmented by using the *daugment*(..., *filter*) function, where *filter* specifies a callback function, which filters-out the invalid configurations from the new design. An interesting feature of D-optimal designs is that they can be repaired on the basis of the experiments already carried out. As a result, new designs will contain all valid configurations exercised so far.

The collected sample is used to derive generalized regression models for each response variable. For each multilevel categorical factor resulting model includes as many terms as the number of degrees of freedom of that factor, i.e. for each factor X_i with K_i treatment levels $[0, 1, \ldots, L_i - 1]$ the linear predictor will include a component $(X\beta)_i = \beta_{i,1}(X_i = 1) + \beta_{i,2}(X_i = 2) + ...\beta_{i,L_i-1}(X_i = L_i - 1)$. The complete linear predictor is calculated as $X\beta = Intercept + \sum_{i=1}^{N}(X\beta)_i$, where $Intercept$ is a constant corresponding to the default setting 0 of all factors.

The response variable is consequently calculated using the mean function, whose canonical form depends on the assumed distribution or response variable: (i) $V = X\beta$ (identity) for normally distributed variables, (ii) $V = {}^{1}/_{X\beta}$ in case of gamma distribution, (iii) $V = exp(X\beta)$ in case of Poisson distribution, etc.

A compact yet statistically representative regression model can be derived using stepwise regression, implemented by the Matlab function *stepwiseglm(. . . , Criterion=Deviance, Penter=Threslod)*. Starting by an intercept term, it iteratively appends (removes) those terms to the model, whose p-value of an F-test of the change in the deviance that results from adding (removing) the term is less than a *Threshold* (assumed 0.05 by-default).

The determination coefficient (R^2) of each inferred regression model is then computed to check the percentage of the variability of estimated PPAD features that is explained by the model. A value of $R^2 = 0.7$ implies that the model explains 70% of the variability of the estimated feature, while a value of 100% means that the model completely explains the variability of that feature, i.e. it provides its exact value. So, as previously, a threshold must be defined in order to guide the acceptance or rejection of the computed regression models. Consequently, accepted models will be those explaining PPAD features with, at least, the established percentage threshold. The rejection of a model means that it does not explain the estimated feature with enough accuracy, thus the size of the design must be augmented to get a more precise estimation. The proposal is to augment the initial size of the D-optimal design each time a model is rejected in small portions (by a number of configurations which can be evaluated in parallel). Accordingly, new configurations must be selected and evaluated within an iteratively repeated process until the computed regression model meets the desired accuracy.

Once all regression models for all PPAD features have been inferred, it is possible to analytically determine which configuration optimizes each of these features. In case of pure linear models, this multi-objective optimization can be carried out by simply setting each factor to the level providing the best improvement according to the regression model. If the interaction model is considered, then one can iterate through the whole design space calculating the model response for each configuration.

It must be noted that, despite taking care of removing all invalid configurations from the D-optimal design, nothing guarantees the validity of optimal configurations. This is not surprising since they may have not been exercised during experimentation. So, before accepting an optimal configuration, it must be also evaluated by the PPAD evaluation engine. If the engine determines that the configuration is invalid, then an alternative must be found. In the methodology, an

iterative process searches for the best sub-optimal configuration that can be implemented. This is done by deriving from the optimal (but invalid) configuration a new set of sub-optimal (but maybe valid) configurations. Each sub-optimal configuration is computed by changing a parameter, that is set to its second best treatment level according to the regression models. The resulting set of sub-optimal configurations are then evaluated and ranked, choosing the best one from those that can be implemented.

7.3 Speeding-up the GA-based DSE by means of iterative selection

Genetic algorithms (GA) is a popular approach to setup a DSE as an iterative process, converging to (sub)optimal solution from initial set of random points in the design space. Each sampled configuration of parameters in GA context is referred to as *individual*, while the set of competing individuals is referred to as *population*.

Conventional GA can be seen as iteratively repeated sequence of evaluation, selection, crossover, and mutation operations, which gradually improves the fitness (quality) of individuals in population [49]. Initial population includes N randomly generated individuals. Each iteration starts by implementing new individuals in population, and evaluating their PPAD attributes. Evaluated individuals are subsequently ranked attending to the optimized PPAD attribute (e.g. robustness). Despite basic GA optimizes a single PPAD attribute, it can be adapted to multicriteria context by aggregating several PPAD attributes within a weighted sum model (WSM), so that GA is driven towards maximization of WSM score. K top-ranked individuals are selected for the inclusion into the next population. Selected configurations are crossed over (the setting S of each factor X in the offspring is randomly chosen from those of its two parents) to generate new offspring configurations. After that, a random factor of each offspring configuration is mutated (changed to a random setting within its range) with a certain probability.

Finally, randomly selected individuals from the offspring pool are added to the current population to bring it back to N configurations (*recombination*). This procedure is iteratively applied until meeting the stop condition: once top-ranked individual reaches the desired level in its optimization goal, or once reaching the convergence (fitness of top-ranked individual is not improving since several last iterations), then DSE is terminated, and the top-ranked individual is returned as the best (optimal) configuration.

The cost of each GA iteration is determined by the PPAD evaluation step, in particular by the performance of dependability evaluation. Depending on the particular experimentation scenario, several strategies presented in chapter 6 can be usually combined to speed-up as much as possible the dependability evaluation of individuals in population. When targeting FPGA designs, a basic strategy (referred to as *combined sampling* from now on) could be to rely on statistical fault injection, targeting optimized essential bits, while accelerating the individual injection runs through checkpointing and multiprocessing.

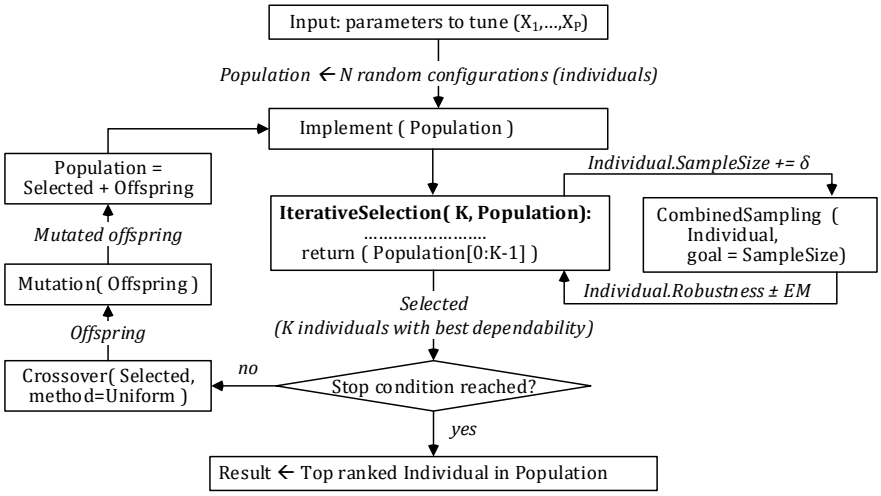


Figure 7.3: Single-objective GA-based DSE algorithm with iterative selection

The statistical fault injection used in combined sampling can follow either a conventional (conservative), or an optimized (iterative) approach as proposed in Section 6. In either way, it requires to define beforehand an error margin goal that should be reached for the estimated dependability attributes. However, it is usually not feasible to define a-priory such error margin that would be narrow enough, as to allow the ranking and selection of individuals, but not too strict, as to lead to very long evaluation time. Hence, to allow the confident selection, a conservatively strict error margin should be usually assumed, which leads to excessively long evaluation time.

The proposal to deal with this problem is to adapt the error margin of each individual to the actual diversity in population, in such a way as to obtain narrower error margins for those individuals that compete (overlap) among themselves during the selection. This can be achieved by transforming the consecutive evaluation and selection steps into an iterative selection process. The optimized selection al-

gorithm is described in Listing 7.3, whereas its integration into the score-based GA-based DSE flow is depicted in Fig.7.3.

The novel selection algorithm starts by executing just a small fraction (δ) of injection experiments for all competing individuals in population, and computes their dependability attributes, taking into account the actual margin of error EM_F, so they are bounded within the confidence interval $FailureRate \pm EM_F$. The latter is converted to $Score \pm EM_S$ in case of multi-objective selection.

Listing 7.3: Proposed selection algorithm based on iterative refinement of dependability confidence intervals

```
IterativeSelection(K, Designs, RankSelected):
  Foreach DUT in Designs :
       DUT.SampleSize = 384     #start by preliminary rough estimation
  Pool ← Designs                #References to DUTs to be evaluated
  #Evaluate and rank
  While not Pool.empty( ):
       #Obtain/Refine the dependability estimates (Injection Runs)
       Foreach DUT in Pool:
            DUT.Stats = CombinedSampling(DUT.MaskFile, DUT.SampleSize)
       #Rank DUTs in Designs attending to their score (best to worst) [1, 2]
       Foreach DUT in Designs:
            DUT.{Score ± EMS}= WSM( DUT.Stats.{Metrics±EMF}, DUT.PPA, Designs )
       Designs.sort ( Descending, key=Score )
       #Check whether confidence intervals of K best DUTs overlap with the rest of Designs
       Pool ← [ ]
       Foreach Top in Designs[0:K-1] :
            Foreach Bottom in Designs [K, Designs.length-1] :
                 If overlap( Top.{Score±EMS}, Bottom.{Score± EMS} ) :
                      #Schedule more injections to refine overlapping confidence intervals[2]
                      If (Top.EMF > EMThreshold) and Top not in Pool:
                           Pool.append(Top)
                      If (Bottom.EMF > EMThreshold) and Bottom not in Pool:
                           Pool.append(Bottom)
       #Check whether confidence intervals of K best DUTs overlap among themselves
       If RankSelected == True:
            Foreach A in Designs[0:K-1] :
                 Foreach B in Designs[0:K-1] :
                      If A != B :
                           If (A.EMF > EMThreshold) and A not in Pool:
                                Pool.append(A)
                           If (B.EMF > EMThreshold) and B not in Pool:
                                Pool.append(B)
       Foreach DUT in Pool:
            DUT.SampleSize += δ
  Return( Designs [0:K-1] )
```

[1] *Score aggregates considered robustness and PPA metrics under weighted sum model (WSM)*

[2] *EM_F, EM_S – error margins of robustness metrics (max) and score respectively*
$EM_{Threshold}$ – smallest error margin that is considered for robustness metrics

The individuals are sorted by the mean value (best to worst). After that the confidence intervals of top K individuals are compared to the rest of population, to determine whether they overlap. If there is no overlapping, then those $N - K$ low-ranked individuals can be confidently rejected, since even though their exact

robustness metric is still unknown, it will never reach that metric of K top-ranked individuals. Increasing their sample size will just narrow the margin of error, but not improve their robustness. In case that some overlapping exist, then the sample size for the involved individuals is slightly increased to reduce their margins of error and proceed again to their comparison. This process continues until the K most robust (best score) alternatives can be clearly distinguished from the rest of population. At this point no further refinement of dependability attributes is required (*Pool* of evaluated individuals becomes empty), and the K top-ranked individuals are selected.

If a final rank is desired (*RankSelected=True*), then the confidence intervals of K top ranked individuals are compared among themselves as well. This procedure keeps running until the confidence intervals of all selected alternatives do not overlap.

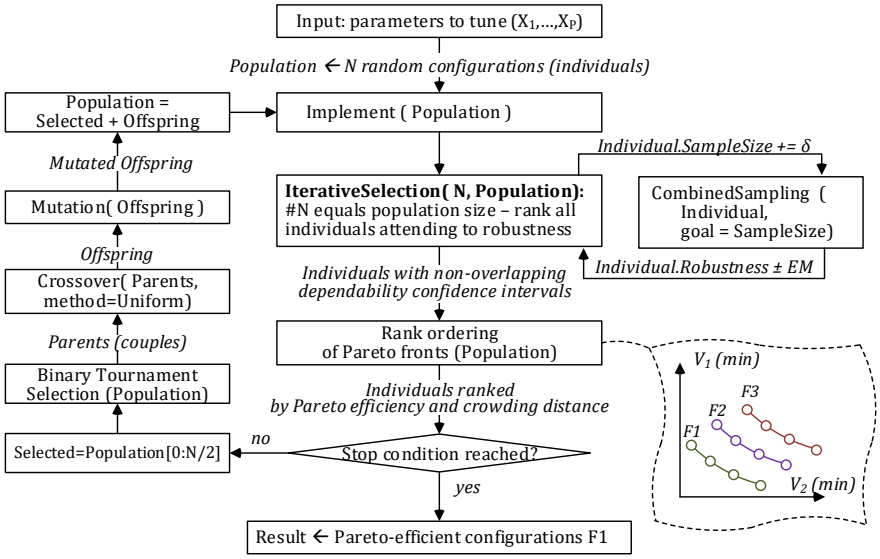


Figure 7.4: Multiobjective GA-based DSE, combining iterative selection and non-dominated sorting

In multi-objective context GA is often combined with the Pareto-optimization, by means of *Non Dominated Sorting Genetic Algorithm* (NSGA) [45]. Fig.7.4 illustrates an NSGA-based DSE flow, optimized by means of the proposed iterative selection.

This algorithm determines the fitness of individuals through the *rank ordering of Pareto fronts* [71]. At each GA iteration individuals are stored in the *pool* and sorted into several consecutive Pareto sets: all non-dominated individuals (such that none of their PPAD features can be outscored without degrading any other PPAD) are included into the primary set $F1$ and removed from the pool. The remaining individuals from the pool are then sorted again to determine the lower-rank (secondary) Pareto set $F2$. This procedure is repeated, obtaining even lower-rank Pareto sets ($F3$, $F4$, etc.), until *pool* becomes empty. Since rank ordering of Pareto fronts requires a pairwise comparison of each individual in the pool with each other, all of them should have non-overlapping robustness confidence intervals. This doesn't allow to quickly reject any configurations by rough robustness estimation. Nevertheless, optimized iterative selection still may be beneficial in NSGA context: by invoking it with $K = N$, and $RankSelected = True$, it will execute the minimum number of injection runs, that is required to confidently distinguish robustness of all individuals, and subsequently to perform the rank ordering.

After each rank-ordering the primary Pareto set $F1$ is checked against the stop condition: if at least one of the PPAD features has improved during several last iterations (e.g. by more than 5%), then NSGA continues. In this case, individuals in population are sorted by two criteria: (i) by ascending Pareto rank (lower Pareto rank is at higher priority), (ii) within each Pareto rank by descending *crowding distance* [45], which prioritizes more outstanding individuals (located further from its neighbours) in order to preserve the population diversity. The $N/2$ top-ranked individuals are selected for the inclusion into the next population. The parents for the offspring are selected through the *binary tournament selection*, which randomly picks two individuals and selects the one that has better (lower) Pareto rank, otherwise (under equal Pareto ranks) the one with higher crowding distance. The crossover, mutation and recombination operators remain the same as in the previously described basic GA. Once the stop condition is reached, the primary Pareto set $F1$ is returned as the best trade-off between the (potentially) conflicting optimization goals, so the designer is in charge of selecting one or another, attending to the requirements for the optimized PPAD attributes.

The attainable reduction in the number of experiments is dependent on each particular DSE scenario, and it will be exemplified in Section 9.3 through a concrete case study. It is worth noting that in a very unlikely worst-case scenario this enhancement will take the same number of experiments as the common selection strategy. So, it can only improve, and never penalise, the number of experiments to be carried out.

7.4 Conclusions

Despite the parameters of EDA tools and IP cores may significantly impact the attainable dependability properties of resulting HW implementations, no solutions have been developed so far for obtaining the most suitable configurations of these parameters for a given scenario. Enabling the tuning of these parameters requires to devise an efficient DSE strategy, which would reduce as much as possible the amount of costly fault injection experiments required to evaluate the dependability features of alternative configurations sampled from the design space. This chapter proposed two approaches to deal with this problem from different perspectives.

The first proposed approach reduces the DSE effort by minimizing the number of evaluated configurations. It relies on (i) the design of experiments (DoE) methodology to representatively sample the design space using the smallest possible set of trials, (ii) regression analysis to quantify the contribution of each parameter towards each PPAD feature and to infer a predictive model for PPAD attributes, and (iii) MCDM techniques to derive the optimal settings of parameters with respect to any optimization scenario. Within this (DoE-based) strategy two alternative experimentation flows have been defined. The first one, based on fractional factorial designs, estimates the main effects and second-order interactions without estimators confounding, but handles only regular design spaces with two-level parameters. Another optimization flow, based on iteratively refining D-optimal designs, supports irregular design spaces with multilevel parameters. However, it is more appropriate for estimating main effects, since it does not guarantee the design orthogonality and, thus, the absence of correlation between low-order and higher-order estimators. At the same time, the D-optimal DSE flow allows to reduce the sample size even beyond fractional factorial designs and, if necessary to improve the quality of obtained results by means of iteratively augmenting the design.

The second proposed approach improves the performance of GA-based DSE, by reducing the number of injection runs that should be carried out to select the best individuals at each GA iteration. It deploys the dependability evaluation and selection processes in an iterative way, so that fault injection is executed only for those individuals that compete for the selection and whose dependability confidence intervals overlap. Researchers do not need to decide beforehand the required error margin for dependability attributes, as they are dynamically refined until several best individuals can be confidently selected from the population. The speed-up gain is expected to be proportional to the diversity of dependability estimates of individuals in the population. Therefore, even though GA-based DSE

techniques may sample more individuals than DoE-based ones, in some cases they may actually require lower fault injection effort.

The particular advantage of the GA-based technique is its ability to optimize any-order interactions between the parameters of the design. Despite its convergence to the global optimum may require a considerable number of GA iterations, it can be stopped at any time, providing the best suboptimal solution found up to that point. The DoE-based DSE approach is more appropriate when the experimental time is strictly limited and/or when it is required to explain (quantify) the contribution of each parameter towards each PPAD feature.

Chapter 8

DAVOS Toolkit

This chapter presents DAVOS, an EDA toolkit which (i) addresses the limitations of currently available fault injection solutions, (ii) implements all accuracy-related and performance-related fault injection optimizations proposed in this thesis, and (iii) provides a customizable framework for a wide range of dependability-driven design and verification processes.

8.1 Introduction

The integration of dependability-driven strategies into the semicustom design flow requires an efficient and flexible fault injection solution that should meet several requirements:

- support HDL models defined in diverse HDLs and at different representation levels;
- support low-intrusive injection procedures for diverse fault models and implementation technologies;
- improve as much as possible the experimentation performance, especially in complex scenarios dealing with multiple alternatives;
- feature fine-grained injection, analysis, and reporting capabilities to enable the identification of weak points in the design.

The analysis of existing fault injection tools, presented in Section 3.3.4, has shown that they are commonly tailored for a certain HDL, representation level, or implementation technology. Furthermore, they cannot be easily extended to support new fault models, provide very basic analysis and reporting facilities, and do not take benefit of existing approaches to speed-up the remarkably slow simulation of implementation-level models. Accordingly, none of them is able to completely support the dependability-driven processes of the hardware design flow.

This chapter presents DAVOS (Dependability Assessment, Verification, Optimization, and Selection), an EDA toolkit that seamlessly integrates dependability-driven processes into the semicustom design flow. On the one hand, it provides a generic fault injection solution, which satisfies the aforementioned requirements, and implements all accuracy and performance improvement techniques proposed in this thesis. On the other hand, it provides a rich infrastructure, enabling i) the robustness assessment and identification of dependability bottlenecks in hardware designs, ii) the verification of deployed dependability-related strategies, iii) the dependability benchmarking (selection) of alternative IP cores, EDA tools, and implementation technologies, from a performance, power, area, and dependability (PPAD) perspective, and iv) the design space exploration (DSE) to tune the configuration parameters of IP cores and EDA tools for optimizing PPAD goals. DAVOS toolkit supports all these scenarios while remaining interoperable with existing standard HLDs and off-the-shelf EDA tools, including synthesizers, placers, routers, and simulators.

The rest of this chapter is structured as follows. Section 8.2 presents the architecture of DAVOS toolkit. Section 8.3 describes in detail the SBFI and FFI tools, which cover the dependability assessment and verification scenarios. Then, Section 8.4 describes the *PPAD evaluation engine*, which automates the evaluation of alternative HW implementations with flexibility and performance in mind. Finally, Section 8.5 describes a decision support tool that automates benchmarking and DSE experimentation scenarios.

8.2 DAVOS architecture

The core of DAVOS is a set of custom python modules that can be executed in any operating system under the basic Python distribution. These modules pertain to one of two categories: (i) DAVOS tools, which can be invoked from command line in standalone mode to perform some dependability-driven process, and (ii) support modules, which perform their dedicated functions at the request of DAVOS tools. Both tools and support modules are configured using XML-

formatted project configuration file that hierarchically describes all required data and/or configuration parameters. Interactive monitoring and data visualization capabilities are provided via web interface. The internal architecture of DAVOS, depicted in Fig. 8.1, responds to the necessity of defining a common interface to make DAVOS tools interoperable among themselves as well as with standard off-the-shelf EDA tools. Each of these modules provides the following basic functionality.

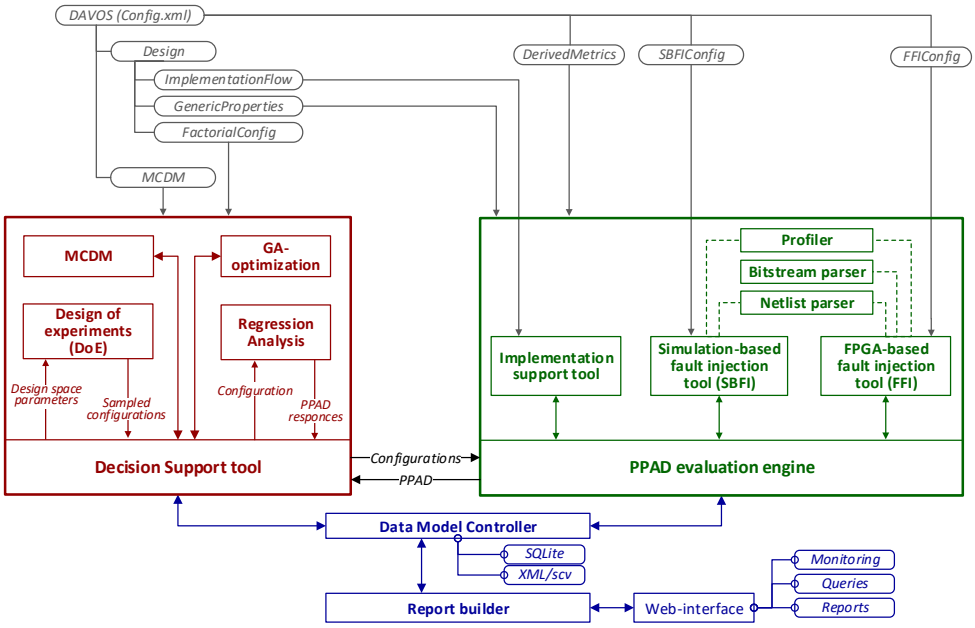


Figure 8.1: Architecture of DAVOS toolkit

1. Implementation support tool is in charge of running the semi-custom design flow for each design/configuration and the selected implementation technology. Different EDA tools can be assigned to each implementation phase. Performance, power, and area (PPA) attributes are estimated for each implemented design. This tool is not required when analysing behavioural/RTL models.

2. Simulation-based fault injection tool (SBFI) is at the heart of the process for estimating the dependability-related attributes of HDL models described in any standard HDL at any representation level. Fault dictionaries, defined after the proposal is Section 4.4, can be supplied to the SBFI tool to support custom fault models for any macrocell library. It can be used in conjunction with any commercial simulator that supports simulator commands similar to those used in

Chapter 4. Out-of-the-box SBFI tool is configured for Mentor Graphics' ModelSim/QuestaSim simulator. This tool can be omitted when focusing on PPA attributes.

3. FPGA-based fault injection tool (FFI) is in charge of estimating the dependability attributes of HW implementations targeting Xilinx's FPGAs. This tool is used for emulating SEUs in configuration memory, block memory, and registers after the approach described in Section 5.4. For any other fault model and/or implementation technology, SBFI tool should be used instead.

4. PPAD evaluation engine is a support module that automates the implementation and evaluation of PPAD attributes of multiple alternative designs and/or their parametrized configurations. It provides a configurable evaluation pipeline, built upon the API of SBFI, FFI, and implementation support tools. Each evaluation stage processes multiple designs in parallel, attending to the capabilities of the target computing platform, such as number of available grid nodes/PC threads (implementation and SBFI stages) and number of connected FPGA evaluation boards (FFI stage). Evaluation stages can be configured in arbitrary order, activated/deactivated, and customized through the dedicated sections of the configuration XML file (the same XML tags used by the implementation, SBFI, and FFI tools in standalone mode). Before returning the processed configurations to the requester, they are attributed by a number of user-defined metrics, computed on the basis of source PPAD attributes.

5. MCDM-Lib is a support module that implements a set of ranking algorithms used in multi-criteria decision making, including those based on WSM-scores and those based on the Pareto-efficiency.

6. GA-Lib is a support module that implements the genetic algorithms with iterative selection used for DSE purposes, as proposed in section 7.3.

7. Design of experiments (DoE) is a support module in charge of sampling the design space, attending to the given hypothesis (type of regression model to fit) and desired properties (Orthogonality, D-optimality, etc.) By interacting with Matlab statistical toolbox, it supports the generation of full factorial, fractional factorial, and D-optimal experimental designs, as well as the extension and repair of D-optimal designs. The resulting design is returned to the requester in the form of a table in which each row denotes one factorial configuration.

8. Regression analysis is a support module in charge of: (i) estimating each factor's significance, (ii) inferring the regression models for PPAD attributes, (iii) compiling the regression models into the compute-efficient form (native python-based or CUDA-based), and (iv) using the compiled models to predict the PPAD

responses for any configuration of factors from the design space. To estimate the significance of effects and to infer the regression models this module interacts with Matlab statistical toolbox. Regression model parameters (order of effects, distribution, p-value threshold for terms acceptance, etc.) can be specified separately for each PPAD attribute through the XML configuration file. Otherwise, most suitable regression parameters are determined automatically in a way that maximizes the model fit and reduces the deviation: (i) normal, gamma, and inverse Gaussian distributions are tested for continuous PPAD attributes, while Poisson and binomial distributions for discrete ones, (ii) a pure linear model for significant factors is tested first, whereas a model with interaction terms is tested (if the sample size allows) when the purely linear model has low explanatory potential.

9. Decision support tool automates those experimentation scenarios that involve decision making in a space of alternatives, such as dependability benchmarking and design space exploration. In case of benchmarking, alternative designs (evaluated by the PPAD engine) are ranked attending to the user-defined WSM-scores and/or Pareto-efficiency. Under the DSE scenario, it reports the best configurations of parameters for each optimization goal, Pareto-optimal configurations for different combinations of PPAD parameters, as well as the contribution of each factor towards the PPAD results.

10. Data Model Controller is a support module which collects and synchronizes the data among all DAVOS modules. It also performs the object-relational mapping between the internal data model and an SQLite database, and provides the interface for database queries.

11. Report builder is a support module in charge of exporting the web-based reports for different experimentation scenarios. It also performs the database queries on the request of web-based reporting and monitoring interfaces.

12. Web-interface is a collection of HTML5/AJAX and Javascript files that provides a user interface for process monitoring, data querying, and visualization of results.

By properly configuring how these modules coordinate their work, it is possible to support a wide range of PPA-driven and/or dependability-driven scenarios. Table 8.1 details which DAVOS tools and modules are employed in various supported dependability-driven scenarios.

Table 8.1: Sample application scenarios detailing which DAVOS tools and modules are used in each of them

		Tools				Support Modules					
	Experimentation Scenarios	SBFI	FFI	Implementation Support	Decision Support	PPAD evaluation engine	MCDM-Lib	GA-Lib	DoE	Regression Analysis	Report builder
Dependability assessment	Identify which components of HW design architecture are most sensitive to transient/permanent faults	✓									✓
	Assess the fault coverage of self-checking/fault-tolerant HW design implemented on standard cells	✓		✓							✓
	Estimate the failure rate of HW design implemented on Xilinx Zynq SoC		✓	✓							✓
Bench-marking	Determine which of 3 synthesizers promises the best PPAD trade-off for a microcontroller implemented on standard-cells or arbitrary FPGA	✓		✓	✓	✓	✓				✓
	Determine which of 3 alternative soft-core processors features better PPAD trade-off when implementing on Xilinx 7-series FPGA		✓	✓	✓	✓	✓				✓
DSE	Quantify the impact of synthesis parameters on PPAD results, and determine optimal configuration of these parameters, when targeting arbitrary implementation technology	✓		✓	✓	✓	✓		✓	✓	✓
	Tune the parameters of Xilinx EDA tools (ISE/Vivado) towards the best PPAD results, when experimentation time is not strictly limited, and optimization of multi-parameter interactions is desired		✓	✓	✓	✓	✓	✓			✓

8.3 Fault injection tools for dependability assessment

The most basic experimentation scenario supported by DAVOS is that related to the assessment of dependability-related properties of hardware designs and/or verification of their fault tolerance mechanisms. This experimentation scenario, automated by means of DAVOS SBFI and FFI tools, comprises three steps:

- customizing an SBFI/FFI environment attending to the given DUT and configuring fault injection experiments;
- executing the configured experimental set-up on the selected platform by invoking an SBFI/FFI tool with the configuration XML file as input:

  'DAVOS/> python SBFI.py config.xml';
  'DAVOS/> python FFI.py config.xml';
- analysing collected statistics through the interactive reporting interface.

This section details the architecture, configuration, and the workflow of DAVOS SBFI and FFI tools.

8.3.1 DAVOS-SBFI tool

Simulation-based fault injection tools (SBFI) is the keystone for all dependability-related scenarios dealing with HDL models at different description levels. The architecture of SBFI tool is depicted in Fig. 8.2. It comprises four main modules that control four subsequent SBFI phases. Each SBFI phase is configured by a dedicated section of the input XML configuration file. The main SBFI configuration options are listed in Table 8.2.

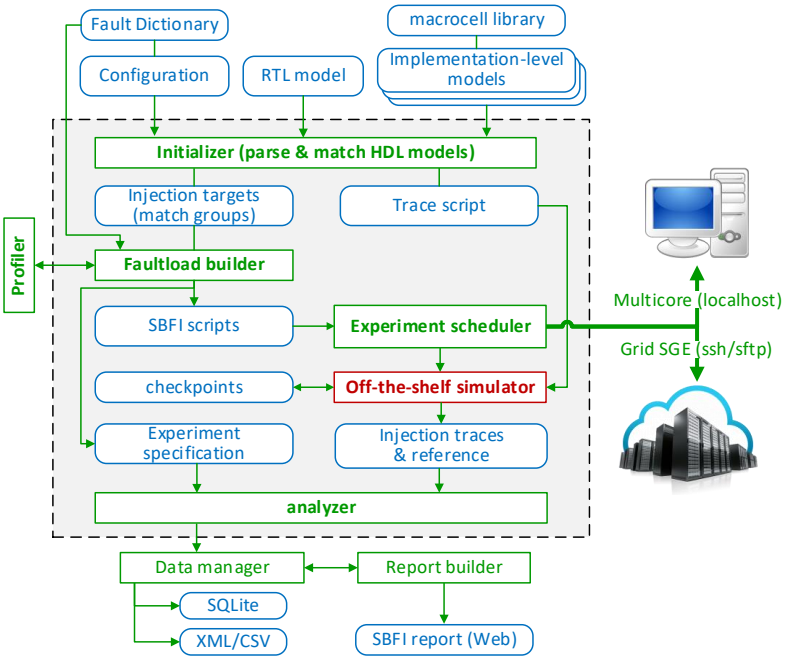


Figure 8.2: Architecture of Simulation-based fault injection tool

After invoking the SBFI tool, first, the *initializer* module parses incoming HDL model(s) in order to build a list of fault targets and a trace script. The list of fault targets includes all those design nodes located within the specified *InjectionScope*, whose basic type matches the *target_logic* property of the faultload configuration. When deploying a multilevel fault injection, the initializer also performs the register matching (as proposed in Section 6.3.1), using the RT-level list of nodes (specified in *match_pattern_file*) as a reference. In this latter case the resulting list of targets comprises a set of *match groups* that map each RTL register to the group of inferred (and possibly replicated) implementation-level macrocells.

The generated trace script instructs the target simulator to capture observation traces. The entries of the trace script correspond to those design nodes that match the *ObservationScope* section of SBFI configuration and configure whether they should trigger the sampling of observation vectors. By-default (*ObservationScope* kept empty) all DUT outputs and all state elements are traced and any change of their value is logged to the observation dump. When dealing with the post-synthesis models, the *register_reconstruction* option can be activated in order to log the corresponding state elements as a merged (RT-like) virtual register. This reduces the size of resulting traces and accelerates their analysis.

On the second SBFI phase, the faultload builder processes the *Faultload* configuration, the fault dictionary, and the fault list, in order to generate a set of SBFI scripts (one script per each injection run). Each *Fault_model* configuration in the *Faultload* section corresponds to one or more fault dictionary entries, as it has been explained in Section 4.4. Basically, it specifies the fault model (such as bit-flip, stuck-at, delay, etc.) and the type of targeted logic (macrocells), along with such parameters as fault multiplicity, fault duration, injection time, injection mode (sampling, exhaustive), and sample size. For each injection case within each matching macrocell, the faultload builder checks the instrumentation rules (if any), computes the parameters of the corresponding injection rule and, on its basis, exports a fault simulation script for the target simulator. The generated faultload is saved to the database and exported in the form of a CSV/XML-formatted specification file for the subsequent SBFI steps.

The default Verilog-based and VHDL-based fault dictionaries, supplied with the toolkit, cover most widely used fault models (bit-flip, stuck-at, indetermination, pulse, delay) applied to RTL primitives, as well as to post-synthesis (unisim) and to post-implementation (simprim) Xilinx macrocells. These dictionaries include ModelSim-specific injection rules so, whenever a different simulator is used, the injection rules from these dictionaries can be adapted accordingly.

The profiling of switching activity (defined in Section 6.2.1) can be activated by means of the *profiler* option of the SBFI configuration. The fautload builder in this case interacts with the support modules (netlist parser and profiler) as depicted in Fig. 6.3, and generates an optimized fautload, which includes only the active fault targets prioritized by their activity time.

It is worth noting that before the generation of SBFI scripts, the faultload builder performs the golden run (to obtain the reference trace) and stores the clustering checkpoints (to support the checkpointing speed-up strategy). The number of checkpoints to generate is configured by the *workload_split_factor* parameter (see

Table 8.2: Main options of DAVOS_SBFI tool, that should be configured to set-up an SBFI experiment

Configuration Tags	Property	Description
SBFI		
	platform	Kind of multiprocessing: 'Multicore' (local) or 'Grid' (SGE)
	maxproc	Number of tasks run in parallel
	workload_split_factor	Number of clustering checkpoint intervals
	force_reinitizlize	'on'/'off' - Cleanup and rebuild the SBFI experimental environment (backup previously collected results)
	profiler	'on' / 'off' – profiling of switching activity for optimized faultload
	injector	'on' / 'off' – run injection phase for existing SBFI environment
	reportbuilder	'on' / 'off' – build SBFI report for collected traces/result
1:1	fault_dictionary	Path the fault dictionary file (XML) for used macrocell library
→ Initializer		
	register_reconstruction	'on' / 'off' – merge related FFs/Latch outputs into virtual signals at tracing instead of logging them separately (reduces the logfile size)
1:N	match_pattern_file	List of RTL design nodes, used for multilevel registers matching
→ InjectionScope		
	unit_path	Path to design unit that should be targeted at injection
1:N	node_prefix	Include into the fault list only those nodes that start with this prefix
→ ObservationScope		
	unit_path	Path to design unit that should be traced (logged)
	node_prefix	Log only those nodes that start with this prefix
	sampling_options	Specify '-notrigger' to prevent logging of trace vectors on switching events of the nodes in this scope
1:1 → WorkloadConfig		
	compile_script	Custom TCL script to build the simulation environment for the DUT
	run_script	Custom TCL script to run the DUT simulation
	std_init_time	Initialization time interval, preceding to the workload
	std_workload_time	Workload time interval (duration), considered at fault injection
	timeout_flag	Custom signal (if any) denoting workload completion
→ Faultload		
→ Fault_model		
1:N	model	Fault model referenced from the dictionary ('bit-flip', 'delay', etc.)
	target_logic	Macrocell type referenced from the dictionary ('X_FF', 'X_LUT', etc)
	faults_per_target	Number times each macrocell can be targeted at injection ('1' for sampling without replacement)
	time_mode	'Absolute' / 'Relative' / 'CLOCK'– bounds of injection interval are specified – in absolute time units (ns) / as percentage of workload duration / in clock cycles
	injection_time_start / injection_time_end	Starting and Ending time of injection interval; to inject at particular time instant ending time should coincide with the start time
	sample_size	Number of fault configurations sampled using uniform distribution in time and space, '0' – for exhaustive experimentation
1:1	max_error_margin	Desired error margin for iterative sampling (empty to disable)
→ Analyzer		
	quick_mode	'on' - Analyse the failure mode using single trace vector at workload completion (first vector with raised timeout flag), 'off' – analyse all vectors in the trace file to enable computation of error and failure latencies
	error_flag	Alarm signal activated by the DUT on error detection (used to distinguish between signalled failures and silent data corruption)
	error_flag_active_value	Active value of error_flag, indicating the error detection

Table 8.2). Subsequently, each generated SBFI script is linked to its corresponding (closest) checkpoint.

On the third phase, the *experiment scheduler* runs the generated SBFI scripts on the target computing platform: Sun-Grid-Engine-based computing cluster or multicore PC (locally). In both cases, the number of parallel simulation processes is configured by means of the *maxproc* parameter. In case of a grid platform, all SBFI scripts are distributed between the available grid nodes, being grouped into *grid jobs* in such a way as to balance their corresponding simulation effort. The execution of grid jobs is managed by the Sun Grid Engine, so the experiment scheduler is in charge of simply monitoring the status, reporting the statistics, and handling the exceptions. In the case of local (multicore) platform, the SBFI scripts are launched dynamically by the scheduler upon availability of computing resources.

On the fourth phase, all the collected SBFI traces are processed by the *analyzer* module. Each trace is parsed into a sequence of trace vectors, representing a state of the DUT outputs and DUT's internal states at the different time instants. By comparing the injection traces with the reference one (golden run), the analyzer determines the failure mode in each SBFI run (masked, latent, silent data corruption, signalled failure - as depicted in Fig 3.1) and computes the latencies. The alarm signal (if supported by the DUT) should be specified through the configuration property *error flag*. By enabling the *quick mode* the analyzer is instructed to perform the reduced/quick analysis, which determines the failure mode merely by analysing the last captured vector corresponding to the workload completion time. The outcome of each SBFI run is stored to the database, while traces are attached to the database as a separate (compressed) zip package.

Finally, SBFI tool launches the report builder module, which loads all the collected results from the database, computes the aggregated metrics (such as failure rate, mean latencies, etc.) for each DUT and for each fault model, and exports a web-based report into the specified directory on the web-server.

8.3.2 DAVOS-FFI tool

DAVOS FFI tool provides an instrumental support for the dependability assessment of FPGA-based designs. It integrates the accuracy and performance-related optimizations proposed in this thesis, including the optimized essential bits, profiling of switching activity, iterative statistical fault injection, and multiprocessing.

The major part of the fault injection flow has been deployed internally on the Xilinx's Programmable SoCs, i.e. chips integrating an ARM microprocessor (PS) and programmable fabric (PL). This enables a drastic reduction of the communi-

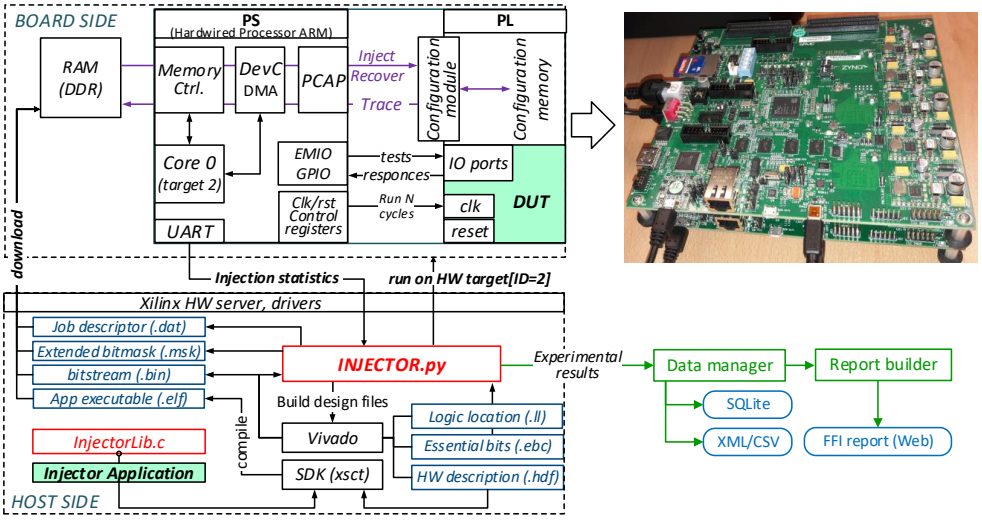


Figure 8.3: Architecture of FPGA-based fault injection tool

cations with the external host controller, which are limited to just uploading the initial configuration and the provision of statistics issued from injections.

The architecture of FFI tool comprises three main components, depicted in Fig. 8.3: i) a set of Xilinx Zynq prototyping boards (board side), ii) a standalone fault injection application executed within these boards, and iii) an experimentation management application running on a PC, which represents the host side. The injection workflow encompasses 3 phases: the initial setup, the realization of the injections, and the monitoring of results.

The host side runs the *setup phase*. During this phase, all the files required for the execution of all the fault injection experiments are generated and uploaded to the prototyping boards. They include i) the injector application required to control the experiments in each board (.elf files), ii) the bitstream files (.bit/.bin) for the implemented DUTs, iii) the optimized essential bit mask file (.msk), which locates the potential fault targets within the DUT, and finally iv) the experiment descriptor, which configures the fault injector application on each board.

The on-board application is responsible for: i) supplying the workload to the DUT, ii) the emulation of SEUs within the randomly or exhaustively selected bits of the optimized essential bitmask, iii) the observation of the DUT behaviour, and periodically, vi) the (re)computation of robustness metrics (exhibited failure

modes) along with their respective error margins, which are logged to the host through the selected communication interface (serial port by-default).

The injector application is defined on the basis of a custom library *InjectorLib.c*. This library provides a rich API for the definition of autonomous FPGA/SoC-based fault injectors that can operate through the PCAP or ICAP configuration interfaces. A fault injector application for a particular design can be easily obtained from the included application template. It requires to customize two DUT-specific callback functions: (i) the *DutEnvelope()*, which supplies the workload to the DUT and verifies the DUT processing results, returning a non-zero result in case of mismatches, and (ii) the *TriggerGSR()* function, which activates the global set-reset signal (required to reinitialize the PL registers and latches from associated CM cells). The access to the GSR line can be obtained by instantiating the *StartupCtrl* module (provided as a part of the injector library) within the block design, and by connecting it to the PS (e.g. through the GPIO interface). The customized injector application is compiled for the target HW platform (board support package generated by Vivado) and the resulting executable (*.elf) is uploaded to the board.

The fault experiment itself is configured using the FFI section of the XML configuration file. Table 8.3 lists the main parameters used to generate the optimized essential bitmask and the job descriptor files. The former locates the fault targets of the selected type of *target_logic* (FF/LUT/BRAM/TYPE0), within the selected *dut_scope*, after the algorithm proposed in Section 5.3. The *custom_lut_mask* can be enabled for the improved (bit-accurate) identification of LUT-specific essential bits, whereas the *profiling* parameter can be enabled to filter-out the inactive bits. The job descriptor communicates the rest of injection parameters to the injector application, including the operation *mode* (sampling, exhaustive), the sample size, the error margin, the fault multiplicity, etc. When the tracing of latent errors is not required, it can be disabled for the sake of higher experimentation performance, as it will spare the time-consuming readback-verify operations of changeable CM frames.

Once all the files are uploaded to the board, the host starts the injector application and enters the monitoring phase, which runs in parallel to the execution phase on the board. The goal is to enable the host to analyse the various received logs in order to determine whether the experimentation should terminate (the required error margin/sample size has been reached) or not. Once this decision is taken, the logged results are saved to the database and the report builder is invoked to export the fault injection report to the selected directory on the web-server. In the case of running the FFI tool from within the top-level framework (e.g.

Table 8.3: Main options of DAVOS_FFI tool, that should be configured (in FFI tag of XML configuration file) to set-up an FFI experiment

FFI configuration parameters	Description
dut_scope	Path to the design unit, that will be targeted at fault injection (e.g. 'DUT/Top/Core')
target_logic	Type of logic primitives that will be targeted at fault injection: 'FF' / 'LUT' / 'BRAM' - selectively target these primitives using optimized essential bits mask; 'TYPE0' – unmapped mode targeting all CM cells included into Xilinx Essential Bits mask
custom_lut_mask	'on' - Enable bit-accurate mapping of LUTs onto configuration memory, 'off' – use LUT mask generated by Vivado
profiling	'on' / 'off' – profiling of LUT switching activity to filter-out inactive CM cells and prioritize active ones
mode	'1XX' – injection using (optimized) essential bit mask: '101' – iterative sampling mode '102' – exhaustive mode '2XX' – injection using fault list, prioritized by activity time (LUTs only) '0XX' – service modes: '000' – handshake identify all available (connected) devices (targets) and relate them with their serial ports '001' – cleanup the cache on the target devices (bitstream and bit mask files) '004' – profiling of FAR register – retrieves a list of valid frame address for a target device part (used for CM mapping, bitstream/bitmask parsing)
error_margin_goal	Error margin threshold in derived robustness metrics (percentage points between '0.0' and '100.0').
sample_size_goal	Minimal number of fault configurations to be sampled.
fault_multiplicity	Number of upsets per injection run, e.g. 1 – single bit upset, ≥ 2 – multiple bit upset
injection_time	Clock cycle at which faults are injected '0' – random injection time (within workload duration) '1' – inject at workload start 'N' – inject at N-th clock cycle
recovery_nodes	Name of BRAMs, whose content should be forcibly recovered after each injection run
injectorapp_path	Path to precompiled InjectorApp (*.elf)
memory_buffer_address	Memory offset on the target device, pointing to a buffer (min 16MB) dedicated for data interchange between InjectorApp and host app.
platformconf	Predefined list of targets linked to their serial ports, e.g. "[{'TargetId':'2', 'PortID':'COM3'}, {'TargetId':'6', 'PortID':'COM1'}]" – two targets (2 and 6) linked to comports 3 and 1 respectively. If empty "[ ]" – handshake will be carried out automatically prior to running the fault injection.

PPAD evaluation engine), the report builder is skipped and the raw dependability estimations are returned to the evaluation requester.

To support the iterative selection process, the board-side application caches the bitstream, the updated state of essential bit mask (targeted bits are excluded from the bitsmask), and the collected injection statistics on the SD card. Accordingly, once the FFI tool is requested to refine the robustness metrics for one of the previously tested DUTs (with narrower error margin or increased sample size), the compilation and uploading steps are skipped, thus minimizing the overhead for the restoration of the injection process.

8.3.3 Interactive reporting interface

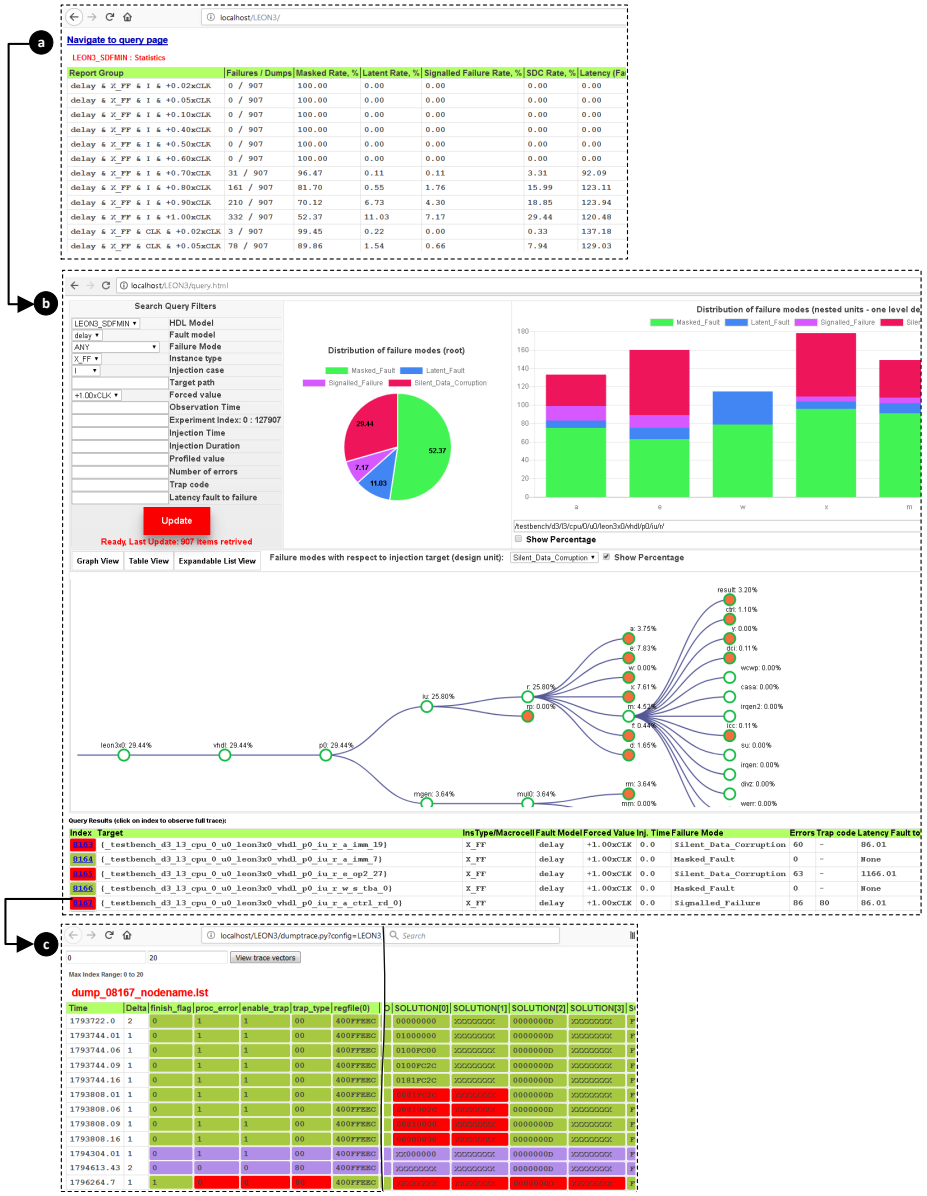


Figure 8.4: Interactive web-based fault injection report: (a) summary page, (b) query interface, (c) detailed trace

Experimental results collected in the database are visualized through the interactive reporting interface, illustrated in Fig. 8.4. The main screen (Fig. 8.4a) of the fault injection report displays a summary of the whole set of SBFI/FFI experiments, including the distribution of failure modes and latencies. Reported results are grouped by the DUT, by the fault model, and by the type of targeted logic. It also provides access to the detailed trace information for each particular injection run (Fig. 8.4c) and another screen for custom queries.

These custom reports can be tailored according to a number of filters, like fault models, target logic, and/or resulting failure mode, among others. Likewise, the HDL model can be hierarchically navigated, displaying the results obtained for each of its constituent parts. For instance, Fig. 8.4b depicts the failure modes distribution for the permanent interconnect delays affecting the I port of X_FF primitives of the DUT (LEON3 microprocessor). A 29.44% of the experiments led to silent data corruption (SDC). By navigating its hierarchy, it can be seen that the execute (e) and exception (x) pipeline stages are the ones contributing the most to SDC, with 7.83 and 7.61 percentage points, respectively.

Finally, the table on the bottom on the query screen lists the details of each injection run that satisfies the filters. The index cell navigates to the detailed trace (build on demand), which highlights all the mismatches (with respect to the golden run) on the DUT's outputs (failures) and on the internal nodes (errors). It is worth noting that the dump trace feature is only supported for SBFI experiments, since the FFI traces are analysed completely on the board side and not dumped to the host for the sake of experimentation performance.

8.4 Automated PPAD evaluation of parametrized designs

The implementation support tool and the PPAD evaluation engine are two instruments offered by DAVOS for the automated implementation and for the PPAD evaluation of parametrized design alternatives. The former links the diverse EDA tools into a custom design flow, recognizes their parameters (flags), tunes them (with input configurations) to produce the DUT implementations, and evaluates their PPA attributes. The latter (evaluation engine) encapsulates the implementation support tool, SBFI tool, and FFI tool within an integrated evaluation flow, which processes multiple design alternatives (configurations) in parallel. This section details the customization and the workflow of the aforementioned tools.

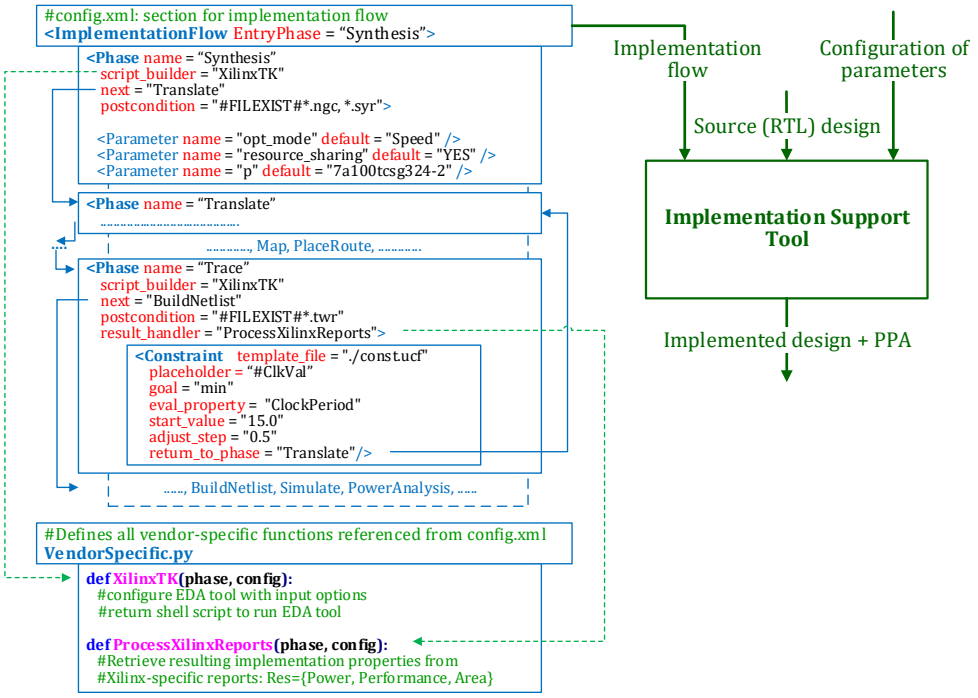


Figure 8.5: Excerpt from an example configuration file, defining an implementation flow under Xilinx ISE toolchain

8.4.1 Implementation support tool

The integration of diverse third-party EDA tools within a single design flow requires to adapt all their specific configuration/invocation methods, data, and reporting formats. This need for flexibility and adaptation has led to the definition of a custom implementation flow as a part of an XML configuration file that must be customised according to the desired control flow. For instance, the excerpt displayed in Fig. 8.5 generates all the required scripts to run the implementation module in association with the Xilinx ISE toolchain.

Each EDA tool or suite comprises a number of different phases, each one with different configuration parameters, that should be consecutively executed to obtain the final implementation of the design. Accordingly, the *<ImplementationFlow>* section of the configuration file includes a *<Phase>* subsection for each one of the stages of that particular instance of the semi-custom design flow.

Each *Phase* includes the following attributes: *name*, used as identifier; *next*, which states the following phase to be executed after that phase successfully finishes; *postcondition*, assertion that must be held true after finishing that phase, like *#FILEXIST#*ngc, *.syr* that checks, for the Xilinx's XST synthesiser, that the netlist file (ngc extension) and the synthesis report file (syr extension) exist; *result_handler*, which references a Python custom function in charge of processing the information obtained from this phase and generating the associated reports in the required format; and *script_builder*, which references a Python custom function that generates the command line script required to execute the associated module and/or tool as defined in that subsection.

Additionally, it can include any number of *<Parameter>* subsections, which define the default values for the given configuration parameters, and *<Constraint>* subsections, which iteratively tune the implementation constraints to ensure it meets the desired requirements. For instance, the *Constraint* subsection displayed in Fig. 8.5 aims at attaining an implementation with the highest possible clock frequency (minimum clock period). It iteratively decrements the clock period requirement by 0.5 ns, starting at 15 ns, and returning to the *Translate* phase to rerun the implementation with the new constraint. Once constraints are so tight that the design cannot be implemented, they are set to the last known valid value.

Finally, a *result_handler* references another custom function (defined in the support script *VendorSpecific.py*), which extracts the PPA features from the vendor-specific reports and returns them in the form of a python dictionary.

Thus, using new modules and/or EDA tools involves the definition of new custom Python files to generate the required command line scripts and obtain the data to be processed by the report management module. After that, configuration files can be customized to integrate these tools into the control flow for any application scenario.

8.4.2 PPAD evaluation engine

The PPAD evaluation engine manages the parallel implementation and PPAD evaluation of multiple design configurations. This module, once instantiated by the evaluation requester, creates a pool of evaluation processes (linked by a set of queues) configured by the dedicated sections of an XML configuration file. For instance, a configuration *DAVOS.EvalSteps = 'Implementation, FFI'* instructs the evaluation engine to assign the first pool of evaluation processes to the implementation support tool, and the second pool to the dependability evaluation using

```
#Config.xml: section for derived metrics
<DerivedMetrics>
    <Metric
      name = "MTTF"
      handler = "Derive_MTTF_FPGA"
      CustomArg = "{'k' : 21, 'FIT.CRAM' : 76E-9, 'FIT.BRAM' : 73E-9}"/>

#Defines callback functions for computation of custom metrics
CustomMetrics.py
    #callback function invoked by PPAD evaluation engine to compute custom metric, derived from:
    #{metrics} – PPAD results obtained by SBFI/FFI/Implementation tools,
    #{CustomArg} - external argument, converted to dictionary at invocation
    def Derive_MTTF_FPGA(metrics, custom_arg):
      LambdaCRAM = metrics['EssentialBits.Type0']*metrics['SDC.Type0']*CustomArg['FIT.CRAM']
      LambdaBRAM = metrics['EssentialBits.BRAM]*metrics['SDC.BRAM']*CustomArg['FIT.BRAM']
      Lambda = CustomArg['k']*( LambdaCRAM + LambdaBRAM )
      return(1/Lambda)
```

Figure 8.6: Excerpt from configuration file defining custom PPAD metrics

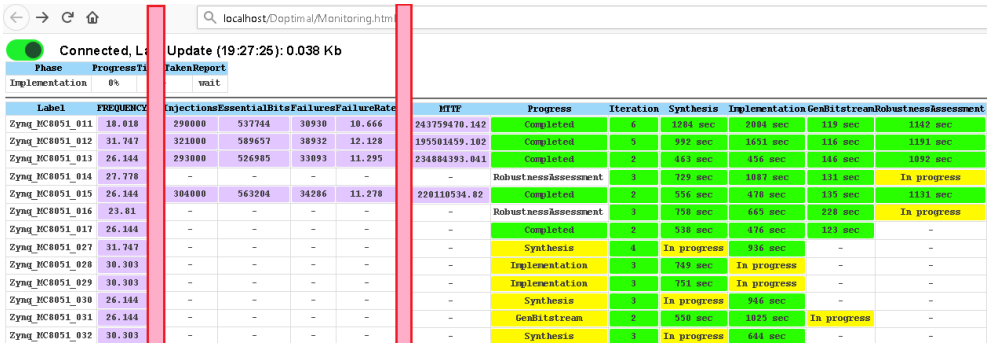
FFI tool. The number of processes in the implementation pool is configured by the *maxproc* property of *DAVOS.Design.ImplementationFlow* section. The FFI pool is configured by *DAVOS.FFI.platformconf* property (see Table 8.3) or (if kept empty) it is configured by the automatic target detection mechanism.

It is worth mentioning that the term configuration refers to the object of DAVOS datamodel which describes a design alternative. It encapsulates such properties as setting of architectural/EDA parameters, dictionary of PPAD attributes, model path, etc.

The configurations can be evaluated by the evaluation engine in a blocking and non-blocking mode. In the former case, the requester invokes the *evaluate(DutList)* method, which pushes all the configurations to the input queue of the evaluation engine and returns when all configuration get processed by the engine (receive their PPAD attributes). In the latter case, the requester pushes the configurations directly to the input queue of the evaluation engine (without blocking the requester's process) and periodically checks the availability of evaluated configurations in the output queue of the engine.

Before returning the evaluated configurations, they are attributed by a set of custom metrics derived on the basis of the raw PPAD estimations. These derived metrics are defined by designers in the dedicated XML section, as it is depicted in Fig. 8.6. Each derived metric is computed by a custom python function (linked by the *handler* attribute). Such custom functions have access to all the previously obtained PPAD metrics (such as frequency, power, area, distribution of failure modes), as well as to the dictionary of constants defined in the *CustomArg* attribute. For instance, an example in Fig. 8.6 illustrates the computation of the

mean time to failure on the basis of utilization of essential bits (in Type0 and BRAM frames), their respective percentage of silent data corruption, the failure rate constant (FIT) for the selected FPGA, and the altitude-related failure rate derating factor (k) [136].



Label	FREQUENCY	Injections	EssentialBits	Failures	FailureRate	MTTF	Progress	Iteration	Synthesis	Implementation	GenBitstream	RobustnessAssessment
Zynq_MC8051_011	18.018	290000	537744	30930	10.666	243759470.142	Completed	6	1284 sec	2084 sec	119 sec	1142 sec
Zynq_MC8051_012	31.747	321000	589657	38932	12.128	195501459.102	Completed	5	992 sec	1651 sec	116 sec	1191 sec
Zynq_MC8051_013	26.144	293000	526985	33093	11.295	234884393.041	Completed	2	463 sec	456 sec	146 sec	1092 sec
Zynq_MC8051_014	27.778	-	-	-	-	-	RobustnessAssessment	3	729 sec	1087 sec	131 sec	In progress
Zynq_MC8051_015	26.144	304000	563204	34286	11.278	220110534.82	Completed	2	556 sec	478 sec	135 sec	1131 sec
Zynq_MC8051_016	23.81	-	-	-	-	-	RobustnessAssessment	3	758 sec	665 sec	228 sec	In progress
Zynq_MC8051_017	26.144	-	-	-	-	-	Completed	2	538 sec	476 sec	123 sec	-
Zynq_MC8051_027	31.747	-	-	-	-	-	Synthesis	4	In progress	936 sec	-	-
Zynq_MC8051_028	30.303	-	-	-	-	-	Implementation	3	749 sec	In progress	-	-
Zynq_MC8051_029	30.303	-	-	-	-	-	Implementation	3	751 sec	In progress	-	-
Zynq_MC8051_030	26.144	-	-	-	-	-	Synthesis	3	In progress	946 sec	-	-
Zynq_MC8051_031	26.144	-	-	-	-	-	GenBitstream	2	550 sec	1025 sec	In progress	-
Zynq_MC8051_032	30.303	-	-	-	-	-	Synthesis	3	In progress	644 sec	-	-

Figure 8.7: Monitoring interface, showing the current status of PPAD evaluation process and summary of collected results

The status and the details of an ongoing evaluation process are visualized through the web-based monitoring interface, depicted in Fig. 8.7. This interface lists the status of each configuration submitted to the evaluation engine (queued, in process, completed), the time taken for each evaluation step, the number of iterations (in the case of constraint adjustment), and the obtained PPAD results. Finally, each cell in the table navigates to the detailed log of the corresponding implementation/evaluation process.

8.5 Decision support tool for selecting and optimizing HW designs

The decision support tool is in charge of those experimentation scenarios which concern with the evaluation of alternatives, namely the dependability benchmarking (selection), and the design space exploration. The experimentation scenario is selected by setting the *mode* property of the *DecisionSupport* section of the configuration XML file to one of the following values:

- 'BENCHMARK' to run the score-based dependability benchmarking;
- 'DSE-GA' or 'DSE-NSGA' to run the design space exploration using the score-based GA or using the non-dominated sorting GA, respectively;

```
#Config.xml: section for Decision Support tool
<DecisionSupport  mode = "Benchmark">

  <MCDM>
    <Scenario name = "Automotive" scoremodel = "WSM">
      <variable goal = "max" name = "FREQUENCY"  weight = "0.40"/>
      <variable goal = "min" name = "POWER"      weight = "0.10"/>
      <variable goal = "max" name = "MTTF"       weight = "0.50"/>
    </Scenario>

    <Scenario name = "Mobile" scoremodel = "WSM">
      <variable goal = "max" name = "FREQUENCY"  weight = "0.25"/>
      <variable goal = "min" name = "POWER"      weight = "0.50"/>
      <variable goal = "min" name = "AREA"       weight = "0.10"/>
      <variable goal = "min" name = "SDC"        weight = "0.15"/>
    </Scenario>
  </MCDM>

  <Model label = "Microblaze" path = "../IpCores/Microblaze" />
  <Model label = "MC8051"     path = "../IpCores/MC8051" />
  <Model label = "AVR"        path = "../IpCores/AVR" />
```

Figure 8.8: Sample configuration of the decision support tool for the dependability benchmarking: two weighted-sum models, and three alternative IP cores

- 'DSE-FACT' or 'DSE-DOPT' to run the DoE-based design space exploration using the fractional factorial designs or using the D-optimal designs, respectively.

The benchmarking experiment comprises two steps: the evaluation of PPAD attributes of alternative HDL designs and their ranking attending to the Weighted Sum Method [160]. It is worth noting that the term *alternative* from the DAVOS viewpoint encompasses the HDL model, the EDA tools used for its implementation, and the target implementation technology. Alternatives are supplied for benchmarking by specifying a *Model* subsection (within the *DecisionSupport* configuration) for each DUT, as it is depicted in Fig. 8.8. Those models that do not share the rest of configuration sections (implementation flow, SBFI, FFI) are specified in a separate XML configuration files. In this case a list of prepared configuration files is supplied to the decision support tool. The decision support tool uses the PPAD evaluation engine to implement all the considered alternatives in parallel and to evaluate their PPAD attributes.

On the second step, each considered model receives a set of WSM scores, which quantify their relative goodness for different application scenarios. The WSM scores are defined by means of *Scenario* configuration subsection (as exemplified in Fig. 8.8). Each scenario considers a certain set of PPAD attributes (including the derived metrics) specified within the *variable* subsections. The variables should be either maximized or minimized (*goal* property) and their relative importance is specified by means of the *weight* property. Computed scores for each scenario are

```
#Config.xml: section for Factorial design
<FactorialDesign ConfigTable = "FactorialConfig.csv">
   <Factor name = "X01"  option = "opt_mode"  phase = "Synthesis">
      <setting factor_value = "0" option_value = "Speed" />
      <setting factor_value = "1" option_value = "Area" />
   </Factor>
   .....
   <Factor name = "X20"  option = "resource_sharing"  phase = "Synthesis">
      <setting factor_value = "0" option_value = "NO" />
      <setting factor_value = "1" option_value = "YES" />
   </Factor>
   .....
```

Figure 8.9: An example of configuration section defining the factorial design: each factor denotes one EDA parameter, and has two or more treatment levels

visualized through the monitoring interface of the PPAD evaluation engine (along with the PPAD features), so designers can select the top-ranked configuration for each of the defined scenarios.

The DSE experimentation scenario takes place when designers need to configure the optimisation flags of selected EDA tools and/or the architectural parameters of considered IP cores to attain the best possible implementation. As in the case of dependability benchmarking, this will also depend on the criteria defined for each application domain.

For DAVOS to generate the required design of experiments, it requires all the optimisation parameters to be considered and the levels at which they can be set. This is accomplished by customising the *<FactorialDesign>* section of an XML configuration file. For each parameter, a new *<Factor>* subsection must be appended, specifying the *option* of the EDA tool and the *phase* of the semi-custom design flow in which it is used. Additional *<setting>* subsections define the levels considered for the design of experiments (*factor_value* attribute) and the actual value of this tool option (*option_value* attribute). Fig. 8.9 depicts an excerpt of this configuration file (for the Xilinx XST synthesiser). For instance, the *opt_mode* synthesis option is denoted as factor *X01* and can be set to either *Speed* (*X01=0*) or *Area* (*X01=1*).

The particular exploration method should be selected attending to the properties of the design space and to the experimentation time constraints. For instance, the DSE based in genetic algorithms is most suitable in the absence of any hypothesis regarding the order of significant effects or the type of regression model, as well as when the experimentation time is not strictly limited. When this method is selected, the decision support tool follows the optimized GA/NSGA

algorithms, proposed in Section 7.3. The GA-based DSE is customized by means of a *GA* subsection in the *DecisionSupport* configuration. Particularly, a *goal* property specifies either a comma-separated list of PPAD attributes, which will be optimized using NSGA (when *mode='DSE-NSGA'*), or a name of WSM score to optimize using score-based GA (when *mode='DSE-GA'*). Other parameters that can be customized are *PopulationSize* (default 12), *SelectionSize* (default 6), *CrossoverType* (default 'Uniform'), and *MutationRate* (default '0.5'). The convergence process, along with the evaluated configurations, are logged into the CSV-formatted file and visualized through the monitoring interface of PPAD evaluation engine.

In the case of limited experimentation time, the DoE-based DSE methods are more suitable. When all the design parameters are quantified at two levels and in absence of incompatible configurations in the design space, the fractional factorial method can be used (*mode='DSE-FACT'*). Otherwise a more flexible D-optimal method is appropriate (*mode='DSE-DOPT*). In either way, the decision support tool first generates the corresponding experimental design (table of sampled configurations) by invoking the DoE support module. Consequently, the sampled configurations are evaluated (using the PPAD evaluation engine). Using the regression analysis module, the obtained results are statistically analysed to determine the significant effects, infer regression models on their basis, and compile them into a compute-efficient form. Finally, the best configurations are determined by the factor-wise optimization (selecting the level for each factor that provides the best response from the regression model) or by exhaustively checking the responses for each configuration in the design space (when interaction terms are considered).

The DSE results are reported to the DSE summary screen, exemplified in Fig. 8.10. In addition to the best configuration for each score-based optimization scenario, it also displays a graph with Pareto-efficient configurations highlighted (for which it is not possible to improve one PPAD property without negatively impacting another). In the example showed in Fig. 8.10, there are two configurations that maximise MTTF or clock frequency in detriment of the other, whereas other two configurations find a trade-off between them. This Pareto-optimal graph can be of interest when no particular application domain is considered and, thus, no weights have been defined for selected criteria.

Scenario	B/W	Score	FREQUENCY	MTTF_K_1	MTTF_K_21	MTTF_K_327	POWER_DYN	SDC	UTIL_FF	UTIL_LUT	X01,X02,X
Consumer	Best	0.95	138.45	4.07e+10	-	-	25.61	30.87	691.62	504.35	0, 0,
	Worst	0.77	103.98	2.52e+10	-	-	24.64	26.99	730.28	1141.03	1, 0,
Mobile	Best	0.91	113.91	4.20e+10	-	-	21.52	31.26	659.75	506.13	1, 1,
	Worst	0.71	127.45	2.18e+10	-	-	37.73	28.81	730.28	1261.34	0, 1,
Industrial	Best	0.99	135.42	-	1.17e+09	-	27.53	26.61	720.38	1045.73	0, 0,
	Worst	0.79	115.03	-	1.61e+09	-	26.46	34.10	659.75	605.14	0, 1,
Automotive	Best	0.91	136.99	-	1.91e+09	-	24.97	30.87	720.00	485.61	0, 0,
	Worst	0.75	127.45	-	1.02e+09	-	37.73	28.81	730.28	1261.34	0, 1,
Aerospace	Best	0.94	113.91	-	-	1.28e+08	21.27	31.26	686.82	506.13	1, 1,
	Worst	0.65	127.45	-	-	6.65e+07	37.73	28.81	730.28	1261.34	0, 1,
										[%] 1:	30, 60,
										[%] 0:	70, 40,
										[%] -:	0, 0,1

Figure 8.10: Example of Web-based DSE report

8.6 Conclusions

This chapter has presented DAVOS, an EDA toolkit that seamlessly integrates into the common semi-custom design flow to offer support for the dependability-driven processes, such as assessment, verification, optimisation (DSE), and selection (benchmarking). Its flexible and modular architecture makes it compatible with the standard HDLs, off-the-shelf EDA tools, and implementation technologies, enabling its extension to support any other processes (even those not dependability related). An interactive web-based interface provides custom query and visualisation features to ease the analysis of obtained data.

Currently, DAVOS offers native support for Xilinx's ISE toolchain, Xilinx's Vivado suite, Mentor Graphics' Precision RTL synthesiser and ModelSim simulator. Although several configuration files should be customised to put into practice the most demanding scenario (dependability-driven DSE), it can be achieved by following the guidelines described in this chapter.

The integration of any other EDA tool in the implementation module requires the development of custom Python functions to generate the appropriate command line scripts and to process the resulting reports. The definition of a new fault dictionary may be required to integrate a new simulator into the SBFI flow or to support a new implementation technology (macrocell library). Existing files for already supported tools can be used as templates.

Although several fault injection tools exist, they are rarely available. Thus, researchers usually develop their own tools and cross-comparison of results is barely possible.

DAVOS is published at `https://github.com/IlyaTuzov/DAVOS` as a free and open-source toolkit under the MIT license. In such a way, the community will benefit from the integration of new modules and EDA tools, and researchers could share their results in a compatible format.

A future work on DAVOS development may focus on: i) extending the toolkit so it can be easily configured through a web-based interface, ii) defining novel fault models for different implementation technologies, iii) extending the FFI tool to support devices other than Xilinx's 7-Series FPGA/SoC, and iv) improving the analysis and reporting capabilities.

Chapter 9

Experimental Evaluation

This chapter evaluates the efficiency and illustrates the usefulness of the proposed methods and tools through a case study of three soft-core processors (DUTs). The case study is structured into three parts according to the dependability-driven scenarios supported by DAVOS. Section 9.2 illustrates the dependability benchmarking of considered DUTs at different design representation levels. After benchmarking the DUTs under the default configuration of synthesis, mapping, placement and routing (EDA) parameters, Section 9.3 studies the impact of these parameters on the resulting dependability. To this end, it carries out several design space exploration experiments which quantify the contribution of individual EDA parameters towards each PPA and dependability metrics, and illustrate the dependability improvement attainable by means the proper tuning of these parameters. Finally, Section 9.4 illustrates the dependability assessment of a resilient HW design and verification of its fault mitigation mechanisms, in application to the instrumented version one of the DUTs (that will be selected at benchmarking). Within the context of aforementioned experimentation scenarios this chapter evaluates the speed-up gain attainable by means of proposed SBFI/FFI optimizations.

9.1 Introduction

This chapter illustrates the application of the proposed fault injection methodology to the dependability-driven processes of FPGA-based design flow through a case study of three soft-core processors (DUTs): an AVR IP core [144], an MC8051 IP core [126], and a Xilinx's Microblaze IP [181]. The case study comprises three parts. The first part considers the DUTs at three representation levels, illustrating how the proposed methodology improves the accuracy and performance of dependability benchmarking experiments. The second part of the case study illustrates how the PPA and dependability of considered DUTs can be improved by tuning the synthesis, mapping, and placement-routing parameters of a target EDA suite (Xilinx Vivado) through the proposed DSE approaches. Finally, the third part of the case study will instrument one of the DUTs (selected at benchmarking) with SEU mitigation mechanisms, in order to illustrate how the proposed methodology improves the dependability assessment and verification of resilient HW designs.

All the considered DUTs are very similar in size (complexity), and can be considered as fair alternatives from the benchmarking viewpoint. The top-level architecture of considered DUTs is provided in Fig B.1. All three DUTs are implemented using the Xilinx's Vivado 2018.3 suite onto the Zynq-7000 SoC (xc7z020).

All the DUTs run the same synthetic matrix manipulation workload, adapted for each DUT from the MiBench suite [66] in a reduced format. The workload is compiled by the C/C++ compiler corresponding to each DUT: (i) CA51 Kit of the Keil μVision compiler [11] for MC8051 IP, (ii) gcc-avr compiler for AVR IP, (iii) mc-gcc C/C++ compiler [180] from Xilinx SDK suite for Microblaze. Table 9.1 lists the number of clock cycles required by each DUT for initialization, workload execution, and readout of results.

Table 9.1: DUT simulation/emulation phases in clock cycles

DUT	DUT Initialization	Workload execution	Result readout
MC8051	0	35000	4000
AVR	0	14000	6000
Microblaze	0	25000	20000

The considered faultload will be detailed further in each part of this case study. It is worth noting, though, that the main fault model considered along the case study is a single event upset (soft error) in the cells of configuration memory (CM), as these faults are of higher priority for SRAM-based FPGA devices [72].

A DUT failure is defined as any mismatch of DUT processing results (resulting integer matrix) with those of the fault-free run (golden run), as well as the absence of such results after the predefined time-out. The source versions of the considered DUTs are not equipped with any error signalling mechanism, thus all their failures are reported as silent data corruption (SDC). After instrumenting the selected DUT with SEU mitigation and error detection mechanisms (in the third part of the case study), the failures will be classified into signalled failures and SDC.

9.2 Dependability benchmarking of soft-core processors

The dependability features of alternative soft-core processors can be assessed and compared at different design representation levels. Each representation level exhibits its specific advantages and limitations related to the accuracy of supported analysis and experimental effort. This section illustrates the dependability benchmarking process in application to the previously described soft-core processors (DUTs) implemented on the Xilinx 7-series SoC FPGA.

First, the dependability of considered DUTs will be assessed at the RT level, implementation level, and FPGA level. After that, the DUTs will be ranked attending to single-objective and multi-objective dependability-aware criteria. It will be shown why the decision regarding the selection of alternative soft-core design must be taken at the FPGA level. Finally, this section will discuss the accuracy and performance improvements attained by the proposed methodology with respect to the existing fault injection approaches.

9.2.1 Experimental procedure

The experimental procedure comprises three steps: (i) defining the DUT ranking criteria and the respective faultload, (ii) running DAVOS to implement the DUTs onto the target FPGA, and to estimate the required PPAD attributes at different representation levels, (iii) analysing the obtained dependability metrics, and ranking the DUTs.

The DUTs will be ranked attending to the individual PPAD attributes, as well as to the multi-criteria WSM scores. The considered PPAD attributes include:

- the workload execution time as an attribute of performance;
- the power consumption, estimated by means of Xilinx power analysis tool with prior simulation of switching activity,
- the utilization of Flip-Flops and LUTs as an attribute of silicon area,

- the SEU-related failure rate λ as an attribute of dependability of FPGA-based implementations.

The failure rate λ for each DUT is computed by aggregating the failure rates of different types of constituting components: Flips-Flops (λ_{FF}), Block RAMs (λ_{BRAM}), distributed memory (λ_{LUTRAM}), and configuration memory (λ_{CM}). It is worth noting that the first three components are attributed to the changeable memory of FPGA, whose SEUs may be recovered on their own by normal circuit operation. Whereas in the case of non-changeable CM the soft error persist in the CM cells until scrubbing or reconfiguration takes place. The failure rates are computed attending to the equation 2.3, in which the upset rate is assumed 75 FIT/Mb for CM and LUTRAM, 72 FIT/Mb for BRAM [179], 2 FIT/Mb for Flips-Flops [72]. Assuming the the harsh application environment (at high-altitudes), the failure rate derating factor is $K = 327.8$, as it is explained in [136]. Finally, the device vulnerability factor DVF is estimated by SBFI and FFI experiments, as the percentage of upsets that lead to a failure (SDC percentage).

The multicriteria ranking is based on the three weighted sum models, listed in Table 9.2. Each defined ranking criteria takes into account the PPAD attributes with different weight coefficients: mission critical score focuses on minimization of failure rate, cost-critical score aims at minimization of area, mobile score aims at minimizing the power consumption without sacrificing too much the performance.

Table 9.2: Weights of PPAD attributes in three considered multi-objective ranking scenarios

Scenario	Performance (Exec. Time)	Power Consumption	Area		Failure Rate (SER)
			FF	LUT	
Mission-critical	0.2	0.2	–	–	0.6
Cost-critical	0.2	0.1	0.3	0.3	0.1
Mobile	0.3	0.6	–	–	0.1

The DUTs are implemented by means of DAVOS implementation support tool in conjunction with the Xilinx Vivado suite. implementation support tool is configured for iterative adjustment of clock constraints to reach the maximum possible clock speed, under the default configuration of synthesis, mapping, placement-routing parameters. Only for the purpose of analysis of register mapping and LUT mapping results, two additional implementations are considered for MC8051 IP: (i) the one which limits the netlist fanout by a factor of 10 (default is 10000), and (ii) another one which disables the LUT combining. The resulting clock frequencies of default configuration are: 26.1 MHz for MC8051, 71.4 MHz for AVR, 142.8 MHz for Microblaze.

The resulting amount of utilized logic resources, and the corresponding fault targets at different representation levels are listed in Table 9.3. As it can be seen, the three soft-core processors are quite similar in terms of logic utilization. The most significant difference is the amount of instantiated BRAM, which however is not expected to lead to significant diversity in terms of dependability metrics, since only a small part of BRAM cells are actually used under the given workload. It is worth commenting that only Microblaze instantiates the distributed memory (LUTRAM) to implement the register file, while in the rest of DUTs the registers are implemented completely on Flip-Flops.

It can be also seen that under the default configuration of EDA parameters the registers are synthesized without logic replication, resulting in close to 100% match at register mapping. When limiting the fanout, many registers become replicated, thus reducing the match rate to just 67%. It is important to note that Microblaze is not available in source codes (as RTL model), but only as macro IP in Vivado suite, therefore it will be only targeted at implementation level (SBFI), and at FPGA level (FFI).

Table 9.3: Fault targets at different design representation levels

	RTL (signals)			Implementation-level (macrocells)			FPGA configuration memory (Kb)			
	Total	Matched Reg.[1]					Changeable		Non-changeable	
		bits	%	FF	LUT	BRAM	LUTRAM	BRAM	LUTs	Rest
MC8051										
- Default	1280	569	99%	576	2573	18	0	645.1	144.3	330.3
- Fan-out limit 10	1280	576 (521)[2]	67%	778	–	–	–	–	–	–
- No LUT combining	–	–	–	–	–	–	–	–	185.9	364.8
AVR	2461	423	100%	423	1747	2	0	73.7	98.6	244.3
Microblaze[3]	–	–	–	982	1303	32	4.0	1179.6	71.9	279.8

[1] RTL registers are located by means of register mapping procedure
[2] Non-replicated registers (direct match 1:1)
[3] Xilinx IP (source RTL model is not available)

The faultload considered at different representation levels for each DUT is listed in Table 9.4. First, the stuck-at-1/0 faults are analysed at RTL and implementation levels only to illustrate the gap in dependability estimates existing between these levels. These faults are not taken into account for the ranking of DUTs, as the latter focuses only on soft-errors. Stuck-at faults are injected following an exhaustive approach, since the population of fault configurations is reasonably small (stuck-at are injected at the workload start).

Table 9.4: Faultload considered at different design representation levels

Fault Model	Injection level			Sampling method	Inj. Time
	RTL (SBFI)	IMPL (SBFI)	FPGA (FFI)		
Stuck-at-1/0	✓	✓		Exhaustive	0
Bit-flip in registers	✓	✓	✓	Statistical $e_{SDC} = 0.5\%$	Random
Bit-flip in LUTRAM		✓	✓	Statistical $e_{SDC} = 0.1\%$	Random
Bit-flip in BRAM		✓	✓	Statistical $e_{SDC} = 0.1\%$	Random
Bit-flip in CRAM (all)			✓	Statistical $e_{SDC} = 0.1\%$	0
Bit-flip in CRAM (LUTs)		✓	✓	Exhaustive	0

Bit-Flips in the registers are injected at all representation levels. The fault space for the bit-flips includes all possible injection points (FFs) at each clock cycle of the workload, resulting in 20.16, 5.92, and 13.75 millions of fault configurations for MC8051, AVR, and Microblaze respectively. Accordingly, bit-flips will be injected following the iterative statistical approach, with the goal of reaching the 0.5% error margin for SDC percentage. The fault space for the rest of changeable memory cells (LUTRAM and BRAM) is even larger, while their SDC percentage is expected to be much lower than that of FFs due to the low utilization rate of LUTRAM/BRAM cells. Thus, to keep a statistical significance of derived results, bit-flips into LUTRAM and BRAM are injected with a narrower error margin goal of 0.1%.

Unlike changeable memory, the upsets in non-changeable CM are not recovered by normal circuit operation. As such, they can be treated as permanent (in the absence of CM scrubbing), and can be injected at the beginning of the workload. This results in the smaller fault space, which nevertheless still accounts for 0.35 to 0.50 millions of fault configurations. Non-changeable CM cells are split in two subsets: (i) those that store the configuration of netlist macrocells (mainly LUT content), and (ii) the rest of them that configure the interconnection and routing. The former subset accounts for 20% to 30% of CM cells, and is reflected in the post-implementation netlist in form of INIT_reg attributes of corresponding macrocells; their upsets can be thus analysed both by means of implementation-level SBFI, and by means of FFI experiments. The larger subset of CM cells (configuring the routing) is not reflected in the post-implementation model. The upsets in these CM cells can be thus analysed only at FPGA level by means of FFI experiments. Under the absence of any hypothesis regarding the possible magnitude of SDC percentage, the conservatively narrow error margin of 0.1% has been selected as a goal for the statistical injection into the CM. Additionally, to verify the LUT profiling approach, a subset of CM cells attributed to the LUT content will be targeted following an exhaustive injection approach.

The scheduled SBFI and FFI experiments have been executed using the DAVOS evaluation engine. SBFI experiments are performed on the Grid computing platform, in which the fault injection campaign has been split into 100 parallel jobs. To perform the FFI experiments the evaluation engine has been equipped with two Zynq ZC702 boards, thus allowing to evaluate the dependability of two DUTs in parallel.

9.2.2 Fault injection results and dependability metrics

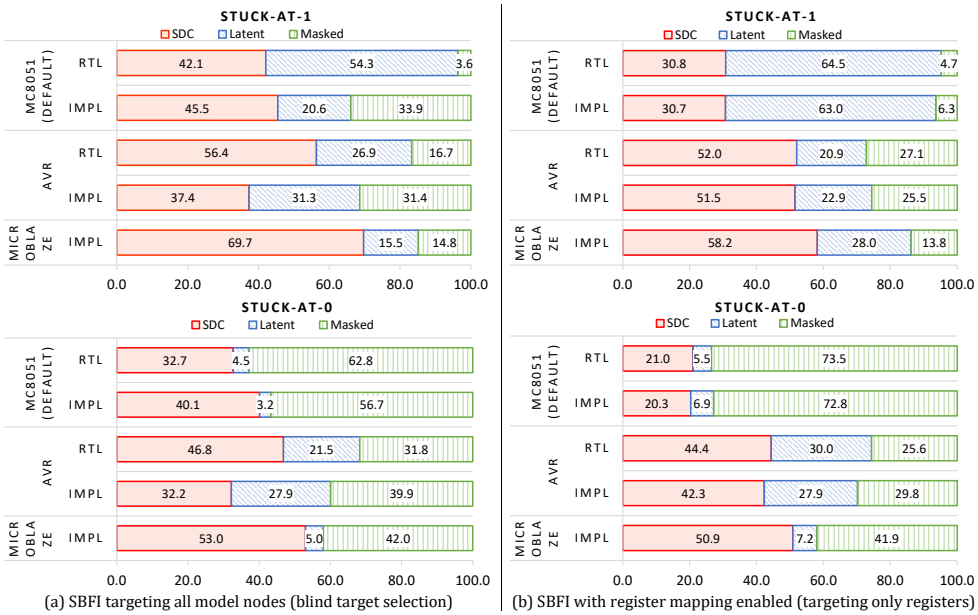
The datasets, containing the resulting fault injection logs with an attached reporting interface for each DUT, are publicly available at [161] (DOI of the dataset: 10.5281/zenodo.3996297). This subsection analyses the obtained results.

9.2.2.1 Stuck-at faults

The stuck-at faults have been first analysed following a common approach, in which all model nodes available at both RTL and implementation levels are targeted without distinguishing between sequential and combinational logic. Fig. 9.1-a illustrates a huge gap existing between RT-level and implementation-level results, obtained following this blind approach. In some cases (AVR) the RT-level SBFI underestimates the SDC by 20 percentage points. By enabling the mapping of registers in DAVOS this gap is reduced to less than 2 percentage points in all cases. As it can be seen from the Fig. 9.1-b, the distribution of failure modes for the stuck-at faults in mapped registers at RT-level is very close to that at implementation level. Minor discrepancies in the percentage of latent errors (in case of AVR) can be explained by the existence of additional (replicated/inferred) registers at the implementation level.

9.2.2.2 Bit-Flips in registers

The bit-flips in registers can be analysed in the proposed methodology at three representation levels: RTL, implementation (IMPL), and FPGA (FFI). Fig. 9.2-A illustrates the resulting distribution of failures modes for each DUT. As it can be seen, the results obtained at different levels match quite well: the difference of SDC percentage between all representation levels is less than the sampling error (0.5%) in all cases. Some minor discrepancies (less than 3 percentage points) have been observed only for the latent errors. They can be partially explained by the impact of synthesis/implementation-time optimizations that are not related to the register replication/removal, and partially by the impact of timing delays



(a) SBFI targeting all model nodes (blind target selection)

(b) SBFI with register mapping enabled (targeting only registers)

Figure 9.1: Distribution of failure modes estimated for the stuck-at-1/0 faults by means of RT-level SBFI and implementation-level SBFI, when blindly targeting all model nodes (a), when enabling the register mapping to target only the sequential logic at both levels (b)

at implementation level. The resulting failure rate λ_{FF} (depicted in 9.2-B) also matches quite well between all representation levels.

Obtained results indicate that under the default configuration of EDA parameters, analysis of bit-flips in registers can be representatively accomplished at any description level. Register mapping in this case is useful to automatically locate the relevant fault targets, i.e. those RTL signals that correspond to the inferred FFs and latches. Under the aggressive register optimization/replication one can not rely solely on RT level SBFI for estimation of bit-flips. For instance, in case of MC8051 with limited fanout 33% of registers were replicated, and the multi-level SBFI has been used instead of pure RT-level SBFI. In this case all replicated registers (detected by mapping) were targeted at implementation level, while the rest of them were targeted at RTL. As it can be seen from the Fig 9.2, the multi-level results match quite well with those obtained by pure implementation-level SBFI.

Apart from analyzing the robustness of the design as a whole, a more fine-grained analysis can be performed by means of DAVOS. For instance, from the distribution of SDC percentage along the Microblaze design tree (depicted in Fig. 9.3)

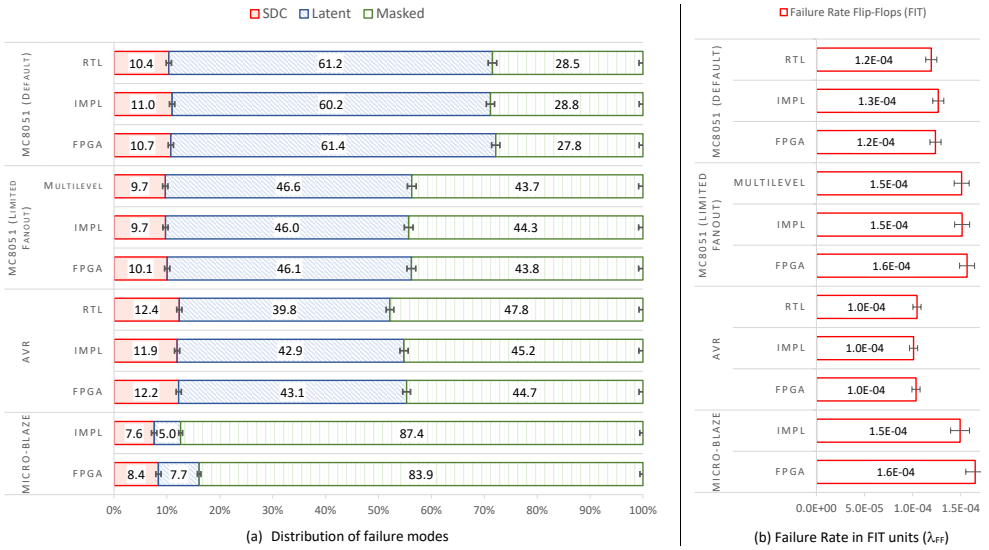


Figure 9.2: Bit-flips in registers at different representation levels: distribution of failure modes and estimated failure rate

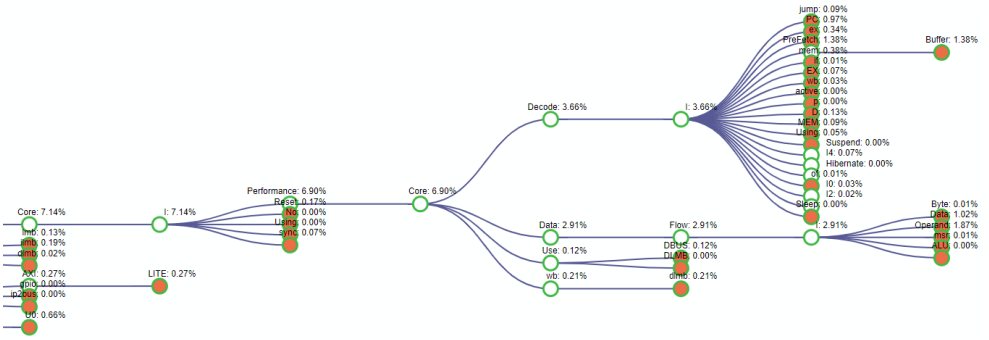


Figure 9.3: Contribution of Microblaze modules into SDC percentage (estimated by FFI)

it can be seen that the DUT scopes that are most vulnerable to the register bit-flips are operand select (1.87% of SDCs out of total 8.40%), and instruction fre-fetch (1.38% SDC)). Likewise, the most vulnerable AVR module is the register file, which contributes 7.7% of SDCs out of total 12.2%. The MC8051 registers that contribute the most into the failure rate are *R0*, *R1*, and *PC* (located in the control memory), being they responsible for 2.9%, 1.6% and 2.5% of SDC respectively (out of total 10.1%).

9.2.2.3 Bit-Flips in LUTRAM and BRAM

Bit-flips in distributed memory (LUTRAM) and in block memory (BRAM) have been analysed at the implementation level (SBFI), and at the FPGA level (FFI). It is worth noting that among the considered DUTs only Microblaze utilizes the LUTRAMs.

The resulting distribution of failure modes for the bit-flips in LUTRAM, and corresponding failure rates λ_{LUTRAM} are depicted in Fig. 9.4. As it can be seen, there are no statistically significant discrepancies in the distribution of failure modes between the two representation levels. The percentage of SDC is roughly three times less than in case of registers, while the resulting failure rate is two orders of magnitude higher. This is explained by the higher number of targeted bits in LUTRAM, as well as by the higher FIT per megabit in case of LUTRAM.

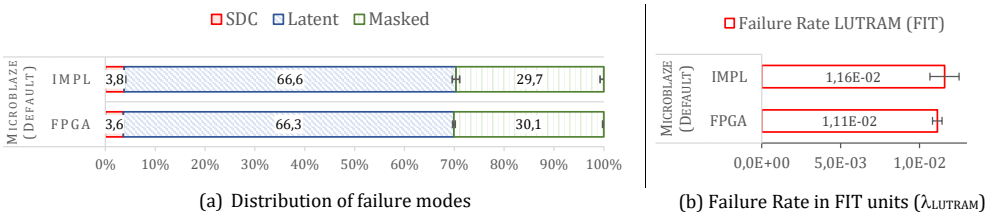


Figure 9.4: Bit-flips in LUTRAM obtained by SBFI and FFI: distribution of failure modes and estimated failure rate

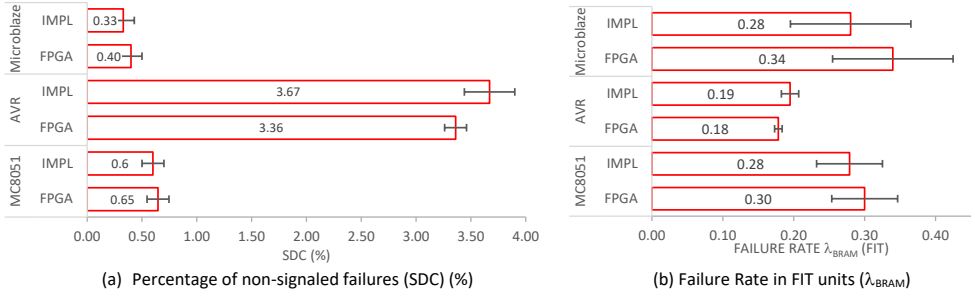


Figure 9.5: Bit-flips in block RAM obtained by FFI: distribution of failure modes and estimated failure rate

Fig. 9.5 illustrates the resulting percentage of SDC and the corresponding failure rate (λ_{BRAM}) for the bit-flips in BRAM. It can be seen that the difference between the SBFI and FFI results does not exceed the sampling error. Estimated SDC percentage of MC8051 (0.42%) and Microblaze (0.65%) is much lower than that

of AVR (3.36%). This is due to the fact AVR has less addressable BRAM, but its utilization rate is much higher that in the case of MC8051 and Microblaze. At the same time, the resulting failure rate (λ_{BRAM}) computed on the basis of SDC percentage by equation 2.3, does not differ that much between the DUTs. Furthermore, AVR is seemingly more robust to BRAM bit-flips (0.18 FIT) than MC8051 (0.30 FIT) and Microblaze (0.35 FIT).

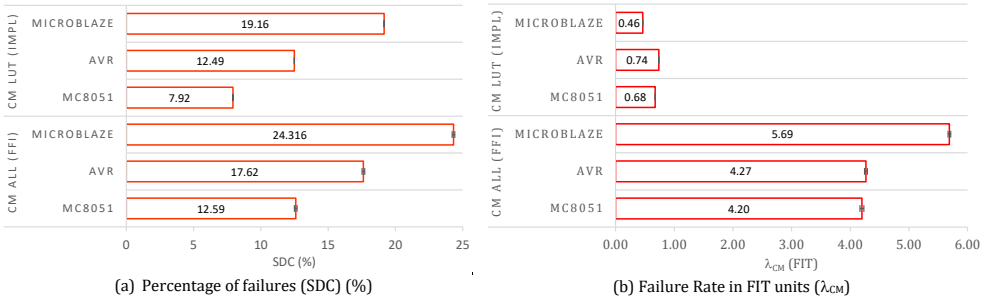
9.2.2.4 Upsets in non-changeable configuration memory

The non-changeable CM cells are divided into two subsets: those that configure the combinational logic (LUT content), and the rest of them (mainly configuring the routing). The CM cells attributed to the LUTs can be targeted both at the level of implementation-level HDL model, and at the level of FPGA prototype. Those CM cells that configure the routing are not reflected in HDL models (even in implementation-level ones), and can only be targeted at FPGA level.

Fig. 9.6 illustrates the resulting SDC percentage and the corresponding failure rate (λ_{CM}), estimated for the LUT-specific CM cells at the implementation level, as well as for the entire set of CM cells at FPGA level. In terms of SDC percentage the Microblaze is the least robust DUT at both description levels, while the MC8051 is the most robust one. In terms of resulting failure rate the situation is more complex. With respect to the LUT-specific CM cells, the Microblaze is better (exhibits lower failure rate) than the rest of DUTs. However, with respect to the entire set of CM cells at the FPGA level, the Microblaze is significantly worse (has higher failure rate) than the rest of DUTs. It is also important to note that the contribution of LUT-specific CM cells into the total CM-related failure rate is quite low, ranging between 8% and 18%. Likewise, by comparing these results with the previously presented ones, it can be seen that the CM-related failure rate is more than an order of magnitude higher than that of BRAM, LUTRAM and FF put together.

Additionally, the LUT-specific CM cells have been targeted exhaustively, after profiling their activity time. It is worth noting, that all non-essential LUT bits have been filtered-out before SBFI/FFI experiments by means of LUT mapping.

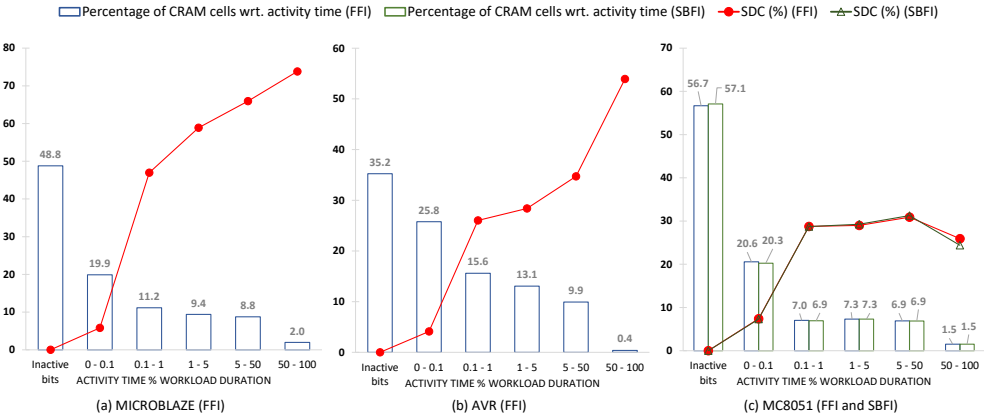
Fig. 9.7 illustrates the percentage of LUT-specific CM cells with respect to the profiled activity time, splitting them into several groups: inactive cells, cells that are active less that 0.1% of workload duration, cells that are active 0.1% to 1% of workload duration, and so on. For each group it plots the corresponding SDC percentage, estimated by means of SBFI and FFI experiments. As it can be seen, the major part of CM cells remain inactive (up to 57% in case of MC8051).



(a) Percentage of failures (SDC) (%)

(b) Failure Rate in FIT units (λ_{CM})

Figure 9.6: Robustness estimates obtained for the bit-flips in non-changeable CM at the level of implementation-level HDL model (LUT-specific CM cells), and at the FPGA level (all essential CM cells)



(a) MICROBLAZE (FFI)

(b) AVR (FFI)

(c) MC8051 (FFI and SBFI)

Figure 9.7: Percentage of LUT bits with respect to the profiled activity time, and respective percentage of failures (SDC)

Conducted FFI and SBFI experiments confirm that their upsets never lead to a failure. For the rest of LUT bits it can be seen that groups with higher activity time tend to be more critical (show higher SDC percentage) than the groups with the lower activity time. It can be noted, though, that in the case of MC8051 this correlation is less pronounced than in the case of Microblaze, and AVR. Overall this supports the hypothesis that the higher is the activity time of a certain LUT bit, the more critical it becomes, i.e. it is more likely that its upset will cause a DUT failure.

From the practical viewpoint this has two implications. First, inactive LUT bits can be safely left out from consideration at SBFI/FFI experiments, since their

upsets can be a-priory accounted as masked. Second, when the SBFI/FFI experiment aims at identification of most critical CM bits in less possible experimentation time, it is worth to target with the higher priority those CM cells, whose activity time is higher.

Finally, thanks to the bit-accurate LUT mapping, it becomes possible to cross-compare the SBFI results with the FFI results. This analysis has been performed for the MC8051. It can be noted that the number of fault targets at implementation-level SBFI is slightly lower than that at FFI; this is due to the fact that some LUT bits are mapped onto several CM cells in case of LUT combining with non-shared inputs (as explained in Section 5.2.1). Fig. 9.7-C illustrates the obtained SBFI results alongside with FFI results. It can be seen, that SBFI results per activity group match quite well with FFI results. At the bit granularity 99.9% of SBFI runs resulted with the same failure mode as corresponding FFI runs. The minor discrepancy of 0.1% can be explained by the imperfections of annotated timing model (SDF). The overall 99.9% match indicates the uniformity of injection procedures deployed at the level of implementation-level HDL model, and at the FPGA level.

9.2.3 Ranking of DUTs

The considered DUTs can be compared on the basis of dependability and PPA attributes available at different design representation levels. Table 9.5 summarizes the obtained PPAD attributes, in order to determine which soft-core processor better satisfies the design goals when implementing them on the selected FPGA.

The RTL models are least informative from the benchmarking viewpoint, since they don't reflect any implementation-specific details. In such a way, the area and power consumption attributes are unavailable at this level. The performance can be compared in terms of clock cycles required for the workload execution – the AVR is the best DUT from this viewpoint. The dependability comparison at this level can take into account only the dependability metrics of the sequential logic. The AVR is the best DUT in this category, since it provides the lowest register-related failure rate λ_{FF}. Finally, the Microblaze cannot be compared to the rest of DUTs at this level at all, since its RTL model is not available.

At the level of implementation-level HDL models (after the design is synthesised, placed and routed), the comparison of DUTs becomes much more informative, and all three DUTs can be now taken into account. The best DUT in terms of area is AVR, since it utilizes the least total number of BELs (FFs and LUTs). The performance now takes into account the maximum clock frequency reached by each

Table 9.5: Comparison of considered DUTs attending to individual PPAD metrics and WSM scores

		Benchmark Metric	Benchmark circuits		
			MC8051	AVR	Microblaze
RTL	Performance	Workload exec. time (cycles)	35000	14000	-
	Dependability	Failure rate registers λ_{FF} (FIT)	0.00012	0.00010	-
Implementation-level HDL model	Area	FF BELs	576	423	982
		LUT BELs	2187	1484	1089
	Performance	Frequency (MHz)	26.1	71.4	142.9
		Workload exec. time (ms)	1.34	0.20	0.17
	Power	Consumption (W)	0.015	0.022	0.033
	Dependability	Failure rate λ (total, FIT)	0.976	0.914	**0.814**
		λ_{FF}	0.00012	0.00010	0.00017
		λ_{LUTRAM}	0.000	0.000	0.011
		λ_{BRAM}	0.300	0.178	0.340
		λ_{CM_LUT}	0.676	0.736	0.463
FPGA prototype	Area	FF BELs	576	423	982
		LUT BELs	2187	1484	1089
	Performance	Frequency (MHz)	26.1	71.4	142.9
		Workload exec. time (ms)	1.34	0.20	0.17
	Power	Consumption (W)	0.015	0.022	0.033
	Dependability	Failure rate λ (total, FIT)	4.499	**4.446**	6.04
		λ_{FF}	0.00012	0.00010	0.00017
		λ_{LUTRAM}	0.000	0.000	0.011
		λ_{BRAM}	0.300	0.178	0.340
		λ_{CM_ALL}	4.199	4.268	5.690
		MTTF (h)	6.78×10^5	6.86×10^5	5.04×10^5
		Mission Time (R_{Th}=0.999) (h)	678	686	504
	Score	Mission-Critical	0.82	0.91	0.73
		Cost-Critical	0.59	0.87	0.75
		Mobile	0.74	0.78	0.65

DUT on the selected FPGA; Microblaze is the best in this category since it features the fastest absolute workload execution time. It is interesting to note that in terms of performance the Microblaze and AVR outscore the MC8051 nearly by an order of magnitude. In terms of power consumption the best solution is MC8051, which is explained by its much lower clock speed in comparison to other DUTs. The comparison of dependability now takes into account the estimates obtained for all technology-specific sequential and combinational macrocells, namely the failure rate of FF, LUTRAM, BRAM, and of the LUT-related CM cells. As it can be seen, in this category the Microblaze in this category is better than the rest of DUTs by 10% to 20%.

Switching to the level of FPGA prototype doesn't change the ranking from the viewpoint of performance, power, or area, but notably changes it from the view-

point of dependability. The only dependability component that can be additionally taken into account at the FPGA level with respect to the implementation-level model, is the one related to the routing-specific CM cells. This additional component, however, has a dominating impact on the resulting failure rate, increasing it by 5 to 7 times with respect to the total failure rate estimated at the level of implementation-level model. Accordingly, as it can be seen from the Table 9.5, the best (lowest) failure rate is provided by AVR (4.446 FIT), which is also quite close to the failure rate of MC8051 (4.499 FIT). Both of them outscore the failure rate of Microblaze by roughly 25%. All the reliability metrics derived on the basis of the failure rate (MTTF and mission time), lead to the same ranking.

From the Table 9.5 it can be also noted that the resulting failure rate is mostly determined by the CM upsets λ_{CM} (93% to 96% contribution). This indicates that non-changeable memory should be given the highest priority in dependability assessment, as well as in protection of considered DUTs against the soft errors (SEUs).

Finally, under the multi-objective ranking AVR receives the highest score in all three defined scenarios: mission-critical (0.91), cost-critical (0.87), and mobile (0.78). Accordingly, since providing both the best dependability, and the best trade-off between dependability and the rest of PPA attributes, AVR is selected as the best soft-core processor among the considered alternatives for implementation on the selected FPGA.

9.2.4 Experimental effort and speed-up

As it has been explained in Chapter 6, the experimental effort is optimized in two ways: (i) by accelerating the execution of individual injection runs through the checkpointing and multi-level SBFI, and (ii) by reducing the total number of required fault injection runs through the optimization of essential bits, profiling, and iterative statistical fault injection.

Table 9.6 lists the mean SBFI and FFI time per injection run, measured for the considered DUTs in the absence of any optimization. As it could be expected, the FFI experiments provide the lowest time per injection run, being FFI roughly two orders of magnitude faster than RT-level SBFI, and four orders of magnitude faster than implementation-level SBFI. When scaling these numbers for the size of injection campaign, the resulting experimental time may seem very high, or even prohibitive in case of implementation-level SBFI. For instance, non-optimized SBFI experiments for MC8051 would require 32/58 hours for the analysis of bit-flips in registers (for the default implementation and implementation with limited

fanout respectively), 802 hours for the bit-flips in BRAM, and 87 hours for the upsets in LUTs.

As it can be seen from the Table 9.7, by applying the proposed optimization, these numbers are reduced to just 0.05 and 3.9 hours for the bit-flips in registers (default implementation and implementation with limited fanout), providing a speed-up factor of 640 and 7.8 respectively, to just 12.6 hours for BRAMs (speed-up factor of 64), and to 26 hours for LUTs (speed-up factor of 3.4). A slightly lower, but also notable speed-up is achieved in case of FFI experiments. The contribution of each optimization into the resulting speed-up is further analysed in this subsection.

Table 9.6: Non-optimized SBFI and FFI time (per injection run)

DUT	Mean time per injection run (sec) [std. deviation]			
	SBFI RTL	SBFI Implementation	FFI (trace latent errors = off)	FFI (trace latent errors = on)
MC8051 default	1.6 [0.5]	301 [93]	0.008 [0.000]	0.026 [0.000]
MC8051 limited fanout	1.6 [0.5]	490 [108]	0.009 [0.001]	0.018 [0.001]
AVR	2.0 [0.1]	72 [8]	0.005 [0.001]	0.026 [0.000]
Microblaze	–	421 [19]	0.007 [0.000]	0.055 [0.000]

Table 9.7: Estimated non-optimized and optimized (resulting) experimental time per MC8051 fault injection campaign

Fault model/ Target logic	Non-optimized time (h)	Optimized time (h)	Applied optimizations				
			Multilevel injection	Check-pointing	LUT mapping	Profiling	Iterative sampling
Bit-flip/Registers							
Default (SBFI*)	32.2	0.05	✓	✓			✓
Limited fanout (SBFI)	51.8	3.9	✓	✓			✓
Bit-Flip/BRAM (SBFI)	802.7	12.6		✓			✓
Bit-Flip/LUT content							
Default (FFI)	1.04	0.59			✓	✓	
No_LC (SBFI)	87.3	26.0			✓	✓	
No_LC (FFI)	1.34	0.69			✓	✓	
Upset/CRAM (FFI)	2.29	1.56			✓		✓

* SBFI experiments are performed with parallelization factor of 100 (using Grid computing cluster)

Before analysing the speed-up provided by each optimization, it is worth to make several observations. First, the total time per SBFI/FFI campaign scales proportionally to the used number of computing nodes (parallel simulation/emulation processes). In this case study SBFI experiments have been carried out using the Grid computing cluster with a maximum of 100 parallel simulation jobs. It

should be noted, however, that the effective parallelization factor in many cases remained lower than the desired 100, since some simulation jobs were waiting for availability of computing nodes (being scheduled by Sun Grid Engine).

Therefore, to properly estimate the speed-up provided by each proposed optimization, the estimations of experimental time provided in this section are based on the pure runtime logged by the simulator for each individual injection run. The estimations of total run time assume the optimistic parallelization factor of 100. Likewise, FFI experiments have been distributed between two available ZC702 evaluation boards for evaluating two different DUTs in parallel. Therefore the FFI parallelization factor for each individual DUT is assumed equal to 1.

Second important observation concerns the impact of DUT tracing procedure (analysis of failure modes) on the resulting experimental time. As it can be seen from the Table 9.6, enabling the tracing of latent errors increases the FFI run time by 2 to 8 times with respect to FFI that analyses only DUT responses (failures). This is explained by the relatively high time overheads (in comparison to the workload execution time) required to readback those frames that contain the state elements (FF, LUTRAM, BRAM), and to compare them with the reference trace. Especially notable this overhead becomes when the state frames are numerous and sparse, since at least on the selected platform (Xilinx SDK 2018.3, ZC702 evaluation board), the PCAP was unable to readback multiple non-aligned (sparse) frames in a single transaction, thus requiring to split the readback in multiple PCAP transactions. Accordingly, when the computation of dependability attributes doesn't take latent errors into account (e.g. the SDC percentage estimated in this section), the tracing of DUT internal state can be disabled in the configuration of DAVOS FFI tool for the sake of much higher FFI performance. In case of SBFI the time overheads imposed by the DUT tracing are significantly less pronounced, generally not exceeding a factor of 1.5 (with respect to SBFI that traces only DUT responses).

9.2.4.1 Checkpointing

The speed-up gain provided by the clustering checkpoints (using 20 clustering intervals) in implementation-level SBFI experiments ranges between 1.32 and 1.67. This result generally coincides with the expected speed-up, estimated by DAVOS (at workload generation) using the model defined in Listing 6.2. Table 9.8 lists the optimized time per injection run for each DUT, and compares the expected speed-up factors with the experimentally obtained ones.

The speed-up gain in case of RT-level SBFI is lower than in the case of implementation-level SBFI. This is explained by the relatively low simulation effort at RT level, under which the checkpoint recovery time (around 0.5 to 1 seconds) takes a notable part of total run time (1.0 to 2.0 seconds). Nevertheless, since the checkpoint recovery is slightly faster than the complete simulator restart, the checkpointing may only improve and never penalise the SBFI performance.

Table 9.8: SBFI time under the checkpointing optimization enabled (20 clustering intervals), expected and resulting speed-up factors

DUT	SBFI RTL		SBFI Implementation	
	Time per inj. run (sec) [std. deviation]	Speed-up experimental/ expected	Time per inj. run (sec) [std. deviation]	Speed-up experimental/ expected
MC8051 default	1.2 [0.5]	1.33 / 1.60	181 [93]	1.67 / 1.69
MC8051 limited fanout	1.2 [0.5]	1.33 / 1.60	299[149]	1.64 / 1.69
AVR	1.6 [0.5]	1.25 / 1.52	48 [16]	1.50 / 1.52
Microblaze	–	–	320 [74]	1.32 / 1.35

Regarding FFI checkpointing, the time required to reload the CM frames with state elements and user memories from a snapshot (up to 20 ms depending on the number and sparsity of CM frames) exceeded the workload execution time itself (5 ms to 26 ms). For that reason, no speed-up gain could be obtained from FFI checkpointing in this case study.

9.2.4.2 Multilevel fault injection

As it has been previously shown in this section, to obtain the accurate dependability estimations by means of RTL-level SBFI, it is necessary to locate those nodes of RTL model that represent the sequential logic, as well as to take into account the logic optimizations performed at synthesis. The proposed register mapping technique has automated this process. When the percentage of mapped registers is close to 100% (under the default configuration of synthesis parameters), the very costly implementation-level SBFI can be safely replaced by much faster RT-level SBFI. As it can be seen from the Table 9.9, the attainable speed-up gain for the considered DUTs ranges between 30 (AVR) and 150 (MC8051 default).

Customization of synthesis parameters may lead to the aggressive optimization or replication of registers, thus significantly reducing the register match rate. In the presented case study this has been illustrated by the implementation of MC8051 with the limited fanout, in which roughly 33% of registers have been replicated. In this case the faultload must be distributed between the RTL and implementation-

Table 9.9: Speed-up gain achieved by multi-level SBFI with respect to implementation-level SBFI when evaluating the effects of bit-flips in registers

DUT	Matched registers	Experimental Time* (hours)		Speed-up
		Implementation	Multi-level	
MC8051 default	100%	7.54	0.05	150.8
MC8051 limited fanout	67%	11.63	3.87	3.0
AVR	100%	2.27	0.08	30.0

level SBFI in such a way that non-optimized registers are targeted at RTL, while the optimized/replicated registers are targeted at implementation level. As it has been shown in Fig 9.2, this multi-level SBFI provides the same dependability estimates as the purely implementation-level SBFI, while the experimental effort is reduced by 3.0 times: 3.87 hours for multi-level SBFI compared to 11.63 hours for implementation-level SBFI (see Table 9.9).

9.2.4.3 Optimized essential bits and profiling

Bit-accurate LUT mapping and profiling of switching activity are two optimizations that reduce the number of injection runs when targeting the combinational logic of FPGA. The LUT mapping identifies those LUT bits that are actually essential for the targeted design, reducing the number of fault targets with respect to the redundant essential bits reported by Vivado suite. The profiling determines which of those essential bits remain inactive under the given workload, and thus can be removed from the fault list, as their upsets do not impact the DUT behaviour.

Table 9.10: Percentage of CM essential bits filtered-out by LUT mapping and profiling, and the resulting speed-up

	Xilinx Essential bits	Optimized essential bits		Active optimized Essential bits		Non-optimized run time (hour)*	Overheads (hour)		Optimized run time (hour)*	Speed-up factor
	(Kb)	Abs (Kb)	Reduction (%)	Abs (Kb)	Reduction (%)		LUT mapping	Profiling		
MC8051 default (FFI)	144.4	113.8	-21.2%	52.9	-63.3%	1.04	0.02	0.19	0.59	**1.75**
MC8051 no_lc (FFI)	186.0	125.7	-32.4%	54.8	-70.6%	1.34	0.02	0.28	0.76	**1.93**
MC8051 no_lc (SBFI)	186.0	125.7	-32.4%	54.8	-70.6%	87.3	0.02	0.28	26.0	**3.36**
AVR	98.6	78.6	-20.3%	50.9	-48.4%	0.58	0.02	0.07	0.39	**1.48**
Microblaze	71.9	32.2	-55.2%	16.5	-77.1%	1.10	0.02	0.14	0.41	**2.69**

* based on mean time per injection, assuming parallelization factor 100 for SBFI, 1 for FFI

Table 9.10 lists the speed-up provided by these optimizations under the exhaustive simulation/emulation of LUT upsets for the considered DUTs. As it can be seen, the LUT mapping has filtered-out 21%, 20% and 55% of LUT-specific essential

bits, reported by Vivado for MC8051, AVR, and Microblaze respectively. When disabling the LUT combining at implementation of MC8051, the percentage of filtered-out bits increases to 32%, being the absolute number of essential bits very similar to those of the default implementation (55 Kb and 53 Kb respectively). By means of FFI experiments it has been verified that bit-flips in these filtered-out bits don't impact the DUT behaviour, thus being non-essential.

In this case study 35% (AVR) to 56% (MC8051) of remaining essential bits were reported as inactive by the profiling of switching activity. Thus, the resulting reduction of faultload (with respect to the essential bits reported by Vivado) amounted to 63% for MC8051, 48% for AVR, and 77% for Microblaze. When taking into account the time overheads for LUT mapping and profiling, the resulting speed-up factor in this case study reaches 2.7 for FFI experiments, and 3.4 for SBFI experiments. It is worth noting, however, that LUTs account for less that 30% of total essential bits. Therefore with respect to the entire set of essential bits the resulting reduction of fault targets amounted to just 19% for MC8051, 14% for AVR, and 16% for Microblaze. A further research is thus required to analyse the possibility of similar optimizations for the routing-related essential bits (which can be targeted only by means of FFI experiments).

9.2.4.4 Iterative statistical fault injection

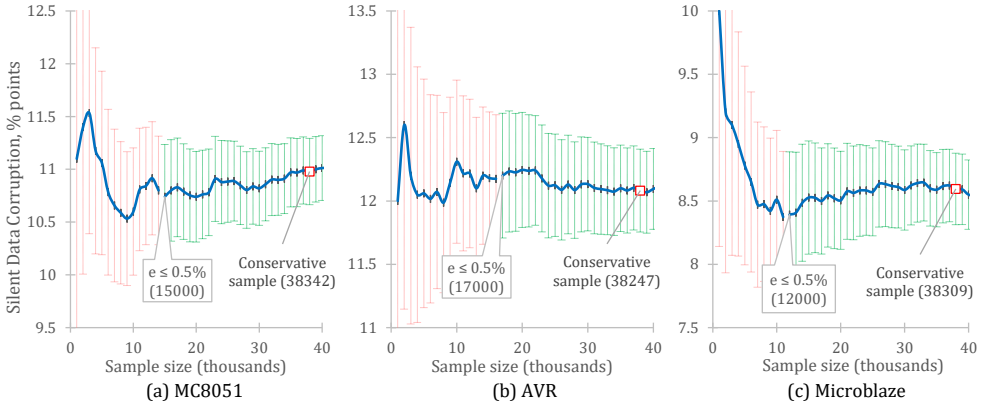


Figure 9.8: SDC confidence intervals obtained by iteratively appending 1000 fault configurations to the sample, in comparison to conservative statistical approach (bit-flips in registers, error margin threshold 0.5%)

Iterative statistical fault injection reduces the number of fault injection runs with respect to the common (conservative) approach by periodically recomputing the

robustness estimates along with their respective error margins, and thus terminating the experiment as soon as all error margins reach the predefined threshold. As it has been explained in Section 6.2.2, the further from 50% is the robustness estimate, the higher speed-up gain the iterative approach is expected to provide with respect to the common (conservative) approach. Fig. 9.8 illustrates speed-up gain attained by this approach for the case of bit-flips in registers (estimated by FFI). DAVOS FFI tools keeps appending 1000 of fault configurations (injection runs) to the sample, until the error margin of SDC metric reaches a threshold of 0.5%. This required 12000 (Microblaze) to 17000 (AVR) of injection runs to reach the desired error margin (under the 95% confidence). Whereas under the conservative sampling each DUT would require roughly 38000 of injections runs. The resulting speed-up factor, measured during the injection of bit-flips, ranges between 2.2 and 3.2 (AVR and Microblaze respectively).

Table 9.11: Speed-up attained by iterative statistical FFI at dependability benchmarking in comparison to the conservative sampling approach

SBFI/FFI Experiment	Conservative sample* / Iterative sample* / (Speed-up)		
	MC8051	AVR	Microblaze
Upsets in registers ($e = 0.5\%$)	38 / 15 / (2.5)	38 / 17 / (2.2)	38 / 12 / (3.2)
Upsets in LUTRAM ($e = 0.1\%$)	–	–	951 / 138 / (6.9)
Upsets in BRAM ($e = 0.1\%$)	960 / 25 / (28.4)	960 / 125 / (7.7)	960 / 17 / (56.5)
Upsets in CRAM ($e = 0.1\%$)	303 / 216 / (1.4)	242 / 205 / (1.2)	236 / 217 / (1.09)
Total	1302 / 256 / (5.1)	1240 / 347 / (3.6)	2185 / 390 / (5.7)

* Thousands of fault injection runs

Table 9.11 summarizes the speed-up gain provided by the iterative statistical injection (with respect to the conservative approach) for the dependability benchmarking of considered DUTs. As expected, the lower is the resulting robustness metric, the more beneficial becomes the iterative injection in comparison to the conservative approach. The maximum speed-up (ranging between 7.7 and 56.5) has been achieved in case of BRAM bit-flips, since the resulting robustness estimate (SDC percentage) remained below 1%. On the opposite, the lowest speed-up up (1.09 to 1.4) has been observed in case of CM upsets. This is explained by the combination of strict error margin (0.1%), with relatively small population size (200 to 480 thousands of fault configurations) and high SDC percentage (12.5% to 24.0%), that have lead to the need to sample a significant part of population (50% to 70%) both when following the conservative and iterative approaches. Based on the total number of injection runs for each DUT, it can be concluded that iterative fault injection has accelerated the dependability benchmarking by 3.6 to 5.7 times with respect to the common statistical injection approach.

9.2.5 Discussion

It is worth commenting several practical implications following from the presented results.

First, the case study has confirmed the gap (up to 20 percentage points) that exists between the RT-level and implementation-level SBFI analysis, when blindly targeting all design nodes available at these levels. To obtain more accurate robustness estimations at RTL it is necessary to locate those RTL nodes that correspond to the inferred (technology-specific) sequential macrocells, and take into account the logic optimizations performed during the synthesis and implementation. The proposed register mapping technique automates this process, and reduces the gap between RTL and implementation-level estimations to less than 2 percentage points. As a result, the robustness assessment of sequential logic can be carried-out by means of fast RT-level SBFI with the accuracy close to that of implementation-level SBFI and FFI experiments. In the case of high percentage of optimized/replicated registers (reported by the register mapping) the multi-level SBFI must be followed instead of RT-level SBFI in order to obtain more accurate estimations, while still reducing the experimental effort with respect to the pure implementation-level SBFI.

Second, the overall contribution of non-changeable CM (both routing and LUT-specific) into the resulting failure rate may exceed 90%. On the one hand, this may be specific to the selected HW designs and workload. For instance, different workload may utilize more of allocated changeable memory (BRAM, LUTRAM, FFs), and this may significantly increase its contribution into the resulting failure rate. On the other hand, in the absence of any SEU protection mechanism the upsets in non-changeable CM should be still considered as potentially more severe, since they affect the functionality/integrity of the circuit itself, and unlike changeable memory can't be mitigated by error correction codes. Therefore, the dependability benchmarking of FPGA-based designs, similar to those considered in the case study, should primarily focus on the non-changeable CM.

Third, most technology-specific faults can be accurately analysed both by means of implemenation-level SBFI and FFI experiments, like for instance the upsets in FF, BRAM, LUTRAM and LUT-specific CM. Nevertheless, the dependability-aware selection of FPGA-based designs should rely on FFI experiments, because only at FPGA level it becomes possible to take into account the failure rate of routing-related CM cells that have a dominating contribution into the total failure rate of FPGA implementations.

Fourth, it has been shown, that the bit-accurate LUT mapping locates the LUT-specific essential bits with finer granularity than those reported by Vivado, resulting in up to 50% reduction of corresponding LUT-specific essential bits. FFI experiments have confirmed that filtered-out bits indeed do not impact the behaviour of the DUTs, and can be safely left out of consideration. On the one hand, this might improve the accuracy of those reliability estimations, that rely on the amount of essential bits without estimating their actual criticality (conservatively assuming all of them as critical), like for instance the work in [64]. On the other hand, when the fault injection is used to determine the exact number of critical bits (like in the presented case study), the bit-accurate mapping allows to reduce the FFI experimental effort.

At the same time, the bit-accurate LUT mapping allows to estimate the activity of LUT-specific CM cells through the proposed simulation-based profiling. The case study has confirmed the hypothesis that increasing activity time of LUT bits increases their criticality, i.e. the probability that their upsets will lead to a failure. With respect to the fault injection experiments, this allows to filter-out the inactive fault targets, and thus to reduce even further the experimental effort, without the risk to overlook any critical bits. From the design viewpoint such activity profiles could be useful for the optimization of SEU mitigation mechanisms, such as CM scrubbing.

Fifth, it has been shown that the speed-up gain, attained by the proposed optimizations, greatly depends on the particular DUT, its representation level, as well as on the considered fault models.

The *bit-accurate LUT mapping* and *profiling of switching activity* in conjunction reduce the number of fault targets by 49% to 77%, and accelerate the dependability assessment of LUT-based combinational logic by 1.5 to 3.4 times. This speed-up is more pronounced in the case of SBFI experiments than in the case of FFI, since the profiling overheads in the former case are much less significant in comparison to the total experimental effort. However, when targeting not only the combinational logic, but a complete set of essential bits, the resulting reduction of fault targets becomes much lower (14% to 19%). Therefore, further research on bit-accurate mapping of FPGA resources is required in order to deploy similar optimizations for the rest (mostly routing-specific) essential bits.

The iterative statistical fault injection accelerated the dependability benchmarking of considered soft-core processors by 3 to 6 times with respect to the the common conservative statistical injection approach [100]. In practice, it is equally efficient at all descriptions levels, and provides the highest speed-up gain when the fault space is huge and the expected failure rate is low.

The multilevel fault injection reaches the highest speed-up gain (two orders of magnitude) when most implementation-level registers can be mapped onto the source RTL model. Whereas the *checkpointing* is most efficient in the case of implementation-level SBFI experiments, when the checkpoint recovery time is negligible in comparison to the runtime of each SBFI run.

It is important to note that most of the proposed optimizations complement each other, and can be also combined with other existing approaches. For instance, it was possible to accelerate the analysis of bit-flips in registers of MC8051 by 620 times, by combing the multilevel injection (150x speed-up), SBFI checkpointing (1.67x speed-up), and iterative sampling (2.5x speed-up). Some speed-up techniques that can be used in addition to those presented are *fault collapsing* and *dynamic fault analysis*. As it has been discussed in Section 6.4, the fault collapsing filters-out the ineffective faults, and may seem similar to the proposed profiling approach. Nevertheless, these two approaches operate at different representation levels and target different types of logic: the fault collapsing (as it is proposed in [24]) applies to RT-level memories, while the profiling applies to FPGA-specific combinational logic (LUTs) at the implementation and FPGA levels. Therefore they are not alternative, but rather complementary techniques (if generalizing the fault collapsing for implementation-level memories). The dynamic fault analysis (that aims at early identification of fault effects [129]), despite being very DUT/workload-specific, can be potentially used in conjunction with all the proposed optimization techniques.

9.3 Dependability-aware design space exploration for optimal tuning of EDA parameters

Vivado Design Suite is a proprietary framework that supports the last generation of Xilinx FPGAs and SoC devices. It provides a total of 30 synthesis [186] and implementation [185] parameters that may impact the performance, power consumption, area, and robustness of the resulting implementation. This section presents a case study which aims at determining the close to optimal configurations of Vivado parameters, that improve as much as possible the PPAD results of considered DUTs (MC8051, AVR, Microblaze) beyond the results of Vivado's default configuration.

9.3.1 Experimental procedure

All considered Vivado parameters have been labelled as factors X_{01} to X_{30}. Each factor has S different valid settings (numbered 0 to $S-1$). Considered factors along with their treatment levels are listed in Table 9.12. The tuning of these parameters is carried out using two different design space exploration (DSE) approaches, proposed in Chapter 7.

Table 9.12: Vivado parameters under study, default level highlighted in bold

	Factor	Parameter	Considered levels
Synthesis parameters	x_{01}	directive	**0: Default**, 1: RuntimeOptimized, 2: AreaOptimized_high, 3: AreaOptimized_medium, 4: AlternateRoutability, 5: AreaMapLargeShiftRegToBRAM, 6: AreaMultThresholdDSP, 7: FewerCarryChains
	x_{02}	flatten_hierarchy	0: None, 1: full, **2: rebuilt**
	x_{03}	gated_clock_conversion	**0: off**, 1: on, 2: auto
	x_{04}	bufg	0: 0, 1: 1, 2:5, **3: 12**, 4: 100
	x_{05}	fanout_limit	0: 10, 1: 100, 2: 1000, **3: 10000**
	x_{06}	retiming	**0: false**, 1: true
	x_{07}	fsm_extraction	**0: auto**, 1: one_hot, 2: sequential, 3: johnson, 4: gray, 5: off
	x_{08}	keep_equivalent_registers	**0: false**, 1: true, 2: off
	x_{09}	resource_sharing	**0: auto**, 1: on, 2: off
	x_{10}	control_set_opt_threshold	**0: auto**, 1: 0, 2: 1, 3: 5, 4: 10, 5: 16
	x_{11}	no_lc	**0: false**, 1: true
	x_{12}	no_srlextract	**0: false**, 1: true
	x_{13}	shreg_min_size	0: 0, 1: 3, 2: 8, **3: 16**, 4: 32, 5: 128
	x_{14}	max_bram	**0: -1**, 1: 0, 2: 1, 3: 8, 4: 32
	x_{15}	max_uram	**0: -1**, 1: 0, 2: 1, 3: 8, 4: 32
	x_{16}	max_dsp	**0: -1**, 1: 0, 2: 1, 3: 8, 4: 32
	x_{17}	max_bram_cascade_height	**0: -1**, 1: 0, 2: 1, 3: 8, 4: 32
	x_{18}	max_uram_cascade_height	**0: -1**, 1: 0, 2: 1, 3: 8, 4: 32
	x_{19}	cascade_dsp	**0: auto**, 1: tree, 2: force
	x_{20}	assert	**0: false**, 1: true
Implementation parameters	x_{21}	opt_design. is_enabled	0: false, **1: true**
	x_{22}	opt_design.directive	0: Explore, 1: ExploreArea, 2: ExploreSequentialArea, 3: AddRemap, 4: NoBramPowerOpt, 5: RuntimeOptimized, 6: ExploreWithRemap, **7: Default**
	x_{23}	power_opt_design.is_enabled	**0: false**, 1: true
	x_{24}	place_design.directive	0: Explore, 1: WLDrivenBlockPlacement, 2: LateBlockPlacement, 3: ExtraNetDelay_high, 4: ExtraNetDelay_medium, 5: ExtraNetDelay_low, 6: SpreadLogic_high, 7: SpreadLogic_medium, 8: SpreadLogic_low, 9: ExtraPostPlacementOpt, 10: SSI_ExtraTimingOpt, 11: SSI_SpreadSLLs, 12: SSI_BalanceSLLs, 13: SSI_BalanceSLRs, 14: SSI_HighUtilSLRs, 15: RuntimeOptimized, 16: Quick, **17: Default**
	x_{25}	post_place_power_opt_design.is_enabled	**0: false**, 1: true
	x_{26}	phys_opt_design.is_enabled	**0: false**, 1: true
	x_{27}	phys_opt_design.directive	**0: Default**, 1: Explore , 2: ExploreWithHoldFix, 3: AggressiveExplore, 4: AlternateReplication, 5: AggressiveFanoutOpt, 6: AddRetime, 7: AlternateFlowWithRetiming, 8: RuntimeOptimized
	x_{28}	route_design.directive	0: Explore , 1: NoTimingRelaxation, 2: MoreGlobalIterations, 3: HigherDelayCost, 4: AdvancedSkewModeling, **5: Default**, 6: RuntimeOptimized, 7: Quick
	x_{29}	post_route_phys_opt_design.is_enabled	**0: false**, 1: true
	x_{30}	post_route_phys_opt_design.directive	0: Explore, 1: AggressiveExplore, 2: AddRetime, **3: Default**

The first series of DSE experiments is based on genetic algorithm (GA) with iterative selection. The goal of single-objective GA is to improve as much as possible the dependability of considered DUTs, while reaching a clock frequency of 20 MHz (individuals with lower frequency are filtered-out). Similarly to benchmarking scenario, dependability will be quantified through the failure rate λ attribute, and

will take into account only the upsets of non-changeable memory λ_{CM}, since it has been previously shown that it has a dominating contribution (93% to 96%) into the total failure rate. The multi-objective NSGA aims at simultaneous improvement of both dependability and frequency, i.e. determines a set of Pareto-efficient solutions with respect to these two goals. The purpose of these experiments is to illustrate the feasibility of GA-based dependability-aware DSE, and to quantify the speed-up attainable through the iterative selection with respect to the common selection approach. The iterative selection algorithm 7.3 is configured as follows: the error margin threshold $EM_F = 0.1\%$, and the sample size increment $\delta = 10000$.

The second series of DSE experiments is based on Design of Experiments (DoE). A D-optimal DoE-based approach is selected for two reasons: (i) Vivado parameters are quantified at more then two levels, and (ii) there is no information available a-priory regarding the compatibility of considered parameters (regularity of design space). The purpose of DoE-based experiments is to (i) quantify the contribution of each Vivado parameter towards each PPAP attribute and infer a predictive models for each of them, (ii) determine the optimal Vivado configurations with respect to each individual PPAD attribute, and with respect to multi-objective goal quantified by mission-critical score (defined in Table 9.2), and (iii) cross-compare the optimization results obtained by the GA-based and DoE-based approaches. To ensure the accuracy of statistical analysis in the case of DoE-based DSE, the robustness of sampled individuals is estimated following the iterative statistical FFI with a strict error margin goal of 0.1%. The threshold that will serve for the acceptance of dependability regression models is set to 0.6 (model should explain at least 60% of variability of response variable). As it is explained in Section 7.2.3, the iterative D-optimal DSE process terminates, once all regressions models have been accepted.

9.3.2 DSE results obtained by GA-based approach

The single-objective genetic DSE process, detailed in Fig. B.2 (Annex), was deployed for two selection scenarios: (i) 9 individuals out of 18 – experiments a, b, c for MC8051, AVR, and Microblaze respectively, and (ii) 6 individuals out of 18 – experiments d, e, f. On the whole it took 4 to 6 GA iterations to reach the convergence in all cases. In experiments b, e, and f the robustness of the circuits gradually improved at each iteration but the last one, whereas in experiments a, c, and d the DSE came across the close-to-optimum solution at the first GA iterations.

The multi-objective NSGA-based DSE was carried out for the selection 9 individuals out of 18 (experiments g, h, i), the corresponding convergence results are detailed in Fig.B.3 (Annex). As it can be seen from the excerpt from the GA/NSGA results, depicted in Fig. 9.9, the frequency and failure rate are conflicting optimization goals. Even though at the first iteration the primary Pareto set $F1$ included just two configurations, by the 5-th (last) iteration the $F1$ set already comprised 8 Pareto-efficient solutions, for which it is impossible to improve the clock frequency without degrading the robustness (failure rate).

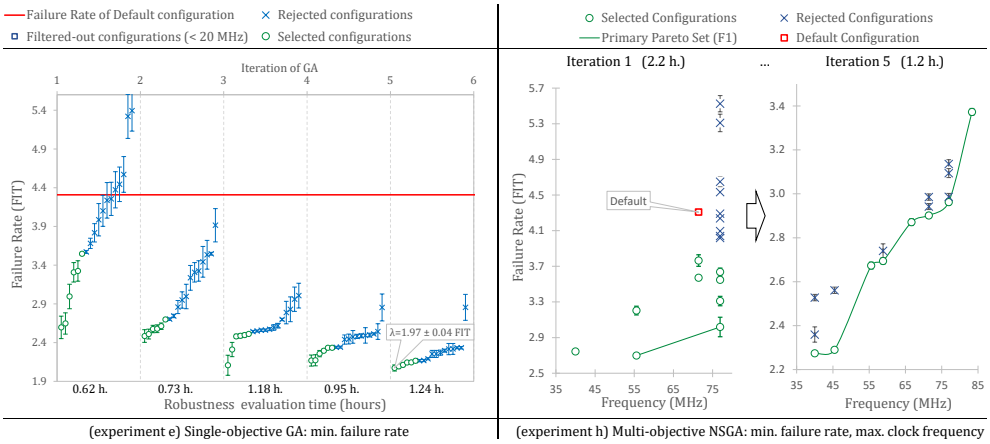


Figure 9.9: Excerpt from GA-based DSE results for AVR: single-objective GA (experiment e), and multi-objective NSGA (experiment h)

It is worth noting that in all DSE experiments (a–i), the resulting solutions significantly improved the robustness of the Vivado's default configuration. Table 9.17 lists the resulting best configurations in each experiment, along with the robustness improvement ω they provide. As it can be seen, the failure rate has been reduced by $25\% - 36\%$ for MC8051, by $46\% - 52\%$ for AVR, and by $16\% - 20\%$ for Microblaze. Whereas multi-objective DSE discovered the configurations that simultaneously improve the robustness and clock speed beyond the Vivado's default results. Particularly, experiments g and h attained 15%/31% and 22%/17% of simultaneous robustness/frequency improvement for MC8051 and AVR respectively.

Even though the GA-based DSE doesn't explain the contribution of individual factors towards the PPAD goals, it can be noted that configurations achieving better robustness are, in general, those minimising the area. For instance, inferring distributed LUTRAMs instead of BRAM significantly increases both the area used for control logic and the failure rate, so the BRAM utilization constraints

(factor X_{14}) should be relaxed. The same area minimization reasoning explains why synthesis parameter *flatten hierarchy*, is set to *full* ($X_{02} = 1$, allows complete cross-boundary optimizations) in all absolute best configurations (experiments d, e, c) in Table 9.17. It can also be seen that the best configurations share $X_{03} = 0$ (disables gated clock conversion), $X_{21} = 1$ (enables implementation-time design optimization), and $X_{29} = 0$ (disables post-route physical optimization). Finally, X_{05} set to either 1 or 2 in all best configurations may indicate that the optimal fanout is in the range [100 : 1000].

Table 9.13: Resulting configurations providing best robustness

Exp.		*Factors' settings*[1]																													ω^2
		X01	X02	X03	X04	X05	X06	X07	X08	X09	X10	X11	X12	X13	X14	X15	X16	X17	X18	X19	X20	X21	X22	X23	X24	X25	X26	X27	X28	X29	X30	
MC8051	(a)	0	**1**	**2**	1	1	**1**	**5**	1	0	**3**	**1**	**0**	2	**4**	**1**	0	5	**3**	2	**1**	**1**	**2**	0	6	**0**	**0**	**7**	**7**	**1**	**1**	25%
	(d)	**2**	**1**	**0**	3	**1**	**1**	**1**	**0**	**0**	**2**	**0**	1	**1**	**4**	0	**2**	4	**5**	2	**1**	**1**	**1**	0	15	**1**	1	6	2	**0**	3	36%
	(g)	2	**1**	**1**	3	**2**	**0**	**0**	1	**2**	**3**	**1**	**1**	4	**4**	**4**	3	**3**	5	**2**	**0**	**0**	2	1	**10**	**0**	0	**7**	2	**0**	**0**	26%
AVR	(b)	**1**	**1**	**2**	**3**	**1**	**1**	4	**1**	**1**	**2**	**0**	**1**	**0**	**3**	**4**	3	**2**	3	**1**	**1**	**1**	7	0	**4**	**1**	**1**	1	**7**	1	1	46%
	(e)	**6**	**1**	**0**	4	2	**0**	**4**	**1**	**1**	**4**	**1**	**1**	3	**0**	1	2	4	2	**1**	1	1	**3**	**1**	**3**	**0**	0	**7**	**7**	**0**	**2**	52%
	(h)	**2**	**1**	**0**	**0**	**1**	0	5	0	**1**	**0**	**1**	**1**	**5**	0	0	0	4	**1**	**2**	**1**	0	2	0	7	1	0	**7**	7	0	3	47%
Micro-blaze	**(c)**	**1**	**1**	**0**	**3**	**2**	**0**	2	**0**	**2**	2	0	0	**3**	**1**	**3**	2	1	**2**	**0**	**0**	**1**	**5**	**1**	13	**1**	0	**2**	**7**	**0**	**2**	20%
	(f)	**1**	**1**	**1**	4	1	**0**	**1**	**1**	**0**	**4**	**0**	**0**	**5**	**1**	**4**	2	**5**	**5**	**1**	**0**	**1**	**2**	**0**	**16**	**1**	**0**	**0**	**3**	**0**	**0**	17%
	(i)	7	**0**	**0**	**4**	**1**	**1**	0	**0**	**0**	1	**0**	**0**	**1**	0	**2**	2	**4**	**0**	**2**	1	**1**	4	**0**	**16**	1	0	7	6	0	0	16%
Default		0	2	0	3	3	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	7	0	17	0	0	0	5	0	3	

[1] setting coinciding with 75% of best selected individuals is highlighted in bold
[2] ω - reduction of failure rate with respect to Vivado default setting

The total robustness evaluation time was roughly two times less for the single-goal GA (3.5 hours (experiment e) to 9.7 hours (experiment f)) than in multi-objective NSGA (7.5 hours (experiment h) to 16.6 hours (experiment i)). This can be explained by differences in the selection process: single-goal GA only keeps running the fault injection process until it becomes possible to select 9(6) out of 18 individuals, while NSGA requires non-overlapping confidence intervals for all 18 individuals in population. It can be also seen from Fig. B.2 that the experimental time is unequally distributed among iterations. As the convergence process goes on, the selection process can be faster or slower depending on how much diversity is present among configurations.

Table 9.14 shows the accumulated number of experiments at each GA iteration for different fault injection strategies. When no optimisations are considered, all 32 Mbit of the Zynq-7000 bitstream are targeted for each configuration, for a total of more than 1000 millions of experiments. The speed-up provided by the proposed iterative selection K approach ranges between 145.4 and 402.5 (experiments (i) and (e) respectively).

Focusing on just how much improvement it provides beyond the state of the art approaches, the final speed-up ranges between 1.8 and 3.3 in single-objective DSE,

Table 9.14: Accumulated millions of fault injection experiments and speed-up (in parenthesis) attained by the proposed iterative selection strategy

	Experiment	Iteration [#Confs]	Strategies			
			Without optimisations	Optimized essential bits	Statistical fault injection (e=0.1%)	Selection of K best
Single objective optimization	(a) MC8051 Sel. 9/18	1 [18]	582 (388.0)	15.5 (10.6)	8.7 **(5.9)**	1.5
		2 [27]	873 (349.2)	18.9 (7.7)	11.4 **(4.6)**	2.5
		3 [36]	1164 (363.8)	22.6 (7.0)	13.9 **(4.3)**	3.2
		4 [45]	1454 (354.6)	26.1 (6.4)	16.8 **(4.1)**	4.1
		5 [54]	1745 (295.8)	29.1 (5.0)	19.3 **(3.3)**	5.9
	(b) AVR Sel. 9/18	1 [18]	582 (970.0)	5.2 (9.4)	4.5 **(8.1)**	0.6
		2 [27]	873 (545.6)	7.2 (4.6)	6.5 **(4.2)**	1.6
		3 [36]	1164 (465.6)	9.2 (3.7)	8.4 **(3.4)**	2.5
		4 [45]	1454 (363.5)	11.2 (2.8)	10.1 **(2.5)**	4.0
		5 [54]	1745 (371.3)	12.8 (2.7)	11.8 **(2.5)**	4.7
		6 [63]	2036 (303.9)	14.7 (2.2)	13.5 **(2.0)**	6.7
	(c) Microblaze Sel. 9/18	1 [18]	582 (388.0)	5.1 (3.3)	4.3 **(2.8)**	1.5
		2 [27]	873 (379.6)	7.4 (3.2)	6.3 **(2.7)**	2.3
		3 [36]	1164 (388.0)	9.0 (3.0)	7.9 **(2.7)**	3.0
		4 [45]	1454 (259.6)	11.3 (2.0)	9.9 **(1.8)**	5.6
	(d) MC8051 Sel.6/18	1 [18]	582 (415.7)	18.3 (13.4)	10.1 **(7.5)**	1.4
		2 [30]	970 (461.9)	23.8 (11.2)	14.1 **(6.6)**	2.1
		3 [42]	1357 (315.6)	27.4 (6.4)	17.2 **(4.0)**	4.3
		4 [54]	1745 (286.1)	30.7 (5.1)	20.1 **(3.3)**	6.1
	(e) AVR Sel. 6/18	1 [18]	582 (831.4)	5.0 (7.3)	4.4 **(6.4)**	0.7
		2 [30]	970 (646.7)	7.4 (5.0)	6.8 **(4.6)**	1.5
		3 [42]	1357 (484.6)	9.8 (3.5)	9.0 **(3.2)**	2.8
		4 [54]	1745 (447.4)	12.3 (3.2)	11.3 **(2.9)**	3.9
		5 [66]	2133 (402.5)	14.7 (2.8)	13.5 **(2.5)**	5.3
	(f) MICROBLAZE Sel. 6/18	1 [18]	582 (529.1)	5.7 (5.1)	3.9 **(3.5)**	1.1
		2 [30]	970 (388.0)	9.4 (3.7)	7.4 **(2.9)**	2.5
		3 [42]	1357 (347.9)	13.1 (3.4)	10.6 **(2.7)**	3.9
		4 [54]	1745 (323.1)	16.7 (3.1)	13.6 **(2.5)**	5.4
		5 [66]	2133 (333.3)	20.0 (3.1)	16.6 **(2.6)**	6.4
		6 [78]	2521 (262.6)	23.4 (2.4)	19.5 **(2.0)**	9.6
Multi-objective (NSGA)	(g) MC8051	1 [18]	582 (187.7)	19.2 (6.2)	10.5 **(3.4)**	3.1
		2 [27]	873 (145.5)	22.9 (3.8)	13.5 **(2.3)**	6.0
		3 [36]	1164 (149.2)	27.6 (3.5)	16.9 **(2.2)**	7.8
		4 [45]	1454 (151.5)	32.2 (3.4)	20.1 **(2.1)**	9.6
		5 [54]	1745 (147.9)	35.7 (3.0)	23.0 **(1.9)**	11.8
	(h) AVR	1 [18]	582 (223.8)	5.0 (1.9)	4.4 **(1.7)**	2.6
		2 [27]	873 (203.0)	7.0 (1.6)	6.2 **(1.4)**	4.3
		3 [36]	1164 (207.9)	8.9 (1.6)	7.9 **(1.4)**	5.6
		4 [45]	1454 (196.5)	10.8 (1.5)	9.6 **(1.3)**	7.4
		5 [54]	1745 (200.6)	12.4 (1.4)	11.1 **(1.3)**	8.7
	(i) MICROBLAZE	1 [18]	582 (176.4)	5.1 (1.5)	4.2 **(1.3)**	3.3
		2 [27]	873 (164.7)	8.2 (1.5)	6.8 **(1.3)**	5.3
		3 [36]	1164 (157.3)	11.5 (1.6)	9.3 **(1.3)**	7.4
		4 [45]	1454 (153.1)	13.9 (1.5)	11.7 **(1.2)**	9.5
		5 [54]	1745 (145.4)	16.7 (1.4)	14.1 **(1.2)**	12.0

and between 1.2 and 1.9 in multi-objective NSGA. This speed-up depends on each particular scenario, but it gets the best results when there is more diversity present in the considered configurations, which in considered case study was observed at initial GA iterations – it can be seen from Table 9.14 that the speed-up at the first iteration ranges between 2.8 and 8.1 in single-goal GA, and between 1.3 and 3.4 in case of NSGA.

9.3.3 DSE results obtained by DoE-based approach

The DoE-based design space exploration started with the smallest allowable D-optimal design, comprising 112 configurations, which equals the sum of degrees of freedom of all parameters plus one. Within the initial design there were several invalid configurations (7 in case of MC8051, 12 in case of Microblaze), i.e. configurations that were either non-implementable, or not meeting the timing constraints under any clock frequency. Following the proposed methodology, these configurations have been discarded, and the new configuration have been appended to repair the design back to 112 configurations.

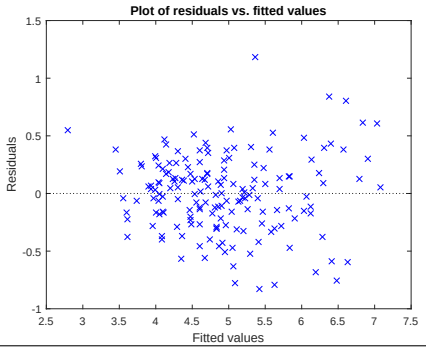
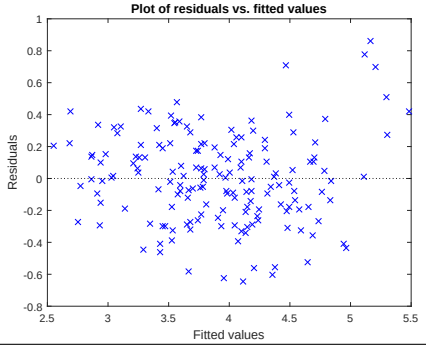
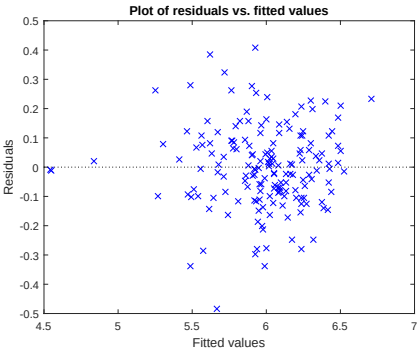
The regression analysis carried out on the basis of these 112 configurations revealed, that such initial design was too small to provide representative PPAD estimators, since the determination coefficients of most PPAD (for all DUTs) were near 0. Accordingly, the design was augmented by 50% of its initial size. Some of new configurations in the augmented design were also invalid, so that it has been repaired again. For the resulting D-optimal design of 168 configurations the determination coefficients R^2 of all regression models for all DUTs exceeded an acceptance threshold (0.60). In particular, the determination coefficient ranges between 0.83 and 0.85 for the failure rate, between 0.78 and 0.91 for the clock frequency, between 0.60 and 0.78 for the power consumption, and between 0.80 and 0.97 for the area. The iterative refinement of D-optimal design has been thus terminated.

The magnitude of this result can be understood if considering that for a total number of 5.35×10^{17} configurations, just 168 were enough to explain 60% to 97% of the effect of Vivado parameters on resulting PPAD attributes.

Table 9.15 lists the resulting regression models for the failure rate, alongside with the corresponding residual plots (difference between the observed and predicted value for the sample). As it can be seen, regression models explain 83% to 85% of variability of the failure rate by the effects of Vivado parameters. At the same time, the residuals are randomly and evenly distributed, there is no observable pattern, and the magnitude of the residuals is relatively small in comparison to the fitted value. Therefore it can be concluded that the obtained models have high enough quality for predicting the failure rate with respect to the arbitrary settings of Vivado parameters.

For each significant multilevel factor the model may include as many terms as the number of levels adopted by this factor. The response is thus computed by including only one term per each factor (which corresponds to the given setting X). For instance, let us consider the regression model for MC8051. When all

Table 9.15: Resulting regression models for failure rate (accounting for significant terms)

Residual plot	Regression Model
MC8051 	$\lambda(X) = 5.27 - 0.35(X_{01}=1) - 0.38(X_{01}=2) - 0.45(X_{01}=6) + 0.45(X_{02}=2) - 1.16(X_{05}=1) - 1.26(X_{05}=2) - 1.36(X_{05}=3) + 0.13(X_{08}=1) + 0.79(X_{14}=1) + 0.78(X_{14}=2) + 0.69(X_{14}=3) - 0.68(X_{28}=7) + 0.40(X_{29}=1);$ $R^2 = 0.85$
AVR 	$\lambda(X) = 4.84 - 0.43(X_{01}=1) - 0.76(X_{01}=2) - 0.30(X_{01}=3) - 0.85(X_{01}=6) - 0.32(X_{02}=1) + 0.31(X_{02}=2) - 0.71(X_{05}=1) - 0.79(X_{05}=2) - 0.78(X_{05}=3) - 0.11(X_{11}=1) + 0.19(X_{27}=1) + 0.25(X_{27}=3) - 0.20(X_{28}=6) - 0.75(X_{28}=7) + 0.18(X_{29}=1);$ $R^2 = 0.83$
Microblaze 	$\lambda(X) = 5.84 + 0.12(X_{01}=3) + 0.26(X_{02}=1) + 0.32(X_{02}=2) - 0.10(X_{03}=1) - 0.07(X_{03}=2) - 0.09(X_{05}=1) - 0.14(X_{05}=2) - 0.15(X_{05}=3) + 0.16(X_{08}=1) - 0.12(X_{11}=1) + 0.18(X_{13}=2) + 0.19(X_{13}=3) + 0.13(X_{13}=4) + 0.15(X_{13}=5) - 0.18(X_{17}=2) - 0.10(X_{17}=4) - 0.16(X_{21}=1) + 0.16(X_{23}=1) - 0.38(X_{24}=16) + 0.07(X_{26}=1) - 0.12(X_{28}=3) - 1.42(X_{28}=7) + 0.07(X_{29}=1);$ $R^2 = 0.85$

factors X are set at level 0, the failure rate λ equals the intercept value (5.27 FIT).

When changing the X_{01} (synthesis directive) to either 1 (*RuntimeOptimized*), or 2 (*AreaOptimizedHigh*), or 6 (*AreaMultThresholdDSP*), this value is reduced by 0.35 FIT, by 0.38 FIT, or by 0.45 FIT respectively. Setting the X_{05} to the level 1, or 2, or 3 (relaxes the fanout limit to 100, 1000, or 1000 respectively) additionally reduces the failure rate by 1.16 FIT, 1.26 FIT, and 1.36 FIT respectively (being thus $X_{05} = 3$ the most beneficial setting). Likewise the failure rate is reduced by 0.68 FIT by setting $X_{28} = 7$. On the opposite, the failure rate is increased by 0.13 FIT when setting $X_{05} = 1$, increased by 0.79 FIT when setting the $X_{14} = 1$, and increased by 0.40 FIT when setting $X_{29} = 1$.

The used stepwise regression included all linear terms for each significant factor. Nevertheless, not all of these terms significantly contribute to the response variable. Accordingly, Table 9.15 lists only those terms that reach the significance threshold ($p \leq 0.05$). Resulting regression models for the rest of PPAD attributes (including all terms exported by Matlab) are provided in the Annex B.3.

It is worth commenting that the regression models for the area (utilization) are inferred under the Poisson distribution, since it is a discrete attribute. The models for the area also take into account the interaction terms (for the significant factors), which notably improve the explanatory potential (determination coefficient) of these models. Regression models for the rest of response variables are inferred either under the normal distribution (failure rate, SDC percentage, clock frequency), or under the gamma distribution (power consumption), and all of them include only main (purely linear) terms.

It must be noted that Vivado's default configuration doesn't coincide with the model intercept. Accordingly, DAVOS relies on the inferred regression models to estimate how much the PPAD results are improved or degraded with respect to the Vivado's default configuration by changing the factors from their default setting to another. These results are summarized in Table 9.16. As it can be seen, there are three factors whose tuning improves the default failure rate by more than 5%, namely:

- $X_{01} = 6$ enables the lower threshold for dedicated DSP block inference – decreases the failure rate by 10% and 19% for MC8051 and AVR respectively;
- $X_{02} = 1$ enables the full hierarchy flattening, allowing inter-module logic optimizations – decreases the failure rate by 13.6% and 14.5% for M8051 and AVR respectively;
- $X_{28} = 7$ sets the route directive to *Quick* – decreases the failure rate of MC8051, AVR, and Microblaze by 12.8%, 19.2%, and 23.8% respectively;

At the same time, there are three factors whose improper configuration degrades the failure rate by more than 5%, namely:

- $X_{05} = 0$ limits the fanout by a factor of 10 – increases the failure rate by 32.1% and 17.8% for M8051 and AVR respectively;
- X_{14} set to 1, 2, or 3 limits the number of inferred BRAMs by 0, 1, or 8 respectively – increases the failure rate of MC8051 by 18.7%, 18.5%, and 16.5% respectively; the rest of DUTs are not affected since their BRAMs are instantiated as macrocells (not inferred);
- $X_{29} = 1$ enables the post-route physical optimization – increases the failure rate of MC8051 by 9.4%.

When considering the rest of PPAD attributes, one can note the opposite effects of some factors. For instance, factor $X_{28} = 7$ (which improves the failure rate) strongly reduces (worsens) the frequency of all DUTs by 31% to 61%, and at the same time reduces (improves) the power consumption by a similar magnitude 31% to 78%. Factor X_{29} set to 1 improves the clock frequency of MC8051 and AVR by 17% and 8% respectively, but on the opposite - worsens the power consumption by 15% and 21%. Likewise, it can be seen that limiting the amount of inferred BRAMs (X_{14} set to 1/2/3) strongly increases the utilization of LUTs and FFs of MC8051 (307% to 379%, and by 15% to 23% respectively), increases the failure rate (16% to 18%), but reduces the power consumption (by 14% to 16%) and the SDC percentage (60% to 71%). Factor $X_{05} = 0$ is the only one that strongly worsens several PPAD attributes (failure rate, power, area), without improving any other PPAD.

On the final DSE step the inferred regression models are used by DAVOS to identify the optimal configuration of factors with respect to each design goal. Table 9.17 lists the resulting best configurations for each single PPAD goal, as well as for multi-objective goals (WSM scores). It also provides the predicted and experimentally obtained improvement, achieved by each best configuration with respect to the Vivado's default configuration. Each best configuration is labelled by letter and number. The corresponding PPAD results and WSM scores (experimental and predicted) for each DUT are listed in Table B.4 (Annex).

First of all, one can note the difference between the predicted and experimentally obtained results. More precisely, the predictions tend to be more optimistic (show higher improvement) than the actual result. This difference is not surprising, since the predictions are computed by models that account for just 60% to 97% of the variability of the considered features. On the one hand, the accuracy of the models can be improved by considering the higher-order effects (interactions between two and more factors). On the other hand, under high number of significant factors,

Table 9.16: Percentage by which the default PPAD results are improved (highlighted in green) or degraded (highlighted in red), when changing the setting of each factor from its default level (listed only the effects with more than 5% impact)

		Failure rate (FIT)			SDC (percentage points)			Frequency (MHz)			Power (W)			Utilization Flip-Flops			Utilization LUTs		
		MC8051	AVR	Mic-z	MC8051	AVR	Mic-z	MC8051	AVR	Mic-z	MC8051	AVR	Mic-z	MC8051	AVR	Mic-z	MC8051	AVR	Mic-z
X01	1	-8.4	-9.7	0.1	44.3	-3.6	–	-9.7	-1.8	–	-0.3	-12.6	5.7	-6.2	-2.0	0	-4.99	-1.2	–
	2	-9.0	-17.4	1.4	2.2	-2.4	–	-13.1	-3.0	–	-3.1	-20 .0	-4.2	2.7	-5.2	0	-14.3	-16.6	–
	3	3.9	-6.9	2.2	8.9	8.5	–	-14.2	-3.6	–	-7.4	-19.6	-10.7	-0.3	-7.4	0	-8.5	-13.4	–
	4	5.3	-2.5	0.0	1.2	-0.7	–	16.4	3.5	–	12.0	-3.3	4.2	-4.1	-0.7	0	2.2	-4.6	–
	5	1.0	1.0	-1.3	-0.1	-0.5	–	3.1	-0.4	–	6.0	-6.4	-1.4	8.3	1.7	0	8.4	0.2	–
	6	-10.8	-19.4	-0.5	-2.3	-10.9	–	0.3	0.7	–	-2.1	-7.8	0	-2.0	-3.9	0	3.2	-13.4	–
	7	0.7	-2.9	0.4	0	0.3	–	13.5	0.5	–	5.2	-6.7	0.1	-3.5	2.8	0	2.0	6.5	–
X02	0	-10.6	-7.1	-5.7	-2.4	0.9	-3.9	–	–	-5.7	-13.8	-5.4	-10.9	0.9	-0.9	16.0	–	-8.8	–
	1	-13.6	-14.5	-1.1	4.0	7.2	-0.8	–	–	-0.5	-1.4	4.7	-0.5	0.8	0.4	0.3	–	-4.9	–
X04	1	–	–	–	–	–	–	0.7	2.4	–	–	5.6	–	–	–	–	–	–	–
X05	0	32.1	17.8	2.7	–	3.7	–	–	–	–	8.2	22.8	–	29.3	35.4	2.6	25.7	24.5	–
X07	1	–	–	–	5.1	–	–	–	–	–	–	–	-9.2	–	–	4.8	–	–	–
	3	–	–	–	7.5	–	–	–	–	–	–	–	0.3	–	–	2.4	–	–	–
	5	–	–	–	8.0	–	–	–	–	–	–	–	1.5	–	–	1.7	–	–	–
X08	1	3.0		2.8	0	–	-2.8	–	–	–	–	–	–	–	2.4	25.5	–	–	9.4
X09	1	–	–	–	-4.4	–	0.8	-3.3	-2.7	–	–	-5.4	–	–	–	–	–	–	–
X10	1	–	–	–	–	-0.6	0.6	–	–	–	-5.6	-1.0	26.9	–	–	–	–	1.1	-2.9
	2	–	–	–	–	1.0	0.2	–	–	–	-3.9	3.6	24.6	–	–	–	–	0.3	-1.1
	4	–	–	–	–	-2.4	-0.4	–	–	–	-9.2	4.7	-20.5	–	–	–	–	13.1	-2.4
	5	–	–	–	–	-1.7	-2.0	–	–	–	-9.4	5.3	-15.8	–	–	–	–	6.2	4.7
X11	1	-2.5	-2.5	-2.1	–	-3.7	-1.0	3.6	–	–	6.0	–	–	–	–	-2.2	–	12.2	5.7
X13	2	–	–	4.0	–	–	–	–	–	–	–	–	–	–	–	18.8	–	–	–
	3	–	–	4.2	–	–	–	–	–	–	–	–	–	–	–	16.9	–	–	–
	4	–	–	3.2	–	–	–	–	–	–	–	–	–	–	–	21.6	–	–	–
	5	–	–	3.5	–	–	–	–	–	–	–	–	–	–	–	17.8	–	–	–
X14	1	18.7	–	–	-71.1	–	–	–	–	–	-15.7	–	–	23.4	–	–	379.3	–	–
	2	18.5	–	–	-66.3	–	–	–	–	–	-14.2	–	–	20.4	–	–	383.2	–	–
	3	16.5	–	–	-60.7	–	–	–	–	–	-16.8	–	–	15.7	–	–	307.8	–	–
X17	4	–	–	-1.8	–	–	–	0	-0.3	–	–	–	–	-5.7	–	–	–	–	–
X21	0	–	1.9	2.9	–	-1.6	-10.5	0	-1.5	–	–	–	30.3	–	–	56.9	–	7.2	20.5
X22	0	–	–	–	9.0	–	–	–	–	–	–	–	–	–	–	–	–	–	–
	6	–	–	–	-5.2	–	–	–	–	–	–	–	–	–	–	–	–	–	–
X23	1	–	–	2.7	–	-3.7	-0.8	-1.8	–	–	-10.6	0	-4.7	–	–	-2.4	–	–	4.5
X24	7	–	–	-0.2	–	-0.7	-0.9	6.7	-1.3	–	–	–	–	–	–	–	–	–	–
	16	–	–	-5.7	–	-7.1	-4.9	-4.3	-6.0	–	–	–	–	–	–	–	–	–	–
X25	1	–	–	–	–	–	–	–	–	1.3	-7.3	–	–	–	–	–	–	–	–
X26	1	–	–	1.2	-5.7		1.0	–	–	–	–	–	–	–	–	–	–	–	–
X27	1	–	4.4	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
	3	–	5.7	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
X28	0	3.4	-2.2	1.3	–	0.6	0.2	6.6	3.2	0.6	-3.3	4.1	0.3	–	–	–	–	–	–
	1	8.3	0.3	2.4	–	-0.2	0.8	5.0	3.9	0.7	-7.1	4.8	-0.2	–	–	–	–	–	–
	2	-0.8	-5.7	1.6	–	-0.4	1.1	0.2	-0.7	-2.1	-11.0	-0.4	1.6	–	–	–	–	–	–
	3	0.7	-4.3	-0.9	–	-0.1	0.4	2.8	0.1	-0.2	-2.2	1.4	6.3	–	–	–	–	–	–
	4	5.6	-1.7	0.7	–	-1.1	0.5	2.2	0.7	0.3	-12.2	3.8	2.2	–	–	–	–	–	–
	6	0.2	-6.9	1.2	–	-2.7	0.5	-1.3	-3.7	-1.3	-3.0	-1.3	2.8	–	–	–	–	–	–
	7	-12.8	-19.2	-23.8	–	-8.2	-7.1	-33.8	-31.4	-61.7	-33.5	-31.6	-78.1	–	–	–	–	–	–
X29	1	9.4	4.0	1.2	–	1.6	0	17.6	8.0	2.7	15.9	21.9	–	–	–	–	–	–	2.0

this would lead to exponential growth of the sample size (increased by the number of degrees of freedom for each interaction term). Such increase of experimental cost under most optimization scenarios will be not reasonable, since (as previously explained) the higher order effects are considered less significant than the main effects. Accordingly, even though the regression models are not perfect, they are

Table 9.17: Resulting best configurations for each optimization goal

Optimization goal	Benchmark circuit	Setting of factors[1,2]																																Conf. Label	Improvement (ω)[3]	
		x_{01}	x_{02}	x_{03}	x_{04}	x_{05}	x_{06}	x_{07}	x_{08}	x_{09}	x_{10}	x_{11}	x_{12}	x_{13}	x_{14}	x_{15}	x_{16}	x_{17}	x_{18}	x_{19}	x_{20}	x_{21}	x_{22}	x_{23}	x_{24}	x_{25}	x_{26}	x_{27}	x_{28}	x_{29}	x_{30}		Predicted	Actual
Min. Failure Rate λ (FIT)	MC8051	6	1	-	-	3	-	-	0	-	-	1	-	-	4(0)	-	-	-	-	-	0	-	-	-	-	-	-	7	0	-	A1 (A2)	-41 %	-34 %
	AVR	6	1	1	-	2	-	-	-	-	-	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0	7	0	-	B1	-59 %	-50 %
	Microblaze	5	0	1	-	3	-	-	0	-	-	1	-	1	-	-	-	2	-	-	-	1	-	0	16	-	0	-	7	0(1)	-	C1 (C2)	-16 %	-20 %
Min. SDC (%)	MC8051	6	0	0	-	-	-	0	-	1	-	-	-	-	1	-	-	-	-	-	1	-	6	-	-	-	1	-	5	-	-	A3	-95 %	-70 %
	AVR	6	0	0	-	2	-	-	-	-	4	1	-	-	-	0	-	-	-	-	-	-	-	0	16	-	-	-	7	0	-	B2	-34 %	-21 %
	Microblaze	-	0	-	-	-	-	-	0	0	4	1	-	-	-	-	-	-	-	-	-	0	-	0	16(10)	-	0	-	7	-	-	C3 (C4)	-24 %	-23 %
Max. Frequency (MHz)	MC8051	4	-	-	1	-	-	-	-	2	-	1	0	-	-	-	3	-	-	-	-	-	-	0	7	-	-	-	0	1	-	A4	+56 %	+38 %
	AVR	4	-	-	1	-	1	-	-	2	-	-	-	-	-	-	-	1	-	-	0	1	-	-	12	-	-	-	1	1	-	B3	+18 %	+8 %
	Microblaze	-	2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1	-	-	1	1	-	C5	+5 %	0 %
Min. Power (W)	MC8051	3	0	-	-	2	-	-	-	-	5	0	-	-	3	-	-	-	-	-	-	-	-	1	-	1	-	-	7	0	-	A5	-99 %	-47 %
	AVR	2	0	-	0	2	-	-	-	1	0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	7	0	-	B4	-42 %	-55 %
	Microblaze	3	0	-	-	-	-	0	-	-	4	-	-	-	-	-	-	-	-	-	-	0	-	1	-	-	-	0	7(1)	-	-	C6 (C7)	-32 %	-18 %
Min. UTIL FF	MC8051	1	2	-	-	-	-	-	0	-	-	-	-	-	0	-	-	4	-	-	-	-	-	-	-	-	-	-	-	-	-	A6	-22 %	0 %
	AVR	3	2	-	-	3	0	-	0	-	-	-	-	-	-	-	-	-	-	-	0	-	-	-	-	-	0	-	-	-	-	B5	-5 %	0 %
	Microblaze	-	2	-	-	2	-	0	0	-	-	1	-	0	-	-	-	-	-	-	0	1	-	1	-	-	-	-	-	-	-	C8	-7 %	0 %
Min. UTIL LUT	MC8051	2	-	-	-	3	-	-	-	-	-	-	-	-	0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	A7	-14 %	-13 %
	AVR	2	0	-	-	1	-	-	-	0	4	0	0	-	-	-	-	-	-	-	-	1	-	0	-	-	-	-	-	-	-	B6	-21 %	-20 %
	Microblaze	-	-	-	-	-	-	-	0	-	1	0	1	-	-	-	-	-	-	-	-	1	-	0	-	-	-	-	-	0	-	C9	-5 %	0 %
Max. Score Mission-Critical	MC8051	6	1	-	-	3	-	4	-	-	-	-	-	-	4(0)	-	-	4	-	-	0	-	-	-	-	-	-	-	7	0	-	A8 (A9)	+37 %	+18 %
	AVR	6	1	0	-	2	-	-	-	-	0	1	-	-	0	-	-	-	-	-	-	-	-	-	-	-	-	0	7	0	-	B7	+47 %	+17 %
	Microblaze	-	0	1	-	2	-	1	0	-	4	0	-	0	-	-	-	-	-	-	-	1	-	0	16	-	-	-	7(3)	-	2	C10 (C11)	+13 %	+36 %
–	–	0	2	0	3	3	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	7	0	17	0	0	0	5	0	3	Default	–	–

[1] statistically insignificant factors denoted by dash (these factors set at their default level)
[2] alternative (second best) setting in parenthesis when absolute best is non-implementable
[3] ω – percentage improvement of optimization goal with respect to the default result

still useful (accurate enough) to determine those configurations which outperform the default one.

In such a way, the achieved (actual) reduction of failure rate (with respect to the default configuration) amounted 34%, 50% and 20% for MC8051, AVR and Microblaze respectively. These results are very close to those previously obtained by the GA-based DSE: 36%, 52%, and 20% respectively. Accordingly, even though GA-based approach implicitly optimizes both main effects and interactions (but not quantifies their contribution into the resulting PPAD), its advantage was rather marginal (at least under the small number of GA iterations).

A notable improvement has been observed for the rest of PPAD goals. The SDC percentage of MC8051, AVR, and Microblaze is reduced by 70%, 32%, and 23%. Such a high SDC improvement for MC8051 is primarily explained by the factor X_{14}, which limits the BRAM inference, and thus drastically increases the amount combinational logic (essential bits) used to control the distributed memory (inferred instead of BRAM). Despite the percentage of critical bits within the inferred logic is significantly reduced, the absolute number of critical bits becomes even higher, which in turn degrades the resulting failure rate. For that reason, minimization of SDC percentage is relevant for dependability improvement only when it is accompanied by simultaneous reduction of essential bits. The most beneficial adjustment of Vivado parameters that improves the failure rate of all

considered DUTs is to (i) set the synthesis strategy for DSP inference $X_{01} = 6$, (ii) allow the full flattening of design hierarchy $X_{02} = 1$, and (iii) set the route directive to *Qiuck* $X_{28} = 7$.

The frequency has been improved by 38% and 8% for MC8051 and AVR respectively, whereas for the Microblaze it was impossible to improve the frequency of default configuration. The most beneficial tuning of Vivado parameters for frequency improvement is to (i) set the synthesis strategy to *alternative routability* $X_{01} = 4$, (ii) set the route directive to *explore* $X_{28} = 0$ or to *no timing relaxation* $X_{28} = 1$, and (iii) enable the post-route physical optimization $X_{29} = 1$.

The power consumption has been reduced by 18% to 55%. Among all the factors impacting the power consumption, the most beneficial adjustment is to (i) set the synthesis strategy to *area optimized* $X_{01} = 3$, (ii) disable the hierarchy flattening $X_{02} = 0$, (iii) limit the BRAM inference $X_{14} = 1/2/3$, and (iv) set the route directive to *Quick* $X_{28} = 7$.

The utilization of Flips-Flops was impossible to improve for any of the DUTs, whereas the utilization of LUTs has been improved for MC8051 and AVR by 13% and 20% respectively. The only adjustment that notably contributed to this improvement was setting the synthesis strategy to *'area optimized high'* $X_{01} = 2$.

Finally, the multi-criteria optimization is illustrated by maximization of mission-critical WSM score, which balances the failure rate, performance and power consumption, giving them 60%, 20% and 20% of relative importance respectively. The resulting best configurations outscored the default configuration by 18%, 17% and 36% for MC8051, AVR and Microblaze respectively. For the resulting mission-critical configuration the failure rate and power consumption have been notably improved in all cases, while the frequency has been improved only in the case of AVR. The main adjustments that lead to this improvement were the same as in the case of single-goal (failure rate) optimization, accompanied by the DUT-specific tuning of minor effects, as it is shown in Table 9.17.

The time required to implement and evaluate these configurations (plus the resulting best ones) is listed in Table 9.18. The total time required for implementation of sampled configurations for MC8051, AVR and Microblaze amounted 59 hours, 66 hours and 33 hours respectively, while the time to evaluate their dependability was measured as 94 hours, 91, hours and 69 hours respectively. Since both implementation and dependability evaluation processes are running in parallel (by PPAD evaluation engine), the total DSE time is determined by the slowest of them (dependability evaluation).

Table 9.18: PPA and dependability evaluation time measured for each DSE experiment

DUT	Total evaluated configurations	Implementation (PPA evaluation) time [h]		FFI (dependability evaluation) time [h]	
		Total time	Per configuration (std. dev)	Total time	Per configuration (std. dev)
MC8051	255	58.9	1.39 (0.74)	94.5	0.74 (0.26)
AVR	261	66.1	1.52 (0.58)	91.0	0.70 (0.30)
Microblaze	187	33.3	1.07 (0.25)	69.2	0.74 (0.28)

Nevertheless, it must be noted that dependability evaluation of each configuration on the average was 1.4 to 2.1 times faster than their implementation (PPA evaluation). Only due to higher parallelism of implementation process (6 configurations in parallel on core-i7 machine) the total implementation time was lower than dependability evaluation. Accordingly, if equipping the PPAD evaluation engine with 4 more evaluation boards, the dependability evaluation would not be the DSE bottleneck anymore. But even with the available computing resources, it was possible to significantly improve the PPA and dependability of considered DUTs just by properly tuning the EDA parameters. This result shows the benefits of combining the efficient dependability-aware DSE strategy with FPGA-based fault injection, optimization of essential bits, and iterative statistical sampling.

9.3.4 Discussion

One may argue that estimating the trend of a population of thousands of trillions of configurations by sampling only 168 of them is clearly inadequate. On the one hand, it is the exact purpose of *Design of Experiments* (D-optimal design in this case) to representatively sample the design space by the smallest possible number of trials. Additional experiments have been carried out (in case of MC8051 and AVR) to increase the design size by another 50% (up to 224 configurations), but no real benefit has been observed on estimations or actual values reported by implemented configurations. On the other hand, (sub)optimal configurations determined by the proposed methodology always outperform the results provided by the default configuration of Vivado.

Likewise one may say that in case of GA-based DSE the performed number of iterations was too small to talk about optimality of derived configurations. On the one hand, indeed, these configurations may be not globally optimal. On the other hand they still outperform the default configuration in all cases. By forcing the genetic algorithm to carry out more iteration beyond the defined stop condition, it

has been checked that additional iterations provide rather marginal improvement (less than 2%) over the presented result. Additionally, the GA/NSGA algorithm itself can be calibrated (by tuning its mutation rate, crossover method, population size), thus preserving higher diversity under slower convergence. This might lead to better solutions, however, the GA calibration was out of the scope of presented work. Nevertheless, this case study has shown that the results obtained by the GA-based approach are in general very close to those obtained by the DoE-based approach, and in some cases even slightly better. This indicates that GA-based DSE, at least, has adjusted all main effects, that according to the principle of *hierarchical ordering* [172] are more significant than interactions (higher-order effects).

Regarding the inferred regression models, it can be noted they are surely far from perfection. Indeed, as it has been shown in Table B.4, there exist a discrepancy between the predicted and experimentally obtained PPAD results. This is explained by a multitude of different factors (not taken into account) that may impact the PPAD results, including the higher-order interaction, and even those not related to the EDA parameters under study. From the practical perspective, however, obtained models are good enough to explain the magnitude of PPAD increase/decrease when changing each significant parameter from one level to another, as well as to find the suboptimal configurations for each PPAD goal.

Finally, one may also argue that the attained dependability improvement, despite being significant, is not that impressive as to justify the high DSE experimental effort. In other words, wouldn't is be more beneficial for dependability improvement to simply instrument the design with fault tolerance mechanisms? First, it must be noted, that DSE not only allows to determine those configurations that improve the default result, but also to avoid those configurations that drastically degrade it. For instance, misconfiguration of Vivado suite may increase the failure rate of MC8051 by more than 70% with respect to the default configuration. Second, as it will be shown in the next section, the resilient versions of the same DUTs also gain a similar relative (and even higher absolute) reliability benefit from optimal tuning of EDA parameters. Furthermore, PPAD attributes can be balanced in such a way as to improve the dependability, while simultaneously compensating the overheads that tolerance mechanisms impose on the PPA attributes.

9.4 Dependability assessment and verification of fault-tolerant HW design

The mitigation of single and multiple-bit upsets is one of the major concerns in the design of safety-critical FPGA-based systems. As it has been shown in Section 9.2, SEUs in configuration memory (CM) are significantly more severe than SEUs in registers, BRAM, and LUTRAM, since they corrupt the very function of the circuit, and lead to much higher failure rates. Two common mechanisms for mitigation of CM upsets are periodical CM scrubbing, and full or partial FPGA reconfiguration on error detection. As it is pointed out in [121], this approach is sufficient only when application tolerates the long error latencies, otherwise additional hardening techniques must be employed. Particularly, the triple modular redundancy (TMR), in spite of its area and power penalties, is considered the most generic and efficient design hardening strategy [117]. Indeed, the combination of these two techniques is a quite common solution for FPGA reliability [25] [74].

This section will adapt this SEU mitigation approach (TMR combined with partial reconfiguration) to build a resilient version of AVR core (selected among the rest of DUTs for its overall best PPAD). On its basis this section will illustrate (i) how DAVOS FFI tool can be exploited for the dependability assessment of resilient HW designs, and (ii) how the tuning of EDA parameters can be exploited in conjunction with SEU mitigation mechanisms for obtaining even higher cumulative reliability gain.

9.4.1 Experimental procedure

The defined TMR assembly of AVR core under study is depicted in Fig. 9.10. It comprises three reconfigurable PL regions (P-blocks), and one static region. Each reconfigurable region allocates one replica of AVR IP core. Each replica has its dedicated data input, clock and reset lines, all driven by the PS part (Cortex-A9 core, which runs the fault injector application). The static region allocates three majority voters (one per replica). Each voter computes its data output following the common majority voting scheme $Res = D_1D_2 \vee D_1D_3 \vee D_2D_3$, which corrects (masks) any single error on any of its three inputs. The corrected result from each voter is supplied to the PS part through the GPIO interface (similarly to the simplex version). Additionally, each voter compares the result from its associated replica with the voted (correct) result Res, and raises an error flag in case of mismatch. This error flag is monitored by an external controller (PS/ARM application in this case study): when the error flag is raised for one of the replicas, the PS part reconfigures its respective PL partition by loading the partial bitstream (stored in the DDR memory) through the PCAP. In such a way,

the TMR with three voters should ensure that the design has no *single point of failure* (any single upset must be masked). Whereas the reconfiguration on error detection is supposed to prevent the accumulation of upsets in reconfigurable regions.

It is worth noting that SEUs still may be accumulated in the static region (which allocates the voting logic and the PS-PL interconnections). It should be thus protected by periodical CM scrubbing, as well as by the complete reconfiguration in case of failure. Nevertheless, since the number of essential bits in the static region is much smaller than in reconfigurable partitions, the described replica recovery mechanism is expected to reduce the accumulation of SEUs with respect to the unprotected DUT. It is also worth noting that reconfiguration of failing replicas is performed when the DUT passes into the idle mode – after completing the workload execution (the workload is executed periodically 50 to 150 times per second). After the reconfiguration, all TMR replicas are reset to the initial state, thus no additional measures were required in this case study to keep TMR replicas synchronized.

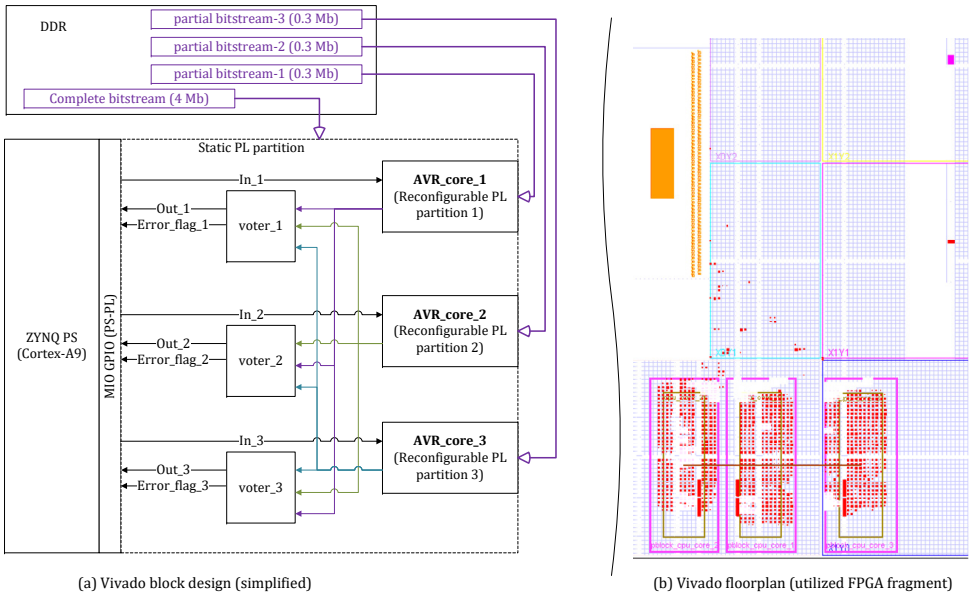


Figure 9.10: AVR assembly under study with integrated SEU mitigation mechanism

The evaluation of defined SEU mitigation mechanism comprises three FFI tests. First, it should be verified that resulting TMR implementation tolerates all single upsets, i.e. has no single point of failure. To this end, the FFI flow is config-

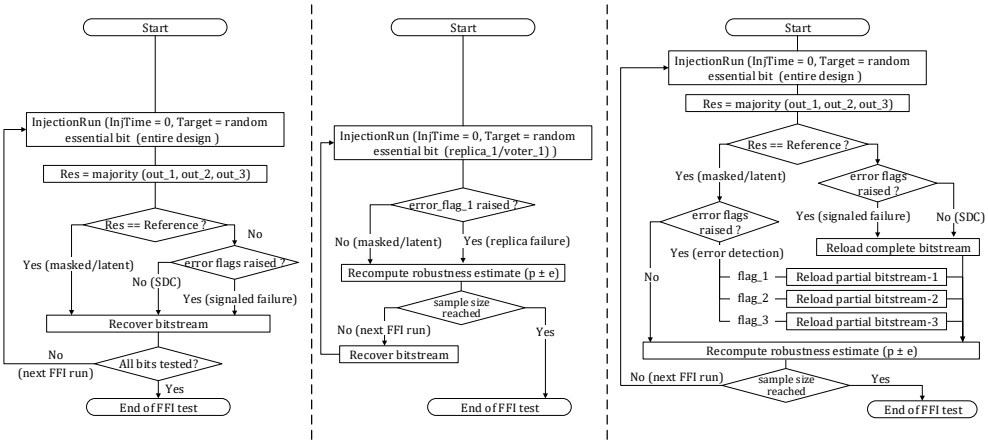


Figure 9.11: Adaptation of DAVOS FFI flow to the evaluation of defined resilient design

ured according to Fig. 9.11-a. Each FFI run flips the value of one essential bit, randomly selected within the global scope (encompassing both static and reconfigurable partition). The DUT processing result is determined at the PS side by the secondary majority voting among the incoming data buses *out_1, out_2, out_3*. If the result matches the reference (golden run), the failure mode is reported as masked. In the case of mismatch, the error flags from the voters are checked: if the error is reported for at least one of the replicas, then the failure mode is reported as signalled failure, otherwise a silent data corruption is reported. The bitstream is recovered at the end of each injection run. In the case of proper TMR implementation each FFI run should report the *masked* failure mode.

Since the first test is expected to register no failures in the case of proper TMR implementation, the FFI experiment doesn't quantify neither the sample proportion (percentage of failures), nor its error margin. Indeed, under less than five positive outcomes the sampling error can't be computed, since the sample proportion p would not be adequately approximated by normal distribution under any sample size n. As it is explained in [155], such approximation requires that $n \times p \geq 5$. Strictly speaking, to ensure that system tolerates all single upsets it is necessary to exhaustively test each of its essential bits (roughly 900 Kb for the AVR TMR assembly in this case).

Despite this can be accomplished in this case study (because of reasonably small DUT and short workload), in the more complex cases under the limited time

budget this would be infeasible. In such cases a compromise solution would be to still rely on sampling of essential bits, and to terminate the experiment either once the complete population is sampled, or once the time budget is exhausted. Despite in the latter case exist a non-zero probability of missing a critical bit (among those that have not been tested), it still should be low enough under the huge non-biased sample. Accordingly, all FFI experiments in this section are executed with the sample size goal of 100 thousands of injections, which is an equivalent of one hour time budget in this case study; such sample size equals roughly 40% of population for the simplex DUT, and roughly 15% of population for the TMR assembly. Additionally, an exhaustive FFI experiment will be performed in the first test (verification of TMR) to ensure that it indeed tolerates all single upsets.

Second FFI test quantifies the failure rate of an AVR replica λ_M, and of one of the voters λ_V, in order to estimate the reliability and the mission time of the resulting TMR assembly. The reliability of TMR with triplicated voters is calculated as:

$$R_{TMR} = (3R_M^2 - 2R_M^3) \times (3R_V^2 - 2R_V^3) \tag{9.1}$$

where $R_M(t) = e^{-\lambda_M \times K \times t}$, and $R_V(t) = e^{-\lambda_V \times K \times t}$ are the reliability of the replica and of the voter respectively. The mission time is calculated as the time in which the reliability falls below the threshold of 0.999, assuming the failure rate derating factor $K = 327.8$ (high-altitude applications). The CM cells corresponding to the AVR replica are located by matching the corresponding partial bitstream with the essential bit mask file. The essential bits of the voter are identified by means of LUT mapping, since the voters are completely implemented on LUTs. As it is depicted in Fig. 9.11-b, the injector application determines the failure mode for the AVR replica by analysing the error flags from the corresponding voter, since the TMR is expected to mask the failures of a single replica. When injecting SEUs into the voter, the resulting data outputs from that voter are also checked.

Finally, an FFI test depicted in Fig. 9.11-c will be used to evaluate the efficiency of SEU correction mechanism. This test targets the essential bits of the entire TMR assembly. The upsets are uniformly distributed along the whole set of essential bits, and accumulated between the injection runs. The DUT is recovered only when asserting an error flag from its replicas. Injector application is in charge of monitoring the error flags and reloading the partial bitstreams. The static partition is reconfigured only in the case of TMR failure.

9.4.2 Experimental results

The TMR verification test, after performing an exhaustive FFI campaign, has shown 100% of masked faults, which indicates that the considered TMR implementation, indeed, has no single point of failure. Furthermore, it is significantly more robust than the simplex version when multiple-bit upsets are considered – Table 9.19 compares the estimated failure rate between simplex and TMR for the SEUs of increasing multiplicity (under 100 thousands sampled upsets). As it can be seen, even when five CM cells are toggled at the same time, most TMR failures are signalled, being the percentage of SDC (unsafe failures) below 0.1%, whereas for the simplex version it exceeds 50%.

Table 9.19: Sensitivity to the SEUs of increasing multiplicity of simplex and protected (TMR) version of AVR IP

SEU multiplicity	Simplex	TMR	
	SDC	SDC	Signaled failures
1	17.57% ± 0.19%	0.0%	0.0%
2	31.42% ± 0.23%	0.008% ± 0.005%	3.21% ± 0.10%
3	42.78% ± 0.25%	0.024% ± 0.009%	6.89% ± 0.15%
4	51.94% ± 0.25%	0.047% ± 0.013%	11.51% ± 0.18%
5	–	0.075% ± 0.016%	16.174% ± 0.215%

Second FFI test, aiming at robustness evaluation of replicas and voters, has been performed for two different TMR implementations: the default and optimized one. The former is obtained under the default setting of Vivado parameters. The latter is obtained the under the optimal setting (labelled B1 in Table 9.17), previously determined for the simplex AVR version by means of DoE-based DSE. The corresponding results of this FFI experiment are summarized in Table 9.20. As it could be expected, the resulting failure rate for the replicas is quite similar to that of simplex version. The optimal tuning of Vivado parameters in case of TMR has reduced the failure rate of the replicas by 44% with respect to the default configuration (compared to 50% reduction for the simplex). The minor difference is explained by the fact that TMR replicas are implemented under the area constraints (bounded within the reconfigurable region (Pblock)), which affected the placement and routing of synthesized logic.

The deployed optimizations have lead to the significantly increased mission time. First, the TMR increased the mission time by 1970% (700 hours for simplex compared to 13790 hours for TMR). Afterwards, the tuning of Vivado parameters has increased the mission time by additional 44%. The overall improvement of mission time thus reached a factor of 28. It is worth noting that tuning of

Vivado parameters has also reduced the utilization of LUTs by 11%, thus slightly alleviating the area overheads of TMR.

Table 9.20: Resulting mission time for the simplex and TMR versions of AVR IP (under the default and optimized EDA parameters)

	Simplex		TMR	
	Default	EDA optimized	Default	EDA optimized
Failure rate (replica) λ_R	4.31 FIT	2.16 FIT	4.10 FIT	2.35 FIT
Failure rate (voter) λ_V	–	–	0.0025 FIT	0.0025 FIT
Mission Time (high-altitude) (R_{TH}=0.999, K=327.8)	700 h.	1410 h.	13 790 h.	19 960 h.

Finally, the SEU accumulation test has estimated the probability of DUT failure (percentage of failures) and the mean number of upsets injected before the failure. The corresponding results for different AVR versions (simplex, TMR and TMR with recovery of replicas) are summarized in Table 9.21. On the average, the simplex AVR fails after six CM upsets, the TMR AVR fails after 14 upsets, and the TMR with the recovery of replicas fails only after injection of 323 upsets. Accordingly, the recovery of replicas on the assertion of error flags is quite efficient in mitigation of SEUs, since its probability of failure is 23 times less than in case of TMR without that SEU correction mechanism. As expected, this mechanism is not perfect, since the static PL region is not protected against SEU accumulation. Accordingly, the CM scrubbing for the static PL region would be still required, but the amount of scrubbed CM cells is significantly reduced (and the scrubbing period can be increased), since the major part of essential bits (corresponding to the TMR replicas) is protected by means of partial reconfiguration.

Table 9.21: Probability of DUT failure under SEU accumulation: simplex (non-robust), TMR, and TMR with partial reconfiguration on error detection

		Simplex	TMR	TMR + PR
Optimized essential bits		322936	885984	885984
Sampled upsets		100000	100000	100000
Failures	Abs. (SDC/Signaled)	17534 / 0	3 / 7231	1 / 309
	p ± e	17.53% ± 0.20%	7.23% ± 0.15%	0.31% ± 0.03%
Mean upsets before failure		6	14	323

9.4.3 Discussion

Regardless of the complexity of defined fault tolerance mechanisms, the designers must verify the correctness of their implementation. Indeed, even for the simplest TMR assembly it must be verified that the synthesis has preserved all the voting logic intact, and no single point of failure has been introduced during placement and routing.

The presented experiment has illustrated how easily the DAVOS FFI tool can be adapted to verification of fault tolerance mechanisms and to dependability evaluation of resulting resilient designs. In comparison to the related works [74] [64], the proposed dependability assessment method has two advantages. On the one hand, by combining the autonomous FFI flow (deployed completely on FPGA chip) with optimized essential bits, the verification of fault tolerance mechanisms becomes faster and may reach higher coverage of fault space (even exhaustive verification becomes possible in some cases). On the other hand, by taking into account the actual number of critical bits within each TMR module, the proposed method leads to more realistic reliability estimations, than those based on approximate estimations of essential bits [74] and conservative assumption that every essential bit is critical [64].

Regarding the achieved reliability improvement, the careful isolation of each TMR replica within its dedicated FPGA region has prevented the occurrence of single point of failure. The estimated mission time of resulting resilient design (in the absence of SEU correction) has increased by 19.7 times. By adjusting the EDA parameters according to the optimal configuration (previously determined by DSE for non-protected design), the mission time has been increased by additional 44%; the overall improvement of mission time thus amounted 28.5 times. This also slightly reduced the TMR area overhead (from 3.0 times to 2.7). The detection of errors for each replica has allowed to reduce the probability of silent data corruption (unsafe failure) to less than 0.1% even under multiple-bit SEUs. Finally, by activating the partial reconfiguration of replicas on the assertion of error flags, the mean number of tolerated upsets has been increased by 23 times. In practice this could allow to proportionally reduce frequency of CM scrubbing.

9.5 Conclusions

The case study presented in this chapter has illustrated how the proposed methodology supports and improves the dependability-driven processes of an FPGA-based design flow. First, within the context of dependability benchmarking it has illustrated that (i) accurate robustness estimations can be obtained at different design representation levels by following the proposed fault injection methodology, (ii) upsets in the non-changeable CM have a dominating impact on the failure rate of considered soft-core processors, and (iii) the proposed SBFI/FFI performance optimizations notably speed-up the dependability assessment process.

After ensuring that FFI experiments are accurate and fast enough, this chapter has illustrated that the robustness of considered DUTs can be improved by tuning the synthesis, mapping, and placement-routing parameters through the proposed design space exploration methodology. It has been shown that both alternative DSE approaches (GA-based and DoE-based) notably improve the robustness and PPA attributes of the considered DUTs beyond the results of the default configuration. Resulting regression models may guide the designers in the tuning of Vivado suite for their own designs. These results allow designers to focus on tuning the most significant parameters (reducing the design space to explore) or even to directly rely on provided optimal settings, since their qualitative impact was very similar across the considered DUTs. Further research may generalize these results by considering a wider set of benchmark circuits.

Finally, this chapter has illustrated that derived optimal configurations are also valid for resilient designs. After instrumenting the design with SEU mitigation mechanisms and even after defining placement (area) constraints, the resulting reliability can still be nearly doubled by simply setting the EDA parameters to their sub-optimal levels without repeating the DSE experiments.

More concrete conclusions drawn from this case study are summarised below under three categories, according to the evaluated proposals of the thesis: (i) improvement of the accuracy of dependability-driven processes, (ii) improvement of fault injection performance, and (iii) dependability-aware design space exploration.

A. Improvement of the accuracy of dependability-driven processes:

- Register mapping reduces the gap in robustness estimations between RT-level and implementation-level SBFI by an order of magnitude;

- The proposed methodology provides accurate dependability estimations at RTL, implementation-level, and FPGA level when representative fault models are considered at each of these levels;
- Bit-accurate LUT mapping indicates that the actual number of LUT-specific essential bits is 20% to 50% less than reported by Vivado;
- Dependability benchmarking of soft-core processors, similar to those evaluated in this case study, should primarily consider upsets in non-changeable CM, since their contribution into the total failure rate may exceed 90%; it should also rely on FFI experiments since only at FPGA level it becomes possible to take into account the whole set of CM cells.

B. Efficiency of SBFI/FFI speed-up optimizations:

- Bit-accurate LUT mapping, in conjunction with the profiling of switching activity, accelerates the analysis of LUT-specific faults up to 4 times in comparison to the analysis relying on standard Xilinx essential bits;
- Multilevel SBFI accelerates the analysis of register-specific faults by 3 to 150 times in comparison to pure implementation-level SBFI.
- Clustering checkpoints accelerates the SBFI analysis of transient faults up to two times; checkpointing speed-up estimated by the proposed model coincides with the experimentally obtained speed-up;
- Iterative statistical fault injection accelerates the dependability benchmarking of FPGA designs up to six times with respect to the common conservative statistical approach;
- By combining the FPGA-based fault injection with the optimization of essential bits and the iterative statistical sampling, the dependability evaluation is accelerated up to the point when it cannot be considered any more as the bottleneck for DSE experiments.

C. Dependability-aware designs space exploration:

- Optimal tuning of synthesis, mapping, and placement-routing parameters improves the robustness of FPGA implementations by 20% to 52% with respect to the default (vendor-defined) configuration of Vivado suite;
- Optimal configurations of EDA parameters obtained by DSE for non-protected designs can be applied to the resilient versions of these designs and, in spite of additional placement constraints, they still provide quite high robustness gain (up to 44%);

- Those Vivado parameters contributing the most to improve the robustness of the final implementation are the synthesis *directive* and *hierarchy flattening*, and the route *directive* parameters;
- Dependability (failure rate) and performance (clock speed) are conflicting optimization goals. The proposed GA-based DSE approach identifies those Pareto-efficient configurations that simultaneously improve both of these attributes by 25% to 38%;
- The DoE-based DSE approach balances all the considered PPAD attributes within the WSM score and identifies the best EDA configurations that improve that score up to 36%;
- The GA-based DSE (implicitly optimizing both main effects and interactions) and the DoE-based DSE (optimizing only main effects) reach very similar optimization results;
- The integration of the proposed iterative selection strategy into the GA-based DSE accelerates the experimentation up to 3.3 times under the single-goal optimization and up to 1.9 times under the multi-objective optimization;
- To explore a design space comprising 5.35×10^{17} configurations by means of the proposed D-optimal DoE-based approach, it was enough to sample a very small fraction (just 168) of these configurations; the predictive regression models derived on its basis are accurate enough as to notably improve each PPAD attribute with respect to the default configuration.

Chapter 10

Conclusions and Future Work

This chapter summarises the conclusions and contributions of the presented work. Section 10.1 presents the conclusions with respect to each of the established objectives of the thesis. Section 10.2 enumerates the contributions to the field and presents the dissemination of research results in international conferences and journals. Section 10.3 describes the results of the international research stay carried out by the student in the framework of his thesis. Finally, Section 10.4 outlines different lines for future research that might be followed on the basis of the presented work.

10.1 Conclusions

In addition to the common performance, power consumption, and silicon area (PPA) constraints, the design of safety-critical embedded systems must meet rigorous dependability requirements. To this end, the common semicustom design flow should be complemented by three dependability-driven processes. First, dependability assessment and verification characterise the robustness of the design against the representative faultload, locate its weak points, and ensure the efficiency of integrated fault tolerance mechanisms. Second, dependability benchmarking allows to select the most suitable alternative IP cores, EDA tools, and implementation technologies, from the viewpoint of their dependability and PPA

features. Third, dependability-aware design space exploration (DSE) allows to optimally configure the parameters of selected IP cores and EDA tools to improve as much as possible the PPAD of resulting implementations. Simulation-based and FPGA-based fault injection (SBFI and FFI) are two main dependability evaluation instruments at the base of these dependability-driven processes.

In order to efficiently integrate the dependability-driven processes into the design flow, this work has tackled several research objectives. First, it was necessary to define a fault injection methodology, covering the entire semicustom design flow and enabling an accurate and detailed robustness analysis at each of its stages. Second, it was necessary to reduce as much as possible the SBFI/FFI experimental effort to make feasible the dependability assessment of complex HW designs at all description levels. It is especially critical the reduction of the experimental effort in the context of design space exploration, which deals with the evaluation of numerous alternative design configurations. Hence, the third problem addressed in this work was to define an efficient DSE strategy that would minimize the fault injection effort at the level of DSE scenario as a whole. Finally, the fourth objective was to develop a tool that would seamlessly integrate the dependability-driven processes into the design flow to support different hardware description languages, representation levels, fault models, EDA tools, and implementation technologies.

This work has proposed solutions for each of the aforementioned objectives, supporting the usefulness of the proposal by an extensive case study of three embedded soft-core processors. The benefits and limitations of the proposals, as well as the lessons learned with respect to each objective are summarised as follows.

Objective 1: *Study the capabilities and limitations of existing fault injection solutions with respect to their integration into the semicustom design flow. Define a new fault injection methodology addressing existing limitations and allowing an accurate and detailed robustness assessment at different stages of the design flow.*

The analysis of existing fault injection solutions has shown that none of them completely covers the needs of a dependability-driven design flow. Most SBFI solutions focus on RTL models, ignoring the impact of the implementation technology and EDA optimizations on the resulting dependability. Those few SBFI approaches that deal with implementation-level models are highly-intrusive and do not take into account the optimizations imposed by Verilog and VITAL standards. Existing FFI solutions, on the other hand, were found to provide rather coarse granularity of robustness analysis, which limits their capability to locate the weak points of the design.

The proposed fault injection methodology has addressed these limitations. The new SBFI approach enables the accurate simulation of logic faults in implementation-level models, while reducing to the minimum both the level of intrusion in the DUT and the simulation time overhead. It is based on generic fault injection operations defined by studying how the macrocells' architecture and optimizations deployed by VITAL and Verilog standards impact on fault injection capabilities. Most faults are injected by means of operations relying solely on the use of simulator commands, being completely non-intrusive. Some faults require operations that automatically tweak VITAL-compliant macrocells to enable the fault injection by means of simulator commands. All the defined tweaks keep the functionality, timing behaviour, and compatibility of macrocells with the VITAL standard. On the basis of defined operations a new fault dictionary format has been proposed. It unifies the specification of fault injection procedures for diverse fault models and macrocell libraries of any complexity. The adoption of this fault dictionary format by designers could standardize the specification of fault models across the macrocell libraries of different vendors, being compact, extensible and tool-independent.

The new FFI approach overcomes the accuracy limitations of existing FFI solutions by optimizing the location of essential bits. To this end, this work has studied the mapping of major types of logic resources of Xilinx 7-series FPGAs onto the underlying configuration memory. This study has led to the definition of bit-accurate LUT and BRAM mapping algorithms that improve the location of essential bits with respect to the Xilinx essential bits technology. On the one hand, the optimized essential bits refine the granularity of the FFI analysis, allowing to selectively target any given design scope (for the mapped macrocells) and, thus, to locate the weak points of the DUT. On the other hand, it also improves the performance of FFI experiments by filtering redundant fault targets, i.e. those LUT/BRAM-specific CM bits that are non-essential despite being conservatively reported as essential by Xilinx Vivado suite.

The defined fault injection methodology has several advantages with respect to existing solutions. First, it allows an accurate and low-intrusive fault analysis at all stages of the design flow. Second, it minimizes the gap in dependability estimations between the different design description levels. Third, the refined granularity of supported analysis allows the accurate location of weak points of the design with respect to its architecture and types of utilized logic resources. Two main limitations of the proposed approach are related to the FFI analysis: (i) it supports only Xilinx 7-series devices, and (ii) the routing-related CM cells remain unmapped, which keeps the analysis of routing faults rather coarse-grained.

Objective 2: *Define new fault injection speed-up techniques and refine existing ones to improve as much a possible the performance (reduce the experimentation effort) of dependability assessment at different design representation levels*

Fault injection performance has been improved from two different perspectives: reducing the fault lists and accelerating individual injection runs. Four different approaches have been proposed to this end.

First, the fault list is reduced (i) by the optimized identification of essential bits through the bit-accurate mapping of macrocells, and (ii) by filtering-out the inactive essential bits through profiling the switching activity. It has been experimentally found that combining these two approaches reduces the fault list up to 80% without any side effects on the accuracy of derived robustness estimates. In addition to that, the CM cells in the fault list are prioritized according to their profiled activity time. This allows to identify the maximum number of critical bits (weak points) under the limited experimental time budget, since the CM cells with higher activity time tend to be more critical than CM cells with lower activity time (as it has been experimentally demonstrated in this work).

Second, the fault list is sampled by the iterative statistical fault injection approach, which ensures the desired error margin in robustness estimates while significantly reducing the number of injection runs with respect to the common (conservative) statistical injection approach. The proposed iterative approach is most efficient under huge fault spaces and under small failure rates. It has been experimentally demonstrated that, under such conditions, it speeds-up the robustness assessment by more than an order of magnitude beyond the common conservative approach.

Third, the multilevel fault injection, supported by the proposed register mapping approach, distributes the SBFI experiments between RTL and implementation-level to minimize the experimental effort. The robustness assessment of sequential logic can be confidently carried out at RT level when the percentage of mapped registers is close to 100%, reducing the experimental effort up to two orders of magnitude. Under aggressive synthesis optimizations the SBFI experiments for the affected registers are offloaded to implementation level, while the rest of them are targeted at RTL. This keeps the accuracy of robustness estimations close to those obtained at pure implementation level while still reducing the experimental effort up to an order of magnitude.

Fourth, the clustering checkpointing approach has been proposed as the means to speed-up SBFI experiments. Its efficiency is dependent on the distribution of fault injection instants along the workload. Under a uniform fault distribution

it nearly doubles the experimental performance. The attainable speed-up and optimal number of clustering intervals for any given workload and faultload can be estimated by means of the analytical model provided in this work.

The case study has shown that combining all the proposed optimizations accelerates the dependability evaluation process up to the point when it cannot be considered any more as the bottleneck for the dependability-driven design flow.

Objective 3: *Define a design space exploration (DSE) methodology for dependability-aware tuning of EDA tools and IP cores.*

Two alternative DSE approaches have been proposed for the dependability-aware tuning of EDA/IP parameters to reduce the experimental effort from different perspectives.

The first proposed approach relies on genetic algorithms (GA) and reduces the experimental effort by means of the proposed *iterative selection* approach. The main advantage of this approach is that researchers do not need to decide beforehand the required error margin for dependability attributes, as they are dynamically refined until it becomes possible to confidently select the best individuals from the population. It has been experimentally demonstrated that integrating the iterative selection into the GA-based DSE process reduces the total dependability evaluation effort up to 3 times with respect to the GA relying on the common selection approach.

The GA-based approach can be recommended as the most generic DSE solution, as it handles the irregular design spaces with multi-level parameters and implicitly optimizes the higher-order effects (interactions of multiple parameters). Its main limitation is that depending on how it is calibrated, it may require to evaluate a significant number of configurations from the design space before reaching the close-to-optimum solution. In spite of that, in optimization scenarios similar to the presented case study, it may be generally sufficient to evaluate less than one hundred configurations (out of trillions of alternatives in the design space) to adjust the most significant effects and, thus, notably improve the robustness with respect to the default (vendor-defined) configuration.

The second proposed DSE approach reduces the experimental effort by minimizing the number of configurations that should be evaluated from the design space. This is achieved by combining the design of experiments (DoE) methodology with the regression analysis. The former representatively samples the design space using the smallest possible set of trials. The latter uses the collected sample to quantify the impact of each parameter on each PPAD attribute and to infer a predictive PPAD model, which in turn is used to determine a sub-optimal configuration of

parameters within the design space. Under regular design spaces the proposed method relies on orthogonal designs, which guarantee non-confounded estimations of parameters' effects and provide a small and deterministic sample size. To handle irregular design spaces with multi-level parameters this work has proposed a DSE flow based on the iterative augmentation and repair of D-optimal designs. Despite the latter does not guarantee the absence of correlation between the effects, its results can be improved on demand by incrementally extending the collected sample.

The main advantage of DoE-based approach is that it not only tunes the EDA/IP parameters, but also explains the impact of each parameter on each PPAD attribute. Its main limitation with respect to the GA-based approach is that the tuning of higher order effects (interactions of parameters) significantly increases the DSE effort. Nevertheless, as the case study has confirmed, interactions can be generally left out of consideration, as they are likely to be less important than main effects.

The experimental part of this work has demonstrated that the optimal tuning of EDA parameters by means of the proposed DSE approaches nearly doubles the robustness of FPGA-based designs with respect to the default vendor-defined EDA configuration. Under multi-objective DSE, they simultaneously improve several PPAD attributes both in the absence of subjective preferences (Pareto-efficient solutions) and in their presence. Furthermore, it has been demonstrated that determined optimal configurations are also valid for the resilient versions of the same designs: they provide nearly the same robustness improvement and simultaneously alleviate the PPA overheads imposed by integrated fault mitigation mechanisms.

Objective 4: *Provide an instrumental support for the efficient integration of dependability-driven processes into the semicustom/FPGA-based design flow.*

All the proposed methods and optimization techniques have led to the development of the *DAVOS toolkit*, which automates and seamlessly integrates the dependability assessment, verification, optimization, and selection processes into the design flow. Its modular architecture and rich customization capabilities make it compatible with standard HDLs, off-the-shelf EDA tools, and implementation technologies. At the same time, it provides an accurate and high-performance dependability evaluation instrument (*PPAD evaluation engine*), that can be used for the automation of any custom dependability-driven strategy.

It is worth noting that the DAVOS FFI tool currently supports only Xilinx 7-series FPGA and SoC devices. On the one hand, the carefully defined DAVOS

architecture should allow the future integration of specific modules to support FFI experiments for the devices of other series and/or vendors. On the other hand, DAVOS SBFI tool, being much more generic, can successfully replace the FFI tool in most dependability-driven processes that target other devices and implementation technologies, although with possibly lower experimental performance.

Finally, DAVOS is published as a free and open-source toolkit under the MIT license (`https://github.com/IlyaTuzov/DAVOS`). Thus, the community can benefit from the integration of new modules and EDA tools and researchers could share their results in a compatible format supported by rich visualization tools.

10.2 Summary of contributions an publications

Most contributions of this thesis have been published in reputed international journals and conferences. This section briefly describes each contribution, referencing the journal and/or conference paper in which it has been published.

10.2.1 Contributions of the thesis

- **Methodology for accurate simulation of fault effects in implementation-level HDL models**, described in Section 4, published in [J.1] and [C.7]. Defines the generic operators allowing low-intrusive fault injection into VITAL-based and Verilog-based macrocells. Establishes a tool-independent fault dictionary format, generalizing the definition of fault injection procedures for diverse fault models and implementation technologies.
- **Bit-accurate LUT and BRAM mapping algorithms**, described in Section 5.2. Locate the LUT and BRAM content within the configuration memory of Xilinx 7-series FPGAs with a bit granularity, taking into account all synthesis and placement optimizations.
- **Optimization of essential bits**, described in Section 5.3. Locates the essential bits corresponding to the LUT and BRAM macrocells with higher precision than by using the essential bits reported by Xilinx Vivado suite. Reduces the fault space for the LUT-specific and BRAM-specific faults without any side effects on the accuracy of SBFI/FFI analysis. Includes an algorithm for generating an optimized bit mask file, which is used in FFI experiments to target those CM cells that correspond to the selected design scope and type of logic resources.

- **LUT profiling approach**, described in Section 6.2.1, published in [C.5]. Identifies the inactive LUT bits, reducing the LUT-specific fault space. Prioritizes the criticality of the rest of LUT bits according to their estimated activity time.
- **Register mapping approach**, described in Section 6.3.1, published in [C.5]. Locates the RTL nodes corresponding to the implementation-level registers, taking into account the logic optimizations performed at synthesis. Reduces the gap in robustness estimations between RTL and implementation-level SBFI. Supports the multi-level fault injection for sequential logic.
- **Iterative statistical fault injection approach**, described in Section 6.2.2, published in [C.4]. Provides robustness estimates with a predefined error margin by sampling the smallest possible number of fault configurations. Reduces the experimental effort with respect to the common conservative statistical injection approach.
- **Iterative dependability-driven selection algorithm**, described in Section 7.3, published in [C.1]. Selects a predefined number of best (most robust) individuals from a list of alternatives, by performing a smallest possible number of SBFI/FFI runs. Reduces the experimental effort of GA-based DSE with respect to the common selection approach.
- **DoE-based dependability-aware DSE approach**, described in Section 7.2. Reduces the DSE experimental effort by minimizing the number of configuration sampled from the design space. Its version based on orthogonal designs (in application to regular designs spaces) has been published in [J.2] and [C.6]. Its version based on iterative refinement of D-optimal designs (in application to irregular designs spaces with multilevel parameters) has been published in [C.4].
- **DAVOS toolkit**, described in chapter 8, published in [C.3]. Seamlessly integrates all the considered dependability-driven processes into the semi-custom design flow. Includes a PPAD evaluation engine, that automates the implementation and dependability evaluation of multiple design configurations. Supports different hardware description languages, abstraction levels, fault models, EDA tools, and implementation technologies. Available in source codes under the MIT licence at `https://github.com/IlyaTuzov/DAVOS`.

10.2.2 Publications

Journal publications

[J.1] Ilya Tuzov, David de Andrés, and Juan-Carlos Ruiz. "Simulating the effects of logic faults in implementation-level VITAL-compliant models". In: *Computing* 101.2 (2019), pp. 77–96. DOI: 10.1007/s00607-018-0651-4. **(JCR Rank Q2)**

[J.2] Ilya Tuzov, David de Andres, and Juan-Carlos Ruiz. "Tuning synthesis flags to optimize implementation goals: Performance and robustness of the LEON3 processor as a case study". In: *Journal of Parallel and Distributed Computing* 112 (2018), pp. 84–96. DOI: 10.1016/j.jpdc.2017.10.002. **(JCR Rank Q2)**

Conference publications

[C.1] Ilya Tuzov, David de Andrés, and Juan-Carlos Ruiz. "Improving Robustness-aware Design Space Exploration for FPGA-based Systems". In: *2020 16th European Dependable Computing Conference (EDCC)* (Munich, Germany). IEEE. 2020. **(Distinguished paper award)**

[C.2] Ilya Tuzov, David de Andrés, and Juan-Carlos Ruiz. "Robustness-Aware Design Space Exploration Through Iterative Refinement of D-Optimal Designs". In: *2019 15th European Dependable Computing Conference (EDCC)* (Naples, Italy). IEEE. 2019, pp. 23–30. DOI: 10.1109/EDCC.2019.00017.

[C.3] Ilya Tuzov, David de Andrés, and Juan-Carlos Ruiz. "DAVOS: EDA toolkit for dependability assessment, verification, optimisation and selection of hardware models". In: *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (Luxembourg City, Luxembourg). IEEE. 2018, pp. 322–329. DOI: 10.1109/DSN.2018.00042. **(Core conference rank: A)**

[C.4] Ilya Tuzov, David de Andrés, and Juan-Carlos Ruiz. "Accurate robustness assessment of hdl models through iterative statistical fault injection". In: *2018 14th European Dependable Computing Conference (EDCC)* (Iasi, Romania). IEEE. 2018, pp. 1–8. DOI: 10.1109/EDCC.2018.00013. **(Distinguished paper award)**

[C.5] Ilya Tuzov, David de Andrés, and Juan-Carlos Ruiz. "Speeding-up robustness assessment of HDL models through profiling and multi-level fault in-

jection". In: *2018 Eighth Latin-American Symposium on Dependable Computing (LADC)* (Foz do Iguaçu, Brasil). IEEE. 2018, pp. 97–106. DOI: 10.1109/LADC.2018.00020. **(Best paper award)**

[C.6] Ilya Tuzov, David de Andrés, and Juan-Carlos Ruiz. "Dependability-aware design space exploration for optimal synthesis parameters tuning". In: *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (Denver,CO,USA). IEEE. 2017, pp. 121–132. DOI: 10.1109/DSN.2017.18. **(Core conference rank: A)**

[C.7] Ilya Tuzov, Juan-Carlos Ruiz, and David de Andrés. "Accurately simulating the effects of faults in vhdl models described at the implementation-level". In: *2017 13th European Dependable Computing Conference (EDCC)* (Geneva, Switzerland). IEEE. 2017, pp. 10–17. DOI: 10.1109/EDCC.2017.26. **(Distinguished paper award)**

[C.8] Ilya Tuzov et al. "Speeding-up simulation-based fault injection of complex hdl models". In: *2016 Seventh Latin-American Symposium on Dependable Computing (LADC)* (Cali, Colombia). IEEE. 2016, pp. 51–60. DOI: 10.1109/LADC.2016.18.

10.2.3 Research projects

The presented thesis has been developed in the Fault-tolerant Systems Research Group (GSTF) of ITACA institute of the Universitat Politècnica de València, in the framework of the *DINAMOS* research project "Mecanismos de adaptacion confiable para vehiculos autonomos y conectados" funded by the Spanish Ministry of Economy and Competitiveness under the reference number TIN2016-81075-R, from 30/12/2016 to 30/12/2020.

10.3 International research stay

Within the framework of the PhD internationalization programme, the student has carried out a 3-month research stay (from 01/09/2019 to 30/11/2019) at the University of Twente (The Netherlands), at the hosting group CAES (Computer Architecture for Embedded Systems). The research stay has been supervised by Dr. Daniel Ziener, a renowned expert in the domain of design and verification of reliable FPGA-based embedded systems.

The research work carried out during this stay has focused on the development of bit-accurate FPGA-based fault injection (FFI) techniques. The practical out-

come of this work was an approach for bit-accurate LUT and BRAM mapping and the related optimizations of essential bits, both presented among other contributions in this work. Overall, the student has gained an invaluable experience and knowledge in the domain of FPGA-based systems, that allowed to verify the proposals and to prove their usefulness for the related research areas. In particular, the collaboration with the CAES group has allowed to extend the application of the proposed bit-accurate FFI techniques and DAVOS toolkit to the domain of security assessment of FPGA-based systems.

10.4 Future work

Several research lines can be followed in the future on the basis of the work presented in this thesis.

First, a bit-accurate mapping of FPGA routing resources could be studied. This may allow additional optimizations of essential bits, aiming at further improving the accuracy and performance of fault injection analysis.

Second, additional strategies can be defined for further acceleration of fault injection analysis and of the related dependability-driven processes.

Third, extending the activity profiles to the rest of FPGA resources could allow to improve the efficiency of scrubbing the FPGA configuration memory. For instance, it could be worth to study whether the criticality of CM frames can be prioritized on the basis of aggregated activity metrics of its constituting CM cells.

Fourth, additional vendor-specific modules and fault dictionaries for the DAVOS toolkit can be developed, thus to extend an out-of-the-box support for different EDA tools and implementation technologies. Likewise, the DAVOS FFI tool can be adapted to FPGAs of different series and vendors.

Finally, DAVOS toolkit can be exploited for targeting different sets of complex IP cores considered as benchmarks and/or EDA tools. This may reveal certain patterns and tendencies among the configuration parameters, depending on the design characteristics or the desired trade-off, leading to the definition of guidelines for the robustness-aware tuning of IP cores and EDA tools.

Appendices

Appendix A

Details of Bit-accurate FPGA-based Fault Injection Approach

A.1 Accessing the configuration memory of Xilinx FPGAs

Configuration memory of Xilinx's FPGAs can be accessed through three different interfaces depicted in Fig.A.1. The JTAG TAP (test access port) provides an external access to the CM, e.g. from host PC. It is usually considered the slowest configuration path, although its throughput depends on the particular device. The Internal Configuration Access Port (ICAP) provide an access to the CM directly from reconfigurable fabric (PL), featuring the highest data rate – up to 400 MB/s. Finally, the Processor Configuration Access Port (PCAP) allows to access the CM from the hardwired part, available in Xilinx Zynq SoC devices. The latter is usually based on single or multi-core ARM APU. The PCAP throughput ranges between 19 MB/s and 100 Mb/s, although some proposals accelerate it up to 380 MB/s, as detailed in [164].

Regardless of the selected configuration path, the CM manipulation itself is performed by internal configuration logic (PL configuration module) according to

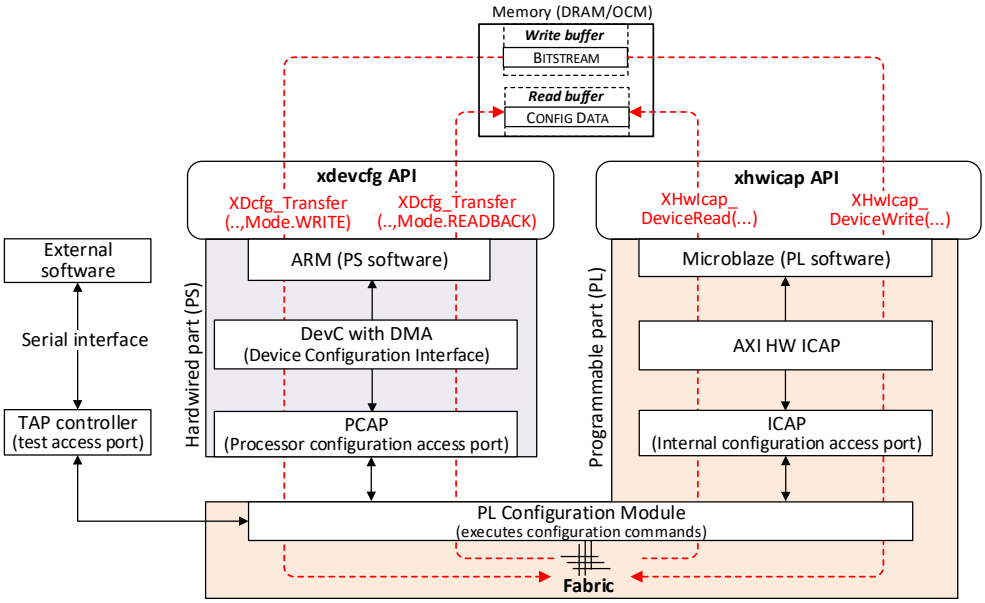


Figure A.1: ICAP, PCAP and JTAG paths for accessing the FPGA configuration memory

the incoming configuration bitstreams. The bitstream comprises a sequence of configuration commands and optional data packets. The bitstream commands instruct the configuration module to perform a required set of operations (read, write, capture, restore, protect, etc.). Hence, any CM access includes two main steps: preparing the configuration bitstream, and transferring it to the internal configuration module through the selected configuration path.

The bitstream composition for reading and writing of CM content (in non-secured unprotected mode), is detailed in Fig.A.2. Both read and write sequences start by synchronization packet, followed by various commands to prepare the CM manipulation (PO packet in Fig.A.2). Subsequently, the read or write parameters should be configured (RP packet). First, the operation mode is set to the command register: read (0x4) or write (0x1). After that, the starting frame address is set to the *FAR* register, and the total number of words to be read/written is set to *FDRO* register. Important to note that valid transaction should read-/write one additional dummy frame, to flush the data buffers, as it is detailed in [176]. The read packet is flushed by 32 dummy words, while the write packet is complemented by an array of CM data frames to be written to the CM. Finally, the bitstream is complemented by de-synchronization packet. Further details on bitstream composition can be found in [176].

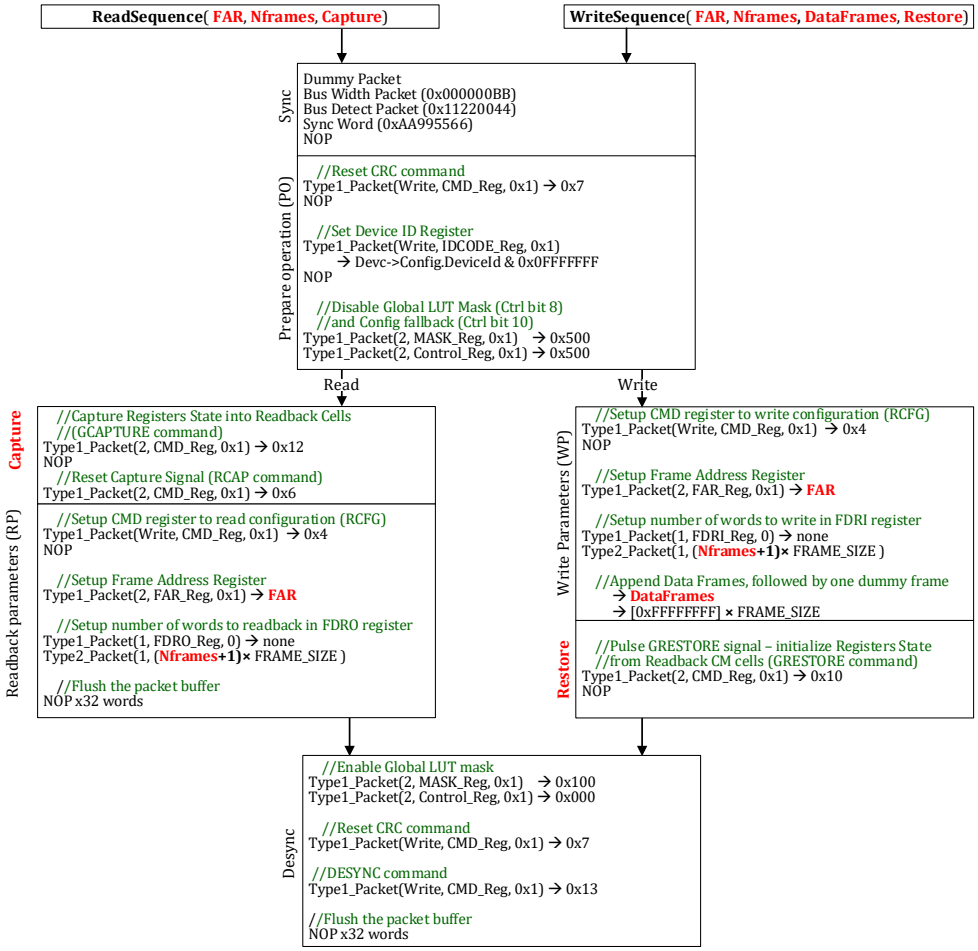


Figure A.2: Bitstream composition for reading and writing the FPGA configuration memory

Additionally, a state of the CLB registers can be readback from the device by appending a *GCAPTURE* command before readback parameters. This command copies the current state of all unprotected registers to their corresponding CM cells (INIT cells). Conversely, to change the state of fabric registers after updating the CM content, the *GRESTORE* command can be appended to the bitstream after the write parameters. Although, as it suggested by [67], the restore command in the bitstream may have no effect. In this case it may be required to forcibly trigger the Global Set-Reset line (GSR) through the *STARTUPE2* primitive af-

ter completing the CM write transaction. This primitive should be explicitly instantiated within the design.

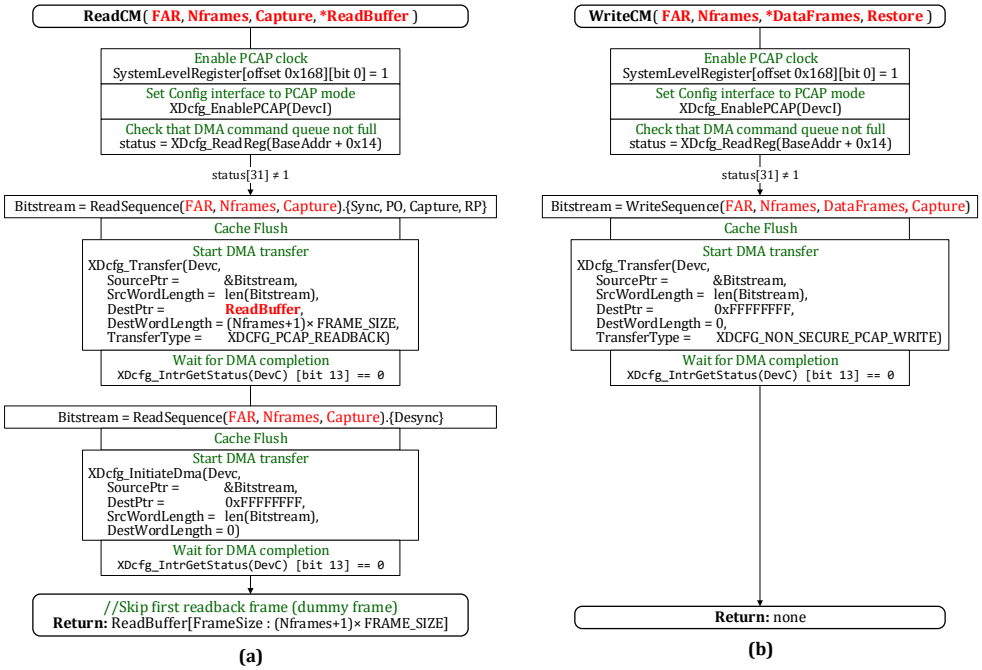


Figure A.3: Procedures to read (a) and write (b) the configuration memory through the PCAP interface

The software stack used to supply the bitstream to the configuration module, as well as to receive the readback data, relies on the API of the selected configuration path. The PCAP operates through the dedicated DMA (direct memory access) controller, and can be interacted from the PS software using the *xdevcfg* API. The ICAP path is accessible from both PS and and PL (Microblaze) software through the *hwicap* API. Fig.A.3 illustrates the procedures to read and write the CM through the PCAP. First of all, the preconditions are checked: (i) the configuration path is set to the PCAP mode, (ii) the PCAP clock is activated, and (iii) the DMA queue is not full. Subsequently, the configuration bitstream is build for a selected operation (write or readback) according to the previously described procedure. The bistream is stored in the on-chip/off-chip memory (WriteBuffer in Fig.A.1), from where it is transferred to the configuration module by invoking the *XDcfg_Transfer* function. The supported transfer types are defined in the PCAP library, particularly *XDCFG_PCAP_READBACK* constant denotes

a readback transaction, *XDCFG_NON_SECURE_PCAP_WRITE* constant denotes a write transaction.

In the readback mode the DMA transaction transfers the bitstream from the write buffer to the PL, and stores the received readback data in the selected read buffer. It is worth noting that in the readback mode the de-synchronization packet is transferred separately, after the readback data are received. In the write mode the DMA transaction is unidirectional (no data are returned from the CM), and the destination address reserved by PCAP API for this kind of transaction is 0xFFFFFFFF. Since the PCAP-DMA transaction is non-blocking, the PS software can perform some useful processing while waiting the DMA to complete. In case of ICAP API, all data transfer operations are blocking, although the throughput of ICAP path is generally higher.

A.2 Bit-accurate mapping of LUTs onto the configuration memory

The objective of bit-accurate LUT mapping is to discover the location of each bit of LUT's truth table (INIT property of LUT macrocell) in the bitstream fragment corresponding to it's placement (BEL). In other words, how the bits of extracted bitsream fragment should be reordered in order to obtain the INIT value of the LUT macrocell.

This can be experimentally achieved by assigning one-hot pattern to the INIT property of LUT macrocell (e.g. one bit set to logic 0, the rest bits are set to logic 1), placing it into the LUT BEL with the predefined coordinates (XY, A-D), and locating this highlighted bit (0) in the generated bitstream. By repeating this procedure for all LUT bits, one can determine their exact location in the extracted bitstream fragment. A detailed algorithm is depicted in Fig.A.4. It comprises two stages: i) generations of a bitstream file for each alternative location of highlighted bit in the LUT INIT (a total of 64 bitstream files), ii) location of this bit within the bitstream fragment for each generated bitstream.

On the implementation stage, first, the location of LUT6 cell (labelled as iLUT) is constrained to the LUT BEL with the predefined coordinates (e.g. X20_Y101, A6_LUT). The mapping of iLUT inputs onto the LUT BEL pins is constrained as well (e.g. direct mapping *I0:A1, I1:A2,..., I5:A6*) to prevent occasional Vivado optimizations impacting the mapping of configuration memory. Subsequently, one of the LUT INIT bits (with *bit_index* ranging from 0 to 63 (LSB to MSB)) is assigned a value of 0, while the rest of INIT bits are kept at 1. Afterwards, the design is implemented and the bitstream is generated. This procedure is

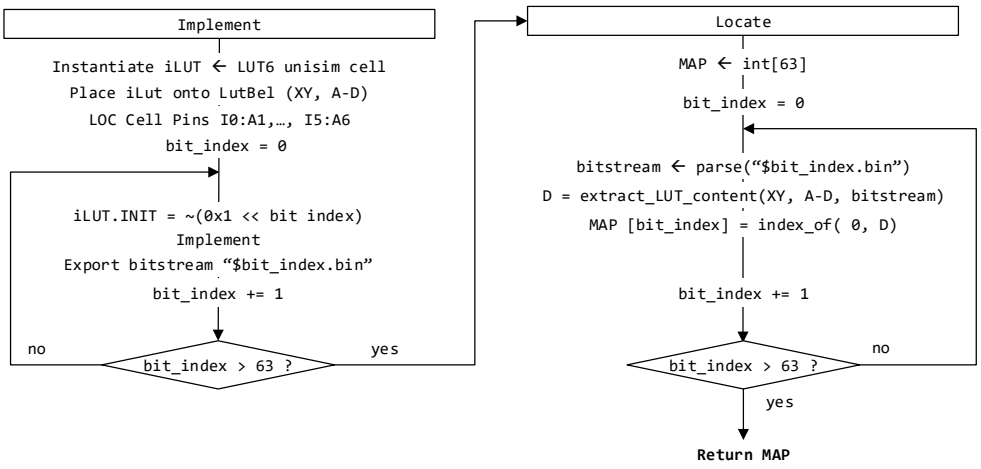


Figure A.4: Algorithm for locating the bits of LUT INIT (truth table) within the bitstream fragment

repeated for each *bit_index* in the range [0:63]. On the second stage the generated bitstreams are parsed in order to extract the bitstream fragment corresponding to the selected BEL coordinates (four halfwords are extracted from four consecutive Frames). Finally, the location of highlighted LUT bit (*bit_index* is determined within the extracted bitstream fragment, as it is depicted in Fig.A.5.

iLUT.INIT (64 bits) BEL: A6LUT (X113_Y18)	**BITSTREAM Fragment (64 bits):** TOP =1, HCLKROW=1, MAJOR = … , WORD=36, BITS [15:0]				**MAP**
	MINOR 26	MINOR 27	MINOR 28	MINOR 29	
1111111111….11111111110	0111 1111 1111 1111	1111 1111 1111 1111	1111 1111 1111 1111	1111 1111 1111 1111	**0 → 63**
1111111111….11111111101	1111 1111 1111 1111	0111 1111 1111 1111	1111 1111 1111 1111	1111 1111 1111 1111	**1 → 47**
1111111111….11111111011	1011 1111 1111 1111	1111 1111 1111 1111	1111 1111 1111 1111	1111 1111 1111 1111	**2 → 62**
1111111111….11111110111	1111 1111 1111 1111	1011 1111 1111 1111	1111 1111 1111 1111	1111 1111 1111 1111	**3 → 46**
… … …….. … …	…	…	…	…	…
1011111111….11111111111	1111 1111 1111 1111	1111 1111 1111 1111	1111 1111 1111 1111	1111 1111 1111 1110	**62 → 0**
0111111111….11111111111	1111 1111 1111 1111	1111 1111 1111 1111	1111 1111 1111 1110	1111 1111 1111 1111	**63 → 16**

Figure A.5: Excerpt of LUT mapping trace for LUT6 Cell under direct ping mapping

Mapping experiments are conducted under i) different coordinates of LUT BEL, ii) different types of CLB slice (L or M), iii) different Cell-BEL pin mapping (e.g. direct or reverse *I0:A6, I1:A5,…, I5-A1*), iv) different LUT cell sizes (LUT2/.../LUT6). Resulting mapping is summarized in Table A.1. As it can be seen, the resulting mapping depends on the type of CLB slice (L or M), as well as on the mapping of LUT inputs onto the BEL pins. In the case of partially

Table A.1: Mapping of the LUT content onto the bits of corresponding bitstream fragment

bit_index	Matching bit of bitstream fragment			
	LUT 6			LUT 4
	Slice L Direct pinmap[1]	Slice M Direct pinmap[1]	Slice L Reverse pinmap[2]	Slice L Direct pinmap[3]
0	63	31	63	51, 55, 59, 63
1	47	15	55	35, 39, 43, 47
2	62	30	59	50, 54, 58, 62
3	46	14	51	34, 38, 42, 46
4	61	29	15	49, 53, 57, 61
5	45	13	7	33, 37, 41, 45
6	60	28	11	48, 52, 56, 60
7	44	12	3	32, 36, 40, 44
8	15	63	61	3, 7, 11, 15
9	31	47	53	19, 23, 27, 31
10	14	62	57	2, 6, 10, 14
11	30	46	49	18, 22, 26, 30
12	13	61	13	1, 5, 9, 13
13	29	45	5	17, 21, 25, 29
14	12	60	9	0, 4, 8, 12
15	28	44	1	16, 20, 24, 28
16	59	27	62	—
17	43	11	54	—
18	58	26	58	—
19	42	10	50	—
20	57	25	14	—
21	41	9	6	—
22	56	24	10	—
23	40	8	2	—
24	11	59	60	—
25	27	43	52	—
26	10	58	56	—
27	26	42	48	—
28	9	57	12	—
29	25	41	4	—
30	8	56	8	—
31	24	40	0	—
32	55	23	47	—
33	39	7	39	—
34	54	22	43	—
35	38	6	35	—
36	53	21	31	—
37	37	5	23	—
38	52	20	27	—
39	36	4	19	—
40	7	55	45	—
41	23	39	37	—
42	6	54	41	—
43	22	38	33	—
44	5	53	29	—
45	21	37	21	—
46	4	52	25	—
47	20	36	17	—
48	51	19	46	—
49	35	3	38	—
50	50	18	42	—
51	34	2	34	—
52	49	17	30	—
53	33	1	22	—
54	48	16	26	—
55	32	0	18	—
56	3	51	44	—
57	19	35	36	—
58	2	50	40	—
59	18	34	32	—
60	1	49	28	—
61	17	33	20	—
62	0	48	24	—
63	16	32	16	—

[1] LOCK_PINS {I0:A1 I1:A2 I2:A3 I3:A4 I4:A5 I5:A6}
[2] LOCK_PINS {I0:A6 I1:A5 I2:A4 I3:A3 I4:A2 I5:A1}
[3] LOCK_PINS {I0:A1 I1:A2 I2:A3 I3:A4}

used LUTs, their content is replicated in the bitstream 2^K times, where K is the number of unused pins of the LUT BEL.

A.3 Determining the state of unused LUT pins

Xilinx Vivado suite replicates the content of partially used LUT BELs in such a way as to make the LUT output independent of the unused BEL pins (they are assumed don't care values). In FPGA, however, unused BEL pins are expected to have certain determined default state. The experimental setup described in this section allows to determine the actual state of these unused LUT pins in FPGA.

Experiment considers a simple design comprising LUT3 macrocell connected to the Zynq PS through the GPIO (three inputs, one output). The LUT inputs are mapped onto the BEL pins according to the Fig.A.6a: *I0:A1, I1:A2, I2:A3*. While the BEL pins *A4,A5,A6* remain unused. The INIT property (truth table) of the LUT can be assigned any non-trivial value, in order to prevent Vivado from optimizing-out the LUT cell. The bitstream is exported, and a simple application for the ARM_core0 (in PS part) is created to interact with the instantiated LUT3 macrocell through the GPIO interface - Fig.A.6b.

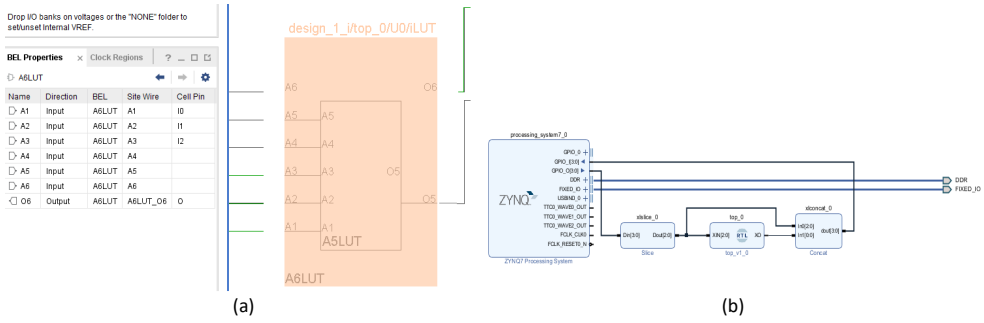


Figure A.6: Instantiated LUT with unused pins A4/A5/A6 (a), LUT connection to PS through GPIO interface

There exist 8 different input vectors (referred to as test sequence) *000,001,..., 111*, each of them drives one corresponding INIT bit to the LUT output. In the bitstream each of INIT bits is replicated 8 times, since there are 3 unused (don't care) pins *A6:A5:A4*. Therefore, with the original LUT content the test sequence will output the same result independently of the state of unused pins. In order determine the state of unused pins, the don't care assumption should be discarded by modifying the replicated LUT bits in such a way as to make the test result

dependent on the actual state of unused pins. For instance, the by modifying the LUT content according to the Table A.2 should allow to uniquely detect the state of *A6:A5:A4* from the test result, as it is listed below:
A6:A5:A4 = 000 → Result = 0x11,
A6:A5:A4 = 001 → Result = 0x22,
A6:A5:A4 = 010 → Result = 0x33,
A6:A5:A4 = 011 → Result = 0x44,
A6:A5:A4 = 100 → Result = 0x55,
A6:A5:A4 = 101 → Result = 0x66,
A6:A5:A4 = 110 → Result = 0x77,
A6:A5:A4 = 111 → Result = 0x88.

Table A.2: LUT content allowing to determine the state of unused BEL pins A6:A5:A4

LUT Input						LUT output		Bitstream bit	
A6	A5	A4	A3	A2	A1			LUT_L	LUT_M
0	0	0	0	0	0	1		63	31
0	0	0	0	0	1	0		47	15
0	0	0	0	1	0	0		62	30
0	0	0	0	1	1	0	0x11	46	14
0	0	0	1	0	0	1		61	29
0	0	0	1	0	1	0		45	13
0	0	0	1	1	0	0		60	28
0	0	0	1	1	1	0		44	12
0	0	1	0	0	0	0		15	63
0	0	1	0	0	1	1		31	47
0	0	1	0	1	0	0		14	62
0	0	1	0	1	1	0	0x22	30	46
0	0	1	1	0	0	0		13	61
0	0	1	1	0	1	1		29	45
0	0	1	1	1	0	0		12	60
0	0	1	1	1	1	0		28	44
0	1	0	0	0	0	1		59	27
0	1	0	0	0	1	1		43	11
0	1	0	0	1	0	0		58	26
0	1	0	0	1	1	0	0x33	42	10
0	1	0	1	0	0	1		57	25
0	1	0	1	0	1	1		41	9
0	1	0	1	1	0	0		56	24
0	1	0	1	1	1	0		40	8
0	1	1	0	0	0	0		11	59
0	1	1	0	0	1	0		27	43
0	1	1	0	1	0	1		10	58
0	1	1	0	1	1	0	0x44	26	42
0	1	1	1	0	0	0		9	57
0	1	1	1	0	1	0		25	41
0	1	1	1	1	0	1		8	56
0	1	1	1	1	1	0		24	40
1	0	0	0	0	0	1		55	23
1	0	0	0	0	1	0		39	7
1	0	0	0	1	0	1		54	22
1	0	0	0	1	1	0	0x55	38	6
1	0	0	1	0	0	1		53	21
1	0	0	1	0	1	0		37	5
1	0	0	1	1	0	1		52	20
1	0	0	1	1	1	0		36	4
1	0	1	0	0	0	0		7	55
1	0	1	0	0	1	1		23	39
1	0	1	0	1	0	1		6	54
1	0	1	0	1	1	0	0x66	22	38
1	0	1	1	0	0	0		5	53
1	0	1	1	0	1	1		21	37
1	0	1	1	1	0	1		4	52
1	0	1	1	1	1	0		20	36
1	1	0	0	0	0	1		51	19
1	1	0	0	0	1	1		35	3
1	1	0	0	1	0	1		50	18
1	1	0	0	1	1	0	0x77	34	2
1	1	0	1	0	0	1		49	17
1	1	0	1	0	1	1		33	1
1	1	0	1	1	0	1		48	16
1	1	0	1	1	1	0		32	0
1	1	1	0	0	0	0		3	51
1	1	1	0	0	1	0		19	35
1	1	1	0	1	0	0		2	50
1	1	1	0	1	1	1	0x88	18	34
1	1	1	1	0	0	0		1	49
1	1	1	1	0	1	0		17	33
1	1	1	1	1	0	0		0	48
1	1	1	1	1	1	1		16	32

The resulting INIT value equals *0x8844CC22AA66EE11*, which after reordering the bits according to the LUT_L mapping, converts to the bitstream fragment *0xAAFF0A0AA0A50550*. Instead of modifying the bitstream file, the LUT content can be updated directly from the PS application by means of a function *UpdateLutINIT(Top, Row, Column, XY, ABCD_index, LutContent)* from the library *InjectorLib.c* (part of the DAVOS toolkit). Listing A.1 illustrates an excerpt from the PS application, accomplishing the described test for discovering the state of unconnected BEL pins.

```
1  //Initialization of GPIO, PCAP, InjDesc
2     ......
3  XGpioPs_SetDirection(&PsGpioPort,    XGPIOPS_BANK2, 0x00000007);
4  XGpioPs_SetOutputEnable(&PsGpioPort, XGPIOPS_BANK2, 0x00000007);
5
6  //Update LUT content at BlockType=0, Top=0, ClkRow=0, TileColumn=20, Xs=28, Y=101, LUT_Index=0(A)
7  UpdateLutINIT(&InjDesc, 0,  0, 0, 20, 28, 101, 0, 0xAAFF0A0AA0A50550);
8
9  //Test the LUT output for I0-I2 = 000 → 111
10 u8 res = 0x00;
11 for(int i=0;i<8;i++){
12         XGpioPs_Write(&PsGpioPort, XGPIOPS_BANK2, i);
13         u8 bit  = (XGpioPs_Read(&PsGpioPort, XGPIOPS_BANK2) >> 3) & 0x1;
14         printf("%d -> %d\n",i, bit);
15         res = res | (bit<<i);
16 }
17 printf("Test Result on iLUT = 0x%02x\n", res);
```

Listing A.1: Excerpt from the PS application used to discover the state of unconnected BEL pins

The test result, obtained after running the application equals 0x88, which corresponds to value of logic '1' of all unconnected BEL pins (*A6:A5:A4 = 111*). By repeating this experiment for different alternative LUT placements, it can be concluded that unused LUT pins are actually driven high (logic '1'). This can be used to uniquely locate each LUT-specific essential bit within the bitstream, even when the LUT used partially and it content gets replicated due to the unused BEL pins.

A.4 Extracting the macrocells descriptors from implementation-level netlist

```
set exportdir [get_property DIRECTORY [current_project]]
set fout [open $exportdir/LUTMAP.csv w]
puts $fout "sep=;"
puts $fout "CellPath;CellType;SliceXY;TileXY;ClkRegionXY;Slice.BelLabel;CellINIT;CombinedLut;CellBelPinmap;"

#EXPORT PROPERTIES OF EACH LUT CELL IN THE NETLIST
foreach cell         [get_cells -hier -filter {PRIMITIVE_GROUP==LUT}] {
    set bel          [get_bels -of_objects $cell]
    set tile         [get_tiles -of_objects $bel]
    set clkreg       [get_clock_regions -of_objects $tile]
    set cellPath     [get_property NAME $cell];                  #Path/Name of LUT cell
    set cellType     [get_property PRIMITIVE_TYPE $cell];        #LUT size (LUT1/2/3/4/5/6)
    set sliceXY      [get_property LOC $cell];                   #XY coord of CLB slice
    set tileXY       [get_property NAME $tile];                  #XY coord of CLB tile
    set clkregXY     [get_property NAME $clkreg];                #XY coord of clock region
    set BelLabel     [get_property BEL $cell];                   #Slice type (L/M) and LUT label (A/B/C/D)
    set cellINIT     [get_property INIT $cell];                  #LUT INIT (truth table)
    set combinedLut  [get_cells -of_objects [get_bels \
        [expr {[string first "5LUT" $bel] >= 0 ? [regsub "5LUT" $bel "6LUT"] : [regsub "6LUT" $bel "5LUT"]}]]]

    puts -nonewline $fout [format "%s;%s;%s;%s;%s;%s;%s;%s;" \
                           $cellPath $cellType $sliceXY $tileXY $clkregXY $BelLabel $cellINIT $combinedLut]

    #extract cell-bel pinmap list of tuples {CelPin : BelPin}
    foreach cellpin [get_pins -of_objects $cell] {
        puts -nonewline $fout [format "{%s:%s}" \
                    [get_property REF_PIN_NAME $cellpin] \
                    [lindex [split [get_bel_pins -of_objects $cellpin] '/'] end] ]
        }
        puts $fout ";"
}

#EXPORT PROPERTIES OF LUT BELS NOT REFLECTED IN THE NETLIST (PASS-THROUGH OR CONSTANT)
foreach bel [get_bels -filter \
    {IS_USED==False && (TYPE==LUT5 || TYPE == LUT_OR_MEM5 || TYPE==LUT6 || TYPE == LUT_OR_MEM6) }] {

    set cellpins [get_pins -of_objects [get_bel_pins -of_objects $bel]];    #List of used bel pins
    set eqn [get_property CONFIG.EQN $bel];        #Equation implemented by LUT Bel (O5/O6= net or constant)

    if {[llength $cellpins] > 0 ||  [llength $eqn] > 0 } {
        set slice        [get_sites -of_objects $bel]
        set tile         [get_tiles -of_objects $bel]
        set clkreg       [get_clock_regions -of_objects $tile]
        set BelLabel     [lindex [split [get_property NAME $bel] '/'] end];  #Slice type (L/M) and LUT label
        set sliceXY      [get_property NAME $slice];                         #XY coord of CLB slice
        set clkregXY     [get_property NAME $clkreg];                        #XY coord of clock region
        set tileXY       [get_property NAME $tile];                          #XY coord of CLB tile
        set combinedLut  [get_cells -of_objects [get_bels \
            [expr {[string first "5LUT" $bel] >= 0 ? [regsub "5LUT" $bel "6LUT"] : [regsub "6LUT" $bel "5LUT"]}]]]

        if {[llength $cellpins] > 0} {
            #Pass-through LUT (one of the inputs forwarded to output)
            puts -nonewline $fout [format "PassThrough;LUT1;%s;%s;%s;%s;;%s;"  \
                                   $sliceXY $tileXY $clkregXY $BelLabel $combinedLut]
            foreach pin $cellpins {
                puts -nonewline $fout [format "{I0:%s}" \
                                       [lindex [split [lindex [get_bel_pins -of_objects $pin] 0] '/'] end] ]
            }
            puts $fout ";"
        }  elseif { [llength $eqn] > 0} {
            #LUT Bel used as constant (O6/O5 = 0/1)
            puts $fout [format "Constant;LUT0;%s;%s;%s;%s;;%s;;" $sliceXY $tileXY $clkregXY $BelLabel $combinedLut]
        }
    }
}
close $fout
```

Listing A.2: Vivado script for extracting the descriptors of LUT macrocells from the implementation-level netlist into a CSV-formatted file

Appendix B

Case Study Details

B.1 Architecture of the DUTs

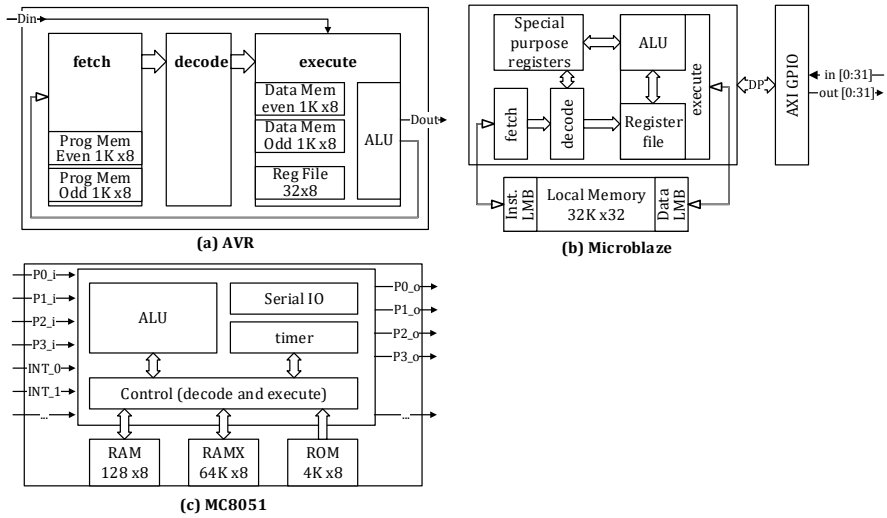


Figure B.1: Architecture of HW designs under study

B.2 Convergence of GA/NSGA-based DSE

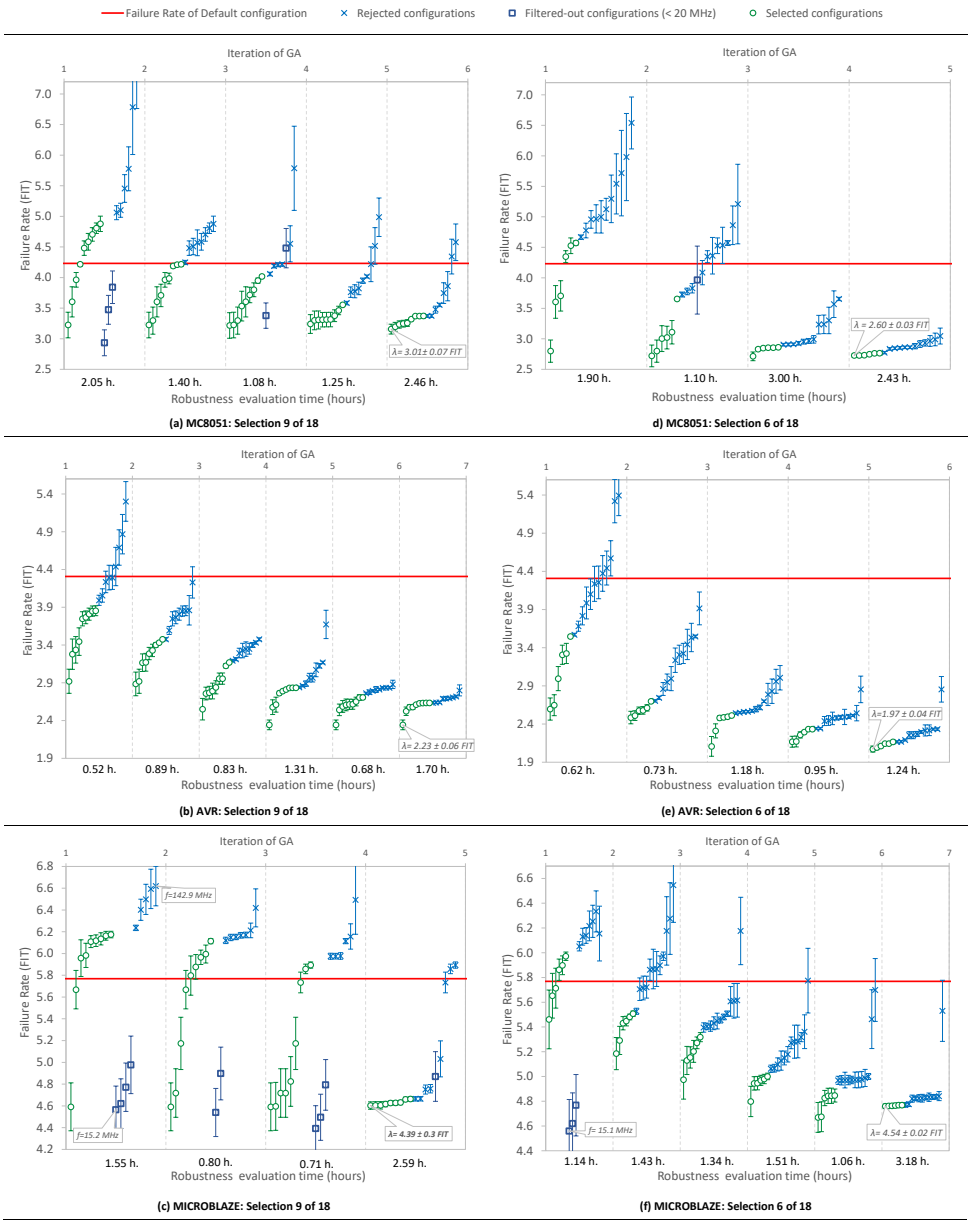


Figure B.2: Convergence of GA-based DSE process (single optimization goal - failure rate)

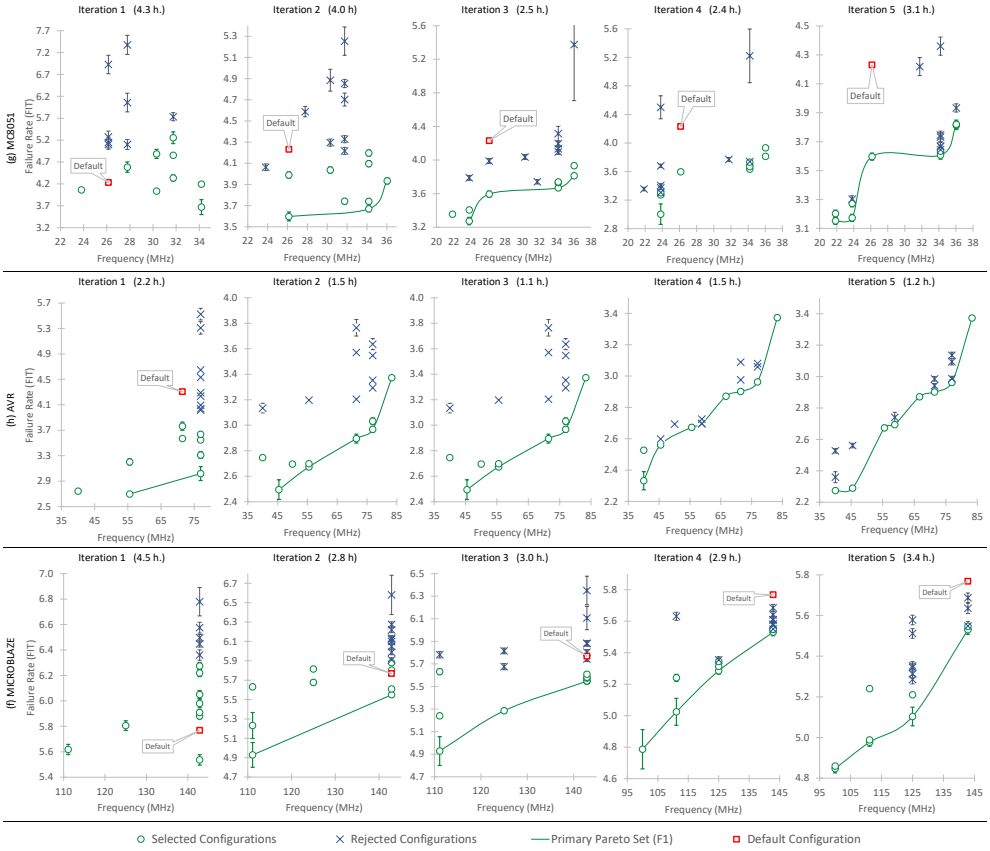


Figure B.3: Convergence of NSGA-based DSE process (two optimization goals: failure rate and frequency)

B.3 Regression models for PPAD attributes

Table B.1: Regression Models for dependability attributes (statistically significant terms)

Terms		Silent Data Corruption (%)			Failure Rate (FIT)		
		MC8051	AVR	Microblaze	MC8051	AVR	Microblaze
Intercept[1]		13.23	18,51	20.43	5.27	4.84	5.84
X01	1	5.50	-0.67		-0.35	-0.43	0.01
	2	0.27	-0.44		-0.38	-0.76	0.08
	3	1.11	1.58		0.16	-0.30	0.12
	4	0.15	-0.13		0.22	-0.11	0.00
	5	-0.01	-0.09		0.04	0.04	-0.07
	6	-0.28	-2.02		-0.45	-0.85	-0.03
	7	0.00	0.05		0.03	-0.13	0.02
X02	1	0.80	1.16	0.74	-0.13	-0.32	0.26
	2	0.30	-0.16	0.93	0.45	0.31	0.32
X03	1	0.07	-0.06			-0.15	-0.10
	2	0.57	0.29			-0.03	-0.07
X05	1		-0.75		-1.16	-0.71	-0.09
	2		-0.89		-1.26	-0.79	-0.14
	3		-0.68		-1.36	-0.78	-0.15
X07	1	0.63					
	2	0.30					
	3	0.93					
	4	0.20					
	5	1.00					
X08	1			-0.68	0.13		0.16
X09	1	-0.55		0.20			
	2	0.10		0.24			
X10	1		-0.11	0.14			
	2		0.19	0.06			
	3		-0.08	0.01			
	4		-0.45	-0.11			
	5		-0.31	-0.48			
X11	1		-0.68	-0.24	-0.11	-0.11	-0.12
X13	1						-0.05
	2						0.18
	3						0.19
	4						0.13
	5						0.15
X14	1	-8.83			0.79		
	2	-8.22			0.78		
	3	-7.54			0.69		
	4	-0.30			-0.05		
X15	1		-0.05				
	2		0.46				
	3		0.07				
	4		0.24				
X17	1						0.01
	2						-0.18
	3						-0.03
	4						-0.10
	5						-0.08
X20	1	-0.52			0.10		
X21	1		0.29	2.56		-0.08	-0.16
X22	1	-0.50					
	2	-0.70					
	3	-0.88					
	4	-1.00					
	5	-0.77					
	6	-1.76					
	7	-1.11					
X23	1		0.19	-0.18			0.16
X24	1		0.54	0.15			0.02
	2		0.36	-0.02			0.05
	3		0.13	-0.13			-0.16
	4		0.27	0.03			0.06
	5		0.80	0.38			0.07
	6		0.36	-0.10			-0.02
	7		0.56	0.20			-0.07
	8		0.58	0.12			-0.02
	9		0.15	0.20			0.04
	10		0.65	-0.07			-0.12
	11		0.63	0.15			0.03
	12		0.47	0.33			0.04
	13		0.61	0.49			-0.02
	14		0.48	0.41			-0.01
	15		0.44	0.35			0.00
	16		-0.64	-0.77			-0.38
	17		0.68	0.42			-0.06
X26	1	-0.71		0.24			0.07
X27	1					0.19	
	2					0.18	
	3					0.25	
	4					0.06	
	5					0.15	
	6					-0.04	
	7					-0.02	
	8					0.11	
X28	1		-0.16	0.14	0.20	0.11	0.06
	2		-0.19	0.21	-0.18	-0.15	0.02
	3		-0.13	0.04	-0.11	-0.09	-0.12
	4		-0.31	0.07	0.09	0.02	-0.03
	5		-0.12	-0.06	-0.14	0.10	-0.07
	6		-0.62	0.07	-0.13	-0.21	0.00
	7		-1.63	-1.78	-0.68	-0.75	-1.42
X29	1		0.29		0.40	0.18	0.07
Distribution		Normal	Normal	Normal	Normal	Normal	Normal
Determination Coefficient R^2		0.91	0.87	0.90	0.85	0.83	0.85

[1] Intercept computed under all factors set to level 0

Table B.2: Regression Models for frequency and power consumption (statistically significant terms)

Terms		Frequency (MHz)			Power consumption (mW)		
		MC8051	AVR	Microblaze	MC8051	AVR	Microblaze
Intercept[1]		26.91	72.23	133.38	0.0128	39.86	26.28
X01	1	-2.50	-1.35		0.0000	6.71	-1.60
	2	-3.37	-2.18		-0.0004	11.65	1.28
	3	-3.67	-2.63		-0.0010	11.35	3.53
	4	4.25	2.58		0.0017	1.61	-1.21
	5	0.80	-0.29		0.0008	3.20	0.42
	6	0.07	0.51		-0.0003	3.93	0.01
	7	3.48	0.37		0.0007	3.33	-0.02
X02	1			7.24	0.0017	-4.73	-3.49
	2			7.94	0.0019	-2.64	-3.64
X04	1	1.27	3.97			-3.53	
	2	-0.03	2.71			-1.90	
	3	1.10	2.17			-1.05	
	4	0.47	0.65			0.69	
X05	1				-0.0007	8.46	
	2				-0.0012	9.25	
	3				-0.0011	8.68	
X06	1		1.10				
X07	1						3.01
	2						1.51
	3						-0.10
	4						1.44
	5						-0.43
X09	1	-0.86	-1.98			2.66	
	2	0.75	0.30			-0.62	
X10	1				-0.0008	0.45	-6.27
	2				-0.0005	-1.63	-5.85
	3				0.0005	2.25	-1.00
	4				-0.0013	-2.10	7.63
	5				-0.0013	-2.33	5.56
X11	1	0.93			0.0008		
X12	1	-0.58					
X14	1				-0.0022		
	2				-0.0020		
	3				-0.0024		
	4				0.0006		
X16	1	0.05					
	2	-0.80					
	3	0.50					
	4	0.01					
X17	1		0.46				
	2		-1.95				
	3		-0.25				
	4		-1.02				
	5		-2.66				
X20	1		-1.18				
X21	1		1.12				6.89
X23	1	-0.47			-0.0015		1.46
X24	1	0.32	0.73				
	2	-0.27	-0.71				
	3	0.04	0.74				
	4	-0.22	-2.46				
	5	0.20	-1.12				
	6	-1.73	0.17				
	7	1.25	-0.17				
	8	0.49	-0.11				
	9	0.31	-0.14				
	10	0.40	0.48				
	11	-0.29	0.85				
	12	0.61	2.55				
	13	-0.86	2.15				
	14	0.62	1.46				
	15	-0.04	0.05				
	16	-1.58	-3.69				
	17	-0.48	0.75				
X25	1			1.76	-0.0010		
X28	1	-0.41	0.45	0.19	-0.0005	-0.31	0.14
	2	-1.64	-2.90	-3.71	-0.0011	1.99	-0.36
	3	-0.98	-2.31	-1.03	0.0002	1.21	-1.65
	4	-1.15	-1.91	-0.35	-0.0012	0.13	-0.52
	5	-1.71	-2.39	-0.78	0.0005	1.83	0.10
	6	-2.05	-5.13	-2.57	0.0000	2.42	-0.71
	7	-10.43	-25.57	-87.51	-0.0042	23.43	105.49
X29	1	4.56	5.90	3.80	0.0022	-8.37	
Distribution		Normal	Normal	Normal	Normal	Gamma	Gamma
Mean Func.		$V = X\beta$	$V = X\beta$	$V = X\beta$	$V = X\beta$	$V = \frac{1}{X\beta}$	$V = \frac{1}{X\beta}$
Determination Coefficient R^2		0.91	0.91	0.78	0.60	0.78	0.78

[1] **Intercept computed under all factors set to level 0**

Table B.3: Regression Models for area attributes (statistically significant terms)

MC8051				AVR				Microblaze			
Utilization FF		Utilization LUT		Utilization FF		Utilization LUT		Utilization FF		Utilization LUT	
Intercept	6.49	Intercept	8.00	Intercept	6.14	Intercept	7.39	Intercept	7.62	Intercept	7.24
X01=3	-0.11	X01=2	-0.15	X02=1	0.30	X01=2	-0.10	X02=1	-0.15	X08=1	0.09
X01=5	0.12	X01=3	-0.09	X02=2	0.29	X01=6	-0.12	X02=2	-0.15	X10=1	-0.03
X02=1	0.27	X05=1	-0.22	X06=1	0.03	X02=1	0.15	X05=1	-0.03	X10=5	0.05
X02=2	0.26	X05=2	-0.22	X08=1	0.02	X02=2	0.19	X05=2	-0.05	X11=1	0.06
X05=1	0.12	X05=3	-0.23	X01=2 : X02=1	-0.06	X05=1	-0.14	X05=3	-0.03	X12=1	-0.03
X14=1	0.37	X14=1	1.40	X01=2 : X02=2	-0.05	X05=2	-0.11	X07=1	0.05	X21=1	-0.19
X14=2	0.44	X14=2	1.41	X01=3 : X02=2	-0.07	X05=3	-0.12	X07=4	0.05	X23=1	0.04
X14=3	0.24	X14=3	1.24	X02=1 : X05=1	-0.30	X10=4	0.12	X08=1	0.23	X29=1	0.02
X17=1	-0.05	X01=1 : X14=1	-1.62	X02=2 : X05=1	-0.27	X10=5	0.06	X11=1	-0.02		
X17=4	-0.06	X01=2 : X14=1	0.14	X02=1 : X05=2	-0.30	X11=1	0.08	X13=2	0.14		
X01=1 : X05=1	-0.21	X01=5 : X14=1	-0.12	X02=2 : X05=2	-0.29	X21=1	-0.07	X13=3	0.13		
X01=5 : X05=1	-0.13	X01=1 : X14=2	-1.61	X02=1 : X05=3	-0.29	X23=1	-0.05	X13=4	0.17		
X01=1 : X05=2	-0.17	X01=2 : X14=2	0.11	X02=2 : X05=3	-0.28	X01=5 : X02=1	0.07	X13=5	0.14		
X01=3 : X05=2	0.13	X01=1 : X14=3	-1.41			X01=7 : X02=1	0.07	X20=1	0.02		
X01=2 : X05=3	0.09	X01=2 : X14=3	0.10			X01=2 : X02=2	-0.08	X21=1	-0.45		
X01=3 : X05=3	0.11	X01=5 : X14=3	-0.12			X01=3 : X02=2	-0.08	X23=1	-0.02		
X01=1 : X14=1	-0.20	X05=1 : X14=1	0.17			X01=6 : X10=2	0.09				
X01=5 : X14=1	-0.15	X05=2 : X14=1	0.15			X01=2 : X10=4	-0.14				
X01=1 : X14=2	-0.16	X05=3 : X14=1	0.16			X01=3 : X10=4	-0.12				
X01=2 : X14=2	-0.17	X05=1 : X14=2	0.17			X01=4 : X10=5	0.09				
X01=1 : X14=4	0.14	X05=2 : X14=2	0.16			X01=6 : X10=5	0.17				
X02=1 : X05=1	-0.34	X05=3 : X14=2	0.17			X01=1 : X11=1	-0.12				
X02=2 : X05=1	-0.34	X05=1 : X14=3	0.16			X02=1 : X05=1	-0.09				
X02=1 : X05=2	-0.27	X05=2 : X14=3	0.15			X02=2 : X05=1	-0.08				
X02=2 : X05=2	-0.27	X05=3 : X14=3	0.16			X02=1 : X05=2	-0.12				
X02=1 : X05=3	-0.27					X02=2 : X05=2	-0.11				
X02=2 : X05=3	-0.27					X02=1 : X05=3	-0.10				
X05=1 : X14=1	-0.14					X02=2 : X05=3	-0.10				
X05=2 : X14=1	-0.13					X05=1 : X11=1	0.08				
X05=3 : X14=1	-0.16					X11=1 : X23=1	-0.04				
X05=1 : X14=2	-0.18					X21=1 : X23=1	0.06				
X05=2 : X14=2	-0.14										
X05=3 : X14=2	-0.26										
X05=3 : X14=3	-0.10										
X05=1 : X14=4	-0.10										
X05=3 : X14=4	-0.11										
Determination Coefficient R^2	*0.96*	*0.99*		*0.95*		*0.97*		*0.94*		*0.80*	
Mean function	$V = e^{X\beta}$	$V = e^{X\beta}$		$V = e^{X\beta}$		$V = e^{X\beta}$		$V = e^{X\beta}$		$V = e^{X\beta}$	
Distribution	Poisson	Poisson		Poisson		Poisson		Poisson		Poisson	

[1] Intercept computed under all factors set to level 0

B.4 Comparison of experimentally obtained PPAD optimization results with the predicted ones

Table B.4: Predicted and actual PPAD results obtained for the best configurations

	Label	Optimization Goal	Predicted PPAD results[2]							Implemented (Actual) PPAD results[2]						
			λ (FIT)	SDC (%)	Freq. (MHz)	Power (W)	Util. FF	Util. LUT	Score Mission Critical	λ (FIT)	SDC (%)	Freq. (MHz)	Power (W)	Util. FF	Util. LUT	Score Mission Critical
MC8051	A1	Failure Rate λ	2.50	12.33	18.10	0.010	640	2172	0.83	–	–	–	–	–	–	–
	A2	Failure Rate λ[1]	2.55	12.63	18.10	0.010	654	2441	0.83	2.80	11.08	18.02	0.01	606	2420	0.76
	A3	SDC	4.21	0.58	25.04	0.010	764	11259	0.66	5.10	3.75	26.14	0.013	824	11317	0.53
	A4	Frequency	4.88	12.66	40.41	0.018	635	2417	0.62	4.64	12.81	36.04	0.021	612	2552	0.60
	A5	Power	4.19	5.68	12.95	0.00009	765	9107	0.64	3.81	11.55	15.87	0.008	682	2301	0.61
	A6	Util. FF	3.42	17.62	23.32	0.012	517	2509	0.74	4.22	12.35	23.81	0.013	607	2335	0.58
	A7	Util. LUT	3.87	17.92	23.32	0.014	680	2027	0.67	4.35	12.50	23.81	0.014	606	1906	0.57
	A8	Score Mission Critical	2.60	12.53	17.17	0.009	604	2172	0.90	–	–	–	–	–	–	–
	A9	Score Mission Critical[1]	2.65	12.84	17.17	0.009	617	2441	0.89	2.89	11.65	15.87	0.008	606	2114	0.76
		Vivado Default	4.22	12.41	25.83	0.01	662	2366	0.65	4.23	12.69	26.14	0.015	606	2187	0.64
AVR	B1	Failure Rate λ	1.80	15.42	49.96	0.014	467	1322	0.83	2.16	15.20	45.46	0.012	460	1474	0.83
	B2	SDC	2.35	12.19	45.15	0.013	465	1563	0.79	2.69	14.04	45.46	0.012	471	1742	0.71
	B3	Frequency	4.47	18.41	86.90	0.029	483	1464	0.60	4.25	17.98	76.93	0.023	445	1459	0.55
	B4	Power	2.46	16.52	43.63	0.012	465	1214	0.78	2.78	16.14	40.00	0.010	461	1193	0.70
	B5	Util. FF	3.99	20.38	70.51	0.017	439	1358	0.61	4.02	19.98	66.67	0.018	453	1273	0.56
	B6	Util. LUT	3.28	17.85	70.95	0.017	465	1160	0.69	3.61	17.44	71.43	0.019	461	1193	0.60
	B7	Score Mission Critical	1.94	15.42	47.78	0.014	467	1322	0.85	3.42	16.57	76.93	0.021	467	1243	0.62
		Vivado Default	4.39	18.52	73.89	0.020	461	1464	0.58	4.31	17.78	71.43	0.022	453	1484	0.53
MICROBLAZE	C1	Failure Rate λ	3.91	20.28	45.87	0.007	1209	1221	0.88	–	–	–	–	–	–	–
	C2	Failure Rate λ[1]	4.78	21.86	132.36	0.031	1209	1221	0.75	4.59	22.10	100.00	0.016	1295	1246	0.86
	C3	SDC	4.30	17.46	45.87	0.007	2326	1761	0.84	–	–	–	–	–	–	–
	C4	SDC[1]	5.42	18.56	132.61	0.030	2326	1761	0.71	5.39	18.90	125.00	0.023	2150	1660	0.77
	C5	Frequency	5.87	24.54	147.08	0.034	1065	1178	0.69	5.93	24.77	142.86	0.034	981	1091	0.71
	C6	Power	4.44	21.33	45.87	0.006	1265	1177	0.86	–	–	–	–	–	–	–
	C7	Power[1]	5.77	23.16	133.58	0.020	1265	1177	0.72	5.40	22.70	125.00	0.027	1295	1239	0.75
	C8	Util. FF	5.74	23.85	140.54	0.032	991	1275	0.70	5.68	23.46	142.86	0.031	981	1276	0.74
	C9	Util. LUT	5.68	24.55	140.54	0.043	1065	1093	0.70	5.64	24.10	142.86	0.043	985	1088	0.72
	C10	Score Mission Critical	4.04	20.23	45.87	0.007	1268	1191	0.91	–	–	–	–	–	–	–
	C11	Score Mission Critical[1]	4.98	21.80	132.36	0.024	1268	1191	0.79	4.63	22.28	111.11	0.012	1295	1257	0.92
		Vivado Default	5.67	24.28	140.54	0.030	1065	1155	0.7	5.77	24.64	142.86	0.033	981	1089	0.68

[1] Alternative configurations (sub-optimal) – if the global optimum is invalid (non-implementable)
[2] Highlighted cells denote optimization goal

Bibliography

[1] Emile Aarts, Jan Korst, and Wil Michiels. "Simulated Annealing". In: *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*. Ed. by Edmund K. Burke and Graham Kendall. Boston, MA: Springer US, 2005, pp. 187–210 (cit. on p. 136).

[2] M. Alderighi et al. "Using FLIPPER to predict irradiation results for VIRTEX 2 devices". In: *2008 European Conference on Radiation and Its Effects on Components and Systems*. 2008, pp. 300–305. DOI: 10.1109/RADECS.2008.5782731 (cit. on p. 37).

[3] Altera Corp. *Quartus Prime Pro Edition Handbook Volume 2: Design Implementation and Optimization*. 2015 (cit. on p. 29).

[4] D. de Andrés et al. *An Aspect-Oriented Approach to Hardware Fault Tolerance for Embedded Systems*. IGI Global, 2014, pp. 123–149 (cit. on p. 30).

[5] David de Andrés et al. "Fault emulation for dependability evaluation of VLSI systems". In: *IEEE transactions on very large scale integration (VLSI) systems* 16.4 (2008), pp. 422–431 (cit. on pp. 50, 62).

[6] David de Andrés Martínez. "Speeding-up model-based fault injection of deep-submicron CMOS fault models through dynamic and partially reconfigurable FPGAS". PhD thesis. 2008 (cit. on p. 48).

[7] L. Antoni, R. Leveugle, and B. Feher. "Using run-time reconfiguration for fault injection in hardware prototypes". In: *Proceedings IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*. 2000, pp. 405–413. DOI: 10.1109/DFTVS.2000.887181 (cit. on p. 48).

[8] Jean Arlat and Yves Crouzet. "Faultload representativeness for dependability benchmarking". In: *IEEE International Symposium on Dependable Systems and Networks (DSN) - Workshop on Dependability Benchmarking*. 2002, pp. 29–30 (cit. on p. 37).

[9] Jean Arlat, J-C Fabre, and Manuel Rodríguez. "Dependability of COTS microkernel-based systems". In: *IEEE Transactions on computers* 51.2 (2002), pp. 138–163 (cit. on p. 36).

[10] Jean Arlat et al. "Fault injection for dependability validation: A methodology and some applications". In: *IEEE Transactions on software engineering* 16.2 (1990), pp. 166–182 (cit. on pp. 35, 36).

[11] Arm Keil. *CA51 Compiler Kit* (cit. on p. 180).

[12] Sameh Attia and Vaughn Betz. "Feel Free to Interrupt: Safe Task Stopping to Enable FPGA Checkpointing and Context Switching". In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 13.1 (2020), pp. 1–27 (cit. on p. 131).

[13] Algirdas Avizienis et al. "Basic concepts and taxonomy of dependable and secure computing". In: *IEEE transactions on dependable and secure computing* 1.1 (2004), pp. 11–33 (cit. on pp. 1, 2, 4, 21–23).

[14] Dimiter Avresky et al. "Fault injection for formal testing of fault tolerance". In: *IEEE Transactions on Reliability* 45.3 (1996), pp. 443–455 (cit. on p. 34).

[15] Juan Carlos Baraza et al. "A prototype of a VHDL-based fault injection tool: description and application". In: *Journal of Systems Architecture* 47.10 (2002), pp. 847–867 (cit. on pp. 41, 47, 48).

[16] Juan Carlos Baraza et al. "Enhancement of fault injection techniques based on the modification of VHDL code". In: *IEEE Transactions on Very Large*

Scale Integration (VLSI) Systems. Vol. 16. 6. 2008, pp. 693–706 (cit. on p. 40).

[17] Giovanni Beltrame, Luca Fossati, and Donatella Sciuto. "Decision-theoretic design space exploration of multiprocessor platforms". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 29.7 (2010), pp. 1083–1095 (cit. on p. 136).

[18] Luis Alberto Contreras Benites and Fernanda Lima Kastensmidt. "Fault injection methodology for single event effects on clock-gated ASICs". In: *IEEE Latin American Test Symposium*. IEEE. 2017, pp. 1–4 (cit. on p. 40).

[19] A Benso and P Prinetto. *Fault Injection Techniques and Tools for VLSI reliability evaluation*. Frontiers In Electronic Testing. Kluwer Academic Publishers, 2003 (cit. on pp. 41, 69).

[20] J. Bergeron et al. *Verification Methodology Manual for SystemVerilog*. Springer US, 2006. ISBN: 9780387255569 (cit. on p. 12).

[21] Cinzia Bernardeschi et al. "Accurate simulation of SEUs in the configuration memory of SRAM-based FPGAs". In: *2012 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*. IEEE. 2012, pp. 115–120 (cit. on p. 39).

[22] Cinzia Bernardeschi et al. "ASSESS: A simulator of soft errors in the configuration memory of SRAM-based FPGAs". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 33.9 (2014), pp. 1342–1355 (cit. on p. 39).

[23] Paolo Bernardi et al. "A hybrid approach for detection and correction of transient faults in SoCs". In: *IEEE Transactions on Dependable and Secure Computing* 7.4 (2010), pp. 439–445 (cit. on p. 38).

[24] Luis Berrojo et al. "New techniques for speeding-up fault-injection campaigns". In: *Proceedings 2002 Design, Automation and Test in Europe Conference and Exhibition*. IEEE. 2002, pp. 847–852 (cit. on pp. 59, 63, 131, 202).

[25] Cristiana Bolchini, Antonio Miele, and Marco D Santambrogio. "TMR and Partial Dynamic Reconfiguration to mitigate SEU faults in FPGAs".

In: *22nd IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT 2007)*. IEEE. 2007, pp. 87–95 (cit. on p. 217).

[26] Thomas Bollaert. "Catapult Synthesis A Practical Introduction to Interactive C Synthesis". In: *High-Level Synthesis: from Algorithm to Digital Circuit*. Springer Netherlands, 2008, pp. 29–52 (cit. on p. 12).

[27] P. Bose. "Ensuring dependable processor performance: an experience report on pre-silicon performance validation". In: *International Conference on Dependable Systems and Networks*. 2001, pp. 481–486 (cit. on p. 30).

[28] M. Boulé and Z. Zilic. *Generating Hardware Assertion Checkers: For Hardware Verification, Emulation, Post-Fabrication Debugging and On-Line Monitoring*. SpringerLink: Springer e-Books. Springer Netherlands, 2008 (cit. on p. 12).

[29] George E. P. Box, J. Stuart Hunter, and William G. Hunter. *Statistics for experimenters: design, innovation, and discovery*. Wiley series in probability and statistics. Hoboken (N.J.): Wiley-Interscience, 2005 (cit. on p. 140).

[30] Ludovica Bozzoli and Luca Sterpone. "COMET: a configuration memory tool to analyze, visualize and manipulate FPGAs bitstream". In: *ARCS Workshop 2018; 31th International Conference on Architecture of Computing Systems*. VDE. 2018, pp. 1–4 (cit. on p. 56).

[31] Ludovica Bozzoli et al. "PyXEL: an integrated environment for the analysis of fault effects in SRAM-based FPGA routing". In: *2018 International Symposium on Rapid System Prototyping (RSP)*. IEEE. 2018, pp. 70–75 (cit. on pp. 40, 56).

[32] Brigham Young University. *BYU EDIF Tools Home Page*. 2015 (cit. on p. 4).

[33] A. A. M. Bsoul, N. Manjikian, and L. Shang. "Reliability- and process variation-aware placement for FPGAs". In: *Design, Automation Test in Europe Conference Exhibition*. 2010, pp. 1809–1814 (cit. on p. 30).

[34] Luis Andres Cardona and Carles Ferrer. "AC_ICAP: a flexible high speed ICAP controller". In: *International Journal of Reconfigurable Computing* 2015 (2015), p. 15 (cit. on pp. 52, 56).

[35] João Carreira, Henrique Madeira, and João Gabriel Silva. "Xception: A technique for the experimental evaluation of dependability in modern computers". In: *IEEE Transactions on Software Engineering* 24.2 (1998), pp. 125–136 (cit. on p. 36).

[36] J. Cavanagh. *Verilog HDL: Digital Design and Modeling*. CRC Press, 2017 (cit. on p. 15).

[37] E. Cerny et al. *SVA: The Power of Assertions in SystemVerilog*. Springer International Publishing, 2014 (cit. on p. 12).

[38] Athanasios Chatzidimitriou et al. "Demystifying soft error assessment strategies on ARM CPUs: Microarchitectural fault injection vs. neutron beam experiments". In: *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE. 2019, pp. 26–38 (cit. on p. 36).

[39] Pierluigi Civera et al. "An FPGA-based approach for speeding-up fault injection campaigns on safety-critical circuits". In: *Journal of Electronic Testing* 18.3 (2002), pp. 261–271 (cit. on p. 49).

[40] Cobham Gaisler AB. *LEON3FT fault tolerant processor* (cit. on p. 4).

[41] B. Cohen. *VHDL Coding Styles and Methodologies*. Springer US, 2012. ISBN: 9781461523130 (cit. on p. 46).

[42] Jason Cong et al. "High-level synthesis for FPGAs: From prototyping to deployment". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30.4 (2011), pp. 473–491 (cit. on p. 12).

[43] Cristian Constantinescu. "Impact of intermittent faults on nanocomputing devices". In: *DSN 2007 Workshop on Dependable and Secure Nanocomputing*. 2007 (cit. on p. 38).

[44] Sunil R Das et al. "An improved fault simulation approach based on verilog with application to ISCAS benchmark circuits". In: *IEEE Instrumentation and Measurement Technology Conference*. 2006, pp. 1902–1907 (cit. on p. 41).

[45] Kalyanmoy Deb et al. “A fast and elitist multiobjective genetic algorithm: NSGA-II”. In: *IEEE transactions on evolutionary computation* 6.2 (2002), pp. 182–197 (cit. on pp. 151, 152).

[46] Stefano Di Carlo et al. “A fault injection methodology and infrastructure for fast single event upsets emulation on Xilinx SRAM-based FPGAs”. In: *Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), 2014 IEEE International Symposium on*. IEEE. 2014, pp. 159–164 (cit. on p. 62).

[47] J.B. Dugan. “Reliability Analysis of Redundant and Fault-Tolerant Products”. In: *Product Reliability, Maintainability, and Supportability Handbook*. Ed. by M. Pecht. CRC Press, 2009, pp. 239–299 (cit. on p. 22).

[48] N. Einspruch. *Application Specific Integrated Circuit (ASIC) Technology*. VLSI Electronics. Elsevier Science, 2012 (cit. on p. 15).

[49] Henrik Esbensen and Ernest S. Kuh. “Design space exploration using the genetic algorithm”. In: *IEEE International Symposium on Circuits and Systems*. 1996, pp. 500–503 (cit. on pp. 136, 148).

[50] K.T. Fang, R. Li, and A. Sudjianto. *Design and Modeling for Computer Experiments*. Chapman & Hall/CRC Computer Science & Data Analysis. CRC Press, 2005. ISBN: 9781420034899 (cit. on p. 138).

[51] Z. Feng, N. Jing, and L. He. “IPF: In-Place X-Filling Algorithm for the Reliability of Modern FPGAs”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 22.10 (2014), pp. 2226–2229 (cit. on p. 30).

[52] Zhe Feng. “Logic Synthesis for FPGA Reliability”. PhD thesis. University of California, Los Angeles, 2013 (cit. on pp. 12, 30).

[53] V Fernandez et al. “Fault Modeling and Injection in VITAL Descriptions”. In: *Third Annual Atlantic Test Workshop*. 1994, o1–o4 (cit. on p. 46).

[54] Christian Fibich et al. “A netlist-level fault-injection tool for FPGAs”. In: *e & i Elektrotechnik und Informationstechnik* 132.6 (2015), pp. 274–281. DOI: 10.1007/s00502-015-0315-4 (cit. on p. 57).

[55] Christian Fibich et al. “FIJI: Fault InJection Instrumenter”. In: *EURASIP Journal on Embedded Systems* 2019.1 (2019), p. 2. DOI: 10.1186/s13639-019-0088-7 (cit. on p. 57).

[56] Shane Fleming and David Thomas. “Injecting FPGA Configuration Faults in Parallel”. In: *International Conference on Field-Programmable Technology (FPT)*. IEEE, 2018, pp. 201–208 (cit. on p. 64).

[57] Roberto Fontana and Sabrina Sampò. “Minimum-Size Mixed-Level Orthogonal Fractional Factorial Designs Generation: A SAS-Based Algorithm”. In: *Journal of Statistical Software* 53.10 (2013), pp. 1–58 (cit. on pp. 139, 144).

[58] David A. Freedman. *Statistical Models: Theory and Practice*. Cambridge University Press, 2009 (cit. on p. 140).

[59] Jesús Friginal et al. “Multi-criteria analysis of measures in benchmarking: Dependability benchmarking as a case study”. In: *Journal of Systems and Software* 111 (2016), pp. 105–118 (cit. on p. 27).

[60] S. Gazut et al. “Towards the Optimal Design of Numerical Experiments”. In: *IEEE Transactions on Neural Networks* 19.5 (2008), pp. 874–882. ISSN: 1045-9227. DOI: 10.1109/TNN.2007.915111 (cit. on p. 145).

[61] Davy Genbrugge and Lieven Eeckhout. “Chip Multiprocessor Design Space Exploration through Statistical Simulation”. In: *IEEE Transactions on Computers* 58.12 (2009), pp. 1668–1681 (cit. on p. 30).

[62] Daniel Gil et al. “Study, comparison and application of different VHDL-based fault injection techniques for the experimental validation of a fault-tolerant system.” In: *Journal of Systems Architecture* 34.1 (2003), pp. 41–51 (cit. on p. 40).

[63] Pedro Gil et al. *Fault Representativeness*. Tech. rep. Dependability Benchmarking project, 2002 (cit. on p. 37).

[64] Robért Glein et al. “Reliability of space-grade vs. COTS SRAM-based FPGA in N-modular redundancy”. In: *2015 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*. IEEE. 2015, pp. 1–8 (cit. on pp. 23, 201, 223).

[65] Qi Guo et al. “Microarchitectural design space exploration made fast”. In: *Microprocessors and Microsystems* 37 (2013), pp. 41–51 (cit. on p. 29).

[66] Matthew R. Guthaus et al. “MiBench: A free, commercially representative embedded benchmark suite”. In: *IEEE 4th Annual Workshop on Workload Characterization*. 2001, pp. 3–14 (cit. on p. 180).

[67] Markus Happe, Andreas Traber, and Ariane Keller. “Preemptive hardware multitasking in ReconOS”. In: *International Symposium on Applied Reconfigurable Computing*. Springer. 2015, pp. 79–90 (cit. on pp. 106, 130, 243).

[68] Jonathan Heiner et al. “FPGA partial reconfiguration via configuration scrubbing”. In: *2009 International Conference on Field Programmable Logic and Applications*. IEEE. 2009, pp. 99–104 (cit. on p. 4).

[69] R Hentschke et al. “Analyzing area and performance penalty of protecting different digital modules with Hamming code and triple modular redundancy”. In: *Proceedings. 15th Symposium on Integrated Circuits and Systems Design*. IEEE. 2002, pp. 95–100 (cit. on p. 3).

[70] David M Hiemstra and Valeri Kirischian. “Single event upset characterization of the Virtex-6 field programmable gate array using proton irradiation”. In: *2012 IEEE Radiation Effects Data Workshop*. IEEE. 2012, pp. 1–4 (cit. on p. 3).

[71] Martin Holzer, Bastian Knerr, and Markus Rupp. “Design space exploration with evolutionary multi-objective optimisation”. In: *2007 International Symposium on Industrial Embedded Systems*. IEEE. 2007, pp. 126–133 (cit. on pp. 136, 152).

[72] Ching Hu and Suhail Zain. “NSEU mitigation in avionics applications”. In: *Xilinx Application Note XAPP1073 v1. 0* (2010), pp. 1–12 (cit. on pp. 24, 180, 182).

[73] K. Huang, Y. Hu, and X. Li. “Reliability-Oriented Placement and Routing Algorithm for SRAM-Based FPGAs”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 22.2 (2014), pp. 256–269 (cit. on p. 30).

[74] Yoshihiro Ichinomiya et al. "Improving the robustness of a softcore processor against SEUs by using TMR and partial reconfiguration". In: *2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE. 2010, pp. 47–54 (cit. on pp. 217, 223).

[75] *IEEE Standard for Standard Delay Format (SDF) for the Electronic Design Process*. Standard. Institute of Electrical and Electronic Engineers, 2001 (cit. on p. 16).

[76] *IEEE Standard for VITAL ASIC (Application Specific Integrated Circuit) Modeling Specification*. Standard. Institute of Electrical and Electronic Engineers, 2000 (cit. on p. 17).

[77] Naveed Imran, R Ashraf, and Ronald F DeMara. "On-demand fault scrubbing using adaptive modular redundancy". In: *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms*. 2013, pp. 22–25 (cit. on p. 4).

[78] Intel. *Fault Injection Intel FPGA IP Core, User Guide*. 2019 (cit. on p. 57).

[79] Alessio Ishizaka and Philippe Nemery. *Multi-criteria Decision Analysis: Methods and Software*. Wiley, 2013, p. 310 (cit. on p. 27).

[80] Eric Jenn et al. "Fault injection into VHDL models: the MEFISTO tool". In: *International Symposium on Fault-Tolerant Computing*. 1994, pp. 66–75 (cit. on pp. 41, 47, 48, 64).

[81] Minyoung Jeong et al. "Extract LUT Logics from a Downloaded Bitstream Data in FPGA". In: *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE. 2018, pp. 1–5 (cit. on p. 56).

[82] P. Jones and P.H.D. Peter Jones. *Statistical Sampling and Risk Analysis in Auditing*. Taylor & Francis, 2017. ISBN: 9781351898010 (cit. on p. 59).

[83] Joseph M. Juran and Joseph A. De Feo. *Juran's Quality Handbook*. McGraw-Hill Education, 2010 (cit. on p. 139).

[84] Hubert Kaeslin. *Top-Down Digital VLSI Design: From Architectures to Gate-Level Circuits and FPGAs*. 1st. Morgan Kaufmann, 2014 (cit. on p. 10).

[85] Ghani A. Kanawati, Nasser A. Kanawati, and Jacob A. Abraham. "FERRARI: A flexible software-based fault and error injection system". In: *IEEE Transactions on computers* 44.2 (1995), pp. 248–260 (cit. on p. 36).

[86] Karama Kanoun and Lisa Spainhower. *Dependability benchmarking for computer systems*. Vol. 72. Wiley Online Library, 2008 (cit. on p. 26).

[87] Johan Karlsson et al. "Application of three physical fault injection techniques to the experimental assessment of the MARS architecture". In: *Dependable Computing and Fault Tolerant Systems* 10 (1998), pp. 267–288 (cit. on p. 36).

[88] Johan Karlsson et al. "Using heavy-ion radiation to validate fault-handling mechanisms". In: *IEEE micro* 14.1 (1994), pp. 8–23 (cit. on p. 36).

[89] Steve Kilts. *Advanced FPGA Design: Architecture, Implementation, and Optimization*. Wiley-IEEE Press, 2007 (cit. on p. 28).

[90] Steve Kilts. "Synthesis Optimization". In: *Advanced FPGA Design*. John Wiley & Sons, 2007. Chap. 14, pp. 205–227 (cit. on p. 42).

[91] Kyechong Kim and Agis A Iliadis. "Operational upsets and critical new bit errors in CMOS digital inverters due to high power pulsed electromagnetic interference". In: *Solid-state electronics* 54.1 (2010), pp. 18–21 (cit. on p. 3).

[92] Dirk Koch, Christian Haubelt, and Jürgen Teich. "Efficient hardware checkpointing: concepts, overhead analysis, and implementation". In: *Proceedings of the 2007 ACM/SIGDA 15th international symposium on Field programmable gate arrays*. ACM. 2007, pp. 188–196 (cit. on pp. 64, 130).

[93] Michael A Kochte et al. "Efficient fault simulation on many-core processors". In: *Design Automation Conference*. 2010, pp. 380–385 (cit. on p. 64).

[94] Johannes Maximilian Ku et al. "Testing reliability techniques for SoCs with fault tolerant CGRA by using live FPGA fault injection". In: *2013 International Conference on Field-Programmable Technology (FPT)*. IEEE. 2013, pp. 462–465 (cit. on pp. 51, 57).

[95] Maciej Kurek et al. “Automating Optimization of Reconfigurable Designs”. In: *IEEE 22nd International Symposium on Field-Programmable Custom Computing Machines*. 2014, pp. 210–213 (cit. on p. 30).

[96] Chris Lavin and Alireza Kaviani. “Rapidwright: Enabling custom crafted implementations for fpgas”. In: *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE. 2018, pp. 133–140 (cit. on p. 51).

[97] Robert Le. *Soft Error Mitigation Using Prioritized Essential Bits. XAPP538 (v1.0)*. 2012 (cit. on pp. 24, 53).

[98] R Leveugle. “Towards modeling for dependability of complex integrated circuits”. In: *5th IEEE International On-Line Testing workshop* (Rhodes, Greece). 1999, pp. 194–198 (cit. on p. 48).

[99] Régis Leveugle and K Hadjiat. “Multi-level fault injections in VHDL descriptions: alternative approaches and experiments”. In: *Journal of Electronic Testing* 19.5 (2003), pp. 559–575 (cit. on pp. 63, 132).

[100] Régis Leveugle et al. “Statistical fault injection: Quantified error and confidence”. In: *Design, Automation and Test in Europe*. 2009, pp. 502–506 (cit. on pp. 61, 110, 132, 201).

[101] Jens Lienig and Hans Bruemmer. “Reliability Analysis”. In: *Fundamentals of Electronic Systems Design*. Springer International Publishing, 2017, pp. 45–73. ISBN: 978-3-319-55840-0 (cit. on pp. 21, 22).

[102] Jacques-Louis Lions et al. *Ariane 5 flight 501 failure report by the inquiry board*. 1996 (cit. on p. 2).

[103] Tomislav Lovric. “Dependability Evaluation Methods”. In: *Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation*. Springer, 2003, pp. 41–48 (cit. on p. 35).

[104] Henrique Madeira et al. “RIFLE: A general purpose pin-level fault injector”. In: *European Dependable Computing Conference*. Springer. 1994, pp. 197–216 (cit. on p. 36).

[105] Azamat Mametjanov et al. “Autotuning FPGA design parameters for performance and power”. In: *IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*. 2015, pp. 84–91 (cit. on p. 30).

[106] Wassim Mansour and Raoul Velazco. “An automated SEU fault-injection method and tool for HDL-based designs”. In: *IEEE Transactions on Nuclear Science* 60.4 (2013), pp. 2728–2733 (cit. on p. 40).

[107] RJ Martínez et al. “Experimental validation of high-speed fault-tolerant systems using physical fault injection”. In: *Dependable Computing for Critical Applications 7*. IEEE. 1999, pp. 249–265 (cit. on p. 36).

[108] Peter Marwedel. *Embedded system design*. Vol. 1. Springer, 2006 (cit. on p. 1).

[109] R.W. Mehler. “Library modeling”. In: *Verilog HDL: Digital Design and Modeling*. Elsevier Science, 2014, pp. 337–360 (cit. on p. 15).

[110] Dimitrios Meidanis, Konstantinos Georgopoulos, and Ioannis Papaefstathiou. “FPGA Power Consumption Measurements and Estimations Under Different Implementation Parameters”. In: *International Conference on Field-Programmable Technology*. 2011, pp. 1–6 (cit. on p. 30).

[111] Mentor Graphics. *Precision RTL Plus*. 2016 (cit. on p. 4).

[112] Mentor Graphics. *Questa SIM Command Reference Manual 10.7b, Document Revision 3.5*. 2016 (cit. on pp. 41, 69).

[113] D. C. Montgomery. *Design and Analysis of Experiments*. 9th. New York: John Wiley & Sons, 2017 (cit. on pp. 139, 140).

[114] Douglas C. Montgomery. *Design and Analysis of Experiments*. USA: John Wiley & Sons, Inc., 2006. ISBN: 0470088109 (cit. on p. 137).

[115] David P Montminy et al. “Using relocatable bitstreams for fault tolerance”. In: *Second NASA/ESA Conference on Adaptive Hardware and Systems (AHS 2007)*. IEEE. 2007, pp. 701–708 (cit. on p. 4).

[116] Aurelio Morales-Villanueva and Ann Gordon-Ross. "HTR: on-chip hardware task relocation for partially reconfigurable FPGAs". In: *International Symposium on Applied Reconfigurable Computing*. Springer. 2013, pp. 185–196 (cit. on p. 107).

[117] Keith S Morgan et al. "A comparison of TMR with alternative fault-tolerant design techniques for FPGAs". In: *IEEE transactions on nuclear science* 54.6 (2007), pp. 2065–2072 (cit. on p. 217).

[118] R. Munden. *ASIC and FPGA Verification: A Guide to Component Modeling*. Systems on Silicon. Elsevier Science, 2004. ISBN: 9780080475929 (cit. on pp. 46, 114).

[119] Jongwhoa Na and Dongwoo Lee. "Simulated fault injection using simulator modification technique". In: *ETRI Journal* 33.1 (2011), pp. 50–59 (cit. on p. 64).

[120] Vijaykrishnan Narayanan and Yuan Xie. "Reliability concerns in embedded system designs". In: *Computer* 39.1 (2006), pp. 118–120 (cit. on p. 3).

[121] Michael Nicolaidis. *Soft errors in modern electronic systems*. Vol. 41. Springer Science & Business Media, 2010 (cit. on pp. 3, 217).

[122] Sergiu Nimara, Alexandru Amaricai, and Mircea Popa. "Sub-threshold CMOS circuits reliability assessment using simulated fault injection based on simulator commands". In: *IEEE International Symposium on Applied Computational Intelligence and Informatics*. 2015, pp. 101–104 (cit. on p. 41).

[123] NIST/SEMATECH. *NIST/SEMATECH e-Handbook of Statistical Methods*. 2013 (cit. on pp. 138, 139).

[124] Jose Luis Nunes et al. "FIRED–Fault Injector for Reconfigurable Embedded Devices". In: *Dependable Computing (PRDC), 2015 IEEE 21st Pacific Rim International Symposium on*. IEEE. 2015, pp. 1–10 (cit. on pp. 51, 57).

[125] Serafim Opricovic and Gwo-Hshiung Tzeng. "Extended VIKOR method in comparison with outranking methods". In: *European Journal of Operational Research* 178.2 (2007), pp. 514–529 (cit. on p. 27).

[126] Oregano Systems GmbH. *MC8051 IP Core, User Guide (V 1.2), 2013*. 2013 (cit. on p. 180).

[127] Patrick S Ostler et al. "SRAM FPGA reliability analysis for harsh radiation environments". In: *IEEE transactions on Nuclear Science* 56.6 (2009), pp. 3519–3526 (cit. on p. 3).

[128] Jacopo Panerati, Donatella Sciuto, and Giovanni Beltrame. "Optimization Strategies in Design Space Exploration". In: *Handbook of Hardware/Software Codesign*. Ed. by Soonhoi Ha and Jurgen Teich. Dordrecht: Springer, 2017. Chap. 6, pp. 189–217 (cit. on p. 136).

[129] B Parrotta et al. "New techniques for accelerating fault injection in VHDL descriptions". In: *ioltw*. IEEE. 2000, p. 61 (cit. on pp. 64, 132, 202).

[130] C. C. Peng, C. Dong, and D. Chen. "SETmap: A soft error tolerant mapping algorithm for FPGA designs with low power". In: *Asia and South Pacific Design Automation Conference*. 2011, pp. 388–393 (cit. on p. 30).

[131] Khoa Dang Pham, Edson Horta, and Dirk Koch. "BITMAN: A tool and API for FPGA bitstream manipulations". In: *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE. 2017, pp. 894–897 (cit. on pp. 40, 56).

[132] Christian Pilato et al. "Computational Intelligence in Expensive Optimization Problems". In: Springer-Verlag Berlin Heidelberg, 2010. Chap. Speeding-Up Expensive Evaluations in High-Level Synthesis Using Solution Modeling and Fitness Inheritance, pp. 701–723 (cit. on p. 30).

[133] Ludovic Pintard et al. "Fault injection in the automotive standard ISO 26262: an initial approach". In: *European Workshop on Dependable Computing*. Springer. 2013, pp. 126–133 (cit. on p. 25).

[134] Irith Pomeranz and Sudhakar M Reddy. "Safe fault collapsing based on dominance relations". In: *2008 13th European Test Symposium*. IEEE. 2008, pp. 7–12 (cit. on p. 58).

[135] Dyadem Press. *Guidelines for failure mode and effects analysis (FMEA), for automotive, aerospace, and general manufacturing industries*. CRC Press, 2003 (cit. on p. 25).

[136] H. Quinn and P. Graham. "Terrestrial-based radiation upsets: a cautionary tale". In: *IEEE Symposium on Field-Programmable Custom Computing Machines*. 2005, pp. 193–202 (cit. on pp. 24, 173, 182).

[137] R. Munden. *Inverter, STDN library, Free Model Foundry VHDL Model List*. 2000 (cit. on p. 19).

[138] Alexis Ramos, Juan Antonio Maestro, and Pedro Reviriego. "Characterizing a RISC-V SRAM-based FPGA implementation against Single Event Upsets using fault injection". In: *Microelectronics Reliability* 78 (2017), pp. 205–211 (cit. on pp. 51, 57).

[139] Chantal Robach and Mathieu Scholive. "Simulation-Based Fault Injection and Testing Using the Mutation Technique". In: *Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation*. Ed. by Alfredo Benso and Paolo Prinetto. Springer, 2003, pp. 195–215 (cit. on pp. 47, 48).

[140] 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*. 2008, pp. 265–272 (cit. on p. 30).

[141] T.L. Saaty. "Decision making with the analytic hierarchy process". In: *International Journal of Services Sciences* 1.1 (2008), pp. 83–98 (cit. on p. 27).

[142] Aitzan Sari and Mihalis Psarakis. "A Flexible Fault Injection Platform for the Analysis of the Symptoms of Soft Errors in FPGA Soft Processors". In: *Journal of Circuits, Systems and Computers* 26.08 (2017), p. 1740009 (cit. on p. 51).

[143] Aitzan Sari, Mihalis Psarakis, and Dimitris Gizopoulos. "Combining checkpointing and scrubbing in FPGA-based real-time systems". In: *2013 IEEE 31st VLSI Test Symposium (VTS)*. IEEE. 2013, pp. 1–6 (cit. on p. 130).

[144] Juergen Sauermann. *How to design your own CPU on FPGAs with VHDL*. 2010 (cit. on p. 180).

[145] Donald Shaw, Dhamin Al-Khalili, and Come Rozon. "Automatic generation of defect injectable VHDL fault models for ASIC standard cell li-

braries". In: *Integration, the VLSI Journal* 39.4 (2006), pp. 382–406 (cit. on p. 46).

[146] David Sheldon. "Design space exploration of parameterized systems using design of experiments". PhD thesis. UC Riverside, 2011 (cit. on p. 137).

[147] David Sheldon et al. "Application-Specific Customization of Parameterized FPGA Soft-Core Processors". In: *IEEE/ACM International Conference on Computer Aided Design*. 2006, pp. 261–268 (cit. on p. 30).

[148] Frederick T Sheldon, Stefan Greiner, and Matthias Benzinger. "Specification, safety and reliability analysis using Stochastic Petri Net models". In: *Tenth International Workshop on Software Specification and Design. IWSSD-10 2000*. IEEE. 2000, pp. 123–132 (cit. on p. 23).

[149] Volkmar Sieh, Oliver Tschache, and Frank Balbach. "VERIFY: Evaluation of reliability using VHDL-models with embedded fault descriptions". In: *International Symposium on Fault-Tolerant Computing*. 1997, pp. 32–36 (cit. on pp. 40, 47, 48).

[150] Leena Singh and Leonard Drucker. *Advanced Verification Techniques. A SystemC Based Approach for Successful Tapeout*. Frontiers In Electronic Testing. Springer US, 2004 (cit. on p. 45).

[151] D.J. Smith and K.G.L. Simpson. *Safety Critical Systems Handbook: A Straight forward Guide to Functional Safety, IEC 61508 (2010 EDITION) and Related Standards, Including Process IEC 61511 and Machinery IEC 62061 and ISO 13849*. Elsevier Science, 2010. ISBN: 9780080967820 (cit. on p. 25).

[152] Jared C Smolens et al. "Detecting emerging wearout faults". In: *Proc. of Workshop on SELSE*. 2007 (cit. on pp. 3, 38).

[153] L. Sterpone et al. "On the design of tunable fault tolerant circuits on SRAM-based FPGAs for safety critical applications". In: *2008 Design, Automation and Test in Europe*. 2008, pp. 336–341. DOI: 10.1109/DATE.2008.4484702 (cit. on p. 37).

[154] Luca Sterpone and Massimo Violante. "A new analytical approach to estimate the effects of SEUs in TMR architectures implemented through

SRAM-based FPGAs". In: *IEEE Transactions on Nuclear Science* 52.6 (2005), pp. 2217–2223 (cit. on p. 24).

[155] D.J. Sweeney and T.A. Williams. *Fundamentals of Business Statistics*. Cengage South-Western, 2010 (cit. on p. 219).

[156] Synopsys. *Synplify Premier*. 2015 (cit. on p. 4).

[157] Pradip A Thaker, Vishwani D Agrawal, and Mona E Zaghloul. "A test evaluation technique for VLSI circuits using register-transfer level fault modeling". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 22.8 (2003), pp. 1104–1113 (cit. on p. 42).

[158] The MathWorks, Inc. *Statistics and Machine Learning Toolbox™User's Guide*. 2016 (cit. on pp. 140, 142).

[159] S.K. Thompson. *Sampling*. CourseSmart. Wiley, 2012. ISBN: 9781118162941 (cit. on pp. 60, 61).

[160] Evangelos Triantaphyllou. "Multi-Criteria Decision Making Methods". In: *Multi-criteria Decision Making Methods: A Comparative Study*. Vol. 44. Applied Optimization. Springer US, 2000, pp. 5–21 (cit. on pp. 27, 143, 174).

[161] Ilya Tuzov, David De Andrés, and Juan-Carlos Ruiz. *Dependability Benchmarking of soft-core processors at different levels of design representation: MC8051, AVR and Microblaze as a case study*. 2020. DOI: 10.5281/zenodo.3996297 (cit. on p. 185).

[162] Pierre Vanhauwaert, Régis Leveugle, and Philippe Roche. "Reduced instrumentation and optimized fault injection control for dependability analysis". In: *2006 IFIP International Conference on Very Large Scale Integration*. IEEE. 2006, pp. 391–396 (cit. on p. 49).

[163] Igor Villata et al. "Fast and accurate SEU-tolerance characterization method for Zynq SoCs". In: *Field Programmable Logic and Applications*. 2014, pp. 1–4 (cit. on pp. 52, 57, 62).

[164] Kizheppatt Vipin and Suhaib A Fahmy. "ZyCAP: Efficient partial reconfiguration management on the Xilinx Zynq". In: *IEEE Embedded Systems Letters* 6.3 (2014), pp. 41–44 (cit. on pp. 52, 241).

[165] Laung-Terng Wang, Yao-Wen Chang, and Kwang-Ting Cheng. *Electronic Design Automation: Synthesis, Verification, and Test*. Morgan Kaufmann, 2009, p. 972 (cit. on p. 10).

[166] Jiesheng Wei et al. "Quantifying the accuracy of high-level fault injection techniques for hardware faults". In: *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE. 2014, pp. 375–382 (cit. on p. 36).

[167] Dagan White. "Considerations surrounding single event effects in FPGAs, ASICs, and processors". In: *White Paper: Xilinx FPGAs, WP402 (v1. 0.1),* (2012) (cit. on p. 39).

[168] J.M. Williams. *Digital VLSI Design with Verilog: A Textbook from Silicon Valley Polytechnic Institute*. SpringerLink : Bücher. Springer International Publishing, 2014 (cit. on p. 17).

[169] Marko Wolf and Michael Scheibel. "A systematic approach to a qualified security risk analysis for vehicular IT systems". In: *Automotive-Safety & Security 2012* (2012) (cit. on p. 25).

[170] Wayne Wolf. *FPGA-Based System Design*. Prentice Hall, 2004 (cit. on p. 27).

[171] Wayne Wolf. *Modern VLSI Design: IP-Based Design*. 4th. Prentice Hall, 2008 (cit. on p. 10).

[172] C.F.J. Wu and M.S. Hamada. *Experiments: Planning, Analysis, and Optimization*. Wiley Series in Probability and Statistics. Wiley, 2011. ISBN: 9781118211533 (cit. on pp. 138, 216).

[173] Hans-Joachim Wunderlich. *Models in hardware testing: lecture notes of the forum in honor of Christian Landrault*. Vol. 43. Springer Science & Business Media, 2009 (cit. on pp. 3, 37).

[174] Xilinx. *Synthesis and Simulation Design Guide, UG626 (v14.4)*. 2012 (cit. on pp. 71, 74, 76).

[175] Xilinx Inc. *7 Series FPGAs Configurable Logic Block. UG474 (v1.8)*. 2016 (cit. on p. 14).

[176] Xilinx Inc. *7 Series FPGAs Configuration UG470 (v1.13.1)*. 2018 (cit. on pp. 103, 242).

[177] Xilinx Inc. *7 Series FPGAs Memory Resources UG473 (v1.14)*. 2019 (cit. on pp. 54, 98).

[178] Xilinx Inc. *Command Line Tools User Guide*. 2013 (cit. on p. 29).

[179] Xilinx Inc. *Device Reliability Report, First Half 2018, UG116 (v10.9)*. 2018 (cit. on pp. 23, 24, 182).

[180] Xilinx Inc. *Embedded System Tools Reference Manual, UG1043*. 2019 (cit. on p. 180).

[181] Xilinx Inc. *MicroBlaze Processor Reference Guide, UG984*. 2019 (cit. on pp. 4, 180).

[182] Xilinx Inc. *Soft Error Mitigation Controller v4.1*. 2018 (cit. on pp. 4, 51, 53, 57).

[183] Xilinx Inc. *TMRTool*. 2016 (cit. on p. 4).

[184] Xilinx Inc. *Vivado Design Suite 7 Series FPGA and Zynq-7000 SoC Libraries Guide. UG953 (v2018.3)*. 2018 (cit. on pp. 15, 51).

[185] Xilinx Inc. *Vivado Design Suite User Guide. Implementation. UG904 (v2017.4)*. 2017 (cit. on p. 202).

[186] Xilinx Inc. *Vivado Design Suite User Guide. Synthesis. UG901 (v2017.4)*. 2017 (cit. on p. 202).

[187] Xilinx Inc. *XST User Guide for Virtex-6, Spartan-6, and 7 Series Devices (v 14.5)*. 2013 (cit. on p. 126).

[188] Xilinx Inc. *Zynq-7000 SoC. Technical Reference Manual. UG585 (v1.12.2).* 2018 (cit. on p. 52).

[189] Kwangsun Yoon. "A reconciliation among discrete compromise solutions". In: *Journal of the Operational Research Society* (1987), pp. 277–286 (cit. on p. 27).

[190] Charles R Yount and Daniel P Siewiorek. "A methodology for the rapid injection of transient hardware errors". In: *Computers, IEEE Transactions on* 45.8 (1996), pp. 881–891 (cit. on pp. 47, 48, 63).

[191] Yangyang Yu and Barry W. Johnson. "Fault Injection Techniques". In: *Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation*. Ed. by Alfredo Benso and Paolo Prinetto. Boston, MA: Springer US, 2003, pp. 7–39 (cit. on pp. 21, 58).