Fernanda Kastensmidt · Paolo Rech
Editors

FPGAs and Parallel Architectures for Aerospace Applications

Soft Errors and Fault-Tolerant Design

Springer

FPGAs and Parallel Architectures for Aerospace Applications

Fernanda Kastensmidt • Paolo Rech
Editors

FPGAs and Parallel Architectures for Aerospace Applications

Soft Errors and Fault-Tolerant Design

Springer

Editors
Fernanda Kastensmidt
Instituto de Informatica
Federal University of Rio Grande do Sul
Porto Alegre, Rio Grande do Sul, Brazil

Paolo Rech
Instituto de Informática
Federal University of Rio Grande do Sul
Porto Alegre, Rio Grande do Sul, Brazil

ISBN 978-3-319-14351-4 ISBN 978-3-319-14352-1 (eBook)
DOI 10.1007/978-3-319-14352-1

Library of Congress Control Number: 2015945021

Springer Cham Heidelberg New York Dordrecht London

Printed on acid-free paper

Springer International Publishing AG Switzerland is part of Springer Science+Business Media (www.springer.com)

Contents

Part I
Introduction

Chapter 1
Radiation Effects and Fault Tolerance Techniques for FPGAs and GPUs

Fernanda Kastensmidt and Paolo Rech

Abstract This book introduces the concepts of soft errors in FPGAs and GPUs. The chapters cover radiation effects in FPGAs, fault-tolerant techniques for FPGAs, use of COTS FPGAs in aerospace applications, experimental data of FPGAs under radiation, FPGA embedded processors under radiation, and fault injection in FPGAs. Since dedicated parallel processing architectures such as GPUs have become more desirable in aerospace applications due to high computational power, GPU analysis under radiation is also discussed.

1.1 Introduction

Field Programmable Gate Array (FPGA) components are very attractive for aerospace applications, as well for many applications at ground level that require a high level of reliability, as automotive, bank servers, processing farms, and others. The high amount of resources available in programmable logic devices can be applied to add flexibility to the on-board computer in satellites and to the automotive industry, for example. As FPGAs can be configured in the field, design updates can be performed until very late in the development process. In addition, new applications and features can be configured after a satellite is launched, or updated in hash environments. Modern FPGAs are System-on-Chip (SoC) composed of variety of soft and hard processors, embedded DSP and memories and a large number of complex configurable logic blocks able to customized to implement the user's design.

Graphics Processing Units (GPUs), traditionally employed to accelerate graphics rendering in personal computers or portable devices. In multimedia applications reliability is not a concern as the probability of failure is pretty low and a given number of errors are tolerated, as human eye could not distinguish them. Nevertheless, lately GPUs start to be employed also in applications in which reliability matters. Thanks to their efficiency, computing capabilities, and low power

F. Kastensmidt (✉) • P. Rech
Federal University of Rio Grande do Sul, Porto Alegre, Brazil
e-mail: fglima@inf.ufrgs.br; prech@inf.ufrgs.br

F. Kastensmidt, P. Rech (eds.), *FPGAs and Parallel Architectures for Aerospace Applications*, DOI 10.1007/978-3-319-14352-1_1

consumption compare to traditional CPUs, GPUs are in fact part of projects in the aerospace and automotive field. GPUs parallel capabilities could be exploited to compress images on satellites, to limit the bandwidth required to send them to ground. Additionally, GPUs are used to implement the Advanced Driver Assistance Systems (ADAS) that helps the driver to avoid accidents. Finally, GPUs are heavily employed as accelerators in High Performance Computing (HPC) centers. A large HPC center has thousands of GPUs that work in parallel, increasing significantly the probability of having at least one GPU corrupted by radiation.

Unfortunately both FPGAs and GPUs have been found to be very sensitive to radiation, mainly as they are fabricated in nanometric process technologies. It is fundamental to experimentally measure the soft error rate of the available resources, as well as the output error rate of specific applications, to evaluate if they meet the project reliability requirements. The experimental characterization of those programmable components and GPU are mandatory to sustain its applicability under transient faults. The test methodology and characterization of FPGAs and GPUs under radiation is needed to appropriate select and evaluate fault tolerant techniques to make those components more resilient to radiation. Radiation experiments, although complex and costly, are the only known and certified way to precisely measure the probability of failure in modern integrated circuits.

1.2 Radiation Effects

Integrated circuits operating in radiation environment are sensitive to transient faults caused by the interaction of ionizing particles with silicon. A particle is considered ionizing if it has the capability of dividing a quite atom into ions. Ionizing radiation generates failures in electronic devices as the deposited charge may perturb a transistor state. The charge may be deposited directly (if the ionizing particle is charged) or indirectly. Neutrons impact, for instance, generates secondary particles (alpha particles, ions, protons), which are charged and then may perturb a transistor. The interaction of the ionizing particles with the transistors may provoke transient and permanent effects depending on the location and amount of charge transferred (directly or indirectly) to the material as a consequence of the particle collision with the silicon.

The effects that are caused by a single event interaction are called Single Event Effects (SEE) and they can be transient or permanent [1]. When the SEE has a transient behavior, it is called a Soft Error, as the device is not permanently damaged. Examples of Soft Errors are Single Event Upset (SEU) and Single Event Transient (SET). An SEU is a bit-flip that occurs when the ionizing particle hitting a transistor of a memory cell deposits enough change to revert the state of the cell. The memory cell still works perfectly in the sense that a write or read operation is performed normally, but the stored information is corrupted. When the ionizing particle hits a logic cell, it generates a voltage spike that, if latched, leads to a SET. Again, the logic cell is not damaged in the sense that a new operation will eventually be

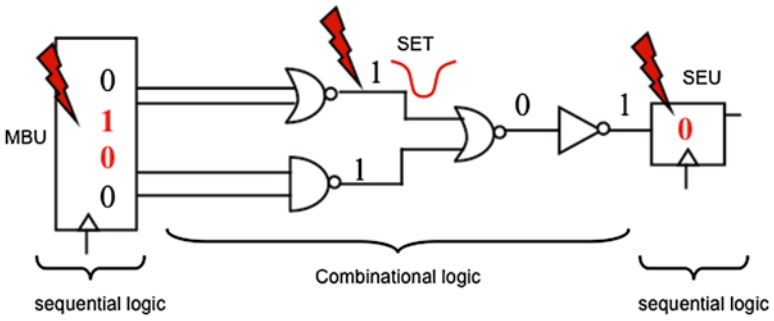


Fig. 1.1 SEU and MBU in the sequential logic and SET in the combinational logic

correctly performed. It is worth noting that the fact of being Soft does not reduce the severity of radiation-induced errors. On the contrary, the propriety of being transient and stochastic makes Soft Errors extremely hard to be identified and corrected. A permanent fault in a memory cell simply marks the cell as unused, while the possibility of having SEU makes the whole memory array as possible faulty. It is worth noting that with the shrink of transistor dimensions it is possible, for one single impinging particle, to interact with more than one transistor, generating a Multiple Cell Upset (MCU) in memory arrays. If the corrupted bits belong to the same memory word the MCU is called Multiple Bit Upset (MBU). MBU are particularly critical as they undermine the effectiveness of Error Correcting Codes (ECC). Figure 1.1 exemplifies SEU, MBU and SET in integrated circuits.

Radiation can generate also permanent faults as Single Event Latchup (SEL), Single Event Gate Rupture (SEGR), or Single Event Burnout (SEB). Finally, the accumulation of particle interactions causes an effect named Total Ionizing Dose (TID) and it represents degradation in the performance of the transistors as it modifies the threshold voltage and leakage current.

The radiation environment is composed of various particles generated by sun and stars activity [2]. The space is full of galactic cosmic rays, which are heavy ions produced by explosion of supernovas or collisions among celestial bodies. The atoms released, wondering around the universe, loses protons or electrons and, thus, gain charge. Interacting with the magnetic fields of planets and stars those ions are accelerated, reaching energies in the order of GeV. The sun produces a flux of protons and electrons, which reach the earth with low energies as they do not have sufficient time to be accelerated.

The particles can be classified as two major types: (1) energetic particles such as neutrons, electrons, protons and heavy ions, and (2) electromagnetic radiation (photons), which can be X-ray, gamma ray, or ultraviolet light. The main sources of energetic particles that contribute to radiation effects are protons and electrons trapped in the Van Allen belts, heavy ions trapped in the magnetosphere, galactic cosmic rays and solar flares. The charged particles interact with the silicon atoms causing excitation and ionization of atomic electrons.

At the ground level, neutrons are the most frequent cause of upset. Neutrons are created by cosmic ion interactions with the oxygen and nitrogen in the upper atmosphere. It is worth noting that while the solar wind is trapped in the Van Allen belts due to its low energy, galactic cosmic rays are so energetic to pass the belts and hit the upper level of the terrestrial atmosphere. The neutron flux is strongly dependent on key parameters such as altitude, latitude and longitude. There are high-energy neutrons that interact with the material generating free electron hole pairs and low energy neutrons. Those neutrons interact with a certain type of Boron present in semiconductor material creating others particles. Alpha particles are secondary types of particles emitted from interactions with radioactive impurities present in the device itself or in the packaging materials and they are the greatest concern. Materials aim to minimize the emission of alpha particles. However, it does not eliminate the problem completely.

As an energetic particle traverses the material of interest for instance a reverse-biased n+/p junction, it deposits energy along its path, as detailed explained in [3]. This energy is measured as a linear energy transfer (LET), which is defined as the amount of energy deposited per unit of distance traveled, normalized to the material's density. It is usually expressed in MeV-cm^2/mg. The total number of charges is proportional to the LET of the incoming particle. Depending on the fabrication details and the electrical characteristics of each sensitive node such as resistance and capacitance, different amplitude and duration of the transient voltage pulse are generated.

1.3 Soft Errors in FPGAs

Field-Programmable Gate Arrays (FPGAs) are configurable integrated circuit based on a high logic density regular structure, which can be customizable by the end user to realize different designs. The FPGA architecture is based on an array of logic blocks and interconnections customizable by programmable switches. Several different programming technologies are used to implement the programmable switches. There are three types of such programmable switch technologies currently in use: SRAM, where the programmable switch is usually a pass transistor or multiplexer controlled by the state of a SRAM bit (SRAM based FPGAs); Antifuse, when an electrically programmable switch forms a low resistance path between two metal layers (Antifuse based FPGAs); and EPROM, EEPROM or FLASH cell, where the switch is a floating gate transistor that can be turned off by injecting charge onto the floating gate.

Customizations based on SRAM are volatile. This means that SRAM-based FPGAs can be reprogrammed as many times as necessary at the work site and that they loose their contents information when the memories are not connected to the power supply. The antifuse customizations are non-volatile, so

they hold the customizable content even when not connected to the power supply and they can be programmed just once. Each FPGA has a particular architecture. Programmable logic companies such as Xilinx, MicroSemi, Aeroflex (licensed for Quicklogic FPGAs), Atmel and Honeywell (licensed for Atmel FPGAs) offer radiation tolerant FPGA families. Each company uses different mitigation techniques to better take into account the architecture characteristics.

1.3.1 Single Event Effects on SRAM-Based FPGAs

The SRAM-based FPGA is composed of an array of configurable logic blocks (CLB), a complex routing architecture, an array of embedded memories (Block RAM), an array of digital signal processing components (DSP) and a set of control and management logic. The CLBs are composed of Look-up Table (LUT) that implements the combinational logic, and flip-flops (DFF) that implements the sequential elements. The routing architecture can be very complex and composed of millions of pre-defined wires that can be configured by multiplexers and switches to build the desirable routing.

The configuration of all CLBs, routing, Block RAMs, DSP blocks and IO blocks is done by a set of configuration memory bits called bitstream. According to the size of the FPGA device, the bitstream can contain millions of bits. The memory bits that store the bitstream inside the FPGA is composed of SRAM memory cells, so they are reprogrammable and volatile. When an SEE occurs in the configuration memory bit of an SRAM-based FPGA, it can provoke a bit-flip. This bit-flip can change the configuration of a routing connection or the configuration of a LUT or flip-flop in the CLB. This can lead to catastrophic effects in the designed circuit, since an SEE may change its functionality.

SEE in the configuration memory bits of an SRAM-based FPGA has a persistent effect and it can only be corrected when a new bitstream is loaded to the FPGA [4]. In the combinational logic, the effect of an SEE is related to a persistent fault (zero or one) in one or more configuration bits of a LUT. Figure 1.2 exemplifies an SEU occurrence in a LUT configuration bit and in a bit controlling a routing connection. SEE in the routing architecture can connect or disconnect a wire in the matrix. This is also a persistent effect and its effect can be a modification in the mapped circuit, as a logic change or a short circuit in the combinational logic implemented by the FPGA. It can take a great number of clock cycles before the persistent error is detected and recovery actions are initiated, as the load of a faulty-free bitstream. During this time, the error can propagate to the rest of the system.

Bit-flips can also occur in the flip-flop of the CLB used to implement the user's sequential logic. In this case, the bit-flip has a transient effect and the next load of the flip-flop will correct it.

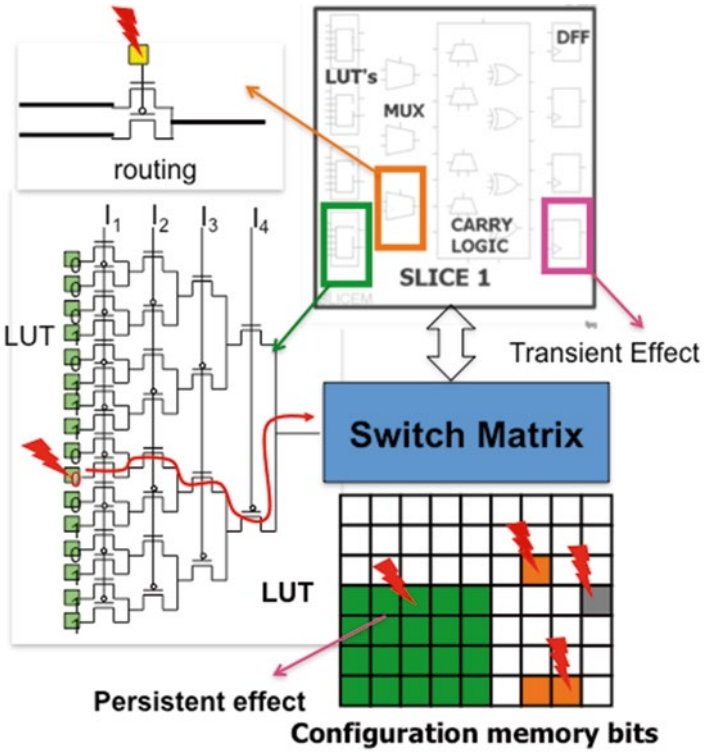


Fig. 1.2 Example of an SEU occurrence in a LUT and in the routing of an SRAM-based FPGA

1.3.2 Single Event Effects on Flash-Based FPGAs

Flash-based FPGAs have a reconfigurable array composed of VersaTiles and routing resources that are programmable by turning ON or OFF switches implemented by floating gate (FG) transistors (NMOS transistor with a stacked gate) [5]. The FG switch circuit is a set of two NMOS transistors: (1) a sense transistor to program the floating gate and sense the current during the threshold voltage measurement and (2) a switch transistor to turn ON or OFF a data-path in the FPGA (Fig. 1.3). The two transistors share the same control gate and floating gate. The threshold voltage is determined by the stored charge in the FG. Figure 1.3 illustrates VersaTiles used to implement some common logic gates. The VersaTiles are connected through a four-level hierarchy of routing resources: ultra-fast local resources; efficient long-line resources; high-speed, very-long-line resources; and the high-performance VersaNet networks.

Each VersaTile can implement any 3-input logic functions, which is functionally equivalent to a 3-inputs Lookup Table (3-LUT). But it is important to highlight that the electrical implementation of the VersaTile is totally different than the electrical

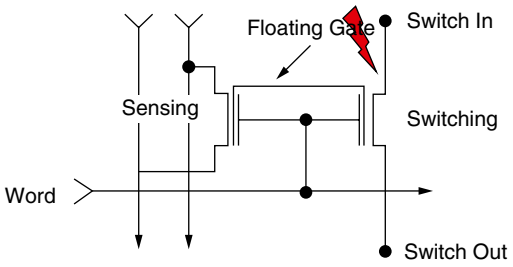


Fig. 1.3 SET in the Flash-based FPGA programmable switch

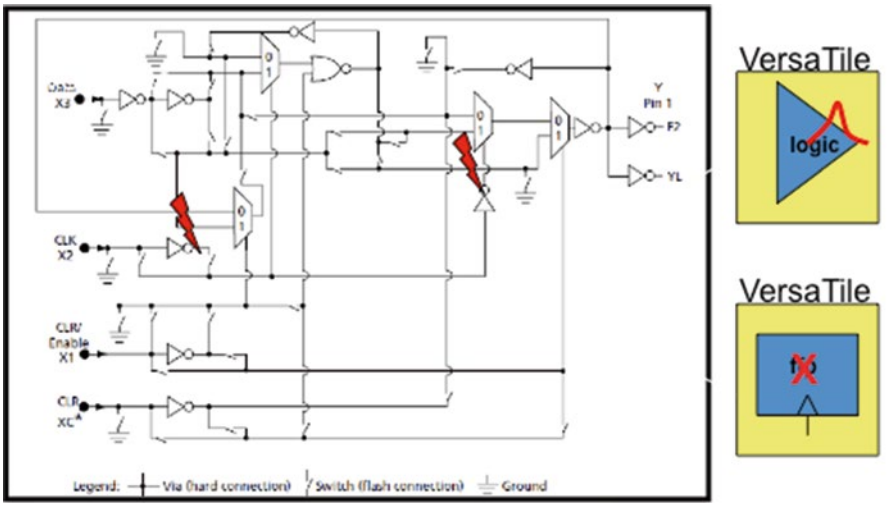


Fig. 1.4 SET and SEU in the Flash-based FPGA VersaTile

implementation of a Lookup Table (LUT). Hence, the VersaTile may have a different electrical behavior to variability effects with respect to a 3-inputs LUT. The VersaTile can also implement a latch with clear and reset, or D flip-flop with clear or reset, or enable D flip-flop with clear and reset by using the logic gate transistors and feedback paths inside the VersaTile block. For each configuration in the VersaTile block, the number of FG switches and transistors in the critical path changes. Single Event Transient (SET) pulses can hit the drain of the transistor at OFF state as presented in Fig. 1.3 provoking a transient pulse in the configuration switches. Or it can hit the sensitive nodes of the transistors in the VersaTile provoking SET or bit-flip according to the customization of the tile (Fig. 1.4). **Chapter 11** is focused on the evaluation of radiation-induced error in 65 nm Flash-Based FPGAs. **Chapter 14** gives an overview of the effects induced by neutrons in Mixed-Signal Flash-based FPGAs.

Table 1.1 Summary of SEU and SET effects in FPGAs

FPGA	SEU/SET in the logic of the configuration basic block	Routing connections	Configurable switches
SRAM-based	persistent	persistent	persistent
Flash-based	transient	no	no
Antifuse-based	transient	no	no

1.3.3 Single Event Effects on Antifuse-Based FPGAs

Antifuse-based FPGAs consists of a regular matrix composed of combinational (C-cells) and sequential (R-cells) surrounding by regular routing channels. All the customizations of the routing and the C-cells and R-cells are done by an antifuse element (programmable switch). Results from radiation ground testing have shown that programmable switches either based on ONO (oxide-nitride-oxide) or MIM (metal- insulator-metal) technology are tolerant to ionization and total dose effect [6]. Therefore, the customizable routing is not sensitive to SEU, only combinational logic and the flip-flops used to implement the design user sequential logic are sensitive to SEE.

Another well known antifuse-based FPGA is from Aeroflex and QuickLogic. Its architecture is composed of a regular matrix of configurable logic cells used to implement the combinational logic and flip-flops, surrounding by a regular routing matrix. Programmable switches called ViaLink connector are used to do all the customizations.

In order to summarize the SEU and SET effects in FPGAs, Table 1.1 shows the susceptible parts of the architectures and classifies the effects as transient or persistent, when it is needed reconfiguration to correct the fault.

1.4 Soft Errors on GPUs

Graphics Processing Units are complex parallel computing systems that dispose of large memory structures as L2 and L1 caches or register files, efficient Arithmetic Logic Units (ALU), and tasks schedulers and dispatchers.

Radiation can produce Single Event Upset as well as Multiple Bit Upset in the memory structures of a GPU. If radiation corrupts a register the process using that register for computation is likely to produce a wrong output. The peculiarity of being parallel makes errors in the caches to be more critical for GPUs than for traditional CPUs. In fact, the L1 cache is shared among all the parallel processes in a Steaming Multiprocessor (SM) while the L1 is shared among all the SMs. So, an error in the L1 cache may, in the worst case, propagate to all the parallel processes assigned to the struck SM. Similarly, an error in the L2 cache may affect all the processes running on the GPU [7].

When the impinging particle hit a logic gate, it may produce a Single Event Transient. As for SEU, the criticality and the overall effect on the output of a SET depends on the struck node. If the SET affects a logic gate inside a single core, the thread assigned to that core for computation will probably produce a single failure in the output. However, if the SET corrupts the parallel processes scheduler or dispatcher, it could affect the computation of several processes, as well as induce an application crash or system hang [8].

To have an exhaustive evaluation of GPU sensitivity is it then not sufficient to measure the radiation sensitivity of the single resources like memories or logic gates. It is also necessary to analyze how those resources are used in computation. To do so, radiation experiments can be performed on a representative set of applications, to have sufficient data to extend to other algorithms. An alternative is to calculate the program Architectural Vulnerability Factor (AVF), i.e. the probability for the corrupted resource to generate an output failure, as done in [9]. **Chapter 20** details the possible radiation effect on GPUs and presents possible way to evaluate GPUs behaviors under radiation.

1.5 Fault Tolerance Techniques

Fault-tolerance is defined as a set of techniques to provide a service capable of fulfilling the system function in spite of (a limited number of) faults. Fault-tolerance on semiconductor devices has been meaningful since upsets were first experienced in space applications several years ago. Since then, the interest in studying fault-tolerant techniques in order to keep integrated circuits (ICs) operational in such hostile environment has increased, driven by all possible applications of radiation tolerant circuits, such as space missions, satellites, high-energy physics experiments and others. Spacecraft systems include a large variety of analog and digital components that are potentially sensitive to radiation and therefore fault-tolerant techniques must be used to ensure reliability.

1.5.1 Resilience Techniques for FPGAs

Different fault tolerance techniques can be applied to FPGAs according to their type of configuration technology, architecture and target operating environment. Techniques can be implemented by the user at hardware description language (HDL) before the design is synthesized into the FPGA. In this book, authors focus on techniques that can be applied by the user at the HDL design.

The main techniques are either based on spatial redundancy or temporal redundancy [10]. Spatial redundancy is based on the replication of n times the original module building n identical redundant modules, where outputs are merged into a majority voter. Usually n is an odd number. The voter decides de correct output by

choosing the majority of the equal output values. The most common case of n-modular redundancy (nMR) is when n is equal to 3, where it is called Triple Modular Redundancy (TMR). In this case, a majority voter is used that is able to vote out 2 out of 3 values that are fault free. The TMR can be implemented in different ways by using large grain TMR, or breaking into small blocks and adding extra voters. There is local TMR when only the flip-flops are triplicated, or global TMR, also known as XTMR, where all the combinational and sequential logic is triplicated. Also Diverse TMR (DTMR) can be used, where each redundant module may present a different architecture implementation.

When dealing with the routing, different techniques can be chosen to increase or decrease fan-out, delay and set of connections, which may have a different impact in the SEE sensitivity. In addition, for those FPGAs programmable by SRAM, reconfiguration is mandatory to correct upsets in the configuration bitstream. The continuously blind full reconfiguration is called scrubbing and it is responsible to fully reconfigure the FPGA by a golden bitstream. Partial reconfiguration can also be used.

For embedded processors, one can use different mitigations based on software redundancy, or processor redundancy like lock-step and recomputation. Software-based fault tolerance techniques exploit information redundancy, control flow analysis and comparisons to detect errors during the program execution. For that purpose, software-based techniques use additional instructions in the code area, either to recompute instructions or to store and to check suitable information in memory elements. In the past years, tools have been implemented to automatically insert such instructions into C or assembly code, reducing significantly the hardening costs.

Time redundancy is based on capturing a value twice or three times in time to vote out a transient fault. The values are shifted by a delay [11]. The idea is to be able to capture 2 out of 3 upset free values to be able to mask the fault.

Each of these techniques can protect SEU or SET, or both, as shown in Table 1.2 and they will be addressed in the chapters of this book.

Very often, System-on-Chip (SoC) implemented in FPGAs use a set of the forehead mentioned mitigation techniques. **Chapters 2 and 3** present a System on Chip (SoC)

Table 1.2 List of mitigation techniques that can be applied by the user in designs targeting FPGAs

Mitigation technique	Abstraction level	SET	SEU
Local TMR	HDL		X
Global TMR or XTMR	HDL	X	X
Large grain TMR	HDL	X	X
Diverse TMR (DTMR)	HDL	X	X
Voter insertion	HDL	X	X
Reliability-oriented place and route algorithm	FPGA Flow	X	X
Temporal redundancy	HDL	X	
Embedded processor redundancy	HDL/software-based	X	X
Scrubbing/partial reconfiguration	System		X

designs using SRAM-based FPGA with embedded processor cores for satellite applications where a set of mitigation techniques is employed. **Chapter 6** details a failure detection, isolation, and recovery framework that takes advantage of the resources available in heterogeneous systems. **Chapter 7** proposes a novel scrubbing strategy for the configuration memory of FPGAs. **Chapter 8** evaluates the power requirements of n-modular redundancy, **Chapter 9** presents a fault-tolerant manager core for dynamic partial reconfiguration in FPGAs. **Chapter 12** proposes the use of C-Slow retiming for safety-critical applications. **Chapter 13** proposes a more efficient implementation of EDAC function in Radiation-Hardened FPGAs. **Chapter 15** presents hardening techniques for embedded processors, while **Chaps. 16 and 19** propose hardening techniques for soft-core processors. **Chapters 17 and 18** study how to reduce the overheads of common hardening solutions for circuits and processors.

1.5.2 Resilience Techniques for GPUs

As GPUs were initially designed to accelerate graphic rendering, the reliability research on GPUs is in its infancy. Most of the available GPUs does not offer any reliability solutions, preferring performances to fault tolerance. Only lately some of the GPUs produced for the High Performance Computing market include Error Correcting Codes in their major memory structures (L1 and L2 caches and internal registers). The available ECC is a Single Error Correction Double Error Detection (SECDED) one. It is then capable of correcting SEU and only detecting MBU. Experimentally, it was measured that about 30 % of the radiation induced failures in modern GPUs memory structures are actually multiple failures. Thanks to memory interleaving (i.e. logic bits belonging to the same word are physically separated), only the 5 % of errors are multiple errors affecting bits in the same word. Moreover, an MBU with more than 2 bits corrupted was never observed experimentally. Thus, the SECDED ECC seems sufficient to guarantee high reliability. Nevertheless, logic resources are computing structures and schedulers are left unprotected and internal flip-flops and queues are not covered by ECC. As a result, the ECC may not guarantee high levels of reliability [12].

Lately, some software-based hardening solutions for parallel codes have been proposed. The basic idea is to try to duplicate the parallel tasks to identify failures or to add coding-encoding procedures to detect and, eventually, correct, failures. Duplication With Comparison (DWC) is extremely easily implemented in a GPU, as the whole programming philosophy of the device is voted to parallelism [12]. Even if DWC seems promising and efficient to detect errors, it introduces a non-negligible computing overhead. As a result, redundancy may be non applicable to HPC applications or embedded systems with strict power consumption constraints. It is also essential to duplicate wisely the parallel processes, avoiding threads belonging to the same domain to be executed on the same Streaming Multiprocessor, as they would share the same cache. An error in a shared location will then propagate to both copies and remain undetected. Another hardening philosophy applied to parallel codes is the Algorithm Based Fault Tolerance (ABFT) one. ABFT is based

on the encoding of input data, the modification of the algorithm to be executed on coded data and, finally, the decoding of the output with error detection and correction. ABFT is algorithm-specific, and requires great algorithm analysis and code implementations efforts to be implemented. At the moment, the only algorithms for which an ABFT strategy is available are matrix multiplication and Fast Fourier Transform [7, 13]. **Chapter 20** provides an overview of the available hardening strategies to apply to modern parallel processors.

1.6 Characterizing FPGAs and GPUs Radiation Sensitivity

1.6.1 Fault Injection

In FPGAs, one very important step of the design flow is the validation of the fault tolerance technique that is usually done by fault injection. The original bitstream configured into the FPGA can be modified by a circuit or a tool in the computer by flipping one of the bits of bitstream, one at a time. This flip emulates a SEU in the configuration memory cells. The output of the design under test (DUT) can be constantly monitored to analyze the effect of the injected fault into the design. If an error is detected, this means that the fault tolerant technique implemented is not robust for that specific fault (SEU) in that target configuration memory bit.

It is possible to inject faults in all the configuration bits and to analyze the most critical parts of the design [14]. This can help to guide designers in early stages of the development process to choose the most appropriated fault tolerant design, even before any radiation ground testing. The entire fault injection campaign can spend from few hours to days depending on the amount of bits that are going to be flipped and the connection to the fault injection control circuit. When the entire system (fault injection control+DUT+golden designs) is implemented at the hardware level (board), avoiding the communication with the computer, the process is speeded up in orders of magnitude.

Chapters 4 and 5 present some techniques for fault injection in SRAM-based FPGAs. **Chapter 10** presents a fault injection framework that reproduce multiple and accumulation of upsets collected from real radiation experiments.

However, fault injection on GPUs has several limitations. Only few resources of the GPU are accessible by the user and to access those resources to inject fault it is necessary to change the flow of the algorithm, introducing artificial behavior. There is one fault injector for GPU available, the GPU-Qin [15], which allows the user to insert faults only on instantiated values.

1.6.2 Radiation Test Methodologies to Predict and Measure SER in FPGAs and GPUs

The test of FPGAs under radiation depends on a test plan developed for each type of FPGA and design architecture. Here we will detail the radiation test for SRAM-based FPGAs. There are two types of tests: the static test and the dynamic test. The static test can be done in SRAM-based FPGAs for instance, where the experiment

consists on configuring the FPGA with a *golden* bitstream containing the test-design and then constantly read back the FPGA configuration memory with the Xilinx iMPACT tool through the JTAG interface. In the experiment control computer, the *golden* bitstream is compared against the *readback* bitstream. If differences are found, the FPGA is reconfigured with the *golden* bitstream and the differences are stored in the computer. Faults are defined as any bit-flip in the configuration memory detected by the *readback* procedure. In this case, it is possible to calculate the upset rate in the configuration memory bits for that specific particle flux.

The cross-section per bit shows the sensitive area of a device and it is used to compare radiation sensitivity between devices. It is calculated as defined in Eq. 1.1.

$$\sigma_{SEU-bit} = \frac{N_{SEU}}{\Phi_{neutron} \times N_{bits}} \tag{1.1}$$

Where N_{SEU} is the number of SEU in the configuration memory bits, $\Phi_{neutron}$ is the neutron fluence and N_{bits} is the number of bits of the device. The fluence is measured by neutron per cm^2, and it is calculated by multiplying the neutron flux by the time the device has been exposed to that flux.

The dynamic test analyzes the design output mapped into the FPGA. In this case, the expected error rate is much lower than the static test. In case of SRAM-based FPGAs, based on the Xilinx Reliability Report [16], in average it is necessary 20 upsets in the configuration memory bits to provoke one error in the design output. This relation may of course vary according to the logic density, mapping, routing and the chosen architecture for the design. In case of using redundancy such as TMR or n-MR, the number of accumulated upsets in the bitstream without provoking functional error can increase significantly. In case of Flash-based FPGAs and antifuse based FPGAs, the soft error rate comes from the susceptibility of the configurable logic to the SET and SEU (bit-flips) only as the programmable cells (antifuse and flash cells) are normally not susceptible to transient upsets.

The static test of GPUs follows the same philosophy as the FPGA one. Basically a known pattern is loaded into the main memory structures of the device and then read back. There is not a special port to access the memory structures, so the test should be engineered to take advantage of normal GPU processes to write the pattern and read it back. The dynamic test of a GPU requires the selection of proper benchmarks to run on the device. It is worth noting that for being useful the benchmark must be representative of a given workload of application. Otherwise results would be valid only for the particular configuration tested. Normally the benchmark is executed with a pre selected input vector and results are checked with a pre computed golden copy of the output. When a mismatch is detected, it should be counted as an error. To evaluate the cross section it is necessary to evaluate the fluence hitting the device only when the code is being executed, and not during results check. Alternatively, one can calculate the cross section diving the observed error rate (errors/s) by the average flux provided by the facility during the test (particles/(cm^2 s)).

There are only few facilities in the world that provides good fluxes and spectrum of energies to ease the scale of experimental result to the expected natural error rate. Examples of neutron facilities are LANSCE, in Los Alamos, NM, USA, TSL, Uppsala, Sweden, TRIUMF, Vancouver, Canada, and ISIS, Didcot, UK.

In this book results were gathered from experiments performed at Los Alamos National Laboratory's (LANL) Los Alamos Neutron Science Center (LANSCE) Irradiation of Chips and Electronics House II and in the VESUVIO beam line in ISIS, Rutherford Appleton Laboratories, Didcot, UK. As shown in [17], both of these facilities provide a white neutron source that emulates the energy spectrum of the atmospheric neutron flux. The ISIS spectrum has a lower component of high-energy neutrons with respect to the LANSCE and the terrestrial one. The relationship between neutron energy and modern devices cross section is still an open question. Nevertheless, ISIS beam has been empirically demonstrated to be suitable to mimic the LANSCE one and the terrestrial radiation environment [17].

Figures 1.5 and 1.6 show the setup of experiments under neutron at ISIS Facility in United Kingdom and Los Alamos, respectively, composed of many different types of FPGAs and GPU performed in parallel.

Fig. 1.5 Neutron experiment Setup in ISIS for FPGAs and GPUs

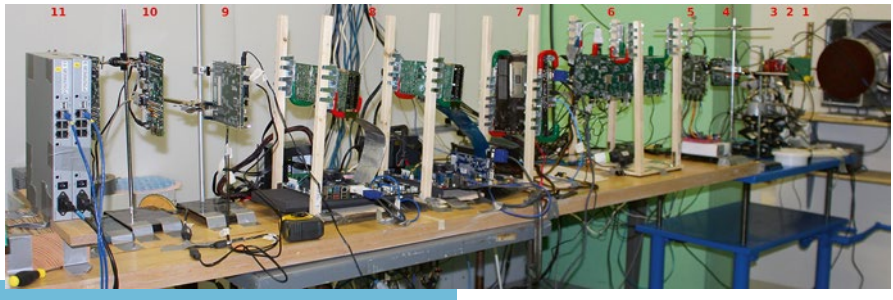


Fig. 1.6 Neutron experiment Setup in Los Alamos for FPGAs and GPUs

References

1. Nicolaidis M (2011) Soft errors in modern electronic systems. Springer, New York, p 318
2. Stassinopolous EG, Raymond JP (1988) The space radiation environment for electronics. Proc IEEE 76:1423–1442
3. Dodd PE, Massengill LW (2003) Basic mechanisms and modeling of single-event upset in digital microelectronics. IEEE Trans Nucl Sci 50(3):583–602
4. Kastensmidt FL, Reis R, Carro L (2006) Fault-tolerance techniques for SRAM-based FPGAs (frontiers in electronic testing). Springer, New York
5. Microsemi. ProASIC3, IGLOO and SmartFusion flash family FPGAs datasheet. www.microsemi.com
6. Rezgui S, Louris P, Sharmin R (2010) SEE characterization of the new RTAX-DSP (RTAX-D) antifuse-based FPGA. IEEE Trans Nucl Sci 57(6):3537–3546
7. Rech P, Aguiar C, Frost C, Carro L (2013) An efficient and experimentally tuned software-based hardening strategy for matrix multiplication on GPUs. IEEE Trans Nucl Sci 60(4):2797–2804
8. Rech P, Pilla L, Navaux POA, Carro L (2014) Impact of GPUs parallelism management on safety-critical and HPC applications reliability. In: Proceeding IEEE international conference on dependable systems and networks (DSN), June 2014, pp 455–466
9. Mukherjee SS, Emer J, Reinhardt SK (2005) The soft error problem: an architectural perspective. In: High-performance computer architecture, 2005. HPCA-11. 11th international symposium on, 12–16 Feb 2005, pp 243–247
10. Schrimpf RD, Fleetwood DM (2004) Radiation effects and soft errors in integrated circuits and electronic devices. Word Scientific, Singapore
11. Anghel L, Alexandrescu D, Nicolaidis M (2000) Evaluation of a soft error tolerance technique based on time and/or space redundancy. In: The Proceedings of symposium on integrated circuits and systems design, SBCCI, 13, pp 237–242
12. Oliveira DAG, Rech P, Pilla LL, Navaux POA, Carro L (2014) GPGPUs ECC efficiency and efficacy. In: International symposium on defect and fault tolerance in VLSI and nanotechnology systems
13. Pilla LL, Rech P, Silvestri F, Frost C, Navaux POA, Sonza Reorda M, Carro L (2014) Software-based hardening strategies for neutron sensitive FFT algorithms on GPUs. IEEE Trans Nucl Sci 61(4):1874–1880
14. Sterpone L, Violante M (2007) A new partial reconfiguration-based fault-injection system to evaluate SEU effects in SRAM-based FPGAs. IEEE Trans Nucl Sci 54(4):965–970
15. Fang B, Pattabiraman K, Ripeanu M, Gurumurthi S (2014) GPU-Qin: A methodology for evaluating the error resilience of GPGPU applications. In: Proceedings of the IEEE international symposium on performance analysis of systems and software (ISPASS)
16. Xilinx, Inc. (2013) Device reliability report third quarter 2013. http://www.xilinx.com/support/documentation/user_guides/ug116.pdf
17. Violante M, Sterpone L, Manuzzato A, Gerardin S, Rech P, Bagatin M, Paccagnella A, Andreani C, Gorini G, Pietropaolo A, Cargarilli G, Pontarelli S, Frost C (2007) A new hardware/software platform and a new 1/E neutron source for soft error studies: testing FPGAs at the ISIS facility. IEEE Trans Nucl Sci 54(4):1184–1189

Part II
Applications

Chapter 2
Brazilian Nano-satellite with Reconfigurable SOC GNSS Receiver Tracking Capability

Glauberto L.A. Albuquerque, Manoel J.M. Carvalho, and Carlos Valderrama

Abstract This paper presents a flexible architecture for a GPS receiver using Partial Reconfiguration (PR) on a System on Chip (SoC) device consisting on an FPGA and two ARM cores. With built-in error-correction techniques offered by modern SOCs, this device meets the requirements of a Brazilian nanosatellite for CONASAT constellation. This receiver benefits from PR, thereby increasing system performance, hardware sharing, and power consumption optimization, among others. Additionally, all the advantages favor in-orbit reconfiguration. The proposed architecture, as requested, uses COTS components.

2.1 Introduction

CubeSats became an affordable alternative for space missions of emerging countries [1] and even for developed ones. Indeed, the CubeSat specification makes possible to decrease launching costs and development time of small satellites. This specification, which began in 1999 from collaboration between the California Polytechnic State University and the Stanford University, has helped universities around the world developing science and space exploration. Although CubeSats were primarily intended for use with educational purposes, nowadays there are commercial, military and interplanetary space missions using this technology, as a valuable alternative for many space mission profiles [2–4].

G.L.A. Albuquerque (✉)
Barreira do Inferno Launch Center—CLBI, Parnamirim, Brazil
e-mail: glauberto@engineer.com

M.J.M. Carvalho
CRN—Centro Regional do Nordeste, Instituto Nacional de Pesquisas Espaciais—INPE, Natal, Brazil
e-mail: manoel@crn.inpe.br

C. Valderrama
SEMi—Electronics and Microelectronics Department, University of Mons, Mons, Belgium
e-mail: carlos.valderrama@umons.ac

F. Kastensmidt, P. Rech (eds.), *FPGAs and Parallel Architectures for Aerospace Applications*, DOI 10.1007/978-3-319-14352-1_2

Advances in electronics and MEMS combined with techniques such as Software Defined Radio (SDR) and Digital Signal Processing (DSP) have contributed to reduce costs while facilitating their development. In particular, Field Programmable Gate Arrays (FPGAs) has proven to be a cost effective tool for the development of projects in different areas beyond SDR. In addition to the reconfiguration flexibility, its main advantage over other devices is their low power consumption [5]. Indeed, this is a very important attribute for space applications. In orbit, a satellite can easily get energy from solar panels and batteries, but at the cost of adding extra weight to the structure. Thus, to reduce the total volume, satellites must be designed from devices with reduced size and low power consumption.

Apart from specificities of each space mission profile, all satellite payload contain some kind of communication link and navigation control, for which Global Navigation Satellite System (GNSS) receivers are envisioned nowadays. Additionally, such sub-systems must be robust and reliable to operate in hostile environments without failure. Regarding this concern, the Partial Dynamic Reconfiguration (PDR) capability of FPGAs could be an additional attribute for space applications [6, 7]. This procedure, not only allows adaptable payload in orbit, but also offers a certain degree of radiation tolerance (e.g. faulty system re-initialization, replacement and upgrade).

This paper proposes a low cost GPS receiver architecture based on FPGA SoC COTS to meet the requirements of CONASAT satellites. This receiver intends to take advantage of modern FPGA-based SoC and Partial Reconfiguration techniques for use in space applications and mission recovering.

2.2 CONASAT

2.2.1 CONASAT Project

CONASAT is a project based on a nanosatellites constellation funded by INPE, Brazil's National Institute for Space Research. Its main mission is to collect environmental data from thousands of DCPs (Data Collection Platforms) distributed throughout the Brazilian territory and its seacoast. This constellation will replace the former SCD1 and SCD2 satellites, still active, although they have already exceeded their design life.

Some relevant guidelines concerning the CONASAT project are [8]:

- To develop expertise in the field of space missions, especially on nano-satellites;
- It must satisfy the lowest possible cost for an acceptable level of reliability and mid-term life-time of 5 years;
- It must use COTS components and commercial subsystems as much as possible;
- It must provide such a flexible and modular platform that could be adopted by subsequent generations of satellites of the constellation;

- CONASAT satellites must be CubeSat compliant;
- It must generate opportunities for Brazilian technology industry.

CONASAT will be the spatial segment of the Brazilian System for Environmental Data Collection (SBCDA). Brazil already produces its own DCPs and some parts of a CONASAT satellite. As much as possible, other parts of the satellite should be produced by Brazilian experts. For instance, the current communication protocol between DCPs and satellites will be modified to allow bidirectional data exchange.

CONASAT satellites will use Low Earth Orbits (LEO—altitudes from 500 to 800 km). Thus, satellites will not be over the Brazilian territory all the time. Downtime will then be occupied by other applications or services. For instance, it is planned to extend SBCDA services for monitoring fishing boats. In these cases, it's desirable to have CONASAT parts implemented on reconfigurable hardware supporting tasks on demand. Regarding radiation tolerance, it is important to note that the satellite orbit, at an altitude of about 600 km, belongs to a region with low ions density.

2.2.2 *The CONASAT Satellite Architecture*

Generic architecture of the satellite, shown in Fig. 2.1, is not remarkable compared to others. It consists on a full redundancy of all major subsystems, including the *Power Management* one. Thus, it can be considered as having two satellites within one mechanical infrastructure. This choice intends to increase overall system reliability due to the fact that the design guidelines of CONASAT allow the use of COTS components. Another reason is the MTBF (Mean Time Between Failures) of CubeSat parts readily available on the market. They are not prepared for a midterm lifetime.

The *Redundancy Control* subsystem decides which sub-system to activate each time. The *Attitude Control* subsystem includes a magnetorquer (iMTQ), stellar gyroscope, 3-Axis gyroscope, star tracker and reaction wheels. This satellite also uses a GNSS Receiver (GPS receiver, in this case) to simplify orbital prediction. The use of multiple sensors obeys to the principle of achieving maximal reliability. However, while the combined use of sensors increases its efficiency. On the other side it also raises the weight of the satellite and its power consumption. Moreover,

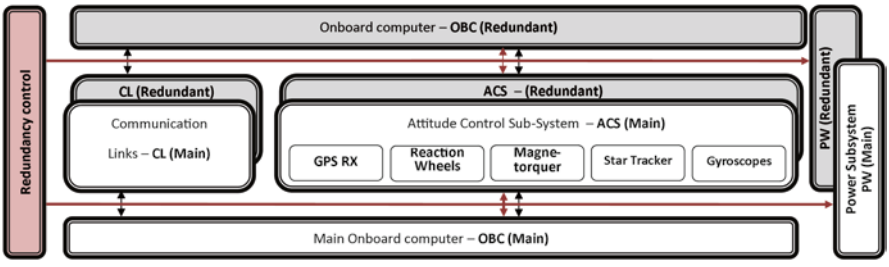


Fig. 2.1 CONASAT functional architecture (adapted from [8])

the processing capability of the GPS receiver must be adapted to the orbital velocities. Therefore, the way of space GPS receivers handle data must be carefully adapted. The *Communication Subsystem* is just composed by an UHF uplink and S-Band downlink. It is responsible to retransmit to ground stations data received from DCPs. The *Power* and *Attitude Control* subsystems have in-orbit so specialized tasks which cannot take other responsibilities.

The GPS Receiver is the only subsystem whose functionality should be modified in orbit, on demand, to accomplish a particularly required task. For that reason, this receiver must be built based on a software platform. Moreover, due to the requirements of performance and power, this flexibility must be supported by reconfigurable hardware. However, there is no such a device on the market, an "on-orbit reconfigurable GPS receiver for Cubesats". With an optimal choice of the FPGA device, unused logic elements could provide added functionality or even, when the receiver is idle, could also be possible to share the entire platform. This would reduce the physical size and the number of electronic devices, with favorable effects on energy consumption and the satellite's overall weight.

2.3 Software GNSS Receivers Architecture

As we saw above, the software-based approach for a GNSS receiver was a natural choice in terms of design, especially because, in the case of a GPS, signals from the GPS satellites constellation use digital modulation (BPSK). Taking this into consideration, the assembly of a GPS Receiver (or other GNSS System), despite some difficulties, is not an unattainable task [9]. Because of the dominance of GPS in this domain, the remainder of this paper will consider the GPS as a reference to explain the proposed architecture.

According to the chipset used in the design we can identify two approaches: hardware or software receivers. Hardware receivers use ASIC devices to accomplish all tracking and navigation tasks. Those commercially available have limited or no applicability in aeronautics or spatial domain. In software receivers, signal processing tasks are programmable, by using a GPP (General Purpose Processor), DSP, GPU, or even reconfigurable hardware (FPGA). Sometimes, developers work with a combination of these devices [10–12].

We can see the GPS receiver basic architecture in Fig. 2.2. Although the different types of GNSS receivers available are tailored to the different target applications, all these basic architectures include the same functional blocks.

After the *Antenna*, required to amplify and filter the incoming radio signal, the *Front-End* is responsible for down-conversion and digitalization of this analog signal. The *Baseband Processing* block acquires and monitors each incoming signal to calculate its own position and speed. For each tracked satellite it is required to have one of these blocks. Thus, it extracts observable and navigation data from each processed channel. Theoretically, up to 12 GPS satellites can be tracked at the same time, but to calculate its position the receiver only needs four of them. After correctly tracking the signals, the measurement data obtained are sent to the *Application Processing* block.

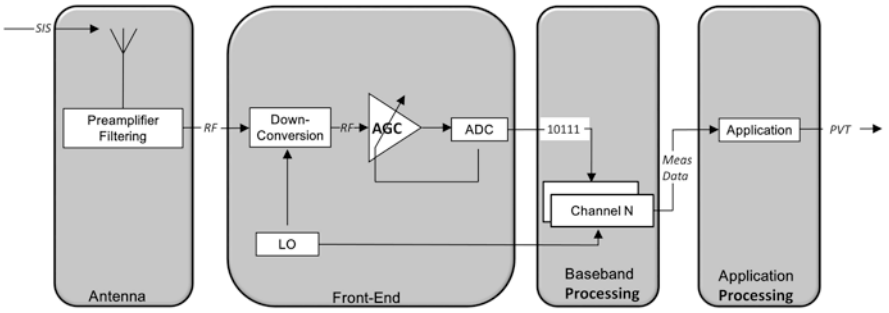


Fig. 2.2 Generic GPS receiver architecture [13]

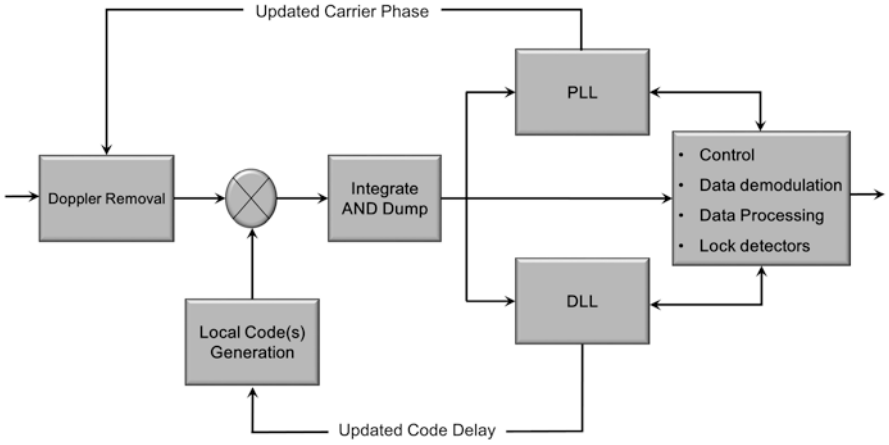


Fig. 2.3 Baseband signal processing [13]

This block uses the information from the tracking loops for different purposes. Typical applications are: ionosphere parameters monitoring, DGPS (Differential Global Positioning System) calculation, static and kinematic surveying.

The processing time of the *Baseband Processing* determines two categories of receivers: real-time and post processing. In post processing, the baseband information is used to obtain correlations between the incoming signals and an internal replica, used as reference. This produces intermediate data stored to be further processed in batch mode by complementary algorithms. Thus, the receiver is not able to locate the position in real-time. That delay is critical for orbital speed navigation, implying additional power processing and control over tracking algorithms.

Baseband Processing includes all the algorithms to find and follow a visible GPS signal, through the synchronization with a known PRN code, and remove errors, as best as possible. This process is built around the principle of signal correlation: the incoming signal is repeatedly correlated with a replica of the expected PRN code, which is known a priori. Its functional structure is depicted in Fig. 2.3. To extract a valid significance from the correlation, the local replica is generated in the receiver

taking into account the signal carrier phase, code delay, Doppler frequency, and PRN code [12]. To obtain maximum correlation, the *DLL* and *PLL* blocks are in charge of follow the code and carrier delay, respectively.

2.4 Hardware Design

2.4.1 The Front-End

Even for software GNSS receivers, most of front-end modules are ASIC devices. On the market there are dozens of options, even a reconfigurable alternative has recently emerged [14]. Brazilian scientists have used the GP2000 chipset to build a GPS receiver for sounding rockets [15]. Moreover, as demonstrated in [16], the GP2000 chipset is sufficiently radiation-proof for use in LEO without major modifications. However, many other GPS receivers for space applications are based on the GP2015 front-end, for instance, those produced by DLR and Surrey Technologies [17, 18]. So, the GP2015 family can be considered as a certified choice.

2.4.2 Baseband Processing Module

Although the GP2015 front-end module is a good choice for this receiver, the use of the other chips of the family will lead us to a hardware receiver; losing all the advantages of the software approach in terms of algorithm flexibility and associated data processing efficiency.

The GP2015 front-end at a sampling frequency of 5.71 MHz provides 2-bit samples. The bandwidth required by the sample data rate is:

$$f_s = 5.71\,\text{Msamples/s} \tag{2.1}$$

$$\text{N}_{\text{Samples}} = 2\,\text{bits}(\text{sign/magnitude}) \tag{2.2}$$

$$\text{BW} = f_s * \text{N}_{\text{Samples}} = 11.42\,\text{Mbps} \tag{2.3}$$

This bandwidth can be easily achieved with modern FPGA transceivers of up to 1 Gbps and, if necessary (e.g. Doppler removal) incoming data can be oversampled.

A generic tracking channel is depicted in Fig. 2.4. This channel, composed of *accumulators* and *carrier/code generation* units, requires around 1.5 k logic elements on a single FPGA [9]. Remaining modules, acquisition and tracking loops, will take 3 and 6 k logic elements, respectively. Since most of operations are binary, random SEU have not major influence on the final correlation.

Modern FPGAs can provide more that 100k logic elements. This is enough to contain a GPS receiver with ten parallel baseband signal processing units. This can be extended by introducing pipeline techniques to share single tracking channels.

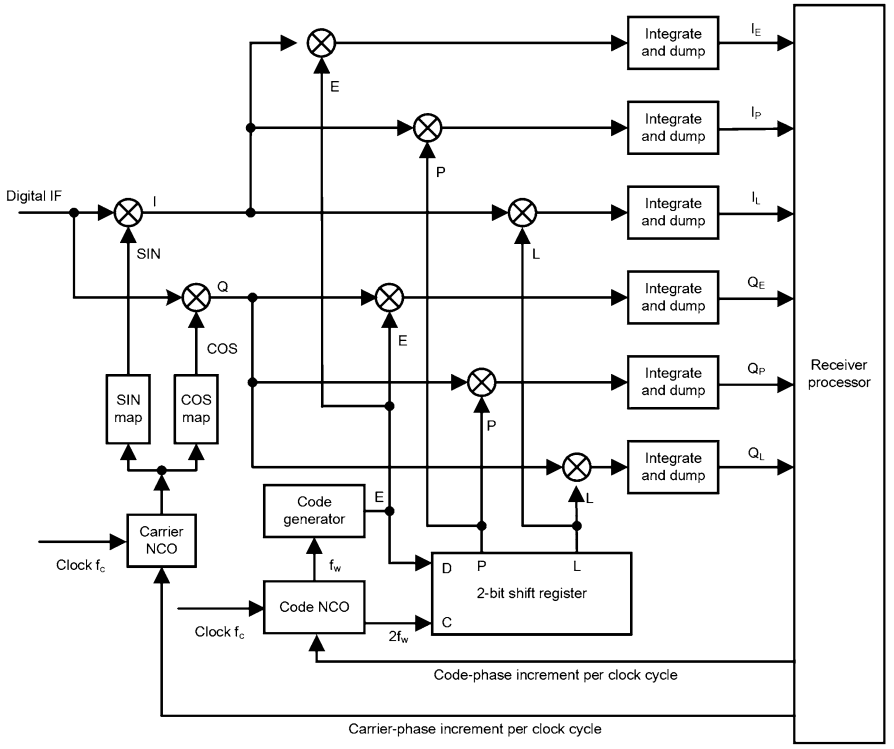


Fig. 2.4 Generic digital receiver channel block diagram

For instance, operating at 200 MHz with an 11.42 MHz sample clock, a given channel can track up to 16 GPS satellites at a time. However, CONASAT imposes orbital velocities, thus parallel tracking channels are better suited.

In the case that power consumption is not a constraint, GPPs, DSPs and GPUs, have enough power processing to build real-time receivers. However, when looking for balance of power processing and low power consumption, FPGA are a better choice. If necessary, additional tracking channels may even become available on demand by using the DPR technique (Dynamic Partial Reconfiguration). Moreover, DPR alone can also be used to mitigate SEU, as in [7, 19, 20], or even combined with TMR as in [21, 22]. As will be shown later, those alternatives have also been considered to meet the requirements of our proposal.

2.4.3 Application Processing Module

Application tasks must be quickly created to support the specifics of a particular mission. This adaptability is a key requirement to ensure the multiplicity of application cases and the sustainability of such a platform. ARM microprocessors appears

as a software processing module in different commercial GPS receivers [17, 18] with the added value of Linux OS. In FPGA there are also softcores like, for instance, NIOS, but not powerful enough for additional tasks. There is also the hardened version of the LEON processor. However, modern SOC FPGAs provide dual core ARM processors on the same package and the possibility to apply some fault mitigation and correction techniques such as in [23]. Although, to assure reliability of the overall system, some radiation hardened devices must still be used. This requirement particularly applies to the memory device, which must keep protected critical data for both, the FPGA and the processors. In addition, preserved application software or reconfiguration data are used when needed or to replace faulty modules.

2.4.4 SEU Mitigation in COTS FPGA and SOC

Radiation hardened devices, combined with Single Event Upset (SEU) error mitigation and CRC, is an important requirement not always supported by FPGAs. Looking at the market of new devices, we found modern ones with built-in SEU error mitigation based on CRC method. This on-chip error detection performs the following operations without any impact on the fitting performance of the device [23]:

- Auto-detection of CRC errors;
- Optional CRC error and identification in user mode;
- Testing of error detection functions by deliberately injecting errors through the JTAG interface.

At the same family of chip there is a SOC device. This device includes high speed transceivers and dual core ARM processors.

Apart of internal mitigation of SEUs, aluminum shielded is included in CONASAT design. According to [24] a 1 mm thick aluminum box absorbs approximately 6000 rad.

2.4.5 Proposed Architecture

As we can see in the Fig. 2.5, the architecture is designed to take advantage of all built-in circuits and Partial Reconfiguration in order to achieve a reliable receiver to be used in spatial applications. This architecture is better than the proposed in [25] in terms of power consumption. Literature survey has showed that high-end FPGAs have a huge throughput advantage over high performance DSP processors for certain types of signal processing applications. FPGAs use highly flexible architectures which can be of greatest advantage over regular DSP processors [26].

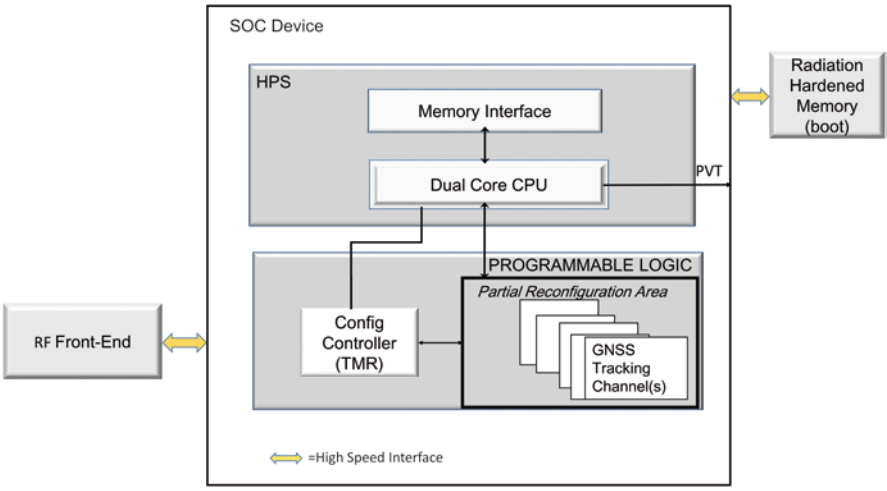


Fig. 2.5 Proposed architecture

The one-chip architecture also take advantages in terms of radiation protection since the area of silicon components are obviously smaller than any other architecture with two or more devices.

The overall architecture is seen in Fig. 2.5. The *Config Controller* is responsible to verify all parts of the algorithm are working correctly. It is also responsible for the FPGA reconfiguration, error recovering or to change the application. After critical errors not recovered by the built-in *CRC control*, the *Config Controller* is able to restart the receiver. To improve reliability of the overall system this part of software is designed using the TMR technique. The two ARM cores in the SOC so the system (HPS block) could take advantages of the dual CPU fault tolerance techniques [27]. Critical parts of the software code are stored in a radiation hardened memory.

2.4.6 Improving Cold Start Time

The *Doppler Removal* module we see in Fig. 2.3 is responsible to correct inaccuracies in the apparent Doppler frequency of the satellite and "zero-beat" the signal. A Doppler shift is the change in frequency of a wave (or other periodic event) for an observer moving relative to its source. If we take the relative motion between the GPS satellite, with orbital speed of 3.9 km/s, and a car, assuming at 40 m/s (150 km/h) traveling over the Equator (greatest Earth rotational speed: about 460 m/s) we could reach, at a maximum, 1.3 km/s, which is equivalent to a Doppler shift of ±6.8 kHz. If we replace the car by a LEO satellite, with orbital speed up to 9 km/s, this generates a significant Doppler frequency shift amounting to ±45 kHz.

On Cold Start mode, when no prior information about Doppler shift, the incoming signal is first stripped of its Doppler frequency, and then correlated with one (or more) PRN code replicas generated locally (according to the current estimation of code delay). When the receiver does not have a good estimation of the initial Doppler, the receiver must correlate the signal with a range of all possible Doppler shifts. Once all Doppler and code shifts have been composed, the peak magnitude is compared to a predefined carrier-to-noise threshold to determine if a GPS satellite has been located. This method consumes fewer hardware resources, but increases the cold start time.

On the ground, a GPS receiver can see a given satellite for several hours. In space applications the visibility time is, in most cases, less than 50 min. Besides that relative motion speed between each GPS satellite and CONASAT changes very quickly, so the receiver must improve the cold start time in order fix a navigational solution.

In this architecture each GPS channel is responsible to track a specific PRN code. Once an entire PRN code is transmitted in 1 ms, the accumulation period is typically between 1 and 20 ms. With a sample data of 5.71 MHz and, for instance, a clock system of 400 MHz, we could make about 70 times the correlation with the same data. In each time slice the generated code is created with different Doppler shifts. With this strategy, the time to track the first GPS satellite signal decreases to some milliseconds.

PR is a useful technique to implement this architecture because after Cold Start all unnecessary FPGA's resources could be released to another application. PR also allows to create an optimal Sleeping Mode, when the CONASAT has no visibility over Brazilian territory and only critical data and applications must be preserved. The receiver could benefits from PR in other phases of the receiver operation since some parts of the hardware resources could run specialized algorithms under certain conditions and thus, this resource can be released when becomes not needed anymore.

2.5 Market Options

Looking at the market of GNSS spaceborne receivers most of available devices have a mass of some kilograms and power consumption of tens of watts. These receivers are not suitable for nanosatellites. Some are constructed with COTS components and can be used in space missions within a low radiation orbit. In [28], we can find a detailed list of spaceborne receivers available on the market. This list was published in 2008 but currently it has no significant changes because performances of new products are very similar to old ones. Other alternatives are the dedicated chips used in Cubesat products. However, some experiences with such miniaturized ASCI receivers fail to provide valid navigation fixes [29, 30]. None of these receivers in the market could be reconfigurable in-orbit to perform a completely different application.

2.6 Conclusions

With the proposed architecture, CONASAT could take advantage of COTS components in order to accelerate design process and decrease costs. This device, using PR presents high level of adaptability. This electronic framework could be used to develop other applications under SDR techniques. One of natural improvement to this receiver is include GALILEO tracking channels.

References

1. Woellert K, Ehrenreund P, Ricco AJ, Hertzfeld H (2010) Cubesats: cost-effective science and technology platforms for emerging and developing nations. Adv Space Res 47:678–679
2. Taraba M et al (2009) Boeing's CubeSat TestBed 1 attitude determination design and on-orbit experience. In: Proceedings of the 23rd annual AIAA/USU conference on small satellites
3. Weeks D, Marley AB, London III J (2009) SMDC-ONE: an army nanosatellite technology demonstration. In: Proceedings of the 23rd annual AIAA/USU conference on small satellites
4. Staehle RL, Anderson B, Betts B, Blaney D, Chow C, Friedman L, Hemmati H, Jones D, Klesh A, Liewer P, Lazio J, Lo M, Mouroulis P, Murphy N, Pingree PJ, Puig-Suari J, Svitek T, Williams A, Wilson T (2012) Interplanetary CubeSats: opening the solar system to a broad community at lower cost. In: Final report on NIAC phase 1 to NASA Office of the Chief Technologist, Jet Propulsion Laboratory, 2012. (Submitted to Journal of Small Satellites. http://www.nasa.gov/pdf/716078main_Staehle_2011_PhI_CubeSat.pdf)
5. Choi S et al (2003) Energy-efficient signal processing using FPGAs. In: Proceedings of the 2003 ACM/SIGDA eleventh international symposium on field programmable gate arrays. ACM
6. Savani VG, Mecwan AI, Gajjar NP (2011) Dynamic partial reconfiguration of FPGA for SEU mitigation and area efficiency. Int J Adv Technol 2(2):285–291
7. Zhang J, Guan Y, Mao C (2013) Optimal partial reconfiguration for permanent fault recovery on SRAM-based FPGAs in space mission. Adv Mech Eng
8. INPE (2011) Constelação de nano satélites para coleta de dados ambientais: documento de descrição da missão DDM. http://www.crn2.inpe.br/conasat1/Documentos/gerais/Documento%20de%20Descri%E7%E3o%20da%20Miss%E3o%20%28Equipe%20CONASAT%29.pdf
9. Shapiro AM (2010) FPGA-based real-time GPS receiver. Dissertations, Cornell University
10. Hobiger T et al (2010) A GPU based real-time GPS software receiver. GPS Sol 14(2): 207–216
11. O'Hanlon B et al (2011) CASES: a smart, compact GPS software receiver for space weather monitoring. In: Proceedings of the ION GNSS meeting
12. Dovis F et al (2001) On the tracking performance of a Galileo/GPS receiver based on hybrid FPGA/DSP board. In: Proceedings of the 18th international technical meeting of the satellite division of institute of navigation (ION GNSS 2005)
13. ESA Navipedia (2014) Generic receiver description. http://www.navipedia.net/index.php/Generic_Receiver_Description#Receiver_overview. Accessed 22 June 2014
14. Noroozi A (2013) A reconfigurable GPS/Galileo receiver front-end for space applications. Dissertations and Theses, Delft University of Technology, Netherlands. Web. 12 June 2014
15. Francisco M, Albuquerque G, Rapôso T (2011) A GPS receiver for use in sounding rockets. In: 20th symposium on European rocket and balloon programmes and related research, vol 700
16. Unwin MJ, Oldfield MK, Underwood CI, Harboe-Sorensen R (1998) In: Proceedings of the 11th international technical meeting of the satellite division of the Institute of Navigation (ION-GPS-1998), Nashville, 15–18 Sep 1998, pp 1983–198

17. Markgraf M et al (2001) A low cost GPS system for real-time tracking of sounding rockets. European space agency-publications-ESA SP 471, pp 495–502
18. Underwood C et al (2004) Radiation testing campaign for a new miniaturised space GPS receiver. In: IEEE radiation effects data workshop, July 22, Atlanta, USA, pp 120–124
19. Graczyk R et al (2012) Dynamic partial FPGA reconfiguration in space applications. In: Photonics applications in astronomy, communications, industry, and high-energy physics experiments 2012. International Society for Optics and Photonics
20. Jan K, Straka M, Kotasek Z (2012) Methodology for increasing reliability of FPGA design via partial reconfiguration. In: The first workshop on manufacturable and dependable multicore architectures at nanoscale (MEDIAN'12), Annecy
21. Azambuja JR, Sousa F, Rosa L, Kastensmidt FL (2009) Evaluating large grain TMR and selective partial reconfiguration for soft error mitigation in SRAM-based FPGAs. In: On-line testing symposium, IOLTS 2009, pp 101–106
22. Pilotto C, Azambuja JR, Kastensmidt FL (2008) Synchronizing triple modular redundant designs in dynamic partial reconguration applications. In: SBCCI '08: Proceedings of the 21st annual symposium on integrated circuits and system design. ACM, New York, pp 199–204
23. Altera Corporation (2012) SEU mitigation for cyclone V devices. http://www.altera.com/literature/hb/cyclone-v/cv_52008.pdf. Accessed 2 May 2014
24. Wertz JR, Larson WJ (1999) Space mission analysis and design. Kluwer Academic, Dordrecht
25. Bedmutha ND, Biraris PN, Shah JP (2013) A low cost GNSS software receiver design with SEE mitigation approach for microsatellites. In: Space science and communication (IconSpace), 2013 IEEE International Conference on IEEE
26. Hayim A, Knieser M, Rizkalla M (2010) DSPs/FPGAs comparative study for power consumption, noise cancellation, and real time high speed applications. J Softw Eng Appl 3(4):391
27. Ferlini F et al. (2012) Non-intrusive fault tolerance in soft processors through circuit duplication. In: Proceedings of the 2012, 13th Latin American test workshop (LATW), IEEE, 2012
28. Montenbruck O (2008) GNSS receivers for space applications. Lecture. In: ACES and future GNSS-based earth observation and navigation
29. Hoyt R, Voronka N, Newton T, Barnes I, Shepherd J, Frank SS, Slostad J, Jaroux B, Twiggs R (2007) Early results of the multi-application survivable tether (MAST) space tether experiment; SSC07-VII-8/048; 21st annual AIAA/USU conference on small satellites, 13–16 Aug 2007, Logan, UT, USA
30. Scholz A, König F, Fröhlich S, Piepenbrock J (2009) Flight results of the COMPASS-1 Mission. http://www.raumfahrt.fh-aachen.de/compass-1/download/COMPASS-1%20Flight%20Results.pdf

Chapter 3
Overview and Investigation of SEU Detection and Recovery Approaches for FPGA-Based Heterogeneous Systems

Ediz Cetin, Oliver Diessel, Tuo Li, Jude A. Ambrose, Thomas Fisk, Sri Parameswaran, and Andrew G. Dempster

Abstract Growing international interest in the development of space missions based on low-cost nano-/microsatellites demands new approaches to the design of reliable, low-cost, reconfigurable digital processing platforms. To meet these requirements, future systems will need to include application-specific processors to handle control-dominated tasks and hardware accelerators to cope with data-intensive workloads. Commercial-Off-The-Shelf (COTS) Field-Programmable Gate Arrays (FPGAs) provide an ideal platform for meeting these requirements with application-specific processors implemented as soft cores along with hardware accelerators on FPGA fabric. However, the main challenge to deploying reconfigurable systems in space is mitigating the impact of radiation-induced Single Event Upsets (SEUs). In considering the design of such heterogeneous systems, we present a survey of techniques commonly employed to guard against soft errors in application-specific processors that are conventionally targeted at ASICs and assess their suitability to FPGA implementation when partial reconfiguration is used to deal with SEUs in logic circuits. Finally, we report on the development of the RUSH payload, to be deployed on the UNSW-EC0 CubeSat due for launch in 2016, to test our design approach.

3.1 Introduction

The low-cost, nano-/microsatellite (1–50 kg) segment, primarily based on the CubeSat standard and with applications in science, Earth Observation (EO) and reconnaissance, is expected to experience between 16.8 % and 23.4 % compound

E. Cetin (✉) • T. Fisk • A.G. Dempster
Australian Centre for Space Engineering Research, School of Electrical Engineering and Telecommunications, UNSW Australia, Sydney, NSW, Australia
e-mail: e.cetin@unsw.edu.au; t.fisk@unsw.edu.au; a.dempster@unsw.edu.au

O. Diessel • T. Li • J.A. Ambrose • S. Parameswaran
School of Computer Science and Engineering, UNSW Australia, Sydney, NSW, Australia
e-mail: odiessel@cse.unsw.edu.au; tuol@cse.unsw.edu.au; ajangelo@cse.unsw.edu.au; sridevan@cse.unsw.edu.au

F. Kastensmidt, P. Rech (eds.), *FPGAs and Parallel Architectures for Aerospace Applications*, DOI 10.1007/978-3-319-14352-1_3

annual growth over the period 2013–2020 [1]. This burgeoning international interest in the development of satellite-based space missions demands new approaches to the design of *reliable*, *low-cost*, *reconfigurable* digital processing platforms.

To meet these requirements, future space systems will need to include application-specific processors to handle control-dominated tasks and hardware accelerators to cope with data-intensive workloads. Some of these applications include secure and reliable communications, attitude determination and control, guidance, navigation and control as well as on-board image and Synthetic Aperture Radar (SAR) data processing and compression. Implementing these systems as Application-Specific Integrated Circuits (ASICs) is not viable due to their high cost, long lead times, and inflexibility. The implementation devices most suited to meeting these requirements are Commercial-Off-The-Shelf (COTS) Field-Programmable Gate Arrays (FPGAs) with application-specific processors implemented as soft cores along with hardware accelerators on FPGA fabric. FPGAs, like custom hardware chips, provide the means for implementing custom processors and accelerators, they can also be reconfigured on demand to perform new or different functions, and have significantly lower lead times and associated costs. Furthermore, by reusing the same device to implement an architectural variation, FPGA reconfiguration can be exploited to reduce mission-critical parameters, such as the system's size, mass and power requirements, which must be kept as small as possible. The main challenge to deploying a reconfigurable system in space, however, is radiation-induced Single Event Upsets (SEUs) [2].

An SEU occurs when deposited charge causes a change of state in dynamic circuit elements. In FPGAs, SEUs can modify not just the memory elements storing application data but also the configuration memory implementing the application circuits. Techniques for mitigating configuration memory errors are of crucial importance and are the subject of ongoing study.

As part of our research activity into rapid recovery from SEUs in reconfigurable hardware [3, 4], we are currently developing a payload for the University of New South Wales—Educational CubeSat Zero (UNSW-EC0) CubeSat as part of the European QB50 project to be launched in 2016 [5]. The RUSH (Rapid recovery from SEUs in Reconfigurable Hardware) payload will enable us to carry out in-situ flight testing of various FPGA-based rapid SEU detection and recovery approaches and compare them with vendor-specific tools such as the Soft Error Mitigation (SEM) controller from Xilinx [6].

This chapter considers heterogeneous systems consisting of application-specific processors and hardware accelerators implemented on FPGAs, and investigates the suitability of various circuit- and processor-based SEU detection and mitigation approaches with a view to final deployment on the UNSW-EC0 CubeSat RUSH payload.

The chapter is organized as follows: Sect. 3.2 provides an overview of Application-Specific Instruction-set Processor (ASIP) soft-error mitigation approaches and assesses their suitability for FPGA-based implementations. Sect. 3.3 provides details of approaches for rapid recovery from FPGA configuration memory upsets and discusses how these approaches could be applied to ASIPs and hardware accelerators. The RUSH payload and experiment are detailed in Sect. 3.4, while concluding remarks are given in Sect. 3.5.

3.2 ASIP Soft-Error Mitigation

Application-Specific Instruction-set Processors (ASIPs) are processors that are tailored by analyzing the characteristics of the specific application(s) that will be executed in the ASIPs [7]. ASIPs are typically used in embedded systems, where properties such as area, power, and performance are critical. An ASIP can be tailored by including custom instructions to improve performance, or by removing unnecessary components based on the mapped application(s) to reduce power, or by adding custom components to improve reliability. In contrast, General-Purpose Processors (GPPs) are designed to support a wide range of applications, and are not therefore customized for a particular set of applications. As embedded systems are commonly used in safety-critical applications such as aerospace, automotive, medical electronics, etc., maintaining the system's reliability is of great importance.

ASIPs are typically implemented in standard cells (such as ASICs), where radiation-induced soft errors mainly impact on sequential logic. For example, the register file and on-chip memory are the vulnerable parts of ASIPs implemented as ASICs, whereas the circuits themselves, such as the adder circuit, remain largely unaffected. However, when an ASIP is implemented in an FPGA device, the entire circuit is implemented in configuration memory, including the combinational circuit elements and the component interconnections. Since SRAM-based FPGA fabrics are susceptible to radiation-induced SEUs, the functionality of FPGA-based ASIPs can be affected, and unless they are corrected, configuration memory SEUs have the appearance of permanent errors in ASICs.

Techniques are therefore needed to detect and recover from SEUs that corrupt the configuration memory of FPGAs implementing ASIP circuits. In the following we survey approaches that have been studied in the context of protecting architectural state such as registers and instruction memory. These processor-level soft-error countermeasures can be grouped into two major categories: hardware (Sect. 3.2.1) and software (Sect. 3.2.2) based approaches. We present and elaborate a few representative genres of techniques in both categories that could also be adopted in FPGA implementations of ASIPs. The fundamental idea behind these techniques is to add redundancy into the system with regards to the architectural state. The techniques are compared with the literature on SEU mitigation for soft FPGA-based GPPs in Sect. 3.2.3. Note that since Error-Correcting Codes (ECC) are well established for storage elements such as the register file and memory, in this discussion we focus on the entire processor or the execution of instructions in the datapath pipeline. For each genre, we introduce the concept, system impact, and applicability to FPGA implementation.

3.2.1 Hardware-Based Soft Error Mitigation Approaches

Instruction Space Triple Modular Redundancy

Instruction space triple modular redundancy (space-TMR) adds two redundant instruction executions in parallel with the usual instruction execution, and recovers the error by selecting the result in majority with minimal overhead on processor

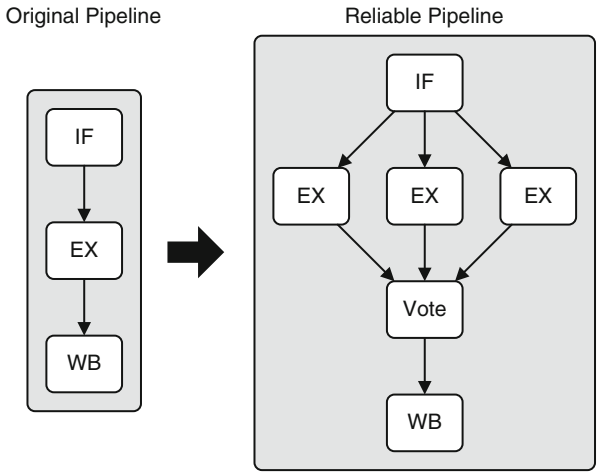


Fig. 3.1 Instruction space-TMR

performance. Theoretically, *N*-MR is able to detect errors when $N=2$ by comparing two results from two modules, and recover errors when $N=3$ by performing majority voting with three results from three modules.

Since ASIPs are typically implemented using pipelined datapaths, each pipeline stage or indeed the entire pipeline can be triplicated based on the cost constraints such as area, power, and performance (delay). Fig. 3.1 depicts an example for space-TMR where the EXecution pipeline stage (EX) is triplicated, and the three outputs are passed to a voter, before the final commit of the instruction at the Write-Back (WB) stage. The other stages could be triplicated as well to achieve better reliability, however this would incur additional area and power overheads. The impact of the approach on the processor architecture is to triplicate hardware components that execute the instructions i.e. the EX unit and to add a majority voting hardware unit. Thus, the main impact is hardware complexity, which leads to additional area and power costs. The additional hardware complexity is slightly more than twice that of the EX unit.

Considering FPGA implementations, instruction space-TMR is applicable to soft processors for which the RTL description of the processor is available so that the required modifications to the architecture can be made. However, modifying the architecture is infeasible for hard-core processors and commercially acquired soft processors for which the RTL is generally not provided.

Instruction Time Triple Modular Redundancy

Instruction time-TMR triplicates the execution of an instruction in a temporal manner. The redundant executions are generated by re-issuing the instruction two additional times. The result of the instruction is committed after majority voting on

the three results. For example, the work in [8] locks the Program Counter (PC) and executes the same instruction three times starting from the Instruction Fetch (IF) stage. In the first two executions, the output of the instruction is saved without committing at the WB stage. In the last execution, the three outputs are voted upon and then committed at the WB stage.

Since the one datapath is involved in re-executing the instructions, time-TMR can reduce errors that affect the architectural state of a processor, but does not specifically guard against, nor aid in the detection of configuration memory errors affecting the processor circuits.

The major architectural impact of instruction time-TMR is the logic to handle re-issuing of the instruction, temporary storage to hold the results before majority voting, and the majority-voting unit. The additional hardware is insignificant in comparison to instruction space-TMR. However, the performance of the processor is decreased by a factor of 3, due to the additional issues per instruction.

The applicability of instruction time-TMR to FPGA implementation is similar to that for space-TMR. For hard and commercial soft IP, adding the re-issue logic, the temporary storage and majority voter are infeasible. For soft processors for which the RTL is available, the approach could be used.

Instruction Checkpoint Recovery

Instruction Checkpoint Recovery (CR) is a recovery-only solution to soft errors or transient faults. Performing CR at each instruction within a basic block [9] allows the processor to save a subset of the architectural state as a backup. These preserved values can be used to re-write/restore the architectural state that was modified by the basic block (this process is called rollback or restoration), when an error is detected at the end of the block. Generally, instruction CR ensures that the execution of the program is backed up and can be recovered periodically.

For example, the original instruction *ADD R2*, *R3*, *R4* adds the values of register *R3* and *R4* in the register file and writes the result into register *R2*. With CR enhancement, this instruction will first save the current value of *R2* into a specialized reliable storage unit before committing the new value at the WB stage. Similarly, all the instructions in the current basic block that modify the values of the register file or data memory are enhanced to store the current values before being committed. If an SEU is detected at the end of the basic block, an interrupt is triggered to execute specialized rollback instructions that fetch the previous values from storage to write them back into the corresponding locations of the register file or data memory. It is worth noting that comparisons (branch instructions) are customized to trigger rollbacks internally when errors are detected.

A variety of detection techniques can be applied with CR. One possibility is a control-flow based detection technique [10]. In this work, a compile-time signature of every basic block of the program is calculated by performing an XOR of the machine code (however more advanced encoding techniques could be applied as well). These signatures are then inserted into the corresponding basic blocks.

At runtime, specialized hardware calculates a signature for the executed instructions. At the end of each basic block specialized hardware in branch instructions is used to compare the compile-time and runtime signatures. A mismatch of the signatures indicates the presence of an SEU in the instruction stream or the processor pipeline.

Instruction CR augments the architecture of the processor with: a checkpoint buffer, logic for managing the update of the checkpoint buffer i.e. reading architectural states and writing to the checkpoint buffer, and logic for flushing the pipeline and rewriting architectural states. The detection method imposes additional architectural modifications. There are similar limitations to the application of this technique to FPGA-based ASIPs as for the previous two approaches.

3.2.2 Software-Based Soft Error Mitigation Approaches

Software-Implemented Error Recovery

Software-Implemented Error Recovery (SIER) is a solely software-based approach. Following TMR principles, SIER triplicates each instruction to allow majority voting as the program is executed [11]. Each instruction copy uses different registers and different memory locations so as to not interfere with the others. As all instructions are processed using the original hardware the processor architecture does not need to be modified. For example, instruction *ADD R2*, *R3*, *R4* is transformed to three instructions *ADD R2*, *R3*, *R4*, *ADD R2′*, *R3′*, *R4′*, and *ADD R2″*, *R3″*, *R4″*. Where *R2*, *R2′* and *R2″* are the different registers representing the same variable in the program. These three instructions are executed sequentially. An extra segment of code is inserted after these three instructions are executed to vote on the value of *R2* at runtime. SIER can therefore protect architectural state, but not processor circuitry.

SIER necessitates modification of the compiler backend e.g., to perform register allocation. The voting segment can be added directly into the program. The SIER program code length is at least three times that of the original code, but the processor hardware is not modified. Applying SIER to FPGA implementations is feasible since SIER does not modify the processor architecture. However, memory costs might increase due to the increased code size.

Profile-Guided Code Transformation

Profile-Guided Code Transformation (PGCT) alters the software code based on an analysis of the program. The program is profiled to understand the dependencies between instructions, liveliness of variables/registers, and the execution frequency of instructions to determine the vulnerability of each instruction. The transformations include loop unrolling and data type reassignment [12]. By transforming the code, the variables that are estimated to be vulnerable to soft errors are enhanced (to reduce their chance of corruption). For example, considering instruction *ADD R2*,

Table 3.1 Summary of processor-level SEU mitigation techniques

Technique	Hardware impact	Software impact	Performance impact
S-TMR	Significant (>3×)	None	Critical path can be impacted by voters
T-TMR	Insignificant	None	Significant (>3×)
CR	Dependent on number of states and storage type	Insignificant (rollback routine)	Insignificant
SIER	None (memory for additional code lines)	Significant (>3×)	Significant (>3×)
PGCT	None (memory for additional code lines)	Dependent on code	Dependent on transformations

R3, *R4*, decreasing the time period that a variable/register (e.g., *R2* or *R3* or *R4*) spends in more vulnerable sequential logic (e.g., register file) and increasing the time period that it spends in less vulnerable sequential logic (such as memory with ECC) can increase the reliability of that variable. Hence, by applying these transformations, the vulnerability of the program can be reduced by up to 90 %, as reported in [12]. However, as with instruction time-TMR and SIER, PGCT does not afford any additional protection to processor circuitry.

PGCT induces no hardware complexity cost. However, the code size might change and the resultant performance can be degraded as well. To implement PGCT, the compiler backend must be modified to allow the transformation. In addition, knowledge of the processor architecture is needed to perform the vulnerability analysis. For example, to calculate the vulnerability of an instruction, the area and logic type of the hardware components occupied by that instruction are used. This technique is applicable to FPGAs since no hardware modifications are needed. However the increase in code size may affect the memory requirement.

3.2.3 Discussion

Table 3.1 summarizes the processor-level techniques discussed in this section. The techniques of column 1 are evaluated with respect to the characteristics of columns 2–4. Overall, the hardware-based techniques induce considerable area overheads, whereas the software-based ones result in execution time and instruction space penalties. With regard to FPGA applicability, most of the hardware-based techniques require the baseline processor architecture to be transparent and described in RTL, while software-based techniques simply require more memory.

SEU mitigation in soft FPGA-based GPPs has been studied extensively—we outline some representative examples of the work. [13] and [14] studied Dual Modular Redundancy (DMR) at the processor level, operating Leon2 [13] and MicroBlaze (MB) [14] in lock step, and performing checkpointing and recovery to correct datapath memory errors. Configuration memory errors were corrected by

scrubbing and partial reconfiguration, respectively. [15] used TMR to protect MBs and synchronized the register state after partial reconfiguration to correct configuration memory errors. [16] have employed DMR at the IF and EX stages of an OpenRISC processor; instruction execution is stalled, the faulty stage is reconfigured, and the instruction is re-executed when an error is detected. The work to date has tended to focus on mitigation techniques and reported the impact on area and performance. In contrast, our research goals are to achieve specified performance criteria (area, speed and power) while meeting recovery time guarantees.

In soft ASIPs targeted at FPGAs, the instruction time-TMR and software-based mitigation approaches do not guard against configuration memory errors because they do not provide any redundancy in the processing hardware. Currently, we therefore focus on spatial-TMR and outline our approach to recovering from configuration memory errors in the next section.

3.3 Rapid Recovery from FPGA Configuration Memory Upsets

The configuration memory of COTS FPGAs, being implemented in SRAM, is as prone to corruption due to radiation as the memory elements (FFs and BRAMs) of user circuits. Therefore, when COTS FPGAs are used in radiation prone environments, it is necessary to provide protection from radiation and/or methods for detecting and recovering from radiation-induced configuration memory errors. Moreover, in time critical applications, it is also desirable to detect and recover from errors very quickly.

There are two principal methods for detecting and recovering from configuration memory SEUs in COTS FPGAs. The first, direct method, typically referred to as scrubbing, involves scanning the configuration memory checking for upsets either via ECC associated with individual configuration memory frames, or by comparison with a golden reference stored off-chip in protected memory. Any elements that have been modified are refreshed in the course of the scan. FPGA vendors, such as Xilinx, provide in-built components to perform this function [6]. An alternative, indirect method, involves checking the behavior of the user circuit, and reloading the circuit configuration if the circuit no longer behaves as expected [4, 17]. In the latter case, TMR is typically employed to identify the module in error, and Dynamic Partial Reconfiguration (DPR) is used to reconfigure the erroneous unit. Built-in self-tests could also be employed to check correct functioning of the user circuits.

The scrubbing technique is usually deployed as a background process that operates periodically. There can therefore be a considerable delay between errors occurring and them being detected and corrected—on average, a delay corresponding to half the complete configuration delay can be expected. The TMR-based approach, on the other hand, is able to detect errors in the unit that is affected by checking for repeated errors. If the module that is triplicated is acyclic, then the occurrence of repeated errors in the same unit suggests its configuration memory is corrupted since transient errors affecting the datapath only give rise to isolated errors [4].

A threshold of 3–5 errors on successive clock cycles could be used to detect an error. Of course, if the module includes feedback paths, then even a transient error can lead to recycling of the erroneous value, and potentially give rise to multiple errors at the output. In any case, when the TMR-based approach determines that a unit is in error, it can trigger a partial reconfiguration of that unit, which can therefore be expected to incur less delay in correcting the error and require less energy than scrubbing since partial reconfiguration is only triggered when it is needed and the size of the unit is typically small compared to that of the complete configuration memory.

Regardless of the detection and configuration memory correction method used, thought must also be given to recovering the state of the affected user circuits. This detail is less comprehensively studied in the literature. When scrubbing is used, the designer needs to employ additional mechanisms, such as TMR and/or checkpointing, in the user circuit to recover the state. TMR-based approaches rely on checking each feedback state [18] or on waiting until the circuit enters a known state before resynchronizing the constituent modules of a TMR component [19]. In [4], as suggested by [20], the circuit to be protected is partitioned into acyclic components with each feedback edge being voted upon (see Fig. 3.2). After a module is reconfigured, its state is resynchronized with that of its siblings when the inputs to the module (including any feedback edges that have been voted upon) have emerged as outputs. The latency of the component therefore determines the resynchronization delay.

As outlined in the previous section, we propose using spatial-TMR to protect ASIPs for which the RTL description is available. It is relatively straightforward to

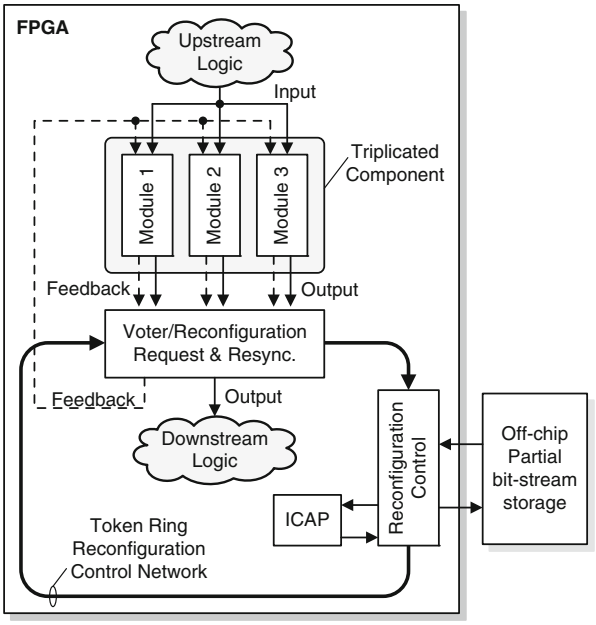


Fig. 3.2 Partial reconfiguration-based recovery from configuration memory SEU errors

then triplicate any single stage of a pipelined architecture whereby the pipeline register contents are voted upon. For example, triplicating just the EXecute stage (as depicted in Fig. 3.1) involves instantiating three copies of the ALU and the result (EX/WB pipeline) registers. The contents of the result registers are voted upon, and the majority value is then again used as a singular value to access memory or to be written back to the register file. This scheme allows transient errors in any single EX unit to be overwritten. Since the EX stage is invariably acyclic in structure, when any one unit is found to be in error over successive clock cycles, it is more likely that this has been caused by a configuration memory upset than for it to have been caused by successive datapath SEUs. A partial reconfiguration of that unit is then triggered. While the unit is being reconfigured, its two siblings continue to operate and the voter continues to check that they agree. After the partial reconfiguration of the erroneous unit has been completed, the output of the reconfigured unit can once again be expected to agree with that of its siblings after the next instruction is executed and its result is registered.

The same approach can be used to protect the instruction decode, register fetch, and register writeback logic after an ALU or memory load instruction. The on-chip control logic for off-chip memory accesses on instruction fetches, loads and stores can also be triplicated. Since off-chip memory is readily protected with ECC, triplicating the storage as well should not be necessary except in the most sensitive of applications.

For the above approach to be applicable, each component that is to be protected must be partitioned into acyclic sub-components. This is also a requirement of any extraneous accelerator or glue logic that is to be protected. Some means of coordinating the requests for reconfiguration between many voters and the reconfiguration controller also needs to be implemented. In [3], we outlined and assessed a token-ring architecture we use to implement a Reconfiguration Control Network (RCN) for this purpose (Fig. 3.2). The resulting system is resilient to radiation-induced errors as long as these errors don't re-occur at time intervals that are shorter than the time needed to recover from each error (comprised of the time to detect the configuration memory error, communicate the reconfiguration request, perform the partial reconfiguration, and resynchronize the reconfigured module). In [4] we found that the delay in recovering from an error is dominated by the time needed to perform the partial reconfiguration, which, in turn, was determined by the performance of the off-chip memory access and the internal configuration control circuits. The minimum expected inter-error period therefore determines the maximum component size we can use for reliable operation [3, 4].

3.4 The QB50 RUSH Payload and Experiment

The QB50 project, funded through the European Union Framework Programme 7 (FP7) and overseen by the Von Karman Institute (VKI) in Belgium, is a planned network of around 50, 2U and 3U CubeSats due to launch in 2016 into Low Earth

Orbit (LEO) that aims to provide a temporal and spatial image of the largely unexplored lower thermosphere. The individual CubeSats of the QB50 mission are being developed by various universities around the world compliant with the QB50 requirements [21] and are expected to carry one of the three VKI sensor payloads.

RUSH is one of three payloads that are currently under development for the UNSW-EC0 QB50 CubeSat. The primary objective of this payload is to demonstrate and validate new approaches to rapidly recovering from SEUs in reconfigurable hardware. The experimental goals of the payload are:

- Demonstrate and validate the partial reconfiguration approach to rapidly recovering from SEUs in reconfigurable hardware.
- Compare reconfiguration time and power consumption of scrubbing with partial reconfiguration approach.
- Map SEU event occurrences in the thermosphere.
- Demonstrate in-orbit reconfiguration.

The block diagram of the RUSH payload is shown in Fig. 3.3. As can be observed from Fig. 3.3, at the heart of the RUSH payload design is a Xilinx Artix 7 XC7A200T FPGA, chosen for its high logic density to power consumption ratio. The FPGA is connected to a flash memory device that stores the base configurations for the

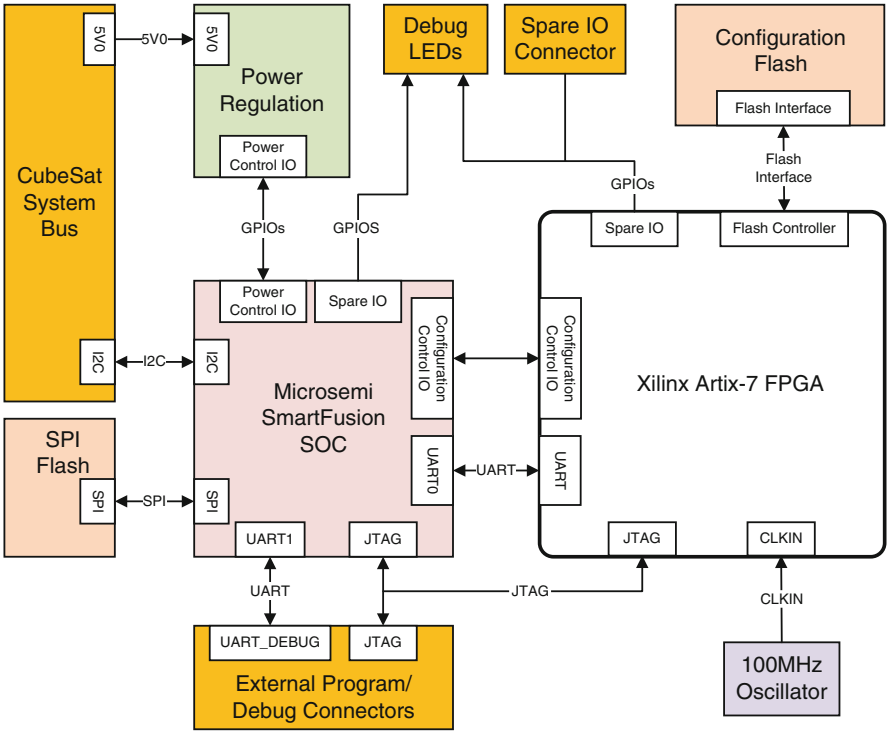


Fig. 3.3 RUSH payload block diagram

FPGA, as well as the partial bitstreams of the modules that can be partially reconfigured via DPR. The FPGA is connected via a UART interface to a Microcontroller Unit (MCU) that acts as an interface between the FPGA and the UNSW-EC0 CubeSat system bus, and communicates with the On-Board Computer (OBC) via the I2C interface. The MCU also oversees the overall operation of the RUSH payload and controls the power-up/down of the FPGA, as well as logging SEU detection and recovery statistics and the power usage. To fulfil the requirements for the MCU in the proposed design, a Microsemi SmartFusion 2 System-On-Chip (SoC) was selected. Furthermore, since the SoC is based on non-volatile FLASH memory it is resilient to SEUs [22]. A small number of additional components provide ancillary functions such as providing regulated power, clock sources, programming interfaces and status indicators.

The primary objective of the RUSH experiment is to test and validate new approaches to rapidly recovering from soft errors in reconfigurable hardware involving accelerator logic and soft ASIPs and to compare the performance of the approach with that of the Xilinx SEM controller [6]. To this end, two essentially identical configurations are being developed. One configuration will employ the partial reconfiguration method outlined in Sect. 3.3 and depicted in Fig. 3.2 to guard against and recover from soft errors in user logic and configuration memory, and the other configuration will utilize the SEM controller to continuously scan and scrub the FPGA configuration memory. To enable comparison of SEU susceptibility and recovery, the two configurations comprise essentially the same circuitry, but the SEM configuration will not partially reconfigure its triplicated components.

The experiment will play a vital role in testing the susceptibility of Artix-7 FPGAs in LEO, and will demonstrate the use of dynamic partial reconfiguration on an FPGA in space. The design will be composed of two base components: a Portable Instruction Set Architecture (PISA)-based Advanced Encryption Standard (AES) custom processor with triplicated execution units, and a Block Adaptive Quantization (BAQ) circuit, chosen for its utilization of all FPGA resource types (LUTs, FFs, DSPs, and BRAMs). These base components will be replicated to fill the FPGA area, thereby creating the largest possible surface for SEUs to be detected.

Within the thermosphere (<400 km orbit) we do not expect more than one error per 1,000 s of FPGA operation on average. Nevertheless, the triplicated components of the test circuits will be sized (see Sect. 3.3) so that error recovery can be achieved within 10 ms to counteract rapid bursts of errors. This component size implies that we may have on the order of 100 voters to manage using the RCN. Based on the experimental results of a previous implementation of the RCN [4], we can therefore expect a communications latency on the order of 100 µs and an overall reconfiguration control latency of under 1 ms. We intend experimentally assessing the availability of the DPR-based configuration by comparison with the performance of SEM controller-based configuration that will be able to identify the precise bits that were affected when it was in operation. These will then be assessed on the ground for their sensitivity.

During the experiment SEU events will be logged by the MCU and the time, location, and time to recover will be transmitted to Earth when UNSW-EC0 passes

over any of the ground stations available for the QB50 mission. Due to power limitations of the UNSW-EC0, the RUSH experiment will not run continuously. To deal with this, the available uptime will be evenly distributed between the two configurations. Furthermore, the activity of both configurations will be scheduled such that they occur at similar times and locations.

3.5 Conclusions

We have argued for the need to support soft ASIPs and logic in COTS FPGAs for future low-cost space missions. We have surveyed techniques commonly employed to guard against soft errors in ASIPs targeted at ASICs, where the processor state is susceptible to corruption and assessed the applicability of these techniques to ASIPs implemented on FPGAs. We have outlined an experiment that is to be conducted as part of the QB50 mission in 2016 involving an off-the-shelf Xilinx Artix-7 FPGA that will be flown into a low Earth orbit. As part of the experiment we will trial approaches to protecting soft processor and logic circuits that are expected to result in quicker recovery and lower power consumption than standard techniques. Our experiment will also help to gauge the susceptibility of modern high-density COTS FPGAs to SEUs in the thermosphere. If our methods prove to be beneficial, we aim to refine and generalize them to provide a low-cost, rapid development platform for protecting FPGA-based processor and logic systems against radiation-induced soft errors.

References

1. SpaceWorks (2013) Nano/Microsatellite market assessment. bit.ly/17p9M5F
2. Asadi H, Tahoori MB, Mullins B, Kaeli D, Granlund K (2007) Soft error susceptibility analysis of SRAM-based FPGAs in high-performance information systems. IEEE Trans Nucl Sci 54(6):2714–2726
3. Cetin E, Diessel O, Lingkan G, Lai V (2013) Towards bounded error recovery time in FPGA-based TMR circuits using dynamic partial reconfiguration. In: Proceedings of the 23rd international conference on field programmable logic and applications (FPL), 2013, pp 1–4
4. Cetin E, Diessel O, Lingkan G, Lai V (2014) Reconfiguration network design for SEU recovery in FPGAs. In: Proceedings of the 2014 IEEE international symposium on circuits and systems (ISCAS), 2014, pp 1524–1527
5. QB50 Project Description https://www.qb50.eu/index.php/project-description-obj
6. LogiCORE IP soft error mitigation controller v4.1 product guide, Xilinx App. Note PG036 2014
7. Praet JV, Goossens G, Lanneer D, Man HD (1994) Instruction set definition and instruction selection for ASIPs. In: Proceedings of the 7th international symposium on high-level synthesis, IEEE Computer Society Press, 1994, pp 11–16
8. Tuo L, Shafique M, Ambrose JA, Rehman S, Henkel J, Parameswaran S (2013) RASTER: runtime adaptive spatial/temporal error resiliency for embedded processors. In: Proceedings of the 2013 50th ACM / EDAC / IEEE design automation conference (DAC), 2013, pp 1–7
9. Tuo L, Ragel R, Parameswaran S (2012) Reli: hardware/software checkpoint and recovery scheme for embedded processors. In: Proceedings of the design, automation & test in Europe conference & exhibition (DATE), 2012, pp 875–880

10. Ragel RG, Parameswaran S (2006) IMPRES: integrated monitoring for processor reliability and security. In: Proceedings of the 2006 43rd ACM/IEEE design automation conference, 2006, pp 502–505
11. Reis GA, Chang J, August DI (2007) Automatic instruction-level software-only recovery. IEEE Micro 27(1):36–47
12. Rehman S, Shafique M, Kriebel F, Henkel J (2011) Reliable software for unreliable hardware: embedded code generation aiming at reliability. In: Proceedings of the 9th international conference on hardware/software codesign and system synthesis (CODES+ISSS), 2011, pp 237–246
13. Reorda MS, Violante M, Meinhardt C, Reis R (2009) A low-cost SEE mitigation solution for soft-processors embedded in systems on pogrammable chips. In: Proceedings of the design, automation & test in Europe conference & exhibition (DATE '09), 2009, pp 352–357
14. Hung-Manh P, Pillement S, Piestrak SJ (2013) Low-overhead fault-tolerance technique for a dynamically reconfigurable softcore processor. IEEE Trans Comput 62(6):1179–1192
15. Ichinomiya Y, Tanoue S, Amagasaki M, Iida M, Kuga M, Sueyoshi T (2010) Improving the robustness of a softcore processor against SEUs by using TMR and partial reconfiguration. In: Proceedings of the 2010 18th IEEE annual international symposium on field-programmable custom computing machines (FCCM), 2010, pp 47–54
16. Vavousis A, Apostolakis A, Psarakis M (2013) A Fault tolerant approach for FPGA embedded processors based on runtime partial reconfiguration. J Electron Test 29(6):805–823
17. Bolchini C, Miele A, Santambrogio MD (2007) TMR and partial dynamic reconfiguration to mitigate SEU faults in FPGAs. In: Proceedings of the 22nd IEEE international symposium on defect and fault-tolerance in VLSI systems, DFT '07, 2007, pp 87–95
18. Carmichael C (2001) Triple modular redundancy design techniques for Virtex FPGAs. Technical report, Xilinx Corp., XAPP197 (v1.0)
19. Pilotto C, Azambuja JR, Kastensmidt FL (2008) Synchronizing triple modular redundant designs in dynamic partial reconfiguration applications. In: Proceedings of the 21st annual symposium on integrated circuits and system design, ACM, 2008, pp 199–204
20. Johnson JM, Wirthlin MJ (2010) Voter insertion algorithms for FPGA designs using triple modular redundancy. In: Proceedings of the 18th annual ACM/SIGDA international symposium on field programmable gate arrays, ACM, 2010, pp 249–258
21. VKI, QB50 System requirements and recommendations and interface control document, Issue 3, VKI, 2013
22. Microsemi Corp., SmartFusion customisable system-on-chip (cSoC) (2013) http://www.actel.com/documents/SmartFusion_DS.PDF

Part III
SRAM-Based FPGAs

Chapter 4
A Fault Injection technique oriented to SRAM-FPGAs

H. Guzmán-Miranda, J. Barrientos-Rojas, and M.A. Aguirre

Abstract Fault injection is an accepted method for emulating the effect of ionizing radiation on digital electronic circuits. It can be oriented either to ASIC designs or to SRAM-FPGA designs. When the target device is an SRAM-FPGA the injection has to be assessed both in the functional plane and in the configuration plane. It has been demonstrated that the classical protections oriented to the functional structure are not enough, so the configuration plane has to be analyzed, in the same way. This paper describes the adaption of the FT-UNSHADES2 platform as a fault injection system that tests faults in the configuration plane. The mechanism that assesses the effect of faults in the configuration is read-modify-write, in cycles of inject and repair, based on partial reconfiguration.

In this paper the authors categorize that there are four types of possible faults in the FPGA that should be considered: unrelated, non-damage, outer-propagated and inner-propagated. Faults in the unrelated and non-damage configuration bits are affordable and can be fixed using scrubbing techniques. The damage and propagated faults propagate from the configuration plane to the current data processed and a complete scrubbing followed by a master reset should be asserted to recover the functional behavior of the device.

Other result found is the relationship between the faults in the functional observability and the configuration bits. A result that only can be found if the injection system can distinguish between the faults over the above mentioned planes.

4.1 Introduction

SRAM-FPGAs are digital electronic devices that provide an attractive solution to many aerospace applications [1]. They introduce certain flexibility to the airborne systems and space payloads which allow the actualization and improvement of the electronic subsystems, and also deal with the possible obsolescence of their components [2].

H. Guzmán-Miranda (✉) • J. Barrientos-Rojas • M.A. Aguirre
Departamento de Ingeniería Electrónica, Escuela Superior de Ingeniería,
Universidad de Sevilla, c/ Camino de los Descubrimientos s/n, Sevilla 41092, Spain
e-mail: hipolito@gie.esi.us.es

F. Kastensmidt, P. Rech (eds.), *FPGAs and Parallel Architectures for Aerospace Applications*, DOI 10.1007/978-3-319-14352-1_4

The main drawback of this kind of devices is their extreme sensitivity to ionizing radiation due to the huge quantity of memory cells that compound their structure and the large critical area exposed.

Faults in configuration bits (CB) react in a different way than faults in user registers. While faults in user registers are treated as transient anomalous values that produce corrupted states in the cadence of the circuit, faults in the configuration bits have to be treated as structural modifications which remain permanent until the configuration is overwritten or repaired. Classical protections introduced in the design structure, like Triple Modular Redundancy (TMR) [3, 4] are still insufficient, because configuration faults can affect simultaneously to circuits belonging to several clock domains, or propagate the fault to user logic.

Errors in the configuration are much more probable than the user register ones, due to the abundance of sensitive points. They can be detected by means of a complete *Readback* of the device, in the same way than a normal SRAM-memory [5].

However, the number of CBs related to a particular design is a small fraction of the total CBs. Xilinx has developed a special mitigation method based on the so called Soft Error Mitigation (SEM) core [6]. It combines with an option in the *bitgen* application known as "essential bits". This option generates two files that determine the CBs that are related to the design. Essential bits are obtained by means of a static analysis of the design; this analysis calculates those CBs that are related to the implementation of the design, regardless of their actual value, and are strictly part of the configuration of any element of the FPGA. LUTs, BRAM contents and FF contents are not included in this file.

We take the advantage of this option for the adaption of the FT-UNSHADES2 tool to the injection of faults oriented to the essential bits. We will characterize the tool with a small example.

The rest of the paper is organized as follows: A general introduction about how the fault injection procedures are, when an SRAM FPGA is the user platform. In the third section the FT-UNSHADES2 system is described and also the skills implemented in the system to target the FPGA as object of injection and finally a case of study is introduced to show the system behavior.

4.2 Fault Injection in SRAM-FPGA

4.2.1 Fault Injection Oriented to User Registers

SRAM-FPGAs are a very attractive solution for fault injection tasks when the designer wants to analyze how the design structure treats the faults: where the weakest elements are and how the protections work within the circuit structure. There are several proposals in the literature for platforms that develop this concept. Basically they consist of the implementation of a mechanism that produces one or several spontaneous changes in the content of implemented registers (WHERE) at any clock cycle of the execution workload (WHEN), and if there is a predefined method

of injection (HOW). The most important characteristic of this procedure is that the injection is performed over the circuit registers, or registers instantiated due to the high level description of sequential statements. A good survey can be found in [6].

A very well-known system based on this approach is "Autonomous Emulation" system [7], that make use of any kind of SRAM-FPGA, instrument each register the circuit for being tested and make a fast emulation of the system in fault. The platform ASTERICS [8] is another example of how to inject faults reconfiguring.

4.2.2 Fault Injection Over the Configuration Plane

Another category that is completely different (but almost always confusing) is those platforms that are dedicated to study the proper SRAM-FPGA as the target device. This problem is completely different because the SEE can impact not only on the instantiated registers but also in the configuration bits of the elements that are related to the design [9, 10]. The consequences of SEE are totally different than in the former category because the faults remain in the configuration bit over time and will only be removed when the configuration is overwritten. During the time that the fault is active the fault can be propagated to the user logic and then the processed state becomes corrupted.

Overwriting the state of the configuration is done periodically, and the timing is known as scrubbing period, so the time between reconfigurations is the vulnerable time. In a scrubbing cycle, the configuration is overwritten "softly", in such a way that the current state represented by the content of the memory elements of the FPGA remains untouched. This method by itself does not detect if the state of the design is currently corrupted or not, and the scrubbing process takes extra and undesired power due to the internal commutations of the transistors. One goal of the design is to optimize time and power consumption.

Mitigation with scrubbing is not enough, because data remains already corrupted after the soft reconfiguration, so it is necessary to introduce another mitigation technique, in this case, focused on the repairing of data.

Several platforms have been created, mainly for the measurement of the global sensitivity of a design to SEE in a particular FPGA device. The main goal consists in studying the design behavior when the device is configured, and then reconfigure it in a blind manner. The number of errors found versus the number of injections is considered as a measurement of how reliable, running in this device, the circuit is. This is a very inefficient mechanism due to the large amount of configuration bits unrelated to the design. These bits are sensitive from the point of view of the device, but most of them are not, considering the configured action. Many of the injections can be saved if we can distinguish between the related bits and the unrelated.

Few platforms have been developed to test designs running on the SRAM-FPGA (e.g. FLIPPER tool) [11–15], and few correct approaches have been addressed because a platform is needed for the exact device that is going to be flown in the final application. One solution is to study the design as a hard macro of the design,

a part of the identical configuration that will integrate the final device. This is a method to migrate the design within the same technology.

4.2.3 Static vs. Time Zero Analysis

There are approaches that provide information about the reliability of the design just by studying the possible related bits. This is done by software tools like STAR [13] and the Xilinx *bitgen* routines for essential bits determination [16]. The former goes ahead, because it provides rules for a new placement that diminishes the number of critical configuration bits: the RORA tool [18].

Static analysis provides information regardless of whether a particular resource is used or not in the execution of the design. Of course this is the best situation but when the user has to take actions for reducing the number of critical resources the situation is not clear, as there is not idea about the sensitivity of each zone of the circuit to make it more reliable. One possible solution is the use of the SRAM-FPGA executing the design with a representative application workload. The configuration is modified in the clock cycle zero and the effect of the fault is recorded during the workload if there is any propagation path.

If any critical point is not detected, either its effect remains latent within the circuit or the resources are not well stimulated by the workload [5].

Time zero analysis is less restrictive and more realistic than the static one. It identifies the part of the circuit that can propagate faults. It consists of injecting the fault before the execution of the circuit is started. Normally it starts with a reset assertion and if the circuit is modified by the fault in the configuration, the fault is propagated during the workload to any primary output. Platforms watch this sequence of values and detect any anomaly or wrong value. If this is done, the injection is representative of an error rate for a specific implementation of the circuit and workload.

There are few but well known platforms described in the literature. All of them are devoted to the study of fault injection rates injecting using several techniques and internal resources of the Xilinx FPGA.

Again the next step is to provide rules for a new and more reliable implementation. The work should be done iteratively to minimize the criticality of the implementation. Next section will present the option of dynamic injection. The idea is to open the injection to any clock cycle of the workload.

4.3 FT-UNSHADES2 in FPGA Mode

Authors intentionally have omitted the platform FT-UNSHADES2 [17]. This platform traditionally has been described and classified in the set of tools dedicated to test SEE oriented to inject faults in the user registers that belong to the custom logic,

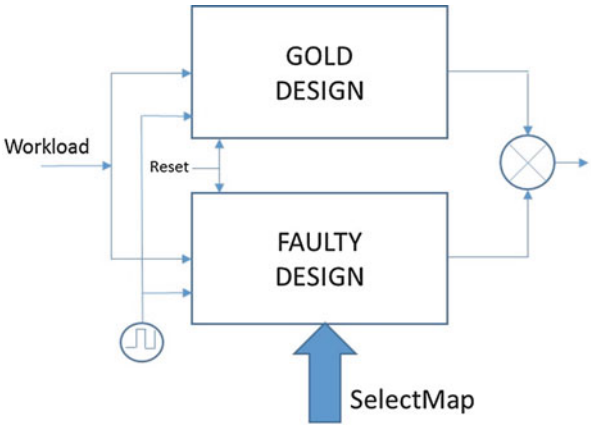


Fig. 4.1 Hardware for injection model

but in this section we are going to describe the adaptation of the tool to the test of designs implemented in FPGAs, so the injection procedures are produced over the *configuration bits*, instead of the *user registers* of the FPGA.

The principle of the method is essentially the same: use partial reconfiguration to read, modify and write a particular frame of the configuration map where the CB is allocated. The identification of the injection point provides a rich information about the reliability of the design, or some critical parts of it (Fig. 4.1).

The adaptation of one method, called *ASIC mode,* to the other, called *FPGA mode,* is at API level. Very low level commands are basically the same. The structure of the system is still based on two identical FPGAs running in parallel, synchronized, both receiving the same sequence of stimuli, and only one of them receiving the injections. The comparison is cycle by cycle at the primary outputs. This procedure is performed repeatedly, always with a known starting state at cycle 1. Every execution of the workload is called *run*. At each run one or several injections are performed selecting the target registers (WHERE and HOW) and clock cycles (WHEN) to inject.

The effect of a fault can be inspected either by on line comparison with the primary outputs coming from the twin FPGAs (*error* faults) or by reading the internal state of all the registers of both FPGAs and comparing their values one to one. This method detects the internal *latent* faults.

In *ASIC mode* the faults are injected only in user registers, faults can be compensated through functional structures, so they can be repaired if the circuit is prepared to. At every injection cycle, the signal reset is asserted in order to initialize the registers content.

In *FPGA mode* the faults are injected in CBs. The abundance or possible target bits (tens of millions) makes the problem very difficult to deal with if there is no previous selection of these CBs. Xilinx has provided a tool very similar to STAR that extracts the CBs that are related to the actual implementation of the design. The rest of CBs are unrelated and should not affect to the design behavior if they receive a bit flip. The tool provides in fact two files, one marking the bits that are related and

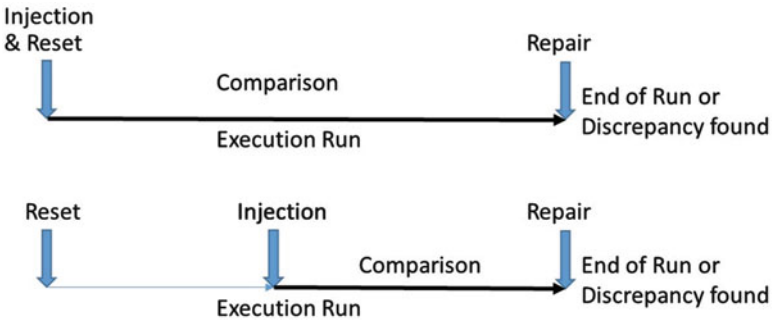


Fig. 4.2 Dynamic injection execution model. (**a**) Time zero injection (**b**) Variable time injection

other with the theoretical value of those configuration bits. These files are part of a mechanism of on-line repairing of the SEE in the configuration plane. Xilinx has developed this procedure for Virtex 5, 6 and 7 families.

FT-UNSHADES2 has taken these files as reference for the FPGA mode for a technique based on inject and repair cycles. The points of injection are determined by the essential bits file and these bits are the ones attacked. The method is based on the idea that when a CB is attacked, this change of value will not affect another configuration bit, otherwise the technique is not strictly valid, because the effect of a fault would remain present in the FPGA after a reset. The attack model is described then, as follows:

1. Selection of the configuration bit and clock cycle that will be attacked (WHERE and WHEN).
2. Initial reset, and execute the application until the injection instant.
3. Using partial reconfiguration, the frame that corresponds to the CB is retrieved from the FPGA
4. (alt) this step can be substituted by the theoretical value coming from the .ebc file.
5. Write the opposite value in the desired CB
6. Resume the execution and compare primary output values.
7. While execution, compare with Gold theoretical values.
8. If a discrepancy is found or end of run is reached, repair the CB, following the step 3.

This mode is repeated in many execution runs following the procedure established in the method of injection selected. If time zero is selected, then the injection is produced just at the beginning of the experiment. If time is a variable, then the system is driven to any clock cycle following the programmed selection pattern (Fig. 4.2).

The user can proceed to send a complete configuration at any certain number of injections in order to refresh it and erase any unexpected lateral effect.

Also the system allows avoiding the step 7 and studying possible accumulated effects.

The most important difference between this system and other developments is the consideration of the time as variable. It is very important to dedicate effort to the elaboration of the test vectors, because they must be representative of the real application, in order to make the results of the test more realistic and valuable.

The second advantage of the current platform is that the designer can compare between how the faults behave in the same framework from the point of view of ASIC mode and FPGA mode, and compare both. This is especially interesting, because in normal flight, the faults are detected using a specific detection circuit and monitored at any primary output.

The current system is based on Virtex 5 technology, and all the transactions are performed through the SelectMap port in parallel mode.

4.4 A Case of Study

This chapter will explain a case of study that characterizes the system. All the results come from the FT-UNSHADES2 platform. We have developed a set of examples to characterize the process. The examples *b01*, *b13*, *b20* and *keccak* sponge function, the former are complex circuits taken from the ITC99 benchmark suite and the latter is part of a cryptocodec found in internet. All of them are examples that have available the high level description code with a stimuli set. In the case of keccak example we have used two different sets of vectors to show the dependence of the observability on the application.

Previous to the experimental activity a study about the essential bits has been performed. For a blind attack, a complete sweep of all the used frames and all the configuration bits has been performed injecting in a blind way, say, if they are in the subset of essential bits or not. Then the essential bits were attacked. All the critical bits were detected in both subsets matching almost perfectly, with the unique difference of several bits in some frames of the blind sweep, corresponding to the LUTs and FF contents, that are not part of the essential bits. This experiment was performed over *b01* and *b13* circuits.

The results of these previous experiments confirm that the essential bits are a good subset for an effective fault injection campaign, as promised by Xilinx. However there are user memories that are not included in the essential bits subset. These bits should be added to those bits that are critical.

The first analysis has been performed to compare static analysis versus dynamic analysis. This experience pursues to compare the basic injection process. The number of injection points is given by the essential bits static analysis generated from the *bitgen* tool. In our examples set, the target device is XCV5FX70T, containing 18,936,096 bits.

For all the benchmarks, the first cycle is the assertion of the reset signal. This vector erases the possible functional value stored in previous execution runs and starts the current one from a known state (Table 4.1).

Table 4.1 Characterization of each benchmark

Benchmark	Registers	Workload	Essential bits
b01	10	245	3,216
b13	66	7,640	14,572
b20	434	10,933	475,230
keccak1	1,683	856	622,168
keccak10	1,683	8,798	622,168

Table 4.2 ASIC mode results

Benchmark	Inject.	Errors	Percentage	Time (s)
b01	10,000	7,666	76.6	5
b13	10,000	8,072	80.7	42
b20	100,000	16,105	16.1	428
keccak1	50,000	45,421	90.8	27
keccak10	50,000	46,420	92.0	239

Table 4.3 FPGA mode in time zero

Benchmark	Inject.	Errors	Percentage	Time (s)
b01	10,000	6,765	67.6	25
b13	10,000	5,960	59.6	426
b20	200,000	1,525	0.75	4,840
keccak1	622,168	40,578	6.52	3,637
keccak10	622,168	49,190	7.91	30,896

The pair workload/circuit is firstly tested as "ASIC mode" in order to test the fault propagation capabilities of each benchmark. The keccak example is used twice with different input vector databases. One is a single frame of data, and the second comprises ten frames (Table 4.2).

This experiment shows how the circuit structures propagate the faults. B01, B13 and keccak are examples that provide a high level of observability of faults, because they can be easily propagated to the primary outputs. It is very important to test, for each design-stimuli pair, their respective fault propagation capacity. Attacking the user registers, it is possible to measure this effect.

The keccak example shows that there is a dependence with how the stimuli set helps this propagation.

The next table shows the examples injecting only over those frames and configuration bits that belong to the CLBs. The injection technique implemented is the previously described inject and repair one. Table 4.3 shows the results for *Time Zero* experiment:

For comparison, the same experiment has been made but randomly selecting the injection *cycle*. A small decrease of the percentage of detected faults is expected, due to faults that could not have enough clock cycles to propagate to the outputs. Table 4.4 shows this effect with a smaller percentage of errors in all the examples. This situation is much more realistic than the previous one.

Table 4.4 FPGA mode in random time

Benchmark	Inject.	Errors	Percentage	Time (s)
b01	10,000	6,275	62.7	26
b13	10,000	5,366	53.6	440
b20	200,000	1,239	0.60	4,937
keccak1	1,000,000	49,131	4.91	5,936
keccak10	2,000,000	153,612	7.68	99,671

The keccak10 experiment is performed about three times per configuration bit. This shows that the experiment becomes similar to the time zero one. As the percentage shows, the time zero will be an upper bound of the real experiment, more pessimistic than the random time experiment.

The first conclusion is that not all the essential bits present errors. That means that the essential bit set is compounded by two subsets: the first one, is the critical ones, where faults introduce errors in the processing data and are detected at the outputs affecting to the processed data. The second is related to those bits whose error produces perturbations only in the propagation time of the connections, so they only change the parasitic capacitances of the wires. They are difficult to detect, but easy to prevent. In fact the critical ones are the candidates of being measured and if possible, mitigated. They give the real vulnerability degree of the design running in the current FPGA. The first group needs to be repaired using any logical mitigation technology plus the necessary scrubbing process to erase the errors.

These results also show that the FPGA mode is strongly related to the ASIC mode. The global observability of a design shows the propagation capacity of a particular design to the detection mechanism, that in these examples are simply the primary outputs. The experiments over B01, B13 and keccak circuits have high controllability and observability so it is expected that faults have an easy propagation to primary outputs. However B20 has a bad architecture for propagating faults. These numbers do not show that there is a high difference between the time zero experiment and dynamic experiment, but they show that the capacity of a design to propagate the perturbation is a very important measurement of its behavior.

4.5 Conclusions

This paper presents, for the first time, a flexible platform that is ready to perform fault injection over designs that are synthesized specifically for FPGA. The paper discusses the differences between the ASIC and FPGA modes, where there is a connection between them. Also this paper shows the procedure for the robustness assessment of a design, and how to implement the design in one device and translate it to another that belongs to the same family. It is also shown the influence of the workload in the processing data, showing that the workload has to be representative of the final functionality. This paper shows how different models of SEU tests can offer results depending on the timing scheme of the study.

Further work will study larger and more complex designs where new conclusions can be extracted.

Acknowledgments The authors would like to thank Junta de Andalucía, Spain for funding the EDELWEISS: Design of a Highly Efficient Intra-Satellite Wireless Communications System project (reference P11-TIC-7095), the European Space Agency for funding the FT-UNSHADES2 project (reference PI-0072/2010), and Luis Sanz for his help and insight with the mass processing of the fault injection results.

References

1. Heiner J, Sellers B, Wirthlin MJ, Kalb J (2009) FPGA partial reconfiguration via configuration scrubbing. In: Proceedings of the field programmable logic conference 2009, PL'09, pp 99–104
2. Guzman-Miranda H, Sterpone L, Violante M, Aguirre MA, Gutierrez-Rizo M (2011) Coping with the obsolescence of safety- or mission-critical embedded systems using FPGAs. IEEE Trans Ind Electron 58(3):814–821
3. Rollins N, Wirthlin M, Caffrey M, Graham P (2003) Evaluating TMR techniques in the presence of single event upsets. In: Proceedings for the 6th annual international conference on military and aerospace programmable logic devices (MAPLD) Washington, DC, NASA Office of Logic Design, AIAA, Sept 2003, p P63
4. Wirthlin M, Johnson E, Rollins N, Caffrey M, Graham P (2003) The reliability of FPGA circuit designs in the presence of radiation induced configuration upsets. In: Proceedings of the 2003 IEEE symposium on field-programmable custom computing machines, 9–11 Apr 2003, pp 133–142
5. Quinn HM, Black DA, Robinson WH, Buchner SP (2013) Fault simulation and emulation tools to augment radiation-hardness assurance testing. IEEE Trans Nucl Sci 54(1):252–261
6. Morgan K, Caffrey M, Graham P, Johnson E, Pratt B, Wirthlin M (2013) SEU-induced persistent error propagation in FPGAs. IEEE Trans Nucl Sci 60(3):2119–2142
7. Lopez-Ongil C, Garcia-Valderas M, Portela-Garcia M, Entrena L (2007) Autonomous fault emulation: a new FPGA-based acceleration system for hardness evaluation. IEEE Trans Nucl Sci 54(1):252–261
8. Velazco R, Mansour W, Pancher F, Costa Marques G (2011) ASTERICS—a platform for the simulation of radiation effects on processors by fault injection. Open access paper: https://www.rd-access.eu/edatools/system/files/_edaTools/ubooth_submission/2011/209.pdf
9. Bernardi P, Sonza Reorda M, Sterpone L, Violante M (2004) On the evaluation of SEU sensitiveness in SRAM-based FPGAs. In: Proceedings of the international on-line testing symposium (IOLTS), 2004, pp 115–120
10. Lima F, Carmichael C, Fabula J, Padovani R, Reis R (2001) A Fault injection analysis of Virtex FPGA TMR design methodology. In: IEEE European conference on radiation and its effect on component and systems, 2001, pp 275–282
11. Nazar GL, Carro L (2012) Fast single-FPGA fault injection platform. Defect and fault tolerance in VLSI and nanotechnology systems (DFT), 2012 IEEE international symposium on, 3–5 Oct 2012, pp 152–157
12. Bolchini C, Castro F, Miele A (2009) A Fault analysis and classifier framework for reliability-aware SRAM-based FPGA systems. In: Proceedings of the international symposium on defect and fault tolerance in VLSI and nanotechnology systems (DFT), 2009, pp 173–181
13. Sterpone L, Violante M, Rezgui S (2006) An analysis based on fault injection of hardening techniques for SRAM-based FPGAs. IEEE Trans Nucl Sci 53(4):2054–2059
14. Sterpone L, Violante M (2007) A new partial reconfiguration-based fault-injection system to evaluate SEU effects in SRAM-based FPGAs. IEEE Trans Nucl Sci 54(4):965–970

15. Alderighi M, et al (2007) Evaluation of single event upset mitigation schemes for SRAM based FPGAs using the FLIPPER fault injection platform. In: Proceedings of the 2007 international symposium defect and fault tolerance in VLSI systems, Rome, Italy, Sept 2007, pp 105–113
16. Xilinx App note Xapp 538. April 2012
17. Mogollon JM, Guzman-Miranda H, Napoles J, Barrientos J, Aguirre MA (2011) FTUNSHADES2: A novel platform for early evaluation of robustness against SEE. Radiation and Its effects on components and systems (RADECS), 2011 12th European conference on, 19–23 Sept 2011, pp 169–174
18. Sterpone L, Aguirre M, Tombs J, Guzmán-Miranda H (2008) On the design of tunable fault tolerant circuits on SRAM-based FPGAs for safety critical applications. Proceeding of the design automation and test in Europe conference DATE 2008. Munich, Germany. 2008, pp 336–341

Chapter 5
A Fault Injection System for Measuring Soft Processor Design Sensitivity on Virtex-5 FPGAs

Nathan A. Harward, Michael R. Gardiner, Luke W. Hsiao, and Michael J. Wirthlin

Abstract This paper presents an FPGA fault injection system, a methodology for soft processor fault injection, and fault injection experimental results for MicroBlaze and LEON3 soft processor designs. The Xilinx Radiation Test Consortium—Virtex 5 Fault Injector (XRTC-V5FI) was built to evaluate the configuration memory sensitivity of soft processor designs. To overcome some of the challenges of soft processor fault injection, we designed the XRTC-V5FI to be fast, flexible, and to fully cover all configuration memory bits. The minimum time to inject a full bitstream is 28 minutes and the individual fault injection can be as fast as 49 μS. The LEON3 has 81.3 % more sensitive bits than the MicroBlaze, yet when normalized by the number of used slices, the MicroBlaze is 26.2 % more sensitive than the LEON3.

5.1 Introduction

Operating microelectronic devices in high radiation environments greatly increases their potential to malfunction. Energized ions colliding with sensitive logic regions within a microelectronic device can change the state of the circuit [1]. When a collision event modifies the state of a memory bit or flip-flop, this is known as a soft error or a single event upset (SEU).

Protection against SEUs is commonly achieved through the use of radiation-hardened components. However, these components are expensive and lag several generations behind standard commercial components due to high development and

N.A. Harward (✉) • M.R. Gardiner • L.W. Hsiao • M.J. Wirthlin
Department of Electrical and Computer Engineering, NSF Center for High Performance Reconfigurable Computing (CHREC), Brigham Young University, Provo, UT, USA
e-mail: nateharward@ieee.org; mikegardiner@byu.edu; lukehsiao@byu.edu; wirthlin@byu.edu

F. Kastensmidt, P. Rech (eds.), *FPGAs and Parallel Architectures for Aerospace Applications*, DOI 10.1007/978-3-319-14352-1_5

testing costs and limited production volume [1]. Field Programmable Gate Arrays (FPGAs) provide a computing platform which is a suitable and flexible alternative to radiation-hardened computers. FPGA reconfigurability allows design upgrades and corrections after a space launch, and the same FPGA can be reused for new designs.

SRAM-based FPGAs use static random-access memory (SRAM) to hold the FPGA configuration and their SRAM is vulnerable to SEUs. A change to a configuration memory bit can affect the function of a look-up table (LUT) or the routing between nodes, and cause failure in the user design. One example of such failure is illustrated in Figs. 5.1, 5.2, 5.3 and 5.4. Figure 5.1 shows the configuration memory that defines a simple circuit within an FPGA and Figure 5.2 shows the routing and logic result of that memory as an AND gate with two inputs. Figure 5.3 shows an SEU routing one of the inputs away from the AND gate and Figure 5.4 shows an SEU changing the AND gate into an XOR gate. The configuration memory on an FPGA can be protected from SEUs with memory scrubbing and/or error detection and correction (EDAC) techniques [2]. FPGA fault-tolerant design techniques such as triple modular redundancy (TMR) can also be employed to detect and mitigate SEUs.

FPGA fault injection is an emulation-based method for discovering which of the configuration bits in a design are sensitive to upset. It can help identify specific system failure modes and determine design vulnerabilities. To determine which configuration bits are sensitive, each bit is changed one by one to emulate an SEU while the design outputs are compared with outputs from a golden model or set of expected outputs. Each changed bit is restored when the next bit is changed to emulate an SEU. When an output mismatch is observed, the fault injector logs the changed bit as a sensitive bit. FPGA fault injection does not completely evaluate the reliability of a design, as it does not test all FPGA components and hard logic. FPGA fault injection does not emulate single event transients (SETs) or multi-bit errors (MBUs).

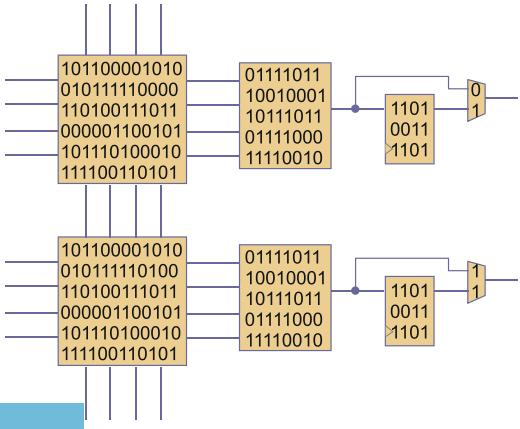


Fig. 5.1 Configuration memory

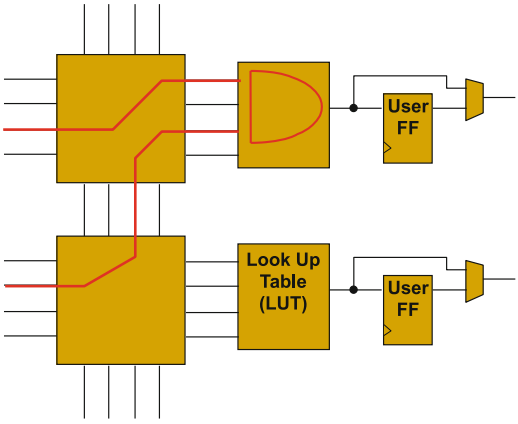


Fig. 5.2 Routing and logic result of configuration memory

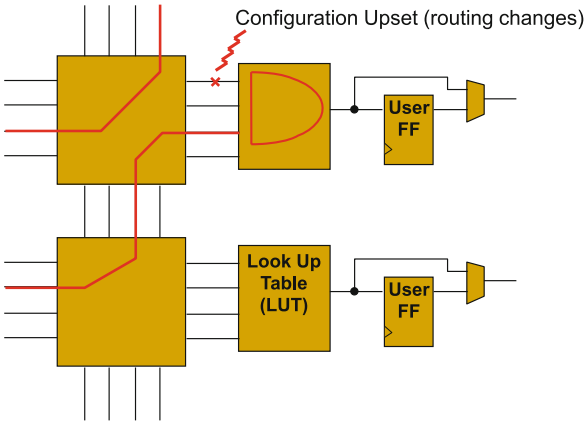


Fig. 5.3 Upset in routing

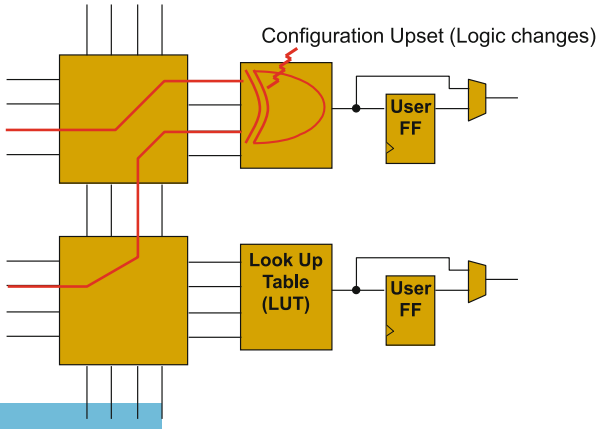


Fig. 5.4 Upset in logic

5.2 Related Works

The need for reliable FPGAs in space environments has motivated the development of FPGA fault injection platforms [2]. Over the years, many notable fault injection tools and platforms were created [3–5]. Johnson et al. used a SLAAC-1V testbed which housed three Virtex (XCV1000) FPGAs [6]. The SLAAC-1V injector was able to test all configuration bits at high speeds and predict where upsets can occur. Alderighi et al. [7] created the FLIPPER fault-injection platform which used a single Virtex-II Pro (XC2VP20) motherboard test fixture that could also be used for radiation tests. Rather than test all configuration bits, they used a probabilistic model to determine design sensitivity. Sterpone et al. [8] used a Virtex-II Pro (XCV2P30) FPGA with an embedded PowerPC microprocessor. Using an internal configuration access port (ICAP), a timing unit, and having the test design internal to the test FPGA, the fault injector operated at very high speeds.

Cieslewski et al. [9] used JTAG to improve fault injector portability with their Simple Portable FPGA Fault Injector (SPFFI). They have also compensated for the speed bottleneck of JTAG by designing SPFFI to only fault inject bits that are representative of a region of interest and/or fault inject random locations. Similar to the FLIPPER, they probabilistically determine design sensitivity. Guzman-Miranda et al. [10] have designed their FT-UNSHADES2 fault injection platform to obtain high-speed fault injection and full coverage. They used a standard Xilinx motherboard: the ML-510 with a Virtex-5 (XC5VFX130T). They can test custom-made daughtercards, which interface with the motherboard via PCI-Express. To maximize fault injection speed, FT-UNSHADES2 utilizes the SelectMAP interface. Their test design can work with a significant 512 bits of virtual input/output ports.

Starting with Virtex-6, Spartan-6, and 7-Series Xilinx FPGAs, Xilinx has releases a proprietary IP core called the Soft Error Mitigation (SEM) Core. The SEM Core is instantiated with the user design and uses the ICAP to detect, correct, and classify soft errors in the configuration memory of an FPGA device [11, 12]. While these fault injectors vary in technologies and methods used, they all have offered invaluable insight into how FPGA designs can be protected from SEUs.

5.3 XRTC Virtex-5 Fault Injector (XRTC-V5FI)

In conjunction with the Xilinx Radiation Test Consortium (XRTC), we built an FPGA fault injection system for testing digital FPGA circuits. Our main objectives in building this system were to achieve high customization and full bitstream coverage at a high fault injection rate. Because it takes a long time to complete fault injection on a full bitstream, we had to have a fast fault injector to increase the number of experiments completed. Also, a highly customizable system lets us conduct a larger variety of experiments and try different methodologies.

5.3.1 Architecture

The XRTC-V5FI fault injector (Fig. 5.5) is built using the XRTC motherboard, a test FPGA daughtercard, a non-volatile programmable read-only memory (PROM) card, and a host computer. The XRTC motherboard is also commonly used as a test fixture for radiation beam testing for other research projects. The test design is placed on the design under test (DUT) FPGA which is on the test daughtercard. The daughtercard allows us to run tests for both commercial and space-grade Virtex-5 FPGAs.

The XRTC motherboard has two service FPGAs (shown in Fig. 5.6) called the Configuration Monitor (ConfigMon) and the Functional Monitor (FuncMon). For our fault injection application, the ConfigMon performs scrubbing and readback and is responsible for configuring the DUT (pulsing PROG) and performing fault injection on the DUT (via SelectMAP), and logging sensitivity data for download. The FuncMon provides clock and reset signals, controls the fault injection sequence, compares design outputs, and signals the ConfigMon when an error occurs. The FuncMon and ConfigMon communicate directly with each other using a 16-bit wide Common Interconnect Bus (CI-Bus). The test design data is held on a PROM card plugged directly into the motherboard. This card contains the DUT golden bitstream file and the DUT mask bitstream file. The mask file is used to differentiate between the configuration bits used for logic, shift register LUTs (SRLs), and LUTRAM inside of configurable logic blocks (CLBs). The ConfigMon reads test design data from PROM card for fast configuration. The host PC computer communicates with both the ConfigMon and the FuncMon service FPGAs using RS-232

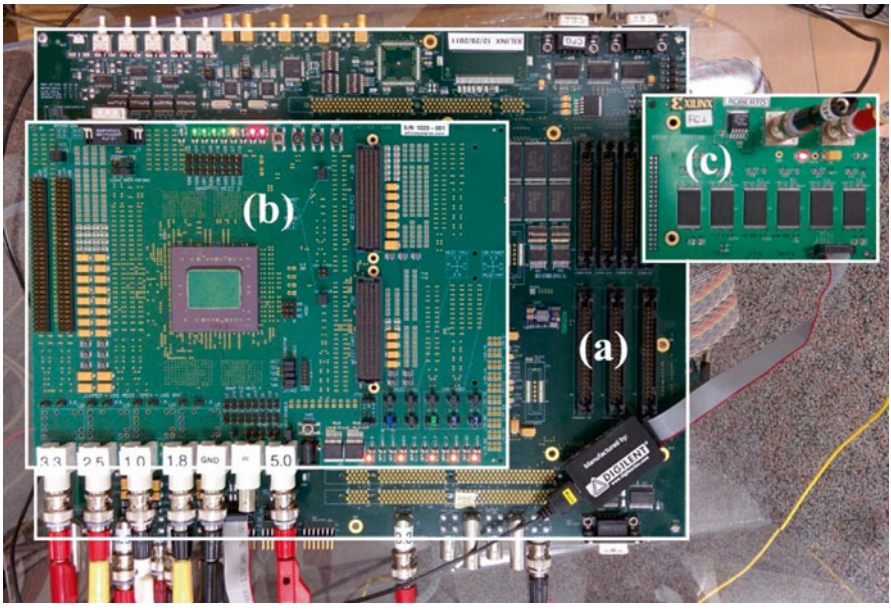


Fig. 5.5 Picture of (**a**) XRTC motherboard, (**b**) V5QV daughtercard, and (**c**) PROM memory card

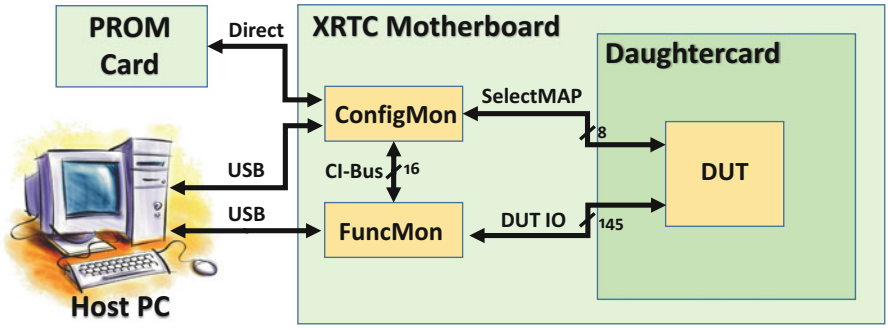


Fig. 5.6 High level view of XRTC-V5FI components

to initialize the system for fault injection, issue commands, and monitor the status and log data. The DUT FPGA receives its clock and reset signals from the FuncMon, and design outputs (145 signals) are sent from the DUT into the FuncMon for comparison. A high level illustration of the system is shown in Fig. 5.6.

The test FPGA for all the experiments described in this paper is the Virtex-5QV (V5QV). It is a 65-nm radiation-hardened by design (RHBD) FPGA manufactured by Xilinx, and it is qualified for space application [13]. The V5QV has 49,227,552 configuration bits, 34,087,072 of which are used for function and routing. There are also approximately 10.9 million bits used for block RAM (BRAM) and 4 million bits used for "testability and diagnostic reasons" [14]. For our experiments, we consider only the sensitivity of the bits used for function and routing.

5.3.2 *Attributes*

One major objective was to design our system to maximize fault injection speed. The current baseline time for a full bitstream fault injection campaign is 28 min. Design execution time and error recovery methods add additional time to the campaign. Each individual fault injection takes at least 49.1 μS. The ConfigMon configures and performs fault injection on the DUT via the SelectMAP port. The SelectMAP data port is 8-bits wide, and uses a 33 MHz clock. The XRTC-V5FI was designed to accurately measure configuration sensitivity by completely covering all 34.1 million configuration bits that control function and routing. The remaining 14.9 million bits in the bitstream are skipped.

Additionally, we have required that fault injection campaigns must be customizable. The FuncMon FPGA can be tailored for each design, allowing us to adjust the design execution time, test stimuli, fault injection procedure, and golden model. When comparing the design outputs, the FuncMon not only provides us with automatic error detection and recovery, but can also classify errors, determine faulty bit locations (e.g. a TMR voter error detection output), or other customizations based on the experiment. The host computer can request a snapshot of the faulty outputs if desired.

5.3.3 Methodology

Our experiments are built by placing two copies of the test design inside of the DUT FPGA. The outputs of each copy are assigned to 72 bits of the 145-bit signal that is outputted to the FuncMon. These outputs are then compared with each other at the end of a run cycle, and any mismatches are reported as errors. Alternatively, we could have had a golden model in the FuncMon and compared its outputs with a copy of the test design in the DUT, but we decided on the previous strategy to avoid any possible timing issues from comparing outputs from separate FPGAs.

Below is the fault injection loop procedure used for our experiments. This procedure is also shown with the diagram in Fig. 5.7.

1. The ConfigMon FPGA toggles the bit in the DUT FPGA's configuration memory.
2. The DUT is reset and its clock is enabled. The DUT is given time to load memories, execute software, and allow any errors to propagate through to its outputs.
3. The DUT's clock is stopped, and the outputs from both copies of the test design are compared with each other.
4. If an error is detected, the FuncMon signals the ConfigMon to record and log the error with the error's location and type.
   (a) For reset recovery experiments only, the configuration memory bit is restored and this process is repeated to determine if the error remained. The error is recorded as either recovered or unrecovered.
   (b) If a Single Event Functional Interrupt (SEFI) error (functional error independent of the test design) [15] is detected, the error is recorded, the DUT is fully reconfigured, and fault injection resumes at the next bit.

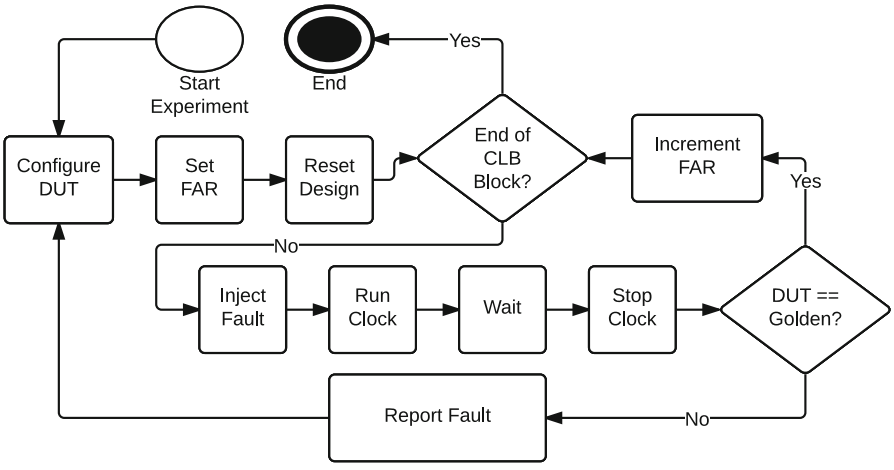


Fig. 5.7 Diagram showing the fault injection procedure

5. If the design contains a soft processor, we fully reconfigure the DUT after each detected output error to ensure the full recovery of memories.
6. The faulty bit is restored as the next bit is toggled.

At the beginning of a fault injection campaign, the host will reconfigure the service FPGAs and load the bitstream for the DUT FPGA onto the PROM card. The host will setup the ConfigMon with the correct parameters for fault injection, and test that the system is setup correctly. The host then commands the FuncMon to sequentially perform fault injection with a user-specified number of bits. The FuncMon will then run the fault injection procedure described above for each bit by issuing commands to the ConfigMon, waiting for the user-specified design execution time for each injected fault, and reporting results. The FuncMon reports the number of bits injected while the host ensures that errors are recovered and retrieves logged faults from the ConfigMon. The host keeps a database of errors with location and type, allowing for later analysis of the data.

5.4 Soft Processor Fault Injection

A soft processor is an implementation of a processor architecture that can be customized by the user for use on an FPGA. The key advantage soft processors offer to their users over standard microprocessors is the ability to optimize the hardware design for a particular application using FPGA resources. The reconfigurability of soft processors is also advantageous in that it allows the design to be updated whenever new features are desired, granting the processors relative immunity to obsolescence and enabling changes even when the FPGA has been deployed in a remote or harsh environment.

With a rise in the use of soft processors in harsh environments, a detailed understanding of soft processor reliability and failure modes is becoming indispensable. Using fault injection, we can test the configuration memory sensitivity of soft processors on FPGAs in an effort to understand their reliability and evaluate soft processor mitigation strategies and recovery methods. However, fault injection for soft processors involves grappling with a number of challenges unique to these designs. First, the reliability of a soft processor system depends not only on the specific hardware modules and features of the processor included in the system, but also on the software application the processor is executing. Since different software programs exercise a processor's functional units and memory in different ways, one software program may result in a different configuration memory sensitivity than another. A second challenge in soft processor fault injection is handling errors that propagate into memories. If an error from an injected fault propagates into a FPGA memory resource such as BRAM, LUTRAM, or an SRL, the error can persist in the memory even after a full system reset. Without special memory scrubbing or a full reconfiguration to repair the error, subsequent configuration bits may be deemed sensitive when a fault injection on a previous configuration bit was the real cause of the

error. A third challenge in conducting fault injection experiments on soft processors is choosing a design runtime long enough to ensure that any bootloader code has completed and the desired software application is executing while also choosing a runtime short enough to minimize overall test time.

5.4.1 Soft Processors Used

For our fault injection experiments, we have used two of the most popular soft processor models: the MicroBlaze soft processor from Xilinx [16] and the LEON3 soft processor from Aeroflex Gaisler [17]. These experiments were run using identical embedded software applications and similar soft processor configurations, although there are still significant differences between the processor architectures.

The MicroBlaze is a 32-bit reduced instruction set computer (RISC) soft processor proprietary to Xilinx, built and optimized for use solely on Xilinx FPGAs [16]. It has a full Harvard architecture with separate data and instruction memory buses. The MicroBlaze is highly customizable, and Xilinx has produced a large number of compatible IP modules and libraries to use with it.

The LEON3 is an open-source 32-bit RISC soft processor from Aeroflex Gaisler [18]. It is based on the SPARC V8 architecture and supports a variety of operating systems such as Linux, RTEMS, and VxWorks. A ROM peripheral provided with the processor is used to decompress an application program stored in the ROM and loads it into processor main memory when no debugger is used. The bootloader code which performs this function is generated automatically by the LEON3 software tools and is stored in the ROM along with the compressed application code. A fault-tolerant version of the LEON3, the LEON3-FT, is commercially available from Aeroflex Gaisler as well.

5.4.2 Soft Processor Test Designs

For both the MicroBlaze and the LEON3, version 13.2 of the Xilinx tool flow was used to generate a bitstream. A simple Towers of Hanoi C program was compiled and run on each platform. Neither processor used an operating system for this test. No FPUs, MMUs, debug modules, or caches were enabled. All program memory was stored in the standard BRAM peripherals that came with the IP libraries for each processor. The MicroBlaze used an 8 KB BRAM while the LEON3 used a 32 KB BRAM. The LEON3 also included an additional 15 KB ROM to hold its bootloader code and a compressed version of the Towers of Hanoi program, which is copied into the RAM on startup by the bootloader. Each design ran on a 50 MHz clock input (supplied by the FuncMon) and was given 16,921 clock cycles to load and execute code memory. For each experiment, full reconfiguration was used to

Table 5.1 Comparison of configuration features used for experiments

MicroBlaze	LEON3
Version 8.20.a	GRLIB Release 1.3.4-b4140
5 Stage Pipeline	7 Stage Pipeline
No Register Windows	8 Register Windows
32-bit Multiplier	32-bit Multiplier
No Divider	32-bit Divider
Barrel Shifter	No Barrel Shifter
Pattern Comparator	No Pattern Comparator
2 BRAMS	16 BRAMS
Data and instruction LMB buses	Single AHB Bus

recover from reported errors to restore memories. Table 5.1 highlights some of the differences between the two processor configurations.

The processor outputs selected for the comparison between the DUT and golden versions of each soft processor design were chosen from each processor's bus signals governing memory access. From these outputs, we can determine if the faults affect the processor state in terms of the executed instructions and the calculated results being saved to memory. This strategy does not cover all possible design errors and would need to be adjusted for designs that interact with peripherals or use very little data memory.

For the MicroBlaze design, we observe the lower 16 bits of the address and data lines for both the data memory (dlmb) and the instruction memory (ilmb). We also monitor the memory enable and write enable nets. For the LEON3, we observe similar signals within the AMBA High-Performance Bus (AHB) Master In (ahbmi) and Slave In (ahbsi) signals from the ahbmi signal we observe the full 32-bit read data line and a 2-bit transaction response signal coming in from the bus slaves. From the ahbsi signal we observe the full 32-bit write data line and the lowest 6 bits of the address line coming out from the processor, which is the bus master.

5.5 Test Results and Analysis

The soft processors are duplicated and placed on the DUT FPGA. Figure 5.8 shows the layout of the MicroBlaze and LEON3 designs that were generated using Xilinx FPGA Editor software. The LEON3 is a larger design, occupying 2.28× the number of slices that the MicroBlaze occupies. Experiments were conducted to test for raw sensitivity and reset-recoverability. Result data was analyzed to determine the normalized sensitivity of a design, to compare the sensitive bit set of the design with the essential bit set generated by the Xilinx tools, and to determine a design's configuration memory error rates.

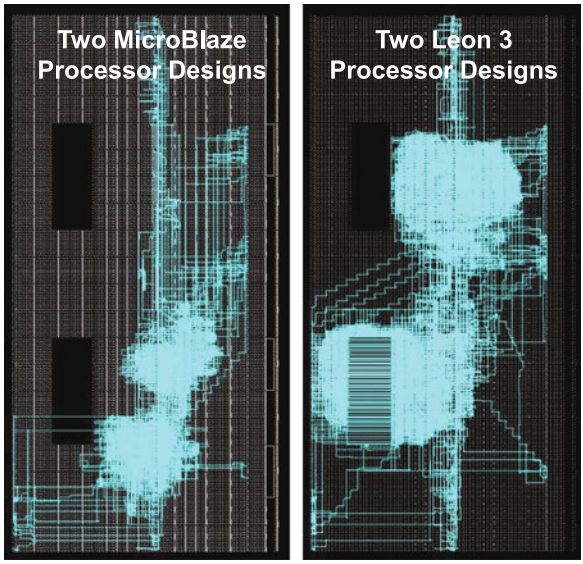


Fig. 5.8 A layout for visual comparison of MicroBlaze and LEON3 designs (Generated with Xilinx FPGA Editor)

Table 5.2 Resource utilization

Design	Slices	Total LUTs	LUTs as logic	LUTs as RAM	Registers	BRAMs
MicroBlaze	1,029	2,493	2,190	128	1,601	4
LEON3	2,354	6,919	6,789	24	2,803	32

Table 5.3 Raw sensitivity results

Design	Sensitive bits	Sensitivity (%)
MicroBlaze	103,893	0.305
LEON3	188,378	0.553

5.5.1 Raw and Normalized Sensitivity

The raw sensitivity and resource utilization numbers for the MicroBlaze and LEON3 test cases are given in Tables 5.2 and 5.3. The LEON3 is both a larger design and had a larger number of sensitive bits than the MicroBlaze. The per-processor sensitivity is 51,946 errors for the MicroBlaze and 94,189 errors for the LEON3 design.

To compare the normalized design sensitivity, we use the following equation:

$$Normalized\ Sensitivity = \frac{Sensitivity}{Utilization} = \frac{(Total\ Slices)(Sensitive\ Bits)}{(Total\ Bits)(Used\ Slices)} \quad (5.1)$$

The normalized sensitivity results are listed in Table 5.4. The normalized sensitivity of the MicroBlaze is 26 % greater than the normalized sensitivity of the

Table 5.4 Total errors normalized over resources utilized

Design	Errors per slice	Errors per logic LUT	Errors per register	Normalized sensitivity (%)
MicroBlaze	100.97	47.44	64.89	6.07
LEON3	80.02	27.75	67.21	4.81

Table 5.5 Sensitive bits that were not recoverable by reset

Design	Sensitive bits	Unrecovered errors
MicroBlaze	104,001	14,271 (13.72 %)
LEON3	188,653	440 (0.28 %)

LEON3. We believe that the higher sensitivity of the MicroBlaze is due to how the two processors are made. The MicroBlaze, by default, is optimized for Xilinx FPGAs and uses LUTRAM and SRL primitives [16]. The LEON3 is for the most part FPGA architecture-independent, except for the primitives it uses to construct its Input/Output Blocks (IOBs), clock management devices, and memories, which are chosen through generics in its HDL code. Because more of the LEON3 design is synthesized than the MicroBlaze, this could result in less functional density and thus less sensitivity to upsets.

The static results from V5QV Single Event Effect (SEE) testing give an error rate of five static upsets per year for this FPGA's configuration memory [14]. Using this error rate, we would estimate a uniprocessor MicroBlaze design to have a mean time to configuration-induced failure (MTTCIF) of 131.24 years in GEO, and a small uniprocessor LEON3 design to have a MTTCIF of 72.38 years. It is important to keep in mind that this error rate does not include BRAMs or other user memories, and it does not account for Digital Clock Managers (DCMs), DSP48Es, Multi-Gigabit Transceivers (MGTs), and other non-CLB elements.

5.5.2 Reset Recovery Experiment

A system-wide reset can be a simple recovery technique for FPGA designs, however it does not always allow recovery of soft processor designs. When errors propagate into design memories, they can persist after a system reset. The goal of the reset-recovery experiment is to identify which configuration bits cannot be recovered. This experiment requires an additional step in our fault injection procedure where the fault-injected bit is corrected, the test design is reset, and the design outputs are again checked for errors. Table 5.5 shows how many unrecovered errors were found in each design. In the MicroBlaze design, about 1 in 7 sensitive bits were not recoverable by reset. In the LEON3, 1 in 429 were not recoverable. The LEON3 has a much better reset-recovery rate than the MicroBlaze design. We believe this is due to the bootstrap loader sequence that the LEON3 uses. When the reset is asserted, the LEON3 in effect scrubs its own program memory.

5.6 Conclusion

We have injected five billion bits over thousands of hours of testing to develop a unique Virtex-5 fault injection system. The fault injector was created with the XRTC motherboard and used to test the MicroBlaze and LEON3 soft-processors. The system performs fault injection successively on all configuration bits that control FPGA function and routing at a speed of 49.1 μS per bit. Our initial soft processor test results were shown, as well as processor reset recovery data. We found that the LEON3 has a lower normalized sensitivity and a higher reset-recovery rate than the MicroBlaze.

Future work with the fault injection system will focus on using the system to conduct experiments on soft processor designs. Fault injection experiments of the ARM Cortex-M0 and OpenRISC soft processors are underway, and other soft processors will be considered. In addition to performing experiments to determine the raw sensitivity of these processors, we will implement SEU mitigation and recovery techniques into the processor designs of the fault injection system and evaluate the effectiveness of each of these techniques in reducing design sensitivity. Using the data gathered from these tests, we will create reliability estimation tools and develop a model for estimating soft processor configuration sensitivity. These tests and tools will enable engineers to more fully understand the reliability tradeoffs in the use of soft processors, speeding up the design process, and allowing engineers to more accurately predict soft processor reliability in a variety of harsh environments.

Acknowledgments This work was supported by the I/UCRC Program of the National Science Foundation under Grant No. 1265957. We also acknowledge the Xilinx Radiation Test Consortium (XRTC) and members for support and use of XRTC test hardware.

References

1. Dodd PE, Massengill LW (2003) Basic mechanisms and modeling of single-event upset in digital microelectronics. IEEE Trans Nucl Sci 50(3):583–602
2. De Kastensmidt LFG, Neuberger G, Hentschke RF, Carro L, de Reis LRA (2002) Designing fault-tolerant techniques for SRAM-based FPGAs. IEEE Des Test Comput 21(6):552–562. doi:10.1109/MDT.2004.85
3. Mansour W, Velazco R (2013) An automated SEU fault-injection method and tool for HDL-based designs. IEEE Trans Nucl Sci 60(4):2728–2733. doi:10.1109/TNS.2013.2267097
4. Nazar G, Carro L (2012) Fast single-FPGA fault injection platform. In: Defect and fault tolerance in VLSI and nanotechnology systems (DFT), 2012 IEEE international symposium on, pp 152–157. doi:10.1109/DFT.2012.6378216
5. Lima F, Carmichael C, Fabula J, Padovani R, Reis R (2001) A fault injection analysis of Virtex FPGA TMR design methodology. In: 6th European conference on radiation and Its effects on components and systems, IEEE (2001), pp 275–282. doi:10.1109/RADECS.2001.1159293
6. Johnson E, Wirthlin MJ, Caffrey M (2002) Single-event upset simulation on an FPGA. In: Proceedings of the international conference on engineering of reconfigurable systems and algorithms (ERSA), CSREA Press, 2002, pp 68–73

7. Alderighi M, Casini F, d'Angelo S, Mancini M, Pastore S, Sechi GR (2007) Evaluation of single event upset mitigation schemes for SRAM based FPGAs using the FLIPPER fault injection platform. In: Proceedings of the 22nd IEEE international symposium on defect and fault-tolerance in VLSI systems, DFT'07, IEEE Computer Society, Washington, DC, USA, pp 105–113. doi:10.1109/DFT.2007.45
8. Sterpone L, Violante M (2007) A new partial reconfiguration-based fault-injection system to evaluate SEU effects in SRAM-based FPGAs. IEEE Trans Nucl Sci 54(4):965–970. doi:10.1109/TNS.2007.904080
9. Cieslewski G, George AD (2009) SPFFI: Simple portable FPGA fault injector. In: Proceedings of military and aerospace programmable logic devices conference (MAPLD)
10. Guzmán-Miranda H, Nápoles J, Mogollón J, Barrientos J, Sanz L, Aguirre M (2012) FT-UNSHADES2: a platform for early evaluation of ASIC and FPGA dependability using partial reconfiguration. La Sociedad de Arquitectura y Tecnologa de Computadores
11. LogiCORE IP soft error mitigation controller (2011) UG764 (v2.1)
12. Schumacher P (2012) SEU emulation environment. WP414 (v1.0)
13. Wang Y (2011) Recommendations for managing the configuration of the RHBD Virtex-5QV. In: Proceedings of military and aerospace programmable logic devices (MAPLD)
14. Swift G, Carmichael C, Allen G, Madias G, Miller E, Monreal R et al (2011) Compendium of XRTC radiation results on all single-event effects observed in the Virtex-5QV. In: Proceedings of NASA military and aerospace programmable logic devices (MAPLD)
15. White D (2011) Considerations surrounding single event effects in FPGAs, ASICs, and processors. WP402 (v1.0.1)
16. MicroBlaze Processor Reference Guide, Embedded Development Kit EDK 13.2 (2011). UG081 (v13.2)
17. Aeroflex gaisler LEON3 processor. http://www.gaisler.com/index.php/products/processors/leon3
18. Learn MW (2011) Evaluation of soft-core processors on a Xilinx Virtex-5 field programmable gate array. Sandia National Laboratories, Sandia Report No. SAND2011-2733, Apr 2011

Chapter 6
A Power-Aware Adaptive FDIR Framework Using Heterogeneous System-on-Chip Modules

Shane T. Fleming, David B. Thomas, and Felix Winterstein

Abstract Reconfigurable field-programmable gate arrays (FPGAs) offer high processing rates at low power consumption and flexibility through reconfiguration which makes them widely-used devices in embedded systems today. Spacecrafts are highly constrained embedded systems with an increasing demand for high processing throughput. Hence, leveraging the power/energy efficiency and flexibility of reprogrammable FPGAs in space-borne processors is of great interest to the space sector. However, SRAM-based FPGAs in space applications are particularly susceptible to radiation effects as single event upsets (SEUs) in the configuration memory can cause the reconfiguration of the chip and an undesired modification of the circuit. Traditionally, this problem is addressed by fault detecting and *scrubbing*, *i.e.* repeated reprogramming of the configuration bitstream. A major disadvantage of this technique is the considerable down-time of the processing system during reprogramming which can lead to the loss of payload data or even affect critical onboard control tasks. This work proposes a novel fault detection, isolation and recovery (FDIR) framework that optimizes the worst case response, power consumption and availability of the processing system together. Our FDIR scheme and fault handling is transparent to the payload application as the system autonomously ensures nearly full availability of the payload processor at all times. A key feature of our technique is the explicit use of commercial-off-the-shelf heterogeneous systems such as Xilinx's Zynq or Altera's Cyclone V system-on-chip devices, which tightly couple FPGA fabric with embedded hard processor cores. This chapter describes the current implementation of our FDIR framework. We present experiment results obtained under fault injection and demonstrate that our framework ensures nearly full availability, whereas the conventional scrubbing approach can degrade to 20 %

S.T. Fleming (✉) • D.B. Thomas
Circuits and System Group, Imperial College London, London, UK
e-mail: shane.fleming06@imperial.ac.uk; d.thomas1@imperial.ac.uk

F. Winterstein
European Space Agency, Robert-Bosch-Straße 5, Darmstadt 64293, Germany
e-mail: felix.winterstein@esa.int

F. Kastensmidt, P. Rech (eds.), *FPGAs and Parallel Architectures for Aerospace Applications*, DOI 10.1007/978-3-319-14352-1_6

availability for high fault rates. An in-orbit demonstration and validation of the proposed technique will follow during an experiment campaign onboard OPS-SAT, a European Space Agency satellite mission set to launch in 2016.

Reconfigurable field-programmable gate arrays (FPGAs) offer high processing throughput at low power consumption and flexibility through reconfiguration which makes them widely-used devices in embedded systems with high processing demand today. However, SRAM-based FPGAs are particularly susceptible to the radiation effects of space applications because single event upsets (SEUs) in the configuration memory can cause a reconfiguration of the device and hence an undesired modification of the circuit. This problem is usually addressed by adding spatial redundancy, i.e. duplicating or triplicating the processing units in the FPGA fabric, in combination with scrubbing, i.e. reprogramming the configuration bitstream after a fault has been detected [1]. Repair via scrubbing potentially causes considerable down-time of the processing system [2] which can lead to the loss of payload data or affect onboard control tasks. This work addresses this issue and focuses on maximizing the availability of the onboard processing system. In particular, we consider the system availability as a third performance metric alongside the processing through-put and power/energy consumption.

This work describes a novel fault detection, isolation, and recovery (FDIR) strategy for onboard satellite payloads which utilizes commercial-off-the-shelf (COTS) reconfigurable FPGAs. Our scheme leverages heterogeneous systems-on-chips (SoCs), such as Xilinx's Zynq chip [3] or Altera's Cyclone V SoC modules [4], which comprise of a tightly coupled reconfigurable hardware and hard-wired processor cores. These SoCs embed one or multiple hard processor cores alongside programmable FPGA logic, enabling low latency and power-efficient communication between these two computing devices. Compared to the FPGA fabric, we consider the hard-wired cores to be more reliable with respect to radiation-induced SEUs for which standard fault mitigation techniques can be applied; for this reason these cores are assigned the role of a hypervisor being in charge of fault detection and scrubbing in the fabric. We define Quality-of-Service (QoS) as the rate at which payload data is processed, for example this could be the frame or pixel rate of an image processing application. A novelty of the proposed concept is that the adaptive framework aims to maintain a constant processing rate of the QoS application during repair of the FPGA configuration memory. To this end, the system is rolled back to the last known acceptable state and the hardware task is migrated to software running on a hard processor core when a fault is detected in the hardware. The task is then continued in a software thread while the FPGA device is reprogrammed, and once scrubbing is completed the task is migrated back to hardware.

Our rollback and repair can be very effectively applied to applications where its internal state is regularly reset to a known state, such as image processing applications. This reset interval also sets the amount of rollback time required since the computation will have to start over from the last point where the state was reset. The work presented in this chapter contains a case study with a HW/SW application that performs K-means clustering on frames of a video. In this example processing each video frame takes a

variable amount of time and may require multiple iterations of the algorithm, which means that the reset interval can vary depending on how long each frame takes to process. Many common onboard processing tasks stem from signal and image processing [1, 5–7] or data compression [6, 8, 9] applications which are normally stream-based. This type of applications have therefore been the primary focus of our work to date.

Besides the optimization of system availability, our FDIR framework also ensures that constraints on the power consumption are maintained. Such constraints could possibly become violated when the payload task is migrated to software and the core frequency is ramped up in order to meet the QoS requirement. The adaptive frequency scaling will use online power monitoring to dynamically trade-off and manage the QoS and power constraints during system runtime.

This chapter describes our FDIR framework implemented on a Xilinx Zynq device and presents experiment results obtained under fault injection. This work represents a precursor system for a subsequent implementation in the payload computer on-board the OPS-SAT satellite [10]. OPS-SAT is a nano-satellite which is devoted to demonstrating novel mission concepts that arise when more powerful computers are available on satellites. The OPS-SAT mission is led by the European Space Agency (ESA) and is set to launch in 2016. OPS-SAT is the first spacecraft that flies COTS Altera Cyclone V SoCs built on 28 nm technology. The hardware is not space qualified and hence our experiment setup focuses on very high SEU-induced fault rates. In summary, our contributions are:

- We present a novel FDIR scheme for FPGA-based space-borne processors. The scheme autonomously migrates processing tasks between the reprogrammable logic and hard processor cores, so as to maximize the availability of the processing system in the presence of SEU-induced faults.
- We extend the FDIR framework by utilizing frequency scaling to create an adaptive, fine-grain optimization of power consumption and processing throughput.
- We present measurements of the system availability, power consumption and processing throughput for different fault rates. We demonstrate that our technique maintains nearly full availability even under harsh conditions where faults occur as frequently as up to once per second.
- We compare our results to ‘traditional’ fault handling approaches based on fault detection and scrubbing.

Section 6.1 discusses related work and highlights the differences to previous work. Section 6.2 describes our adaptive FDIR framework. Section 6.3 briefly outlines our benchmark application. We present experiments in Sects. 6.5 and 6.6 concludes the paper.

6.1 Related Work

Repairing the configuration memory of reconfigurable SRAM-based FPGAs in high-radiation environments is usually based on scrubbing, *i.e.* periodic rewriting of the FPGA configuration memory, while faults in the form of radiation-induced bit

flips in the configuration memory are detected by including redundant modules and comparators or majority voters in the circuit. A common approach is to use *triple modular redundancy* (TMR) at the netlist level in combination with periodic scrubbing of the entire configuration memory [11]. The drawbacks of this strategy are large overheads in terms of chip area and power consumption, and long scrubbing times, especially for large COTS FPGAs. In addition to *blind scrubbing* at a fixed rate, the time spent for repair can be reduced by *triggered scrubbing* which is performed only if a fault has been detected.

Instead of adding spatial redundancy at netlist level, alternative approaches implement a module replication at coarse-grained unit level. Azambuja *et al*. [12] describe an approach where faulty modules are detected with unit-level TMR and repaired with selective partial scrubbing using dynamic partial reconfiguration (DPR). Their approach is notable in that it further reduces the scrubbing time and energy spent in the repair process compared to the netlist-level TMR approach [11] while keeping the resource overhead similar. Nazar *et al*. [2] propose an alternative approach to reduce the scrubbing time by leveraging DPR and applying the repair only to *critical configuration bits* that are used by the logic configuration and by determining the optimal starting point for the scrubbing process.

Several recent approaches address a reduction of the resource overhead in terms of chip area and power consumption caused by the redundancy scheme, in particular TMR. Jacobs et al. [13] propose a framework that, instead of using TMR by default, can adapt the amount of redundancy needed according to the degree of required protection and changing failure rates using DPR. The authors integrate three redundancy schemes in their framework: TMR with voting, self-checking pairs (module duplication with comparison, DWC [14]), and a high-performance mode without module-level replication. Siegle *et al*. [1] present a comprehensive framework that allows the designer to select and analyze different redundancy and repair schemes: netlist-level TMR with scrubbing, no redundancy, duplication with comparison, and module-level TMR with partial scrubbing to speed up repair. In line with this work they focus on maximizing the availability of the processing system. However, we completely abandon the expensive TMR approach and propose a reliable onboard processor based on the more economic DWC strategy which involves hard on-chip processor cores in addition to SRAM-configurable logic.

Ilias *et al*. [7] propose an FDIR strategy which is similar to this work in that it uses DWC and migrates the processing task to embedded hard PowerPCs during scrubbing. The hard processor core has a smaller cross section and is less susceptible to radiation-induced faults than the reconfigurable logic. The authors demonstrate their framework with a finite impulse response (FIR) filter application and report a 40 % area reduction compared to a standard TMR implementation at the cost of a reduction in the processing throughput. This work builds on the same basic idea, but we extend the approach to address the drawback of a drop in processing rate by adding an adaptive frequency scaling. The adaptive availability optimization is a distinguishing feature of the proposed technique compared to the FDIR approaches discussed above. Additionally, we include a fine-grain online optimization of the power consumption.

Re-synchronization of state-dependent logic after repair is a crucial task in the fault mitigation strategies discussed above. We choose a checkpoint and rollback approach [15] where the system is rolled back to the last known acceptable state before the task migration. This approach works particularly well for processing tasks that can be split into small independent chunks, such as stream-based processing tasks. The hardware/software task migration and slicing of the processing task is more difficult to implement for other types of applications which exhibit many dependencies between the processed data items. However, many typical onboard processing tasks, such as image of signal processing applications, are stream-based which makes this approach applicable to a wide range of onboard processing applications.

6.2 A Workload-Adaptive FDIR Framework

Our management system is divided into two main components: the fault recovery system (FRS), used to manage the repair of the system once a fault has been detected; and the adaptive management system (AMS), which dynamically monitors the processing progress of the system and scales the performance while still meeting power constraints. The AMS is built upon previous work known as the Heterogeneous Heartbeats which is a framework for adaptive reconfigurable SoCs and is discussed in subsection 6.2.2. This section discusses the implementation details of the FRS, then outlines the Heterogeneous Heartbeats framework, and finally discusses the details of the adaptation management system.

6.2.1 Fault Recovery Management System

The goal of the fault recovery system (FRS) is to detect and recover from errors that arise in the system's hardware task. Figure 6.1 shows a flowchart of both the recovery process and task execution indicating whether each stage is executed in the hard processor system (PS) or within the programmable logic (PL). To demonstrate this an image processing case study is presented, where frames of an input video are iteratively processed. At the start of each iteration the PS is responsible for capturing the frame data and storing it into memory accessible by both the PS and the PL. It then checks to see if PL is fully configured and that the HW task and its duplicate are available. If available, they are sent a signal indicating that the input frame is present in memory and that they can start processing; however, if not then a software version of the task is started instead.

Both the hardware task and its duplicate process their data in lock step, and every time they complete a frame their outputs are compared. If there is no difference in the output then we assume that no error has occurred and the process continues as normal. However if a difference is detected between the outputs of the hardware

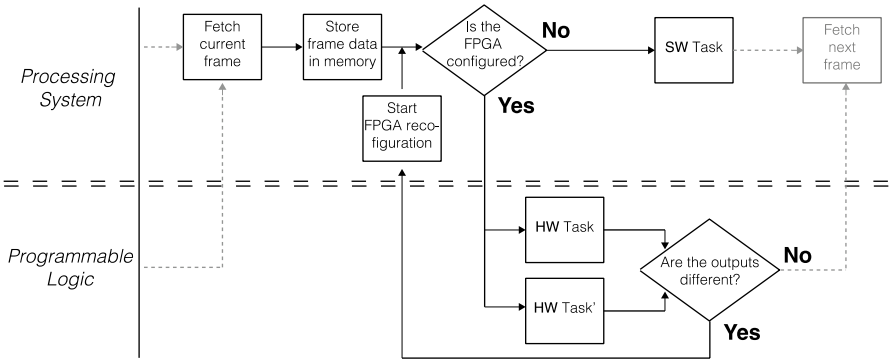


Fig. 6.1 Flowchart of the FDIR scrubbing system

tasks then we assume that an error has occurred and an exception is thrown. This exception triggers the FRS, running on the PS, to start reconfiguring the FPGA fabric. While the reconfiguration process is occurring the same input frame is then recomputed, however this time a software instance of the task is used instead of a hardware instance. Once the frame has been successfully processed the computation of the next frame is started in software until the hardware has been reprogrammed and validated.

6.2.2 Heterogeneous Heartbeats

Heterogeneous Heartbeats is the basis for the satellites adaptive management system. It aims to facilitate chip level adaptation, focusing on systems contained within a single package, such as the Altera SoC or Xilinx Zynq devices. The Heterogeneous Heartbeats framework is an enhanced version of preliminary work presented in [16]. In the development of such systems it is becoming increasingly common to use large collections of intellectual property (IP) packages, all with different characteristics from different locations, such as IP vendors or generated via high-level synthesis (HLS) tools. As the amount and variety of IP increases the interactions between sub components become increasingly complex potentially increasing the run time dynamics of the system. These dynamics make it difficult to statically optimize and tune parameters to meet constraints such as temperature, power, or frame rate offline and necessitates the need for online adaptive approaches. Heterogeneous Heartbeats extends the Heartbeats Application Programming Interface (API) [17], a standardized interface to monitor task progress, by allowing the seamless addition of both hardware (FPGA) resident heartbeat producers and heartbeat consumer.

The Heterogeneous Heartbeats framework considers three separate portions: sensors, adaptive engines, and actuators. Sensors collect data on the current state of the system, examples could be the applications progress via the Heartbeats API or a

devices power consumption; these are heartbeat producers. Adaptive engines use the collected sensor data and predictions on how changes in the system will alter future sensor readings to make decisions on how the system should alter its behavior; these are heartbeat consumers. Actuators change the behavior of the system, examples could be the frequency multiplier value in the phase locked loop (PLL) for the systems clock or the cache replacement policies.

The Heartbeats API is used as the basis for the interaction between the heartbeat producers (sensors) and the heartbeat consumers (adaptive engines). Application developers use the Heartbeats API by first calling an initialization function at the start of their application. This function sets up a publicly available heartbeat record that can be generated and accessed by either software or hardware, where individual heartbeat entries are stored and the goals of the application are set. The goals of the application are expressed in terms of the sensors that the application is interested in. For example in a video processing application one goal might be to maintain a particular frame rate and power consumption, so this would require the availability of a timer and a power monitor on the sensor side.

A heartbeat function is then called at important milestones of the applications progress. This function is used to create a sensor stamped heartbeat which are then saved as an entry in the publicly available heartbeat record. In our image processing example this would mean that the sensor stamps would be a timestamp from an internal or external system clock, and a power stamp from a power monitoring unit. Further operations are then provided for external heartbeat consumer applications to query the heartbeat record. These functions perform operations such as, fetch the current heartrate, fetch the history of the last n heartbeats, fetch the average heartrate, and fetch applications goals.

On the other side of the adaptive engines is the actuator portion. These are methods that cause changes in the systems behavior. Examples are the frequency of the PS or hardware tasks in the PL, the cache replacement policy, or what version of a particular algorithm is running.

6.2.3 Adaptation Management System

The adaptive management system (AMS) dynamically tries to maintain an overall system goal while subject to certain constraints. In this particular case, the goal of the AMS is to maintain a particular QoS deadline (frame rate) while using as little power as possible and always ensuring that a system-wide power constraint is met. This adaptation needs to be performed in two cases: as the application workload varies, and while the FRS is repairing faults in the system. Figure 6.2 shows both the controller's architecture in (a) and (b), and algorithmic flow in (c). In (a) and (b) we can see that the deviation of the application's ideal heartrate and the current heartrate are turned into an error signal which is used to drive the controller. This is the signal that the controller will attempt to minimize in the presence of disturbances.

When the system is being repaired due to faults the structure of system dramatically changes along with its behavior. In control theory an approach known

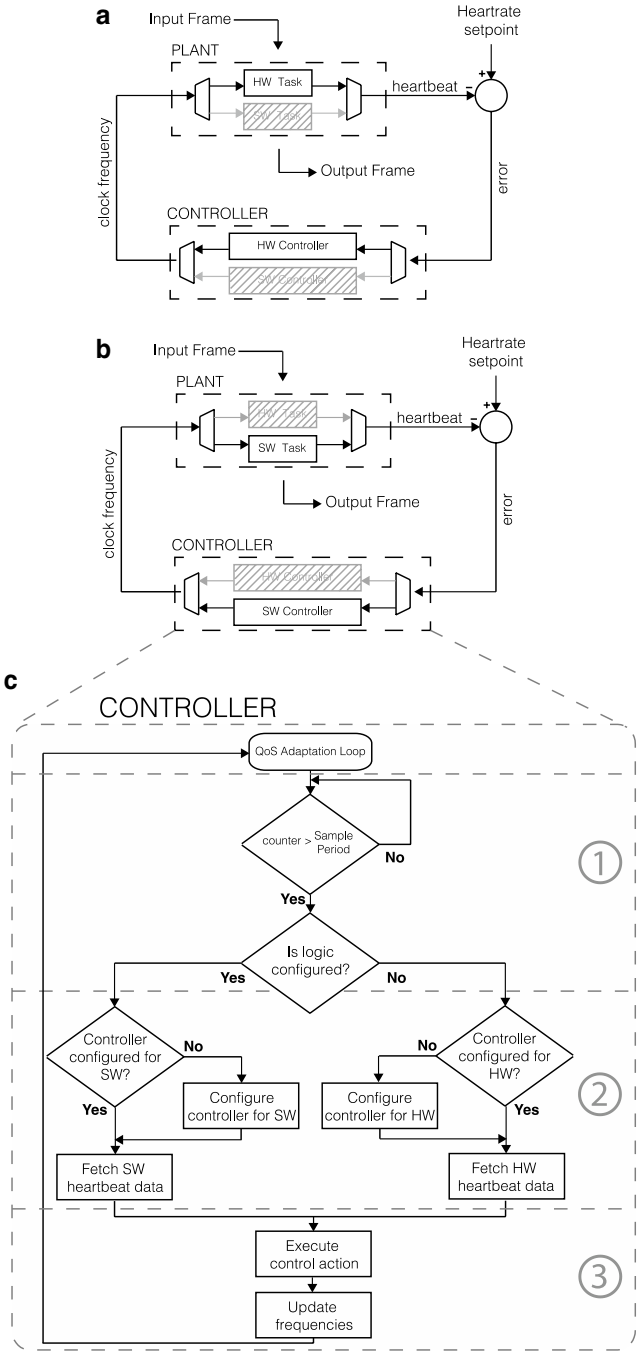


Fig. 6.2 Diagram to show the QoS adaptation controller setup with; (*a*) architecture view where the controller is configured to scale the clock frequency of FPGA resident hardware tasks during normal, error free operation; (*b*) architecture view where the controller is configured to scale the hard Processor Systems (PS) clock frequency when errors are detected in hardware; and (*c*) algorithmic view showing the flow of the execution and reconfiguration of the controller

as gain scheduling, where a suitable linear controller is selected depending on the current operating region of the controller, is used to handle such non-linear effects. We adopt a similar approach here, adapting the controller and scheduling different models and parameters based on the current configuration of the system. The process of adapting the controller can be divided up into three coarse stages, labeled in Fig. 6.2c, below is a description of each stage.

1. Initially the control algorithm determines whether the application is currently running within hardware or software.
2. Based on this information the parameters and state of the controller are scheduled, the parameters and control models are selected depending on whether the application is resident in hardware or software.
3. Finally the control action is executed and a new frequency is calculated. This is then used to update the clock controllers for both the software and the hardware systems.

This preliminary work uses a simple heuristic to control the various components of the system; however work is underway to develop a more sophisticated controller where each control action consists of the following stages. Firstly, a learning algorithm takes the error signal and generates a performance scaling factor, which is the multiple of the current heartbeat that we require in the future to maintain our QoS. Secondly, this is fed into a model that determines the frequency required to achieve the required increase or decrease in performance. Thirdly, the new frequency value is fed into another model that is used to determine the predicted power consumption that the change in frequency will cause. Finally, this predicted power consumption is used along with the current power consumption to determine if the power constraint will be satisfied. If the power constraint is not satisfied then the controller will iteratively search to find the next highest frequency value that will give the best performance, while still meeting the constraint.

6.3 Benchmark Applications

We demonstrate the adaptive FDIR system using a benchmark application which processes image data from the high-resolution camera onboard the satellite. A software implementation and an FPGA implementation are uploaded onto the SoC, while the FDIR system automatically schedules the execution of either the software or the hardware task. The following briefly describes our benchmark application.

6.3.1 K-Means Clustering

A common remote sensing application is the creation of maps of vegetation type or land cover, for instance used in crops/forestation monitoring such as the objective of Planet Labs dove fleet [18]. A central component of these image processing systems

is the unsupervised classification of image data based on the pixel values. *K*-means clustering is among the most popular machine learning techniques for assigning observation (in this case pixels) to classes (clusters). Clustering is also often used for analyzing multi- and hyperspectral imagery. *K*-means algorithms partition the *D*-dimensional point set $X = \{x_1, \ldots, x_N\}$ into clusters $\{S_1, \ldots, S_k\}$ where k is provided as a parameter. The goal is to find the optimal partitioning which minimizes the objective function given in (6.1) where μ_i is the geometric center (centroid) of S_i.

$$J\left(\{S_i\}\right) = \sum_{i=1}^{K} \sum_{x_j \in S_i} \left\| x_j - \mu_i^2 \right\| \tag{6.1}$$

Finding optimal solutions to this problem is NP-hard [19]. A popular heuristic version, known as Lloyd's Algorithm uses an iterative refinement scheme which, for every data point, computes the nearest cluster center based on the smallest squared Euclidean distance to it and then updates each cluster center position according to the data points assigned to it.

Clustering produces an output image with each pixel assigned to a cluster. Apart from classification, clustering provides a locally optimal solution to color quantization which results in a reduction of the data volume as a pixel can be represented with $log_2(K)$ bits in the new image. Remote sensing systems including a cluster analysis of satellite imagery usually perform the clustering offline after reception of the original image by the ground station. In this experiment we perform the clustering step onboard and benefit from the data volume reduction prior to downlinking telemetry. We use a software implementation of Lloyd's algorithm for *K*-means clustering in C++ and an FPGA implementation in VHDL which builds on the work described in [20].

6.4 Experiments

Our measurements focus on three sub-experiments:

- A 'naive' fault recovery mechanism where the task only runs in hardware and the entire system is stalled while the hard processing system performs scrubbing of the fabric's configuration memory. This is the traditional approach to fault mitigation for FPGAs and is used as the base line comparison with the second and third experiment where the task is automatically migrated to software so as to maintain the availability of the payload processor.
- The second experiment then includes the use of the FRS to migrate the task from software to hardware while the recovery process is taking place, demonstrating that the availability of the system can be improved through the use of the heterogeneous platform. We compare the FRS-based fault handling on the basis of system availability and processed blocks per time interval.

- Finally, we combine the FRS with the AMS to automatically manage the QoS deadline and power constraints and to demonstrate that fault tolerance is achieved while maintaining a particular QoS via frequency scaling. The AMS controller monitors the instantaneous power consumption and adapts the frequency according to the allowable power budget.

For the in-orbit experiment, all three sub-experiments are packaged up in a single image which is uploaded onto the onboard SoC. A thread running on the PS is in charge of scheduling the three experiment phases. The payload processor requires access to a high-resolution camera in order to retrieve input data for the image processing benchmark applications. In addition to the processed image data, the experiment setup collects downlinks information about the system availability (i.e. down-time during scrubbing and violation of QoS deadlines), the number and time stamps of faults that occurred, the selected frequency scalar values, and the power consumption (drawn from online power monitoring sensors) as well as several status indicators such as the presence of permanent circuit failures (e.g. due to latch-ups in the reconfigurable logic) and fault statistics.

6.4.1 Prototype Test Setup

The prototype system used for the measurements presented in this paper was developed on a Xilinx ZC702 Zynq development board, a very similar device to the Altera Cyclone V SoC used in the payload onboard OPS-SAT. Like the Cyclone V SoC this device contains a dual core ARM processing system (PS) tightly coupled to an FPGA fabric (PL) in a single package. Two identical K-means clustering IP cores were implemented using the Xilinx Vivado HLS high-level synthesis tool. The clustering cores are placed in the PL and connected to PS via various AXI busses. The output of the identical clustering cores are compared and an error flagged if their outputs do not match. In order to configure and control the IP cores from the PS their AXI locations were memory mapped and Linux drivers were developed. A configuration bit stream was then generated for use in both the initial configuration of the device, and for reprogramming the device during repair.

Petalinux, developed by Xilinx, is the Linux kernel running within the PS and on top of this OpenCV is used to manage the image data sent to the device and check the for errors in the output. Getting data from the PS to the clustering cores required a portion of the DDR memory to be reserved, input frames were obtained in OpenCV and were then passed to this reserved memory. AXI masters within the clustering cores were then used to fetch the input frame without any intervention from the PS, the same AXI masters were then used to send the output to different reserved memory location that can be read by the OpenCV application. In order to reconfigure the hardware during repair drivers provided by Xilinx allowed us to write the bitstream to a device file *xdevcfg*, which connects to the PCAP and allows us to reconfigure the PL from within the embedded Linux environment.

For the current implementation of the FRS & AMS system a preliminary frequency controller is used where we distinguish between two states: the state where the task has been migrated to the PS and the state where the task is running in the PL . Each state is given a high frequency and a low frequency respectively. Future work plans to explore more sophisticated controllers that scale the frequency of both hardware and software separately in order to meet constraints. To change the frequency of the PS the System Level Control Registers (SLCR) was used, where the clock multiplier values for the PS clocks PLL can be edited.

6.4.2 *Experiment Setting*

For each experiment stock ESA/NASA footage of Earth from a low earth orbit was streamed into the Zynq device over a network connection. OpenCV then sent the frames of this video to the reserved memory where the hardware could access them. In order to emulate SEU-induced faults, a separate software thread running on the PS randomly injects faults by corrupting a bit from one of the hardware blocks' output. This causes a discrepancy between the outputs of the two cores which causes the fault detection to trigger initiating the repair process. The application is instrumented with the heartbeats API, a power monitor, and timers and each of the three fault handling methods above is tested for a certain number of frames at a various error rates. For each experiment three metrics are evaluated:

- The availability AV of the system is defined as

$$AV = \frac{t_{up}}{t_{up} + t_{down}} \tag{6.2}$$

- where t_{up} is the amount of time that the systems objective task is active, and t_{down} is the time spent during repair after a fault occurred.
- The heartrate of the system which is generated using the Heterogeneous Heart-beats API and represents the processing performance. In this case we use the heartrate to measure the QoS of the system in terms of image blocks processed per second.
- The power consumption of the system which is measured using the inbuilt ZC702 development board power monitoring, allowing us to measure the power consumed from the PS and PL portions of the device separately.

6.5 Results

In Fig. 6.3 we examine the availability of the two recovery systems under different fault rates. Firstly, we observe that the average availability of the naive system, where the entire system stalls during repair, is always lower than the availability of

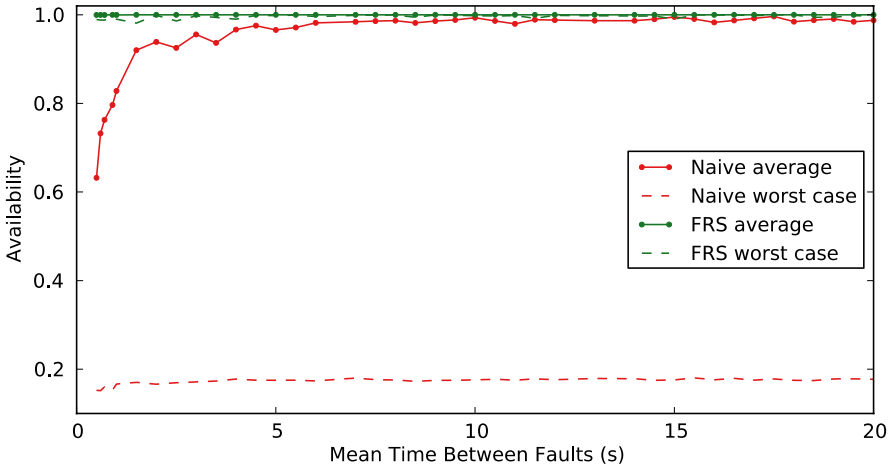


Fig. 6.3 Average and worst case availability of the conventional fault recovery (naive), task migration (FRS), and combined task migration and adaptive frequency scaling (FRS & AMS) systems

the FRS system where the task is migrated to the PS when a fault occurs. At high error rates the gap between the average FRS availability and that of the conventional approach becomes increasingly large. This runaway effect is due to an increased probability of errors occurring during the repair process causing the system to make less and less progress. This problem potentially becomes increasingly significant as the recovery time increases due to larger configuration memories in larger FPGA devices, increasing the need for more intelligent scrubbing techniques that make use of partial reconfiguration as pointed out in [1].

Secondly, we show the worst case availability of the different recovery systems. Critical onboard processing tasks can require hard task completion deadlines and hence we consider the worst case metric as ultimately more important for mission criticality. It can be seen from the graph that for the naive implementation the worst case availability deteriorates to values below 0.2, while with our FRS the worst case availability remains at nearly 100 % and is never lower than the average availability of the naive case.

Figure 6.4 shows how the worst case performance of the different systems change as the error rate is increased. In a similar fashion to the availability, we can see that the worst case heartrate for the naive recovery method performs poorly, the FRS improves this by ensuring that a certain level of QoS can always be maintained no matter what the current error rate is. However, maintaining this level of QoS is not free: In Fig. 6.5 we observe an increased power consumption of the FRS. This increased power consumption can be significantly improved through combining the FRS with the AMS which is used to scale the frequency of the PS only when it is required because a task has been assigned to it. Figure 6.4 also shows that, when we combine the FRS with the AMS, we are able to obtain a higher worst case heartrate than using the FRS alone, and Fig. 6.5 demonstrates that there is a comparable power consumption to the naive implementation and significant saving over using the FRS alone.

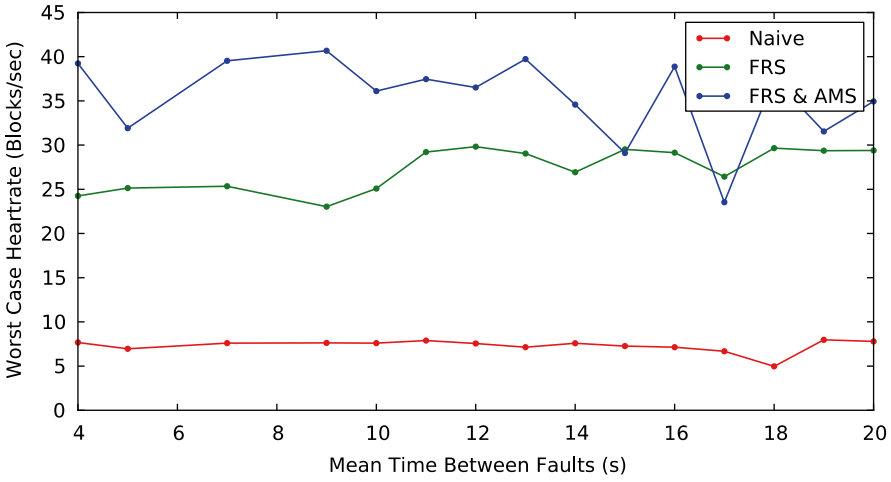


Fig. 6.4 Average heartrate (QoS) achieved by the conventional fault recovery (naive), task migration (FRS), and combined task migration and adaptive frequency scaling (FRS & AMS) systems

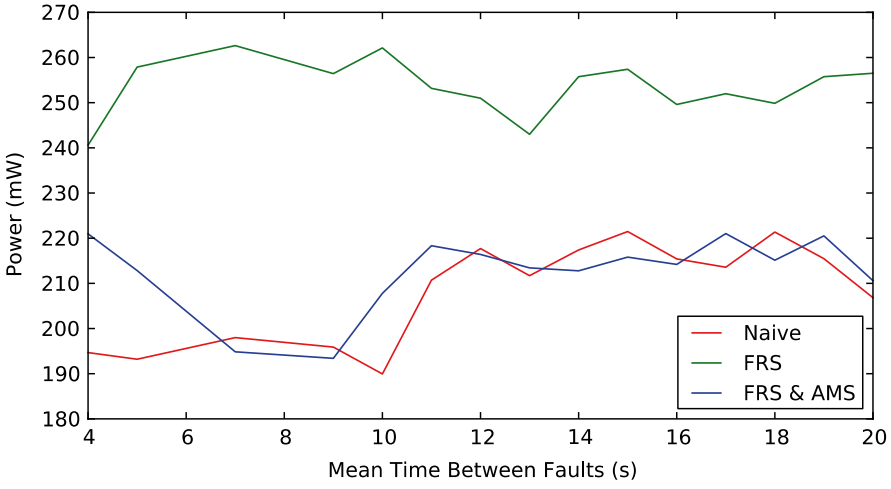


Fig. 6.5 Instantaneous power consumption of the conventional fault recovery (naive), FRS, and FRS & AMS systems

6.6 Conclusion and Outlook

We present a prototype implementation of a novel FDIR scheme for FPGA-based space-borne processors that will undergo an in-orbit test and validation campaign onboard the ESA OPS-SAT satellite, set to launch in 2016. A distinguishing feature of our technique is the autonomous migration of processing tasks between the

reprogrammable logic and hard processor cores in heterogeneous SoCs, which maximizes the availability of the processing system in the presence of SEU-induced faults and required scrubbing of the programmable logic. Our measurement results show that, under all conditions, our task migration technique maintains nearly full availability of the processor at all times. We compare the availability results to the mean and worst-case availability of a conventional fault detection and repair approach which is degraded to 20 % in the worst case in scenarios with frequently occurring SEU-induced faults. In addition, our FDIR framework features frequency scaling for an adaptive, fine-grain optimization of power consumption and processing throughput.

There are three major directions we plan to explore in future work. Firstly, we will investigate more sophisticated controller implementations for the adaptive power and throughput management. Our current prototype implementation switches between high and low clock frequency according to the current state of the task migration. More complex controllers provide a better quality of the frequency adaptation at the expense of an increased inherent power and resource consumption, and we plan to explore this trade-off in future work.

Our task migration effectively mitigates the degradation of the system availability during FPGA repair. However, high-throughput applications may still experience a significant drop in heartrate when the task is migrated to software, which is especially true as the recovery time increases due to larger FPGA devices being repaired. Partial reconfiguration combined with a more fine-grain error detection and localization will lead to faster recovery times. Hence, we plan to integrate partial reconfiguration into our FDIR framework.

A third aspect of future work is to maintain the operability of the onboard processor in the presence permanent faults caused by radiation-induced latch-ups by leveraging the reprogrammability of SRAM-based FPGAs. Over time parts of the FPGA configuration memory may become permanently damaged, especially when COTS FPGAs are used in long-lasting missions. We plan to address this issue by storing multiple pre-mapped copies of the same circuit which will allow us to ‘re-place’ the circuit around the damaged area.

References

1. Siegle F, Vladimirova T, Emam O, Ilstad J (2013) Adaptive FDIR framework for payload data processing systems using reconfigurable FPGAs. In: Proceedings of the NASA/ESA conference on adaptive hardware and systems (AHS), Torino, June 2013, pp 15–22
2. Nazar G, Santos L, Carro L (2013) Accelerated FPGA repair through shifted scrubbing. In: Proceedings of the 23rd international conference on field programmable logic and applications (FPL), Sept 2013, pp 1–6
3. Xilinx Zynq-7000 All Programmable SoC. http://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html
4. Altera Cyclone V SoC Overview. http://www.altera.com/devices/processor/soc-fpga/cyclone-v-soc/overview/cyclone-v-soc-overview.html

5. Yuhaniz S, Vladimirova T, Sweeting M (2005) Embedded intelligent imaging on-board small satellites. In: Srikanthan T, Xue J, Chang C-H (eds) Advances in computer systems architecture, ser. Lecture notes in computer science, vol 3740. Springer, Berlin, pp 90–103
6. Trautner R, (2011) ESA's roadmap for next generation payload data processors. In: Proceedings of the international space system engineering conference (DASIA), Malta, May 2011, pp 1–5
7. Ilias A, Papadimitriou K, Dollas A (2010) Combining duplication, partial reconfiguration and software for on-line error diagnosis and recovery in SRAM-based FPGAs. In: Proceedings of the 18th IEEE annual international symposium on field-programmable custom computing machines (FCCM), May 2010, pp 73–76
8. Yeh P-S, Armbruster P, Kiely A, Masschelein B, Moury G, Schaefer C, Thiebaut C (2005) The new CCSDS image compression recommendation. In: Proceedings of the IEEE aerospace conference, March 2005, pp 4138–4145
9. Cabral M, Trautner R, Vitulli R, Monteleone C (2010) Efficient data compression for spacecraft including planetary probes. IPPW-7 S/C incl
10. OPS-SAT ESA Operations. http://www.esa.int/OurActivities/Operations/OPS-SAT
11. Carmichael C, Tseng C (2009) Correcting single-event upsets in Virtex-4 FPGA configuration memory. Xilinx Inc., Tech. Rep., Oct 2009
12. Azambuja J, Sousa F, Rosa L, Kastensmidt F (2009) Evaluating large grain TMR and selective partial reconfiguration for soft error mitigation in SRAM-based FPGAs. In: Proceedings of the 15th IEEE international on-line testing symposium (IOLTS), June 2009, pp 101–106
13. Jacobs A, George A, Cieslewski G (2009) Reconfigurable fault tolerance: a framework for environmentally adaptive fault mitigation in space. In: Proceedings of the international conference on field programmable logic and application (FPL), Aug 2009, pp 199–204
14. Johnson J, Howes W, Wirthlin M, McMurtrey D, Caffrey M, Graham P, Morgan K (2008) Using duplication with compare for on-line error detection in FPGA-based designs. In: Proceedings of the IEEE aerospace conference, March 2008, pp 1–11
15. Elnozahy ENM, Alvisi L, Wang Y-M, Johnson DB (2002) A survey of rollback-recovery protocols in message-passing systems. ACM Comput Surv 34(3):375–408
16. Fleming S, Thomas D (2013) FPGA based control for real time systems. In: Proceedings of the 23rd international conference on field programmable logic and applications (FPL), Sept 2013, pp 1–2
17. Hoffmann H, Eastep J, Santambrogio MD, Miller JE, Agarwal A (2010) Application heartbeats: a generic interface for specifying program performance and goals in autonomous computing environments. In: Proceedings of the 7th international conference on autonomic computing. ACM, 2010, pp 79–88
18. Marshall W, Boshuizen C (2013) Planet labs remote sensing satellite system. In: Proceedings of the AIAA/USU conference on small satellites, pre-conference: CubeSat developers' workshop, SSC13-WK-15. 2013
19. Drineas P, Frieze A, Kannan R, Vempala S, Vinay V (2004) Clustering large graphs via the singular value decomposition. Mach Learn 56(1–3):9–33
20. Winterstein F, Bayliss S, Constantinides G (2013) FPGA-based K-means clustering using tree-based data structures. In: Proceedings of the 23rd international conference on field programmable logic and applications (FPL), 2013, pp 1–6

Chapter 7
Hybrid Configuration Scrubbing for Xilinx 7-Series FPGAs

Michael Wirthlin and Alex Harding

Abstract Configuration memory scrubbing is an essential component of any reliable FPGA-based system used in harsh radiation environments like space. Configuration scrubbing involves the periodic writing of configuration data onto the FPGA to repair configuration upsets that occur within the FPGA due to high-energy ionizing radiation. Configuration scrubbing typically requires external memory and hardware to manage the scrubbing process. This paper presents a novel configuration scrubber for the Xilinx 7-Series FPGAs that requires less external circuitry than traditional scrubbers by exploiting the on-chip Frame ECC and internal scan capability. By exploiting the on-chip features, this scrubber operates faster than traditional scrubbers and with less external hardware. The effectiveness of this scrubber was validated with a radiation test at the Los Alamos Neutron Scattering Center (LANSCE). This scrubber will be used on a Xilinx 7-Series based space processor and a high-energy physics experiment.

7.1 Introduction

FPGAs are increasingly used in non-traditional applications such as harsh environments and in safety critical systems. There has been great interest in using reprogrammable FPGAs within spacecraft to perform computationally demanding tasks such as remote sensing [1, 2]. The use of reconfigurable FPGAs within a spacecraft allows the use of application-specific hardware in place of programmable processors. The ability to customize the datapath within an FPGA to an application-specific computation allows the FPGA to perform many operations faster and more efficiently than the use of traditional programmable processors.

In addition to improved computational efficiency, the use of SRAM-based FPGAs within a spacecraft allows the programmable hardware to perform any user-specified operation. Unlike application-specific integrated circuits (ASICs),

M. Wirthlin (✉) • A. Harding
Department of Electrical and Computer Engineering, NSF Center for High-Performance Reconfigurable Computing (CHREC), Brigham Young University, Provo, UT 84602, USA
e-mail: wirthlin@ee.byu.edu; zaren171@gmail.com

F. Kastensmidt, P. Rech (eds.), *FPGAs and Parallel Architectures for Aerospace Applications*, DOI 10.1007/978-3-319-14352-1_7

FPGAs can be configured after the spacecraft has been launched. This flexibility allows the same FPGA resources to be used for multiple instruments, missions, or changing spacecraft objectives. Errors in an FPGA design can be resolved by fixing the incorrect design and reconfiguring the FPGA with an updated configuration bitstream. Further, custom circuit designs can be created to avoid FPGA resources that have failed during the course of the spacecraft mission.

While the use of FPGAs within a spacecraft several advantages over conventional computing methods, SRAM-based FPGAs are sensitive to the radiation found in most satellite orbits. FPGAs are sensitive to both heavy ion and proton induced single event upsets (SEUs) [3, 4]. Single-event upsets in the FPGA affect the user design flip-flops, the FPGA configuration bitstream, and any hidden FPGA registers, latches, or internal state. Upsets within the FPGA configuration bitstream are especially troublesome as they change the behavior of the circuit. Such upsets may change the contents of look-up tables, routing, or other design-specific functionality.

There is an active research community investigating the effects of radiation on FPGAs and developing methods to mitigate against these effects. There has been significant progress over the last decade in the understanding and development of FPGA technology that is resistant to and tolerant of the effects of radiation. The success of these efforts has facilitated the use of FPGAs in a number of existing spacecraft systems.

The most common way to operate FPGAs in a radiation environment is to provide both active SEU mitigation and configuration scrubbing. Triple-modular redundancy (TMR) is the most common method of providing structural redundancy and involves triplicating circuit resources and inserting voters to choose the correct result [5]. TMR provides protection against all single-bit failures and many multi-bit failures. Configuration scrubbing involves the periodic writing of configuration data into the configuration memory to repair radiation-induced upsets. Although configuration scrubbing does not mitigate against the effects of SEUs, it prevents the build-up of multiple upsets that could break the effectiveness of a SEU mitigation technique such as TMR. Together, TMR and configuration scrubbing have been shown to provide a reliable approach for using FPGAs in radiation environments [6].

There have been many different scrubbing techniques introduced to perform this important function. Configuration scrubbers typically involve external memory storage to hold the "golden" configuration memory and external circuitry to access the memory, read the configuration, compare the configuration with the golden, and if necessary, write the updated configuration. The scrubber presented in this paper performs the same function as traditional scrubbers but does so with much less external hardware. This scrubber exploits the built-in features of the Xilinx 7-Series FPGA to provide internal scrubbing for single-bit upsets and external scrubbing for multi-bit upsets. A low resource JTAG interface is used to perform the external scrubbing functions. The contributions of this paper include a novel multi-level approach for performing configuration scrubbing, a low-resource scrubbing architecture, and a methodology for verifying the scrubber in a radiation beam.

7.2 Configuration Scrubbing

Memory scrubbing is a common technique used in space systems and other systems with high reliability requirements to preserve the integrity of dense memory components. Most memory systems used in such environments support error correction coding (ECC) to correct errors that invariably occur. Error correction logic is typically used to correct data during a memory read. If the state of a memory word has been corrupted, the error correction logic recognizes the error and computes the correct word value. In most systems, the state of the internal word is not corrected—only the value read from the memory is correct. To fix the internal state of the memory word, the corrected value must be written back into the memory. Memory scrubbing is typically used to repair upset words and to prevent the buildup of errors that would break the ECC code. Memory scrubbing involves periodically reading each address of memory and writing the result back into memory. The rate of scrubbing is set to meet a system-level mean-time between failure specification [7].

This scrubbing process is also used for the configuration memory of SRAM-based FPGAs operating in a radiation environment. Using the configuration interface, the correct configuration data is written into the FPGA during operation. Configuration scrubbing is typically implemented by exploiting the partial reconfiguration capability of the FPGA. Configuration data is written into the FPGA at a fixed period to correct configuration upsets and prevents the build-up of upsets within the device. Because the configuration logic is glitch-free, the circuit will continue to operate correctly while configuration scrubbing takes place.

To perform FPGA scrubbing, the configuration data is typically read in sequential order from start to finish. As discrete blocks of configuration data are read, the scrubbing system compares this data against a golden data set or a golden configuration check code such as a cyclic redundancy check (CRC). If a discrepancy is found between the configuration data within the device and the golden configuration data, the scrubbing system will repair the corrupted data by writing the correct, golden data into the FPGA (see Fig. 7.1). If there is no discrepancy between the configuration data and the golden data, the scrubber moves on to the next set of data. Once the scrubber has reached the end of the configuration data, the process is repeated again from the beginning. This process of reading configuration data and repairing upsets that are found in the data continues indefinitely to preserve the configuration data.

The system architecture of a typical configuration scrubber is shown in Fig. 7.2. Most scrubbing systems include a non-volatile memory to store the golden configuration memory. In addition, many configuration scrubbers include a custom circuit to perform the configuration readback, compare, and configuration repair. This external scrubbing hardware is often implemented within a radiation tolerant anti-fuse FPGA or a radiation-hardened ASIC to provide reliable scrubbing in the presence of radiation.

There are many variations to this standard scrubbing architecture. An important characteristic of scrubbers is whether the scrubbing is performed “blind” or

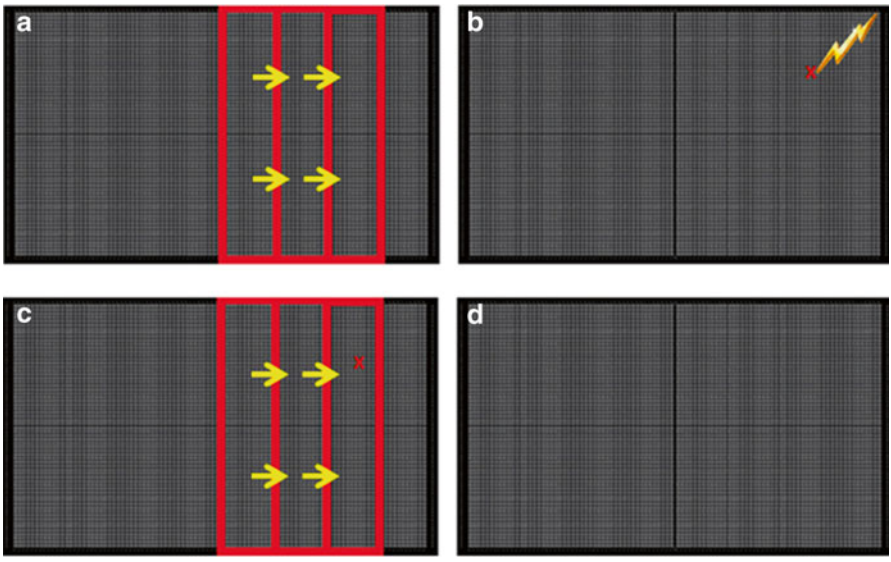


Fig. 7.1 Scrubbing example. (**a**) The device is read frame by frame, progressing from left to right. (**b**) A radiation strike cause an upset in the FPGA configuration memory. (**c**) As the scrubber progresses through the device it eventually will find the frame with the error and fixes it. (**d**) After the frame is scrubbed the error is gone

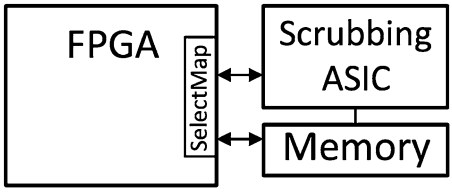


Fig. 7.2 Typical organization of an FPGA scrubber

not [8]. A blind scrubber will continuously write configuration frames into the FPGA without reading the data or determining whether an error was present in the configuration frame. Blind scrubbers are simple to implement and very fast. The disadvantage of blind scrubbers is that they do not provide feedback on the upsets in the configuration memory. Many scrubbers employ configuration readback to read the state of each configuration frame. The configuration frames are compared against a "golden" configuration codebook (using ECC or a direct comparison). Readback scrubbers are more complex and are slower than blind scrubbers. Readback scrubbers, however, provide real-time configuration upset data and can be used to monitor the radiation environment and the radiation response of the FPGA.

7.3 Xilinx 7-Series Configuration

The Xilinx 7-Series FPGA provides a number of novel features that facilitate the ability to create unique configuration scrubbing approaches. This section will summarize the key configuration mechanisms of this FPGA family and discuss how these features are used in the multi-layer scrubber described in this paper.

The lowest granularity of configuration for 7-series FPGAs is the configuration "frame". For the 7-series FPGA, each frame is 101 words of 32-bits each (3,232 bits per frame) [9]. The middle word (word 50) of each configuration frame contains an ECC word (see Fig. 7.3) that provides single-bit error correction and double-bit detection (SECDED). A memory check is performed on a frame when it is read back using the configuration readback mechanism.

Dedicated (non-configurable) logic is built into the FPGA to compute a check word for each frame during configuration readback. This logic compares the check word with the internal frame ECC word and determines whether the frame is error free, contains a single error, or multi-bit error. The FRAME ECCE2 primitive allows a user design to monitor the status of this internal error checking (see Fig. 7.4). This block provides the user design with the location of the last frame checked (FAR or Frame Address Register), signals indicating the status of the last frame check

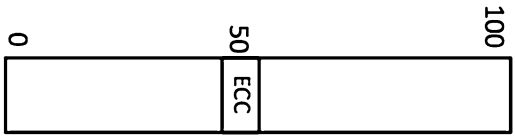


Fig. 7.3 7-Series configuration frame

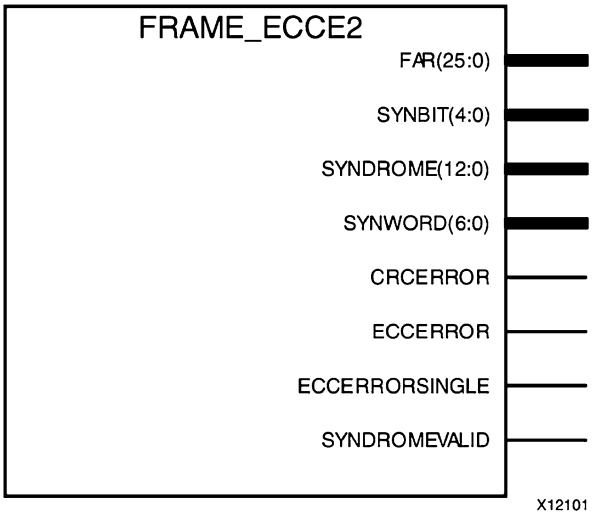


Fig. 7.4 FRAME_ECCE2 primitive

(ECCERRORSINGLE for single-bit errors, ECCERROR for multi-bit errors, and CRCERROR for CRC errors), and the location of the error for single-bit errors (SYNBIT and SYNWORD).

Because of the limitations of the ECC code, some multi-bit errors (odd errors of three bits or more) within a frame may not be detected by the FRAME ECCE2 block. To detect this condition, a global CRC is provided for the entire set of frames. This CRC is recomputed during each full scan of the configuration memory and compared against an internal global CRC. If a multi-bit error occurs that is not detected by the individual frame ECC, the recomputed CRC will differ from the global CRC signifying that an undetected error exists somewhere in the configuration memory.

Configuration frames are organized into different "blocks". Block 0 configuration frames are used to define the function of the logic, I/O, routing, DSPs, etc. Block 1 configuration frames are used for defining the initial contents of the BRAM and other undocumented blocks exist to perform proprietary functions. Typically, only Block 0 configuration frames are scrubbed—these frames are essential for the proper operation of the circuit operating in the FPGA. Configuration scrubbing is not needed for the BRAM as BRAM contents can be protected by the built-in BRAM ECC logic or other well-known memory protection schemes.

An important feature of the configuration logic within Xilinx 7-Series FP-GAs is an "Internal Scan" function that provides built-in self-scrubbing. Dedicated logic within the FPGA can be configured to continuously read configuration frames in Block 0 in sequential order and repair single-bit upsets within the frame (using the internal FRAME ECCE2 logic). This internal scan operates quickly and can complete a full scan of the Kintex7 325 device in 30 ms.

Xilinx offers an intellectual property (IP) block called the "Soft Error Mitigation" (SEM) core to facilitate easy use of these 7-series configuration features [10]. This block provides a number of useful features for controlling the configuration and scrubbing including fault-injection, external communication and control via a UART, and several modes of operation. To repair multi-bit errors, the "Correction by Replace" mode is used. This mode reads configuration frames from an external memory much like the scrubbing architecture shown in Fig. 7.2.

7.4 Hybrid Scrubbing Architecture

As described earlier, the internal configuration scan and Frame ECC can only repair single-bit upsets and external mechanisms are required to repair intra-frame multi-bit upsets. Results from radiation testing on the 28-nm Kintex7 FPGA suggest that intra-frame multi-bit upsets account for 9.9 % of the events observed [11]. This suggests that the internal scrubbing mechanism will be able to fix 90.1 % of the events and external mechanisms are needed for the other events.

The hybrid scrubbing architecture presented in this paper supplements the built-in configuration mechanisms of the 7-Series FPGA (inner layer) with an outer,

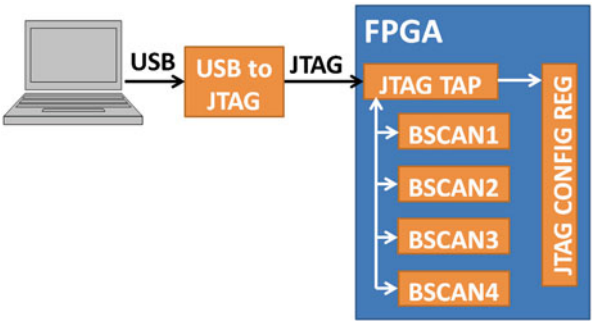


Fig. 7.5 Dual-layer configuration scrubbing: Internal scan and external host (via JTAG)

external scrubber operating on a remote host (see Fig. 7.5). A low-cost, low-bandwidth JTAG connection is used to communicate between the two scrubbing layers. The internal scrubber performs a continuous scan of the FPGA Block 0 frames and repairs all single-bit upsets as described in Sect. 7.3. The internal scrubber reports all single-bit upsets and indicates when any multi-bit upset occurs or when a global CRC error is found.

The FPGA communicates to the external scrubbing manager through JTAG registers within the FPGA. The FPGA design instances "BSCAN" primitives within the FPGA logic to provide the single-bit upset information, multi-bit upset information, and CRC errors. The host communicates with the internal scrubbing system through these JTAG registers.

The Kintex-7 KC705 development board used for this work contains a Digilent JTAG-USB surface mount programming module [12] to facilitate communication between the FPGA and the host through the JTAG port. This module contains an API that allows the user to write host programs for user-specific communication. A library of JTAG communication routines were created to support the scrubbing specific communication described above.

The flow-chart of the hybrid configuration scrubber is shown in Fig. 7.6. The internal self-scan configuration scrubber continuously reads configuration frames and computes a syndrome for the readback frame. The syndrome is compared with the internal ECC word of the frame to determine whether there is an error or not. If there is no error, the frame address register is incremented and the process of frame readback and compare continues with the next frame. If there is an error, an error handling process is initiated (described in the next paragraph). This process continues through all Block 0 frames of the configuration bitstream. After scanning the all Block 0 frames, the process repeats with the first frame in the configuration.

If an error is found during a frame compare, the continuous frame scan halts. If a single-bit error is found, the internal ECC circuitry computes the location of the error and toggles the upset bit (see Fig. 7.4 for the signals provided by the FRAME ECCE2 primitive during an error condition). The internal scan unit then writes the corrected frame back into the configuration memory. If a multi-bit error is found (i.e., ECCERRORSINGLE = false), a message is sent over JTAG to the host using the internal boundary scan primitive. This message contains the frame number of

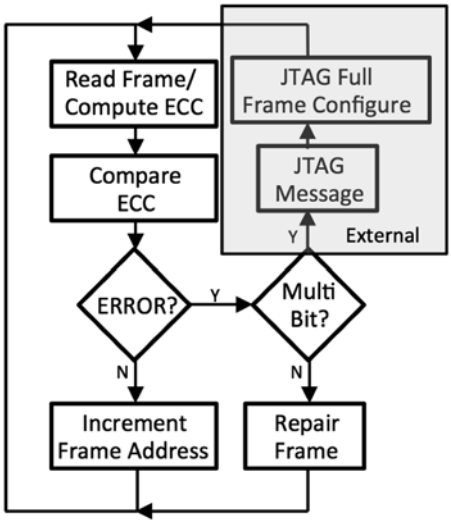


Fig. 7.6 Hybrid scrubber flow chart

the upset frame and instructs the host to reconfigure the frame remotely. The host then performs a full frame reconfigure over JTAG of the upset frame. After configuring the frame, the internal scan process continues.

During this process of internal configuration scan, a global CRC is computed for the full bitstream. At the end of the scan, the computed CRC is compared against the known good CRC value. If the CRC computed during the scan does not match the internal CRC, the CRCERROR signal is asserted. If this signal occurs without any ECC errors, an undetected multi-bit upset has occurred. Since no ECC error was asserted during the scan, the scrubbing system does not know the location of upset frame. In this situation, the scrubber will configure (scrub) every frame to restore the proper configuration value.

The internal scan scrubber completes a full scan with no errors in 30 ms or 1.3 µs per frame. The external JTAG scrubber can configure a full device in 115 s or 5 ms per frame. A full reconfiguration for a CRC error requires over 2 min using the JTAG SMT1 module.

7.5 Radiation Test

The hybrid scrubbing approach described in the previous section was verified in a radiation test in September 2013 at the Los Alamos Neutron Scattering Center (LANSCE). The hybrid scrubber was implemented on a Kintex-7 KC705 Evaluation board with a Kintex-7 325 device. The FPGA was configured with a design that contained the internal scrubber, the JTAG interface to the external scrubber, and a large design full of counters and block memories to consume most of the logic

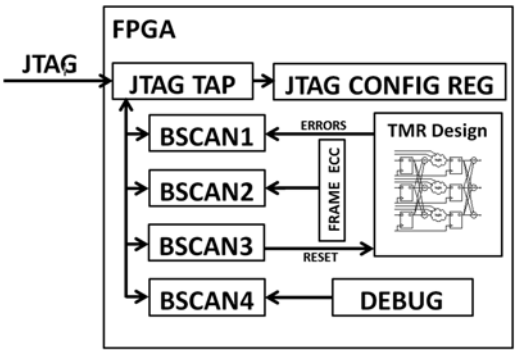


Fig. 7.7 The hybrid scrubbing architecture: Host + JTAG + internal scan

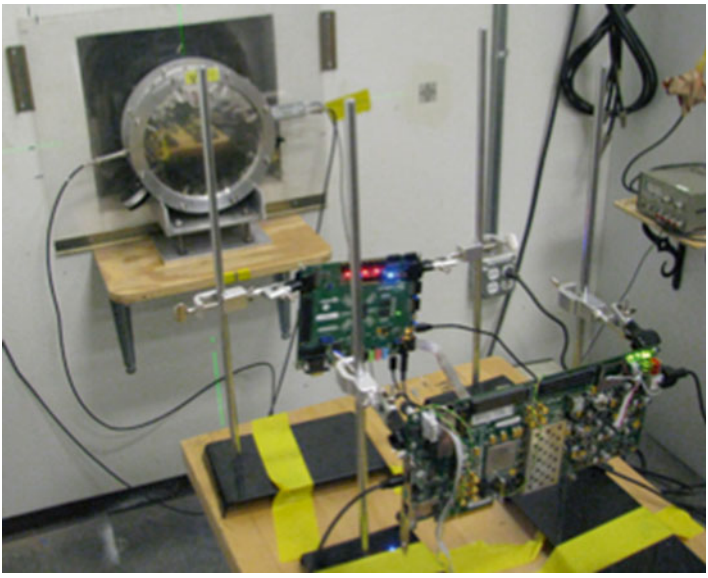


Fig. 7.8 Radiation test of the hybrid scrubber on the KC705 evaluation board (the board is in the *lower right corner*)

resources. Triple modular redundancy was employed to mitigate against temporary configuration upsets. A block diagram of design is shown in Fig. 7.7.

The Kintex-7 device was placed in the radiation path of the neutron beam to verify the operation of the scrubber. The FPGA was configured with the design shown in Fig. 7.7. The scrubber was enabled and the scrubber behavior was monitored over JTAG through a remote host. The scrubber reported single-bit errors, multi-bit errors, CRC errors, and the location of these errors. In addition to scrubbing events, the system reported errors with circuit functionality (BRAM upsets, TMR upsets, etc.). A photograph of the radiation test setup is shown in Fig. 7.8.

Table 7.1 Event types

Event type	Count	% of Errors
Single-bit	758	80.5
Multi-bit	183	19.5

During the testing the scrubber was able to correct all events encountered. Table 7.1 shows the results of the radiation testing. From this table it can be seen that the internal scrubbing mechanism fixed 80 % of all upsets that occurred within the FPGA configuration. Only one fifth of the events required the JTAG connection for recovery. No CRC errors were observed during the test (CRC errors are due to multi-bit errors that are not caught by the internal Frame ECC block).

One challenge that was encountered when validating this hybrid approach was handling the built-in frame ECC update feature of the internal scrubber. To support partial reconfiguration of the configuration memory, the internal scan will recompute the ECC word of each configuration frame anytime external reconfiguration occurs. While this feature is very helpful in making sure that the ECC words and global CRC match the actual contents of the configuration frame, it introduced a unique problem in the radiation beam. When a multi-bit upset was reported by the FRAME ECCE2 block, the external scrubber responds by reconfiguring the upset configuration frame. This external configuration triggers the internal scan unit to recompute the ECC words of each frame and global CRC (it assumes that the reconfiguration is due to new data being configured onto the device). The problem with this ECC update feature is that it computes the incorrect ECC word when it evaluates frames that have upsets—rather than fixing the upsets, it assumes the frame is valid and updates the ECC word to reflect the "new" data. This built-in ECC update feature was disabled to prevent incorrect ECC words from being computed.

7.6 Conclusion and Future Work

This paper describes a robust scrubbing architecture for the Xilinx 7-Series FPGA that requires limited external hardware resources. The scrubber exploits the internal scan scrubbing capability of the FPGA architecture to quickly handle single-bit upsets and adds a slow, external scrubber to handle multi-bit upsets and CRC errors. The external scrubber relies on the slower JTAG interface for communication and configuration data transfer. Although slower than external scrubbers, this interface requires fewer resources than other configuration alternatives.

This configuration scrubber is being evaluated for use in a number of space missions using the 7-Series device. Additional radiation testing on the 7-Series device and on the hybrid scrubber is scheduled to further validate its functionality. In addition, this scrubber is being considered for several high-energy physics experiments in which FPGAs are used with a high and constant radiation environment. In particular, this scrubber is being prepared for use in the Liquid Argon Calorimeter (LAr) of the ATLAS experiment at CERN [13].

This work is currently being extended to support the Xilinx ZYNQ family of system-on-chip processors. The ZYNQ architecture combines dual ARM processor cores with 7-Series FPGA logic. This hybrid processor, FPGA architecture introduces a new way to access the configuration port called the "PCAP". This style of hybrid scrubbing can be adapted by replacing the JTAG interface with the PCAP. This PCAP hybrid scrubber is being developed for the CHREC Space Processor (CSP), a cube-sat space processing board currently being planned for two space missions.

Acknowledgment This work was supported by the I/UCRC Program of the National Science Foundation under Grant No. 1265957.

References

1. Caffrey M (2002) A space-based reconfigurable radio. In Plaks TP, Athanas PM (eds) Proceedings of the international conference on engineering of reconfigurable systems and algorithms (ERSA), June 2002, CSREA Press, pp 49–53
2. Caffrey M, Morgan K, Roussel-Dupre D, Robinson S, Nelson A, Salazar A, Wirthlin M, Howes W, Richins D (2009) On-orbit flight results from the reconfigurable cibola flight experiment satellite (CFESat). In 17th IEEE symposium on field programmable custom computing machines, 2009. FCCM '09, pp 3–10
3. Fuller E, Caffrey M, Blain P, Carmichael C, Khalsa N, Salazar A (1999) Radiation test results of the Virtex FPGA and ZBT SRAM for space based reconfigurable computing. In MAPLD proceedings, Sept 1999
4. Quinn HM, Graham PS, Morgan K, Baker ZK, Caffrey MP, Smith DA, Bell R (2012) On-orbit results for the Xilinx Virtex-4 FPGA. Technical report. Los Alamos National Laboratory (LANL), New Mexico
5. Sterpone L, Violante M (2005) Analysis of the robustness of the TMR architecture in SRAM-based FPGAs. IEEE Trans Nucl Sci 52(5):1545–1549
6. Ostler PS, Caffrey MP, Gibelyou DS, Graham PS, Morgan KS, Pratt BH, Quinn HM, Wirthlin MJ (2009) SRAM FPGA reliability analysis for harsh radiation environments. IEEE Trans Nucl Sci 56(6):3519–3526
7. Saleh AM, Serrano JJ, Patel JH (1990) Reliability of scrubbing recovery-techniques for memory systems. IEEE Trans Reliab 39(1):114–122
8. Berg M, Poivey C, Petrick D, Espinosa D, Lesea A, LaBel KA, Friendlich M, Kim H, Phan A (2008) Effectiveness of internal versus external SEU scrubbing mitigation strategies in a Xilinx FPGA: design, test, and analysis. IEEE Trans Nucl Sci 55(4):2259–2266
9. Xilinx corporation. 7 series FPGA configuration user guide. UG470 (v1.7), 22 Oct 2013
10. Xilinx. LogiCORE IP Soft Error Mitigation Controller v4.0. PG036 19 June 2013
11. Wirthlin M, Lee D, Swift G, Quinn H (2014) A method and case study on identifying physically adjacent multi-cell upsets using 28nm interleaved and SECDED-protected arrays. IEEE Trans Nucl Sci 61(6):3080–3087
12. Digilent. JTAG SMT1 programming module for Xilinx FPGAs. 10 June 2011
13. Wirthlin M, Takai H, Harding A (2014) Soft error rate estimations of the Kintex-7 FPGA within the ATLAS liquid argon (LAr) calorimeter. IOP Sci J Instrum 9(1), C01025

Chapter 8
Power Analysis in nMR Systems in SRAM-Based FPGAs

Jimmy Tarrillo and Fernanda Lima Kastensmidt

Abstract Triple Modular redundancy technique is mostly used to mask transient faults in circuits operating in dependable systems. The generalization of this technique (known as nMR) allows the use of more than three redundant copies of the circuit to increase the reliability under multiple faults. The main drawback of nMR is its high power consumption, which usually implies in n times the power consumption of a single circuit. In this work, we present a mathematical model that predicts the power consumption overhead based on the power characteristics of the basic module. We estimate power consumption in some case-study circuits protected by nMR in a commercial SRAM-based FPGA and compare to a proposed model that estimates power consumption penalty. Results demonstrate that nMR can be implemented with low power overhead in FPGAs and therefore it is a suitable technique for most applications synthesized into this type of programmable devices that need to cope with massive multiple faults.

8.1 Introduction

Aerospace and automotive applications require very complex electronic devices to control and process information with high reliability [1, 2]. In order to reach high reliability and also availability capabilities, systems may use redundant schemes such as multiple modular redundancies (nMR) to mask faults. nMR uses n redundant modules running in parallel and requires the use of a voter to select the correct outputs [3]. The most common nMR implementation is when $n=3$, known as triple modular redundancy (TMR). In TMR, 2 out of 3 modules must work properly to provide the correct result chosen by the voter.

J. Tarrillo (✉)
e-mail: jtarrillo@utec.edu.pe

F.L. Kastensmidt
Federal University of Rio Grande do Sul, Porto Alegre, Brazil
e-mail: fglima@inf.ufrgs.br

F. Kastensmidt, P. Rech (eds.), *FPGAs and Parallel Architectures for Aerospace Applications*, DOI 10.1007/978-3-319-14352-1_8

When considering implementing an entire system into a single chip, Field Programmable Gate Arrays (FPGAs) customized by SRAM cells are very attractive due to their high capability of design integration, low NRE cost and reconfigurability. However, due to its high device integration and high number of memory cells, FPGAs can be vulnerable to radiation effects such as soft errors. Soft errors may occur due to the interaction of secondary particles generated by neutron in the atmosphere with the silicon resulting temporally charging or discharging of sensitive transistor nodes. Designs are configured into SRAM-based FPGAs by loading millions of bits into the configuration memory bitstream. These SRAM memory cells are susceptible to soft errors such as Single Event Upsets (SEU) or bit-flips [4]. The number of faults needed to provoke an error in design output may vary from 20 accumulated faults or more according to the design masking factor capability [5]. In order to cope with SEU in SRAM-based FPGAs, it is required to have designs able to mask upsets, and to correct the accumulated upsets from time to time by reloading the correct (faulty-free) configuration bitstream into the FPGA.

TMR is efficient to mask single errors but it cannot cope with multiple bit upsets that provoke multiple errors in the outputs. Considering that according to the technological trend, FPGAs have more and more probability of having multiple faults [6–8], the use of nMR may be an attractive solution at design level on systems integrated into a single FPGA as presented in [9]. However, the main drawback of using a higher number of multiple redundant modules is the power consumption. FPGAs are designed to have a suitable size configurable matrix that can fit many types of designs projected by the user. So, the amount of transistors of a FPGA is the same for all implemented designs, and the static power consumption is almost independent to the implemented design [10]. Moreover, despite being used 100 % of logic blocks and user flip-flops, about 35 % of the static power is dissipated in the unused transistors of unused interconnect switches [11]. The dynamic power of the customized design is the one that plays the main difference among designs but it represents a small overhead in the majority of the cases.

In this chapter, we present a generic model to estimate the power penalty in nMR designs synthesized into SRAM-based FPGA. The goal is to use the model to help to predict in early stages of the design process the power overhead when using nMR. The target FPGA family in this section is Virtex-5 from Xilinx [12], but this work can be extended to other families of the same fabricant. We discuss the proposal model in terms of number of redundancies (n) in the nMR technique, the relation between static and dynamic power (r) and the size of the FPGA matrix. Then, we provide a power consumption analysis of a synthetic circuit (chain of adders), and a microprocessor (running a matrix multiplication application) using nMR, where n varies from 3 to 7. All implemented designs were synthesized into different sizes of Virtex-5 SRAM-based FPGAs. Comparisons between the power consumption estimated by XPower tool and the model are presented. The model can guide designers to predict the impact of a design protected by nMR in SRAM-based FPGAs. And the low overhead power results may impulse designers to use more often nMR in high reliability applications when using SRAM-based FPGAs.

8.2 Modeling Power Consumption in SRAM-Based FPGAs

Total power consumption (P_{TOT}) is composed by static power P_{STAT} and dynamic power P_{DYN} defined by Eq. 8.1.

$$P_{TOT} = P_{STAT} + P_{DYN} \tag{8.1}$$

In CMOS devices, the static power is linearly related to the voltage level (V_{CC}), and to the leakage current of the device (I_{CC}), as defined in Eq. 8.2. The leakage current of the device is the sum of the transistor leakage currents, which depends of the voltage and operational temperature of the transistor.

$$P_{STAT} = V_{CC} \times I_{CC} \tag{8.2}$$

On the other hand, the dynamic power is related to the switching activity of transistors, and the capacitance and voltage level that powers the device, as defined in the Eq. 8.3. Notice that if all transistors are powered with the same voltage level V_{CC} and the same frequency, the Eq. 8.3 can also be written as Eq. 8.4

$$P_{DYN} = \sum_{i=1}^{n} \alpha_i C_i f V_{CC}^2 \tag{8.3}$$

where:

n = number of toggling nodes
α_i = switching activity
C_i = load capacitance of the node i
f = clock frequency
V_{CC} = transistor source voltage

$$P_{DYN} = f V_{CC}^2 \sum_{i=1}^{n} \alpha_i C_i \tag{8.4}$$

Both Eqs. 8.2 and 8.4 are valid for designs implemented as ASIC or into FPGAs. However, the total power consumption of a design depends on the specific design characteristics of target circuit. In ASIC, the number of transistors is optimized for area and performance and interconnections are implemented directly by metal traces. Consequently, the static power consumption is designed to be as minimum as possible, and the dynamic power is the main contributor for the total power consumption. On the other hand, SRAM-based FPGA devices are composed by fix number of transistors, which comprise the arrangement of logical blocks, configurable interconnects and special blocks as internal RAMs and DSP modules. These elements are the key of the versatility, which is the main feature of the SRAM-based FPGA, but also all these resources are the cause of extra static power consumption.

Table 8.1 Maximum and recommended voltage levels in supply voltage lines of Virtex-5 FPGA (65 nm) [13]

Symbol	Description	Absolute maximum voltages (V)	Performance impact
V_{CCINT}	Internal supply voltage relative to GND	–0.5 to 1.1	0.95–1.05
V_{CCAUX}	Auxiliary supply voltage relative to GND	–0.5 to 3.0	2.375–2.625
V_{CCO}	Output drivers supply voltage relative to GND	–0.5 to 3.75	1.14–3.45
V_{BATT}	Key memory battery backup supply	–0.5 to 4.05	1.0–3.6

As it is well known, the same design implemented in ASIC and into a FPGA using the same process technology will has much less power consumption when implemented as ASIC [10]. Moreover, it is expected that in ASIC implementations, the power overhead caused by the use of redundant modules to be increased in the same factor of the number of redundancies. In case of FPGAs, this proportion may not be true due to the fact that the static power play an important task in the total power consumption.

Aiming to minimize static power in FPGA, vendors offer devices with different number of configurable resources for every family. For example, in case of Virtex-5 LXT, the number of slices (each one contains 4 LUTs and 4 flip-flops) for LX20T, LX30T, LX50T, LX85T, LX110T, LX155T, LX220T, LX330T are 3,120, 4,800, 7,200, 12,960, 17,280, 24,320, 24,560 and 51,840 respectively [11]. In addition, to have a better optimization of power consumption, FPGAs use diverse supply voltage lines for powering their internal components [13] as presented in the Table 8.1.

In order to determine the static power of a FPGA device, it is possible to calculate it by multiplying the typical quiescent supply current at 85° junction temperature (T_j) with the correspondent voltage supply [13]. In order to determine the total power consumption, a tool provided by Xilinx called XPower can be used. It considers the current and power consumption for each voltage line, since different FPGA families have multi voltage power line for internal core, input/output pins, and other elements. XPower is an accurate power estimation tool because it relies in the libraries from the vendor with specific technology and fabric information used in the target FPGA. Static power results are presented in Fig. 8.1, where P_{CCINTq}, P_{CCAUXq} and P_{CCOq} are the static power consumption in lines V_{CCINT}, V_{CCAUX} and V_{CCO} respectively. Note that the size of the device impacts drastically the static power consumption P_{CCINTq} that powers the internal configurable elements.

8.2.1 Power Considerations for nMR FPGA Implementation

Modular redundancy may be implemented considering the replication of the input and outputs pins, or only replying the internal logic as shown in Fig. 8.2. Since all the transistors of the FPGA are turned on independently to the design synthesized into the configurable matrix, it is expected that the static power (P_{STAT}) of a design is almost constant when compared to the total power consumed of the device.

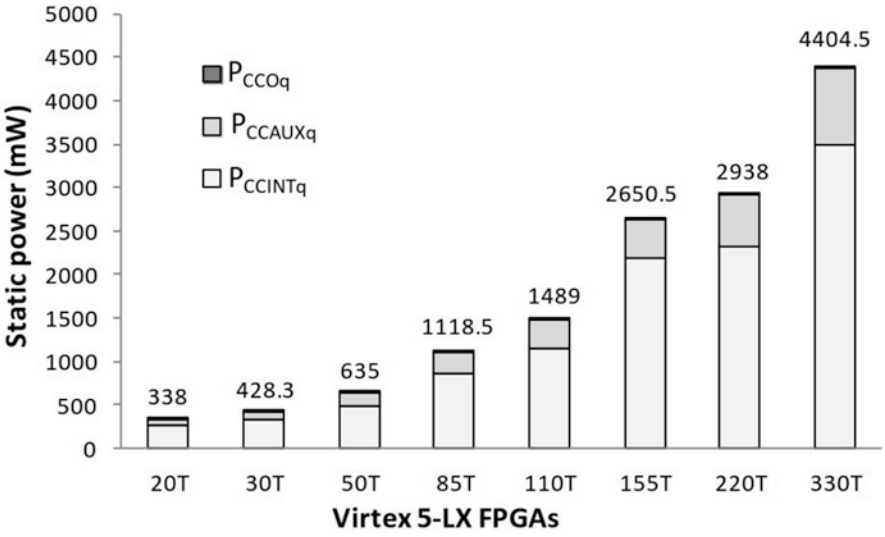


Fig. 8.1 Typical static power consumption for LX Virtex-5 FPGAs by supply line calculated from the typical quiescent supply current values according to [13] and XPOWER tool

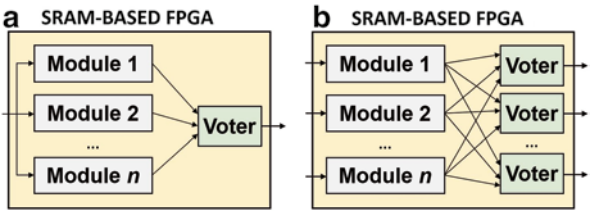


Fig. 8.2 Possibilities for modular redundancy. (**a**) Replaying only internal logic. (**b**) Replaying also input and outputs

In order to have an estimative of the power overhead of an nMR system implemented in a SRAM-based FPGA, we assume that the use of n modules will mainly impact the dynamic power component (P_{DYN}), and consequently the power consumed by the original module can be defined by:

$$P_1 = P_{STAT} + P_{DYN} \tag{8.5}$$

In the case of all inputs and outputs are replied as shown in Fig. 8.2a, the total power consumed by an nMR circuit can be approximated by Eq. 8.6. Note that we are not considering the impact of the power consumption of the voter, since ideally, the voter is very small compared to the redundant module.

Hence, the total power consumed by the nMR circuit (Pn) when inputs and outputs are replicated can be approximated defined by Eq. 8.6. Note that we are not considering the impact of the power consumption of the voter, since ideally, the voter is very small compared to the redundant module.

$$P_n \approx P_{STAT} + n \cdot P_{DYN} \tag{8.6}$$

Consequently, the power overhead ($P_{OV\text{-}IO}$) of an nMR circuit implemented in an SRAM-based FPGA which replies all input and outputs as in Fig. 8.2a can be defined by:

$$P_{OV\_IO} = \frac{P_n}{P_1} = \frac{P_{STAT} + n \cdot P_{DYN}}{P_{STAT} + P_{DYN}} \tag{8.7}$$

Note that the corners of P_{OV_IO} are determined by the relation between dynamic and static power consumption, as shown:

- If $P_{STAT} \gg P_{DYN} \Rightarrow P_{OV\_IO} \approx 1$
- If $P_{STAT} \ll P_{DYN} \Rightarrow P_{OV\_IO} \approx n$

In other words, the minimum power overhead is obtained, when dynamic power is very low compared to the static power. On the other hand, the maximum power overhead is the number of redundancies n, for designs in which the dynamic power is very high compared to the static power.

Considering r as the proportion between dynamic and static power of the original module, the Eq. 8.7 can be rewritten as

$$P_{OV-IO} = \frac{P_n}{P_1} = \frac{n \cdot r + 1}{r + 1} \tag{8.8}$$

where $r = P_{DYN}/P_{STAT}$, of the original module.

Following the same logic, we can model the expected power overhead $P_{OV\text{-}int}$ of nMR when only the functional logic block is replicated as depicted in Fig. 8.2b. In such case, we subtract from the Eq. 8.8 the power consumed by the replicated input and outputs ports ($P_{DYN\text{–}IO}$) as follows:

$$P_{OV-\text{int}} = \frac{n \cdot r + 1 - (n-1) \cdot P_{DYN-IO} / P_{STAT}}{r + 1} \tag{8.9}$$

We can also rewrite the Eq. 8.9 as a function of $P_{OV\text{–}IO}$

$$P_{OV-\text{int}} = P_{OV-IO} - \frac{(n-1)}{r+1} \cdot P_{DYN-IO} / P_{STAT} \tag{8.10}$$

Hence, the power overhead of an nMR system which replicates all input and outputs $P_{OV\text{–}IO}$ can be predicted by the Eq. 8.9, and by the Eq. 8.10 when only the internal logic blocks are replicated. Both equations are based on the number of redundancies, and the dynamic and static power rate characteristics of the original module.

However, the number of modular redundancies is limited by the amount of available resources into the target FPGA. Hence, designers may have two different

project scenarios: when the original FPGA has enough available sources to implement n redundant modules and when it does not and a larger FPGA device of the family must be used.

- ***Option 1***: ***target FPGA is capable to implement the nMR technique***
  In this case, the FPGA part is the same independently of the number of the redundant modules selected, consequently the P_{STAT} is almost constant for all *n* cases. The power overhead model presented in Eq. 8.8 is plotted in Fig. 8.3, for six different values of *r* (ratio between dynamic and static power) and for *n* redundant modules. Notice that for designs with $r<0.5$ ($P_{DYN}<0.5\ P_{STAT}$), the power overhead is very low: for example, for 11 redundancy modules and $r=0.5$, the expected overhead $P_{11}/P_1=4.33$ times larger. Such overhead is considerable very much lower than in the case of an ASIC implementation, when nMR with 11 redundant modules would present an expected overhead in power consumption of approximately 11 times larger power.
- ***Option 2***: ***target FPGA is not capable to implement the nMR technique***
  If the resources required to implement more redundant modules are not available in the original target FPGA device, a larger FPGA must be selected to fit the *n* redundancies. In such case, *r* will be different according to the FPGA selected. Considering FPGAs belonging to the same family product, the main difference will be the number of configurable logics available in the device, and consequently, P_{STAT} will be greater for larger FPGAs. Since r is equal to P_{DYN}/P_{STAT}, it is expected that the power overhead will increase more smoothly as presented in Fig. 8.4.

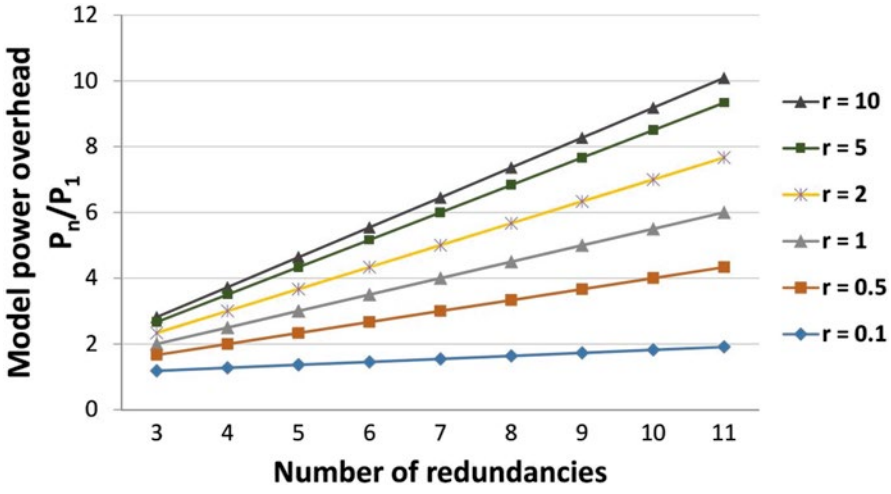


Fig. 8.3 nMR power overhead penalties as function of the number of redundant modules *n*, and the ratio *r* between dynamic and static power considering the Eq. 8.8

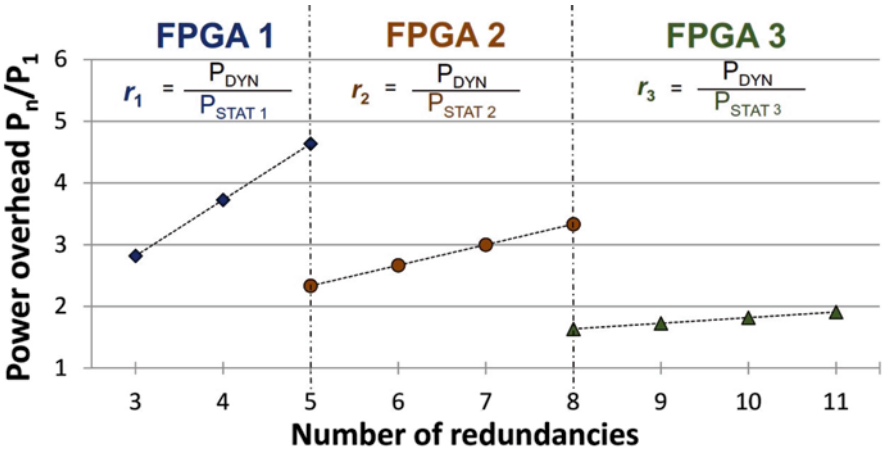


Fig. 8.4 Example of different expected power overheads depending on the target FPGA device capable of implementing the selected nMR considering the Eq. 8.8. Since sizeFPGA1 > sizeFPGA2 > sizeFPGA3, then PSTAT1 > PSTAT2 > PSTAT3, and $r1 < r2 < r3$

8.3 Estimating Power in Case-Study Circuits Implemented in SRAM-Based FPGA

In order to analyze the power overhead in nMR designs and compare it with the proposed model, we estimate the dynamic and static power consumption using XPower Xilinx tool [14] for two case study circuits synthesized into Virtex-5 family FPGAs [12]. The first case study circuit is a miniMIPS soft-processor [15] running a 6×6 matrix multiplication. The last one is a chain of adders implemented by only LUTs and flip-flop slices (no DSP blocks are considered). Although it does not represent a typical application circuit, this circuit allows the exploration of corner case due its high switch activity representing a very high r.

8.3.1 Case-Study Circuit 1: MiniMIPS

MiniMIPS is a soft-core version of MIPS 32-bit microprocessor. The nMR system was implemented in four different versions: $n = 1$ (the original module), $n = 3$, $n = 5$ and $n = 7$, where each miniMIPS runs a 6×6 matrix multiplication algorithm and results are delivered in 12 bits. The system uses the SAv as voter as shown in Fig. 8.5.

Table 8.2 shows the synthesis results for Virtex-5 LX50T, Virtex-5 LX30T and Virtex-5 LX20T FPGA in terms of occupation resources. As shown, if we are looking for the smallest device of Virtex-5 LX family, Virtex-5 LX20T can only be

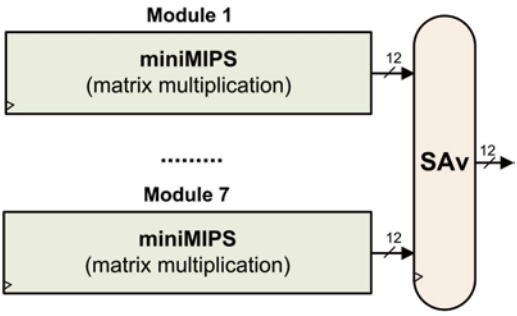


Fig. 8.5 Diagram of 7MR 16-bit adders for power test

Table 8.2 Resources used by miniMIPS-nMR in three Virtex-5 devices

	Virtex-5 LX50T			Virtex-5 LX30T			Virtex-5 LX20T		
n	LUTs (%)	Reg. (%)	BRAM (%)	LUTs (%)	Reg. (%)	BRAM (%)	LUTs (%)	Reg. (%)	BRAM (%)
1	12.18	5.21	5	18.27	7.81	8.3	28.10	12.02	5
3	34.17	15.83	15	51.26	23.75	25	79.53	36.54	15
4	–	–	–	68.36	31.36	33.3	–	–	–
5	56.88	26.34	25	–	–	–	–	–	–
7	79.76	36.85	35	–	–	–	–	–	–

implemented tree modular redundancies. If we need to use 4MR system, the smallest FPGA is Virtex-5 LX30T. If we have a Virtex-5 LX50T, it is possible to implement until seven redundancies (7MR). The SAv voter uses 0.30 and 0.21 % of available LUTs and flip-flops in a Virtex-5 LX50T. These values are very small compared with the size of the original module.

Figure 8.6 shows the dynamic and static power distribution for each case obtained from XPower tool. Notice that static power is constant for all the cases as the FPGA has the same size for all nMR and frequencies, while the dynamic power increases with the number of redundant modules n and the frequency.

Considering the Option 1, we analyze the effect of power consumption in the nMR designs of miniMIPS. For our analyzes propose, we present in Table 8.3 total power consumed for the processor running at 25, 33, 50 and 66 Mhz estimated by the XPower, the r obtained using the XPower results, the power overhead $P_{OV-inter}$ obtained by XPower and by the model defined in Eq. 8.10, and the error of the model proposed respect to XPower results. We highlight that r values are far lower than one, and consequently we expect that power overhead will be low as shown in Fig. 8.3. According to Table 8.3, the highest overhead obtained by XPower is 1.57 times the higher power of the original module, for the 7MR working at 66 Mhz ($r=0.217$). As shown, the overhead obtained from the Eq. 8.10 is very close to results obtained from XPower tool. Notice that the maximum error is 6.54 % for $f=66$ Mhz and $n=7$, and lower errors are obtained for lower r and n values. Results of power overhead obtained from XPower tool and the model proposed in Eq. 8.10 are plotted in Fig. 8.7.

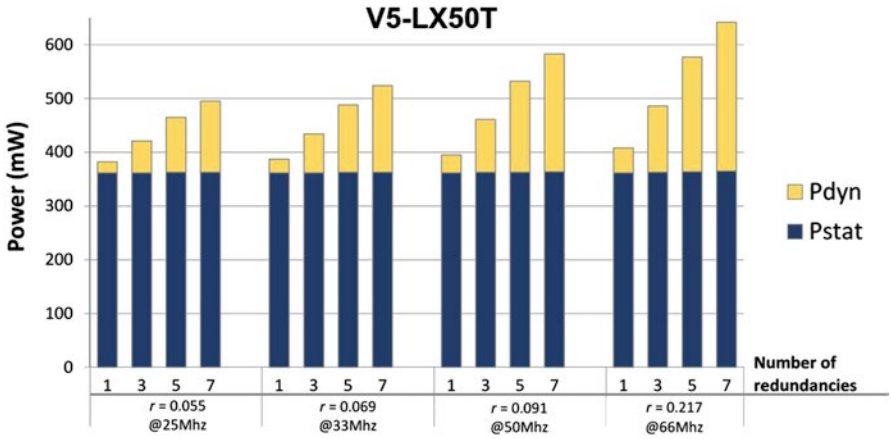


Fig. 8.6 Measured static and dynamic power using XPower of a miniMIPS processor implemented using three different nMR ($n=3$, $n=5$ and $n=7$) synthesized into the same XC5VLX50T FPGA

Now, considering the Option 2, we analyze the effect of power in the nMR designs of the miniMIPS when the target FPGA is not capable to implement the selected nMR cases and a larger FPGA is selected. Aiming the use of the maximum resources in each device, the FPGAs selected were V5LX20T, V5LX30T and V5LX20T. Similar to previous case, Fig. 8.8 shows the power distribution for all nMR circuits implemented. Note that in this case, the static power is not constant for all nMR as the FPGA device changes and n increases, but we can observe that the main contribution of the power comes also from the static power.

Table 8.4 shows the resources used by nMR implementation for $n=3$, 4, 5 and 7, and their power characteristics in Table 8.4. The highest power overhead obtained by XPower is 1.42 times the higher power of the original module, for the 7MR working at 66 Mhz ($r=0.178$). As expected in Fig. 8.4, larger FPGAs have lower r values, and consequently the power overhead increases smoothly. About the error, notice that Eq. 8.10 is pessimistic for all cases. According to the results, the maximum error is always lower than 5 %. Figure 8.9 shows the power overhead obtained from XPower tool for all implemented circuits in three selected devices.

8.3.2 Case-Study Circuit 2: Adders Chain

Considering Eqs. 8.8 and 8.10, a large power overhead is reached when dynamic power is very high too. Since dynamic power is related to the switching activity, any circuit switching a large number of flip-flops and LUTs can be considered as a bad case from the point of view of power overhead.

A synthetic adder chain circuit composes by 190 16-bit adders was selected to explore the power overhead of a circuit with high dynamic power consumption.

Table 8.3 Power consumption estimated by XPower and by the model proposed in the Eq. 8.10 for the miniMIPS-nMR running at 25, 33, 50 and 66 Mhz in XC5VLX50T FPGA

	25 Mhz				33 Mhz				50 Mhz				66 Mhz			
	XPower		Eq. 8.10		XPower		Eq. 8.10		XPower		Eq. 8.10		XPower		Eq. 8.10	
n	P_{TOT} (mW)	P_{OV}	$P_{OV\text{-}int}$	Error (%)	P_{TOT} (mW)	P_{OV}	$P_{OV\text{-}int}$	Error (%)	P_{TOT} (mW)	P_{OV}	$P_{OV\text{-}int}$	Error (%)	P_{TOT} (mW)	P_{OV}	$P_{OV\text{-}int}$	Error (%)
1	382	1	1	0	387	1	1	0	395	1	1	0	408	1	1	0
3	421	1.10	1.11	0.24	434	1.12	1.13	0.69	461	1.17	1.17	0	486	1.19	1.23	2.88
5	465	1.22	1.21	0.65	488	1.26	1.26	0.20	532	1.35	1.33	0.94	577	1.41	1.45	2.60
7	495	1.30	1.31	1.41	524	1.35	1.35	2.48	583	1.48	1.50	1.72	642	1.57	1.68	6.54
r	0.055		–		0.069		–		0.091		–		0.127		–	

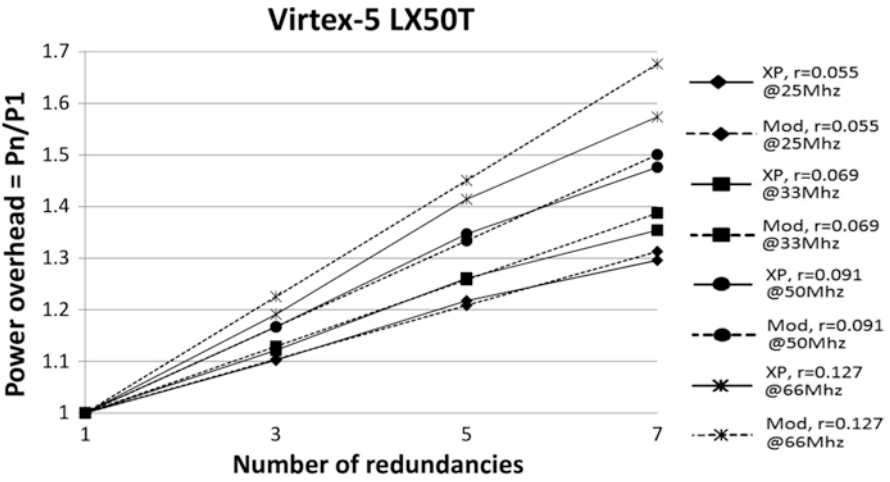


Fig. 8.7 Power overhead of nMR of miniMIPS obtained by XPower (XP) and by the proposed model from Eq. 8.10 for XC5VLX50T FPGA

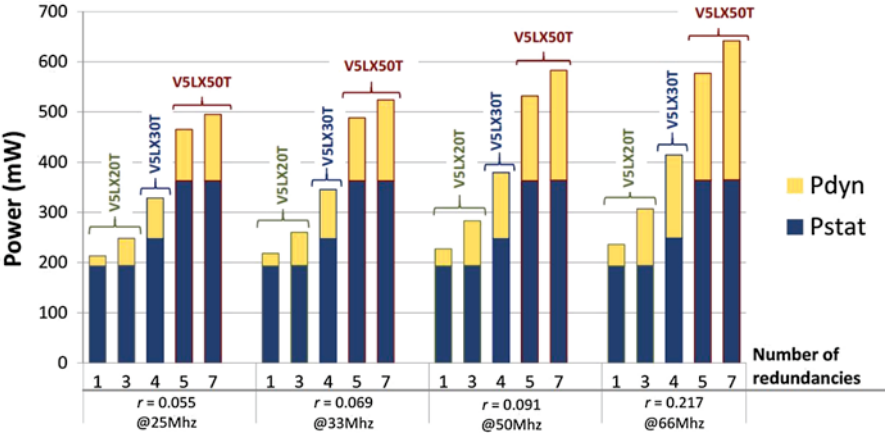


Fig. 8.8 Measured static and dynamic power using XPower of a miniMIPS processor implemented using three different nMR synthesized into the three different FPGAs (XC5VLX20T, XC5VLX30T, XC5VLX50T)

Then, the nMR system analyzed is composed by seven adder chain circuit (basic module) working with a SAv as shown in Fig. 8.10. The number of redundancies and adders aimed to use the more amount of resources of a Virtex-5 LX50T considering a dedicated placement. Each module has the same inputs sourced by a generator pattern based on a 32-bit LFSR to guarantee a high and random switching activity. The switching activity file (vsd file) was created using the post routing model.

Table 8.5 shows the synthesis results for Virtex-5 LX50T FPGA for 3, 5 and 7 redundancies. The total power and power overhead estimated by XPower and by the

Table 8.4 Power consumption estimated by XPower and by the model proposed in the Eq. 8.10 for the miniMIPS-nMR running at 25 and 33 Mhz, 50 and 66 Mhz in XC5VLX30T and XC5VLX20T FPGAs (Option 2)

		25 Mhz				33 Mhz				50 Mhz				66 Mhz			
		XPower		Eq. 8.9		XPower		Eq. 8.9		XPower		Eq. 8.9		XPower		Eq. 8.9	
	n	P_{TOT} (mW)	P_{OV}	$P_{OV\text{-}int}$	Error (%)	P_{TOT} (mW)	P_{OV}	$P_{OV\text{-}int}$	Error (%)	P_{TOT} (mW)	P_{OV}	$P_{OV\text{-}int}$	Error (%)	P_{TOT} (mW)	P_{OV}	$P_{OV\text{-}int}$	Error (%)
V5LX30T	1	268	1	1	0	272	1	1	0	281	1	1	0	292	1	1	0
	3	307	1.15	1.16	–0.98	319	1.17	1.18	–0.94	344	1.22	1.24	–1.46	368	1.26	1.3	–3.27
	4	329	1.23	1.24	–0.61	346	1.27	1.28	–0.29	380	1.35	1.35	–0.79	415	1.42	1.45	–2.17
	r	0.085		–		0.101		–		0.138		–		0.178		–	
V5LX30T	1	213	1	1	0	218	1	1	0	277	1	1	0	236	1	1	0
	3	248	1.16	1.19	–2.02	260	1.19	1.23	–3.08	283	1.25	1.30	–4.24	307	1.3	1.36	–4.89
	r	0.104		–		0.130		–		0.176		–		0.223		–	

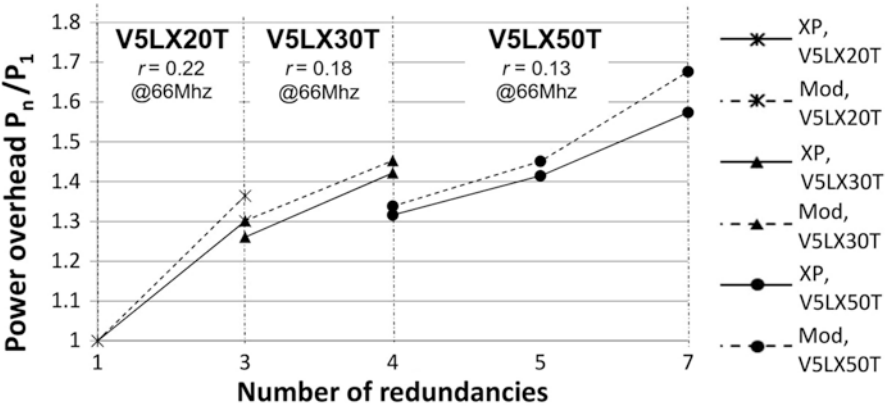


Fig. 8.9 Power overhead of nMR of miniMIPS obtained by XPower (XP) and by the proposed model from the Eq. 8.10 synthesized into the different FPGA Virtex-5 devices (XC5VLX20T, XC5VLX30T, XC5VLX50T)

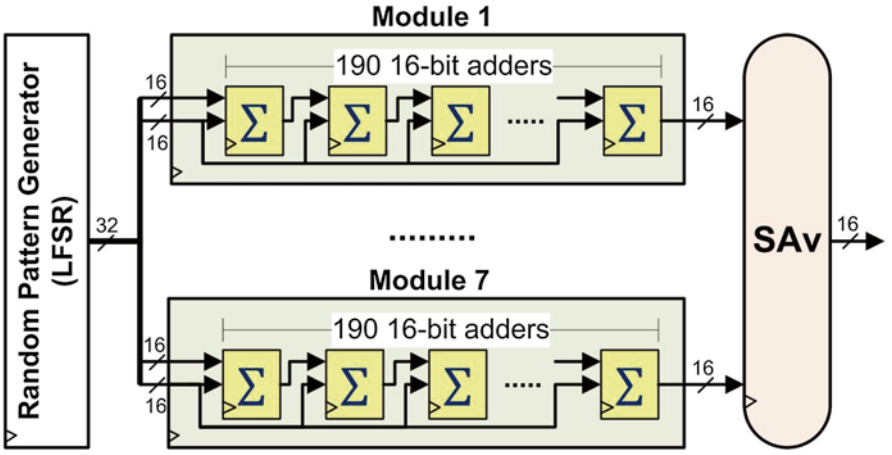


Fig. 8.10 Diagram of 7MR 16-bit adders used in the power analysis

Table 8.5 Resources used by adder chains nMR in Virtex-5 LX50T FPGA

n	LUTs (%)	Reg. (%)	BRAM (%)
1	10.56	10.83	0
3	32.48	32.23	0
5	53.60	54.84	0
7	74.96	76.62	0
SAv	0.72	0.86	0

proposed model presented in Eq. 8.10, running at 25, 50, 100 and 200 Mhz are presented in Table 8.6. The maximum operational frequency is 260 MHz, and the average static power (obtained from XPower tool) is 211.6 mW. Using the dynamic

Table 8.6 Power consumption estimated by XPower and by the model proposed in the Eq. 8.10 for the Adder chain nMR running at 25, 50, 100 and 200 Mhz in XC5VLX50T FPGA

	25 Mhz				50 Mhz				100 Mhz				200 Mhz			
	XPower		Eq. 8.9		XPower		Eq. 8.9		XPower		Eq. 8.9		XPower		Eq. 8.9	
n	P_{TOT} (mW)	P_{OV}	$P_{OV\text{-}int}$	Error (%)	P_{TOT} (mW)	P_{OV}	$P_{OV\text{-}int}$	Error (%)	P_{TOT} (mW)	P_{OV}	$P_{OV\text{-}int}$	Error (%)	P_{TOT} (mW)	P_{OV}	$P_{OV\text{-}int}$	Error (%)
1	498	1	1	0	562	1	1	0	684	1	1	0	931	1	1	0
3	608	1.22	1.21	1.23	762	1.36	1.35	0.27	1,068	1.56	11.55	0.90	1,678	1.80	1.79	0.44
5	706	1.42	1.41	0.41	953	1.70	1.71	−0.52	1,440	2.11	2.09	0.50	2,412	2.59	2.59	0.08
7	803	1.61	1.62	−0.33	1,132	20.2	2.06	−2.11	1,787	2.61	2.64	−1.13	3,094	3.32	3.38	−1.80
r	0.153				0.297				0.572		–		1.120		–	

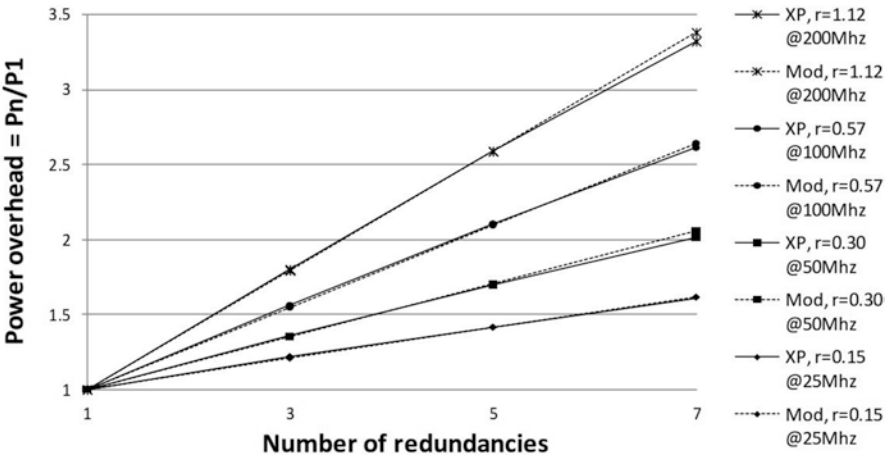


Fig. 8.11 Power overhead of nMR of adder chains obtained by XPower (XP) and by the proposed model (Mod) from Eq. 8.10 for Virtex-5 LX50T FPGA

and static power consumption obtained from XPower Tool, the r values for 7MR are 0.153, 0.297, 0.572, and 1.121, for the system running at 25, 50, 100 and 200 Mhz respectively. We want to highlight that although replicating seven times the original circuit, using almost the totality of LUTs and flip-flops of the FPGA and having a high switching activity, the higher r that we got is 1.120 with a power overhead of 3.32. We interpret these results as the fact that for common circuits, the penalty for using n modular redundancies in SRAM-based FPGA is much lower than n.

Powers overhead estimated by XPower and by the Eq. 8.10 are plotted in Fig. 8.11, and Table 8.6 also presents error of the model respect to XPower results. We can notice the good accuracy of the model. According to the results, the Eq. 8.10 estimate the power overhead with a maximum of error of 2.11 %.

8.4 Conclusions

In this work, the authors analyze the effect on power overhead due to the implementation of n modular redundancy (nMR) designs in SRAM-based FPGAs. A generic model of the power overhead that considers the rate between dynamic and static power consumption (r) of the original module and the number of redundancies is introduced. As case studies, two designs were chosen: miniMIPS processor and adders chain running to different frequencies. Results show that the power overhead when using nMR increases in a much lower proportion than the number of redundancies, and consequently the use of nMR may be a suitable fault tolerant technique for designs implemented in SRAM-based FPGAs to cope with multiple faults. The model to predict the power overhead presented a good agreement with XPower results. Future works will consider including the DSP and other special features in the model and application in other FPGA families.

References

1. Story C (2010) Xcell50 FPGA on Mars. Nat Chem 2(3):147
2. Quinn H, Graham P, Morgan K, Baker Z, Caffrey M, Smith D, Wirthlin M, Bell R (2013) Flight experience of the Xilinx Virtex-4. IEEE Trans Nucl Sci 60(4):2682–2690
3. Instrument N (2008) Redundant system basic concepts. 1–3
4. Baumann R (2005) Radiation-induced soft errors in advanced semiconductor technologies. IEEE Trans Device Mater Reliab 5(3):305–316
5. Chapman AK (2010) SEU strategies for Virtex-5 devices. XILINX 864:1–16
6. Quinn HM, Graham PS, Wirthlin MJ, Pratt B, Morgan KS, Caffrey MP, Krone JB (2009) A test methodology for determining space readiness of Xilinx SRAM-based FPGA devices and designs. IEEE Trans Instrum Meas 58(10):3380–3395
7. Taniguchi H, Yahagi Y, Shimbo K, Toba T (2010) Impact of scaling on neutron-induced soft error in SRAMs from a 250 nm to a 22 nm design rule. IEEE Trans Electron Devices 57(7): 1527–1538
8. Raine M, Hubert G, Gaillardin M, Artola L, Paillet P, Girard S, Sauvestre J, Bournel A (2011) Impact of the radial ionization profile on SEE prediction for SOI transistors and SRAMs beyond the 32-nm technological node. IEEE Trans Nucl Sci 58(3):840–847
9. Tarrillo J, Kastensmidt FL, Rech P, Frost C, Valderrama C (2014) Neutron cross-section of N-modular redundancy technique in SRAM-based FPGAs. IEEE Trans Nucl Sci 61(4): 1558–1566
10. Kuon I, Member S, Rose J, Member S (2007) Measuring the gap between FPGAs and ASICs. IEEE Trans Comput Aided Des Integr Circuits Syst 26(2):203–215
11. Tuan T, Lai B (2003) Leakage power analysis of a 90nm FPGA. In: Proceedings of the IEEE Custom Integrated Circuits Conference, pp 57–60
12. Xilinx (2009) Virtex-5 family overview, DS100 (v5.0). Xilinx
13. Xilinx (2010) Virtex-5 FPGA data sheet: dc and switching characteristics, DS202. Xilinx
14. Xilinx (2011) Xilinx power tools tutorial, UG733 (v13.1). Xilinx
15. Hangout L, Jan S (2009) The minimips project. www.opencores.org

Chapter 9
Fault-Tolerant Manager Core for Dynamic Partial Reconfiguration in FPGAs

Lucas A. Tambara, Jimmy Tarrillo, Fernanda L. Kastensmidt, and Luca Sterpone

Abstract Critical applications must rely on fault-tolerant systems in order to guarantee an error-free execution since the cost of a system fault can be paid in terms of millions of dollars or, even worse, in terms of human lives. In this context, Dynamic Partial Reconfiguration (DPR) enables a more optimized and reliable usage of state-of-the-art Xilinx SRAM-based Field Programmable Gate Arrays (FPGA) resources over space and time. DPR techniques make use of the Internal Configuration Access Port (ICAP), an internal FPGA interface that allows changing on the fly the functionality of a portion of its logic. Unfortunately, a standard DPR flow requires the use of at least a microprocessor (MicroBlaze, PowerPC or ARM), extra memories due to the microprocessor and several peripherals, which results in dense and complex designs that may be easily corrupted by radiation incidence. This chapter presents a generic DPR manager core that has been optimized to provide high reliability. Results are shown in terms of performance, resources utilization and fault tolerance capability, which reinforce its advantages over traditional solutions.

9.1 Introduction

System designs operating in high-reliability applications, such as particles accelerators, aircrafts and satellites require minimal probabilities of a fault affecting the system output. Moreover, in recent years, many Commercial Off-The-Shelf (COTS) products have been employed in these critical areas. The Large Hadron Collider (LHC) is a clear example. There are several areas of LHC in which are used commercial

L.A. Tambara (✉) • J. Tarrillo • F.L. Kastensmidt
Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre, Brazil
e-mail: latambara@inf.ufrgs.br; jtarrillo@inf.ufrgs.br; fglima@inf.ufrgs.br

L. Sterpone
Dipartimento di Automatica e Informatica, Politecnico di Torino, Torino, Italy
e-mail: luca.sterpone@polito.it

F. Kastensmidt, P. Rech (eds.), *FPGAs and Parallel Architectures for Aerospace Applications*, DOI 10.1007/978-3-319-14352-1_9

electronics devices not specifically designed to be radiation-tolerant [1]. Moreover, adopting COTS brings benefits to the project as they include low-cost hardware and software, and they are widely available in the commercial market. On the other hand, COTS are not specially developed for highly reliable applications, which mean that engineer systems with such devices with the same level of reliability as a custom-designed fault tolerant system is a major challenge. In this scenario, reconfigurable architectures such as Xilinx SRAM-based FPGAs have gained more and more attention over the past years.

State-of-the-art Xilinx SRAM-based FPGAs (in this chapter, shortened to only FPGAs) present a set of features that are relevant for systems operating in high-reliability applications, such as flexibility, high performance and fast time-to-market. Moreover, as fabricated with the latest semiconductor manufacturing processes, modern FPGAs are high-density chips that integrate an uprising number of functionalities with reduced voltage threshold and higher frequencies operation [2]. Such advances have the drawback of significantly reducing the COTS FPGAs reliability by making them more susceptible to faults caused by radiation.

One of the major reliability concerns for FPGA is Soft Errors, which are transient faults provoked by the interaction of ionizing particles with the device PN junction. This upset temporally charges or discharges circuit nodes, generating transient voltage pulses that can be interpreted as internal signals, thus provoking an erroneous result [3]. When a fault changes the state of an SRAM cell, this event is referred as Single Event Upsets (SEU). Once SRAM-based FPGAs are composed of millions of SRAM cells to store their configuration [3], they are very susceptible to SEU and Multiple Bits Upsets (MBU) [4]. SEUs and MBUs in configuration memory bits have a persistent effect and can only be corrected by reconfiguring the FPGA.

The integration of COTS FPGAs in critical systems may then require hardening techniques able to mitigate SEU effects, especially if the device is employed in radiation harsh environments. To ensure the correct functionally of the design implemented into an FPGA, it is mandatory to mask and eventually correct radiation-induced SEUs and MBUs. Two well-known techniques are the use of a global Triple Modular Redundancy (TMR) to mask SEUs and the use of reconfiguration of the FPGA's bitstream to correct SEUs in its configuration memory bits.

The reconfiguration process is one of the most important steps to ensure that SEUs are not going to accumulate in the SRAM memory cells, and they will be corrected in an expected required repair time. Most of FPGAs from Xilinx offer the capability of partially change the configuration (i.e. functionality) of the device on the fly. This process is known as Partial Reconfiguration [5] or Dynamic Partial Reconfiguration (DPR) when it is done at run-time and it is performed internally through the Internal Configuration Access Port (ICAP). DPR has been often employed in several types of applications, such as multimedia, avionics, aerospace and other intelligent systems [6–9], either for change the functionality of a system or to fix faults within it. Concerning fault tolerance, a DPR scheme helps to extend the lifetime of a system by periodically rewriting parts of its bitstream in order to avoid the accumulation of multiple SEU.

In order to increase the fault tolerance of a system, a DPR manager core must be highly reliable and with high performance to be able to reconfigure the FPGA within minimal time. Furthermore, solutions based on complex microprocessors, such as MicroBlaze, PowerPC and ARM are not suitable for safety-critical applications as they are too complex and require a lot of peripherals and memory, which may result in an extremely susceptible DPR control system. In this chapter, we present a novel DPR Manager (DPRM) fault-tolerant core against SEUs. The core uses a dedicated control flow design implemented in hardware to control the ICAP interface. For this reason, the DPRM core is optimized for area and performance. The proposed DPRM fault-tolerant core was evaluated in a Xilinx Virtex-5 LX50T FPGA through fault injection, where thousands of SEUs were injected into the SRAM cells to predict the behavior of the proposed DPRM under multiple faults.

This chapter is organized as follow. Section 9.2 presents the classical DPR approach with a brief description of the ICAP interface, the configuration bitstream organization and the DPR implementation in Virtex-5 FPGAs. Section 9.3 presents the architecture of the DPRM and its fault-tolerant version. Section 9.4 presents the DPRM implementation and the obtained results for performance and fault tolerance. Finally, Sect. 9.5 concludes this chapter.

9.2 Classical DPR Approach

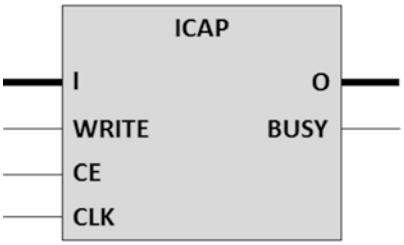
The functionality of an FPGA is defined by a unique configuration data set called *bitstream* that is stored in its internal configuration memory. Since the configuration memory of the Xilinx's FPGAs is volatile, the bitstream is usually loaded from an external non-volatile memory when the FPGA is powered up. A configuration bitstream can be loaded by using serial (Master/Slave, Serial Peripheral Interface—SPI) or parallel (SelectMAP, Byte Peripheral Interface—BPI) modes [10].

The dynamic partial reconfiguration capability in FPGAs aims at changing the FPGA design by loading partial bitstreams, typically stored in a Flash memory, through any available configuration port, i.e. Slave SelectMAP, Slave Serial, JTAG, or ICAP (which is, in fact, an internal representation of the SelectMAP interface) [11]. The most common port nowadays to perform the DPR is the ICAP due to its flexibility of access inside the chip.

Figure 9.1 illustrates the ICAP interface. It is worth mention that the data bus is selectable among 8, 16 or 32 bits. According to [10], in a Virtex-5 FPGA the ICAP interface can run up to a clock frequency of 100 MHz.

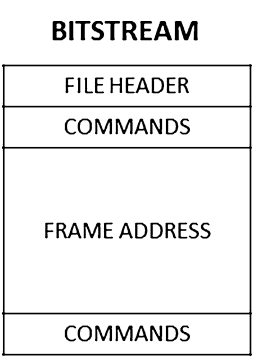
The configuration bitstream structure is shown in Fig. 9.2. In the Virtex-5 family, the smallest group of configuration bits is composed of 1,312 bits and is it known as Frame. The number of Frames in a bitstream depends on the size and type of the resources existing in the reconfigurable region.

In order to perform DPR, Xilinx's proposed flow is handled by an embedded soft-core microprocessor, such as MicroBlaze or hard-core processors as PowerPC or ARM that uses an instance of OPB-HWICAP, XPS-HWICAP or AXI-HWICAP



I	Input data bus (8, 16 or 32 bits)
WRITE	Read/Write control
CE	Enable select
CLK	Interface clock
O	Output data bus (8, 16 or 32 bits)
BUSY	Used in read operations

Fig. 9.1 ICAP primitive description



File header	File information
Commands	Synchronization and others commands used to perform configuration
Frame data	Frames containing the configuration bits
Commands	Commands to end the configuration process and desynchronize the ICAP

Fig. 9.2 Configuration bitstream structure

cores to control the ICAP [10]. Although these cores provide flexibility access to the ICAP when combined with a microprocessor. However, it is worth mention that they require buffers, ICAP control capabilities, and an external memory controller. Together, these components introduce extra area and time overhead and make the HWICAP interfaces totally dependent of the referred processors. Consequently, such HWICAP cores do not provide the best solution for systems that require efficient use of the ICAP in terms of performance and fault tolerance.

The most adopted DPR architecture is illustrated in Fig. 9.3: the processor commands the memory controller (1) to load a bitstream from an external memory (2 and 3) to BRAM; then, when required, the processor transfers a chosen bitstream from the BRAM to the ICAP controller (4) to implement the selected reconfigurable module (5). In this way, the data bus is used during reconfiguration process to transport the configuration bits from the BRAM or a memory controller to the ICAP controller.

The reconfiguration process is started by sending the synchronization command *5599AA66'h* to the ICAP input (*I*, in Fig. 9.3). This word makes ICAP output (*O*, in Fig. 9.3) changes from *9F'h* to *DF'h*, which means that the component is synchronized. During all the time the ICAP output status *DF'h*, the FPGA loads new configuration frames. When configuration data is sent, the bitstream contains a

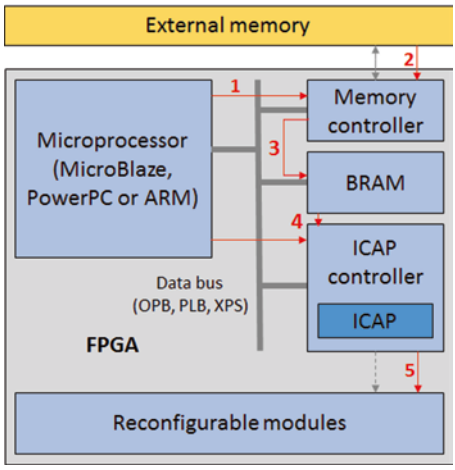


Fig. 9.3 Classical DPR implementation approach showing that a microprocessor controls all the time both memory and ICAP

desynchronization word, which is *000000B0'h*. When ICAP receives this word, its output status changes to *9F'h*, indicating that the component is desynchronized, and the configuration ended.

Several ICAP controllers have been proposed aiming to perform DPR [6–9, 12–21]. Few of them focus to be fault-tolerant [12, 13]. Most are focused to improve resource utilization efficiency over time [6–9, 14–21]. The use of Direct Memory Access (DMA) was proposed by Claus et al. in [6] and by Bhandari et al. in [18] to increase the bitstream throughput through preloading partial bitstreams from a Compact Flash (CF) memory to a DDR SDRAM during system setup. Liu et al. proposed a similar approach in [17], where authors made use of the internal FPGA's Block RAM (BRAM) as cache memory together with externals DDR SDRAM and CF. Lai and Diessel presented in [20] a similar approach as the one that is presented in this article, once they developed both ICAP controller and memory interface. Finally, the work reported by Lamonnier et al. in [14] resembles our proposed architecture and the one in [20] regarding the fact that they do not employ buffers and avoid the use of microprocessors. Except the works presented in [14] and [21], all the aforementioned designs require the use of a processor to manage the DPR process through a data bus and a memory controller, which increases the system complexity and consequently, the susceptibility of it.

Bayar and A. Yurdakul presented a rather different approach in [21]. They made use of the Parallel Configuration Access Port (PCAP) to perform DPR through the SelectMAP port and using the internal BRAM to store the partial configuration bitstreams. A variation of PCAP known as cPCAP [22] considers a decompression mechanism to store bigger bitstreams in a compressed format into BRAM. The cPCAP implementation in [22] uses 324 slices (Spartan-3 FPGA), and its reconfiguration speed reaches 50 MB/s. Although there is no data bus in this approach, the reconfiguration capability of the modules is bounded by BRAM availability.

The problem of most of the mentioned DPR controller is that they are based on complex microprocessors with several peripherals and BRAM. These characteristics make the controllers too large and vulnerable to SEU.

9.3 Proposed DPR Manager Core

9.3.1 DPRM Architecture

The first contribution of this work is to propose a fault-tolerant high-performance DPR Manager core IP (DPRM). The developed DPRM supports dynamic partial reconfiguration by providing low-level hardware services such as storage/retrieval of configuration bits between memory and ICAP, relieving any (hardware or software) controller unit of monitoring the configuration task. This way, DPRM aims to serve as an interface between a system and its reconfigurable logic. The proposed DPRM is based on a specific control flow design and not in a microprocessor to limit the SEU occurrence and produce a more efficient reconfiguration. Moreover, it mainly differs from previous works in two points. First, no data bus (OPB, PLB or AXI) is necessary to transport partial configuration bits. The second difference is in terms of memory usage. The user has the choice of selecting the amount of desired BRAM (if any). The versatility of our solution allows the user to create its own memory driver and plug it to the DPRM for using with different memories or boards. Our proposed architecture is implemented with a Triple Modular Redundancy (TMR) scheme of the DPRM and a special placement of its Majority Voters (MV) to guarantee high reliability. The second contribution of this work is a memory controller capable of interfacing with a BPI Flash memory without the need of any processor.

Figure 9.4 shows the connections of our DPRM architecture. Our proposed DRPM frees the main control unit (generally a microprocessor) of performing the DPR task. As aforementioned in previous sections, if a data bus is used along with a processor, the configuration bitstream has to pass through it. The DPRM frees the data bus of transporting the bitstream. Figure 9.5 details the interface signals and components of the DPRM core. Internally, the *Unit Interface* (UI) interacts with three different modules: the *Controller* (an FSM or a microprocessor), the *Data Manager* (DM), and the *ICAP Control* (IC).

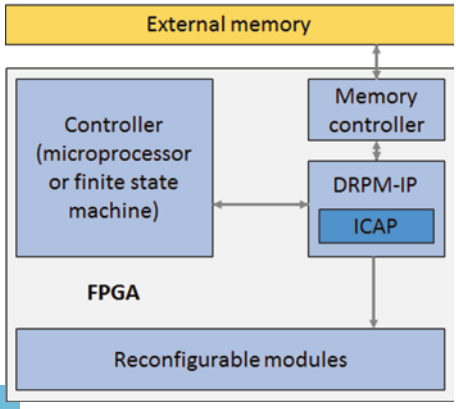


Fig. 9.4 Connections of the DRPM core

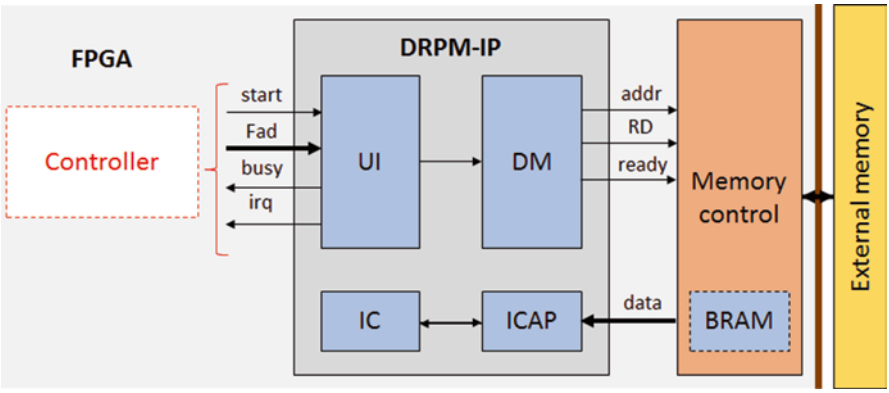


Fig. 9.5 DRPM block diagram

To start the DPR process, the bitstream memory address is placed on the *First Address* bus (*Fad*), and the *Start* signal is triggered. Throughout the reconfiguration process, the *Busy* signal remains asserted, and the *Irq* output is used to indicate when the configuration ends. Once the external memory address is registered into the UI module and the start signal is asserted, the DM module reads the reconfiguration bitstream from the *External Memory* through the *Memory Control* unit. This unit implements the interface with the external memory to retrieve all data. This way, bitstream words are transported from the memory control unit to the ICAP input bus (*data*). The following signals are used to control the memory block:

- *Fad*: bitstream address;
- *RD*: read signal;
- *Ready*: data on output bus is valid.

The memory control unit can make use of internal BRAM to improve the partial configuration speed. However, a tradeoff between configuration speed, resources and fault tolerance will exist. Our proposed architecture allows the following scenarios according to the use of buffer elements:

1. *No buffer is used*. This case is used when the lightest hardware is required. When no buffer is implemented, the data output of the external memory is connected (if possible) directly to the ICAP data lines. This approach depends on the external memory architecture, and consequently, the reconfiguration speed is strongly dependent on the memory. Since external devices usually are slower than internal FPGA elements, this approach represents the slowest configuration option. However, this is the most fault-tolerant option due to the no use of BRAMs, which is known as the most sensitive elements of and FPGA [23].
2. *Buffer is used*. Two scenarios are possible under this option. The first one, when the required partial configuration bitstream is known, it can be preloaded into BRAM early and later sent into the ICAP when needed, reaching the highest

configuration speed. A second scenario is possible when the partial bitstream required is not known in advance; it can be brought into BRAM right after the reconfiguration process is requested, but the configuration speed will be slower than the previous case. Notice that the amount of BRAMs used is proportional to the biggest bitstream and thus, if there are not enough BRAMs to store the bitstream, it can be loaded in chunks, yielding the slowest configuration speed. This will be the most vulnerable scenario, as typically BRAM cells are more susceptible than the configuration memory cells [23]. Fault tolerance will be addressed in more details along the next subsection.

In this work, as the focus is to have a high level of fault tolerance, we implemented a version with no buffering that interfaced a BPI Flash memory which can be read with a simple address/read-enable interface in 2-byte words.

When the BRAM resources selected are not enough to load a complete bitstream, the number of transfers from external memory to the BRAM expressing the bitstream size in bytes, can be computed as:

$$N_1 = ceil\left(\frac{Size_{bitstream}}{Size_{buffer}}\right) \tag{9.1}$$

We name t_1 as the time employed to read one data unit from the external device; for the BPI Flash, one data unit is 2 bytes whereas for the Compact Flash it is 512 bytes. In addition, we name t_2 as the memory handshaking time when starting a new read after inactivity. Finally, we denote N_2 as the number of times a new read transaction is initiated by the FPGA after inactivity with the external device; this value will depend on the buffer size and particular memory constraints. The overall time used to retrieve the partial bitstream from outside into the FPGA can be modelled as:

$$t_{Mem-FPGA} = (t_1.N_1) + (t_2.N_2) \tag{9.2}$$

When using BRAM buffering, the time employed to send the bitstream to the ICAP is only determined by the frequency at which the BRAM is read and cannot be higher than 100 MHz due to ICAP's limitations. The time to send one 32-bit word from BRAM to the ICAP is, therefore:

$$t_{BRAM-ICAP} = \frac{1}{100MHz} \tag{9.3}$$

The reconfiguration time for the no-buffering scenario is given by Eq. 9.4. Since the operating frequency of the external device is usually much less than 100 MHz, the second term in the equation mentioned above only adds a minor effect on the overall time.

$$t_{rec1} = t_{Mem-FPGA} + \left(\frac{Size_{bitstream}}{4}.t_{BRAM-ICAP}\right) \tag{9.4}$$

In the case BRAM buffering is used, pre-loading is possible, thus allowing to decrease $t_{Mem-BRAM}$ to zero when executing the reconfiguration. This scenario is given by Eq. 9.5. Maximum reconfiguration speed is obtained with this configuration.

$$t_{rec2} = \frac{Size_{bitstream}}{4} . t_{BRAM-ICAP} \tag{9.5}$$

9.3.2 Fault-Tolerant DPRM

The DPRM circuit is susceptible to SEU as it is implemented in the FPGA fabric. An SEU occurring within its circuitry can cause the circuit to output incorrect data or cause the circuit to fail. A DPRM failure may have severe repercussions on the entire implemented circuit as a faulty DPRM may incorrectly rewrite configuration bits. It is then fundamental to implement a reliable and SEU-immune DPRM module.

Three different mitigation strategies were applied to add fault tolerance to the plain DPRM module.

The first fault-tolerant version is using a single DPRM with its output signals triplicated (DPRM TMR-Sig). These triplicated signals are the ones that control the ICAP.

The second fault-tolerant version is a TMR version of the DPRM module (TMR-DPRM), as it is shown in Fig. 9.6. In the TMR-DPRM, the original circuit is triplicated by adding two extra copies of the original circuit. The three copies of the TMR approach operate in parallel, and the copies outputs are delivered to majority voters. If an error happens in one of the copies, two of them will continue to operate

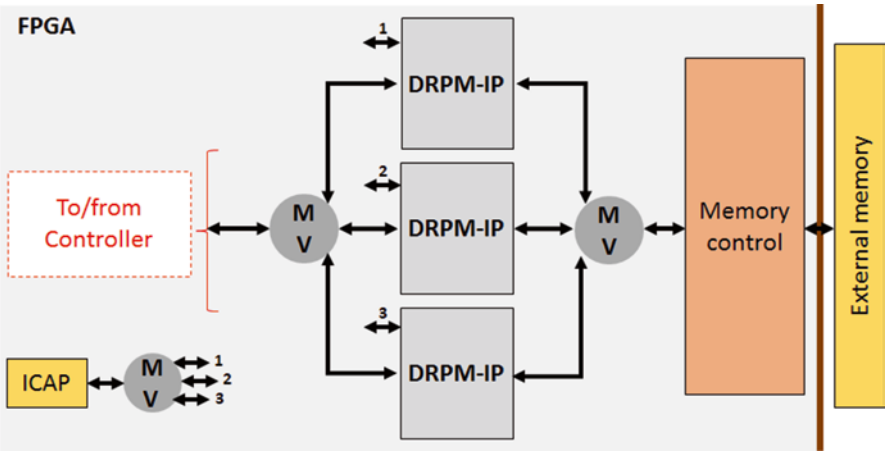


Fig. 9.6 TMR-DRPM block diagram

correctly and the majority voter can correctly mask the output of the faulty module. The majority voters are placed bit-a-bit, which totalizes 84 MV.

A third fault-tolerant version was designed aiming at reducing the probability of a failure occurring in one of them. An optimization was proposed based on a special placement (TMR-DPRM-SP). The goal was to place the majority voters as closer as possible to the output ports, like the ICAP interface. In this way, the critical non-triplicated paths after the MV outputs are reduced. It is worth mention that the TMR-DPRM does not have any special placement of the majority voters.

9.4 Test Setup and Fault Injection Results

The case-study reconfigurable module is a simple counter, which uses 12,818 configuration bytes. The DRPM was implemented in a Xilinx Virtex-5 LX50T. A PicoBlaze embedded processor running at 50 MHz was used to support the DPRM, and a BPI Flash memory was used to store the configuration bitstreams. Moreover, the memory controller did not use any BRAM. Thus the configuration speed depends on the external memory access time Eq. 9.4.

Table 9.1 shows the synthesis and performance results obtained for both DPRM and TMR-DPRM.

Area wise, compared to Xilinx's proposed DPR flow, DPRM has a reduced ratio from 6.4 to 17 times [15]. Comparing to other fault-tolerant ICAP controllers, TMR-DPRM has a reduced area ranging from 2.1 times [13] to 2.6 times [12].

Regarding configuration speed, DPRM presents an improvement of 1.3 times when comparing to OPB-HWICAP [15]. However, in comparison with PLB-HWICAP [15] and other fault-tolerant ICAP controllers [12, 13], the present TMR-DPRM setup has a lower performance. The reason is that the present setup does not use any memory to increase the performance, like BRAM, a microprocessor with cache-enabled or Direct Memory Access feature. Aiming to prove that the DPRM module is able to achieve a higher performance, a preliminary work was performed implementing the DPRM core with BRAM as buffers and an SD Flash memory as external memory in a Virtex-6 LX240T device. First results show a resource usage

Table 9.1 Synthesis and performance results obtained for the TMR-DPRM setup

Version	Block	External memory type	Resources			Max. freq. (MHz)
			Flip-flops	LUTs	BRAM	
DPRM system	DPRM core	–	8	18	0	600.2
	Memory control	BPI Flash	111	135	0	291.3
	Processor	–	76	144	1	162.7
TMR-DPRM system	DPRM core	–	24	88	0	600.1
	Memory control	BPI Flash	111	135	0	291.2
	Processor	–	76	144	1	160.9

of 247 flip-flops, 662 LUTs and 5 BRAM. The achieved configuration speed was about 253 MB/s. If the same conditions were held at 100 MHz, the configuration speed reached would be 384.29 MB/s, which is closer to the highest possible (400 MB/s) and, for example, 1.1 times faster than the solution presented in [13].

A fault injection campaign was performed with the purpose of evaluating the DPRM behavior under multiple faults. The fault injection campaign was done by flipping random configuration bits in the area where different versions of the DPRM are mapped. A set of 2,000 fault injection campaigns was performed. Each campaign injects single to multiple faults: 1, 10, 20, 30, 40, 50, 60, 70, 80, 90 to 100 accumulated faults. All errors occurred in the DPRM outputs were computed.

Once the number of errors due to fault injection is computed, it is possible to calculate the Mean Time Between Failures (MTBF) of the system. MTBF can be defined as the average time (in hours) between two radiation-induced failures within the device. By definition, the MTBF is evaluated as:

$$MTBF = \frac{1}{\left[\left(\frac{number\ of\ errors}{total\ of\ injected\ upsets}\right)\cdot\left(\sigma_{static}\right)\right]\cdot flux} \tag{9.6}$$

where *number of errors* is the number of observed errors in the design output, *total of injected upsets* is the total number of injected upsets in the design during the fault injection campaigns, σ_{static} is the sensitive area to upsets (measured static cross section for Virtex-5 devices is 6.70×10^{-15} cm^2 from [23]) and *flux* is the average neutron flux at sea level (about 13 n/(cm^2·h)) [24].

Figure 9.7 shows the obtained results in terms of MTBF. From the results, it is possible to observe that the use of TMR presented a significant improvement in reliability. The TMR-DPRM showed a fault tolerance 2.5 higher than the unmitigated design. However, with regard to the TMR-DPRM-SP design, it showed a very significant improvement in terms of fault tolerance, achieving a fault tolerance 4.1 times higher than the unmitigated design and 1.67 when comparing with the TMR-DPRM design. For the sake of comparison, the fault tolerance improvement (unmitigated design versus mitigated design) of the work presented in [12] was of 1.5 times. Authors did not present results about fault tolerance in [13].

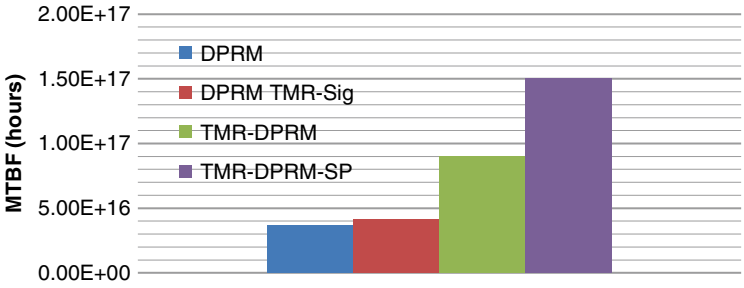


Fig. 9.7 MTBF calculated from fault injection campaign in Virtex-5 FPGA with several DPRM cores

9.5 Conclusions and Future Work

This chapter presented a new DPRM module to perform DPR in an easier, more efficient and more fault-tolerant way when compared to the traditional workflow. The versatility of the proposed DPRM module enables us to retrieve bitstream from a BPI Flash, as well as other types of external memories. The selection of the unit (an FSM or a processor) to control the DPRM and number of BRAM to increment the reconfiguration speed is also configurable by the user.

In this work, the DPRM setup was implemented in a Virtex-5 LX50T with a BPI Flash memory controller. This memory controller does not consider the use of BRAM in the control, resulting in the usage of only 111 flip-flops, 135 LUTs and a reconfiguration speed of 6.5 MB/s. However, a version using an SD memory controller and BRAM as buffers was implemented to prove the possibility to achieve a performance closer to the highest possible (384.29 MB/s).

References

1. Roed K, Brugger M, Kramer D, Peronnard P, Pignard C, Spiezia G, Thornton A (2012) Method for measuring mixed field radiation levels relevant for SEEs at the LHC. IEEE Trans Nucl Sci 59(4):1040–1047
2. ITRS (2014) International technology roadmap for semiconductors: 2013 edition [Online]. http://www.itrs.net/
3. Dodd PE, Massengill LW (2003) Basic mechanism and modeling of single-event upset in digital microelectronics. IEEE Trans Nucl Sci 50(3):583–602
4. Quinn H, Morgan K, Graham P, Krone J, Caffrey M (2007) Static proton and heavy ion testing of the Xilinx Virtex-5 device. In: Proceedings of the IEEE radiation effects data workshop, July 2007, pp 177–184
5. Xilinx (2010) Partial reconfiguration user guide. UG702 (v 12.1), 3 May 2010
6. Claus C, Ahmed R, Altenried F, Stechele W (2010) Towards rapid dynamic partial reconfiguration in video-dased driver assistance systems. In: Reconfigurable computing: architectures, tools and applications. Springer, Berlin, pp 55–67
7. Psarakis M, Apostolakis A (2012) Fault tolerant FPGA processor based on runtime reconfigurable modules. In: Proceedings of the 17th IEEE European test symposium, Annecy, France, May 2012, pp 1–6
8. Viswanathan V, Nakache B, Ben Atitallah R, Nakache M, Dekeyser JL (2012) Dynamic reconfiguration of modular I/O IP cores for avionic applications. In: Proceedings of the international conference on reconfigurable computing and FPGAs, Mexico, Dec 2012, pp 1–6
9. Echanobe J, del Campo I, Finker R, Basterretxea K (2012) Dynamic partial reconfiguration in embedded systems for intelligent environments. In: Proceedings of the 8th international conference on intelligent environments, June 2012, pp 109–113
10. Xilinx (2012) Virtex-5 FPGA configuration user guide. UG191 (v. 3.11), 19 Oct 2012
11. Xilinx (2012) Partial reconfiguration of Xilinx FPGAs using ISE design suite. WP374 (v. 1.2), 30 May 2012
12. Heiner J, Collins N, Wirthlin M (2008) Fault tolerant ICAP controller for high-reliable internal scrubbing. In: Proceedings of the IEEE aerospace conference, March 2008, pp 1–10
13. Ebrahim A, Benkrid K, Iturbe X, Hong C (2012) A novel high-performance fault-tolerant ICAP controller. In: Proceedings of the NASA/ESA conference on adaptive hardware and systems, June 2012, pp 259–263

14. Lamonnier S, Thoris M, Ambielle M (2012) Accelerate partial reconfiguration with a 100% hardware solution. Xcell J 79:44–49
15. Claus C, Muller FH, Zeppenfeld J, Stechele W (2007) A new framework to accelerate Virtex-II Pro dynamic partial self-reconfiguration. In: Proceedings of the IEEE international parallel and distributed processing symposium, Long Beach, March 2007, pp 1–7
16. Claus C, Zhang B, Stechele W, Braun L, Hubner M, Becker J (2008) A multi-platform controller allowing for maximum dynamic partial reconfiguration throughput. In: Proceedings of the international conference on field programmable logic and applications, Sept 2008, pp 535–538
17. Liu M, Kuehn W, Zhonghai L, Jantsch A (2009) Run-time partial reconfiguration speed investigation and architectural design space exploration. In: Proceedings of the international conference on field programmable logic and applications, Sept 2009, pp 498–502
18. Bhandari S, Subbaraman S, Pujari S, Cancare F, Bruschi F, Santambrogio MD, Grassi PR (2012) High speed dynamic partial reconfiguration for real time multimedia signal processing. In: Proceedings of the 15th Euromicro conference on digital system design, Sept 2012, pp 319–326
19. Hubner M, Gohringer D, Noguera J, Becker J (2012) Fast dynamic and partial reconfiguration data path with low hardware overhead on Xilinx FPGAs. In: Proceedings of the IEEE international parallel & distributed, workshops and PhD forum, April 2010, pp 1–8
20. Lai V, Diessel O (2009) ICAP-I: a reusable interface for the internal reconfiguration of Xilinx FPGAs. In: Proceedings of the international conference on field-programmable technology, Sydney, Dec 2009, pp 357–360
21. Bayar S, Yurdakul A (2008) Dynamic partial self-reconfiguration on Spartan-III FPGAs via a Parallel Configuration Access Port (PCAP). In: Proceedings of the HiPEAC workshop on reconfigurable computing, Goteborg
22. Bayar S, Yurdakul A (2008) Self-reconfiguration on Spartan-III FPGAs with compressed partial bitstreams via a parallel configuration access port (cPCAP) core. In: Proceedings of the Ph.D. research in microelectronics and electronics, Istanbul, June 2008, pp 137–140
23. Xilinx (2014) Device reliability report—first quarter 2014. UG116 (v. 9.8), 18 March 2014
24. JEDEC (2006) Measurement and reporting of alpha particle and terrestrial cosmic ray-induced soft errors in semiconductor devices JEDEC standard, Tech. Rep. JESD89A. http://www.jedec.org/sites/default/files/docs/jesd89a.pdf

Chapter 10
Multiple Fault Injection Platform for SRAM-Based FPGA Based on Ground-Level Radiation Experiments

Jorge Tonfat, Jimmy Tarrillo, Lucas Tambara, Fernanda Lima Kastensmidt, and Ricardo Reis

Abstract Fault injection by emulation is a well-known method to analyze the reliability of a circuit. SRAM-based FPGAs provide the hardware infrastructure to implement fault injectors taking advantage of dynamic partial reconfiguration. This chapter presents the details of a Multiple Fault Injection Platform and the analysis of the configuration memory upsets of the FPGA. Results of fault injection campaigns are presented and compared with accelerated ground-level radiation experiments.

10.1 Introduction

Field-Programmable Gate Arrays (FPGAs) nowadays are not only used for ASIC prototyping but also to replace them in some ground-level and space applications. SRAM-based FPGAs take advantage of the latest semiconductor fabrication processes, allowing high-density logic integration. This scenario allows them to achieve expected performance levels in a variety of applications. Moreover, the reconfigurability feature of SRAM-based FPGAs allows the same device to perform multiple functionalities during its lifetime.

These characteristics make SRAM-based FPGAs attractive to critical applications. But since configuration bits are stored into volatile SRAM cells, radiation effects can generate single or multiple bit-flips in the configuration memory. Such single event upsets (SEUs) or multiple bit upsets (MBUs) can induce functional errors in the implemented design. In order to tolerate these faults, many techniques were proposed in the literature. However, it is necessary to validate the efficiency of these techniques closest to the real effect as possible, but also considering the controllability, observability and cost.

Jimmy Tarrillo • Lucas Tambara • Ricardo Reis • J. Tonfat (✉) • F.L. Kastensmidt
Instituto de Informática, Universidade Federal do Rio Grande do Sul (UFRGS),
Porto Alegre, Brazil
e-mail: jorgetonfat@ieee.org; fglima@inf.ufrgs.br

F. Kastensmidt, P. Rech (eds.), *FPGAs and Parallel Architectures for Aerospace Applications*, DOI 10.1007/978-3-319-14352-1_10

Fault injection by emulation is an important method to predict in the early stages of the design phase the susceptibility of the design under upsets. Emulation of SEUs and MBUs by flipping the configuration bits on an FPGA is an attractive technique to evaluate the behavior of a design before it is working in radiation environments. In addition, fault injectors can take advantage of partial reconfiguration capabilities of FPGAs to reduce even more the time to inject upsets. The main goal of this approach relies on the fact that it allows fast injection campaigns, once the circuit under test (CUT) executes at the full FPGA speed and not on simulation speed.

Moreover, the amount of injected faults per unit of time (upset rate) is higher compared to radiation tests on particles accelerators because a bit-flip is directly injected in the memory cell. The control of the test is also superior compared to a radiation test, since a precise location is flipped (a known bit), which allows the user to reproduce a real radiation test.

The fault injection can be performed by an external or internal programmable port of the FPGA. The internal configuration access port (ICAP) [1] provides some advantages such as the possibility to reconfigure frame by frame without the necessity of using input/output pins. The ICAP can be controlled by the SEU controller macro [2] and an embedded soft-core as PicoBlaze; or by a specific control design developed by the user [3]. SEUs can be injected in the bitstream in random locations, sequentially (every configuration bit or configuration control register is flipped in sequential order), or user-defined.

10.2 Related Works

Other fault injection platforms are available to inject SEU in SRAM-based FPGAs as described in [4]. FLIPPER [5] that is targeted to Virtex-2 devices is one example. It uses a scheme based on a control motherboard and a DUT board. The fault injector is implemented in the mother-board FPGA and a host PC. The DUT board contains the target FPGA. The configuration memory of this FPGA is modified with partial reconfiguration using an external configuration port. In [6] the fault injector and the DUT are implemented in the same FPGA and in order to inject faults a host PC creates faulty bitstreams. FT-SHADES [7] and [8] are other examples of fault injectors but in this case they use an internal injection approach using the ICAP to inject single faults in the bitstream.

With internal fault injection [7–9], we do not need to reconfigure the entire FPGA, so the fault injection speed is increased, but a problem arises. The quality of the fault injection can be reduced by fault injection side-effects as shown in [9]. A fault injected in the configuration memory can affect the fault injector itself. So the fault injection can stop unexpectedly or even worst, the fault injector can wrongly report that a fault is injected.

In this work, we present a multiple fault injector platform able to emulate SEU and MBU in the configuration memory bits of an SRAM-based FPGA. Our goal is to replicate the effects of radiation to validate protection techniques and improve the

radiation test methodologies and test plans under accumulated multiple faults. The proposed Fault Injection Platform uses the ICAP module to flip a configuration bit, and takes the bit location from an external database bank. The bit-flip locations were taken from previous experiments in neutron radiation test from ISIS facilities [10] and also generated by a MATLAB pseudo-random generator. During the fault injection procedure, the fault injector takes the necessary actions to guarantee a correct fault injection and minimize the side-effects improving the quality of the results.

10.3 Hardware Implementation of the Multiple Fault Injection Platform

The proposed Multiple Fault Injection Platform is composed of a single SRAM-based FPGA, a flash-based external memory and a host computer. We use the Digilent Genesys prototype board containing a Xilinx Virtex-5 FPGA, part XC5VLX50T-FFG1136 and other resources. For our fault injection platform, we use the external flash memory connected to the FPGA to store the bit-flip locations. This memory stores the SEU locations database bank. A block diagram of the Multiple Fault Injection Platform is shown in Fig. 10.1.

The FPGA contains the DUT (Design Under Test) and the fault injector. It is well-known that internal injectors suffer from side-effects because an injected fault can provoke an error on the injector itself. But to mitigate these effects, the fault injector can avoid bit-flips in its configuration bits.

The fault injector is composed of an ICAP controller, a flash memory controller and a PicoBlaze 8-bit soft processor.

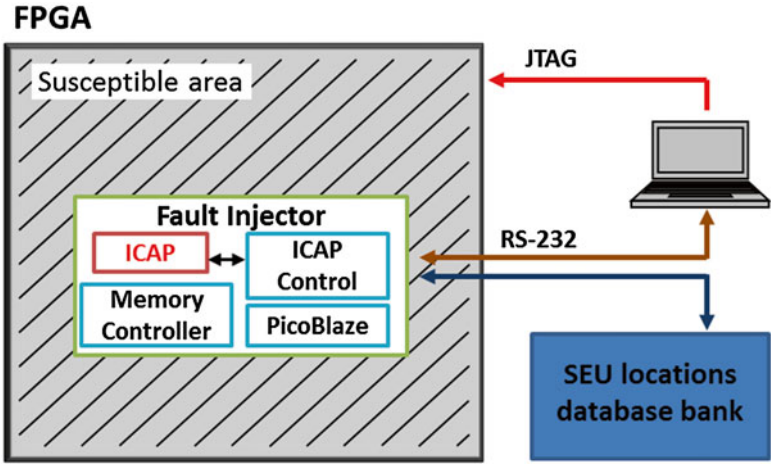


Fig. 10.1 Architecture of the Multiple Fault Injection Platform

Table 10.1 Frame address field descriptions

Field	Description
Type	Defines the type of frame. Can be a configuration frame (type 0), BRAM content (type 1) and other two types not well documented in the literature
Top/bottom	Defines the half (top or bottom) of the FPGA where the frame is located
Row	Defines the frame row. The row number increases from the middle of the FPGA
Column	Defines the frame column. A column is defined by the type of resource (ex. CLB, DSP, etc.)
Frame in column	Defines the frame position inside the column

The main function of the PicoBlaze is to control the execution of the fault injection campaign. The ICAP controller manages all the commands to read and write frames from the configuration memory using the ICAP. The ICAP is the interface that enables access to the configuration memory from an internal circuit in the FPGA. With a suitable set of commands, we can modify the configuration memory without stopping the application running in the FPGA. This method is also known as dynamic partial reconfiguration.

In order to control the ICAP, we must understand the configuration memory of the FPGA and the way to read and write in this memory.

10.3.1 Organization of Virtex-5 FPGA Configuration Memory

The FPGA can be seen as a device with two layers. One is the logic layer that includes all the user application resources such as the Configurable Logic Blocks (CLB), the Block RAMs, I/O blocks, etc. The other is the configuration layer that comprises the configuration memory and the associated access ports.

Understanding the organization of the configuration memory will allow us to know the relation between configuration bits and resources of the FPGA.

The following information is based on the Virtex-5 Configuration User Guide [1].

The FPGA configuration memory is composed of small memory segments called *configuration frames*. So a configuration frame is the smallest addressable segment of the FPGA configuration memory, and the frame size varies among FPGA families. In the case of Virtex-5, it is composed of 41 words of 32 bits (1,312 bits).

Each frame has a unique address that is related to the physical position in the FPGA floorplan. Each frame address has five fields. Each field is described in Table 10.1 and corresponds to the organization of the FPGA floorplan.

Due to this organization, frame addresses are not consecutive. A graphical description of the organization of the floorplan is shown in Fig. 10.2.

The floorplan is divided into two main regions: top and bottom. Each region is organized in rows and columns. One frame has the height of a row, and the columns are organized according to the type of resource (ex. CLB, BRAM, DSP, etc.). Each

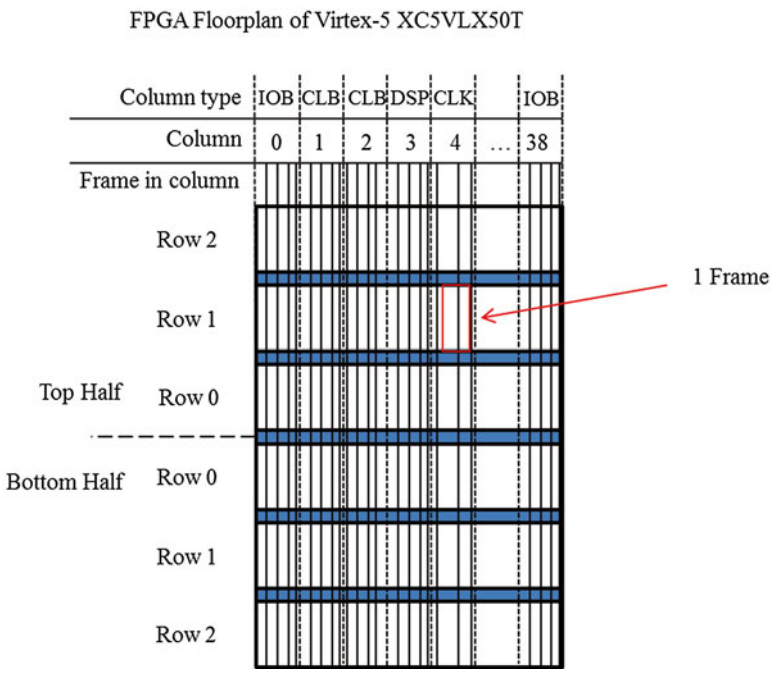


Fig. 10.2 Example of the organization of the configuration memory of a Virtex-5 FPGA

Table 10.2 Number of frames per column

Column type	Number of frames
CLB	36
DSP	28
Block RAM (configuration)	30
IOB	54
CLK	4

column contains a group of frames. The number of frames on each column depends on the type of column as shown in Table 10.2.

Depending on the device selected, some of the frames in this organization are not implemented. This case is common for IOB columns, where not all the rows of an IOB column have the corresponding frames since the IOB resources depend on the number of pins of the FPGA.

10.3.2 Methodology for a Fault Injection Campaign

With the information about the organization of the configuration memory and the specific commands sequence to read and write frames, we can flip any bit of the configuration memory thus emulating the effect of an SEU.

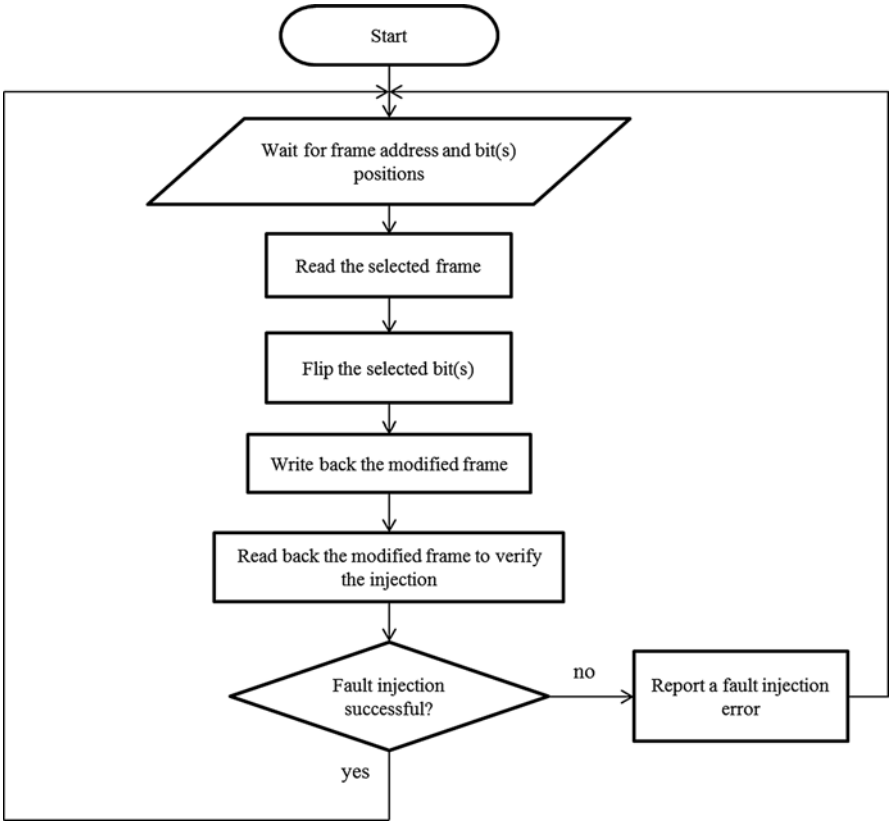


Fig. 10.3 Flow diagram of the procedure to inject one fault

Figure 10.3 shows the procedure executed by the ICAP controller to inject one fault into the configuration memory. The only information needed to flip a bit is the selected frame address and the selected bit inside this frame. This information comes from the SEU database stored in the external memory and is managed by the PicoBlaze soft processor. It is important to mention that this method can also emulate intra-frame multiple bit-flips.

Since the smallest segment of the configuration memory is a frame, the ICAP controller needs to read the entire frame and store it in a temporal buffer. Then the selected bit(s) position(s) are flipped. Finally, the modified frame is written back to the configuration memory. In order to verify the correct insertion of the fault, the frame is read back again and compared to the modified frame stored in the temporal buffer. If differences are found between them, the ICAP controller reports a fault injection error.

Most of the time injection errors are due to the inexistence of the selected frame address in the FPGA as mentioned in the previous section. This type of error injection does not interfere with our results since these missing frames cannot be flipped

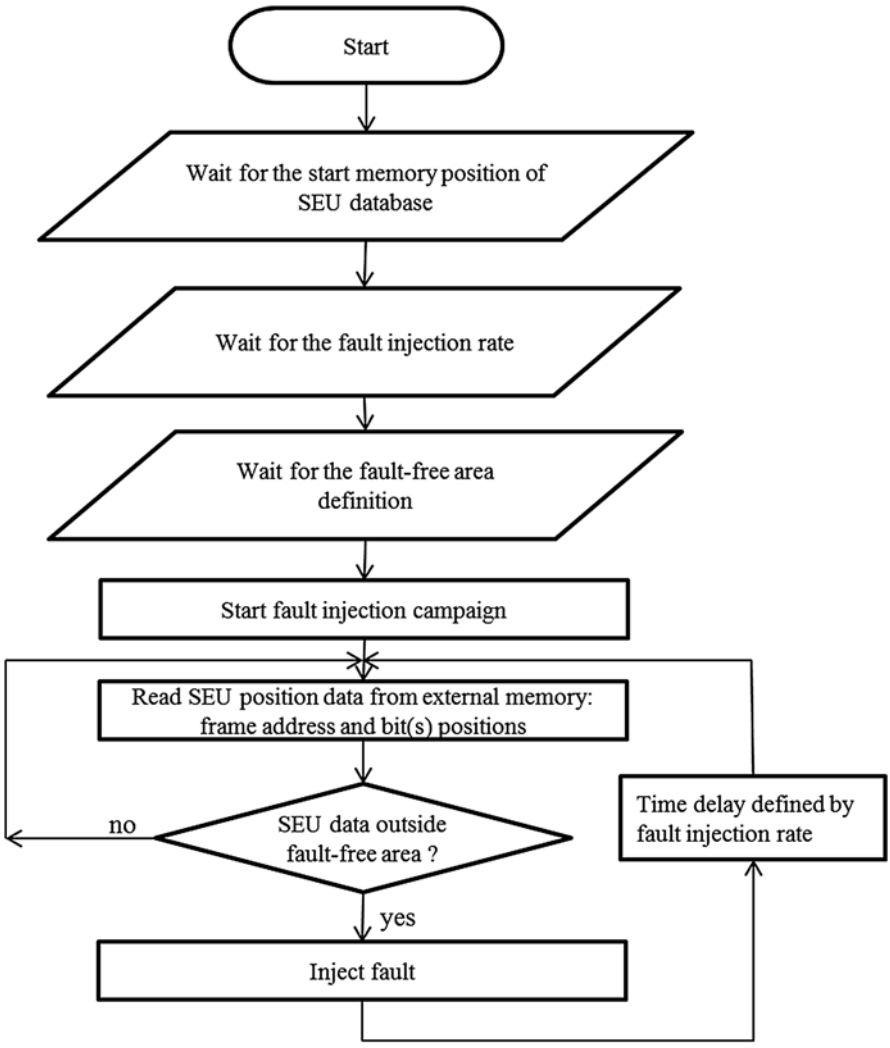


Fig. 10.4 Flow diagram of the procedure to control a fault injection campaign

by real SEUs. The ICAP controller reports failed injections to take into account this information when the fault campaign report is generated.

So a complete fault injection is completed in 310 clock cycles. With a clock frequency of 50 MHz, one injection is completed in 6.2 µs.

The PicoBlaze manages the execution of a complete fault injection campaign. The procedure is described in Fig. 10.4. The procedure starts with the definition of the parameters of the campaign. These parameters are the start memory position of the SEU database, the fault injection rate and the definition of the fault-free area.

The start memory position of the SEU database is the reference point to the PicoBlaze in order to read consecutively from this point the bit-flip data stored in the external memory. The fault injection rate defines the amount of faults injected per time unit. This parameter can be used to emulate different radiation environments.

The definition of the fault-free area is to protect the circuits that can interfere with the execution of the fault injection campaign. For instance, the fault injector area needs to be included in this protected area. This method minimizes the possibility of a functional error in the fault injector itself that is one of the side-effects of internal fault injection. Other circuits that can be included are, for example, the circuit that controls the execution of the DUT. Since a functional error in this block can generate a false functional error of the DUT, we must protect this block from bit-flips. The fault-free areas need to be in agreement with the placement constraints set during the design implementation phase.

So when the fault injection campaign starts, each SEU position read from the external memory is analyzed to determine if it is inside the fault-free area. When the bit-flip position is inside the protected area, the bit-flip is not injected, and the next SEU position is loaded. If not, the PicoBlaze commands the ICAP controller to inject the corresponding fault.

At the top level, the host PC is in charge of the execution of multiple fault injection campaigns. The procedure is shown in Fig. 10.5. The first step is to set the corresponding parameters.

The first parameter is the maximum time for a single fault injection campaign. This time is variable and depends on the DUT and the fault injection rate. This setting helps to determine when a fault injection campaign reaches an unknown state.

The start memory position of the SEU database defines the starting point of the first fault injection campaign. The subsequent campaigns will start from the last injected SEU position. In this way, we assure different SEU patterns for each fault injection campaign.

The fault injection rate and fault-free areas are also defined. These parameters can be fixed for all the fault injection campaigns or can be variable among campaigns according to the user needs.

When all parameters are set, the host PC configures the FPGA with the DUT and the fault injector module through the JTAG interface and the fault injection campaigns begins.

To recognize the end of a fault injection campaign, it is necessary a DUT end condition event. In our case, we want to test the maximum number of accumulated faults that a design can tolerate before it starts to fail. When it reaches a certain condition, the DUT sends a signal that is captured by the host computer. It also receives the information of SEU positions injected and the information when a fault injection has failed.

The fault injector was implemented into the XC5VLX50T FPGA on the Genesys Digilent board and the synthesis result is detailed in Table 10.3.

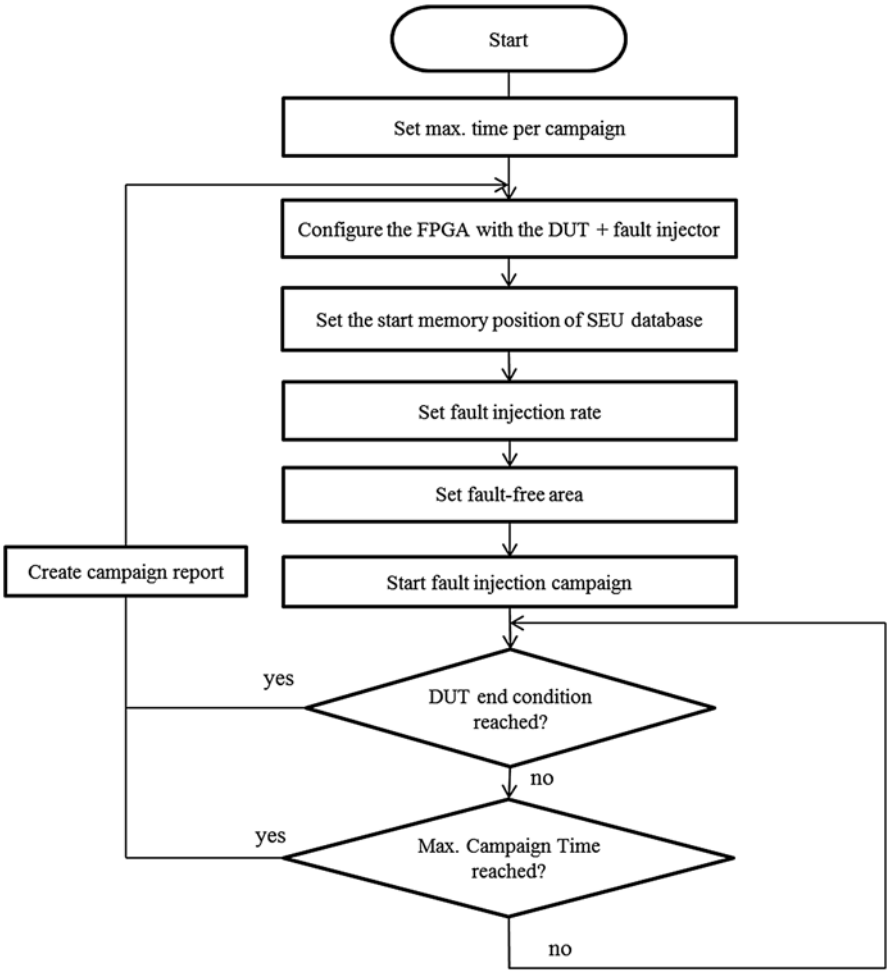


Fig. 10.5 Flow diagram of the procedure to control multiple fault injection campaigns

Table 10.3 Resource utilization of the fault injector

	LUTs	Registers	Block RAMs
PicoBlaze soft processor	147	76	1
Flash memory controller	86	68	0
ICAP controller	705	417	1
Total	938	561	2

10.4 Methodology for Capturing and Modeling Single Bit Upsets

The injected faults are modeled mainly with two different approaches:

- By using a radiation database from previous radiation experiments.
- By using a computer generated database based on a pseudo-random generator with a uniform distribution.

10.4.1 Modeling Using Data from Previous Ground-Level Radiation Experiments

The database is composed of multiple and accumulated faults in Virtex-5 FPGA. These faults were obtained from previous radiation experiments at ISIS facilities of Rutherford Appleton Laboratory (Didcot, United Kingdom).

During the tests, bit-flips in the configuration memory were detected using a readback procedure as described in Fig. 10.6. It is important to mention that this procedure logs bit-flips in the configuration memory and the content of block RAMs. So we use the mask file (generated by Xilinx tools) to filter our logs from bit-flips in block RAMs and bit-flips due to shift registers or LUT RAMs used by the DUT.

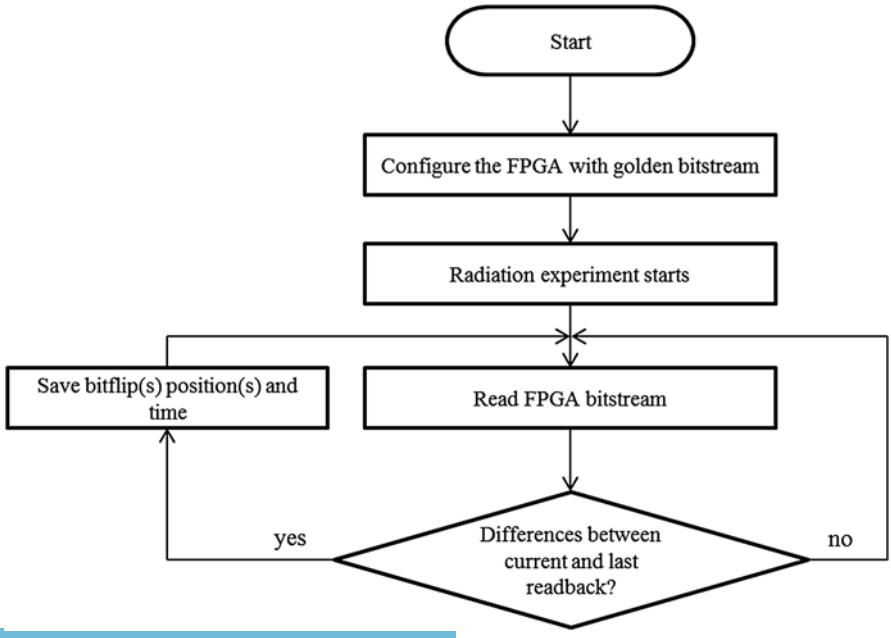


Fig. 10.6 Procedure to capture bit-flips in the configuration memory

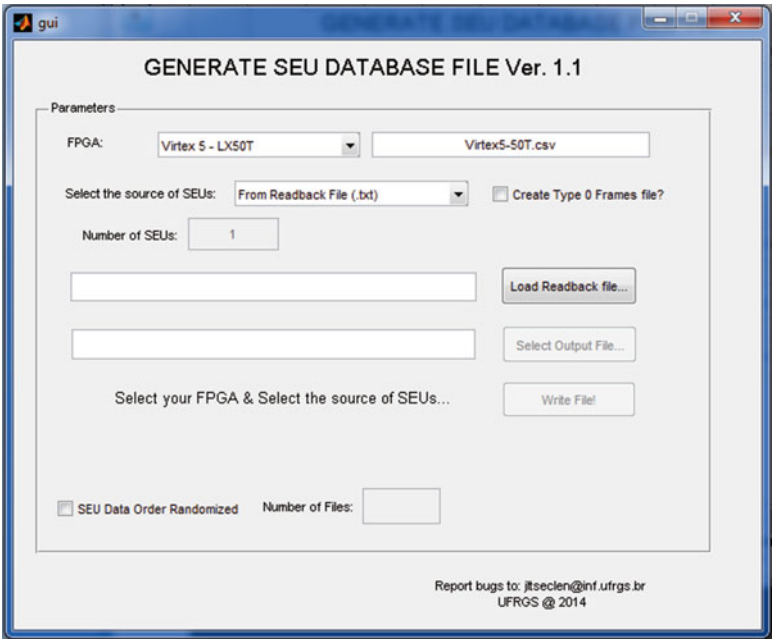


Fig. 10.7 GUI of the tool to create SEU databases

Based on our knowledge of the FPGA configuration memory and the readback bitstream, we can precisely determine the frame address and bit position of each SEU registered during the experiment. The location of the bit-flip is the information needed by the fault injector to inject a bit-flip.

We developed a software tool to automate this process. The tool takes the text reports from the radiation experiments and creates the binary file for the external flash memory automatically. Figure 10.7 shows a screenshot of the GUI of this tool.

In our previous radiation experiments, more than 2,600 SEUs were identified. This information is stored in the external flash memory. In the case of the Genesys board, it has a flash memory of 256 Mbit (organized as 16-bit by 16 Mbytes) for non-volatile storage of FPGA configuration files. We used three memory addresses to store the information of each SEU. The first two positions store the frame address and the last position store the bit position inside the frame. So, up to five million SEUs can be stored in this memory.

10.4.2 Modeling SEUs Using Computer Generated Data

Based on the analysis of the accumulated bit-flips obtained from radiation experiments at ISIS, we also generate bit-flips locations that resemble the original ones. We achieve this using MATLAB and a pseudo-random generator with a uniform

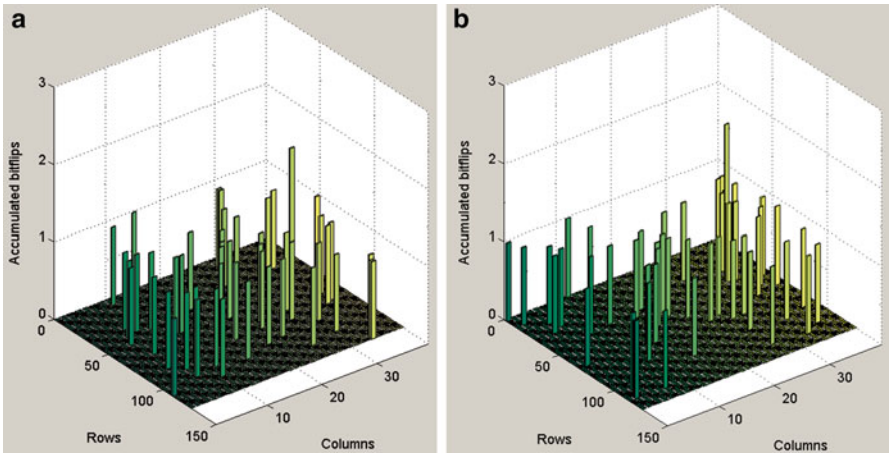


Fig. 10.8 Comparison of bit-flips from radiation experiments and MATLAB generated. (**a**) 50 ISIS bit-flips, (**b**) 50 MATLAB generated bit-flips

distribution. Figure 10.8 shows a graphical comparison between collected bit-flips and generated bit-flips. Each bar represents the number of accumulated bit-flips per resource in the FPGA (ex. 1 CLB). The color scale is only for visualization purposes. In the case of the Virtex-5 XC5VLX50T FPGA, the resources form a matrix of 120 rows by 39 columns.

The option to generate bit-flips is also included in the same tool that creates the SEU database from radiation experiments.

10.5 Fault Injection Campaign Results and Comparisons

In order to validate the fault injection platform, we have evaluated one case study design. Then we have compared the fault injection results with the neutron radiation experiments results.

This design implements an N-modular redundancy (nMR) scheme as a technique to tolerate multiple fault accumulation. The nMR is composed of n functionally identical modules, which receive the same m-bits input and deliver p-bits output to the Self-Adapted voter (SAv), Fig. 10.9 [11].

The SAv receives $n \times p$ bits from all modules and generates the fault-free p-*output*, n-error status flags (ESF), and a non-masked fault signal (NMF). In this scheme, the system allows the accumulation of defective modules, until remaining at least two modules without fault. The SAv is a majority voter, considering as population fault-free modules.

The implemented design is a 7-MR adder chain. The architecture is shown in Fig. 10.10. The criteria for selecting this design were the low logic masking of faults

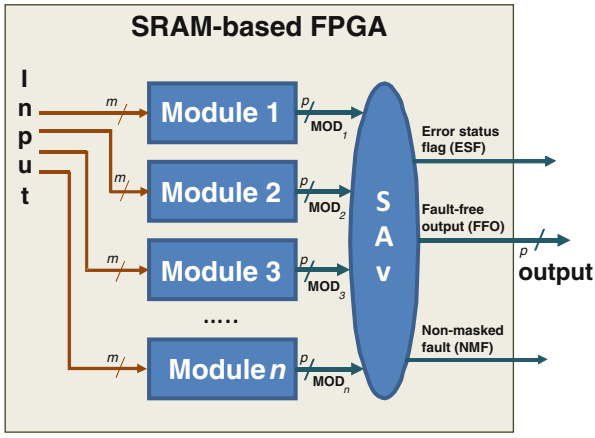


Fig. 10.9 nMR-based technique with SAv voter

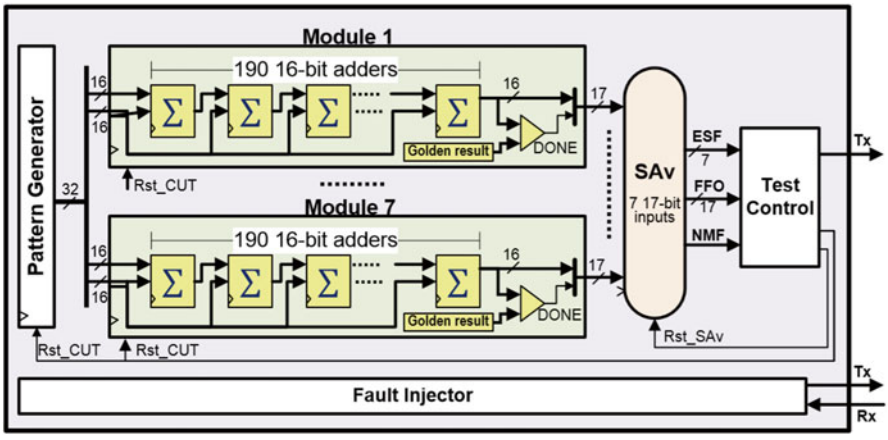


Fig. 10.10 Block diagram of the adders chain DUT and the fault injector

and the ease to scale. This design has a control module to manage the input pattern generator of the adder chains and to monitor the correct response of the 7-MR system. When a functional error is detected, the control block sends error signals to the host PC, and the fault injection campaign ends.

Figure 10.11 shows the final placement of the 7-MR adder chain and the fault injector. The areas of the fault injector and the control module are included in the fault-free area of the fault injector.

The objective of the test is to determine if the fault injector can predict the tolerance of this design under neutron radiation. So the test reports the number of accumulated faults needed to provoke the failure of each of the seven modules. The end condition of the test is when only two correct modules remain.

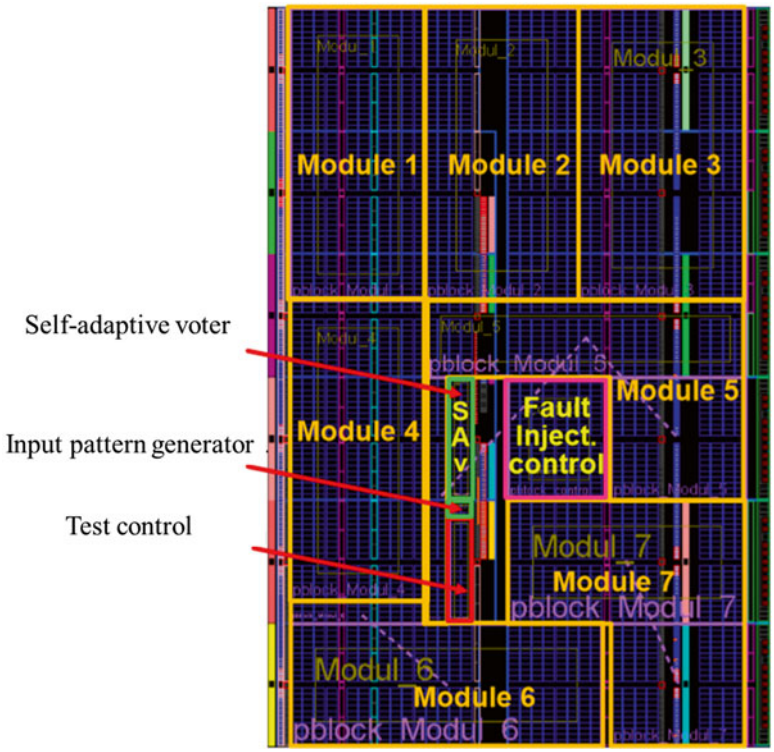


Fig. 10.11 Placement of the adders chain DUT and the fault injector

Figure 10.12 presents the results of the fault injection campaigns. We run 25 injection campaigns and it was injected an average of 98.33 faults per campaign.

Figure 10.13 shows the results from the radiation experiment. Due to beam time restrictions, we were able to run the test few times.

And Fig. 10.14 shows the comparison between the results from fault injection and radiation experiments. Both present similar average accumulated faults for each of the faulty modules count.

10.6 Conclusions

This work presents a multiple fault injection platform to evaluate accumulated SEU effects in Virtex-5 FPGA. The platform uses bit-flip positions generated by a pseudo-random generator or taken from a database composed of pre-collected real bit-flips location detected from previous neutron accelerated experiments at ISIS facilities. The flipped bits distribution of real radiation test and fault injector were shown and analyzed. Also, the effects of accumulation SEUs on a design using real

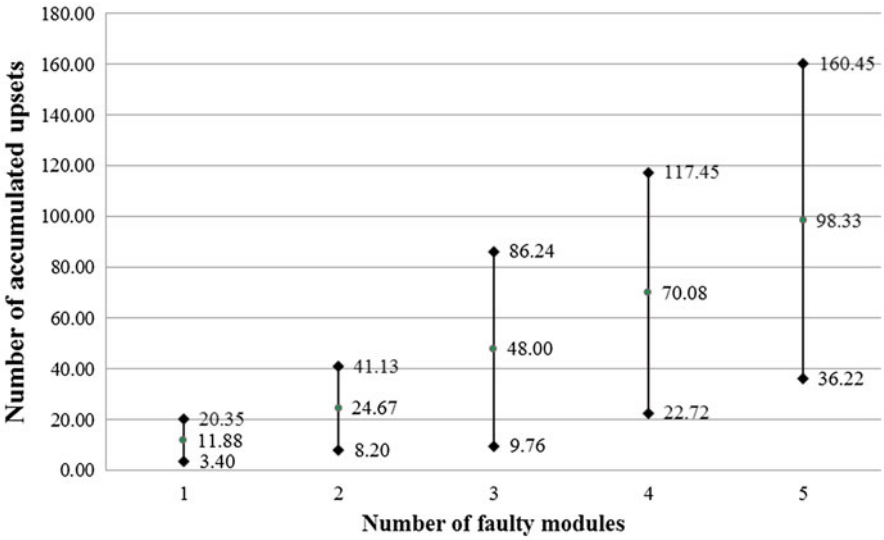


Fig. 10.12 Number of accumulated faults needed to provoke multiple faulty modules under fault injection for the adder chain case-study

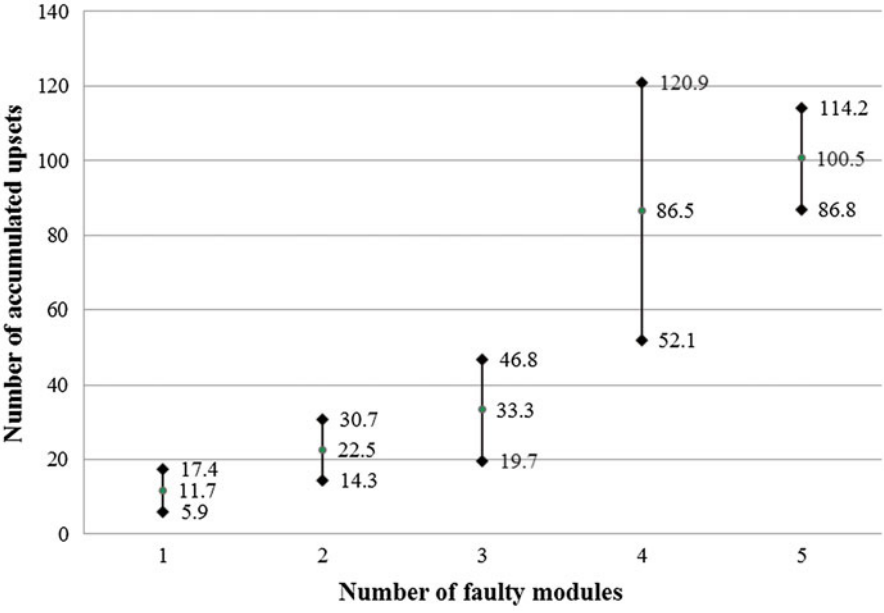


Fig. 10.13 Number of accumulated faults needed to provoke multiple faulty modules under radiation experiment for the adder chain case-study

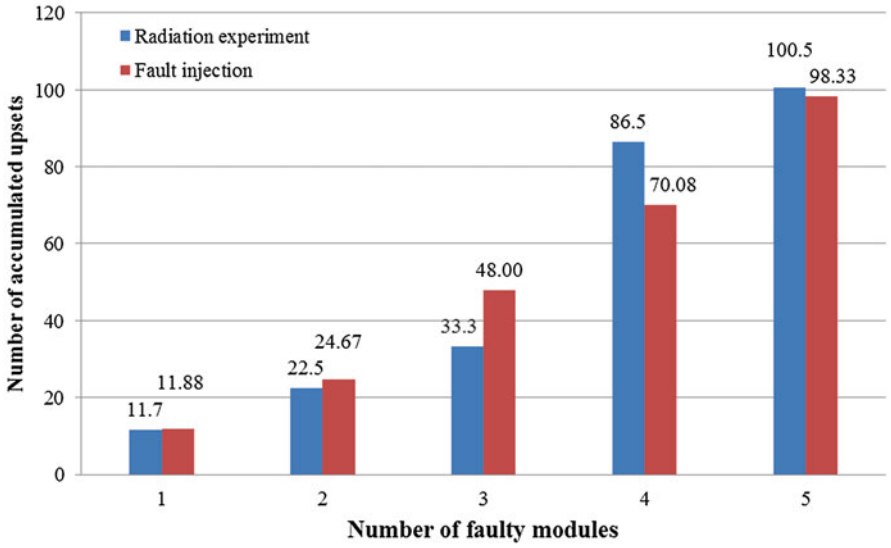


Fig. 10.14 Comparison between fault injection and radiation experiment results of adder chain case study

radiation test and fault injection were tested. Results show the real capability of the platform proposed to predict the effects of radiation in FPGA designs and mitigate successfully the side-effects related to internal fault injectors.

References

1. Xilinx, UG191 (2012) Virtex-5 FPGA configuration user guide, 19 Oct 2012
2. Chapman K (2010) SEU strategies for Virtex-5 devices. XAPP864 v2.0
3. Tarrillo J, Escobar FA, Lima Kastensmidt F, Valderrama C (2014) Dynamic partial reconfiguration manager. In: 2014 IEEE 5th Latin American symposium on circuits and systems (LASCAS), 25–28 Feb 2014, pp 1–4
4. Alexandrescu D, Sterpone L, Lopez-Ongil C (2014) Fault injection and fault tolerance methodologies for assessing device robustness and mitigating against ionizing radiation. IN: 2014 19th IEEE European test symposium (ETS), 26–30 May 2014, pp 1–6
5. Alderighi M, Casini F, Citterio M, D'Angelo S, Mancini M, Pastore S, Sechi GR, Sorrenti G (2008) Using FLIPPER to predict irradiation results for VIRTEX 2 devices. In: Radiation and its effects on components and systems (RADECS), pp 300–305
6. Sterpone L, Violante M, Rezgui S (2006) An analysis based on fault injection of hardening techniques for SRAM-based FPGAs. IEEE Trans Nucl Sci 53(4):2054–2059
7. Guzman-Miranda H, Tombs JN, Aguirre MA (2008) FT-UNSHADES-uP: a platform for the analysis and optimal hardening of embedded systems in radiation environments. In: IEEE international symposium on industrial electronics, ISIE 2008, pp 2276–2281
8. Nazar GL, Carro L (2012) Fast single-FPGA fault injection platform. In: 2012 IEEE international symposium on defect and fault tolerance in VLSI and nanotechnology systems (DFT), 3–5 Oct 2012, pp 152–157

9. Kretzschmar U, Astarloa A, Jimenez J, Garay M, Del Ser J (2014) Compact and fast fault injection system for robustness measurements on SRAM-based FPGAs. IEEE Trans Ind Electron 61(5):2493–2503
10. Violante M, Sterpone L, Manuzzato A, Gerardin S, Rech P, Bagatin M, Paccagnella A, Andreani C, Gorini G, Pietropaolo A, Cardarilli G, Pontarelli S, Frost C (2007) A new hardware/software platform and a new 1/E neutron source for soft error studies: testing FPGAs at the ISIS facility. IEEE Trans Nucl Sci 54(4):1184–1189
11. Tarrillo J, Rech P, Kastensmidt F, Valderrama C, Frost C (2013) Neutron cross-section of N-modular redundancy technique in SRAM-based FPGAs. In: 2013 14th European conference on radiation and its effects on components and systems (RADECS). IEEE, Oxford, pp 1–6

Part IV
Flash-Based FPGAs

Chapter 11
Radiation Effects in 65 nm Flash-Based Field Programmable Gate Array

Jih-Jong Wang, Nadia Rezzak, Durwyn DSilva, Chang-Kai Huang, Stephen Varela, Victor Nguyen, Gregory Bakker, John McCollum, Frank Hawley, and Esmat Hamdy

11.1 Introduction

Since it was first introduced, Flash-based FPGA had been well received by digital designers in aerospace and high-reliability applications. Its popularity owes to, unlike other commercially available FPGA based on antifuse or SRAM technologies, that the Flash-based FPGA has the unique advantage of being both non-volatile and reprogrammable. It is advantageous to antifuse-based for programmability and to SRAM-based for non-volatility. This characteristic warrantees small foot-print and resiliency in hazardous operating environment, especially against bit-errors by particle radiations.

Its development has been successfully following footsteps of continuously scaled CMOS technologies. Architecturally the first product, 0.25 μm ProASIC, is simple. It has tiles of user logic and embedded-SRAM blocks which have dual usage either as two-port SRAM or FIFO. The second product, 0.22 μm ProASIC^{PLUS}, is an improved ProASIC with similar capability. The third product, 130 nm ProASIC3, has many new and advanced features and it quickly replaces the previous FPGAs as the main force to present day. The most significant improvement in ProASIC3 is using standard digital-CMOS power supply of 3.3 VDC to perform the in-system programming. This is achieved by integrating a charge pump to provide high voltage on-chip for the programming. Also, the derived siblings, Igloo, Fusion, and SmartFusion1, have special features of low-power operation, and embedded Intellectual Properties (IPs) to provide wide spectrum of functions [1]. The introduction of SmartFusion1 is a significant milestone because it is the first Flash-based

J.-J. Wang (✉) • N. Rezzak • D. DSilva • C.-K. Huang • S. Varela • V. Nguyen • G. Bakker
J. McCollum • F. Hawley • E. Hamdy
Microsemi SOC, San Jose, CA, USA
e-mail: jih-jong.wang@microsemi.com

F. Kastensmidt, P. Rech (eds.), *FPGAs and Parallel Architectures for Aerospace Applications*, DOI 10.1007/978-3-319-14352-1_11

FPGA to be also an SOC. Indeed, it has and embedded hard-wired ARM Cotex-1 microcontroller to enable the full function of a digital system.

Radiation-induced TID effects in Flash-based FPGA have been studied by research groups [2–9]. These TID effects include Flash cell V_T shift, propagation delay degradation, power-supply current increase, FPGA function failure, and programming failure. In general, all radiation-induced changes of parameters can be related to known physical mechanisms: charge loss/gain in floating gate of irradiated Flash cell, which can cause threshold-voltage shift [10]; leakage current increase, timing skew and functional failures in CMOS transistors [11, 12].

The studies on single event effects (SEE) of Flash-based FPGA are abundant [13–21], especially on heavy-ion induced single event transients (SET) in 130 nm Flash-based FPGA. Even an SET-mitigation software package is available for ProASIC3 users [19]. Beside the practical usage reason, there is a valid motivation studying SET by using Flash-based FPGAs: first the continuing decreasing transistor sizes exacerbates the SET effects; second using Flash-based FPGA to study SET is very convenient because it is reprogrammable but doesn't have radiation-induced configuration upset which will plague the operation of SRAM-based FPGA.

This chapter will focus on the radiation effects in 65 nm Flash-based FPGA-SOC: The characteristics of this new Flash-based FPGA will be introduced; similarities and differences between the Flash cell used in FPGA and Memory applications will be highlighted; radiation tests results showing TID and SEE effects will be presented and discussed. Qualitative models will be constructed to elucidate how the physical mechanisms caused the observed radiation effects. Based on test data, single event upsets on the Flash configuration cell, fabric flip-flop, and fabric SRAMs are evaluated. A novel 3D-TCAD simulation generated SEU cross-sections on fabric FF will be compared with the test data, and its usefulness in the future will be contemplated.

11.2 Flash Configuration Cell

The Flash memory technology, meaning floating-gate (FG) technology here, had been studied and published extensively in recent years. The motivation mainly was driven by enormous commercial activities. Relevant knowledge such as device physics, circuit design, programming system operation, and reliability can be found in review literatures (e.g. see reference [22]). In this section, the Flash configuration cell in FPGA will be introduced and its references to Flash memory are often made.

The Flash configuration cell has similarities and differences when compared to a Flash memory cell. Like a memory cell, it also uses floating-gate NMOS transistor as the basic device to enable non-volatility. The physical mechanisms, in Write mode, for both Program and Erase action, are the well-known channel Fowler-Nordheim tunneling. However, its geometry is significantly different from that of a memory cell: the Flash cell enabling configuring, named "sense" device, combines with a Flash cell gating critical signals, named "switch" device, to form a twin

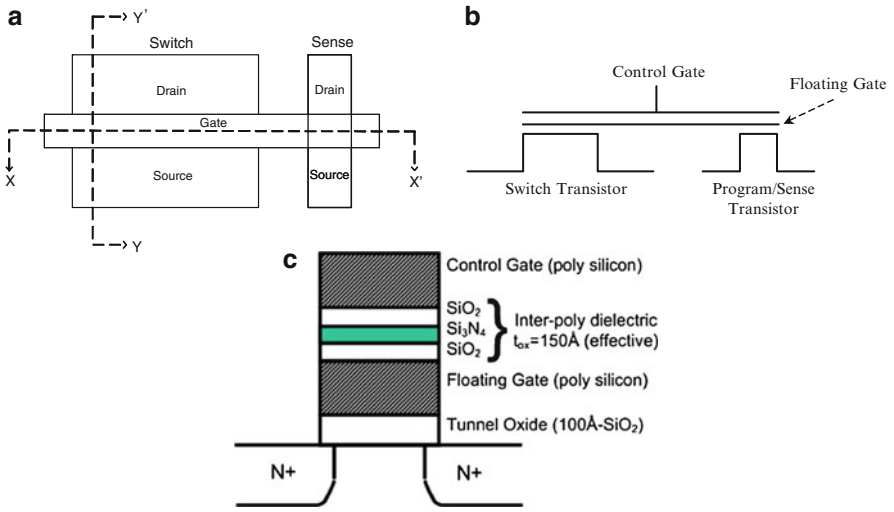


Fig. 11.1 (**a**) Layout of the Flash cell: each cell contains one switch and one sense FG transistor; the control gate and FG are shared by both the switch and sense transistor. (**b**) Schematic showing the cross-section of X-X′ cut. (**c**) Schematic showing the cross-section of Y-Y′ cut

structure shown in Fig. 11.1. The floating and control gate are shared by sense and switch devices. Note that the switch is the wider one for ease passing of signals.

When a Flash-based FPGA is used in a system, during configuration programming and testing Write/Read is performed through the sense device, and during normal operation Read is performed on the switch device. Figure 11.2 depicts a 2×2 array of Flash configuration cells to illustrate these actions. It also shows that by using the sense-switch construct in a single cell greatly simplifies the design enabling the FPGA operation while leaves the implementation of the Flash technology very much the same as that of the Flash memory. Indeed, the sense devices are arranged exactly the same as a typical NOR-Flash memory. The reason of using NOR architecture is that FPGA is performing normal operation function by reading code stored in Flash cell in executed-in-place (XIP) mode. The drawback is that, in radiation environments, NOR-Flash is more sensitive to TID effects than NAND [23].

Similar to Flash-memory operation, Write action programs the Flash-configuration cell into one of the state of Erase or Program. The FG transistor of a cell at the Erase state has a low threshold voltage (V_t) and at Program state high V_t. Note that the V_t measurement in Flash-configuration cell can be performed on either sense or switch device. Another difference, in FPGA the FG at Erase state is in the depletion mode (Fig. 11.3) while in memory it is usually not programming Erase-state into depletion mode.

Finally, during normal operation switch device at Erase state is the On-state passing signals and Program state the Off-state isolating logic circuits from adverse effects during operation. To pass robust signal, the switch device is designed to have a large enough width and this makes the total area of a Flash-configuration cell

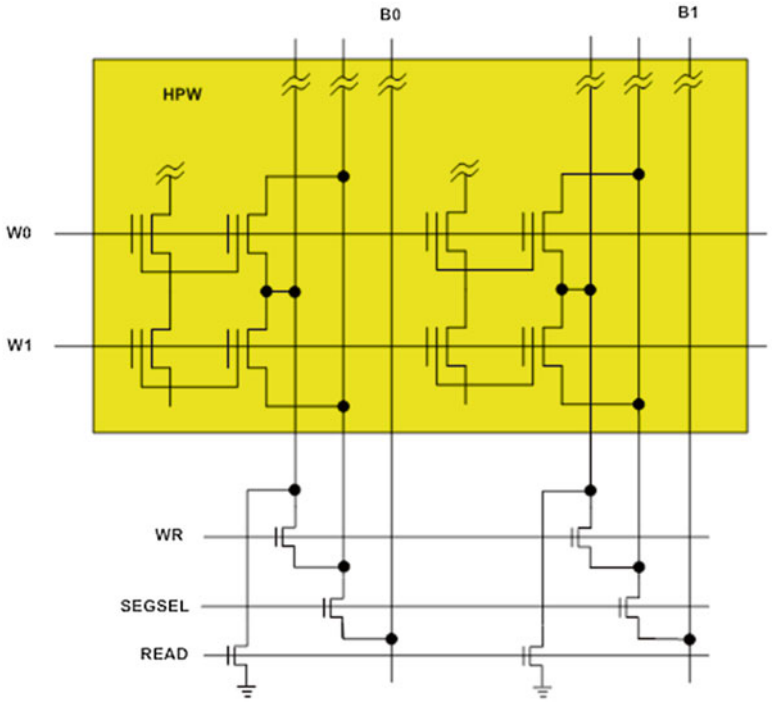


Fig. 11.2 Schematic of a 2×2 flash-cell array shows sense transistors arranged in typical NOR Flash-memory architecture

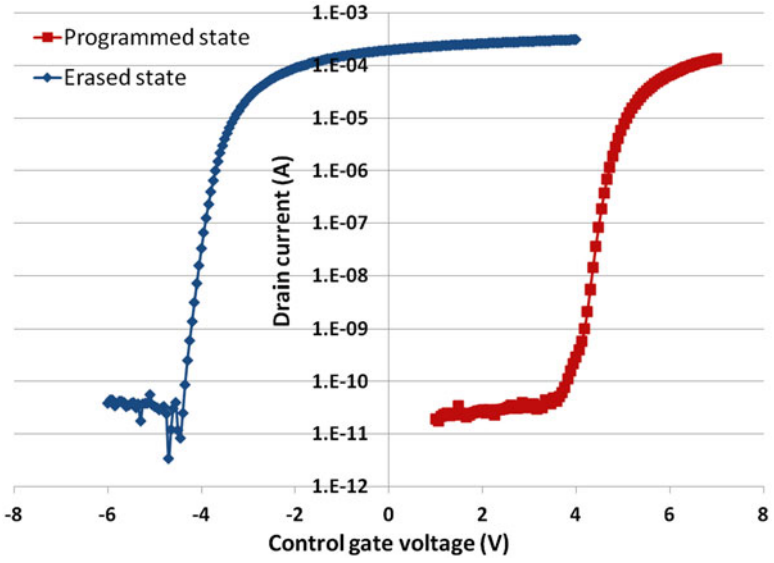


Fig. 11.3 Id-Vg plot of typical Program state and Erase state of a Flash configuration cell in 65 nm Flash-based FPGA

large, approximately 35 μm^2 in area, and significant larger than that of a Flash-memory cell. This area difference will be reflected in the difference between their radiation effects to be discussed in the following sections.

11.3 Radiation Testing

A device in SmartFusion2 family, coded M2S050, was radiation tested for total ionizing dose (TID) effects and single event effects (SEE). It is true silicon-on-chip (SOC) device manufactured by United Microelectronics Corporation (UMC) using 65 nm wafer-fabrication technologies. Figure 11.4 shows its floor plan indicating the location of each functional block. The device reliable Flash-based fabric logic and SRAM, and embedded with an ARM® Cortex™-M3 microprocessor together with instruction cache and advanced security processing accelerators, digital signal processing (DSP) blocks, eSRAM, eNVM, and industry-required high-performance communication interfaces. SmartFusion2 also differentiates itself from FPGAs using other configuring technologies by low power capabilities, high reliability and advanced security which is particularly important for military, aviation, communication and medical applications.

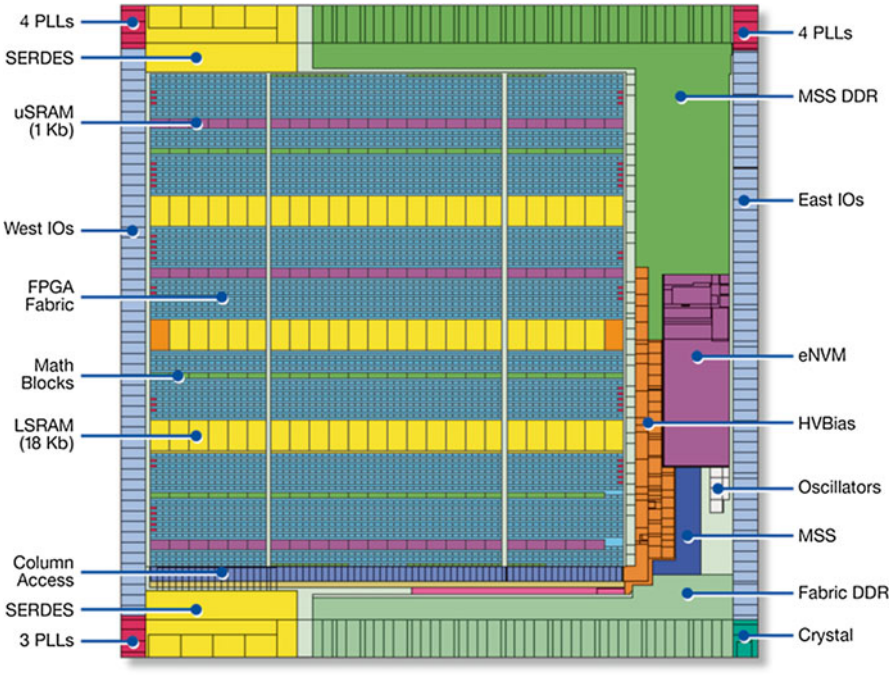


Fig. 11.4 Plot shows floor plan of M2S050 device and the location of each functional block

By no means can radiation effects, especially SEE, of an FPGA-SOC be completely tested at this moment. Here the focus is on the core configurable part of FPGA designed by Microsemi, which is often referred as fabric. The extra embedded IPs to make FPGA an SOC are hard-wired ASICs; their radiation tests, albeit very important, will be investigated in the future and not in the scope of this chapter.

SmartFusion2 family is not designed for applications operated in harsh radiation environments such as satellite operating in geosynchronous orbit. However, for moderate radiation environments, e.g. in particle accelerator, it can be very attractive for its non-volatile configuring ability and mild resistance to radiation effects.

Test dies with transistor level devices as well as FPGA dies are co-manufactured by wafer fabrication processes. Their purpose is to be tested standalone to facilitate the understanding of radiation effects at the transistor level, and subsequently helps to elucidate the radiation effects at the circuit and system levels.

11.3.1 Radiation Testing for TID Effects

The radiation testing performed on test chips was conducted at Vanderbilt University in Nashville, Tennessee, using ARACOR X-ray Irradiator. The testing on FPGA was at defense microelectronics activity (DMEA) in McClellan, California, using gamma ray irradiator. Both testing were performed at ambient temperature.

On the test chip, the Flash cell are programmed and tested by an Agilent 4156 controlled by a laptop PC. The CMOS transistors are tested using the same hardware/software. For propagation delay measurement, the design programmed in FPGA is a long inverter-string with 7,200 stages. Electrical data are recorded over the entire irradiation duration to finally more than 100 krad(SiO_2). The input signal is supplied from a function generator and waveforms of the input/output signals are observed and the propagation delay is recorded on the oscilloscope. The in-flux standby power-supply currents I_{DD} are monitored by an Agilent 6629 power supply and recorded by the laptop PC.

11.3.2 Radiation Testing for Single Event Effects

The test designs, illustrated in Fig. 11.5: shift registers consisted of various stages of configured fabric-flip-flops (FF) and fabric-SRAM blocks which include both μSRAM and LSRAM types. Figure 11.6 depicts a fabric Logic Element from which D-type FF with active low clear (DFN1C0) [24] is configured to be the testing target for SEU. The test setup is illustrated in Fig. 11.7 where the function of each subsystem is shown.

Heavy-ion irradiations were performed on FPGA and conducted at two facilities: 10 MeV/n cocktail beam [25] was used in vacuum at 88-inch Cyclotron facility of

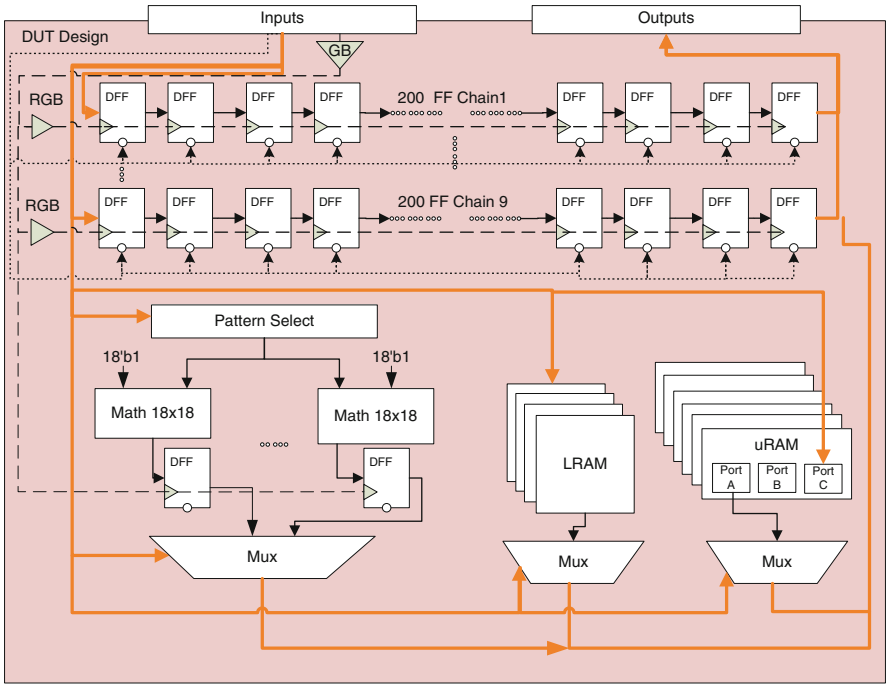


Fig. 11.5 Block diagram shows the FPGA design for radiation testing for SEU effects

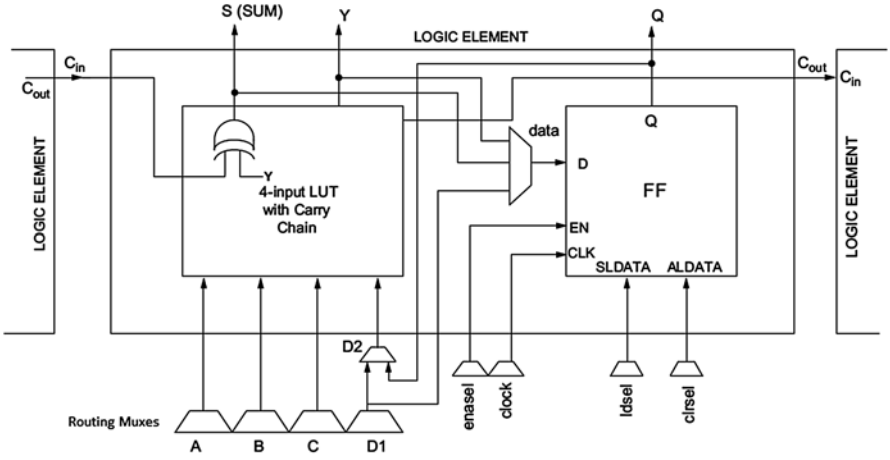


Fig. 11.6 Block diagram shows contents of Logic Element which includes FF, 4-input LUT and Routing-Mux connections [1]

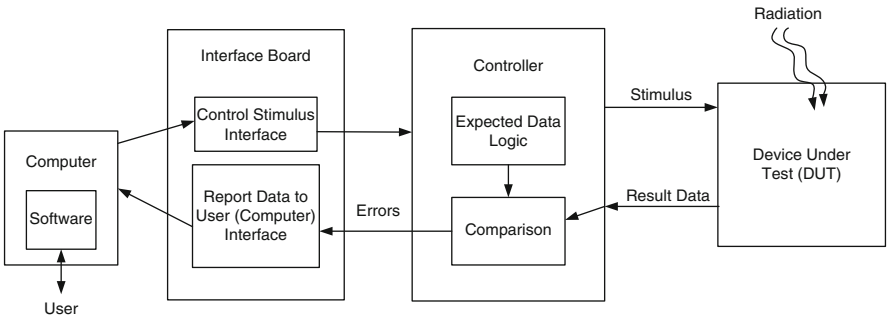


Fig. 11.7 Block diagram illustrates the SEE testing setup and data flow

Lawrence Berkeley National Laboratory in Berkeley, California; 15 MeV/n beam [26] was used in air at Cyclotron Center of Texas A&M University in College Station, Texas. For testing SEL effect, FPGA is biased to maximum operating voltages (nominal + 5 %) and tested at temperature up to 95 °C. Testing SEU effect is under nominal operating bias at ambient temperature.

11.4 Radiation Test Results on TID Effects

In this section the TID characteristics of the Flash cell and CMOS transistors used in FPGA will be presented, and followed by TID effects on two key FPGA electrical parameters which are propagation delay and standby power-supply current. It has been established that, in Flash-based FPGA [2–5], these two parameters determine the FPGA TID tolerance on the performance and power consumption. They degrade significantly before FPGA fails to function. These degradations owing to the ramification of Flash cell and transistor TID effects will also be discussed.

11.4.1 TID Effects on Flash Cells

In general TID effects on 65 nm Flash configuration cell are neutralizing the charge storage in the floating gate. As shown in Fig. 11.8, where both sense and switch device data are displayed, the Erase state V_t shifts to higher and Program state lower with TID. In principal these two states will finally neutralized by TID to a neutral state. However, the transistor effects will be strong at high dose level and render the V_t measurement impractical. The mechanisms responsible for these V_t shifts can be found in published literatures [2–5, 10].

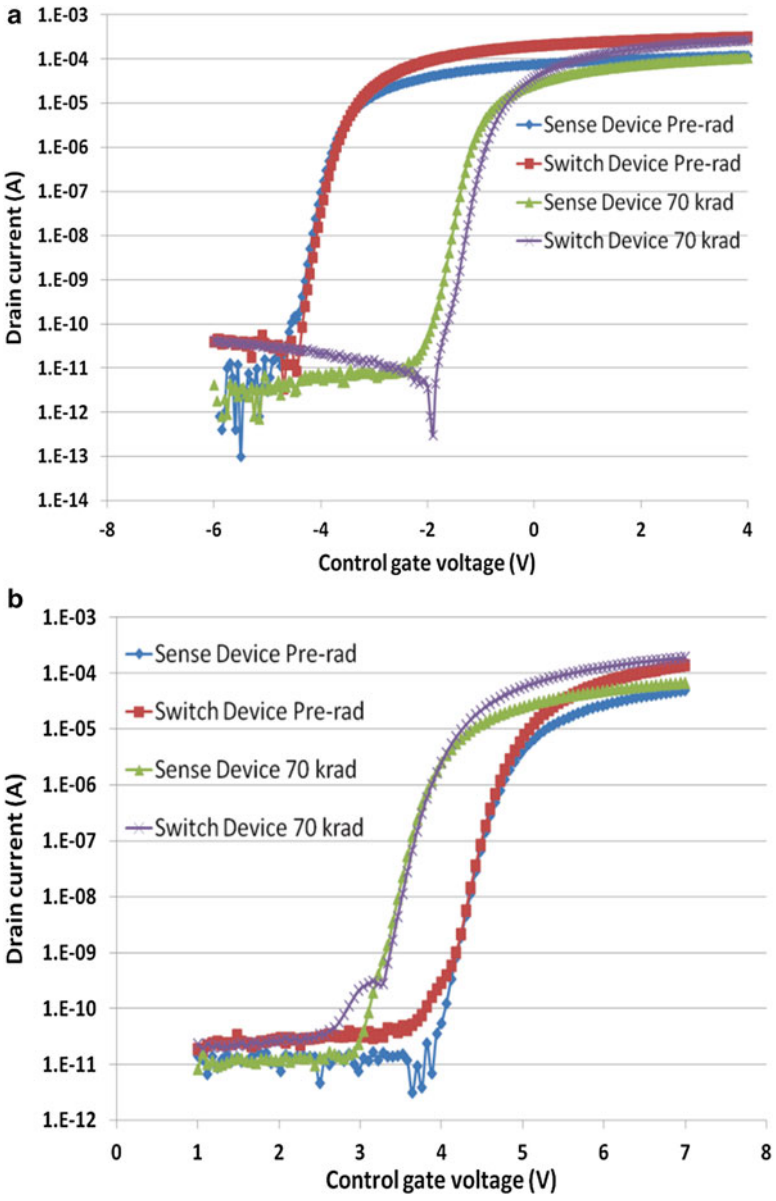


Fig. 11.8 Pre- and post-irradiation Id-Vg characteristics of Flash cell at (**a**) Erase state showing Vt increasing with TID, and (**b**) Program state Vt decreasing with TID

11.4.2 TID Effects on CMOS Transistors

In normal operation the Read action applies a bias to the control gate of switch devices. The circuit feeding the bias to the Flash cell contain thick oxide NMOSFET because they also pass high voltage (>15 V) during Write. If the driving ability of these high voltage devices is compromised, the bias on the switch will be degraded. Figure 11.9 plots Id-Vg curves of pre- and post-irradiated high-voltage device.

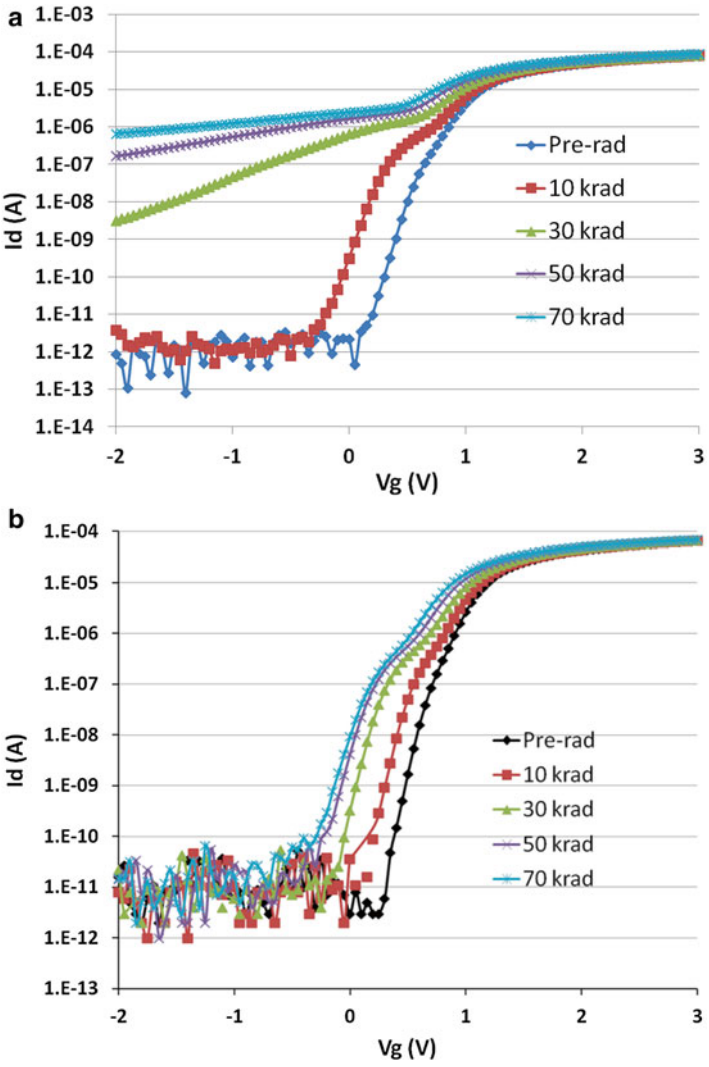


Fig. 11.9 Pre- and post-irradiation Id-Vg characteristics of high voltage NMOS device, with W/L = 10/0.68, and Vds = 0.1 V: (**a**) On-state irradiation bias Vg = VDD and Vd = Vs = Vb = GND; (**b**) Off-state irradiation bias Vd = VDD and Vg = Vs = Vb = GND

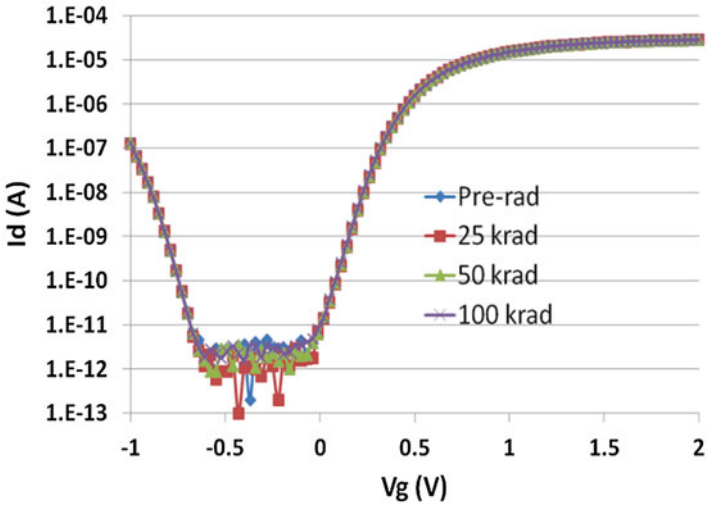


Fig. 11.10 Pre- and post-irradiation IdVg characteristics of low voltage NMOSFET, with W/L = 1/1 and Vds = 0.1 V, irradiated at on-state

Two bias conditions exist during normal operation, on-state and off-state. The on-state shows significant radiation-induced V_t shift and sub-threshold leakage current while off-state only V_t shift. Obviously the on-state has the worst case bias condition under irradiation.

For comparison, the radiation effects of low-voltage NMOS transistor used for logic functions are also tested and its irradiated I_d-V_g characteristics are illustrated in Fig. 11.10. Even irradiated under the worst case bias, there is no significant radiation effect on it.

In the following two sub-sections, these radiation effects at the transistor level will be used to elucidate the radiation effects on critical electrical parameters, propagation delay and standby power supply current, for FPGA function.

11.4.3 TID Effects on Propagation Delay

The in-flux propagation delay measured on the inverter-string is shown in Fig. 11.11. The propagation delay reaches 10 % degradation after 24–29 krad(SiO_2); it reaches 100 % after approximately 70 krad(SiO_2). All parts remain functional after 100 krad(SiO_2).

To relate TID effects on propagation delay to Flash cells, the signal path in Fig. 11.12 shall be examined. For an inverter string configured in the FPGA, the two $V_{control\_gate}$-biased switch devices in the signal path are at the Erase state to pass the signal. The TID effects on a switch at the Erase state as shown in Fig. 11.5a increase

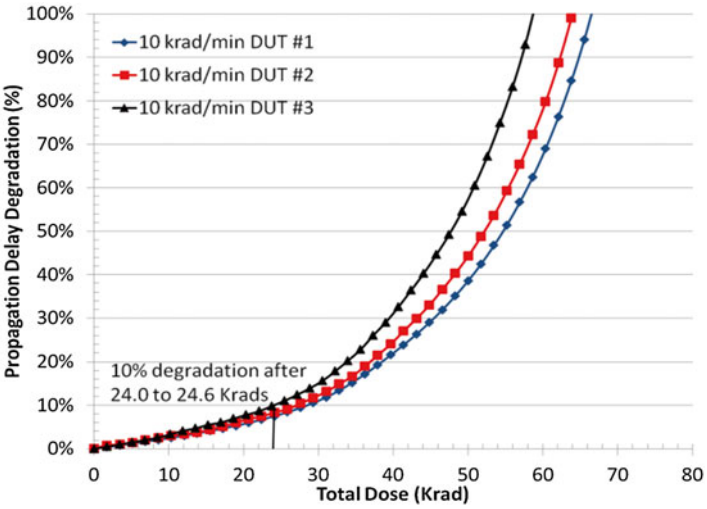


Fig. 11.11 Percentage of propagation-delay degradation versus TID at 10 krad(SiO_2)/min for three DUT

its V_t and decrease the driving strength. Consequently, the propagation delay in the inverter string increases.

High-voltage NMOS transistors also play an important role in the propagation delay degradation. Referring to Fig. 11.12 again, the first $V_{control\_bias}$ is provided by the V_{DD} input through the multiplier (MUX) with the V_{DD} input through M1 and GND through M0. Indeed, during operation, the level shifter (LS) is configured to connect V_{PP} to X_outb and GND to X_out to pass V_{DD} through the MUX, then M2, to bias the first Flash switch with $V_{control\_bias}$ on its gate. Note that during operation V_{DD} and V_{PP} are biased to 1.2 and 3.3 V respectively. Also, in this circuit, every NMOS transistor to be biased by V_{PP} has to sustain high voltage (>15 V) during programming. Therefore aforementioned thick oxide NMOS transistor has to be used on M0, M1 and M2.

As indicated in Fig. 11.12a, the radiation effects of M1 and M2, being biased at on-state, will increase their current drive and not degrade the propagation delay. On the other hand, Fig. 11.12b indicates that M0, being biased at off-state, will be turned on gradually by irradiation. Consequently, V_{DD} passing M1 will be comprised leading to the degradation of $V_{control\_bias}$ and subsequently degrades the propagation delay.

11.4.4 TID Effects on Standby Power-Supply Currents

The radiation effect on static power supply currents is dominated by the core power supply current I_{DDA}; hence it is the only component discussed here. Figure 11.13a plots I_{DDA} versus TID of the same three irradiated DUT as those mentioned in

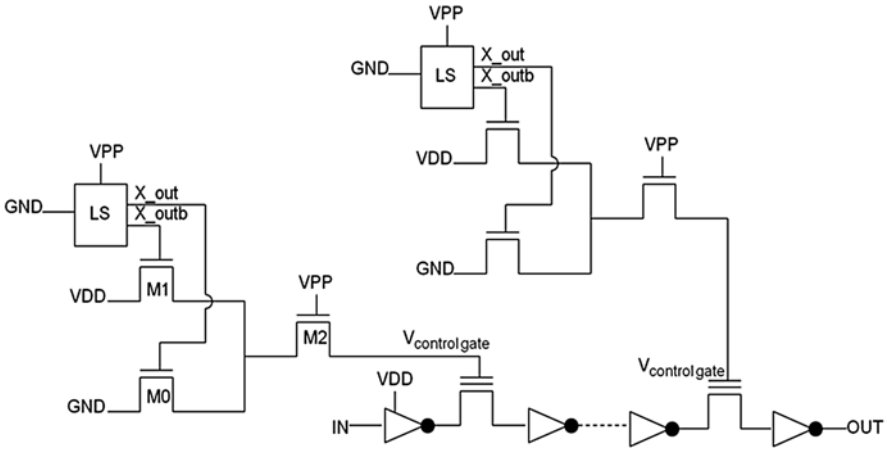


Fig. 11.12 Simplified schematics of DUT design for TID testing, VPP = 3.3 V and VDD = 1.2 V

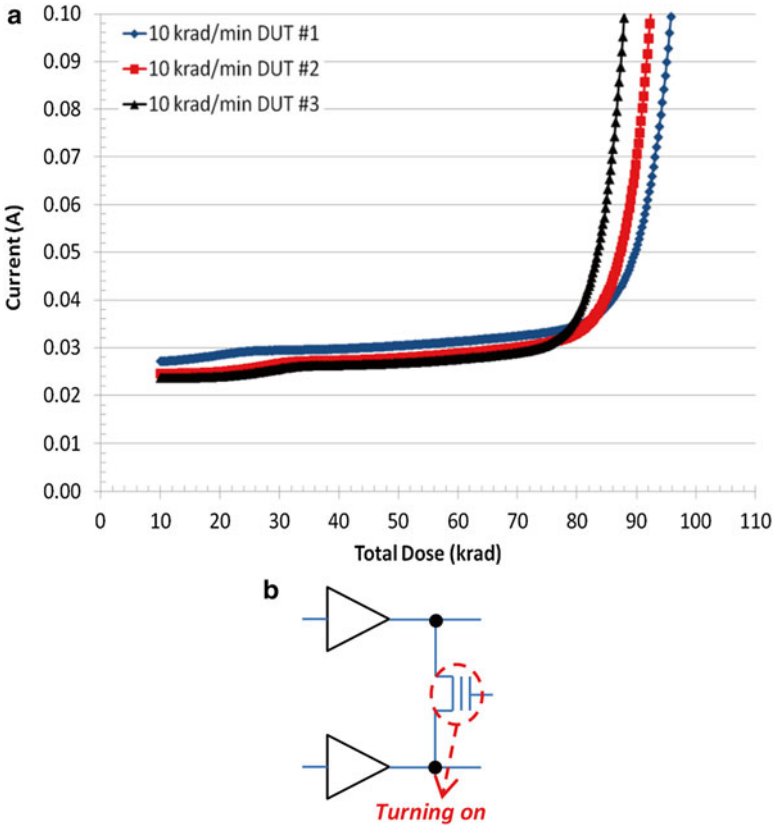


Fig. 11.13 (**a**) IDDA versus TID at 10 krad(SiO_2)/min for three DUT. (**b**) Simplified schematic shows root-cause of radiation induced IDDA due to turning on programmed Flash switch which originally isolates outputs of two drivers

previous sub-section. Initially the currents increase mildly but increase significantly suddenly when certain TID level is reached. All three DUT exhibit this threshold behavior and their threshold total doses are all very close to 80 krad(SiO_2).

This radiation-induced I_{DDA} can also be explained by the radiation effects on the Flash cell and high-voltage NMOS transistor. A simplified schematic in Fig. 11.13b aids to explain the Flash cell case. In an FPGA, a Flash cell is often connected as this schematic, e.g. in a routing multiplier (routing MUX) made of Flash cells (see Fig. 11.6), in which a Flash cell in Program-state isolates two drivers. Radiation degrades the isolation and consequently induced current flows from the driver output high to the driver output low. The case due to radiation effects of high-voltage NMOS has been exposed previously. In Fig. 11.12, the leakage due to M0 will connect V_{DD}, which is the power supply for I_{DDA}, through M1 and M0 to GND, and subsequently contributes to the increase of I_{DDA}. Although not quantitatively proven, it is believed that the Flash cell degradation caused driver contention is the dominant effect. For two reason, there are more cases of Fig. 11.13b than Fig. 11.12, and the threshold behavior in Fig. 11.13a fits better to the model of driver contention by Flash cell degrading to a certain level. Further analysis on this topic is beyond the scope of this paper and will be presented in the future.

11.5 Radiation Test Results on Single Event Effects

This section presents the results of first phase of SEE testing. The SEL of the FPGA is tested to prove its avionics worthy; SEU of the Flash cell is extracted from tests targeted for fabric SEU; fabric FF and SRAM SEUs are tested for static data pattern and dynamic patterns up to 10 MHz.

3D-TCAD simulation is also performed to calculate the cross-section of fabric FF and compared data with test results. The intention is using it to extend the boundary by which heavy-ion testing can reach.

11.5.1 FPGA SEL

The linear energy transfer (LET) threshold for SEL acquired by heavy-ion irradiated FPGA, biased at maximum operating V_{DD} and heated up to 95 °C, is above 15 MeV-cm^2/mg. Indicating the FPGA is immune to neutron-induced SEL and suitable for avionics. Five DUT were tested with total fluence on each DUT higher than 5×17 ions/cm^2.

11.5.2 Flash-Cell SEU

Here the SEU is defined as the single-event induced Flash-cell state flip: Program state flip to Erase state, or Erase to Program. However, from user point of view only flips causing functional failure will be detected. Therefore this is the Flash SEU measured. FPGA programed with SEU testing designs for fabric FF and SRAM were tested to not exceeding a TID limit, usually 10 krad(Si), to ensure performance and functionality. With this restrain, every FPGA been heavy-ion irradiated so far didn't fail functionality. Based on this result, the standard SEU cross-section versus LET plot of Flash cell at Erase state can be generated. Since the FPGA functionality depends on the critical Flash cell at Erase state passing the critical signal, the number of sampling bits has to be estimated from the programming architecture.

The aggregate Flash-cell SEU rates of numerous tested FPGA are plotted as cross section per flash cell versus LET. Figure 11.14 shows the data points. Each point represents an irradiation run. Since no functional failure has ever been observed, there is no SEU in the critical Flash cells. Therefore the cross section is the inverse of the total fluence on a critical Flash cell. In other words, the cross section for each irradiated LET is smaller than the lowest boundary data on this plot. For example, at the maximum tested LET of approximately 90 MeV-cm^2/mg the cross section is below 10^{-13} cm^2. For SEU of Flash cell at Program state, the correlation between state flip and FPGA functionality is not easily established. But nevertheless, non-existence of functional failure after more than 50 runs indicates very low SEU sensitivity for Program state too. In conclusion, these data practically show Flash cell immune to SEU.

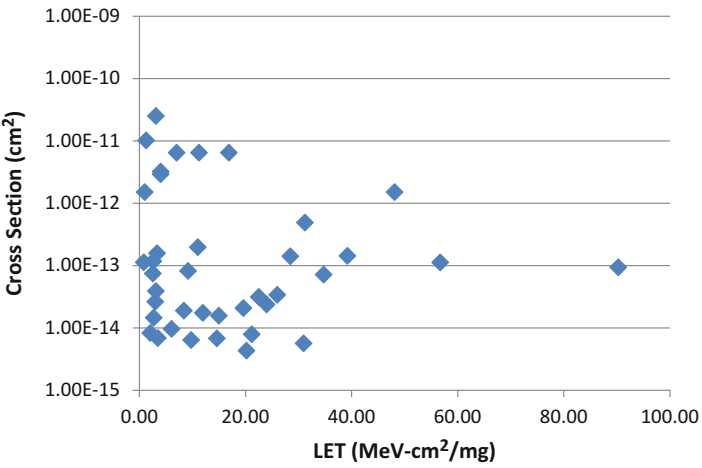


Fig. 11.14 Plot of heavy-ion test results, it displays critical Flash-cell SEU cross-section versus LET

11.5.3 Fabric-Embedded SRAM SEU

Figures 11.15 and 11.16 show the SEU cross-section versus LET plot and Weibull fitting curves of fabric µSRAM and LSRAM respectively. Both SRAM cells have LET_{TH} of 0.85 MeV-cm^2/mg. The saturated cross section for µSRAM and LSRAM are 4.5×10^{-9} cm^2/bit and 3.0×10^{-9} cm^2/bit respectively. Using the Weibull fitting parameters and Crème 96, the upset rate for µSRAM and LSRAM is calculated, for solar minimum and geosynchronous orbit with 100 mils aluminum shielding, to be 2.79×10^{-8} upset/bit/day and 4×10^{-8} upset/bit/day respectively.

11.5.4 Fabric Flip-Flop SEU

Figure 11.17 shows the measured SEU cross-section as a function of LET for the fabric FF, and the corresponding Weibull fitting curve with threshold LET (LET_{TH}) 0.74 MeV-cm^2/mg and saturated cross section 1.0×10^{-7} cm^2/bit. The signal passing through the shift registers are checkerboard data-pattern running at clock rates of 1, 3 and 10 MHz, and static data all-1 and all-0 patterns. Hundreds of errors are captured to gain a good confidence level in results. Each data point represents an average of three test results. There is no observable frequency dependence; hence data of different frequency are lumped together. Using the Weibull fitting parameters and

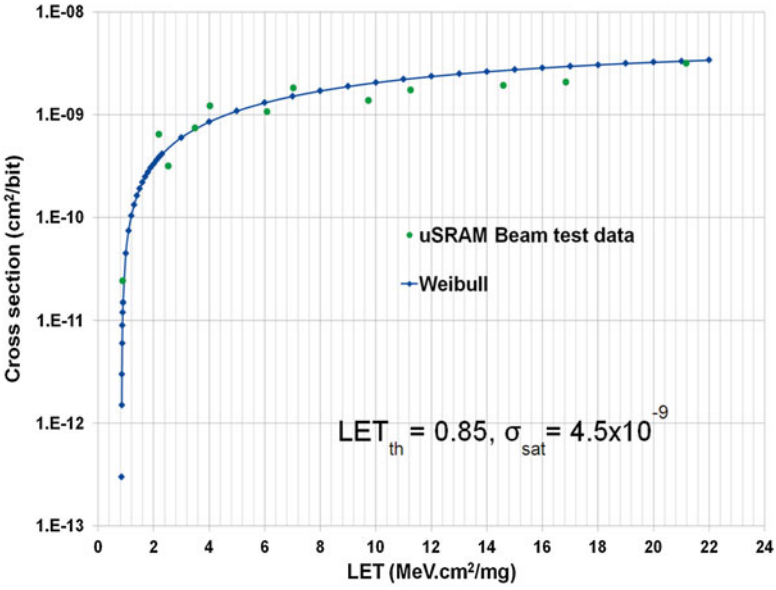


Fig. 11.15 Plot of heavy-ion test results, it displays SEU cross-section of µSRAM versus LET and Weibull fitting curve

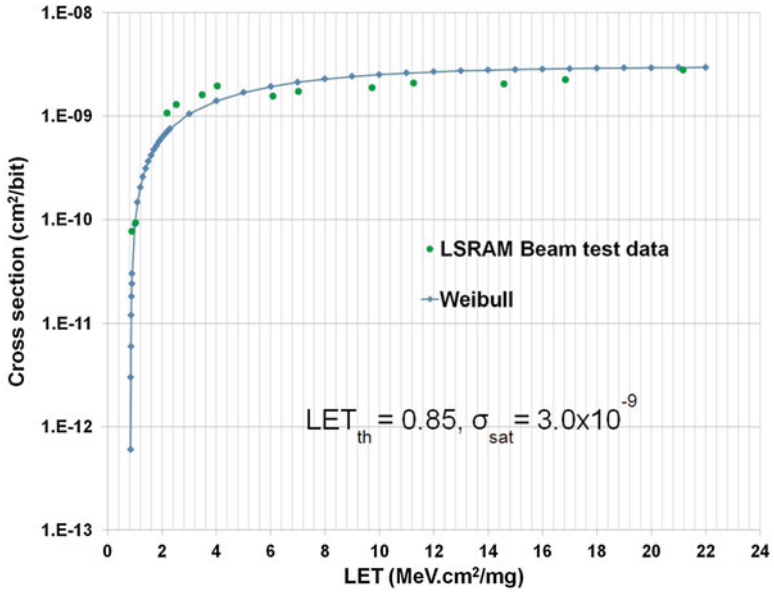


Fig. 11.16 Plot of heavy-ion test results, it displays SEU cross-section of LSRAM versus LET and Weibull fitting curve

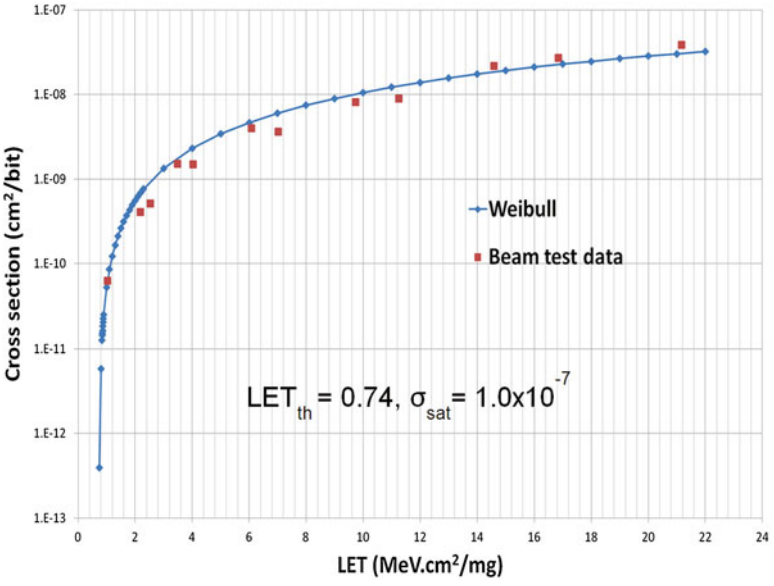


Fig. 11.17 Plot of heavy-ion test results, it displays SEU cross-section of fabric FF versus LET and the corresponding Weibull fitting curve

Crème 96, the upset rate is calculated, for solar minimum and geosynchronous orbit with 100 mils aluminum shielding, to be 1.76×10^{-7} upset/bit/day.

3D TCAD simulations using Robust Chip Inc (RCI) tools are performed to compare with test results. Figure 11.18a shows the 3D structure, which includes the

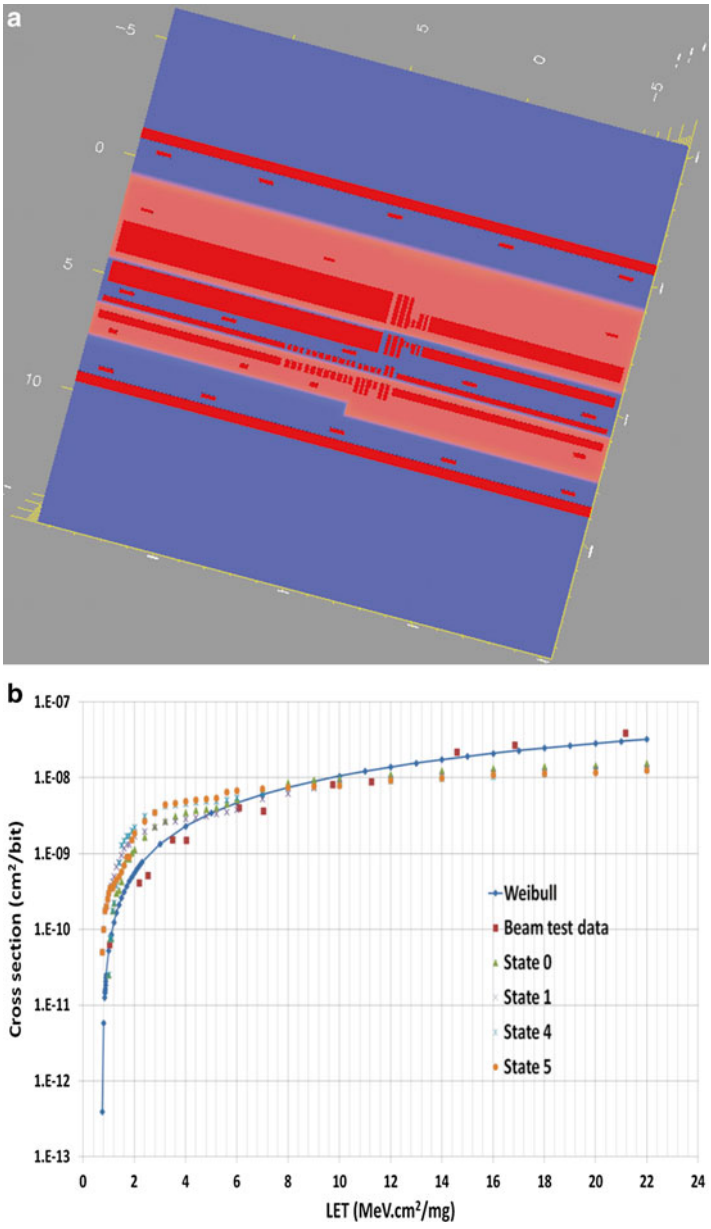


Fig. 11.18 (**a**) Simulated 3D TCAD structure including the FF cell and a simplification of the FF surrounding cells (represented by added "source-ties") existed in the real layout. (**b**) 3D-TCAD simulation results compare to heavy-ion test

fabric FF cell and the relevant neighboring cells, referred to as "source-ties". This FF is configured to a master-slave edge-triggered D-type FF for SEU testing. Its layout is more complex and with more transistors than a typical hardwired ASIC FF because it can be configured to many variants.

Extensive simulations follow to generate the cross-section as a function of LET. The results for different states (blanket 1 and 0 patterns) of the FF cell are plotted in Fig. 11.18b to compare with heavy-ion test results. The simulation results show a good agreement with test data, especially data near LET_{TH}.

11.6 Future Works

For TID effects, the physical mechanisms causing the propagation delay degradation and power supply current increase, the V_t shifts in the Flash cell and V_t shifts and sub-threshold leakages in the high-voltage NMOS transistors, will be modeled in the context of SPICE simulation. Then quantitative investigations can be performed to reach the finally goal of predicting TID tolerance.

For SEE effects, the next phase of heavy-ion testing will target IPs in which MSS, SerDes and high-speed DDR interface are especially interested by FPGA users. Also high-speed testing up to few 100 MHz on fabric logic and IO will be performed to complete the evaluation.

References

1. http://www.microsemi.com
2. Wang JJ, Samiee S, Chen HS, Huang CK, Cheung M, Borillo J, Sun SN, Cronquist B, McCollum J (2004) Total ionizing dose effects on flash-based field programmable gate array. IEEE Trans Nucl Sci 51(6):3759–3766
3. Wang JJ, Kuganesan G, Charest N, Cronquist B (2006) Biased-irradiation characteristics of the floating gate switch in FPGA. In: 2006 IEEE radiation effects data workshop, Vedra Beach, FL, pp 101–104
4. Wang JJ, Charest N, Kuganesan G, Huang CK, Yip M, Chen HS, Borillo J, Samiee S, Dhaoui F, Sun J, Rezgui S, McCollum J, Cronquiest B (2006) Investigating and modeling total ionizing dose and heavy ion effects in flash-based field programmable gate array. In: Proceedings of radiation effects on components and systems workshop, Athens, Greece
5. Rezgui S, Wilcox E, Lee P, Carts M, LaBel K, Nguyen V, Telecco N, McCollum J (2012) Investigation of low dose rate and bias conditions on the total dose tolerance of a CMOS flash-based FPGA. IEEE Trans Nucl Sci 59(1):134–143
6. Poivey C, Grandjean M, Guerre F (2011) Radiation characterization of microsemi ProASIC3 flash FPGA family. In: 2011 IEEE radiation effects data workshop, Las Vegas, NV, pp 92–96
7. Tarrillo J, Azambuja J, Kastensmidt F, Fonseca E, Galhardo R, Goncalez O (2011) Analyzing the effects of TID in an embedded system running in a flash-based FPGA. IEEE Trans Nucl Sci 58(6):2855–2862
8. Kastensmidt F, Fonseca E, Vaz R, Goncalez O, Chipana R, Wirth G (2011) TID in flash-based FPGA: power supply-current rise and logic function mapping effects in propagation-delay degradation. IEEE Trans Nucl Sci 58(4):1927–1934

9. Rezzak N, Wang JJ, Huang CK, Nguyen V, Bakker G (2014) Total ionizing dose characterization of 65 nm flash-based FPGA. To be published in 2014 NSREC radiation effects data workshop
10. Snyder E et al (1989) Radiation response of floating gate EEPROM memory cells. IEEE Trans Nucl Sci 36:2131–2139
11. Oldham T, McLean F (2003) Total ionizing dose effects in MOS oxides and devices. IEEE Trans Nucl Sci 50(3):483–498
12. Barnaby H (2006) Total-ionizing-dose effects in modern CMOS technologies. IEEE Trans Nucl Sci 53(6):3103–3121
13. Allen G, Swift G (2006) Single event effects test results for advanced field programmable gate arrays. In: 2006 IEEE radiation effects data workshop, Vedra Beach, FL, pp 115–120
14. Rezgui R, Wang JJ, Chan Tung E, Cronquist B, McCollum J (2007) New methodologies for SET characterization and mitigation in flash-based FPGAs. IEEE Trans Nucl Sci 54(6): 2512–2524
15. Rezgui R, Wang JJ, Sun Y, Cronquist B, McCollum J (2008) Configuration and routing effects on the SET propagation in flash-based FPGAs. IEEE Trans Nucl Sci 55(6):3328–3335
16. Battezzati N, Gerardin S, Manuzzato A, Paccagnella A, Rezgui S, Sterpone L, Violante M (2008) On the evaluation of radiation-induced transient faults in flash-based FPGAs. In: 14th IEEE international on-line testing symposium 2008, Washington, DC, pp 135–140
17. Battezzati N, Gerardin S, Manuzzato A, Merodio D, Paccagnella A, Poivey C, Sterpone L, Violante M (2009) Methodologies to study frequency-dependent single event effects sensitivity in flash-based FPGAs. IEEE Trans Nucl Sci 56(6):3534–3541
18. Sterpone L, Battezzati N, Ferlet-Cavrois V (2010) Analysis of SET propagation in flash-based FPGAs by means of electrical pulse injection. IEEE Trans Nucl Sci 57(4):1820–1826
19. Sterpone L, Battezzati N, Kastensmidt F, Chipana R (2011) An analytical model of propagation induced pulse broadening (PIPB) effects on single event transient in flash-based FPGAs. IEEE Trans Nucl Sci 58(5):2333–2340
20. Sterpone L, Du B (2014) Analysis and mitigation of single event effects on flash-based FPGAs. In: 19th IEEE European test symposium, Paderborn, pp 1–6
21. Rezzak N, Wang JJ, DSilva D, Huang CK, Varela S (2014) Single event effects characterization in 65 nm flash-based FPGA-SOC. To be published in 2014 Proceedings of SEE symposium
22. Marotta G, Naso G, Savarese G (2008) Memory circuit technologies. In: Brewer J, Gill M (eds) Nonvolatile memory technologies with emphasis on flash. IEEE, Piscataway
23. Gerardin S, Bagatin M, Paccagnella A, Grurmann K, Gliem F, Oldham T, Irom F, Nguyen D (2013) Radiation effects in flash memories. IEEE Trans Nucl Sci 6(3):1953–1969
24. SmartFusion2 Macro Library. http://www.microsemi.com
25. http://cyclotron.lbl.gov/base-rad-effects/heavy-ions/cocktails-and-ions
26. http://cyclotron.tamu.edu/ref/LET_vs_Range_15.pdf

Chapter 12
Using C-Slow Retiming in Safety Critical and Low Power Applications

Tobias Strauch

Abstract This paper shows the usage of C-Slow Retiming (CSR) in safety critical and low power applications. CSR executes C copies of a design by reusing the given logic resources in a time sliced fashion. It is already used in the 1960s, for example in the Barrel processors from the CDC 6000 series. Publications about this technique have been rare throughout the last decade. This paper shows that CSR offers great benefits when used in safety critical and low power applications.

12.1 Introduction

Safety critical applications use redundant designs and design state comparison techniques to detect potential design misbehavior. An example is a motor control circuit, where a malfunction could generate life threatening conditions. A full stop and restart of an application is sometimes costly and should be avoided with an on-the-fly recovery mechanism.

Another application for using redundant designs are the control systems of a satellite. Single event upsets (SEUs) must be detected before they could endanger costly missions in the orbit. It is beneficial when the power consumption of the redundant systems can also be reduced.

C-Slow Retiming (CSR) provides C copies of a given design by inserting registers and reusing the combinatorial logic in a time sliced fashion. The paper shows how CSR can be used when redundant designs are needed.

T. Strauch (✉)
R&D, EDAptix, Adelgundenstr. 5, Munich 80538, Germany
e-mail: tobias@edaptix.com

F. Kastensmidt, P. Rech (eds.), *FPGAs and Parallel Architectures for Aerospace Applications*, DOI 10.1007/978-3-319-14352-1_12

12.2 Background

The ever increasing demands for higher performance and higher throughput of cores have led to various techniques. [1] presents an efficient retiming algorithm and in [2] a retiming algorithm for FPGAs is shown. Retiming for wire pipelined SoC buses is discussed in [3]. Automatic pipelining of designs is outlined in [4]. The pipelining of sequential circuits with wave steering is shown in [5]. Leiserson et al. introduces the concept of C-Slow Retiming (CSR) in [6]. [7] presents a formulation as a general model for retiming and recycling, which also accepts an interpretation of the CSR problem. The effects of CSR is presented on three different benchmarks in [8] and the post-placements CRS for a microprocessor on an FPGA is shown in [9], whereas [10] presents an abstraction algorithm for the verification of generalized C-slow designs. In recent publications, CSR is used to maximize the throughout-area efficiency in [11] and on simultaneous multithreading processors in [12].

12.3 Contribution and Paper Organization

To the best of the author's knowledge, power consumption (P) has not been considered in publications about C-Slow Retiming (CSR). Results are given for the P of CSR-ed based designs on an FPGA.

The paper demonstrates how to use CSR for single event upset (SEU) detection and design state on-the-fly recovery. The method is then further developed and optimized to reduce area (FPGA utilization) and the P of the application. Results of two 32-bit processors on a low-cost FPGA are given.

Section 12.3 outlines the CSR technology. In Sect. 12.5 the power consumption of CSR-ed designs is discussed. A method to detect single event upsets is shown in Sect. 12.6 and how the CSR algorithm can be adapted for that. The paper finishes with results and a summary in the Sects. 12.7 and 12.8.

12.4 C-Slow Retiming

12.4.1 Theory of CSR

Figure 12.1a shows the basic structure of a sequential circuit with its combinatorial logic (CL), in- and outputs and original registers (OR). In Fig. 12.1b, the CL is sliced into three (C = 3) parts, and each original path has now two (C-1) additional registers. This is the basic idea behind CSR. It results in C functionally independent design copies which use the logic in a time sliced fashion. It shows how different parts of the logic are used during different cycles. It now takes three micro-cycles to achieve the same result as in one original cycle. In Fig. 12.1, inputs and outputs are

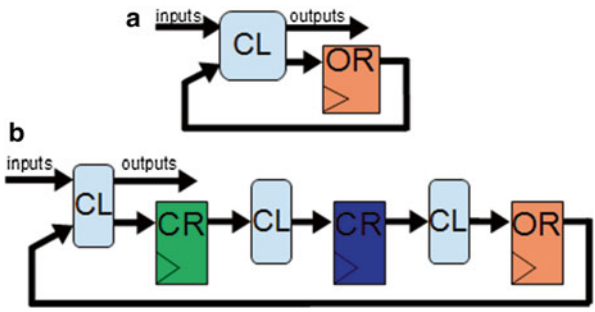


Fig. 12.1 (**a**) Simplified design. (**b**) Applying CSR technique

valid at the same time slice. The implemented register sets are called "C-Slow Retiming Registers", CRs. They are placed at different C-levels. Figure 12.1b shows one basic rule of CSR. There are only paths between consecutive CRs and also from the last CRs to the original register set and from the original register set to the first CRs.

We define the maximum frequency of the given design (Fig. 12.1a) as Fd and the maximum frequency of a CSR-ed design (Fig. 12.1b) as Fcsr, whereas:

$$Fcsr \sim Fd * C \tag{12.1}$$

The individual cycle of a CSR-ed design is called a micro cycle. By generating C independent copies of the design, all running—theoretically—at Fd, we can also say that the system frequency Fsys is equal to Fcsr:

$$Fsys = Fcsr \sim Fd * C \tag{12.2}$$

In theory, this could lead to an unlimited performance increase. Evidently this cannot be done endlessly and register insertion becomes inefficient for higher C again. The results section at the end of this paper shows examples for that.

12.4.2 CSR on RTL

CSR clearly changes the behavior of the design and can only be fully utilized when the CSR-ed core is embedded in a new logic environment. With the right wrapper logic, the CSR-ed design then behaves exactly as the original core, but multiple and functional independent versions are available. These modifications have a dramatic impact on the design flow. It is of great advantage to have a solution on higher level such as RTL. The CSR-ed version must be used as a new core in the design and verification process. A technique has been demonstrated, which automatically modifies the design to enable CSR on RTL by the author in [13]. The results given in this paper are generated using this technology.

12.4.3 Verification of CSR Design Modifications

It is non-trivial to verify the correctness of CSR based designs. Static timing analysis (STA) can be used for that. When each C-level gets its own clock tree, only paths from one C-level to the next one exist. Additional paths exist from the last C-level to the original registers and from the original registers to the first C-level. It can be checked during a standalone design level synthesis and STA run, if additional paths exist, which should not exist. The static analysis verifies the correctness of the register insertion process. The individual clocks can then be connected to a single clock again.

12.5 Power Consumption of CSR-ed Designs

12.5.1 Overview

Two (out of many) sources for power consumption (P) in digital designs are clock tree activity and switching activity generated by combinatorial logic. When a design is instantiated N-times, the number of resulting registers is N-times higher, but the clock speed remains the same. When using CSR, the number of resulting registers is less or roughly C-times higher. The difference is, that the clock speed must also be C-times higher to achieve the same performance. This results in a higher P of the clock tree in CSR-ed designs than the one of the alternative approach to instantiate individual designs. This is also shown in the result section on two different processors.

It has been demonstrated that register insertion can reduce the P of a design. For example Lim et al. use flip-flops with shifted-phase clocks in [14]. This looks similar to the CSR approach, although the register placement in the used CSR algorithm is timing driven. The used CSR algorithm places registers at the end of each path and then moves individual registers throughout the combinatorial logic until the best timing is achieved (timing optimization process).

In Fig. 12.2 the "CSR 4 P" line shows the relative P of one thread compared to the P when running the thread on the original core ("1-line"). It starts with 71 % P overhead at the beginning of the timing optimization process. This is due to the facts, that the signals generate toggling activity when passing through the additional registers and that the higher register load (and clock frequency) generate a higher clock tree P. The P overhead drops from 71 to 45 % during the timing optimization process when a better register distribution throughout the logic—mainly on the datapath—is reached. It can be argued, that this P reduction comes from the fact that the number of longer logic paths is reduced and therefore the probability to generate power consuming signal glitches is reduced. It was not successful to combine this timing driven approach with power aware optimization techniques (as in [14]).

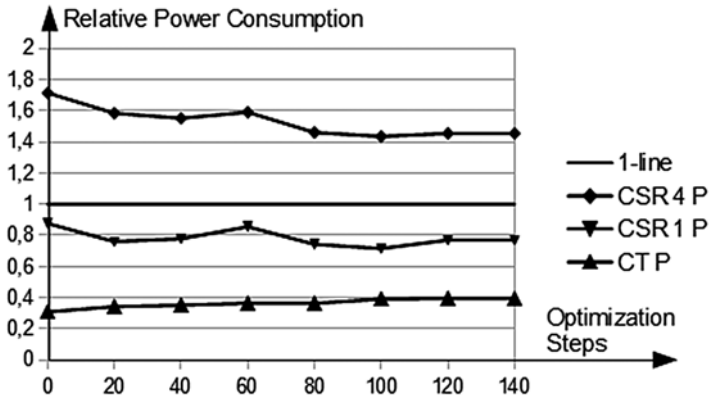


Fig. 12.2 Relative P of a CSR-ed design (C = 4) during timing optimization process

In Fig. 12.2 the relative P of the clock tree compared to the P of the original thread during the timing optimization process ("CT P") is shown. The relative P of the clock tree increases due to the rising number of registers when improving the timing of an CSR-ed design. The line "CSR 1 P" shows the P of a single thread when only identical threads are executed. This will be discussed in the next section.

When CSR is used on an ASIC, it can be argued, that the smaller CSR-ed design consumes less Iddq compared to the larger design of the alternative approach to instantiate individual designs.

12.5.2 Using Both Clock Edges in CSR

For completeness we will show the results of a special CSR approach with inverted clock edges for every other C-level. This approach makes only sense when an even number of design slices exists (C = 2, 4, …). The number of resulting design copies will be half of the design slices C/2. The P numbers are presented in the result section.

12.5.3 P When Running Identical Threads

In Fig. 12.2 we see how the P changes when applying the CSR algorithm (C = 4) on a given example design and identical threads are executed ("CSR 1 P" line). In this case the P of a single thread is in the range of 87–77 % of the P of the same thread executed on the original design. It can be assumed that the clock tree P increase (due to the higher clock speed) is less than the P reduction that comes with the register insertion into the datapath.

In the sequel of this paper we elaborate on CSR-ed designs with an identical input stimuli for each design copy. We use processors to demonstrate the effectiveness of the proposed method, but the method is not limited to processors only. Nevertheless, we use the word “thread” synonym for the execution of a processor program or execution of an algorithm on a digital design.

12.6 Detecting a Single Event Upset (SEU) Using CSR

12.6.1 Detecting an SEU with Standard CSR

One way to detect a single event upset (SEU) is the duplication of a design (redundancy) and to compare key registers and/or outputs. When an SEU occurs, at least one design runs different and further actions can be taken. CSR supports this feature when executing (a group of) identical threads. In Fig. 12.3, all threads execute the same algorithm (or program) and use the logic in a time shared fashion. Therefore only a limited number of threads are affected when an SEU occurs. Multiple identical threads are most likely affected differently because each one of them is in a different design state. When this difference affects the state of key registers, it can be detected by a certain support logic.

We applied this technique on two different processors. We added SEU detection logic to the design and run identical threads on each processor. In both cases we used the program counter and the data-bus access registers to detect different thread behaviors. We were not able to detect an SEU when running the application on an FPGA, but we used error injection techniques in simulation (as discussed by Braza et al. in [15]) to verify the behavior.

Based on empirical data we can assume that design duplication techniques using CSR generate less registers and certainly need less combinatorial logic than the alternative techniques using individual design instantiations. It can be argued, that this reduced register and logic count (compared to multiple individual instantiations of the design) also reduces the possibility to generate an SEU.

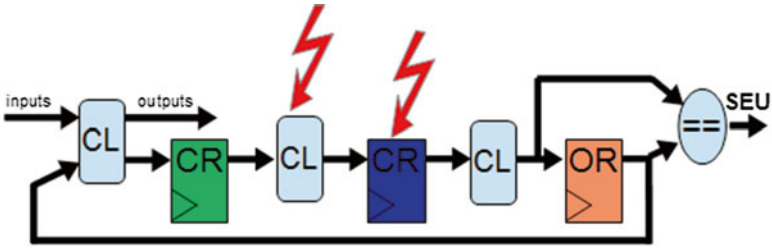


Fig. 12.3 Comparing signal values at key registers to detect an SEU

12.6.2 Recovery

When an SEU is detected, safety critical designs can restart or execute predefined software recovery routines. When using CSR, an on-the-fly recovery is possible. In Fig. 12.4 we see the CSR-ed design enhanced by the SEU detection circuit. When $C \geq 3$, the SEU detection circuit uses a majority decoder to detect the failing thread by comparing the key register values of C identical threads. This is done every C micro-cycles.

A modified write enable sequence then overwrites the specific Rn register associated with the failing thread. For the on-the-fly recovery mechanism all other OR which are not used for SEU detection (Fig. 12.4) must be enhanced by signal hold (not enable) mechanism to overwrite the failing thread.

The technique has been successfully simulated on RTL using a simple 1-out-of-3 majority decoder and an error injection mechanism. The result was a full design on-the-fly recovery. The area overhead of this approach is reported in the result section. This on-the-fly recovery mechanism is almost impossible to achieve when using the standard SEU detection concept of individual redundant design implementations.

12.6.3 Reducing Shift Register Count

Figure 12.5a shows a design after applying the CSR method. It can be seen that CSR generates a high number of shift registers by adding registers to the feedback loop of the original registers. Additional shift registers are generated on the paths through the combinatorial logic. These shift registers contribute to the majority of area and P increase.

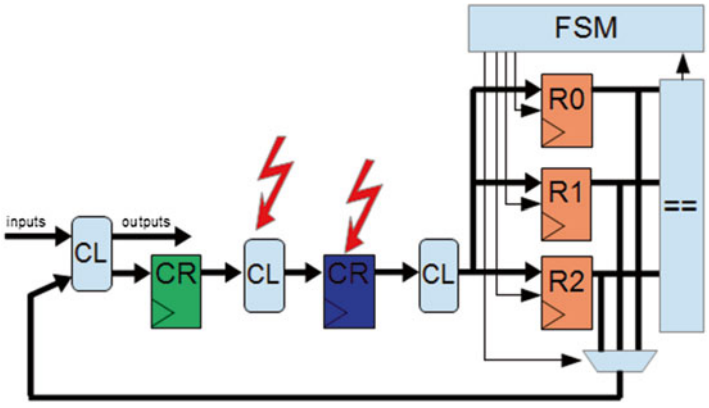


Fig. 12.4 On-the-fly recovery

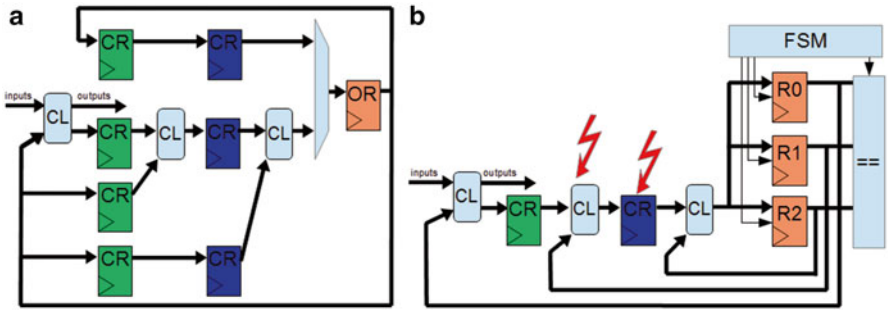


Fig. 12.5 (**a**) Shift Registers generated by Register Feedback Loops and adjacent C-Slow Retiming Registers (CRs). (**b**) CSR-ed design with SEU detection circuit and reduced set of C-Slow-Retiming Registers (CRs)

When identical threads are executed, the number of shift registers can be reduced by using a modified CSR algorithm. In this case the original registers are replaced by a slightly enhanced logic, which is shown in Fig. 12.5b. Each original register is now instantiated C times, they are called "Rn" (R1, R2, R3 in the example). A FSM (same for all registers in the design) generates individual capture enable signals, so that the Rns take over the incoming bit stream at different consecutive cycles. Also the outputs of the Rns drive the combinatorial logic at different C-levels, so that shift registers generated by consecutive CRs can be removed by connecting the combinatorial logic with the relevant Rn. This has a positive impact on the overall register count (area) and P of CSR-ed designs running identical threads. Empirical data on two different processors is shown in the result section.

An additional comparator logic (see Fig. 12.5b) continuously compares 2 register values of the Rn registers. In case of a mismatch, the logic indicates that 2 threads run differently. This logic can be used for all original registers or only for certain key registers (like the program counter for instance).

At a certain timeslot (every C microcycles), all threads can be compared at the same time. When $C \geq 3$, an on-the-fly recovery feature can be implemented by using a majority decoder and a slightly enhanced FSM logic. The FSM then uses the write enable signal to overwrite the Rns associated with a failing thread with the correct value.

This proposed method generates a system of redundant designs with SEU detection feature, a reduced area and a reduced P compared to a system using standard CSR or compared to a system using the alternative approach of instantiating individual designs. For more details see the result section. This method is particular useful when implemented on area sensitive ASICs used in safety critical and low power applications.

12.7 Results

The numbers in this results section are based on two CPUs. The RTL code for the ARM3 core ("Amber", 32-bit RISC processor, 3-stage pipeline, ARM v2a) and the OR1200 ("OR1000", 32-bit scalar RISC, 5-stage pipeline, MMU, DSP capabilities, TLB, instruction and data cache) can be found at [16]. The designs are implemented on a Xilinx Spartan-6 LX16 (-2ftg256). The clock is generated externally. The algorithm for CSR used in this paper is described in [13].

The P of a design during the CSR timing optimization is shown in Fig. 12.2. Two scenarios are tested. In a standard scenario, different threads are executed. In an alternative scenario, all threads execute the same program synchronously so that no combinatorial logic switches between the individual threads.

The P can be considered as relatively constant (Fig. 12.2) when moving the registers throughout the combinatorial logic. This was not expected. It was assumed, that the pipelined logic reduces the P by reducing the number of net glitches as shown in [14]. It can therefore be assumed, that the placement of additional registers (CRs) to reduce P needs to be carefully chosen. The author was not successful to combine this work with the technique demonstrated in [14].

Table 12.1 shows the results of a CSR-ed ARM3 core. When multiplying the functionality by $C=2\ldots5$, the number of registers increases up to 330 %. At the same time, the number of occupied slices remains relatively stable. This indicates, that the additional registers nicely fit into the already used slices. In other words, you have five times the functionality with just an area overhead of 43 % when using CSR.

The performance increases with each C step. Although it does not reach the performance (200, 300, …, 500 %) of the alternative concept by implementing individual processors (called "A" in the sequel of this section), it has a reasonably timing of 6.234 ns. This is a performance increase of up to 293 % compared to a single core implementation ("rel to 1"), but it only reaches 59 % ("rel to A") of the performance of A. Better results can be achieved with more advanced technologies like the Virtex family, as can be seen in [13], and most likely in ASIC technologies.

When a single core with 825 occupied slices can run at 18.250 ns, we can calculate the performance per area (PpA) factor to 66.42 kHz/slice (Table 12.1). We can see in the PpA column, that this factor increases by up to 205 % for $C=5$. In other words, when CSR can be used, more performance can be realized on a given size. Nevertheless, increasing C becomes less efficient for higher C.

Table 12.1 Results for CSR-ed ARM3 core, part I

C	Registers		Occupied slices		Performance [ps]			PpA
1	670	rel to 1	825	rel to 1	18,250	rel to 1	rel to A	66.42
2	1,683	251 %	1,015	123 %	11,850	154 %	77 %	125 %
3	1,768	264 %	1,018	123 %	8,917	205 %	68 %	166 %
4	2,091	312 %	1,029	125 %	7,210	253 %	63 %	203 %
5	2,211	330 %	1,177	143 %	6,234	293 %	59 %	205 %

The P of the original ARM3 core is 22.1 mW, running at maximum speed (18.250 ns). When instantiating individual ARM3 processors, the P multiplies accordingly (see Table 12.2, P column). We distinguish between running the same program on all available designs or running different programs.

When running the same program at the maximum possible speed, the P decreases to 40 % compared to A. This is certainly due to the fact, that the maximum possible speed is less than the one of A.

Even when the CSR-ed core could be run at the theoretical possible speed (cycle time = 18.250 ns/C), the P would only be in the range of 68–77 % of the A. The P seams to be relatively constant and independent of C when running the same program. We have already seen in Fig. 12.2, that the P is relatively independent of the CSR timing optimization process when moving registers throughout the combinatorial logic.

These findings show that CSR is great for safety critical applications (see Sect. 12.6). By running the same program on C copies of a CSR-ed design/CPU, you can decrease the area and power consumption at the same time, compared to A. By removing the obsolete registers on the register feedback paths, the increase of the occupied slices is only 13 % for C = 5 (See SEU column).

The P changes relatively to the P of the A from 113 to 85 % when increasing C and running different programs. When running the design at the theoretical possible speed (18.250 ns/C), the P is around 147 % of the P of A. It turned out that this number is relatively constant for different Cs. A CSR-ed design uses less registers than A, but can run (theoretically) C times faster, which results in a higher P of the clock tree than the one of A.

Similar numbers can be found for the CSR-ed implementation of the OR1200 core. The relative number of registers increases to up to 329 % (Table 12.3) whereas the number of occupied slices only reaches 137 % for C = 5. The performance increase is less optimal over an increasing number of copies. This is due to the already fast cycle time of the original core and the relatively slow technology (Spartan 6). Better results can be reached on a more advanced technology (Virtex 5), as reported in [13]. The P of the original core is 42.4 mW (Table 12.4). The P when running the same or different programs and with increasing numbers of copies is listed as well. When running the same thread and removing obsolete shift registers, the area increase is only 11 %.

Table 12.5 shows the P of the CSR-ed demo processors (C = 4) using inverted clock edges on consecutive C-levels. With C = 4 it is only possible to run two

Table 12.2 Results for the CSR-ed ARM3 core, part II

C	PD [mW]	P same [mW]			SEU	P diff [mW]		
1	22.1	22.1	@ perf.	@ max	rel to 1	n.a.	@ perf.	@ max
2	44.2	22.79	52 %	67 %	109 %	50.05	113 %	147 %
3	66.3	35.00	53 %	77 %	109 %	66.72	101 %	148 %
4	88.4	41.01	46 %	73 %	112 %	81.50	92 %	146 %
5	110.5	43.91	40 %	68 %	113 %	94.27	85 %	146 %

Table 12.3 Results for the CSR-ed OR1200 core, part I

C	Registers		Occupied slices		Performance [ps]			PpA
1	1,280	rel to 1	994	rel to 1	14,008	rel to 1	rel to A	71.82
2	2,741	214 %	1,254	126 %	9,080	154 %	100 %	122 %
3	3,573	279 %	1,335	134 %	7,127	197 %	85 %	146 %
4	3,901	305 %	1,316	132 %	6,334	221 %	72 %	167 %
5	4,210	329 %	1,361	137 %	5,973	235 %	61 %	171 %

Table 12.4 Results when using SEU detection

	SEU detection		SEU det. + recov.	
C=3	oc. slices	rel to 1 (%)	oc. slices	rel to 1 (%)
ARM3	1,073	130	1,087	132
OR1200	1,361	137	1,373	138

Table 12.5 Results when using different clock edges

	P 2 identical threads [mW]			P 2 different threads [mW]		
C=4		@ perf. (%)	@ max (%)		@ perf. (%)	@ max (%)
ARM3	24.38	55	73	55.5	126	159
OR1200	70.09	83	121	96.56	114	162

Table 12.6 Results for the CSR-ed OR1200 core, Part II

C	PD [mW]	P same [mW]			SEU	P diff [mW]		
1	42.4	42.4	@ perf.	@ max	rel to 1	n.a.	@ perf.	@ max
2	84.8	43.55	51 %	67 %	108 %	126.50	149 %	193 %
3	127.2	54.8	43 %	65 %	110 %	163.14	128 %	196 %
4	169.6	64.18	38 %	68 %	110 %	174.71	103 %	186 %
5	212	69.61	33 %	70 %	111 %	197.00	93 %	198 %

identical threads or two different threads. When the relevant numbers of Table 12.5 are compared with the one of Tables 12.2 and 12.6, we see that this method generates more P on the ARM3, but less on the OR1200.

Table 12.4 shows the area overhead of the CSR-ed demo processors (C=3) when an SEU detection or an SEU detection and recovery mechanism is used. CSR offers in these cases the possibility to get an ARM3 (OR1200) implementation with SEU detection logic with just 30 % (37 %) overhead compared to the original implementation. A comparison logic for all registers is used. If only key registers are compared, the area overhead is reasonable lower. The additional area overhead when an on-the-fly recovery mechanism is added is minor (1–2 %). This is due to the fact that the write enable signal nicely fits on the used FPGA technology.

Table 12.7 Results when using Register Reduced CSR Version

	Red. SEU + recov.		P 3 identical threads [mW]		
C = 3	oc. slices	rel to 1 (%)		@ perf. (%)	@ max (%)
ARM3	1,045	127	30.2	46	71
OR1200	1,278	129	49.7	39	63

Table 12.7 shows the area overhead of the CSR-ed demo processors (C = 3) when an SEU and recovery mechanism with reduced register count is used. The are overhead could be reduced to only 27 % (ARM3) or 29 % (OR1200) of the single core implementation. Also the P when running three identical threads is reduced to just 46 % when running the possible speed on the ARM3, and just 71 % of the P when running the core at the theoretical maximal speed.

12.8 Summary

C-Slow Retiming is known for running C copies of a design to increase the system performance per area for a given design. This paper elaborates on running identical threads and on using the resulting redundancy for SEU detection and design state recovery. In order to further reduce the area and power consumption various methods are discussed.

In general it can be said, that an individual thread runs always slower on an CSR-ed design compared to its execution on the original design. The multithreaded CSR solution needs less area but consumes roughly 40 % more power than the alternative approach to instantiate individual designs. Whereas when identical threads are executed, the power consumption is in favor of CSR because a thread consumes 20 % less power on a CSR-ed design than on the original core implementation. This fact as well as the possibility to use a design state recovery mechanism makes CSR ideal for safety critical and low power designs.

Although this paper concentrates on running CSR-ed designs on FPGAs, it looks promising to use this method also on ASICs and design implementations, where SEU detection, design state recovery, power consumption and design area play an important role.

References

1. Lin C, Zhou H (2006) An efficient retiming algorithm under setup and hold constraints. In: ACM/IEEE design automation conference (DAC) 2006, San Francisco, pp 945–950
2. Singh D, Brown S (2002) Integrated retiming and placement for field programmable gate arrays. In: FPGA 2002, Monterey, CA, 24–26 Feb 2002, pp 67–76
3. Lin C, Zhou H (2003) Retiming for wire pipelining in system-on-chip. In: ICCAD '03, San Jose, 11–13 Nov 2003, p 215

4. Kroening D, Paul W (2001) Automated pipeline design. In: Proceedings of 38th ACM/IEEE design automation conference (DAC 2001), Las Vegas, 18–22 June 2001, pp 810–815
5. Macchiarulo L, Shu S, Marek-Sadowska M (2004) Pipelining sequential circuits with wave steering. IEEE Trans Comput 53(9):1205–1210
6. Leiserson C, Saxe J (1991) Retiming synchronous circuitry. Algorithmica 6(1):5–35
7. Bufistov D, Cortadella J, Kishinevsky M, Sapatnekar S (2007) A general model for performance optimization of sequential systems. In: IEEE international conference on CAD, San Jose, 4–8 Nov 2007, pp 362–369
8. Weaver N, Wawrzynek J (2002) The effects of datapath placement and C-slow retiming on three computational benchmarks. In: Proceedings of FCCM 2002, Napa, 24 April 2002, pp 303–304
9. Weaver N, Markovskiy Y, Patel Y, Wawrzynek J (2003) Post-placement C-slow retiming for the Xilinx Virtex FPGA. In: FPGA 2003, Monterey, 23–25 Feb 2003
10. Baumgartner J, Tripp A, Aziz A, Singhal V, Anderson F (2000) An abstraction algorithm for the verification of generalized C-slow designs. In: CAV 2000, LNCS 1855. Springer, Heidelberg, pp 5–19
11. Su M, Zhou L, Shi C (2007) Maximizing the throughput-are efficiency of fully-parallel low-density parity-check decoding with C-slow retiming and asynchronous deep pipelining. In: ICCD 2007, Lake Tahoe, 7–10 Oct 2007, pp 636–643
12. Afram M, Khan A, Sarfaraz M (2001) C-slow technique vs multiprocessor in designing low area customized set processor for embedded applications. Int J Comput Appl 6(7)
13. Strauch T (2013) Timing driven c-slow retiming on RTL for multicores on FPGAs. In: ParaFPGA 2013, Munich, 10–13 Sept 2013. Available: www.edaptix.com/ParCo2013_Strauch_CSR_RTL.pdf
14. Lim H, Lee K, Cho Y, Chang N (2005) Flip-flop insertion with shifted-phase clocks for FPGA power reduction. In: ICCAD 2005, San Jose, 6–10 Nov 2005, pp 335–342
15. Braza J, Gracia J, Blanc S, Gil D, Gil P (2008) Enhancement of fault injection techniques based on modification of VHDL code. IEEE Trans Very Large Scale Integr Syst 16(6): 693–706
16. OpenCores, Stockholm, Sweden (2007) www.opencores.org/project

Chapter 13
Improving the Implementation of EDAC Functions in Radiation-Hardened FPGAs

Carlos Colodro-Conde and Rafael Toledo-Moreo

Abstract Error Detection and Correction (EDAC) codes have been widely used for protecting memories from single event upsets (SEU), which occur in environments with high levels of radiation or in deep submicron manufacturing technologies. This paper presents a novel synthesis algorithm that provides area-efficient implementations of EDAC functions on FPGAs, where resource utilization usually needs to be kept to a minimum in order to embrace more logic in a single die. The algorithm under consideration has been tested selecting two models of radiation-hardened FPGAs: one from the RT ProASIC3 series (flash-based) and another one from the RTAX-S series (antifuse-based). The results show that, when compared to the commercial synthesis tool provided by the vendor of the selected FPGA models, the proposed algorithm reduces number of used combinational cells up to a 23.5 %, while providing generally better timing performances (up to 23.6 % faster maximum path delays for the post-place and route implementations).

13.1 Introduction

With the continuous downscaling of the VLSI fabrication technologies, radiation induced errors have become a major concern in modern digital electronics. Even at ground level, high-energy particles like neutrons coming from the cosmic background create undesired current pulses that may invert the value stored in a memory element such as a flip-flop [1, 2]. This kind of errors, called single event upsets (SEU), compromise the reliability of the systems if no action is taken to mitigate them. In the space environment, outside the protection of the magnetosphere of the Earth, SEUs become a critical concern because of the high radiation levels. SEUs can have serious consequences for the spacecraft, including loss of information, functional failure or loss of control [3].

Error detection and correction (EDAC) functions have proven to be an effective way to protect computer systems against SEU [4–6]. Extra redundant bits or check

C. Colodro-Conde (✉) • R. Toledo-Moreo
Department of Electronics and Computer Technology, Universidad Politécnica de Cartagena, Cartagena, Spain
e-mail: carlos.colodro@upct.es; rafael.toledo@upct.es

F. Kastensmidt, P. Rech (eds.), *FPGAs and Parallel Architectures for Aerospace Applications*, DOI 10.1007/978-3-319-14352-1_13

bits are stored in memory along with the original information, so that it can be checked at the time of reading whether there have been any alterations due to SEU. This process is usually done by hardware, by means of dedicated EDAC core located between the protected memory and the CPU that wants to access the data. Memory scrubbing, either by hardware or software, is often used to improve reliability [7].

Among all the existing EDAC functions, the odd-weight column codes proposed in [8] are extensively used in many applications, as a result of their SEC-DED (single-error correction and double-error detection) capabilities and their relatively low hardware needs [9–11]. The studies presented in this paper will focus on this type of codes.

The main contribution of this paper consists on a custom synthesis algorithm, specialized in the particular problem of many-input logic gates (i.e., gates with a number of inputs much higher than the number of inputs per LUT of the target FPGA). This algorithm is applied to EDAC encoders and decoders, which can be seen as a set of many-input XOR gates. Each individual XOR gate is mapped into FPGA LUTs in such a way that the area utilization and the length of the critical path is minimized, which results in an overall improvement when the synthesized circuits are compared to those obtained by a commercial synthesis tool.

Most of the previous work about optimizing the implementation of EDAC has been aimed to ASIC devices [5, 6, 9]. The optimization goal is usually the number of transistors, which is required to be low so that die area is kept to a minimum. Nonetheless, in some cases the speed of the resulting circuit is also considered.

In FPGA devices, as opposed to ASICs, one does not have the freedom to create custom cells at layout level, so the optimization cannot be done following this approach. Another difference is that the main parameters that determine area utilization in FPGAs are usually the number of utilized LUTs and flip-flops, rather than the number of transistors. In a 6-input LUT FPGA, the hardware cost of instantiating any logic function up to 6 inputs is virtually the same, while in ASICs the number of inputs of a gate makes a great difference. Because of these reasons, a specific synthesis method was developed for this paper, with the focus on FPGA devices.

The paper is structured as follows. Section 13.2 explains the theory behind the type of EDAC codes that will be the focus of our study. Section 13.3 describes the proposed algorithm formally, including an example for illustrative purposes. Section 13.4 presents the results obtained with the proposed method, and compares those results to the ones achieved by a commercial synthesis tool. Finally, Sect. 13.5 draws the main conclusions.

13.2 SEC-DED EDAC Codes

According to coding theory [12], a SEC-DED EDAC code can be defined via its parity-check matrix $\mathbf{H}_{r\times n}=\left\{h_{ij}\right\}$. This matrix has r rows and $n=r+k$ columns, being r the number of parity bits, k the number of data bits and n the codeword length. The codeword $\mathbf{m}=\left\{m_i\right\}$ is formed by concatenating the input data bits $\mathbf{d}=\left\{d_i\right\}$ and the calculated check bits $\mathbf{c}=\left\{c_i\right\}$, and it is the one that is actually stored in memory for protecting the original data.

For calculating the bit i ($1 \le i \le r$) of the check bits **c**, one has to take the row i from the **H** matrix and check the positions where $h_{ij} = 1$, with $1 \le j \le k$. These positions indicate the elements of the data bits vector **d** that have to be XOR'ed between themselves in order to obtain c_i.

In most applications, the codeword **m** is saved in a memory which may suffer from undesired bit alterations caused by SEU. When the data needs to be recovered, the syndrome vector $\mathbf{s} = \{s_i\}$ has to be calculated in order to know if the codeword has been altered. With SEC-DEC codes, the syndrome vector can be used to spot the location of single-bit errors so that they can be corrected with a bit flip. If the obtained syndrome is equal to 0, it means that the retrieved codeword is the same as the one that was originally saved. If the syndrome is not equal to 0 and the parity of the syndrome is even, a double-bit error flag can be raised.

For calculating the bit i ($1 \le i \le r$) of the syndrome vector **s**, one has to take the row i from the **H** matrix and check the positions where $h_{ij} = 1$, with $1 \le j \le n$. These positions indicate the elements of the codeword **m** which have to be XOR'ed between themselves in order to obtain s_i.

Let us illustrate the procedure explained above with the following example **H** matrix:

$$\mathbf{H} = \begin{bmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{13.1}$$

The **H** matrix above represents the only odd-weight-column SEC-DEC code that generates an 8-bit length codeword ($n = \mathbf{8}$) for a 4-bit data word ($k = \mathbf{4}$). In coding theory, this is denoted as an (8, 4) code. For **H** under consideration, the check bits shall be calculated as follows:

$$\begin{aligned} c_1 &= d_1 \oplus d_2 \oplus d_3 \\ c_2 &= d_1 \oplus d_2 \oplus d_3 \\ c_3 &= d_1 \oplus d_3 \oplus d_4 \\ c_4 &= d_2 \oplus d_3 \oplus d_4 \end{aligned} \tag{13.2}$$

Assuming an input data vector $\mathbf{d} = [1011]$ and applying Eq. 13.2, the check bits would be $\mathbf{c} = [0010]$. The codeword, that is, the actual bits that will be saved in memory would be $\mathbf{m} = [10110010]$.

According to the procedure described previously, the syndrome shall be calculated with the following equations:

$$\begin{aligned} s_1 &= m_1 \oplus m_2 \oplus m_3 \oplus m_5 \\ s_2 &= m_1 \oplus m_2 \oplus m_4 \oplus m_6 \\ s_3 &= m_1 \oplus m_3 \oplus m_4 \oplus m_7 \\ s_4 &= m_2 \oplus m_3 \oplus m_4 \oplus m_8 \end{aligned} \tag{13.3}$$

If neither the data bits or the check bits have been altered, the resulting syndrome is $\mathbf{s} = [0000]$, as expected. However, if we flip m_3, for example, we would get $\mathbf{s} = [1011]$. By inspecting $\mathbf{H}$, we can spot that the error occurred at d_3, as $\mathbf{s} = [1011]$ matches with the third column of $\mathbf{H}$.

13.3 Description of the Algorithm

The purpose of the proposed algorithm is to synthesize a given many-input logic gate in such a way that it allows an efficient implementation on the desired FPGA technology.

It was explained in Sects. 13.1 and 13.2 that EDAC coders or decoders could be seen as a set of r XOR gates, one per each check bit or syndrome bit. Therefore, this algorithm can be used to synthesize EDAC functions by applying it for each one of those defining gates.

One input of the algorithm is the logic function to implement. The other input is the maximum number of inputs per LUT (K), which is used to model the target FPGA. Other characteristics of the selected FPGA technology like the routing architecture are not taken into account.

The input logic function has to be associative, so that it can be implemented with a number of logic gates with K or less inputs, being K the maximum number of inputs per LUT of the target FPGA technology. Additionally, the logic function needs to be commutative so as to allow changing the order of the operands freely. The XOR operation, in which the EDAC functions are based on, meet both properties. The same happens with the AND, OR and XNOR operations. If one wants to implement an n-input NAND or NOR function, they have configure the algorithm to select the non-negating equivalent (e.g., AND instead of NAND) for all the gates of the circuit except the one at the output.

The output of the algorithm is the optimized netlist. In the current software implementation of the algorithm, the resulting netlist can be given to the user in three different ways. One possible way is a textual report from which the actual circuit can be easily inferred. Another way is a graphical representation of the generated circuit, with all the gates and nets drawn. The last output format is a technology-dependent VHDL source file, aimed to be added to the FPGA design flow and integrated with other source files.

13.3.1 Optimization Goals

The main optimization goal is the area occupation, which has to be kept as low as possible. Less hardware not only allows more logic to be integrated in the same FPGA, but it also improves reliability, as every component has an intrinsic rate of failure [6]. Nevertheless, some effort is also put in optimizing processing speed.

The area occupation can be calculated as the number of utilized LUTs. Other specialized FPGA resources such embedded multipliers will be ignored as they cannot be exploited for implementing generic logic gates.

For measuring processing speed, path delays will be considered instead of clock periods, as the EDAC block will be a purely combinational circuit. A combinational circuit is preferred in SEU-sensitive systems, because adding a memory element like a flip-flop inside the EDAC block itself would require additional protecting circuits as in TMR (Triple Modular Redundancy), thus worsening the area and speed parameters. Nevertheless, if faster processing speeds are required, the designer is free to manually add as many pipelining flip-flops as needed.

The resulting maximum path delay, which will determine the processing speed, can only be obtained by the tools provided by the vendor of the target FPGA after the synthesizing stage. Because of this reason, the parameter to minimize will not be the maximum path delay itself. Instead, another variable which is directly related to the maximum path delay will be considered, namely the maximum number of LUTs that a path has to go through (from now on, *levels*). Each LUT adds a certain delay to the path, as well as the nets used to connect two consecutive LUTs, so it can be said that the levels are good estimators of the processing speed.

In summary, the algorithm will be designed to minimize both the number of LUTs and the maximum level.

13.3.2 Step-by-Step Procedure

A generic scenario is composed by a group of n signals, which may come from different gate levels. The objective is to merge all of these signals together using a certain logic function and retrieve the result in a single output. The level of a signal is defined as the number of gates that it has gone through, starting from level 0, which corresponds to the output of a flip-flop. Given such scenario, the steps which have to be followed in order to obtain the desired solution are the following:

1. Take the K_0 signals with lower level and connect them to a new logic gate of the same kind than the target function. The value of K_0 is given by Eq. 13.4.

$$K_0 = \big((n-2) \bmod (K-1)\big) + 2 \tag{13.4}$$

2. The output of the newly instantiated logic gate is assigned a level equal to the maximum level of its inputs plus 1.
3. Take all the unconnected signals (both the original inputs and the outputs of the logic gates) and create a new gate for the K signals with lower level. Again, the output of the newly instantiated logic gate is assigned a level equal to the maximum level of its inputs plus 1.
4. Repeat Step 2 iteratively until there is only one output signal remaining.

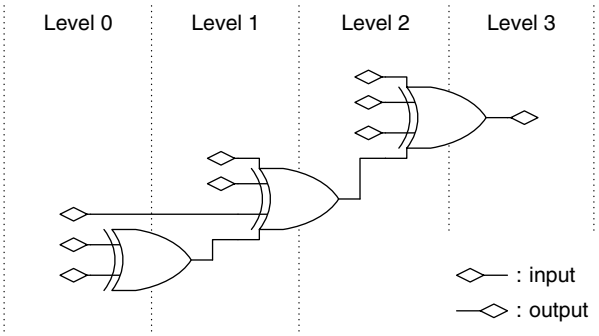


Fig. 13.1 Example of application of the proposed algorithm

The algorithm above minimizes the area utilization and the maximum path delay of the implementation of any n -input logic gate, in terms of the number of utilized LUTs and the maximum level. The resulting number of LUTs (or gates) can be easily calculated using the following formula:

$$\text{num\_LUTs} = \left\lceil \frac{n-1}{K-1} \right\rceil \tag{13.5}$$

with $n > 1$ and $K > 1$.

In the cases where the level of all the input signals is the same, the maximum level can be calculated as follows:

$$\text{max\_level} = \lceil log_K n \rceil \tag{13.6}$$

There is no simple mathematical expression for calculating the maximum level when the input signals have different initial levels. In such cases, the easier way to obtain the value of this parameter is to inspect the resulting circuit.

As an example, Fig. 13.1 shows the result of applying the proposed algorithm to eight signals with different initial levels, for $K = 4$. The output of the circuit is the logic combination of all inputs, in this case, a XOR function. The resulting circuit has a total of 3 gates or LUTs and a maximum level of 3.

13.4 Results

In this section, the results of applying the algorithm described in Sect. 13.3.2 will be compared to those obtained with a commercial synthesis tool, namely *Synplify Pro Microsemi Edition*® [13]. This synthesizer is shipped along with *Libero SoC* and *Libero IDE*, which are the software suites provided by Microsemi for designing with their *RT ProASIC3* [14] and *RTAX-S* [15] radiation-hardened FPGAs. Due to

compatibility issues, version I-2013.09 M-SP1 of Synplify will be used for the RT ProASIC3 FPGAs, while version G-2012.09A-SP4 will be used for the less recent RTAX-S models.

The two synthesis methods under study have been fed with the same input, though expressed in different forms. For the proposed algorithm, the input logic functions are provided as a set of vectors containing the level of each input signal. This means that the steps described in Sect. 13.3 have to be applied separately to each output of the EDAC function, as the algorithm can only handle one many-input logic gate at a time. All the inputs will be considered to have a level equal to 0, which means that they come directly from the outputs of a set of flip-flops, with no additional combinational circuits in their way.

In the case of commercial synthesis tools such as Synplify, the input function is usually specified with hardware description languages (HDLs) like VHDL or Verilog. In this work, VHDL has been selected. The EDAC functions have been defined in this language just as they appear in the equations, with no further manipulations. For example, the EDAC decoder represented by Eq. 13.3 would be defined as follows:

```
s(1) <= m(1) xor m(2) xor m(3) xor m(5);
s(2) <= m(1) xor m(2) xor m(4) xor m(6);
s(3) <= m(1) xor m(3) xor m(4) xor m(7);
s(4) <= m(2) xor m(3) xor m(4) xor m(8);
```

Signals `m` and `s` are connected to a set of flip-flops, which in turn are connected to the input and output ports of the top level entity, so they are associated with physical pins of the FPGA. The existence of the flip-flops allows the placing tools to make the circuits more compact, so the comparison of the maximum path delay is more fair. Given that no registers have been instantiated between signals `m` and `s`, the core of the circuit will remain purely combinational.

When a synthesis tool is given a piece of VHDL code like the one shown above, it generates a netlist in EDIF format which is specific for the target technology. The netlist defines a circuit that can only contain the components available in the target FPGA, without specifying where they will be placed or how the connections will be routed throughout the die. Those tasks are in charge of the place and route tools provided by the FPGA vendor, in this case, Libero SoC and Libero IDE from Microsemi.

The proposed algorithm produces a simplified circuit, but such circuit has to be inserted somehow in the design flow, so that the place and route tools can finish implementing it. One possible way is to generate an EDIF netlist, emulating a normal synthesis tool. However, the authors chose to generate a technology-dependent VHDL file, which is a VHDL file that includes specific libraries for the target technology and only instantiates components that are present in such technology. Using this technique, the generated VHDL cores can be easily integrated in larger designs by adding them into the design flow just like any other HDL source file. If the VHDL file is defined correctly, a commercial synthesis tool like Synplify will not modify the circuits defined in those files. Instead, it will perform a direct translation from VHDL to EDIF.

One of the target FPGA models for this study will be an RT3P600L, from the relatively recent Microsemi RT ProASIC3 family. The core of this FPGA consists of a sea of 13,824 cells called *VersaTiles*. Each cell can be configured either as a 3-input LUT (C-cell), a D-flip-flop or a latch (R-cell), and they may be connected between themselves through any of the four levels of routing hierarchy. In the case of the present study, where the circuits will be purely combinational, all the used cells will be configured as LUTs. Unlike other radiation-tolerant FPGAs, which use antifuse programming technologies, devices in the RT ProASIC3 family use flash cells to store configuration information. This fact worsens the tolerance to radiation, but accelerates the development process and reduces its cost considerably.

The other FPGA model that will be selected for the tests is a RTAX250S, from the Microsemi RTAX-S family. The high reliability against radiation of these antifuse FPGAs makes them very popular for space applications. The RTAX architecture comprises a sea of two types of logic modules: the combinatorial cell (C-cell) and the register cell (R-cell). Each C-cell can implement a selection of more than 4,000 types of functions of up to 5 inputs, and they also contain carry logic for implementing arithmetic operations efficiently. As in the RT ProASIC3, there are four kinds of interconnecting lines, with different lengths and delays.

For dealing with EDAC functions, the synthesis tools need to instantiate a number of XOR gates of different size. The maximum number of inputs that a XOR gate can have in the selected FPGA architecture is specified in the corresponding macro library guide. For example, a RT ProASIC3 C-cell can implement either a 2-input XOR or a 3-input XOR. The same happens with the RTAX-S C-cells, though it is also possible to connect two adjacent C-cells through the dedicated carry logic in order to obtain a 4-input XOR with the delay of a single cell. The effect of using this feature or not will be analyzed later in this section by setting $K = 3$ or $K = 4$ for the proposed algorithm ($K = 3$ would disable the instantiation of these XOR4 components).

Given the described setup, the post place and route implementation results for an EDAC decoder are presented in Tables 13.1 and 13.2. The first one corresponds to the results with the RT3P600L, while the latter refers to the RTAX250S. For each table, three different EDAC functions taken from [8] have been tested. The area and timing figures have been extracted from the reports generated by the place and route tools. The maximum path delay is measured from the input to the output of the EDAC entity.

Table 13.1 Post place and route implementations of a set of EDAC decoders in a RT3P600L FPGA

EDAC function	Synthesis method	C-cells	Max. level	Max path delay (ns)
Hsiao (22, 16)	Synplify Pro ME I-2013.09 M	29	3	5.207
	Proposed algorithm ($K=3$)	24	2	4.306
Hsiao (39, 32)	Synplify Pro ME I-2013.09 M	58	4	6.030
	Proposed algorithm ($K=3$)	49	3	5.348
Hsiao (72, 64)	Synplify Pro ME I-2013.09 M	116	5	7.980
	Proposed algorithm ($K=3$)	104	3	6.098

Table 13.2 Post place and route implementations of a set of EDAC decoders in a RTAX250S FPGA

EDAC function	Synthesis method	C-cells	Max. level	Max path delay (ns)
Hsiao (22, 16)	Synplify Pro ME G-2012.09A	30	2	4.701
	Proposed algorithm ($K=3$)	24	2	4.929
	Proposed algorithm ($K=4$)	30	2	5.018
Hsiao (39, 32)	Synplify Pro ME G-2012.09A	63	3	6.272
	Proposed algorithm ($K=3$)	49	3	5.884
	Proposed algorithm ($K=4$)	63	2	5.072
Hsiao (72, 64)	Synplify Pro ME G-2012.09A	136	3	7.275
	Proposed algorithm ($K=3$)	104	3	6.746
	Proposed algorithm ($K=4$)	136	3	7.188

All the synthesis, place and route tools have been configured to their default parameters. It is worth mentioning that neither Synplify nor the proposed algorithm can establish the optimization goal (area or speed) or the optimization effort of the synthesis process. Synplify does allow to enable/disable the *Resource sharing* and *Retiming* functionalities, but they have no effect over the application under study.

Attending to the results shown in Table 13.1, it is clear that the proposed algorithm outperforms Synplify in terms of number of utilized C-cells, at least for the RT ProASIC3. The reduction obtained with respect to Synplify is between 10.3 and 17.2 %. This means that the main objective of reducing the area utilization has been accomplished.

It was said in Sect. 13.3.1 that the algorithm would also try to reduce the maximum path delay as much as possible by minimizing the maximum level (i.e., the maximum number of gates that a path goes through). In all of the tested cases, the proposed algorithm produces better results than Synplify in this sense, with differences between 11.3 and 23.6 % for the maximum path delay. The largest difference in the maximum path delay occurs when the difference in the maximum level is also the largest, as could be expected from the assumptions made in Sect. 13.3.1. However, it should be noted that the maximum path delay depends heavily on the performance of the place and route tools, in fact, it even depends on the random seed that these tools start with. It was tested that the relative difference between the smallest and largest maximum path delays obtained after 25 runs of the place and route tools, with the same input netlist and different initial random seeds, was around 10 %. Considering this number, we may conclude that the smallest timing improvement obtained by the proposed algorithm (11.3 %) could be considered significant, despite the indeterministic nature of the place and route tools.

After this first battery of tests, an analogous study was performed selecting the RTAX-S FPGA. With the RT ProASIC3, the proposed algorithm was always configured with $K = 3$, meaning that it can only instantiate XOR gates up to 3 inputs, each one of them corresponding to a C-cell. It was said in previous paragraphs that the different architecture of the RTAX-S FPGAs also allows to implement a 4-input

XOR with two C-cells with a delay similar to that of one single C-cell. In order to study the effect of using this feature, the proposed algorithm was configured first with $K = 4$ and then with $K = 3$. By analyzing the synthesis reports generated by Synplify, it was discovered that this tool always tries to take advantage of the two-cell 4-input XORs (i.e., it acts as if it was set with $K = 4$).

Table 13.2 shows that both Synplify and the proposed algorithm (with $K = 4$) produce the same results in terms of number of utilized C-cells for the RTAX250S. This could be explained because Synplify uses a different algorithm for this architecture of FPGA, now focusing on minimizing the area utilization of each logic function that defines an output, which are processed independently. This behaviour is exposed by the fact that every instantiated C-cell has a fan-out equal to 1. This is different than with the RT ProASIC3 FPGA, where Synplify treated the input EDAC function as a whole, allowing to re-use some terms that are common to two or more outputs of the circuit. If done correctly, this more elaborate approach would theoretically allow to obtain smaller circuits with shorter critical paths, but Synplify was not able to exploit this approach efficiently for the case of the EDAC functions.

It is interesting to note that, even though Synplify has followed a similar approach to that of the proposed algorithm, resulting in an equal number of C-cells, it has obtained a higher maximum level for the Hsiao (39, 32) function. Besides revealing a flaw of Synplify in the independent processing of outputs, this fact ultimately results in a significantly larger maximum path delay, with a difference of 19.1 % for this function. In the rest of the cases, the identical maximum level results in similar maximum path delays, with differences below 6.5 % (recall that the performance of the place and route tools also have a remarkable impact on this parameter).

It remains to be checked whether the fact of combining two C-cells to form a 4-input XOR is beneficial to the application under study. Table 13.2 demonstrates that ignoring this feature (that is, setting $K = 3$) is more beneficial for the case of the EDAC functions. This result is specially relevant because Synplify does not allow to disable the use of combined C-cells. The difference in area utilization is above 20 %, reaching 23.5 % for the largest tested function. In spite of using smaller XOR gates, the maximum path delay is not significantly affected because the maximum level happens to be the same with $K = 3$ and $K = 4$ for the functions under study. Note that not every possible input EDAC function would follow this property, as can be deduced by applying Eq. 13.6.

13.5 Conclusions

In this work, a new synthesis algorithm that improves the implementation of EDAC codes in radiation-hardened FPGAs has been presented, and the results were compared to those obtained by the commercial synthesis tool that ships with the software suite provided by the vendor of the selected FPGA models.

The proposed algorithm has proven to have less area utilization than Synplify for all the tested cases, which include different input EDAC functions and target FPGAs architectures. The reduction of the area utilization is substantial (up to 23.5 % for the best case), and still, the speed of the inferred circuits is either maintained or improved (up to a 23.6 %).

The results given in Sect. 13.4 reveal that there is a considerable margin for improvement in the world of synthesis tools. In the case of the EDAC functions, the authors discovered that processing each output separately in an optimal way (in terms of number of instantiated LUTs and length of the critical path) can have a positive impact on the entire circuits. Moreover, it was concluded that Synplify's default behaviour of using the 4-input XORs available in the RTAX-S macro library is not beneficial for the implementation of EDAC functions.

In which each bit indicates one possible cause of an error, all the bits can be OR'ed together in order to obtain a general error flag. Another example is a simple parity check over a register, which can be based on a many-input XOR gate.

As future work, it would be interesting to investigate whether processing several outputs jointly allows improving the results even further. A strategy similar to the one followed by Synplify for the RT ProASIC3 may be used, consisting on re-using some terms that are common to two or more outputs of the circuit. Given that Synplify was not able to process the single outputs in an optimal way (see Table 13.2), it would not be surprising to discover that there is also room for improvement when multiple-output functions are considered as a whole, instead of processing each output separately.

Acknowledgments This work was supported by the Spanish Ministry of Educación, Cultura y Deporte under the grant FPU12/05573, and by the Spanish Ministry of Economía project AYA2012-39702-C02-02, in the frame of the activities of the Instrument Control Unit of the Infrarred Instrument of the ESA Euclid Mission carried out by the Dept. of Electronics and Computer Technology of the Universidad Politécnica de Cartagena.

References

1. Chandra V, Aitken R (2008) Impact of technology and voltage scaling on the soft error susceptibility in nanoscale CMOS. In: IEEE International Symposium on Defect and Fault Tolerance of VLSI Systems, 2008. DFTVS'08, IEEE, Boston, pp 114–122. doi:10.1109/DFT.2008.50
2. Mahatme N, Jagannathan S, Loveless T, Massengill L, Bhuva B, Wen SJ, Wong R (2011) Comparison of combinational and sequential error rates for a deep submicron process. IEEE Trans Nucl Sci 58(6):2719–2725
3. Barth JL, Dyer CS, Stassinopoulos EG (2003) Space, atmospheric and terrestrial radiation environments. IEEE Trans Nucl Sci 50(3):466–482
4. Bentoutou Y (2012) A real time EDAC system for applications onboard earth observation small satellites. IEEE Trans Aerosp Electron Syst 48(1):648–657. doi:10.1109/TAES.2012.6129661
5. Gao W, Simmons S (2003) A study on the VLSI implementation of ECC for embedded DRAM. In: Canadian Conference on Electrical and Computer Engineering, 2003. IEEE CCECE 2003, vol 1, pp 203–206. doi:10.1109/CCECE.2003.1226378

6. Hao L, Yu L (2008) A study on the hardware implementation of EDAC. In: Third International Conference on Convergence and Hybrid Information Technology, 2008. ICCIT'08, vol 2, pp 222–225. doi:10.1109/ICCIT.2008.14
7. Saleh A, Serrano J, Patel J (1990) Reliability of scrubbing recovery-techniques for memory systems. IEEE Trans Reliab 39(1):114–122. doi:10.1109/24.52622
8. Hsiao M (1970) A class of optimal minimum odd-weight-column SEC-DED codes. IBM J Res Dev 14(4):395–401. doi:10.1147/rd.144.0395
9. Aymen F, Belgacem H, Chiraz K (2011) A new efficient self-checking hsiao SEC-DED memory error correcting code. In: 2011 International Conference on Microelectronics (ICM), IEEE, Hammamet, pp 1–5. doi:10.1109/ICM.2011.6177346
10. Chen PY, Yeh YT, Chen CH, Yeh JC, Wu CW, Lee JS, Lin YC (2006) An enhanced EDAC methodology for low power PSRAM. In: IEEE International Test Conference, 2006. ITC'06, pp 1–10. doi:10.1109/TEST.2006.297689
11. Johansson R (1996) Two error-detecting and correcting circuits for space applications. In: Proceedings of the Twenty-Sixth Annual International Symposium on Fault-Tolerant Computing (FTCS'96), FTCS'96, IEEE Computer Society, Washington, DC, pp 436–439
12. Fujiwara E (2006) Code design for dependable systems: theory and practical applications. Wiley, Hoboken
13. Synopsys: Synplify Pro ME. http://www.microsemi.com/products/fpga-soc/design-resources/design-software/synplify-pro-me
14. Microsemi: radiation-tolerant ProASIC3 low power spaceflight flash fpgas with flash*freeze technology. http://www.microsemi.com/products/fpga-soc/radtolerant-fpgas/rt-proasic3
15. Microsemi: RTAX-S/SL and RTAX-DSP radiation-tolerant fpgas. http://www.microsemi.com/products/fpga-soc/radtolerant-fpgas/rtax-s-sl

Chapter 14
Neutron-Induced Single Event Effect in Mixed-Signal Flash-Based FPGA

Lucas A. Tambara, Marcelo S. Lubaszewski, Tiago R. Balen, Paolo Rech, Fernanda L. Kastensmidt, and Christopher Frost

Abstract This chapter describes a neutron-induced Single Event Effect test in a commercial Mixed-Signal Programmable System-on-Chip FPGA from Microsemi. The main objective is to investigate the digital and analog parts reliability for critical application projects. The case-study circuit is a data acquisition system that uses analog blocks, buses and interfaces, embedded processor and programmable digital data processing. Two different architectures using design diversity redundancy were implemented, each one composed of specific redundant schemes. The setup was exposed to a neutron source at the CCLRC Rutherford Appleton Laboratory—ISIS in order to investigate the occurrence of SEE ranging from single to bursts of errors. The results are important to characterize the device and to demonstrate the importance of using design diversity redundancy to improve the robustness of a system.

14.1 Introduction

Recent advances in silicon technology have allowed the integration of complex systems into a single chip. Embedded standard processor devices, dedicated processing blocks, interfaces to various peripherals, on-chip bus structures, analog blocks and even configurable logic arrays compose the most recent mixed-signal System-on-Chip (SoC) devices [1]. Further commercial and aerospace market are targeting more and more nowadays low power, cost, high integration and computational capability, which drives the growth of this type of programmable mixed-signal SoC [2]. Such components can help board integration and it adds configurability and flexibility to the design project.

L.A. Tambara (✉) • M.S. Lubaszewski • T.R. Balen • P. Rech • F.L. Kastensmidt
Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre, Brazil
e-mail: latambara@inf.ufrgs.br; luba@ece.ufrgs.br; tiago.balen@ufrgs.br; prech@inf.ufrgs.br; fglima@inf.ufrgs.br

C. Frost
Rutherford Appleton Laboratory—ISIS, Didcot, UK
e-mail: c.d.frost@rl.ac.uk

F. Kastensmidt, P. Rech (eds.), *FPGAs and Parallel Architectures for Aerospace Applications*, DOI 10.1007/978-3-319-14352-1_14

Integrated circuits operating at ground level can be exposed to Single Event Effects (SEE) effects induced by neutrons. Such neutrons are generated by the collision of cosmic galactic rays with atoms in the atmosphere producing high-energy neutrons. It is known that these high-energy neutrons can lead memory cells to change their states, causing the appearance of Single Event Upset (SEU) [3], especially in modern semiconductor electronics. In addition, transient effects can also be observed in combinational and analog components known as Single Event Transient (SET) pulses that vary in amplitude and time duration. To deal with these transient effects, traditional techniques based on redundancy are used [4]. A very successful example is the *N* Modular Redundancy (NMR). In the NMR, *N* copies of the design are implemented and operate in tandem. Double Modular Redundancy (DMR) is an example of NMR use for error detection, where two copies work in parallel receiving the same inputs or copies, and the outputs are constantly compared to detect errors.

Neutron-induced SEU in commercial programmable mixed-signal SoC must be investigated to analyze the chances of using those components in avionics and aerospace applications. Such devices are composed of analog blocks, buses and interfaces, embedded processor and programmable digital array. This work presents results from a neutron experiment at the CCLRC Rutherford Appleton Laboratory—ISIS that characterizes the Flash-based SmartFusion SoC FPGA from Microsemi. The goal is to analyze the errors signatures in the analog and digital parts and the embedded processor.

In addition, a redundant scheme is proposed to investigate the robustness and detecting errors in the FPGA platform under neutrons. Neutrons reactions can generate many secondary particles that may provoke multiple upsets. When dealing with multiple transient effects or burst of errors, it is necessary to analyze the impact of such multiple faults in a redundant design. If both copies are affected in the same way, the output comparator may not be able to detect errors. However, if each design copy in a redundant system is built with a distinct approach, the probability of multiple faults affecting more than one copy and having the same effect is reduced, since each system copy may have different levels of resilience associated with diverse fault generation mechanisms and sources [5].

For this reason, we developed as a case-study circuit a Design Diversity Redundancy (DDR) scheme composed of a data acquisition system that uses analog blocks, buses and interfaces, embedded processor and programmable digital data processing. Each redundant copy is implemented in a distinct way using different replicas and algorithms. Results show transient effects in the analog and digital parts. A cross-section of the analog data acquisition is presented. These results are important to characterize the device and to indicate the importance of using DDR to improve the robustness of a system. Spice simulations were also performed to enhance the understanding of the error mechanisms on the analog-to-digital converters of the studied device.

14.2 SmartFusion Mixed-Signal SoC Platform

There are many mixed-signal SoC platforms available in the market today. Examples are: Zynq-7000, from Xilinx, which have beyond several peripherals, a dual 800 MHz 32-bit ARM Cortex-A9, a SRAM-based FPGA and a dual 12-bit analog-to-digital converter [6]; Cyclone V, from Altera, which also have beyond several peripherals, a dual 800 MHz 32-bit ARM Cortex-A9 and a SRAM-based FPGA [7]; PSOC from Cypress, and SmartFusion, from Microsemi [8]. Beyond these options, there are also similar technologies, like the Field-Programmable Analog Arrays (FPAAs) and the Multiprocessors SoCs (MPSoCs).

Previous works have investigated either analog components or flash-based or SRAM-based FPGAs separately, but not a mixed-signal SoC in the context of neutron-induced SEE effects. The SmartFusion from Microsemi was chosen for this first work due to the existence of a programmable array in addition to a greater number of standard embedded analog resources when compared to the others cited, a processor and memories. The programmable array can add flexibility to a system using DDR because many implementations can be designed and tested in it, improving the redundant copies implementations.

This device is composed of a Microcontroller Subsystem (MSS), with a 100 MHz 32-bit ARM Cortex-M3 and several peripherals and interfaces, like embedded memories (eSRAM and eNVM) and buses (Advanced Peripheral Bus—APB and an Advanced High-Performance Bus—AHB); a Flash-based Field Programmable Gate Array (FPGA) based on ProASIC3 architecture; and an Analog Compute Engine (ACE) with 8/10/12-bit successive approximation analog-to-digital converters (ADCs), 8/16/24-bit $\Sigma\Delta$ digital-to-analog converters (DACs), internal voltage reference, active bipolar prescalers, voltage monitors, current monitors, temperature monitors, voltage comparators and direct inputs. All the system was mounted using a proprietary tool called *Libero SoC* [8] and configured to run at 10 MHz. Figure 14.1 shows the floorplanning of the A2F200-FG484 SoC.

14.3 Proposed Case-Study Approach Using Redundancy

A case-study circuit composed of two redundant copies was developed, each one using distinct analog parts, the Cortex-M3 processor and part of the FPGA matrix.

14.3.1 Mixed-Signal DUT with Design Diversity Redundancy (DDR) Approach

A data acquisition architecture using DDR was proposed as depicted in Fig. 14.2. In this case, the DDR design is applied in a DMR approach, here called DDR-DMR approach. This scheme considers different system levels (software and hardware) to build the system copies. The two system copies perform the same function, but are

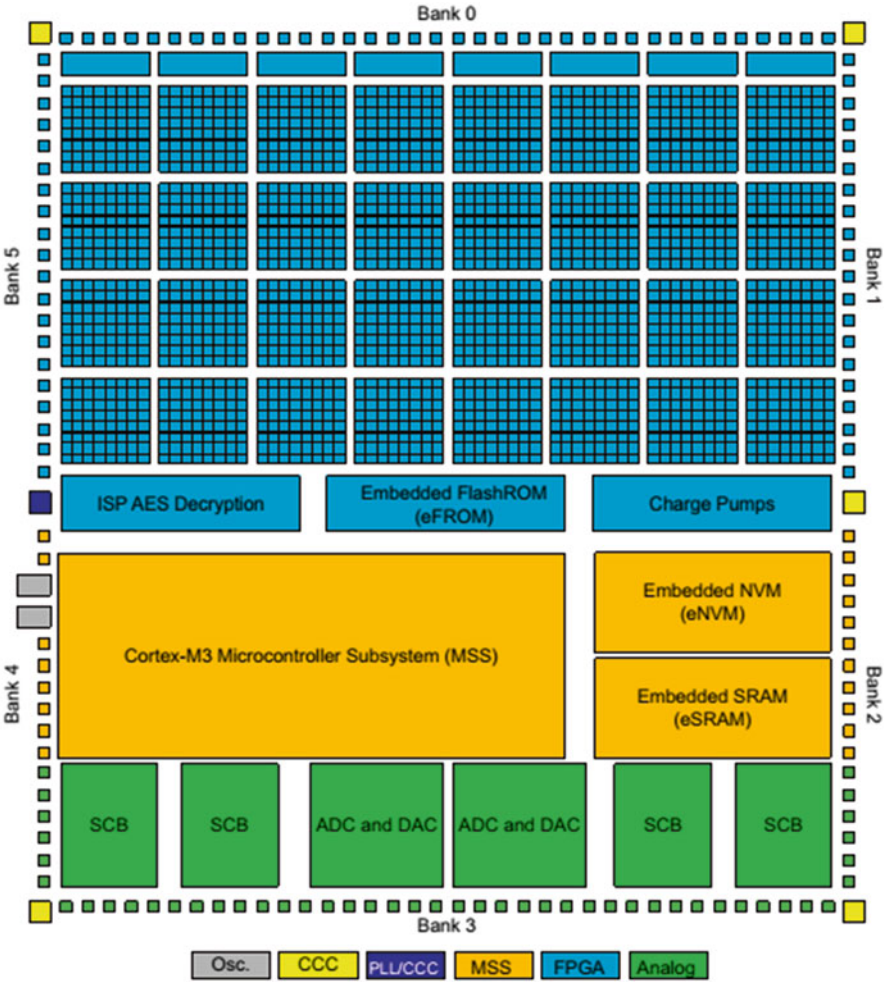


Fig. 14.1 SmartFusion floorplanning [8]

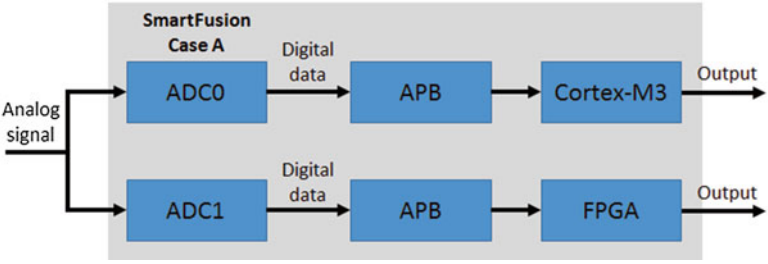


Fig. 14.2 Redundant diversified architecture proposed

implemented in different levels, as follows: a digital copy implemented by software into the Cortex-M3 processor and a digital copy implemented by hardware into the programmable FPGA array. Basically, the implementation includes a sequence of analog-to-digital conversions and digital signal processing.

Redundancy is a well-known approach frequently applied in modern digital systems, which require a high level of reliability [9]. Design diversity redundancy was widely discussed for the first time in [5], where the use of different approaches or architectures in order to generate a redundant scheme is an alternative to increase the reliability of a system. To implement this technique, each circuit copy is implemented with different technologies, algorithms or architecture. Then the basic idea is that, with different implementations, one fault will probably cause different errors.

However, past works have not addressed the challenging of using design diversity redundancy (DDR), especially for error detection in complex mixed-signals systems, where data must be acquired by analog blocks, specific buses needed to be used, and the digital data must be processed and stored in specific units. In this case, all the paths must be redundant to achieve minimal fault isolation. For example, the previous works [10, 11] have shown Diversity Triple Modular Redundancy (DTMR) based on a PSoC platform in order to show that such approach is a feasible technique for error detection and to increase the reliability of some classes of state-of-art mixed-signals circuits. However, in [11] the setup had shared resources between the redundant copies. Moreover, the results of both works are based on hardware and fault simulation.

In this context, the digital copy implemented by software was developed using the standard libraries of C language and the proprietary Microsemi libraries responsible for the microcontroller subsystem manipulation. The digital copy implemented by hardware in the FPGA matrix uses a Finite State Machine (FSM) and a dedicated data path designed in VHDL. Figure 14.3 illustrates the application architecture, where an identical analog input signal is provided to the SmartFusion with the DDR-DMR approach embedded. The data flow works as follows: first, the analog signal is received by the two ADCs and processed by the ACE; then, the converted values are allocated in distinct AHB addresses; and finally, both Cortex-M3 and FPGA get your respective value from the bus to process it.

The FPGA and the Cortex-M3 were configured to run at 10 MHz and the analog blocks were configured to run at 2.5 MHz. Both ADCs were set to work with a resolution of 12 bits and an internal voltage reference. A significant difference between the Cortex-M3 and the FPGA is the fact that the data acquisition in the FPGA is continuous, which not happens in the Cortex-M3, where the acquisition is periodic. To deal with this, a trigger was configured from the Cortex-M3 to the FPGA in order to synchronize the data acquisition.

14.3.2 Complementary Digital Designs

Two 1,000-stages shift registers were embedded into the FPGA array as a test circuit to evaluate the neutron-induced SEU sensitivity of the programmable flash-based array. Both circuits were set to have all their bits in 0 and run at 10 MHz, the same

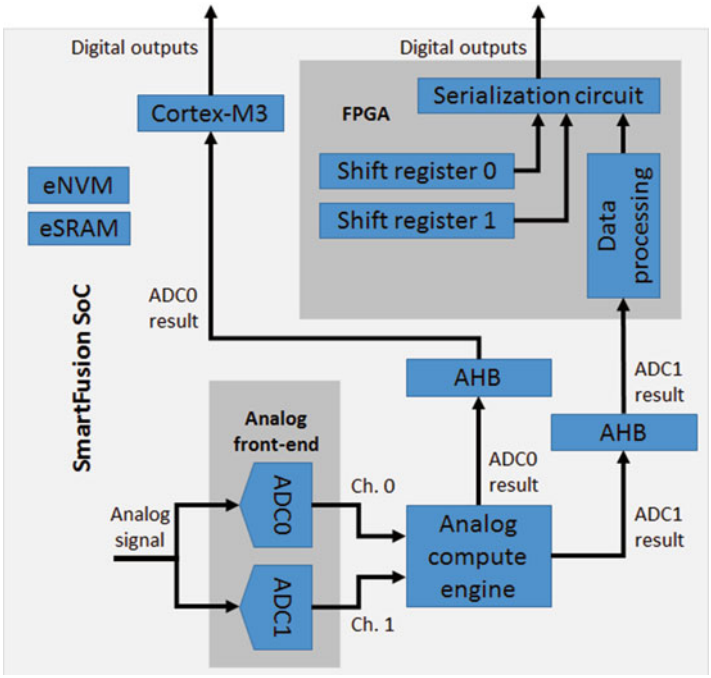


Fig. 14.3 DDR-DMR proposed architecture

clock of the other digital circuits and processor. A serialization circuit was also implemented in the FPGA in order to provide a redundant data acquisition. All the circuits were manually placed side by side within the FPGA array. Figure 14.3 illustrates all these complementary circuits together with the proposed setup.

14.4 Neutron Test Setup

The test setup is composed of a motherboard and the Device Under Test (DUT) board connected each other point-to-point. Both boards make use of an A2F200-FG484 SmartFusion. All the collected data are transferred to a laptop through a Universal Asynchronous Receiver Transmitter (UART) bus module embedded in the motherboard and stored in .txt archives for posterior analysis.

The motherboard has the follow circuits embedded: a power-on reset circuit to ensure that the synchronous circuitry will start in a known state after the bring-up (power-up and reprogramming cycle); a signal generator circuit through one of the DACs available in the SmartFusion that is responsible to generate a periodical

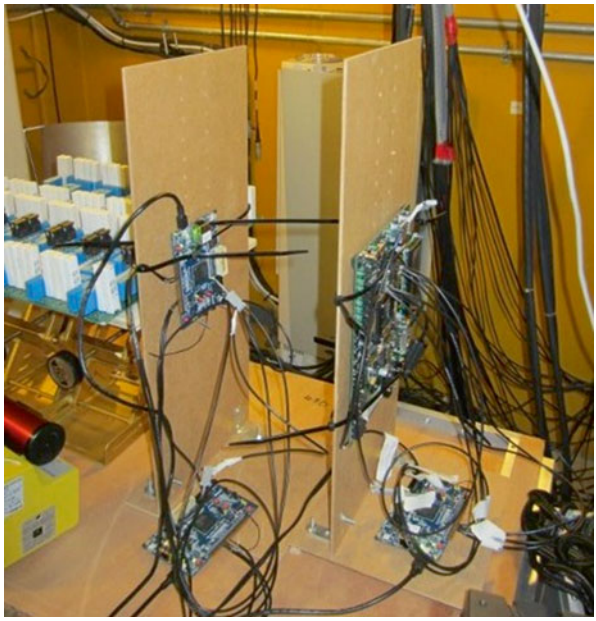


Fig. 14.4 Experiment setup mounted (the second from the right to the left) in the VESUVIO irradiation chamber at ISIS facility. The DUT is in the vertical platform

10 Hz ramp signal with a Least Significant Bit Voltage (Vlsb) of 5 mV and amplitude range from 0 to 2.56 V; and a receptor circuit, responsible to perform the digital data acquisition (shift registers data and converted data) from the DUT. The DUT board composed of the SmartFusion under test has both mixed-signal and digital designs described in Sect. 14.3. The power-on reset circuit consists of two 17-bit counters in cascade. The first counter has the function to generate a delay to the second counter, which generates the reset signal to the rest of the motherboard and the DUT.

The receptor circuit performs two functions. First, it receives the converted values from the DUT and sends them to the UART module of the motherboard. Second, the receptor circuit detects upsets in the shift registers through the follow scheme: if a bit 1 is received, then the next 999 bits of each shift register are analyzed and, at each bit 1 detected, a counter is incremented. If some upset is detected in the shift registers of the DUT, the result of the counter is sent to the UART module and then the data is recorded in the laptop.

The device was tested in a neutron source located at the CCLRC Rutherford Appleton Laboratory—ISIS (Didcot, UK). Neutrons are produced at ISIS by the spallation process: a heavy-metal target (tungsten) is bombarded with pulses of highly energetic protons, generating neutrons from the nuclei of the target atoms [12]. The mean flux obtained from the source was 3.08×10^4 n/cm^2/s for energies above 10 MeV. Figure 14.4 shows the experiment setup mounted inside the VESUVIO irradiation chamber at ISIS.

14.5 Test Results

The DUT setup occupies 94 % of the global resources available in the A2F200-FG484 SmartFusion device. Both processor Cortex-M3 and FPGA are running at 10 MHz, and the analog blocks are configured to run at 2.5 MHz. Without taking into account the interruptions that the software copy suffers, the observed delay between the output of the Cortex-M3 and the FPGA is approximately 2 ms.

Practical measurements were performed aiming to verify the reliability of the SmartFusion in the context of SEU caused by neutrons. We exposed the device to neutron particles at a mean flux of 3.08×10^4 n/cm^2/s by 24 h.

14.5.1 Mixed-Signal Scheme with Diversity Redundancy (DDR-DMR)

Related to the ADCs, it was observed (Table 14.1) a mean cross-section of 8.18×10^{-5} cm^2 for the ADC0 and 7.35×10^{-5} cm^2 for the ADC1. These values are based on all samples (converted values by the ADCs) recorded, including samples with SEU effects and possible burst of errors. Figures 14.5 and 14.6 show examples of

Table 14.1 Test results for the ADCs

Time (h)	Number of samples	Samples with errors (%)		Flux (n/cm^2/s)	Cross section (cm^2)	
		ADC0	ADC1		ADC0	ADC1
06:00	399,492	1.69	2.35	3.07×10^4	5.52×10^{-5}	7.65×10^{-5}
12:00	676,136	2.93	2.22	3.06×10^4	9.58×10^{-5}	7.25×10^{-5}
18:00	413,746	2.65	2.19	3.08×10^4	8.61×10^{-5}	7.11×10^{-5}
24:00	329,319	2.79	2.28	3.09×10^4	9.03×10^{-5}	7.38×10^{-5}

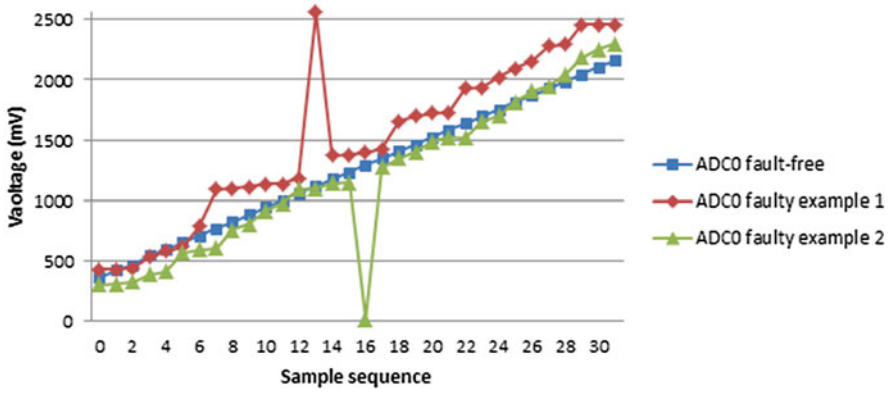


Fig. 14.5 ADC0 samples examples

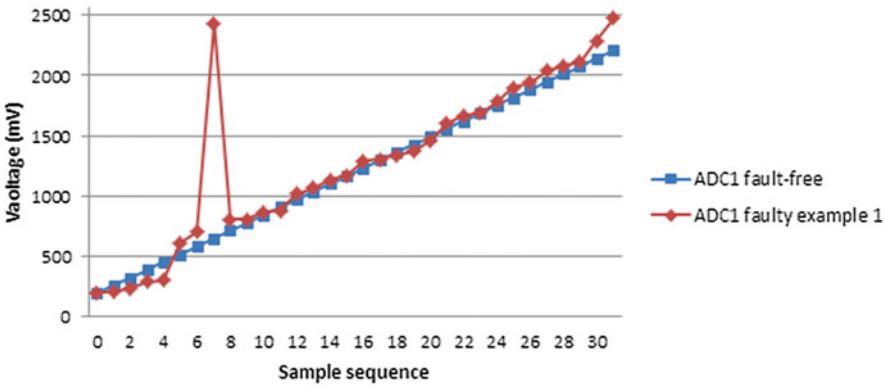


Fig. 14.6 ADC1 samples examples

fault-free and faulty samples caused by SEU. Considering the DDR-DMR scheme, it is possible to observe that the redundant copy that uses the ADC1, controlled by the FPGA array, showed a better regular behavior than other redundant copy that uses the ADC0, controlled by the Cortex-M3 processor. A possible justification for this behavior is in the fact that the continuous acquisition by the FPGA creates a scenario of data oversampling, which it does not happen with the Cortex-M3. It is important to mention that there were not observed negative peaks in the ADC1.

From 1,818,693 redundant converted samples (from ADC0 and ADC1) collected during the neutron test, no errors were observed in both ADCs at the same time. This confirms that using diversity redundancy we can detect faults in the ADCs data path. For specific cases, a pass filter could also be used to filter out the transient error, according to the application and expected data. Other solutions can be further investigated.

When the ADC topologies are investigated, it is well known that there are ADC topologies more robust to radiation than others. For example, the $\Sigma\Delta$ ADC architecture has been proved to have a high level of radiation capability [13]. However, the ADCs of the SmartFusion are a switched-capacitor successive approximation register (SAR) (Fig. 14.7) [8], which contain an expressive digital part that can be easily upset by neutrons.

There are two main possibilities for transient upset occurrence in those ADCs:

- As one can observe in Fig. 14.7a, the SAR ADC is based on a set of capacitors, a DAC, a comparator and a sample and hold (S/H) circuit to acquire the input voltage. One possibility is to have SEU occurrences in the output register of the DAC or transient pulses in the comparator.
- Another possibility is related to the switched-capacitor array (Fig. 14.7b). If a switch temporarily changes its state, the equivalent capacitance of the array will also be temporarily modified. Therefore, a charge redistribution process between the branches will occur [14], affecting the final converted value.

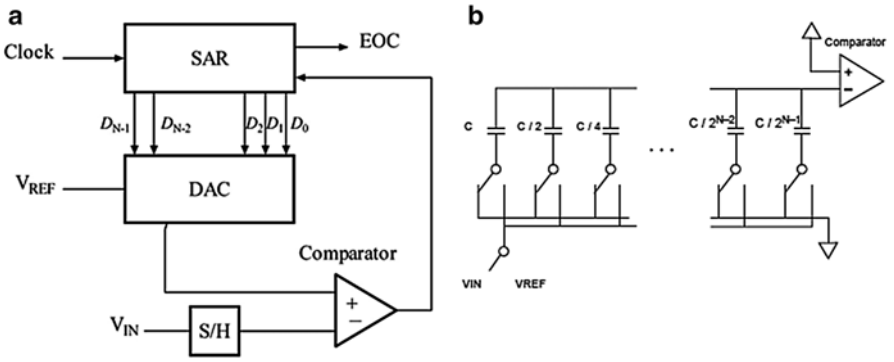


Fig. 14.7 Example of SAR architecture (**a**) and ADC switched-capacitor array architecture (**b**) [8]

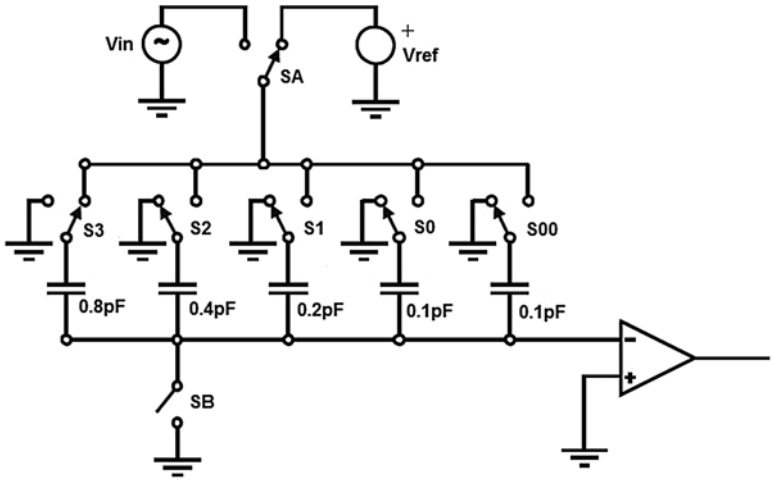


Fig. 14.8 Analog part of a 4-bit charge redistribution SAR ADC simulated in this work

14.5.2 Simulation of a Charge Redistribution SAR-ADC

In order to aid the understanding of the observed effects on the ADCs, Spice simulations were performed. Because detailed information about the ADCs' internal architecture and technology are not available to the user, a 130 nm PTM (Predictive Technology Model) [15] technology model was used in the simulations (performed with HSpice software). For the sake of simplicity and better understanding, the simulated circuit consists in the analog part of 4-bit charge redistribution SAR ADC. Despite this simplification, it is possible to extend the results to real converters, with higher resolutions. Figure 14.8 depicts the simulated circuit. The digital circuit that controls the switches of the capacitor array is not shown.

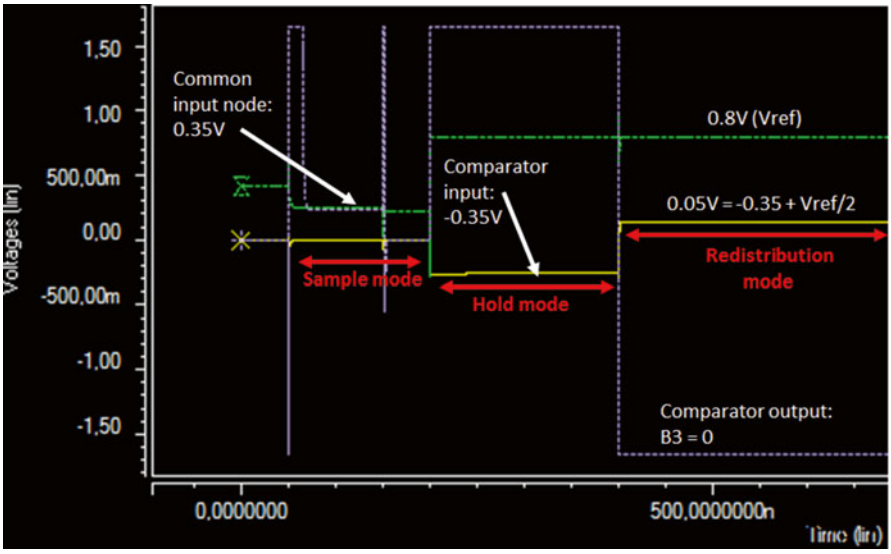


Fig. 14.9 Simulation of sampling, hold and charge redistribution process (only bit B3), considering an input sample of 0.35 V and Vref = 0.8 V

The charge redistribution SAR operates in three distinct phases to convert an analog sample [16]. The first step is the sample mode, in which all capacitors of the array are connected to V_{in} (analog input) through the switches S_A and S_3 to S_{00} (S_B connects the array to ground). This way, an equivalent capacitor of $2C = 1.6$ pF (in this case) is charged with the V_{in} voltage. Then, the hold mode takes place: S_B is opened, S_3 to S_{00} connect all the capacitors to ground and S_A turns to V_{ref} (converter's reference voltage). At the end of this process, a voltage equal to $-V_{in}$ is held at the comparator input. These two modes naturally execute a sample-and-hold process.

The conversion itself is performed in the third phase: the redistribution mode. This mode starts by connecting the MSB (Most Significant Bit) capacitor to V_{ref} through S_3. This way, a capacitive divider with two equivalent capacitors of $C = 0.8$ pF is formed, in a way that the voltage of the comparator input is now $V_c = -V_{in} + V_{ref}/2$. The output of the comparator gives the value of bit B_3 and, depending on its value, the control circuit decides the position of S_3 to the next redistribution steps (if $B_3 = 1$, S_3 remains connected to V_{ref}, otherwise it is grounded). The process is repeated to the other capacitors, and the output of the comparator in each redistribution cycle (or the position of the switches at the end of the redistribution mode) represents the digital converted value. Figure 14.9 shows the simulation of this process, considering the evaluation of the MSB (B_3), for an input sample of 0.35 V and a reference voltage of 0.8 V.

The switches of the capacitor array may be implemented as transmission gates. A Transmission Gate (TG) consists in the interconnection, in parallel, of a PMOS and an NMOS transistor, which need complementary signals to control their states.

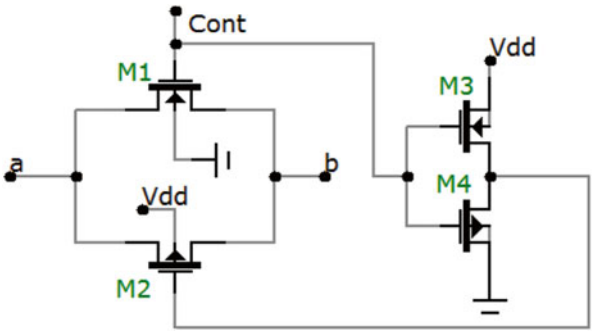


Fig. 14.10 Analog transmission gate controlled by inverter gate

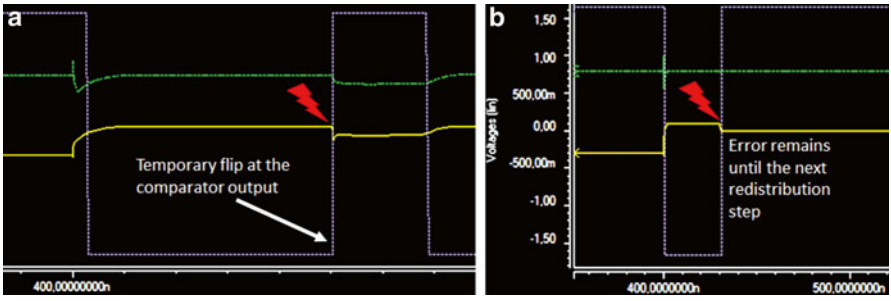


Fig. 14.11 Simulated effect of an SET during the redistribution process: (**a**) at the switch S3, and (**b**) at the drain of NMOS transistor of SB

For this reason, each transmission gate-based switch needs a digital control element, which in the simplest case is a digital inverter. Figure 14.10 shows a transmission gate controlled by a digital inverter. In this work, the sizing of the transistors of the TGs is: $L_1 = L_2 = 0.8$ µm; $W_1 = 8$ µm, $W_2 = 16$ µm; $L_3 = L_4 = 0.15$ µm; $W_3 = 0.8$ µm, $W_4 = 0.4$ µm. All the switches, except S_B, are composed of two TGs with counter phase controls to allow the connection to more than a single node.

A single event transient may modify the digital value of the inverter output, thus, affecting the control and the state of a given switch. Depending on the affected switch, an erroneous charge redistribution process may occur. If the voltage under comparison changes its value from positive to negative (or the opposite situation) the comparator output may be flipped, and the control circuit may capture this value, configuring a bit-flip error in the conversion. Since the state of the switches in the subsequent charge redistribution steps depends on the value of the former obtained bits, a single error in one bit may propagate to the remaining of the conversion. This may lead to multiple bit errors in the converted digital word.

In these simulations, the injected SETs were modeled as current sources at the sensitive nodes of the circuit, following the double exponential model [17]. Figure 14.11a shows the simulation result of an injected transient pulse on the output of the control inverter of switch S_3, during the evaluation of bit B_3 (redistribution

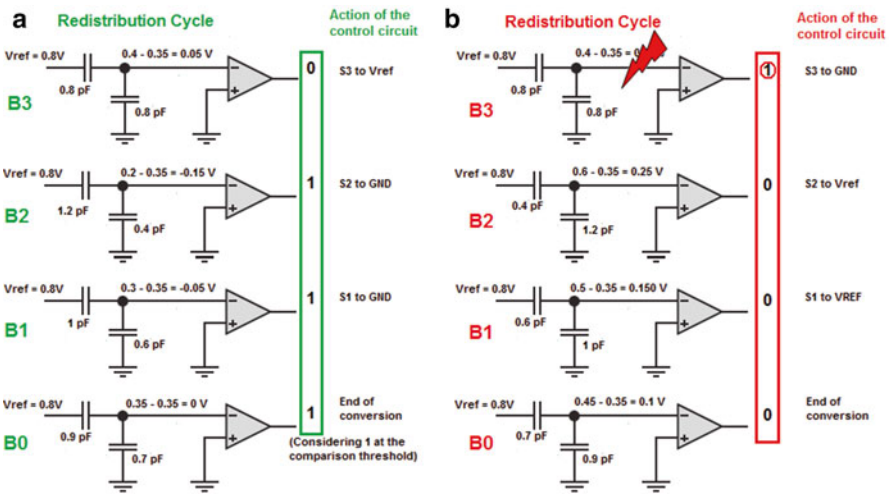


Fig. 14.12 Equivalent circuits during the redistribution cycles in a 4-bit conversion, with $V_{in} = 0.35$ V and $V_{ref} = 0.8$ V: (**a**) normal operation and correct digital value, and (**b**) error in the MSB during the redistribution process is propagated to the remaining of the conversion

mode; with $V_{in} = 0.35$ V and $V_{ref} = 0.8$ V). In this case, the transient current pulse has a peak value of 10 mA and 850 ps width. This SET tends to temporarily and partially disconnect the MSB capacitor from the V_{ref} node, connecting it to ground (partially discharging it). However, after the end of the current pulse, the capacitor is fully reconnected to V_{ref} and the effect on the output comparator disappears. This way, a conversion error will occur only if the digital control circuit captures the comparator output during the SET occurrence.

However, depending on the affected node of the circuit, the effects may be more severe. Figure 14.11b shows the simulation result of a SET injected at the drain of the NMOS transistor of the S_B switch (according to Fig. 14.8), during the redistribution mode. In this mode, switch S_A is off, therefore the drain-bulk junction of the NMOS transistor is reversed biased (in this case, the voltage of the comparator input is positive). Therefore, the current pulse temporarily creates a current path to the ground, discharging the capacitors. In this case, a transient pulse with 0.5 mA peak and 550 ps width was sufficient to discharge the affected node, and flip the comparator output. When the SET effect vanishes and the current path to ground is interrupted, the charge lost in this process is not replaced. Thus, the error at the comparator output remains until the next redistribution step.

The cumulative effect of an error during the charge redistribution process is depicted in Fig. 14.12, in which the equivalent circuit to each redistribution step is shown. In this case, V_{in} is 0.35 V and $V_{ref} = 0.8$ V, and, since the resolution is 4 bits, the LSB voltage is 0.05 V. Therefore, this sample must be converted to 0111 (Fig. 14.12a). However, due to an error in the first redistribution cycle and its propagating effect, the final value of the converted data is 1,000 (Fig. 14.12b).

This cumulative effect may explain the multiple bit errors observed during the experiment (evidenced in Figs. 14.5 and 14.6).

The performed simulations also help to explain another point that was observed in the experimental data: the most of the larger deviations on the converted data occurred near the voltage mid-range of the converter limits. Since at the first redistribution cycle the input voltage is compared to $V_{ref}/2$, samples near this value generate small voltages to be delivered to the comparator input. These low voltages are prone to be easily disturbed by SEE, therefore, increasing the probability of an observable error.

14.5.3 Complementary Digital Designs

Related to the shift registers data, it was observed the occurrence of 9 single events, 6 multiple events and 5 bursts of errors. Once a serial transmission between the DUT and the motherboard takes 880 ns to be completed, these bursts of errors may have two sources. They can be due a SET in the FSM logic that performs the serialization or to errors in the output blocks, e.g., registers and buffers. Figure 14.13 shows three abstraction levels of the implemented shift registers. First, Fig. 14.13a shows the shift register through block diagram. Second, in Fig. 14.13b shows how the synthesis tool implemented each instance of the shift registers. Third, Fig. 14.13c shows the *VersaTile* architecture, which is the basic cell where each block of the Fig. 14.13b is implemented. In this case, through the analysis of the *VersaTile* architecture is possible to note that if a SET occurs in one instance of it, the state of a switch or multiplexer may change, leading to a bit-flip in one of the shift registers.

14.6 Conclusions

We performed a neutron-induced SEE test in the A2F200-FG484 SmatFusion SoC at the ISIS facility located at the CCLRC Rutherford Appleton Laboratory.

A design diversity redundancy scheme was implemented based on a data acquisition system in order to make use of the main components of the SoC and to detect single-events. Results indicate that the system is able to detect functional deviations. Furthermore, a DDR scheme increases the degree of reliability since each redundant module may have a different level of resilience.

Spice simulations considering a charge redistribution ADC were performed. Results allowed us to explain the error mechanisms and the origins of multiple bit errors observed on the experimental data, concerning the analog-to-digital converters.

Current activities of this research are focused on performing, beyond the fault detection, a fault tolerant coverage scheme in the SmartFusion SoC. Other mixed-signal platforms from other manufacturers are also under study to verify the applicability of a DDR scheme in them.

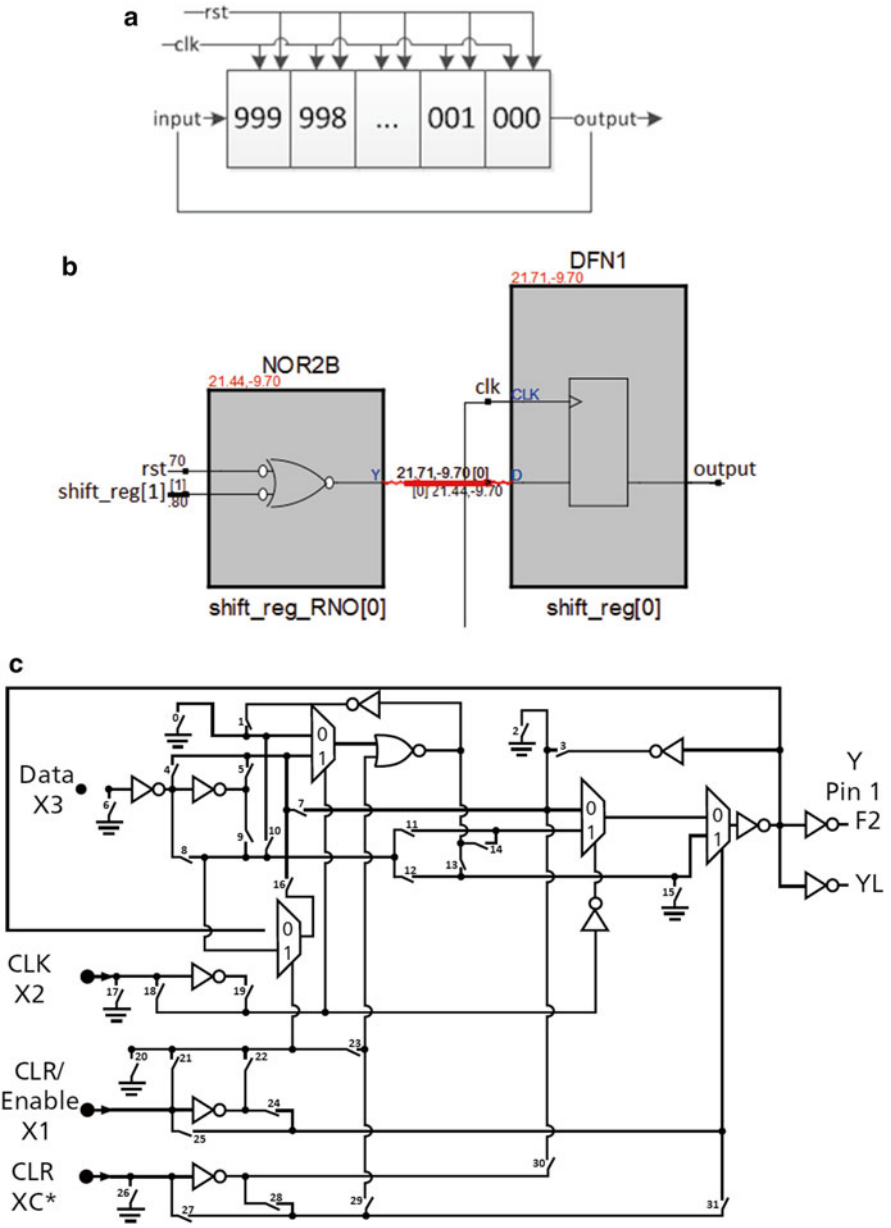


Fig. 14.13 (**a**) Shift register scheme implemented. (**b**) How the tool synthesized each instance of (**a**). (**c**) *VersaTile* architecture

References

1. European Space Agency (2007) System-on-Chip (SoC) development [online]. http://www.esa.int/TEC/Microelectronics
2. Paschalidis NP (2002) Advanced system on a chip microelectronics for spacecraft and science instruments. In: Proceedings of the IEEE Aerospace conference, vol 4, Big Sky, pp 1993–2003
3. Letaw JR, Normand E (1991) Guidelines for predicting single-event upsets in neutron environments. IEEE Trans Nucl Sci 38(6):1500–1506
4. Anghel A, Alexandrescu D, Nicolaidis M (2000) Evaluation of a soft error tolerance technique based on time and or hardware redundancy. In: Proceedings of the IEEE Integrated Circuits and Systems Design, Manaus, pp 237–242
5. Avizienis A, Kelly JPJ (1984) Fault tolerance by design diversity: concepts and experiments. IEEE Comput 17(8):67–80
6. Xilinx (2012) Zynq-7000 extensible processing platform [online]. http://www.xilinx.com/products/silicon-devices/epp/zynq-7000
7. Altera (2012) Cyclone V FPGAs [online]. http://www.altera.com/devices/fpga/cyclone-v-fpgas
8. Actel (2011) SmartFusion customizable system-on-chip [online]. http://www.actel.com/documents/SmartFusion_DS.pdf
9. Morgan KS, McMurtrey DL, Pratt BH, Wirthlin MJ (2007) A comparison of TMR with alternative fault tolerant design techniques for FPGAs. IEEE Trans Nucl Sci 54(6):2065–2072
10. Hiari O, Sadeh W, Rawashdeh O (2012) Towards single-chip diversity TMR for automotive applications. In: Proceedings of the IEEE international conference on electro/information technology, Indianapolis, pp 1–6
11. Borges GM, Gonçalves LF, Balen TR, Lubaszewski MS (2010) Diversity TMR: proof of concept in a mixed-signal case. In: Proceedings of the IEEE Latin American Test Workshop, Pule del Este, pp 1–6
12. Violante M, Sterpone L, Manuzzato A, Gerardin S, Rech P, Bagatin M, Paccagnella A, Andreani C, Gorini G, Pietropaolo A, Cardarilli G, Pontarelli S, Frost C (2007) A new hardware/software platform and a new 1/E neutron source for soft error studies: testing FPGAs at the ISIS facility. IEEE Trans Nucl Sci 54(4):1184–1189
13. Cortes FP, Carro L, Girardi A, Suzim A (2002) A A/D converter insensitive to SEU effects. In: Proceedings of the 8th international On-Line Testing Workshop, Porto Alegre, pp 89–93
14. Balen TR, Cardoso GS, Gonçalez OL, Lubaszewski MS (2011) Investigating the effects of transient faults in programmable capacitor arrays. In: Proceedings of the 12th Latin American Test Workshop, pp 1–6
15. Nanoscale Integration and Modeling Group (2012) 130nm BSIM3 model card for bulk CMOS [online]. http://ptm.asu.edu. Accessed Sept 2012
16. Jespers PGA (2001) Integrated converters. Oxford University Press, Oxford
17. Messenger GC (1982) Collection of charge on junction nodes from ion tracks. IEEE Trans Nucl Sci 29(6):2024–2031

Part V
Embedded Processors in System-on-Chips

Chapter 15
Mitigating Soft Errors in Processors Cores Embedded in System-on Programmable-Chips

Stefano Esposito and Massimo Violante

Abstract Newer generations of Field Programmable Gate Arrays (FPGAs) embed advanced intellectual property (IP) cores, such as fast digital signal processors (DSPs), memory blocks, and processors, which are implemented in dedicated parts of the silicon, without consuming reconfigurable fabric that is left available for system designers. The new class of devices combining firm computing cores along with programmable fabric is often referred to as system-on-programmable-chip (SoPC). Several application domains, like industrial control and automotive, where computing intensive algorithms have to be performed in real-time by embedded processors, already recognized the benefit of SoPCs. Space application domain may benefit as well from SoPCs, provided that the problems specific to such application domain are solved. In particular, being the SoPC devised for ground-based applications, the consequences of the interaction of ionizing radiations with SoPC silicon, triggering effects like Total Ionizing Dose (TID) or Single Event Effects (SEE), are of particular interest for designers willing to deploy SoPC in space. This chapter first summarizes the effects of radiation in SoPC with particular emphasis on SEE in the processor cores the device embeds. Then, it reports an overview of the techniques to cope with them, looking in particular to Software Implementer Fault Tolerance (SIFT) techniques. Finally, a novel architecture is proposed.

15.1 Introduction

Newer generations of Field Programmable Gate Arrays (FPGAs) embed advanced intellectual property (IP) cores, such as fast digital signal processors (DSPs), memory blocks, and processors. The IPs are implemented in dedicated portions of silicon (they are firm IP cores), and do not consume resources belonging to the configurable fabric. Developers of embedded applications can therefore exploit the computing capabilities of FPGAs configurable fabric to implement custom interfaces and/or dedicated hardware accelerators in combination with the versatility of

S. Esposito (✉) • M. Violante
DAUIN—Politecnico di Torino, C.so Castelfidardo, 29, Torino 10129, Italy
e-mail: stefano.esposito@polito.it; massimo.violante@polito.it

F. Kastensmidt, P. Rech (eds.), *FPGAs and Parallel Architectures for Aerospace Applications*, DOI 10.1007/978-3-319-14352-1_15

processors, all integrated in a single device known as system-on-programmable-chip (SoPC) [1].

Several application domains already recognized the benefit of SoPCs, such as industrial control applications, and automotive applications where computing intensive algorithms (like for example image processing to recognize the features of an object that should be manipulated by an handler, or to identify obstacles on the path of a vehicle) have to be performed in real-time by embedded processors, which are often connected to custom devices through suitably-designed hardware components.

Space application domain may benefit as well from SoPCs, provided that the problems specific to such application domain are solved adequately. Space applications are deployed in radioactive environment where ionizing radiation interacts with the silicon provoking a number of effects, such as total ionizing dose effects (TID) and single event effects (SEEs) [2]. The IP cores SoPCs embed are typically designed for ground applications where natural radiation is negligible hence they do not include specific mechanisms to cope with radiation effects; as a result, IP-core behavior can be affected significantly when deployed in space.

As far as embedded processors are considered, SEEs could be categorized as follows (for the sake of this chapter TID effects and destructive events such as latch-up are not taken into account):

- *Persistent effect*: SEEs alter the behavior of the processor core in such a way that it no longer provides its service, i.e., it is no longer able to run software. This effect, also known as Single Event Functional Interruption (SEFI), is provoked by radiation hitting the control circuitry of the processor, such as the phased-locked-loops (PLLs) responsible for clocking the core. This effect is persistent as the processor is not able to recover its expected functionality autonomously, and an external intervention is needed (e.g., reset or power cycle).
- *Transient effect*: SEEs alter the content of storage elements (either due to single event upsets, SEUs, in memory elements such as registers, cache lines, or RAM cells, or due to the propagation of SETs that are latched by memory elements) that either store the data the processor manipulates (*data error*) or that control the order in which the instructions of the program are executed (*control-flow error*). These effects are transient as they can be removed by the processor autonomously during the execution of the program (e.g., an SEU in a variable is removed as soon as a new correct value is loaded in the variable), or can be detected and removed by running a suitable recovery action implemented by software.

Given the above effects, suitable countermeasures are needed to deploy successfully SoPCs in space applications, depending on the mission requirements, and the characteristics of the radioactive environment the mission aims at. In this chapter we analyze Software Implemented Fault Tolerance (SIFT) solutions: after stating the assumptions we use in the chapter, we discuss a behavioral model for SEE in processor cores to set the basis for understanding the SIFT techniques available in literature, which we summarize shortly. We then present a possible architecture to cope with SEE in the processor core SoPCs embed, along with the description of a use case. Finally, we draw some conclusions.

15.2 Assumptions

In this chapter we focus on SEE affecting the processor cores SoPCs embed, while SEEs affecting the SoPC fabric are out of the scope of this chapter, as well as TID effects and destructive effects.

Moreover, we assume that the system we are designing entails two computers:

- A platform computer responsible for supervising the operations of the space application (e.g., management of telemetry and remote control communications), and for implementing the predefined recovery action when a payload computer signals that an error is detected. This computer is assumed to be implemented resorting to traditional space-grade components, which do not require any of the techniques presented in this chapter.
- A payload computer responsible for running the actual application task, which is implemented using a SoPC and that requires the hardening techniques presented in this chapter. Cold stand-by redundancy is assumed for the payload computer: one instance of the payload is powered and serves as primary payload, the second instance is powered-off and serves as redundant payload. The application program the payload implements is organized in three phases: *acquisition phase*, during which the data to be processed are acquired through suitable input channel, *processing phase*, during which the data are transformed according to an algorithm, and the *presentation phase* during which the computed results are committed through a suitable output channel.

15.3 A Behavioral Fault Model for SEE in Processor Cores

To understand the concepts at the base of SIFT techniques it is worth analyzing the effects of SEEs in a processor core by looking at their effects on the processor behavior.

As far as persistent effects are considered, such as SEFI, as the processor core is no longer able to execute software, it appears as unresponsive from the user point of view. As a result, the processor behaves as it entered an endless loop.

As far as transient effects are considered, let's analyze the information the processor handles as suggested in [3]. By looking at this information, we can identify the following behavioral error models:

- *Data error*: it is defined as a logical error affecting the program data stored in the processor core. Please note that this definition does not consider the location where the data are actually stored: they may be stored either in the processor data cache, or in its register file.
- *Code error*: it is defined as a logical error affecting one instruction of the program's code. The erroneous instruction may either be in the processor instruction cache, or in the processor pipeline. Two types of code error can be defined.

```
%% Error-free code

      MOV R0, 10
      MOV R1, 1
LOOP: ADD R1, R1
      SUB R0, 1
      BNZ LOOP
```

```
%% Erroneous code

      MOV R0, 10
      MOV R1, 1
LOOP: SUB R1, R1
      SUB R0, 1
      BNZ LOOP
```

Fig. 15.1 Code error of type 1. An ADD instruction is modified in a SUB instruction

```
%% Error-free code

      MOV R0, 10
      MOV R1, 1
LOOP: ADD R1, R1
      SUB R0, 1
      BNZ LOOP
```

```
%% Erroneous code

      MOV R0, 10
      MOV R1, 1
LOOP: ADD R1, [R1]
      SUB R0, 1
      BNZ LOOP
```

Fig. 15.2 Code error of type 1. The addressing mode of an ADD instruction is modified

```
%% Error-free code

      MOV R0, 10
      MOV R1, 1
LOOP: ADD R1, R1
      SUB R0, 1
      BNZ LOOP
```

```
%% Erroneous code

      MOV R0, 10
      MOV R1, 1
LOOP: ADD R1, R1
      SUB R0, 1
      BNZ elsewhere
```

Fig. 15.3 Code error of type 2. The target address of a branch is changed

```
%% Error-free code

      MOV R0, 10
      MOV R1, 1
LOOP: ADD R1, R1
      SUB R0, 1
      BNZ LOOP
```

```
%% Erroneous code

      MOV R0, 10
      MOV R1, 1
LOOP: ADD R1, R1
      SUB R0, 1
      BZ LOOP
```

Fig. 15.4 Code error of type 2. The branch condition of a conditional branch is changed

- *Type 1*: it is defined as a code error that modifies the operation the instruction executes, not affecting the execution flow. Examples of this error model are reported in Figs. 15.1 and 15.2.
- *Type 2*: it is defined as a code error that modifies the expected program execution flow. Examples of this error models are reported in Figs. 15.3 and 15.4.

15.4 Error Detection Techniques

As far as persistent errors are considered, they can be detected resorting to monitoring facilities that are independent from the processor core, which constantly monitors the processor to identify whether it becomes unresponsive. For an extensive reference of these techniques the reader should refer to [4].

As far data and code errors are considered, Software Implemented Fault Tolerance techniques can be used when hardware redundancy, entailing processor duplication/triplication, is not applicable. This is typically the case of SoPCs where the processor core is embedded in the device and it cannot be replicated, unless the entire SoPC is replicated.

SIFT techniques are a set of methods all having in common the goal of hardening a processor against errors by modifying its software. SIFT techniques are subdivided into two main categories:

- *Data hardening techniques* are a set of methods sharing the basic idea of protecting the data by duplicating both data and computations. Data handling techniques are intended to cope with data errors and with code errors of type 1.
- *Control flow check techniques* are a set of methods sharing the goal of protecting the processor against code errors of type 2, also known as *Control Flow Errors* (CFE).

15.4.1 Data Hardening Techniques

The duplication of computation introduced by these methods can be described at different granularities:

- *Instruction level* duplication [5–7] entails the duplication of single instruction. It can be implemented either at assembly level or at high level. Both solutions usually employ special compilers to facilitate implementation. The basic scheme is given in Fig. 15.5: both data and computations are duplicated and an error is detected by comparing the replicas each time a read operation is performed. The scheme can also applies to functions, in which case the prototype must be modified to allow replication of inputs and outputs. The main advantage of assembly level instruction duplication is the ability to exploit instruction-level parallelism implemented in modern architectures, but it also introduces code size and memory occupation overheads higher than high level instruction duplication [8]. Moreover, the latter can exploit special considerations like those used by the authors of [9], allowing to reduce the number of instructions actually duplicated.
- *Procedure level* duplication basic idea is to duplicate call to a procedure rather than each instruction in a program. In this approach, data to and from the procedure are duplicated and an error is detected by comparing the replicas after each read operation. Authors of [10] propose the *Selective Procedure Call*

```
a = b+c;
```

```
a0 = b0+c0;
a1 = b1+c1;
if (b0 != b1 || c0 != c1)
        error();
```

Fig. 15.5 High level instruction duplication

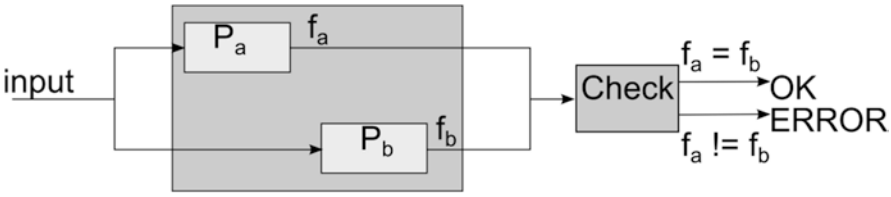


Fig. 15.6 VDS block diagram

Duplication technique, in which some procedures are modified to implement instruction duplication, while others are left unmodified. In this approach:

- A procedure with duplicated instructions can detect error, a procedure without duplicated instructions must be called twice to detect errors.
- A procedure without duplicated instructions cannot call a procedure with duplicated instructions.
- If a global variable is used in a procedure without duplicated instructions, the global variable must be duplicated and a new version of the procedure must be added which uses the duplicated global variable. This is in order to avoid errors due to access to a global variable in two subsequent calls to the same procedure.

- *Program level* duplication basic idea is to duplicate the whole computation. Transient errors are detected by temporal redundancy, as for instance in a *Virtual Duplex System* (VDS) scheme, represented in Fig. 15.6. The main drawback of this solution is the overhead due to the repetition of tasks, but this overhead can be mitigated by techniques exploiting multithreading [11] or multicore processors. In such technique two computations are executed in parallel, thus exploiting spatial redundancy rather than time redundancy. VDS can also benefit from *Design Diversity* techniques as proposed in [12, 13], both manual and automatic.

15.4.2 Control Flow Check Techniques

In order to describe CFC techniques some important concepts must be introduced. The first of such concepts is the *Basic Block* (BB). A BB is defined as a sequence of instruction with one entry point and one exit point, meaning that no instruction of a

BB, except the first, can be the target of a jump instruction and no instruction of a BB, except the last, can be a jump instruction. A program is composed of several BBs and can be described by a *Control Flow Graph* (CFG). A CFG is an oriented graph $P = \{V, B\}$ composed of a set of BBs $V = \{v_1, v_2, \ldots, v_n\}$ and a set of branches $B = \{b_{i1,j1}, b_{i2,j2}, \ldots, b_{im,jm}\}$ connecting nodes as they are executed. For each BB v_i are defined a set of predecessors $pred(v_i)$ and a set of successors $succ(v_i)$ as:

$$v_k \in pred(v_i) \Leftrightarrow b_{ki} \in B$$
$$v_j \in succ(v_i) \Leftrightarrow b_{ij} \in B$$

Once the CFG of a program has been defined, executed branches can be classified as:

- *Legal* if the branch is in the CFG.
- *Wrong* if the branch is in the CFG but is taken unexpectedly.
- *Illegal* if the branch is not in the CFG.

Based on this classification, CFE can be classified in five types as suggested in [3]:

- *Type 1* a wrong branch.
- *Type 2* an illegal branch from the last instruction of a BB v_i to the first instruction of another BB v_j not included in $succ(v_i)$.
- *Type 3* an illegal branch from the last instruction of a BB to any instruction, except the first, of any other BB.
- *Type 4* an illegal branch from any instruction of a BB, except the last, to any instruction of any other BB.
- *Type 5* an illegal branch from any instruction of a BB to any instruction of the same BB.

Many approaches have been proposed to address CFE detection. The basic idea shared by all such techniques is that some check is added in order to grant that the control flow executed up to the check is the correct one. This is achieved in different ways, here are presented some of the main techniques.

Path Identification

This solution was proposed in [14]. The CFG is partitioned in loop-free intervals and to each interval is associated a table which in turn associates to the identifier of every legal path entering the loop-free interval *CIID* a path predicate, i.e. a Boolean predicate that must evaluate true at the beginning of the loop-free interval, and a next loop-free interval identifier *NIID*. Checks are performed at loop-free interval level.

At the beginning of a loop-free interval the current path identifier *RPI* is used to retrieve the correct row from the table. If *RPI* is not found in the table an error is detected, else the path predicate is evaluated and an error is detected if it evaluates to false. If the path predicate evaluates to true, *NIID* is compared to the id of the current loop-free interval and an error is detected in case of mismatch. At the end of

described checks, *RPI* is initialized to 1 and it is then updated at the beginning of each BB by multiplying it by the prime node identifier associated to the BB.

This technique is able to detect type 1, 2 and 3 CFEs, but it has a significant memory-overhead (123.6 % in average) and a non-negligible performance overhead (between 69.6 and 87 % in average). Moreover, detection at loop-free interval level introduces error latency.

ECCA

The *Enhanced Control flow Checking using Assertion* is an approach using assertion in order to detect CFEs [15]. Two version of ECCA have been proposed, either at high level, *ECCA-HL*, or at intermediate level *ECCA-IL*. While the first modifies high level sources of the program, the second modifies an intermediate level, called *RTL*, used by the GCC and exploits some characteristics of this level in order to mitigate the performance overhead introduced by ECCA-HL.

Both versions perform a partitioning of the CFG, identifying some blocks, i.e. sequences of BBs with one entry and one exit, and both versions use two assertions called *SET* and *TEST*. Both versions assign to each block a *Block identifier BID*. ECCA-HL uses the assertions to modify the value of a variable *id*, while ECCA-IL works on two registers, r_1 and r_2. In both versions, the assertions are designed so that if a CFE occurs, the SET assertion causes a *divide-by-zero* exception to be triggered by the execution unit of the CPU, thus detecting the error. The SET and TEST assertion for ECCA-HL are reported in Eq. 15.1, the SET assertions for ECCA-IL are in Eq. 15.2 and the TEST assertions for ECCA-IL are in Eq. 15.3.

$$\begin{aligned} id &= \frac{BID}{\overline{(id \bmod BID)} \times (id \bmod 2)} \\ id &= NEXT + \overline{(id - BID)} \end{aligned} \tag{15.1}$$

$$\begin{aligned} r_1 &= (r_1 - BID) \times (r_2 - BID) \\ r_1 &= \frac{BID}{(\frac{r_1 + 1}{2r_1 + 1})} \end{aligned} \tag{15.2}$$

$$\begin{aligned} r_1 &= (r_1 - BID) \times NEXT_1 \\ r_2 &= (r_1 - BID) \times NEXT_2 \end{aligned} \tag{15.3}$$

YACCA

The *Yet Another Control flow Check using Assertion* solution uses assertions to check for the correctness of the current control flow [16, 17]. To each BB v_i are assigned two identifiers $I1_i$ associated with the beginning of the BB and $I2_i$

associated with its end. A *code* variable is updated through a *TEST* assertion at the beginning of each BB so that after the assertion *code* is equal to $I1_i$. At the end of the same BB, a *SET* assertion modifies again the *code* variable so that after the assertion, *code* is equal to $I2_i$. The TEST assertion at the beginning of a BB v_i verifies that *code* is equal to the $I2_j$ of a BB $v_j \in pred(v_i)$. The SET assertion at the end of a BB v_i verifies that the *code* variable is equal to $I1_i$. In both assertions, the *code* variable is updated as follows

$$code = (code \,\&\, M1) \oplus M2$$

M1 and *M2* are both constants computed at compile time. *M1* depends on $pred(v_i)$, while *M2* depends on both the expected value for *code* and $pred(v_i)$. Their definition is different for the TEST and SET assertion, and are both reported in the following equations, first for the TEST assertion, then for the SET assertion

$$M1 = \overline{\left(\underset{j:v_j \in pred(v_i)}{\&} I2_j\right) \oplus \left(\underset{j:v_j \in pred(v_i)}{V} I2_j\right)}$$

$$M2 = (I2_j \,\&\, M1) \oplus I1_i$$

$$M1 = 1$$

$$M2 = I1_i \oplus I2_i$$

YACCA is able to detect all faults of type 1,2,3 and 4. It also has a performance overhead lower than ECCA-HL, since its assertions do not use multiplications and divisions. The drawback is the addition of conditional branches that might be target of CFE. This can be avoided by moving the check at the end of the program, at the cost of introducing some error latency.

15.4.3 Fault Tolerance

The techniques discussed so far are only capable of detecting a fault. To actually implement fault tolerance in the system, some other measure must be taken. Main techniques all share a basic idea that is *Design Diversity* [18].

Design diversity implies the development of two or more versions of the same program in such a way that they cannot incur in common mode faults. Design diversity usually prescribe that the different versions of the program are also designed differently, for instance using different algorithms to perform the same task and implemented differently, possibly by different programmers using different methodologies and different compilers. The different versions produced are called *variants*. In the following, some of the main techniques exploiting design diversity are briefly described.

- *N-version programming* [19, 20]. In this approach, several (*N*) variants of a program are produced and run in parallel in the system. At fixed points in execution,

each version saves a state vector, called *c-vector*, and a decider compares all the c-vectors looking for a consensus. This method has been used in several applications [21–26].

- *Temporal Redundancy*. The main approach exploiting the concept of temporal redundancy is the *Virtual Duplex System*, already mentioned and represented in Fig. 15.6. In [11] a kind of VDS is used in which two versions are used to detect an error and a third version is used to recover.
- *Recovery Block* [20]. Several variants are produced, but at any given time only one is running in the system. A *decider* is in charge of performing an acceptance test on the outputs of the active variant. If the acceptance test is failed, the decider selects one of the alternate and executes it starting from a safe state previously saved from the active variant. Many methods exist to design the acceptance test [27].
- *Algorithm Based Fault Tolerance* [28, 29]. This method uses mathematical proprieties of the algorithm implemented in the system to perform both detection and recovery. The first method proposed was used to implement fault tolerant matrices operations and then a method was introduced to implement fault tolerant FFT.

15.4.4 Hybrid Methods

Besides the methods described so far in this section, which are purely software, several methods have been proposed to implement fault tolerance through a cooperation of hardware and software. This is achieved chiefly by adding special purpose hardware to the system called a *watchdog* [30]. Watchdogs are used to control the system behavior and to detect error situations. There are several kinds of watchdogs, the simpler ones are essentially timers triggering an interrupt when the CPU fails to reset them or if the watchdog does not perceive any activity on the system bus within a given timeout. The system can then recover through the interrupt service routine associated to the watchdog interrupt. More complex watchdogs are properly called *watchdog processors* and can implement several techniques. These are used to implement CFC techniques via an external hardware, thus relieving the CPU from the task of checking for CFEs. Several techniques have been proposed using watchdogs timers or watchdog processors using assertions or memory access checks or signatures [4, 31–33].

15.5 Dealing with SEE in Processors Cores in SoPCs

We propose a reference architecture targeting both the persistent and the transient effects provoked by SEEs, and in particular:

- *Persistent effects*: As persistent effects (e.g., SEFIs) inhibit the capability of processor cores to run programs, a *watchdog* [30] is needed that sits beside the

processor and that is capable to operate autonomously. The watchdog is responsible to recognize that the processor no longer executes the program, and to initiate the predefined recovery action. Moreover, the watchdog implements advanced features to support the detection of data and control-flow errors, as detailed in the following section.

- *Transient effects*: redundancy is proposed to deal with transient effects, and in particular:
  - Instruction redundancy is employed to detect data errors by replicating each computation twice and comparing the two results. In case of mismatch indicating a transient error is detected, the predefined recovery action is initiated. In our architecture we propose to adopt program-level redundancy, as detailed in the following section. As far as error detection is concerned, the watchdog includes a memory comparison feature that is responsible for comparing the results produced by the redundant execution and for signaling mismatches to the platform computer.
  - Control-flow checking is employed to identify whether SEEs corrupted the expected sequence of instructions composing the program. Instructions are inserted in the program to communicate with the watchdog; each communication instruction transmits to the watchdog a pre-computed keyword that is function of the location of the instruction in the program control-flow graph [34]. The watchdog checks whether the expected sequence of keywords defined on the basis of the program control-flow graph is received; in case an unknown sequence or an out-of-sequence keyword is received, the predefined recovery action is initiated.

The proposed architecture for the SoPC-based payload computer is schematized in Fig. 15.7. The payload computer is composed of a SoPC and an off-chip memory (I/O interfaces that may be needed for specific applications are not shown here for the sake of simplicity). The SoPC embeds one processor IP core including its own memory (e.g., the L1/L2 cache memory, if any, and possibly SRAM memory for small-footprint applications), and the configurable fabric, where some resources are used to implement the watchdog. The interface between the processor core and the watchdog is a low-speed bus such as the AMBA peripheral bus (APB) or general purpose I/O (GPIO). Although this appears a limitation to the monitoring capabilities of the watchdog, it must be underlined that in SoPCs the interface between the processor and its memory is seldom accessible. Therefore, it is not possible to directly observe the processor bus as proposed for example in [35].

The watchdog features a second interface that connects it directly to the external memory. This interface is exploited to access the two copies of the computed results, to provide a hardware-implemented consistency check of results. Upon error detection, the watchdog signals the platform computer the need for initiating the predefined recovery action through the Error Detection Interrupt line.

In our architecture the payload computer is designed adopting cold stand-by sparing to mitigate the risk of unavailability due to permanent failures. The platform

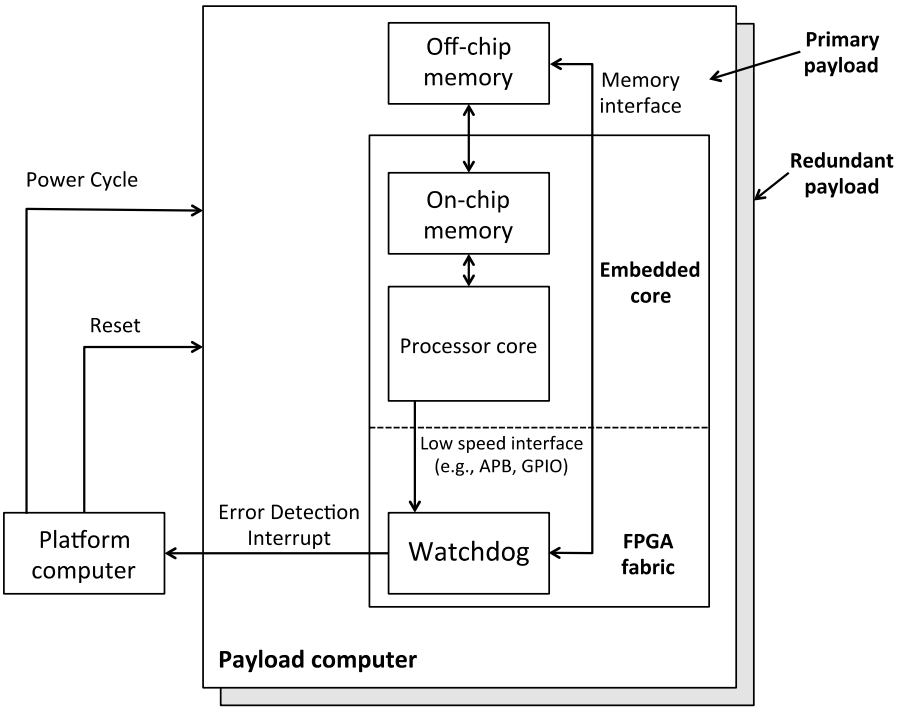


Fig. 15.7 Reference architecture

computer manages the switch between primary payload and the redundant payload according to the Finite State Machine shown in Fig. 15.8, which illustrates the predefined recovery action. The payload computer can be in one of three possible states:

- *Healthy* state where the primary payload is powered on and the redundant is powered off;
- *Recovery* state where the primary payload is off and the redundant is on
- *Faulty* where both payloads are powered off.

The system enters initially in the healthy state; each time a fault is detected, the platform computer resets the payload. When the number of detected faults exceeds a given threshold, the platform computer switches to the recovery state, powering off the primary payload and powering up the redundant payload. In the recovery state each detected fault leads to a payload reset, until a given threshold is reached, which forces the system entering in the faulty state where both payloads are powered off.

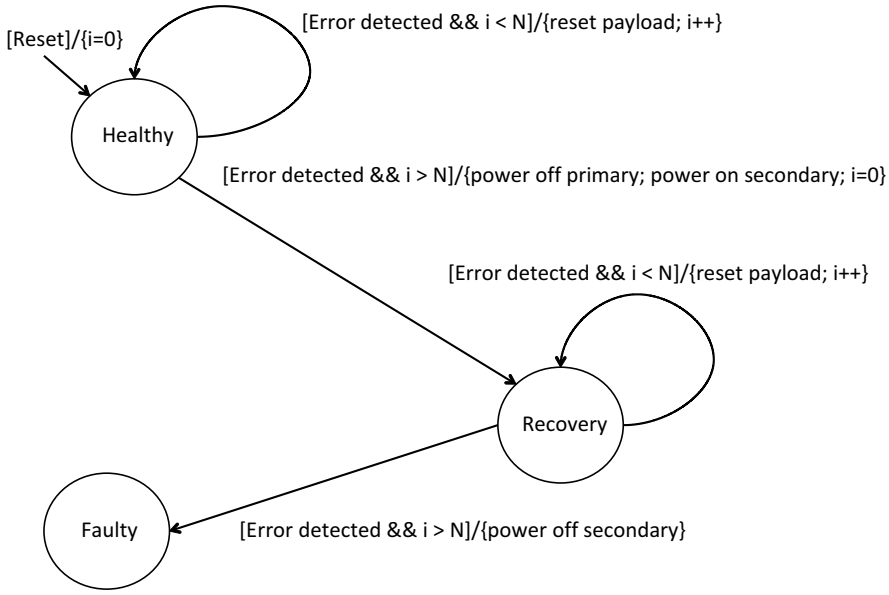


Fig. 15.8 Recovery action concept

15.5.1 Watchdog Design for the SoPC

The watchdog our architecture adopts is implemented in the SoPC configurable fabric, and its components are depicted in Fig. 15.9:

- *Memory comparison feature*: its purpose is to access the off-chip memory storing the results of the program the processor core executes to compare the outputs of the two instances of the program. As program-level redundancy is exploited, two identical instances of the program will be executed, producing two copies of the output results. Upon completion of the two executions, the watchdog is triggered to read from memory the two copies of the outputs and compare them. In case of mismatch the Error Detection Interrupt is generated. To minimize the duration of the memory comparison, the two program replicas compute two 32-bit signatures of their respective output results, which are compared through the memory comparison feature.
- *Processor interface*: its purpose is to establish the communication between the processor core and the watchdog, to receive the keywords used to trace the control flow. The interface is composed of two channels, one for each replica of the program obtained according to the program-level duplication.

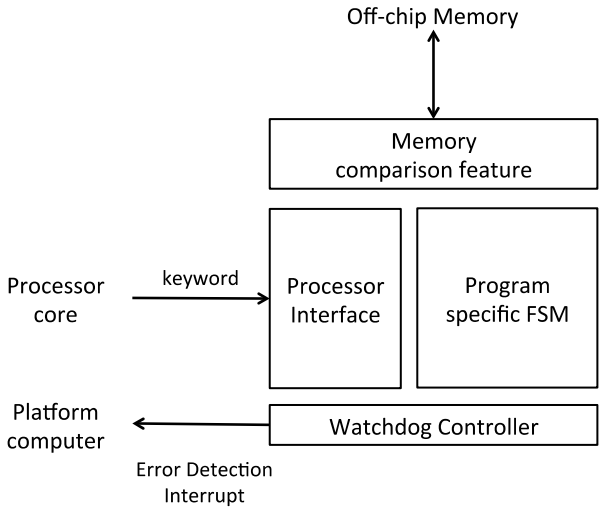


Fig. 15.9 Architecture of the watchdog for SoPC

- *Watchdog controller*: its purpose is to orchestrate all the operations of the watchdog components, and in particular:
  - It monitors the evolution of the Program Specific FSM as detailed in the following.
  - Upon detecting the program completion, it triggers the Memory comparison feature.
  - It keeps a counter to measure the time since the last reception of a keyword from a program replica. In case a predefined threshold is reached the Error Detection Interrupt is triggered.
- *Program specific FSM*: its purpose is to check the coherency of the keyword sequence with the expected program execution flow.

All the components of the watchdog, but the Program specific FSM, are program-independent, so they have to be designed once when the interfaces with the processor and the memory are selected. Conversely, the Program specific FSM is synthesized ad-hoc, starting from the definition of the expected sequence of keywords to be received from each program replica. In our current implementation each keyword is an 8-bit unique identifier K_i defined by the programmer that is responsible to partition the program in chunks, to assign a keyword to each program chunk, and to place an instruction at the beginning of each program chunk to send the associated keyword to the watchdog. During program execution, each program replica sends to the watchdog a sequence $\{K_0, K_1, \ldots, K_n\}$ of keywords according to the programmer decisions, being K_n the keyword indicating the program execution is completed. The Program specific FSM implements a finite state machine

whose states space is {*Reset*, K_0, K_1, …., K_n}, and where each state transition happens either from *Reset* state to state K_0, or from state K_i to $K_{(i+1)\ \%\ (n+1)}$. Each time a new keyword K_i^j is received from program replica *j*, the following operations are performed:

- The Program specific FSM associated to the program replica makes a state transition reaching state K_e^j;
- The Watchdog controllers checks whether $K_i^j = K_e^j$. In case of mismatch the Error Detection Interrupt is activated.

15.5.2 Program-Level Duplication for the SoPC

The basic idea of program-level duplication is to execute twice the program the computer implements, and to vote among the produced results. Transient SEEs affecting the computer can be detected provided that:

- The program execution has no side effect on the input data it processes, so that it can be repeated obtaining the same result.
- Each execution instance is independent from the other, i.e., the operations performed by each instance cannot interfere with the outcomes of the other instance.

To fulfill the above requirements, the programmer could exploit the memory protections features the adopted processor provides, which consists in either a hardware memory protection unit, or a hardware memory management unit. Although viable, this solution has some drawbacks:

- It lacks portability: being the protection mechanism processor-dependent, porting the same application to different platforms may require substantial rework.
- It is error-prone: being the programmer responsible for memory protection, an intensive validation is required to guarantee that the program is free of bugs introduced when hand-coding the protection scheme.

To overcome the above drawback, a different solution can be exploited, which resorts to a software layer, called hardware abstraction layer (HAL), between the application software and the hardware, which provides memory protection services. If the HAL exposes a well-define programming interface that the software exploits, the application becomes hardware-independent, increasing its portability and reducing the development/validation costs:

- The HAL needs to be implemented and validated once for a given hardware architecture. Then it can be reused as is, without the need for a new validation at each new project, thus saving development/validation costs.
- When the application software has to be ported on new hardware architectures, only the HAL must be adapted, thus minimizing development costs.

In our architecture we assume the availability of a HAL that provides the following services:

- *Memory partitioning*. Each program instance is assigned to a dedicated memory area. The memory areas assigned to the two instances are not overlapping. When a program instance is executed, any attempt to access to memory area different from the assigned one results in an Error Detection Interrupt.
- *Resource partitioning*. Each program instance is assigned to a dedicated I/O area. The I/O areas assigned to the two instances are not overlapping. When a program instance is executed, any attempt to access to I/O area different from the assigned one results in an Error Detection Interrupt.
- *Time partitioning*. The processor time is divided in time slots, and each program instance is assigned to a set of time slots. Each program instance is preempted from the processor at the end of its time slot, thus guaranteeing to each program instance a fair access policy to the processor.

Being the HAL a piece of software running on the same processor that runs the application software, its execution can be affected by SEEs. However, due to the nature of operations the HAL performs we can expect that any SEEs will lead to effects detected by the watchdog our architecture exploits, an in particular:

- In case the SEE affects the memory-partitioning scheme by altering the configuration of the memory protection unit or the memory management unit, we expect a misalignment of the output memory areas produced by the two instances, leading to a error detection by means of the memory comparison feature the watchdog provides.
- In case the SEE affects the resource-partitioning scheme, we expect that an incorrect sequence of keyword is sent to the watchdog, resulting in error detection.
- In case the SEE affects the time partitioning, we expect that either one instance is not timely scheduled, leading to a timeout in the watchdog, or incorrectly scheduled, leading to a wrong keyword sequence issued to the watchdog, leading again to error detection.

15.6 A Use Case

To assess the proposed architecture, we considered a use case where a Zynq SoPC device is used to build a payload computer for handling data coming from a camera that captures 1024×1024 8-bit per pixel images. The application the payload computer executes is a lossless compression software based on an algorithm developed at the European Space Agency: the RICE compressor [36].

RICE compressor exploits the Rice coding that derives from Golomb codes. In Golomb codes, a set of data is encoded by optimally finding a divisor for the set, performing the division and encoding quotient and remainder of such division.

The quotient is encoded in a particular way called unary encoding, while the remainder is usually encoded in binary. In unary, a number is encoded as a sequence of equal symbols, usually delimited by a different symbol. Rice codes are a subset of Golomb codes where the divider is constrained to be a power of two. Even though this limits the efficiency of the coding, since the divider can be sub-optimal, the coding procedure requires less computational effort, since divisions by powers of two can be easily implemented by shift operations. However, unary encoding is only convenient for compression of an image if the number of bits it takes is less than the number of bits one pixel is normally encoded on, plus the number of bits needed to encode the remainder. For instance, if the original image is represented with 8 bit per pixel, encoding 10 in unary would result in an overhead of 3 bits instead of a reduction. The problem of avoiding a coding overhead that would result in an encoded image bigger than the original one is solved in the RICE compress by encoding the entropy of the image to be compressed. A measure of such entropy is derived using a prediction scheme. The RICE algorithm exploits a static prediction scheme where the value of a pixel is predicted to be the same as the one of the pixel before in a sequential scan of the image. The algorithm works as follow:

1. The image is subdivided in blocks of fixed length.
2. The values in the block are checked:
   - If the values are all zero, the block is encoded in a special way to reduce size. See [36] for details.
   - Otherwise the predictions for each pixel in the block is computed, the optimal Rice divisor for the block is computed, and the block is encoded using Rice coding with the identified divisor

We initially developed an implementation of the RICE compression algorithm in C code for the Zynq SoPC, which account for 399 lines of C code. When compiled for the Cortex A9 processor embedded in the Zynq SoPC, the application occupies about 2 Mbytes for the input and output data buffers, and about 14 Kbytes for the binary code.

We then implemented the proposed architecture adopting the PikeOS real-time operating system as HAL, which satisfies the requirements we stated in the previous section, and also guarantees a minimal area/performance footprint.

Using PikeOS, we developed a software architecture containing three partitions (a partition identifies one program and its memory, resource and timing partitioning definition):

- RICE instance 0, it is the first instance of the RICE compressor, which is assigned to the core 0 of the Cortex A9 the Zynq includes.
- RICE instance 1, it is the second instance of the RICE compressor, which is assigned to the core 1 of the Cortex A9 the Zynq includes.
- Coordinator, it is in charge or enabling the execution of the two instances of the RICE program, and to communicate via the serial debug console the result of the memory comparison when both RICE instances completed their task.

The RICE has been subdivided in 16 chunks, and the watchdog as been synthesized to receive 16 couples of 8-bit keywords. The code of the two RICE instances has been enriched with the instructions to communicate with the watchdog. As far as the watchdog is concerned, its current implementation takes less than 2 % of the Zynq device we used (Zynq 7Z020).

As expected, the proposed software architecture introduces some overhead with respect to the initial implementation, and in particular:

- As far as the memory occupation is considered, the data memory is increased to about 4 Mbytes as two instances of RICE are used. The code memory is increased to about 35 Kbytes as two instances of the RICE code are placed in memory along with the coordinator partitions and the PikeOS run-time software (scheduler, memory manager, I/O manager, and inter-process communication manager).
- As far as the execution time is considered, as we exploit both the Cortex A9 cores inside the Zynq SoPC, we recorded a time overhead equal to about 25 % of the execution time of the original RICE application. This figure accounts for the rime needed to run the PikeOS run-time software, to send the keywords to the watchdog during RICE execution, and to perform memory comparison at the end of the execution of the application software.

We performed a preliminary analysis of the robustness of the proposed architecture by injecting faults in the processor register file. The injected faults either triggered the Error Detection Interrupt, or produced no visible effect (i.e., both the instances of RICE produced the expected results), thus suggesting the robustness of the proposed architecture.

15.7 Conclusions

SoPCs are very appealing for space applications as they allow integrating an entire system comprising high-performance processors and custom hardware accelerators on a single device, thus contributing in saving mass, area, and power. However, in order to deploy successfully SoPCs in space application, being these devices not intended for being used in radioactive environments like space, suitable countermeasures are needed to mitigate the effects of radiation-induced errors.

In this chapter we presented an overview of existing techniques for coping with radiation-induced errors, focusing on soft errors affecting the processor cores the SoPCs embed, and discussing a number of Software Implemented Fault Tolerance techniques. Moreover, a novel architecture is presented specifically designed for the processors embedded into SoPCs, which makes use of a combination of know techniques: a custom watchdog is synthesized and mapped to the SoPC reconfigurable fabric to cope with persistent SEE effects, while program-level instruction redundancy is exploited to cope with data and code errors. An implementation of the proposed architecture is finally discussed where an image compression algorithm is implemented using the Cortex A9 processor a Zynq SoPC offers. From the

experimental results we gathered, using the Sysgo PikeOS embedded hypervisor are hardware abstraction layer, we observed a memory overhead of about 100 %, in line with the expectation as we employ duplication, and a time overhead of about 25 %.

Acknowledgement The work described in this chapter has been developed in the frame of the project entitled "ECM^2: Embedded Multi-Core systems for Mixed Criticality applications in dynamic and changeable real-time environments" under the ARTEMIS Joint Undertaking action number 621429.

References

1. Hamblen JO, Hall TS (2006) Using system-on-a-programmable-chip technology to design embedded systems. Int J Comput Appl 13(6):142–152
2. Ma TP, Dressendorfer V (1989) Ionizing radiation effects in MOS devices and circuits. Wiley, New York. ISBN 978-0-471-84893-6
3. Goloubeva O, Rebaudengo M, Sonza Reorda M, Violante M (2006) Software implemented hardware fault tolerance. Springer, New York
4. Mahmood A, McCluskey EJ (1988) Concurrent error detection using watchdog processors—a survey. IEEE Trans Comput 37(2):160–174
5. Rebaudengo M, Sonza Reorda M, Torchiano M, Violante M (1999) Soft-error detection through software fault-tolerance techniques. International symposium on defect and fault tolerance in VLSI systems, 1999, IEEE, pp 210–218
6. Rebaudengo M, Sonza Reorda M, Torchiano M, Violante M (2001) A source-to-source compiler for generating dependable software. In: Proceedings of the first IEEE international workshop on source code analysis and manipulation, 2001, IEEE, pp 33–42
7. Cheynet P, Nicolescu B, Velazco R, Rebaudengo M, Sonza Reorda M, Violante M (2000) Experimentally evaluating an automatic approach for generating safety-critical software with respect to transient errors. IEEE Trans Nucl Sci 47(6):2231–2236
8. Oh N, Shirvani PP, McCluskey EJ (2002) Error detection by duplicated instructions in super-scalar processors. IEEE Trans Reliab 51(1):63–75
9. Benso A, Chiusano S, Prinetto P, Tagliaferri L(2000) A C/C++ source-to-source compiler for dependable applications. In: Proceedings international conference on dependable systems and networks, IEEE, pp 71–78
10. Oh N, McCluskey EJ (2002) Error detection by selective procedure call duplication for low energy consumption. IEEE Trans Reliab 51(4):392–402
11. Reinhardt SK, Mukherjee SS (2000) Transient fault detection via simultaneous multithreading. In: Proceedings of the 27th international symposium on computer architecture, pp 25–36
12. Echtle K, Hinz B, Nikolov T (1990) On hardware fault detection by divers software. In: Proceedings of the 13th international conference on fault-tolerant systems and diagnostics, Bulgarian Academy of Science
13. Engel H (1996) Data flow transformations to detect results which are corrupted by hardware faults. In: Proceedings of the high-assurance systems engineering workshop, 1996, IEEE, pp 279–285
14. Yau SS, Chen F-C (1980) An approach to concurrent control flow checking. IEEE Trans Softw Eng 6(2):126–137
15. Alkhalifa Z, Nair VS, Krishnamurthy N, Abraham JA (1999) Design and evaluation of system-level checks for on-line control flow error detection. IEEE Trans Parallel Distrib Syst 10(6):627–641
16. Goloubeva O, Rebaudengo M, Sonza Reorda M, Violante M (2003) Soft-error detection using control flow assertions. In: Proceedings of the 18th IEEE international symposium on defect and fault tolerance in VLSI systems, IEEE, pp 581–588

17. Goloubeva O, Rebaudengo M, Sonza Reorda M, Violante M (2005) Improved software-based processor control-flow errors detection technique. In: Proceedings of the annual reliability and maintainability symposium, IEEE, pp 583–589
18. Avizienis A, Laprie J-C (1986) Dependable computing: from concepts to design diversity. Proc IEEE 74(5):629–638
19. Avizienis A et al (1985) The n-version approach to fault-tolerant software. IEEE Trans Softw Eng 11(12):1491–1501
20. Randell B (1975) System structure for software fault tolerance. IEEE Trans Softw Eng 1(2):220–232
21. Price CE (1991) Fault tolerant avionics for the space shuttle. In: Proceedings of the 10th IEEE/AIAA digital avionics systems conference, IEEE, pp 203–206
22. Briere D, Traverse P (1993) Airbus A320/A330/A340 electrical flight controls—a family of fault-tolerant systems. In: Digest of papers of the twenty-third international symposium on fault-tolerant computing, IEEE, pp 616–623
23. Riter R (1995) Modeling and testing a critical fault-tolerant multi-process system. In: Digest of papers of the twenty-fifth international symposium on fault-tolerant computing, IEEE, pp 516–521
24. Hagelin G (1998) Ericsson safety system for railway control. Software diversity in computerized control systems, Springer, pp 11–21
25. Kantz H, Koza C (1995) The Elektra railway signaling system: field experience with an actively replicated system with diversity. In: Digest of papers of the twenty-fifth international symposium on fault-tolerant computing, IEEE, pp 453–458
26. Amendola A, Impagliazzo L, Marmo P, Mongardi G, Sartore G, Trasporti A (1996) Architecture and safety requirements of the acc railway interlocking system. In: Proceedings of the IEEE international computer performance and dependability symposium, IEEE, pp 21–29
27. Echtle K, Hinz B, Nikolov T (1990) On hardware fault detection by diverse software. In: Proceedings of 13th international conference on fault-tolerant systems and diagnostics, pp 362–367
28. Abraham JA, Huang K-H (1984) Algorithm-based fault tolerance for matrix operations. IEEE Trans Comput 100(6):518–528
29. Abraham JA, Jou J-Y (1988) Fault-tolerant FFT networks. IEEE Trans Comput 37(5):548–561
30. Connet JR, Pasternak EJ, Wagner BD (1972) Software defenses in real-time control systems. In: Digest of the 1972 international symposium on fault-tolerant computing, pp 94–99
31. Namjoo M, McCluskey EJ (1995) Watchdog processors and capability checking twenty-fifth international symposium on fault-tolerant computing, highlights from twenty-five years, IEEE, p 94
32. Saib S (1979) Distributed architectures for reliability. In: Proceedings of the AIAA computers in aerospace conference II, Los Angeles
33. Mahmood A, Ersoz A, McCluskeym EJ (1985) Concurrent system-level error detection using a watchdog processor. In: IEEE proceedings of the 15th international test conference
34. Allen FE (1970) Control flow analysis. SIGPLAN 5(7):1–19
35. Bernardi P, Bolzani LMV, Rebaudengo M, Sonza Reorda M, Rodríguez-Andina JJ, Violante M (2006) A new hybrid fault detection technique for systems-on-a-chip. IEEE Trans Comput 55(2):185–198
36. Caleno M, Fertin D, Giulicchi L, Monteleone C (2007) On-board data reduction. ESA report S2-EST-RP-XXXX, 5 Nov 2007

Chapter 16
Soft Error Mitigation in Soft-Core Processors

Antonio Martínez-Álvarez, Sergio Cuenca-Asensi, and Felipe Restrepo-Calle

Abstract This chapter aims to present different approaches and techniques available in literature regarding the fault mitigation on soft-core processors, with an especial emphasis on those ones involving hardware/software hybrid-based solutions.

16.1 Introduction

Every advance in lithography technology is usually followed by a technological shrinking of electronic components which implies important improvements in microprocessors, mainly the remarkable increase of their performance. Nevertheless, this trend also reports adverse consequences mainly due to the narrower voltage source level and noise margins; in fact, this induces electronic devices to be more susceptible to transient faults induced by radiation [1–3] and finally having less reliable microprocessors. The term transient fault is used to define intermittent faults which are caused by external events, as those ones induced by radiation. Although these faults do not provoke a permanent damage, they may cause incorrect circuit behavior by: altering a signal transfer or altering a stored value [2]. In this way, it is clear that these faults can affect seriously the behavior of a given system [4].

This chapter is focused on the type of radiation-induced transient faults known as *Single Event Upset* (SEU), which is characterized by the logic state alteration of a single memory element in the system [5]. SEUs were considered in the past as a concern only for aerospace applications, where they are more frequent. However, in recent decades, this problem has been extended to electronic circuits operating in the atmosphere [6], and even at ground level [7], and thus, this issue have become a major source of system failures. Summarizing, radiation-induced transient faults have become a major source of system failures of electronic products even at ground level [6, 8].

A. Martínez-Álvarez (✉) • S. Cuenca-Asensi
Department of Computer Technology, University of Alicante, Alicante, Spain
e-mail: amartinez@dtic.ua.es; sergio@dtic.ua.es

F. Restrepo-Calle
Department of Systems and Industrial Engineering, Universidad Nacional de Colombia, Bogotá, Colombia
e-mail: ferestrepoca@unal.edu.co

F. Kastensmidt, P. Rech (eds.), *FPGAs and Parallel Architectures for Aerospace Applications*, DOI 10.1007/978-3-319-14352-1_16

16.1.1 The Necessity for Fault Mitigation

The need to mitigate radiation-induced transient faults has become evident in several reports published by technical committees around the world, which define detailed qualification requirements that electronic components must meet for their use. Some examples of which are among others:

- ESA PSS-01-609 (*The Radiation Design Handbook*) [9] for aerospace application
- DO-254 (*Design Assurance Guideline for Airborne Electronic Hardware*) [10] and IEC/TS 62396 (*Process Management for Avionics—Atmospheric radiation effects*) [11] for avionics
- MIL-HSBK-817 (*System Development Radiation Hardness Assurance*) for military systems [12],
- AEC-Q100 (*Stress Test Qualification for Integrated Circuits*) for automotive industry [13]

16.1.2 Possible Approaches

Three main different approaches to mitigate radiation-induced transient faults can be distinguished: in one hand, we can implement several pure software or hardware solutions, and in the other hand, we may select a hybrid hardware/software approach. Of course, we are restricted to improve our system in those places where it is practicable in relation to the inherent technological restrictions of the system. For example, we cannot apply hardware redundancy in the internal resources of a hard-core processor, but we may, if possible, take advantage of external built-in or ad-hoc hardware resources to improve reliability in some way. This chapter focuses in those techniques and approaches regarding FPGA technology, the common substrate to implement soft-cores.

16.2 FPGA as Technological Platform for Soft-Cores

The present trend to integrate in the same encapsulated an ever increasing number of different computing units and resources has coined the term of *System on Chip* (SoC). The computing performance and functionalities is a growing tendency as well. However, these chips, heterogeneous in nature, suppose a new challenge if we are interested in designing a fault tolerant application. FPGA technology not only offers a possible implementation resource of SoC, but also permits the exploration of a rich design space focusing in fault tolerance systems.

FPGA vendors offer three main technological implementations of these chips: SRAM-based FPGA, Flash-based FPGA, and antifuse-based FPGA.

SRAM-based FPGAs are currently the most demanded FPGA technology due mainly to its performance, convenient costs, and its inherent reconfiguration capabilities. Indeed, the configuration memory is implemented as a SRAM memory, and thus, it allows any number of reconfigurations actions. Regarding fault sensibility, however, this technology presents a serious drawback because of the low immunity to radiation effects of the SRAM memory. Just take into account that a SEU affecting a configuration bit may modify logic functions, connections and may affect the normal functioning of the system (SEFI). Bits from configuration memory suppose up to 95 % of the total bits susceptible suffering a SEU in a SRAM-based FPGA. Accordingly, it is necessary to protect from SEEs not only the design, but also the configuration memory.

Flash-based FPGAs have also the quality of being reconfigurable in spite of being based on a non-volatile type of memory. Moreover, the Flash cell presents immunity to radiation caused by heavy ions.

Antifuse-based FPGAs have the characteristic of being one-time programmable devices, because they are configured by means of antifuses. However they present immunity to SEEs.

In the case of having Flash-based or Antifuse-based FPGA, it is only necessary to protect from SEEs only the design, and no the configuration memory as in the case of SRAM-based FPGA.

16.2.1 Alternatives

We can find different possible alternatives of mitigating the radiation effects in an FPGA-based system:

- Application of solutions based on the improvement of the fabrication technology and the internal architecture by using radiation tolerant resources. These FPGA chips are known as rad-hard FPGAs. Although cancelling the effects of SEUs, these devices present less performance than non rad-hard FPGAs, have poorer integration capabilities and are too costly in many cases. In fact, its use is commonly limited to mission critical systems. As examples of rad-hard FPGA we can find Actel RTAX FPGAs [14], and Xilinx Virtex-5QV [15].
- Some authors have presented system level approaches consisting in the application of redundant devices using double or triple FPGAs together with majority voters commanding the system output [16].
- Design level hardening by applying redundancy in the HDL (Hardware Description Language) implementation of a system. Redundancy may be applied to harden the user logic, embedded memories, multiplexors, registers, etc. [17]. With this alternative, competitive commercial FPGAs can be used at a relative low cost. The implementation can be either manual, which is more costly and more prone to design errors, and automatic by mean of tools such as *Mentor Precision Rad-Tolerant* [18], *Synopsys Synplify Pro* [19] and *Xilinx TMR Tool* (*XTMR*) [20].

- Recent works propose altering the task of *place and route* in the implementation of an FPGA-based system in such a way to make the final result more reliable (e.g. lowering the multiple bits upsets in a given TMR module) [21, 22].
- To minimize the problems related to the configuration memory of SRAM-based FPGAs, it is common using a periodic device reconfiguration or *scrubbing*. The following two main flavors of *scrubbing* come to scene:
  - *Bitstream scrubbing configuration*: reconfiguring the FPGA as a rate higher than the expected fault frequency.
  - *Bitstream repair configuration or advanced scrubbing*: A read-back process calculates the CRC of the bitstream at a certain rate and corrects it using partial reconfiguration in case of mismatching CRCs [17].

Other works to improve the fault tolerance in SRAM-based FPGAs can be consulted in [17, 23–26].

16.3 Hardware Approaches

Among the protection techniques based on adding some kind of hardware redundancy, we can distinguish two main approaches: those ones based on providing systems with redundant information for protecting memories, and those ones based on mitigating faults by means of custom modifications of the circuit logic, arranging from a logic gate level, up to a system level architectural strategy.

16.3.1 Memory Protection Based on Information Redundancy

Although memory protection is not under the scope of this chapter it is worth mentioning that these devices represent a ubiquitous resource in practically every modern computing system. Their inherent high integration density makes them significantly susceptible to ionizing particles causing *SEEs*, and therefore they are the first candidate to be protected when designing a fault tolerant system [27].

Whereas permanent faults in memory can be solved by the so called *Built-In Self-Repair* techniques (BISR) [24], this procedure is not applicable to radiation-induced transient faults. In those cases, *Error Detection and Correction Codes* (*EDAC*), that is, information redundancy techniques to mask these faults must be applied (see [28, 29]).

16.3.2 Memory Protection Using the Circuit Logic

Several solutions based on the use of redundant hardware to protect the circuit logic can be implemented. At the less possible level of actuation (transistor level), we can find proposals to harden the memory cells by design, whereas at a higher level, that

is, Logic Gate Level, *Register Transfer Level* (*RTL*) or even at system level, some redundancy can also be applied during the system designing.

At the Logic Gate Level, different circuit flavors as: microprocessors, memories, ASICs or reprogrammable circuits among others, can be protected by replacing the usual memory cells (SRAM cells, flip-flips, latches) by their respective hardened versions. This task can be achieved by modifying the cell *layout*, as in the case of scaling its VLSI design to increase the minimal necessary electric charge (critical charge) to switch every gate node. Other approaches are based on hardening the cell architecture/structure. As an example, the reference [30] presented a hardened by design SRAM cell (*Dual Interlocked Cell*—DICE), which is tolerant to any transient fault occurring in a single node. The main drawbacks of these approaches consist in the overheads due to the increase of silicon area and the loss of gate performance. Note that these techniques are not applicable to FPGA technology.

Working at RTL, the common strategies to accomplish the protection con consist in replicating logic structures (or modules) from the circuitry to obtain redundancy. Thus, we can have DMR (Double Modular Redundancy) to detect faults in the case of mismatched results, or TMR (Triple Modular Redundant) for detection and correction capacities by using a majority voter [31]. More examples applying these principles with a different granularity can be consulted in [16, 32, 33].

Temporal redundancy can also be used to detect and mask transient faults. This technique requires much less hardware modules for its implementation, and can be implemented by either repeating a calculus in different time instants or [34, 35], or registering output data in different time instants without the need for repeat any calculus [36, 37].

Although all presented techniques suppose effective ways of protecting the memory by altering the logic circuitry, in many cases the generated overheads in silicon area, power and performance are excessive and discourage the designer from implementing them.

16.4 Software Approaches

For those systems based on or having microprocessor parts, software redundancy can be used to achieve fault tolerance. This strategy supposes logically the use of no new hardware or any hardware modification whereas permits a high level of flexibility when detecting or correcting transient or permanent faults at a relative low cost of implementation (because it is possible to use COTS—*Commercial Off-The Shelf* components).

It is a fact that to obtain a fault tolerant system we have to detect and correct faults. However these two tasks are easily decoupled because they can be implemented independently. In addition, the relative low occurrence of faults makes the correcting routines to be executed at a lower rate than detecting routines. This is the reason why fault tolerance literature was centered from the beginning in optimizing detecting techniques which are supposed to be executed continuously.

We can distinguish two types of effects of faults affecting the software executed by a microprocessor-based system; those affecting the control flow of a program (i.e. a fault changing the Program Counter or even an operation code at a given time instant), and those affecting the data within a program. Several techniques exit to face these two effects:

16.4.1 Techniques to Protect the Control Flow of a Program

The so called *Control Flow Errors* or CFEs are those software errors driving the processor to execute an unexpected instruction. Common testing control flow techniques are based on splitting the program in its *basic blocks* and inspecting the execution flow among them. A program *basic block* is a set of consecutive instructions without any jump or call instruction except possibly the last one in the set, and without any instruction being the destination of an external jump or call instruction with the exception of the first instruction of the set. The control flow of a program can be depicted using a *Control Flow Graph* (CFG), which is a directed graph with nodes representing the basic blocks, and every transition representing the jumps among basic blocks. Figure 16.1 shows the CFG of a hypothetical program having five basic blocks with a different number of instructions (I_i).

The basic of the techniques to protect the control flow are based on building the CFG, and identifying univocally every node with a signature. In this way, at the very end of every node execution the correctness of the signature is checked to detect a fault. Techniques working in this way are known as *Signature Monitoring Techniques*.

There are five different types of possible faults affecting a CFG. Note that the first four take place between different nodes, whereas the last one is about the same node:

- Faults causing an illegal jump from the last instruction of a node up to the beginning of another one.
- Faults causing a legal jump from the last instruction of a node up to the beginning of an incorrect one.

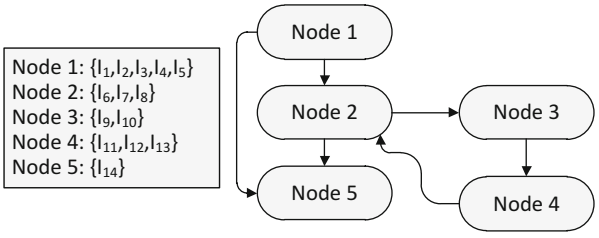


Fig. 16.1 Example of a control flow graph for ah hypothetical program

- Faults causing a jump from the last instruction of a node to any instruction of another one with the exception of the first one.
- Faults causing a jump from any instruction of a node except the last one, to any instruction of another one with the exception of the first one.
- Faults causing a jump from any instruction of given node except the last one, to any instruction of the same node.

There are several proposals for mitigating control flow faults depending on the different use of resources involved, the induced overheads (mainly overheads in time and program size) and the type of fault affection the CFG under consideration. In [38] a quite exhaustive study of the most representative techniques can be consulted. Some of them merit being mentioned:

- In [39], a signature monitoring technique implemented using the multithreading and multiprocessing capabilities of an operating system is presented.
- In [40], a technique known as *Assertion for Control Flow Checking* (ACFC) is presented. It consists on assigning a bit from a special variable known as *execution state* to each node from the CFG. The bit corresponding to each node under execution is set. When the program finishes, the *execution state* is compared with a precalculated constant having bit set in those places matching a correct execution.
- The so called *Yet Another Control Flow Checking Approach* (YACCA) is presented on [41]. In this technique, a pair of unique identifiers is assigned to each node from CFG, one for the input and the other one for the output. A special code inserted in the program can detect faults in the control flow by making some calculus taking into account the completed set of identifiers.
- *Control-Flow Checking by Software Signatures* (CFCSS) is presented on [42]. It consists in adding special instructions in the program at compile time to control the flow between nodes, which are univocally identified with a signature. Signatures are calculated at runtime and compared with those precalculated at compile time, and thus, a mismatch supposes the detection of a fault.

16.4.2 Techniques to Protect Data

The classic approximation to solve the problem of mitigating faults affecting data is known as *N-versions programming* [43]. It consists in replicating N times every software piece of interest producing a given output and obtaining the correct value by mean of a majority voter. This produces an increase up to $100(N-1)\%$ in area and execution time. Different techniques have been proposed in literature aiming to improve the mentioned overheads. They can be distinguished by their application granularity that can be program, procedure or instruction.

Methods based on *software redundancy at program level* can follow three different strategies:

- Temporal redundancy: In this case, the same program will be executed two times at different instants. The last execution of the program will compare both outputs to detect any fault. An example of this can be consulted on [44].
- Simultaneous execution: The increasing parallel resources of modern microprocessors with multithreading and multiprocessing capabilities, permits the concurrent execution of different instances or *versions* of the same code, and thus, the detection of faults reducing the execution time overhead.
- Data diversity: This technique consists in executing two programs having the same functionality (and therefore a different implementation code) and diverse input data. Both outputs are then compared to detect transient or permanent faults. From a given program, the new version is commonly generated by multiplying by a *diversity factor k* every variable and constant within the program. Depending on *k* value, each version of the program may use different hardware resources and thus, may propagate the errors in different ways. As an example, Fig. 16.2 present a given piece of code and its diversified version generated by applying a diversity factor $k=-2$. An example of this technique can be consulted on [45].

Methods providing *software redundancy at procedure level* are based on the so called *Selective Procedure Call Duplication* (SPCD) of the execution of some procedures [46] (a block of code that performs a single task and returns some values). This technique duplicates the execution of each procedure from a set of procedures, saves the respective outputs and provides by this way the error detection. Re-computation is performed a third time if a discrepancy between the previous two computations occurs. Figure 16.3 shows an example of this technique, where a given procedure is called two times.

Finally, software redundancy methods at instruction level are based on repeating single instructions or a set of them in such a way that it is possible to detect and correct errors. The proposal differs in the granularity of the software to be replicated, which can be defined either at a high level language (e.g. C/C++) or at low level (e.g. assembly code). Both of them try to reduce the inherent overheads in size and execution time, as well as optimize the provided fault coverage.

Original version	*Diversified version*

`x = 1; y = 5; i = 0;`
`while (i<5) {`
`  z = x + i * y;`
`  i = i + 1;`
`}`
`i = 2 * z;`
	`x = -2; y = -10; i = 0;`
	`while (i>-10) {`
	`  z = x + i * y / (-2);`
	`  i = i + (-2);`
	`}`
	`i = (-4) * z / (-2);`

Fig. 16.2 Example of a program in its original and diversified version with diversity factor $k=-2$

Fig. 16.3 Example of which applies software redundancy at procedure level

```
int a, a1, b, c;
void A2() {
  a = B(b);
  a1 = B(b);
  if (a <> a1)
    error();
  c = c + a;
}
int B(int b) {
  int d;
  d = 2 * b;
return d;
}
```

Original code	*Modified code*
`int a,b;` `a = b;`	$int\ a_0, b_0, a_1, b_1;$ $a_0 = b_0;$ $a_1 = b_1;$ $if\ (b_0 != b_1)\ error();$
`a = b + c;`	$a_0 = b_0 + c_0;$ $a_1 = b_1 + c_1;$ $if\ ((b_0 != b_1) \| (c_0 != c_1))\ error();$

Fig. 16.4 ARBT application example to detect errors on every read operation

On [47], a method called *Automatic Ruled Based Transformation* (ARBT) is presented. This method is based on the application of high-level instruction redundancy following a set of rules which can be automatically applied to the software [48] using a source-to-source compiler. The high-level code transformation makes this technique independent from the microprocessor architecture. Detention capabilities are provided by duplicating every variable from the program, and the insertion of consistency checkers at the end of a read operation. Figure 16.4 shows an example of its application.

On [46], the *Error Detection by Duplicated Instruction* (EDDI) is presented. Redundancy at assembly instruction is proposed to reduce the code size and execution time overheads. In addition *Instruction Level Parallelism* (ILP) can be used in superscalar processor to speed-up the execution (Fig. 16.5).

Other well-known fault detection technique called *SoftWare Implemented Fault Tolerance* (SWIFT) is presented on [49]. This technique also reduces the inherent time and size overheads, by using ILP also on *Very Long Instruction* Word (VLIW) architectures. This technique, based on EDDI + CFCSS, is implemented by inserting

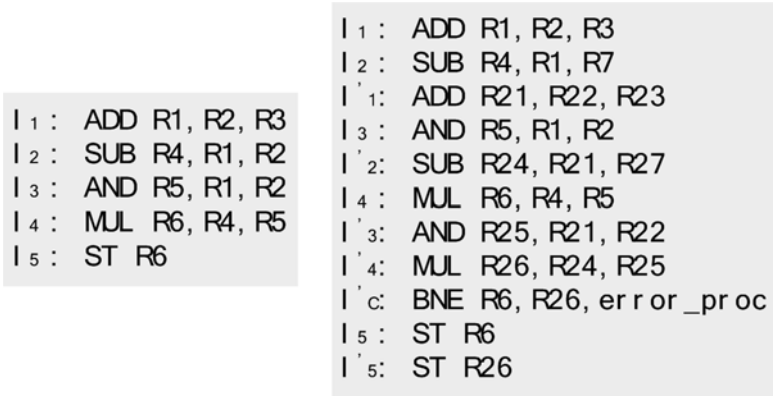


Fig. 16.5 EDDI example of a hardened block of code (*right* side) from its original version (*left* side)

Table 16.1 Comparative results when applying different software-based detecting techniques to improve fault tolerance

Proposal	Granularity	Architecture	Code overhead	Data overhead	Execution overhead	Detected faults (%)
ARBT	High-level	Transputer T255, 8051, MC68040, LEON	×5	×2	×3	63
EDDI	Low-level	MIPS R10000 ISA-II	×1.5–×2	×2	<×2	97
CFCSS	Low-level	MIPS R4400 ISA-II	×1.2–×1.4	No data	×1.2–×1.7	96.9
SWIFT	Low-level	VLIW	×2.4	No data	×1.4	100

appropriate assembly instruction at compile time in certain point of the program. Other technique called SWIFT-R [50], improved SWIFT to prevent both detection and correction capabilities. SWIFT-R uses TMR at an assembly level and provides majority voters when necessary.

It is a fact that as mentioned before, every software-based technique causes unavoidable drawbacks regarding to time, code and data size overheads.

To show an evidence of such variability, Table 16.1 presents results when comparing the different software-based hardening techniques (ARBT, EDDI, CFCSS, and SWIFT) to detect faults. In other hand, Table 16.2 shows the same targets when applying software-based recovering techniques (ARBT-FT, SWIFT-R) which are focused con fault mitigation. Both tables present the fault coverage for each technique (in the last column), the different induced overheads, and the granularity of the technique (which can be either *low-level* for instruction-based ones or high-level for techniques having redundancy at C/C++ or procedural/functions).

Table 16.2 Comparative results when applying different software-based mitigation techniques to improve fault tolerance

Proposal	Granularity	Architecture	Code Overhead	Data overhead	Execution overhead	Detected faults (%)
ARBT-FT	High-level	8051	×2	×2–×3	×2.5	99.5
SWIFT-R	Low-level	VLIW	No data	No data	×1.9	97.27

In direct relation with the above mentioned tables, we can conclude:

- Those methods performing its operation at a low-level of granularity, it is, at an instruction or assembly level present less impact on the code and data sizes overhead.
- The impact on execution time overhead is also lower in the case of using low-level techniques.
- An improvement on performance can be obtained in the case using superscalar and VLIW microprocessors.

16.5 Hybrid Approaches

Among the different techniques aiming to increase the reliability of a microprocessor-based system, which are commonly substantiated in the implementation of some level of redundancy at pure software or hardware level, we can find in literature the so called hybrid approaches. They are defined by means of a hardware/software implementation of different techniques using some level of redundancy in both software and hardware, or the appropriate combination of using software redundancy plus an external or internal hardware support.

Regarding the motivation of using hybrid techniques to increase the fault tolerance of a microprocessor-based system, it is clear that the application of pure software or pure hardware techniques suppose different drawbacks. Indeed, the inherent variability when applying these pure techniques makes the election of the best tuple (microprocessor system, fault tolerance technique) a really hard engineering problem to solve. Indeed, it is a question concerning to the optimization of a multi-objective problem.

Table 16.3 represents how variability of different parameters could affect the election of the appropriate technique and thus elicits a non-trivial problem.

In addition to all this, the application of some software techniques, may require the addition of more microprocessor data or memory resources, and thus may demand the change of the COTS or soft-core microprocessor device. Moreover, pure software techniques may rapidly degrade the overall system performance, especially if a low-performance microprocessor is used.

Having all the previous considerations into account, the applicability of pure software or hardware techniques is not feasible in many cases, especially in the

Table 16.3 Relative a-priori impact of variability of different parameters when applying pure software or hardware techniques to protect a microprocessor-based system

	Pure software techniques	Pure hardware techniques
Code size overhead	++	
Memory consumption	+	+
Execution time	+++	+
Power consumption	+	+
Area consumption		++
Fault coverage	+	++
Non recurrent engineering costs (NRE)	+	+++
Budget	+	++

notable case of embedded-system scenario, where low-performance and low-power processors are commonly objectives that can be as important as reliability. Indeed, a solution representing an intermediate point in between pure software and hardware techniques can be a suitable solution, that is, the use of a hybrid software/hardware technique, combining protection in both scenarios and achieving the adequate balance in every parameter from Table A.

In the last years, several and very promising hybrid proposals have entered the scene. On [51, 52] a technique called *CompileR Assisted Fault Tolerance* (CRAFT) is presented. This technique is based on a modification of SWIFT technique [49] where hardware redundancy is obtained by using RMT (*Redundant Multi-Threading*) [53] in hardware structures. This technique presented three variations. The first of them combines SWIFT with a hardware structure that protects data loads from memory. The second one combines SWIFT with a hardware mechanism aiming to duplicate and protect data on memory. The last one combines the two previous hardware structures to provide protection for both, loading and storing events.

On [54, 55] a fault detection hybrid technique is presented. Its definition is based on ARBT (*Automatic Ruled Based Transformation*) [34] to protect the code, and YACC algorithm [41] to protect the control flow. The combination of both methods is achieved by means of an external mechanism connected to the system bus and assuring less computation effort, in such a way that it is possible to improve both the program performance and the detection rate. Recently this technique has been extended [56, 57] to provide also fault mitigation by applying the code transformation rules proposed by Rebaudengo et al in [58].

A new hybrid technique for detecting both SEUs and SETs is presented on [59, 60]. It is based on code transformation rules that permit signature monitoring. On the hardware side, an external hardware module is used. It consists in a *watchdog* plus a *decoder* module which are intended for detecting control flow faults and verifying data and addresses flow between processor and memory. A similar approach is presented in [61], where in addition SMT (*Simultaneous Multi-Threading*) is used to provide fault recovering by means of redundant execution of diverse copies of each thread.

Other proposals define an architecture for superscalar microprocessors to detect and correct faults with a little impact on system performance [62].

A partial protection of the register file of a microprocessor to find the best tradeoff between reliability and power consumption is presented in [63].

The presented hybrid solutions are still very specific and have no much flexibility to find the best compromise between design constraints and reliability requirements.

Another new hybrid technique intended to protect both the data and the control-flow of embedded applications is presented on [64]. On the software side, two different hardening techniques are confronted: SWIFT-R [49] which is based on instruction replication and *Procedural Replication* (PR) [46] where the replication unit is the procedure (function). On the hardware side, a dedicated hardware module (*CFC module*) performs *Control-Flow Checking* (CFC) of the program execution. This module accesses the internal resources of the microprocessor (μP) by means of the available debug infrastructure which enables support for software debugging in embedded system development (see Fig. 16.6).

These resources can be easily reused for online monitoring in an inexpensive way, because they are useless during normal operation. In addition, they provide internal access to the processor without disturbing it and do not require any modification neither to the hardware nor to the software so no performance penalties have to be taken into account. Overheads incurred by this technique can be perfectly assumable in low-cost systems. Both, SEUs and SETs fault can be mitigated using this technique.

Other hybrid approach to mitigate soft-errors which is based on low level automatic refreshing of system configuration registers by taking advantage of usual microcontroller resources as programmable timers is presented on [65]. Results demonstrate that SEU and SET effects can be effectively mitigated in interrupt-driven applications. In direct relation with the criticality and the access frequency of configuration registers two hardening flavors are proposed: static configuration hardening (devoted to the hardening of those peripheral configuration registers which remain unchanged during the whole program execution) and dynamic configuration hardening (those processor or peripheral configuration registers which are occasionally modified during program execution). Figures 16.7 and 16.8 show

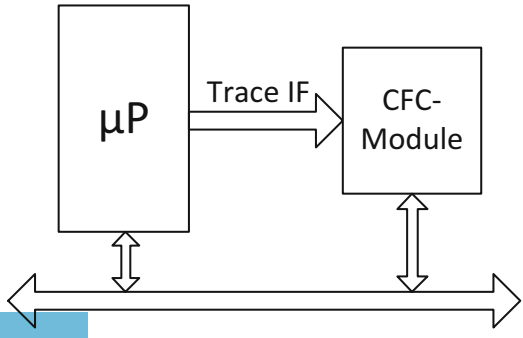


Fig. 16.6 System structure hardened with a CFC-module

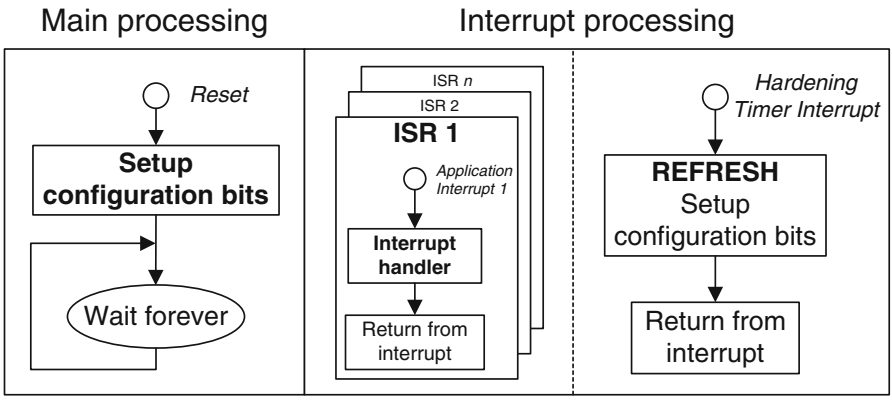


Fig. 16.7 Static configuration hardening in event-driven applications

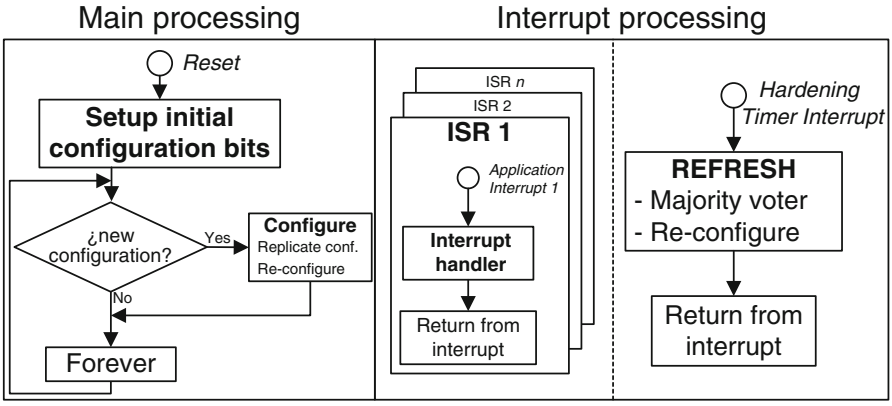


Fig. 16.8 Dynamic configuration hardening in event-driven applications

the hardening scheme followed by each flavor. In the first case (Fig. 16.7), an *Interrupt Service Routine* (ISR) refreshes the configuration bits of the system at a given and modifiable frequency. In the second case (Fig. 16.8), the timer-driven hardening routine does not know beforehand what value to refresh into the configuration register, as this may vary during program execution. Dynamic hardening requires the following actions. Firstly, the main algorithm writes the new configuration data also into a protected (e.g. replicated) register every time it changes (copies should be performed before proceeding to re-configure the system). Secondly, the timer-driven hardening routine includes a majority voter to check correctness, and finally, refreshes the configuration data accordingly. As in the static case, the hardening interrupt service routine is triggered periodically.

Table 16.4 shows a summary comparison between some of the reported hardware/software hybrid techniques. Each result is taken from the referenced papers from the first column.

Table 16.4 Comparative results of presented hybrid mitigation techniques

Approach	SW support	Monitored bus	Latency	Case study		Overheads			Faults injected	Error detection
				Processor	Benchmarks	Area	Time	Code		
[55]	Data & ctrl	Memory buses	Low	PowerPC	MMult, Ellip–filter, LZW, Viterbi	366 slices	2.06–2.81	2.07–3.09	SEUs 100K	SEU: 90.6–95.6 %
[59]	Data & ctrl	Memory buses	Low	miniMIPS	BubleSort, MMult	15 % (128 FFs)	2.33–2.53 (ctrl. only 1.33–1.60)	3.26–3.60	SEEs 50K	SEU: 100 %
										SET: 100 %
[66]	Data & ctrl	Memory buses	Low	miniMIPS	BubleSort, MMult	11 %	(ctrl. only 1.08–1.34)	2.79–2.91	SEEs 100K	SEU: 100 %
										SET: 100 %
[67]	Data & ctrl	Dbg./trace interface	High	LEON, ARM7	Fibbonaci, Ellip–filter	16 %	2.00	–	SEUs 10K	SEU: 99 %
SWIFT-R + CFC [64]	Data only	Dbg./trace interface	Low	Picoblaze	MMult, PID, FIR filter	435 gates	2.55–2.68 (ctrl. only 1)	2.37–3.02	SEEs 130M	SEU: 97.72–99.31 %
						119 FFs				SET 99.83–99.94 %
Proc. Rep. (PR) + CFC [64]	Data only	Dbg./trace interface	Med (data) low (ctrl)	Picoblaze	MMult, PID, FIR filter	435 gates	1.96–2.03 (ctrl. only 1)	1.07–1.65	SEEs 130M	SEU 98.60–99.85 %
						119 FFs				SET 99.89–99.98 %

As both techniques require no software modifications for control-flow error detection, the execution time overhead is only due to their data hardening capabilities. When the SR (SWIFT-R) technique is used, the overhead is slightly larger because it includes both detection and recovery. In contrast, code size overhead is smaller or similar. Each approach differs in how the fault injection technique+ is implemented: [55] uses software-based fault injection, while [59, 66, 67] inject faults directly in the VHDL signals from the code defining the soft-core microprocessor. Fault injection into every node of the final synthesized netlist is performed, considering the real delays as estimated by the synthesis tool. With respect to error detection, the last two rows are close to 100 %, but the fault injection experiments are much larger and more accurate.

16.6 Conclusion

The very first conclusion of this chapter has to do with the fact that choosing the correct protection technique in between the several proposals presented (from pure hardware or software schemes or by means of hybrid solutions) represents a priori a difficult task involving several important tradeoffs.

Experimental hybrid results make evidence of an important increase in the system reliability, which is even superior to two orders of magnitude, in terms of mitigation of both SEUs and SETs. However, further studies may be taken into consideration due to the inherent variability of the factors involved in fault detection and/or mitigation, as well as the different nature and available resources for every particular system.

In addition, the unavoidable induced overheads should encourage and leads us to the search of new improved techniques and approaches for mitigating soft-errors by optimizing both, the harmful impact of these overheads while, and the fault coverage.

References

1. Baumann RC (2005) Radiation-induced soft errors in advanced semiconductor technologies. IEEE Trans Device Mater Reliab 5:305–316. doi:10.1109/TDMR.2005.853449
2. Shivakumar P, Kistler M, Keckler SW, Burger D, Alvisi L (2002) Modeling the effect of technology trends on the soft error rate of combinational logic. In: Proceedings of the international conference on dependable systems and networks, IEEE Computer Society, pp 389–398. doi:10.1109/DSN.2002.1028924
3. Benedetto JM, Eaton PH, Mavis DG, Gadlage M, Turflinger T (2006) Digital single event transient trends with technology node scaling. IEEE Trans Nucl Sci 53:3462–3465. doi:10.1109/TNS.2006.886044
4. Perry F, Mackey L, Reis GA, Ligatti J, August DI, Walker D. (2007) Fault-tolerant typed assembly language. In: Proceedings of the 2007 ACM SIGPLAN conference on programming language design and implementation—PLDI'07. ACM Press, New York, p 42. doi:10.1145/1250734.1250741

5. Karnik T, Hazucha P (2004) Characterization of soft errors caused by single event upsets in CMOS processes. IEEE Trans Dependable Secure Comput 1:128–143. doi:10.1109/TDSC.2004.14
6. Edwards R, Dyer C, Normand E (2004) Technical standard for atmospheric radiation single event effects, (SEE) on avionics electronics. In: Proceedings of the 2004 IEEE radiation effects data workshop (IEEE Cat. No. 04TH8774), IEEE, pp 1–5. doi:10.1109/REDW.2004.1352895
7. Barth JL, Dyer CS, Stassinopoulos EG (2003) Space, atmospheric, and terrestrial radiation environments. IEEE Trans Nucl Sci 50:466–482. doi:10.1109/TNS.2003.813131
8. Michalak SE, Harris KW, Hengartner NW, Takala BE, Wender SA (2005) Predicting the number of fatal soft errors in Los Alamos national laboratory's ASC Q supercomputer. IEEE Trans Device Mater Reliab 5:329–335. doi:10.1109/TDMR.2005.855685
9. Agency ES (1993) The radiation design handbook ESA PSS-01-609. European Space Agency technical report
10. Fulton R (2014) Airborne electronic hardware design assurance: a practitioner's guide to RTCA/DO-254. CRC Press, Boca Raton
11. Commission IE (2006) IEC/TS 62396-1. Technical report, International Electrotechnical Commission
12. Council AE (2003) Stress test qualification for integrated circuits, AEC-Q100-Rev-F.2. Technical report
13. AEC-Q100 (1994) Stress test qualification for integrated circuits for automotive industry
14. Corporation A (2010) RTAX-S/SL and RTAX-DSP radiation-tolerant FPGAs. Data Sheet Rev 13
15. Xilinx Inc. (2010) Radiation-hardened, space-grade Virtex-5QV FPGA data sheet: DC and switching characteristics. Data sheet DS692 (v1.0.1)
16. Kubalík P, Kubátová H (2008) Dependable design technique for system-on-chip. J Syst Archit 54:452–464. doi:10.1016/j.sysarc.2007.09.003
17. Kastensmidt FL, Carro L, Reis R (2006) Fault-tolerance techniques for SRAM-based FPGAs (frontiers in electronic testing). Springer, Secaucus
18. Mentor Graphics Corporation (2010) Advanced FPGA synthesis: precision rad-tolerant. Data sheet 1028010
19. Inc. S (2010) Synopsys FPGA synthesis synplify pro reference manual. Technical report, Actel edition
20. Xilinx Inc. (2009) Aerospace and defense: Xilinx TMRtool. Technical report
21. Huang K, Yu H, Li X (2011) Cross-layer optimized placement and routing for FPGA soft error mitigation. In: Proceedings of the 2011 design, automation test in Europe conference exhibition, IEEE, pp 1–6. doi:10.1109/DATE.2011.5763018
22. Sterpone L, Violante M (2006) A new reliability-oriented place and route algorithm for SRAM-based FPGAs. IEEE Trans Comput 55:732–744. doi:10.1109/TC.2006.82
23. De Lima Kastensmidt FG, Neuberger G, Hentschke RF, Carro L, Reis R (2004) Designing fault-tolerant techniques for SRAM-based FPGAs. IEEE Des Test Comput 21:552–562. doi:10.1109/MDT.2004.85
24. Nicolaidis M, Achouri N, Boutobza S (2003) Dynamic data-bit memory built-in self-repair. In: Proceedings of the international conference on computer aided design ICCAD-2003, pp 588–594. doi:10.1109/ICCAD.2003.1257870
25. Lima F, Carro L, Reis R (2003) Designing fault tolerant systems into SRAM-based FPGAs. In: Proceedings of the 2003 design automation conference (IEEE Cat. No. 03CH37451), IEEE, pp 650–655. doi:10.1109/DAC.2003.1219099
26. De Lima FG, Cota E, Carro L, Lubaszewski M, Reis R, Velazco R, et al (2000) Designing a radiation hardened 8051-like micro-controller. In: Proceedings of the 13th symposium on integrated circuits and systems design (Cat. No. PR00843), IEEE Computer Society, pp 255–260. doi:10.1109/SBCCI.2000.876039
27. Nicolaidis M (2001) Soft errors in modern electronic systems, vol 41. Chapter 8. Front electron testing, 1st edn. Springer, New York
28. Neuberger G, de Lima Kastensmidt FG, Reis R (2005) An automatic technique for optimizing Reed-Solomon codes to improve fault tolerance in memories. In :Proceedings of the IEEE Des Test Comput 22:50–8. doi:10.1109/MDT.2005.2

29. Hentschke R, Marques F, Lima F, Carro L, Susin A, Reis R (2002) Analyzing area and performance penalty of protecting different digital modules with Hamming code and triple modular redundancy. In: Proceedings of the 15th symposium on integrated circuits and systems design, IEEE Computer Society, pp 95–100. doi:10.1109/SBCCI.2002.1137643
30. Calin T, Nicolaidis M, Velazco R (1996) Upset hardened memory design for submicron CMOS technology. IEEE Trans Nucl Sci 43:2874–2878. doi:10.1109/23.556880
31. Von Neumann J (1956) Probabilistic logics and synthesis of reliable organisms from unreliable components. In: Shannon C, McCarthy J (eds) Automata studies. Princeton University Press, Princeton, pp 43–98
32. Mahmood A, McCluskey EJ (1988) Concurrent error detection using watchdog processors—a survey. IEEE Trans Comput 37:160–174. doi:10.1109/12.2145
33. Austin TM (1999) DIVA: a reliable substrate for deep submicron microarchitecture design. In: Proceedings of the 32nd annual ACM/IEEE international symposium on microarchitecture, MICRO-32, IEEE Computer Society, pp 196–207. doi:10.1109/MICRO.1999.809458
34. Reed IS, Solomon G (1960) Polynomial codes over certain finite fields. J Soc Ind Appl Math 8:300–304. doi:10.1137/0108018
35. Johnson BW (1989) Design and analysis of fault-tolerant systems for industrial applications. In: Görke W, Sörensen H (eds) Fault-tolerant computer systems, vol 214, pp 57–73. doi:10.1007/978-3-642-75002-1_5
36. Martínez-Álvarez A, Restrepo-Calle F, Vivas Tejuelo LA, Cuenca-Asensi S (2013) Fault tolerant embedded systems design by multi-objective optimization. Expert Syst Appl 40:6813–6822
37. Nicolaidis M (1999) Time redundancy based soft-error tolerance to rescue nanometer technologies. In: Proceedings of the 17th IEEE VLSI test symposium (Cat. No. PR00146), IEEE Computer Society, pp 86–94. doi:10.1109/VTEST.1999.766651
38. Goloubeva O, Rebaudengo M, Reorda MS, Violante M (2006) Hardening the control flow. In: Software-implemented hardware fault tolerance. Springer, New York, pp 63–116. doi:10.1007/0-387-32937-4
39. Benso A, Carlo SD, Natale GD, Prinetto P, Tagliaferri L (2001) Control-flow checking via regular expressions. In: Proceedings of the 10th Asian test symposium, IEEE, pp 299–303. doi:10.1109/ATS.2001.990300
40. Hayes JP, Murray BT. (n.d.) Low-cost on-line fault detection using control flow assertions. In: Proceedings of the 9th IEEE on-line test symposium 2003. IOLTS 2003, IEEE Computer Society, pp 137–143. doi:10.1109/OLT.2003.1214380
41. Goloubeva O, Rebaudengo M, Reorda MS, Violante M (2003) Soft-error detection using control flow assertions. In: Proceedings of the 16th IEEE symposium Comput. Arith., IEEE Computer Society, pp 581–588. doi:10.1109/DFTVS.2003.1250158
42. Oh N, Shirvani PP, McCluskey EJ (2002) Control-flow checking by software signatures. IEEE Trans Reliab 51:111–122. doi:10.1109/24.994926
43. Avizienis A (1985) The N-version approach to fault-tolerant software. IEEE Trans Softw Eng SE-11:1491–1501. doi:10.1109/TSE.1985.231893
44. Jochim M (2002) Detecting processor hardware faults by means of automatically generated virtual duplex systems. In: Proceedings of the international conference on dependable systems and networks, IEEE Computer Society, pp 399–408. doi:10.1109/DSN.2002.1028925
45. Oh N, Mitra S, McCluskey EJ (2002) ED/sup 4/I: error detection by diverse data and duplicated instructions. IEEE Trans Comput 51:180–199. doi:10.1109/12.980007
46. Oh N, McCluskey EJ (2002) Error detection by selective procedure call duplication for low energy consumption. IEEE Trans Reliab 51:392–402. doi:10.1109/TR.2002.804735
47. Rebaudengo M, Sonza Reorda M, Torchiano M, Violante M (1999) Soft-error detection through software fault-tolerance techniques. In: Proceedings of the 1999 IEEE international symposium on defect fault tolerance VLSI Systems, IEEE Computer Society, pp 210–218. doi:10.1109/DFTVS.1999.802887
48. Rebaudengo M, Reorda MS, Violante M, Torchiano M (2001) A source-to-source compiler for generating dependable software. In: Proceedings of the 1st IEEE international workshop on

source code analysis and manipulation, IEEE Computer Society, pp 33–42. doi:10.1109/SCAM.2001.972664
49. Reis GA, Chang J, Vachharajani N, Rangan R, August DI (2005) SWIFT: software implemented fault tolerance. In: Proceedings of the international symposium on code generation and optimization, IEEE, pp 243–254. doi:10.1109/CGO.2005.34
50. Chang J, Reis GA, August DI (2006) Automatic instruction-level software-only recovery. In: Proceedings of the international conference on dependable systems and networks, IEEE, pp 83–92. doi:10.1109/DSN.2006.15
51. Reis GA, Chang J, Vachharajani N, Rangan R, August DI, Mukherjee SS (2005) Software-controlled fault tolerance. ACM Trans Archit Code Optim 2:366–396. doi:10.1145/1113841.1113843
52. Reis GA, Chang J, Vachharajani N, Rangan R, August DI, Mukherjee SS (2005) Design and evaluation of hybrid fault-detection systems. In: Proceedings of the 32nd international symposium on computer architecture, IEEE, pp 148–159. doi:10.1109/ISCA.2005.21
53. Mukherjee SS, Kontz M, Reinhardt SK (2002) Detailed design and evaluation of redundant multi-threading alternatives. In: Proceedings of the 29th annual International symposium on computer architecture, IEEE Computer Society, pp 99–110. doi:10.1109/ISCA.2002.1003566
54. Bernardi P, Bolzani LMV, Rebaudengo M, Reorda MS, Vargas FL, Violante M (2006) A new hybrid fault detection technique for systems-on-a-chip. IEEE Trans Comput 55:185–198. doi:10.1109/TC.2006.15
55. Bernardi P, Sterpone L, Violante M, Portela-Garcia M (2006) Hybrid fault detection technique: a case study on Virtex-II Pro's PowerPC 405. IEEE Trans Nucl Sci 53:3550–3557. doi:10.1109/TNS.2006.886221
56. Bernardi P, Bolzani Poehls L, Grosso M, Sonza RM (2010) A hybrid approach for detection and correction of transient faults in SoCs. IEEE Trans Dependable Secure Comput 7:439–445. doi:10.1109/TDSC.2010.33
57. Bernardi P, Bolzani L, Reorda MS (2007) A hybrid approach to fault detection and correction in SoCs. In: Proceedings of the 13th IEEE international on-line test symposium (IOLTS 2007), IEEE, pp 107–112. doi:10.1109/IOLTS.2007.8
58. Rebaudengo M, Reorda MS, Violante M, Nicolescu B, Velazco R (2002) Coping with SEUs/SETs in microprocessors by means of low-cost solutions: a comparison study. IEEE Trans Nucl Sci 49:1491–1495. doi:10.1109/TNS.2002.1039689
59. Azambuja JR, Lapolli Â, Rosa L, Kastensmidt FL (2011) Detecting SEEs in microprocessors through a non-intrusive hybrid technique. IEEE Trans Nucl Sci 58:993–1000. doi:10.1109/TNS.2011.2109398
60. Azambuja JR, Souza F, Rosa L, Kastensmidt F (2010) Non-intrusive hybrid signature-based technique to detect SEU and set faults in microprocessors. In: Proceedings of the 11th European conference on radiation and its effects on components and systems RADECS 2010, Längenfeld
61. Li X, Gaudiot J-L (2009) Tolerating radiation-induced transient faults in modern processors. Int J Parallel Prog 38:85–116. doi:10.1007/s10766-009-0114-9
62. Scholzel M (2010) HW/SW co-detection of transient and permanent faults with fast recovery in statically scheduled data paths. In: Proceedings of the 2010 design automation and test in Europe conference exhibition (DATE 2010), IEEE, pp 723–728. doi:10.1109/DATE.2010.5456957
63. Lee J, Shrivastava A (2010) A compiler-microarchitecture hybrid approach to soft error reduction for register files. IEEE Trans Comput Des Integr Circuits Syst 29:1018–1027. doi:10.1109/TCAD.2010.2049050
64. Parra L, Lindoso A, Portela M, Entrena L, Restrepo-Calle F, Cuenca-Asensi S et al (2014) Efficient mitigation of data and control flow errors in microprocessors. IEEE Trans Nucl Sci 61:1590–1596. doi:10.1109/TNS.2014.2310492
65. Martínez-Álvarez A, Restrepo-Calle F, Cuenca-Asensi S, Reyneri LM, Lindoso A, Entrena L (2012) A hybrid technique for soft error mitigation in interrupt-driven applications. In:

Proceedings of the 13th European conference on radiation and its effects components and systems RADECS 2012, Biarritz
66. Altieri M, Becker J, Kastensmidt FL (2013) HETA: hybrid error-detection technique using assertions. IEEE Trans Nucl Sci 60:2805–2812. doi:10.1109/TNS.2013.2246798
67. Portela-Garcia M, Grosso M, Gallardo-Campos M, Sonza Reorda M, Entrena L, Garcia-Valderas M et al (2012) On the use of embedded debug features for permanent and transient fault resilience in microprocessors. Microprocess Microsyst 36:334–343. doi:10.1016/j.micpro.2012.02.013

Chapter 17
Reducing Implicit Overheads of Soft Error Mitigation Techniques Using Selective Hardening

Felipe Restrepo-Calle, Sergio Cuenca-Asensi, and Antonio Martínez-Álvarez

Abstract The use of COTS FPGAs as deployment platform of microprocessor based systems represents an attractive alternative on aerospace applications, because their programmability, performance and cost-effectiveness. However, traditional hardening has a remarkable impact on resources and performance that limits their applicability. Selective hardening, that is protecting only the design's most error-sensitive parts, reduces significantly overheads keeping a reasonable reliability at the same time. This chapter describes and illustrates, with experimental results, this method and presents a hybrid strategy, called co-hardening, to leverage the benefits of adopting selective hardening on both hardware and software.

17.1 Introduction

During last decades, scientific and industrial concerns about radiation effects on electronic components have increased significantly. It is now well-known that these effects can affect the components operation permanently (permanent faults) or temporary (transient faults) [1]. In particular, transient faults, the so-called soft errors, affect the component behavior temporarily, affecting digital signal transfers on the circuit combinational logic (Single Event Transient—SET) or stored values on the circuit sequential logic (Single Event Upset—SEU) [2].

Commercial-Off-The-Shelf (COTS) electronic components (including FPGAs) are highly sensitive to radiation-induced effects, particularly soft errors, which limit their applicability in the near future. Consequently soft error mitigation has become

F. Restrepo-Calle (✉)
Department of Systems and Industrial Engineering, Universidad Nacional de Colombia, Bogotá, Colombia
e-mail: ferestrepoca@unal.edu.co

S. Cuenca-Asensi • A. Martínez-Álvarez
Department of Computer Technology, University of Alicante, Alicante, Spain
e-mail: sergio@dtic.ua.es; amartinez@dtic.ua.es

F. Kastensmidt, P. Rech (eds.), *FPGAs and Parallel Architectures for Aerospace Applications*, DOI 10.1007/978-3-319-14352-1_17

a mandatory requirement for the system to leverage the important benefits provided by the combination of COTS FPGAs and soft-cores.

Besides costly technological solutions to cope with this problem, system designers and researchers have proposed fault mitigation approaches based on the design of the system. These are based on hardware [3], software [4], or hardware/software [5] considerations. They are mainly aimed at designing fault detection/recovery mechanisms by applying redundancy on hardware, software, time, information, etc. [6]. In addition, FPGAs are excellent platforms to design and deploy fault-tolerant soft core based systems taking into account that their plasticity (on both hardware and software) permits to explore several trade-offs between hardware and software protection strategies [7].

Although many of the design-based approaches provide an effective solution to the transient faults, in general, these techniques cause non-negligible overheads to the systems. The impact of the hardware-based hardening approaches is mostly related to the increase of used resources, power consumption, die size, design time, and economic costs; whereas overheads of software-based hardening techniques are associated with the increase of the execution time, data and code size of programs [8]. In either case, this may prevent the applicability and feasibility of this kind of protection strategies in several application domains.

Moreover, recent hybrid hardware/software approaches have shown promising results in terms of fault detection/recovery rates. These techniques combine software redundancy with additional hardware support [5, 9–12]. However, the combination of fully implemented hardening techniques (hardware and software) could result in an over-redundant design with some unacceptable features such as large area and power costs, and disproportionate penalties in performance [13].

In this context, it is necessary to propose reduced-overhead fault mitigation schemes. Recent works pursuit to reduce the implicit overheads of the protection mechanisms by applying them in a selective way. That is, on the hardware side, adding protection only to the most vulnerable hardware parts [14], reducing the performance degradation by applying partial redundant threading [15, 16]; and on the software side, protecting only specific parts of the program code or the microprocessor architectural resources (reachable from the instruction set architecture—ISA) by means of redundant software [17, 18].

This chapter presents an overview of selective hardening techniques based on software and hardware. In addition, a methodology to apply these selective approaches jointly is presented as well, which is called *co-hardening*. It applies the co-design principles to design a customized hybrid strategy, which is based on the combined, selective, and incremental application of software and hardware techniques. In this way, this chapter will show how it is possible to design dependable embedded systems with reduced overheads, which not only satisfy dependability requirements and design constraints, but also avoid the excessive use of costly protection mechanisms in terms of hardware and software.

The rest of this chapter is organized as follows. Next section provides background information on selective hardening based on software and presents the

selective fault tolerance approach called S-SWIFT-R. Section 17.3 focuses on selective hardening approaches based on hardware. Section 17.4 presents the *co-hardening* methodology. Finally, Sect. 17.5 summarizes some final remarks and suggests directions for future works.

17.2 Selective Hardening Based on Software

Given the current rise of processor based systems and the need for dependable low-cost solutions, several fault mitigation techniques based on redundant software have been proposed. These techniques can be applied to both hard-cores and soft-cores since in no case the modification of the underlay hardware is needed. The so-called *Software Implemented Hardware Fault Tolerance* (SIHFT) [6] techniques are classified in two main categories according to the type of error they pretend to detect/correct: errors that may affect the program data [19]; or errors that may affect the control flow execution [20]. However, as mentioned above, the main limitations of this kind of approaches are the non-negligible overheads that they cause to the system. In many cases the performance degradation and/or the increase of the program code and data size affect severely the applicability of these proposals.

To reduce these overheads, recent works have proposed the selective hardening based on software [17, 18, 21, 22]. Instead of fully applying the protection approach to the program, several redundancy mechanisms are applied only to a selection of the program code. This strategy is aimed at hardening specific critical subroutines or a subset of the ISA-accessible microprocessor resources by means of redundant software.

Besides the overhead reduction, the main advantage offered by selective software-based techniques is flexibility. Designers are provided with a wide spectrum of alternatives, being able to explore deeply the design space on the software side, taking into account factors such as code overhead, performance degradation, and reliability level. In case that applying a particular set of hardening routines results inconvenient according to the requirements of an application (e.g., maximum execution time is exceeded), the technique can be applied partially depending on the critical program resources or sections. In short, the designer is able to fine-tune a tailored fault mitigation strategy based on software.

A few works based on selective hardening on software propose the selective instruction replication to guarantee the application-level correctness in multimedia applications [21, 22]. This kind of applications can tolerate, in some cases, an execution which is not 100 % numerically correct, and the program results can still appear to be correct from the user perspective [23]. In mission-critical systems, however, applications require the architecture-level correctness. More recent proposals are working on that direction [17, 18], applying selective hardening on software for the detection and recovery of data-flow errors.

17.2.1 Selective SWIFT-R

S-SWIFT-R stands for *Selective-SoftWare Implemented Fault Tolerance-Recovery* [17]. It is based on the SWIFT-R technique [35], which is a software-only recovery approach based on low-level instruction transformation rules (assembly code). This fault tolerance technique addresses the protection of the data stored in the register file, which is one of the most critical parts in processor-based applications. It intertwines three copies of the program and adds majority voting before critical instructions, based on the well-known *Triple Modular Redundancy* (TMR). In short, SWIFT-R consists of the triplication of data and instructions, jointly with the insertion of verification points to check data consistency (by means of majority voters). Based on this concept, S-SWIFT-R is a selective technique that allows applying the protection to different register subsets from the microprocessor register file looking for a reduction in the overheads, but keeping high fault coverage and offering more flexibility to designers.

Figure 17.1 illustrates an example of a simple program (assembly code) hardened using the original SWIFT-R. Note that two copies (e.g., s0′, s0″) are created for each register, which are stored in other available registers from the processor register file, i.e., unused registers in the program. Moreover, majority voters are recovery procedures that compare the correspondence of at least two registers, correcting the third copy if necessary (possibly corrupted).

As can be seen *2n* additional registers are necessary to fully implement SWIFT-R (where *n* is the number of used registers by the non-hardened program). This fact makes that SWIFT-R may not result suitable in many application domains where limited processors are used. Furthermore, due to its fault recovery capabilities, SWIFT-R produces high overheads that, regarding the application, can easily surpass 3× the original code size and execution time.

Line	*Non-hardened code*	*SWIFT-R code*
1	`main: LOAD s0, 00`	`main: LOAD s0, 00`
2		***Create s0 copies***
3	`LOAD s1, 2A`	`LOAD s1, 2A`
4		***Create s1 copies***
5	`ADD s0, s1`	`ADD s0, s1`
6		`ADD s0', s1'`
7		`ADD s0'', s1''`
8	`CALL incr`	`CALL incr`
9		***Majority voter for s0***
10	`STORE s0, 00`	`STORE s0, 00`
11	`RETURN`	`RETURN`
12		
13	`incr: LOAD s2, 0F`	`incr: LOAD s2, 0F`
14		***Create s2 copies***
15	`ADD s0, s2`	`ADD s0, s2`
16		`ADD s0', s2'`
17		`ADD s0'', s2''`
18	`RETURN`	`RETURN`

Fig. 17.1 Example of a simple program hardened using SWIFT-R

S-SWIFT-R proposes several improvements to the original technique to increase its flexibility and make it suitable for reduced-overhead embedded systems. As its predecessor, it is applied by means of low-level instruction transformation rules, but the strategy consists of applying software protection mechanisms only to some selectively chosen registers from the microprocessor register file. Prior to this selective proposal, the alternatives were only two, whether the use of the non-hardened program or the use of the fully hardened version. Now design space is enriched with several new possibilities, which offer more flexibility to designers, and facilitate to find the best trade-offs among reliability, performance, and code size. Furthermore, S-SWIFT-R can be useful in cases when is not possible to apply SWIFT-R completely, for instance, due to the limitations of the microprocessor (e.g., a low number of registers available in the register file, reduced space in program memory, …), or high resources utilization in the program (e.g., if the non-hardened code uses most registers available in the register file and, therefore, there are not enough available registers to create the required redundant copies). In these cases, it is possible to prioritize the registers depending of their impact of overheads and/or reliability to protect only a subset of them.

To implement this selective approach, the concept of *Sphere of Replication* (SoR) [36] is used in a flexible way. SoR defines the logic domain of redundant execution, which means that the architectural resources located within it are considered to have redundant mechanisms; consequently, they are protected against faults. Thus, the SoR delimits the protection coverage of hardening techniques. Moving the borders of the SoR, it is possible to modify the protection level of different fault tolerance techniques by including or excluding various components inside the sphere (i.e. different subsets of register file or memory subsystem).

Instructions causing a data flow crossing through the sphere frontiers must be handled in a special way. To do so, in S-SWIFT-R all instructions whose execution imply a data flow crossing the borders of the SoR are classified in a special manner. In case only the register file is located inside the SoR, when an instruction causes that some data enter inside the SoR (e.g., reading an input port, loading a value into a register or reading a value from memory), it is classified as *inSoR*. In contrast, when an instruction provokes data to go out from the SoR (e.g., writing to an output port, storing a value into the memory), it is classified as *outSoR*. Otherwise, instructions whose execution do not imply a data flow (e.g., an unconditional branch) are classified as *none*.

The algorithm to apply S-SWIFT-R to a given source code (assembly code) can be summarized as follows:

1. Define the components to protect, i.e., these will be considered to be inside the SoR.
2. Classify each program instruction accordingly to the direction of the data flow it provokes with regard to the SoR (*inSoR*, *outSoR*, *none*).
3. Triplicate data the first time that any data enter to the SoR. That is, for each instruction classified as *inSoR*, two additional copies of the data entering to the sphere will be created. These redundant copies are created by copying the register values, avoiding repeating memory or input port accesses.

4. Triplicate instructions that perform any data operation (e.g., arithmetic, logic, shift, rotation instructions). Notice that redundant instructions should operate using register copies (replicated data).
5. Insert majority voters and recovery procedures at several key points:
   (a) Before *outSoR* instructions: to verify the correctness of the data involved in the instructions classified as *outSoR* before their execution. This is necessary to avoid erroneous data leaving the sphere, because once the data have left the SoR, recovery will be not possible, and the corrupted data may cause a system error.
   (b) Before the last operation prior to a conditional branch: this instructions may alter the ALU flags (zero, carry, …). This verification is necessary because if a register value is corrupted, an operation using this register may produce an erroneous resultant flag, and consequently, this may provoke an incorrect branch somewhere in the program's control flow graph after the conditional branch execution.
6. Release redundant registers (copies) if they are not needed anymore in the rest of the program; otherwise, copies should be kept along the program execution.

Contrarily to the original SWIFT-R that considers the whole register file included in SoR, the selective version consists in moving out of the SoR the registers that are not required to be protected, while some other registers remain within the SoR and, consequently, code transformations are responsible for protecting only this subset of registers. To illustrate the approach, Fig. 17.2 shows an example with different versions of a basic program hardened using S-SWIFT-R applied to several register subsets. Notice that the fully hardened version obtained by S-SWIFT-R, i.e., the version with protection in all the used registers ('s0 and s1 protected'), is the same than the one obtained by the original SWIFT-R approach.

As can be seen in Fig. 17.2, in the version 'register s1 protected', only the s1 register is considered within the SoR. The instruction `ADD s0, s1` (s0=s0+s1),

#	*Non-hardened*	*Protected register:* `s0`	*Protected register:* `s1`	*Protected registers:* `s0, s1`
1	`LOAD s0, 00`	`LOAD s0, 00`	`LOAD s0, 00`	`LOAD s0, 00`
2		***Create s0 copies***		***Create s0 copies***
3	`LOAD s1, 2A`	`LOAD s1, 2A`	`LOAD s1, 2A`	`LOAD s1, 2A`
4			***Create s1 copies***	***Create s1 copies***
5			***Voter for s1***	
6	`ADD s0, s1`	`ADD s0, s1`	`ADD s0, s1`	`ADD s0, s1`
7		`ADD s0', s1`		`ADD s0', s1'`
8		`ADD s0'', s1`		`ADD s0'', s1''`
9		***Voter for s0***		***Voter for s0***
10			***Voter for s1***	***Voter for s1***
11	`STORE s0, (s1)`	`STORE s0, (s1)`	`STORE s0, (s1)`	`STORE s0, (s1)`

Fig. 17.2 Example of hardened program using S-SWIFT-R (several versions)

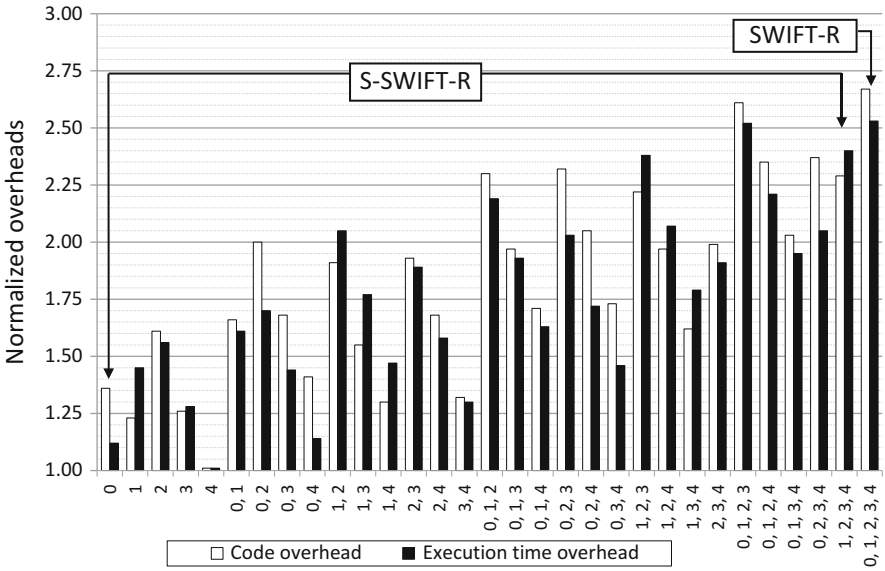


Fig. 17.3 Normalized code size and execution time overheads using S-SWIFT-R for FIR code

line 6, is classified as *outSoR* since its execution provokes a data flow from inside the SoR (s1) going outward (s0); therefore, a majority voter should be inserted to verify the correctness of the value stored in s1, before it leaves the redundancy domain.

To show an example of the flexibility of S-SWIFT-R, Fig. 17.3 presents the overhead results for all the variations of the selective protection applied to a single test program: *Finite Impulse Response* (FIR) filter running on a *PicoBlaze* soft-core. Static code size overhead and execution time overhead are normalized with a baseline built with the non-hardened version of the program. In X axis all software versions are represented. These are named with the number of the registers protected, i.e. "0, 2" correspond to the version where only registers 0 and 2 are protected.

Note that overheads increase incrementally when more registers are protected. In this case, code overhead varies from 1.01× (in the "4" version) to 2.67× in the fully protected version, and execution time overhead ranges from 1.01× (in the "4" version) to 2.53× (in the SWIFT-R version).

As a matter of fact, more resources (code lines, data, execution time) are required when more protection is implemented (more registers are protected). However, it is very important to note two additional considerations related to the contribution to the overheads of each register when protected. Firstly, each register makes its contribution to code overhead and execution time overhead independently. For example, the "0" version causes a considerable code overhead (1.36×) while its execution time overhead is only 1.12×. Secondly, different registers make their contribution to overheads in different manners. There are versions in which the protection of some registers causes an almost negligible impact, such as in the "4"

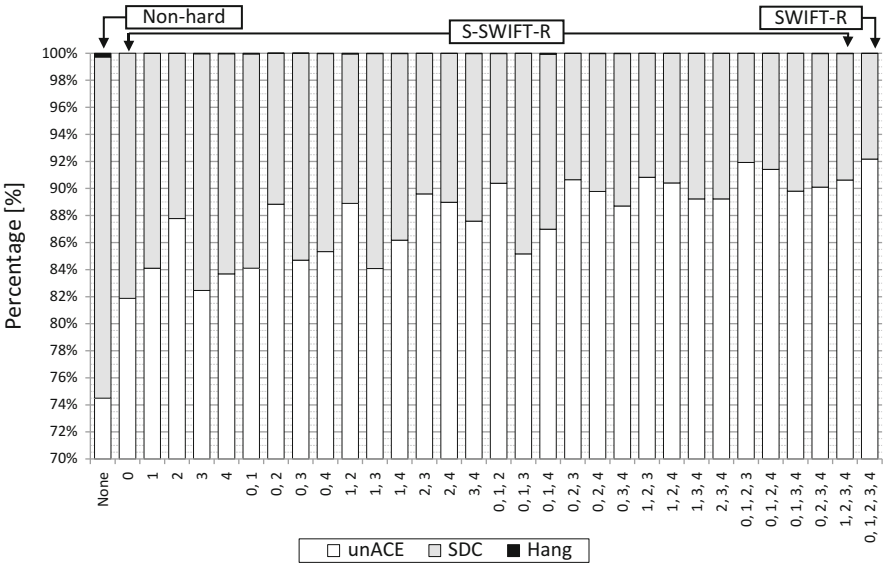


Fig. 17.4 Fault classification percentages for every selective hardened version of FIR

version (code size and execution time overheads are both 1.01×), while there are other versions in which the protection of only one register can provoke a high impact, like in the "2" version (code overhead 1.61× and execution time overhead 1.56×).

To evaluate the fault coverage provided by S-SWIFT-R, a fault injection campaign was carried out for each version of the system using FTUnshades (using the real implementation of the different systems) [37]. For each hardened version of the program, the fault injection campaign consisted of injecting 80,000 faults (SEUs), emulating only one single fault per program execution. Each fault was emulated by means of a single bit-flip in a randomly selected bit from the microprocessor, including: register file (16-byte-wide registers), program counter, stack pointer, ALU flags, and pipeline registers. Each fault was injected in a randomly selected clock cycle from all the workload duration. Injected faults were classified according to their effect on the expected system behavior as follows:

- If the system completes its execution, and obtains the expected output, the memory element (bit) affected by the fault and, consequently the fault itself, are classified as *unnecessary for Architecturally Correct Execution*—unACE.
- In case the fault was not detected/corrected and provokes the program terminates with an erroneous output, this fault is called *Silent Data Corruption*—SDC.
- If the fault causes an abnormal program termination or an infinite execution loop, the fault is categorized as a *Hang*. Note that SDC and Hang are both undesirable effects (categorized together as ACE faults).

Figure 17.4 presents the fault classification percentages obtained for each software version in the fault injection experiments.

One can observe the remarkable increase in the fault coverage that it is obtained using S-SWIFT-R considering the complete microprocessor. Fault coverage goes from 74.50 to 92.17 % unACE faults. These results represent the percentages of injected faults that do not provoke any undesirable behavior to the circuit operation. However, this increase is even more notorious if we consider only the microprocessor register file. In this case, fault coverage ranges between 79.19 and 99.26 % unACE.

In addition, there are several intermediate-protected versions that might be suitable for many applications depending on the requirements. For example, when the protection is applied to the registers 2 and 3 ("2, 3" version), fault coverage goes up to 89.60 % unACE faults, which is remarkable taking into account that only two registers are being hardened resulting in time and code overheads of 1.89× and 1.93× respectively. In the same manner that each register impacts the overheads independently when it is protected, each register contributes apart to the fault coverage improvements. This can be seen, for instance in the "0" version, in which the fault coverage is only 81.87 % unACE when only the register 0 is protected, whereas protecting only the register 2, this percentage goes up to 87.78 % unACE (a 5.91 % difference).

In many cases, the selective protected versions can be better candidates for systems where not only the fault coverage is important, but also the time execution. Protecting all registers, using a software technique, could result in the best fault coverage, but at the same time, it provokes the highest performance degradation. Hence, overheads and fault coverage results have to be studied jointly, representing several trade-offs among code size, performance, and fault coverage. This analysis guides the design decisions to find the solutions having the best reliability/overhead compromise. For instance, the "1, 2, 4" version is an interesting choice, because it offers both, high fault coverage (90.42 % unACE faults), and acceptable code size and execution time overheads (1.97× and 2.07×, respectively).

Moreover, it is worth mentioning that triplication of instructions imply the protection not only of the register data, but also of all datapaths where instructions pass through. Replicas of instructions will pass all pipeline paths, so these are indirectly protected as well. That is, the software protection not only covers the specified register subset but also many all components in the execution pipeline.

17.3 Selective Hardening Based on Hardware

Hardware methods for soft-error mitigation are a common topic in fault tolerance and there are a plethora of approaches in literature. These can be classified in two categories: technological-based techniques and design-based techniques. The firsts involve especial fabrication processes that require a great effort and investment, and consequently very few designs have adopted this approach. The second category is related to apply time and hardware redundancy at architectural level, for example Triple Modular Redundancy or hardened memory cells. These solutions offer a high reliability but, when apply to FPGA devices, a full redesign of the fabric is needed.

This is the case of Xilinx Virtex5QV device [24] specifically conceived for the space market. In Virtex5QV different design-based techniques have been used in selected basic elements: dual-node latches in configuration memory, triple modular redundancy in configuration and JTAG control logic, dual-node latches and transient filters in CLB user registers, embedded EDAC in integrated BlockRAM, etc.... The cost of these devices can be two orders of magnitude greater than its commercial-grade counterpart.

A less expensive approach is based on the application of hardware methods at high-level, that is at the HDL design, to protect the basic blocks of the user design. Among all methods Modular Redundancy has become a common practice at this level, because its versatility to be applied at different granularities. However, the MR technique presents some drawbacks because of its full hardware redundancy, such as area and power dissipation. To avoid these overheads, selective insertion of Triple Modular Redundancy has been proposed to protect only the nodes of the circuit that present a bigger vulnerability to SEUs [14]. Other approach proposes an automatic selective insertion of TMR, based on an iterative optimization method that assures the minimum possible area, in terms of protected registers, while meeting the reliability constrains specified for the circuit [25]. Additionally to registers and combinational logic, selective TMR has been also applied for redundancy of wires to prevent the distorted signal from propagating to an output or a storage element [26] and for protecting whole circuit sections that affect structures which cause a persistent error [27].

Apart to the hardening technique, the key point when TMR has to be selectively applied is how to figure out what are critical spots in the design that need to be protected. Authors have proposed several approaches, and at different levels, to estimate the vulnerability of gates and flip-flops in a circuit. In [28] authors perform an electrical analysis of the primitive cell library to determine gates susceptible to single-event transients (SETs). Also simplified electrical models have been used to determine the gates with the highest soft error rate (SER) for hardening [29].

Architectural Vulnerability Factor (AVF) is a metric widely spread for discriminating the most sensitive parts of a microprocessor [30]. In [31] AVF is estimated by means of Register Transfer Level simulations to rank the control state elements of a soft-core taking into account the high degree of architectural masking inherent in modern microprocessors. Unlike methods that compute the AVF based on performance models, his method operates at RTL and is, therefore, more accurate. Extensive fault injection campaign is the other approach to raise the accuracy of estimations but at the cost of prohibitive experimentation times. To overcome this problem specific FPGA based emulation tools has been developed. Using this kind of tool, authors in [32, 33] were able to rank the sensibility to SET and SEU of every gate and flip-flop of a PIC18 clone microprocessor during the execution of different workloads. They conclude, in case of SEU, that some microprocessor areas should be protected with independence of the application, meanwhile other parts of the circuit depends of the workload. Specifically for the tested workloads, they observe that hardening a 24 % of the flip-flops using TMR the failure rate obtained is lower than 1 %.

17.3.1 Selective TMR

The selective hardening of specific microprocessor parts can be used as a stand-alone technique or as a complement to the protection offered by other software techniques. In the second case, there are two key points where the participation of hardware techniques can significantly improve the fault tolerance strategy. On the one hand, the reduction of impact produced in performance due to protection of both application data and control-flow. Although selectiveness in data protection and its application at assembly level can alleviate the overheads problem, in the case of control flow, the complexity of methods may yield a high overhead in execution time. On the other hand, hardware redundancy allows a more efficient protection of control flow. In fact, the lack of clear criteria to prioritize the control flow of different sections of code makes the selection infeasible and compels to protect all the application. For example, the protection of the control flow of a specific function does not guarantee its correct execution since an error in a previous section of code can exclude this function from execution. Furthermore, the visibility and accessibility to the micro-architectural registers involved in the control flow of applications (i.e. Program Counter, ALU flags, etc…) may be limited by the microprocessor instruction set (ISA). For instance, some microprocessors do not expose the Program Counter and Status Register to the instruction set, and consequently the protection code is unable to observe or modify them.

As mentioned before, Triple Modular Redundancy is one of the most common hardware methods and its selective application to FPGAs can generate an important saving of scarce resources like flip-flops. To illustrate the benefits of selective protection we will analyze its application for mitigating the effect of SEUs in a technology-independent clone of Xilinx picoBlaze-3 soft-core. The following six versions of the soft core are evaluated:

- P0: Nonhardened
- P1: TMR in Program Counter (PC), Flags, and Stack Pointer (SP)
- P2: TMR in Pipeline registers
- P2f: TMR in Register file (only 5 registers)
- P3: TMR in PC, Flags, SP, and Pipeline
- P4: Full TMR protection PC, Flags, SP, Pipeline and Register File

Figure 17.5 shows at a glance the different trade-offs, between reliability and cost. During all the experiments the microprocessor executes the same workload: a Proportional-Integral-Derivative controller (PID), and the setup is similar to that described in Sect. 17.2.1. On the left axis, the hardware cost of the micro is expressed in terms of combinational logic and flip-flops reported a synthesis tool (Xilinx XST v10.1). The left axis depicts the fault coverage of each version in terms of unACE bits. These numbers are normalized with respect to the baseline version (P0). As can be seen, the full protected version reaches to 100 % of fault coverage but increasing the hardware cost up to 2.92× flip-flops and latches, and 1.93× combinational logic. When selective TMR is applied (P1, P2 and P3) the cost rises moderately and

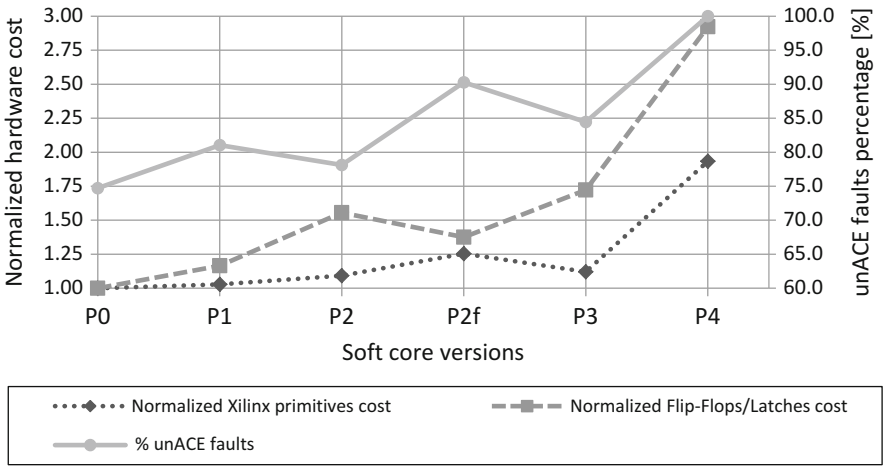


Fig. 17.5 Normalized hardware cost and percentage of unACE bits per soft core version

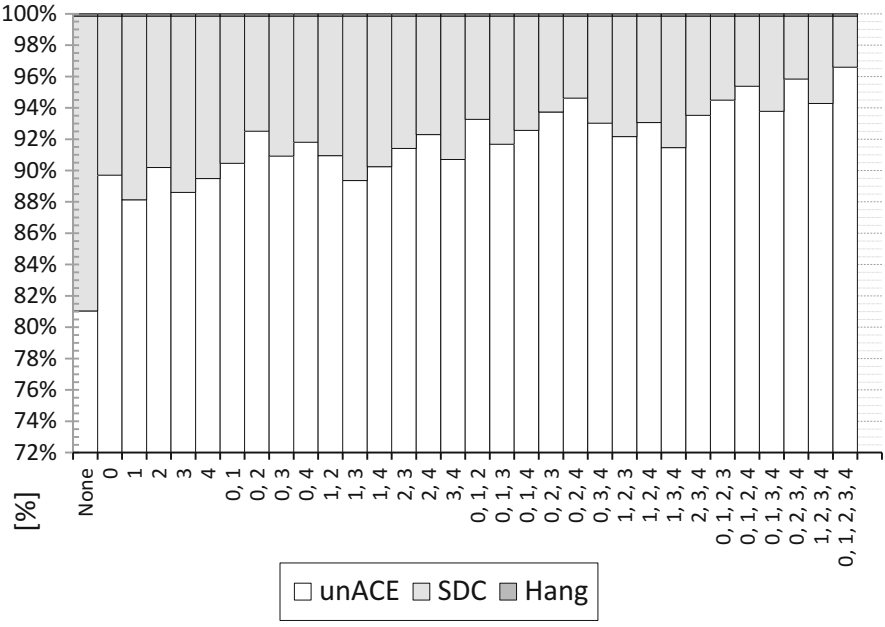


Fig. 17.6 Fault classification percentages for P1 version with different subsets of registers hardened

remains below 1.75× for flip-flops and 1.25× for combinational logic in all the cases. From this coarse-grain exploration the contribution to reliability of every register subset can be deduced. Flags, SP and PC contribute with an improvement of 6.3 % meanwhile Pipeline registers only reach a 4 %. The larger increment, a 15.3 %, is produced when the five register of the Register File used in the code are protected. Figure 17.6 completes the study with a fine-grain exploration of

these registers. In this P1 version incorporates S-TMR to different subsets of the register. As can be seen, protecting only PC, SP, Flags and one additional register ("3") the coverage is above 90 %. With two additional registers ("0, 2") the unACE is 92.5 % and selecting "0, 2, 4" 94.6 %.

17.4 Co-hardening: Co-design of Selective Hardware/ Software Fault Mitigation Techniques

Co-hardening is a methodology that tries to reduce protection overheads complementing software mitigation techniques with hardware techniques in a selective way. For this purpose it is necessary to perform a fine-grained exploration of the design space by means of the selectively controlled application of protection approaches on both sides: software and hardware. This controlled selectiveness consists of protecting the most critical parts of the system on each side. Furthermore, designers should choose where the protection will be applied, whether to software or to hardware, taking into account that this selection will affect the system overheads in a different manner. In this way, designers are able to fine-tune a tailored fault mitigation hybrid approach to achieve a dependable solution, which not only best meets the design constraints and dependability requirements of the application, but also avoids the excessive use of costly hardening artifacts.

This strategy is especially useful in the design of critical soft-core based embedded systems as they offer the necessary plasticity and flexibility on both sides: hardware and software. Thus, the final deployment platform for the design could be an ASIC or an FPGA. However, it should be noted that, in case of SRAM based FPGAs, additional mechanisms are required for the protection of the configuration memory.

The general *co-hardening* strategy may result in a large amount and concentration of possibilities which give the required flexibility to designers, but at the same time, it could complicate the design space exploration due to the same reasons. In the majority of cases the exhaustive exploration of the different solutions is impracticable in terms of time and costs of design, implementation, and evaluation. This is especially true when selective/partial protection approaches are being considered. Therefore, it is necessary to reduce the exploration area to converge rapidly to an optimal solution in the design space. For this reason, the proposed *co-hardening* design flow is not based on an exhaustive exploration, but a design flow directed by the application. In this way, the *co-hardening* prioritizes the fault mitigation based on software techniques and then, if necessary, the protection approach is complemented with additional hardware mechanisms. Figure 17.7 illustrates the *co-hardening* design flow.

The first step is the specification of system requirements. These include design constraints and dependability requirements. In general, design constraints are related to silicon area, performance, power consumption and hardware cost; whereas, dependability parameters are concerned with fault detection/recovery rate,

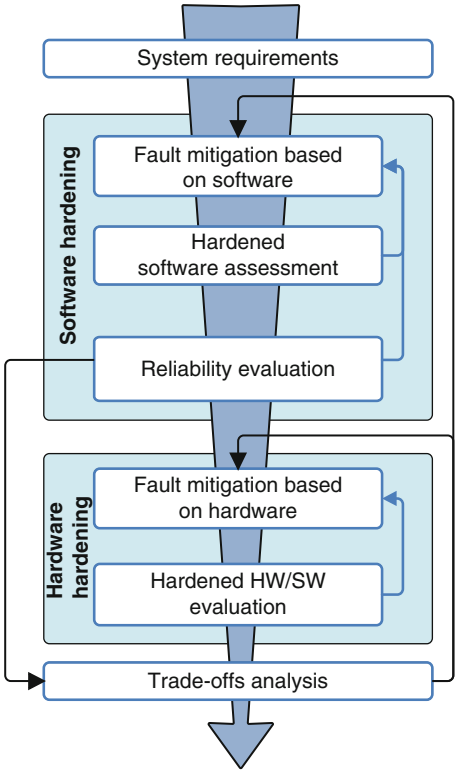


Fig. 17.7 Co-hardening design flow

reliability, availability, safety, security, and recovery time. As the design flow is driven by the application, design decisions must be motivated taking into account both, design constraints and dependability parameters.

The adoption of several software fault mitigation techniques can determine a set of suitable implementations of the software of the system. Software techniques can be fully or selectively applied. At this point, every software version is functionally equivalent to its original, with variations in the redundancy level, and possibly, in the location of the protection. These software versions are then evaluated to estimate the caused overheads in comparison to the non-hardened program in terms of code size, data, and execution time. In addition, in case there was available a simulation-based reliability evaluation tool, such as [34], it could be used to make preliminary dependability analyses of the several versions.

Based on the evaluation results, and according to the specifications, the best candidates are selected to be tested on the real microprocessor implementation. It is necessary to evaluate the reliability offered by each hardware/software configuration. To do so, evaluation tools such as fault emulation platforms based on FPGAs might be used [37]. At this point, designers can explore several trade-offs among code size, performance, and reliability. This software hardening process is iterative because it could be required to fine-tune some of the program versions.

In case that still none of the evaluated configurations meets all the requirements, the protection strategy must be complemented by applying hardware-based techniques. Thus, designers should study suitable strategies to selectively protect the hardware, specifically looking for protecting the most vulnerable parts of the design and inserting redundancy selectively in those parts of the microprocessor where software-based techniques cannot do it. As a result, a new parameter should be considered within the trade-offs analysis: hardware cost (in terms of area, power consumption, and economic costs).

After combining the best candidates on the software side with the protected versions of the hardware, the hybrid fault mitigation approach is then evaluated in terms of reliability. The optimal point within the design space is not necessarily a single point but may result in a set of suitable hardware/software configurations that meet the application requirements in terms of design constraints and dependability. At this point, designers have sufficient information to select the best system configuration based on the trade-off analysis.

The complete co-hardening process can be iterative. In case that as a result of the trade-off analyses one finds that none of the hardware/software configurations fully meet the system requirements. This means that the strategy still requires continuing being fine-tuned.

17.4.1 Co-hardening Case Study: S-SWIFT-R + S-TMR

As a case study, a hybrid fault mitigation strategy has been designed which combines the S-SWIFT-R technique on the software side with S-TMR on the hardware side. The target application is a FIR filter software (same as in Sect. 17.2) running in an RTL version of Picoblaze (same as in Sect. 17.3). The original version of microprocessor integrates a total of 197 flip-flops, therefore the number of different hardware configurations for selective hardening are really big. For demonstrative purposes only hardware configurations defined in Sect. 17.3 are considered (P0, P1, P2, P3 and P4 versions). On the contrary, all the combinations of the five ISA registers involved in the execution of FIR are taken into account.

Figure 17.8 illustrates the fault classification percentages obtained for each selectively hardened version of the software running on versions P1, and P3 of the processor. Each test campaign uses the same setup as in Sect. 17.2. Results for P0 version can be seen in Fig. 17.4, meanwhile P4 numbers are not showed because 100 % of injected faults were classified as unACE, as expected.

Consideration should be given to the fact that combining S-SWIFT-R with hardware protection applied to only a few critical registers, such as PC, ALU Flags, and SP (P1 version), reliability increases remarkably (up to 97.85 % unACE faults). Furthermore, obtained results for the P2 microprocessor (not shown), indicate that hardware redundancy on the pipeline does not improve the fault coverage of the system considerably (in the best case, achieving 93.85 % unACE faults), even though the amount of protected registers is by far higher than for the P1 microprocessor.

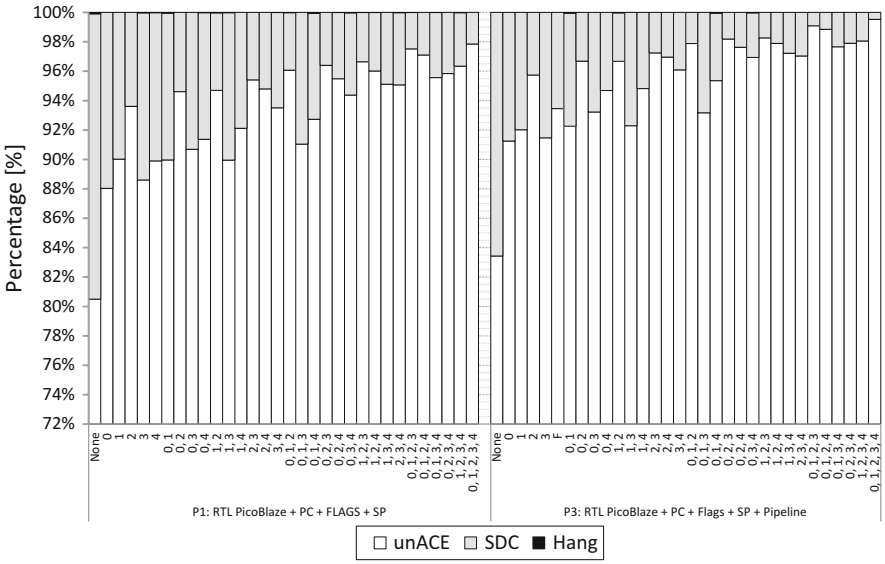


Fig. 17.8 Fault classification percentages for every software version running on different hardened microprocessors (P1 and P3)

However, hardware protection in both sets of registers are complementary and, therefore, the highest reliability levels are achieved by the P3 microprocessor (up to 99.52 % unACE faults), which brings together the hardware protection of both P1 and P2 versions.

As can be seen, the reliability increases when hardened programs are combined with protected hardware approaches. Nevertheless, the more hardware protection is implemented, the higher hardware overheads are. This is an important restriction that has to be considered in the co-hardening process.

The information gathered in this case study, permits to represent several trade-offs among performance, code size, reliability, and hardware cost. However, the previous analysis is missing to consider the program time overhead caused by the software technique. For this purpose it is necessary to take into account the metric known as Mean Work To Failure—MWTF, which captures the trade-offs between reliability and performance. Figure 17.9, on the one hand, shows the MWTF of the hybrid systems normalized to a baseline built with the non-hardened software/hardware version (represented in logarithmic scale); and on the other hand, it also depicts, in a secondary axis, the normalized hardware costs. Since the MWTF constitutes the balance between reliability and performance, this figure permits to see at a glance, the representation of several trade-offs among reliability, execution time and hardware costs for each one of the systems. Again the full hardware-protected microprocessor (P4) is not represented considering its high hardware cost over P0, 2.92× Flip-Flops and Latches, and 1.93× combinational logic.

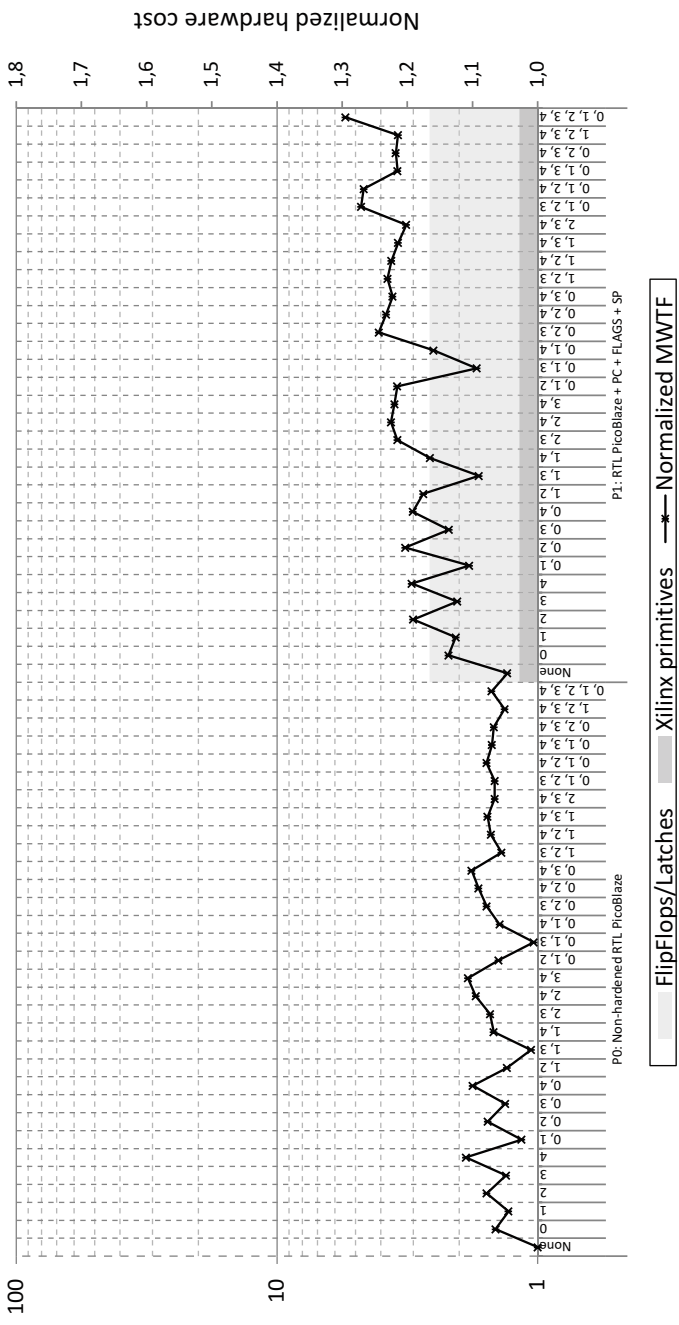


Fig. 17.9 Normalized MWTF and normalized hardware cost by hardware/software configuration

This figure clearly allows an in-deep study and exploration of the design space. Notice that even though the most remarkable increases on the MWTF are for the microprocessors running the software version with the "0, 1, 2, 3, 4" register subset hardened with S-SWIFT-R, there is also a wide set of partially protected system configurations that might result suitable for many applications. In this case study, for instance, it might result as a suitable configuration the system with the P1 microprocessor and S-SWIFT-R applied to the register subset 0-1-2-3 on the software side, because it offers a high increase in the mean work to failure (4.76 more than the non-protected system) with low hardware costs. In some other applications, with higher reliability requirements, the hardware cost should be increased, and the same software version could be chosen jointly with the P3 microprocessor, which presents a normalized MWTF of 12.92. Moreover, the full protected software version (0-1-2-3-4) running on the P3 microprocessor achieves a MWTF of 24.73 more than the non-hardened system.

17.5 Conclusions

Soft error mitigation is a key task in the development of reliable microprocessor based systems. FPGAs represent an interesting alternative since its reconfigurability allows the modification of the microarchitecture of the soft cores and the protection of software running on it. However different refinements of traditional fault tolerant techniques are needed to cope with the inherent overheads produced by them.

Selective hardening, that is the protection of selected hardware or software components depending of their error sensitivity, was presented and several approaches were reviewed. Results showed that high reliability can be reach at a fraction of the cost if appropriate selection is performed. To illustrate the procedure three techniques, one hardware-based (S-TMR), one software-based (S-SWIFT-R) and a combination of them, were evaluated. Some results were analyzed to show the possibilities and benefits of the methods. However, further research is needed to design new metrics and procedures that allow and efficient selection of the critical parts, avoiding the excessive time consuming fault injection campaigns.

Acknowledgment This work was funded in part by the Spanish Ministry of Education, Culture and Sports with the project "Developing hybrid fault tolerance techniques for embedded microprocessors" (PHB2012-0158-PC).

References

1. Baumann RC (2005) Radiation-induced soft errors in advanced semiconductor technologies. IEEE Trans Device Mater Reliab 5:305–316
2. Karnik T, Hazucha P, Patel J (2004) Characterization of soft errors caused by single event upsets in CMOS processes. IEEE Trans Dependable Secure Comput 1:128–143

3. Nicolaidis M (2005) Design for soft error mitigation. IEEE Trans Device Mater Reliab 5:405–418
4. Oh N, McCluskey EJ (2002) Error detection by selective procedure call duplication for low energy consumption. IEEE Trans Reliab 51:392–402
5. Bernardi P, Poehls LMB, Grosso M, Reorda MS (2010) A hybrid approach for detection and correction of transient faults in SoCs. IEEE Trans Dependable Secure Comput 7:439–445
6. Goloubeva O, Rebaudengo M, Reorda MS, Violante M (2006) Software-implemented hardware fault tolerance. Springer, New York
7. Cuenca-Asensi S, Martínez-Álvarez A, Restrepo-Calle F, Palomo FR, Guzmán-Miranda H, Aguirre MA (2011) Soft core based embedded systems in critical aerospace applications. J Syst Archit 57:886–895
8. Azambuja JR, Pagliarini S, Rosa L, Kastensmidt FL (2011) Exploring the limitations of software-based techniques in SEE fault coverage. J Electron Test 27:541–550
9. Azambuja JR, Lapolli Â, Rosa L, Kastensmidt FL (2011) Detecting SEEs in microprocessors through a non-intrusive hybrid technique. IEEE Trans Nucl Sci 58:993–1000
10. Hu J, Li F, Degalahal V, Kandemir M, Vijaykrishnan N, Irwin MJ (2009) Compiler-assisted soft error detection under performance and energy constraints in embedded systems. ACM Trans Embed Comput Syst 8:1–30
11. Lee J, Shrivastava A (2010) A compiler-microarchitecture hybrid approach to soft error reduction for register files. Comput Des 29:1018–1027
12. Ragel RG, Parameswaran S (2011) A hybrid hardware–software technique to improve reliability in embedded processors. ACM Trans Embed Comput Syst 10:1–16
13. Cuenca-Asensi S, Martínez-Álvarez A, Restrepo-Calle F, Palomo FR, Guzmán-Miranda H, Aguirre MA (2011) A novel co-design approach for soft errors mitigation in embedded systems. IEEE Trans Nucl Sci 58:1059–1065
14. Samudrala PK, Ramos J, Katkoori S (2004) Selective triple modular redundancy (STMR) based single-event upset (SEU) tolerant synthesis for FPGAs. IEEE Trans Nucl Sci 51:2957–2969
15. Reddy VK, Parthasarathy S, Rotenberg E (2006) Understanding prediction-based partial redundant threading for low-overhead, high-coverage fault tolerance. ACM SIGPLAN Not 41:83–94
16. Vera X, Abella J, Carretero J, González A (2010) Selective replication: a lightweight technique for soft errors. ACM Trans Comput Syst 27:8:1–8:30
17. Restrepo-Calle F, Martínez-Álvarez A, Cuenca-Asensi S, Jimeno-Morenilla A (2013) Selective SWIFT-R. J Electron Test 29:825–838
18. Chielle E, Azambuja JR, Barth RS, Almeida F, Kastensmidt FL (2013) Evaluating selective redundancy in data-flow software-based techniques. IEEE Trans Nucl Sci 60:2768–2775
19. Oh N, Shirvani PP, McCluskey EJ (2002) Error detection by duplicated instructions in super-scalar processors. IEEE Trans Reliab 51:63–75
20. Oh N, Shirvani PP, McCluskey EJ (2002) Control-flow checking by software signatures. IEEE Trans Reliab 51:111–122
21. Cong J, Gururaj K (2011) Assuring application-level correctness against soft errors. In: Proceedings of the IEEE/ACM international conference on computer-aided design (ICCAD), pp 150–157
22. Sundaram A, Aakel A, Lockhart D, Thaker D, Franklin D (2008) Efficient fault tolerance in multi-media applications through selective instruction replication. In: Proceedings of the 2008 workshop on radiation effects and fault tolerance in nanometer technologies, pp 339–346
23. Yeh TY, Reinman G, Patel SJ, Faloutsos P (2009) Fool me twice: exploring and exploiting error tolerance in physics-based animation. ACM Trans Graph 29:5:1–5:11
24. Xilinx (2010) Radiation-hardened, space-grade Virtex-5QV FPGA data sheet: DC and switching characteristics, data sheet DS692 (v1.0.1), Xilinx Inc.
25. Ruano O, Maestro JA, Reviriego P (2009) A methodology for automatic insertion of selective TMR in digital circuits affected by SEUs. IEEE Trans Nucl Sci 56(4):2091–2102

26. Almukhaizim S, Makris Y (2008) Soft error mitigation through selective addition of functionally redundant wires. IEEE Trans Reliab 57(1):23–31
27. Pratt B, Caffrey M, Carroll JF, Graham P, Morgan K, Wirthlin M (2008) Fine-grain SEU mitigation for FPGAs using partial TMR. IEEE Trans Nucl Sci 55(4, pt. 1):2274–2280
28. Mohanram K, Touba NA (2003) Partial error masking to reduce soft error failure rate in logic circuits. In: Proceedings of the 18th IEEE international symposium on defect and fault tolerance in VLSI Systems (DFTS 03), IEEE CS Press, pp 433–440
29. Nieuwland A, Jasarevic S, Jerin G (2006) Combinational logic soft error analysis and protection. In: 12th IEEE international on-line testing symposium (IOLTS 2006), Como, 10–12 July 2006
30. Mukherjee SS, Weaver C, Emer J, Reinhardt SK, Austin T (2003) A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In: International symposium on microarchitecture, pp 29–40
31. Maniatakos M, Makris Y (2010) Workload-driven selective hardening of control state elements in modern microprocessors. In: Proceedings of the 28th VLSI test symposium (VTS 10), IEEE CS Press, pp 159–164
32. Entrena L, Lindoso A, Valderas MG, Portela M, Ongil CL (2011) Analysis of SET effects in a PIC microprocessor for selective hardening. IEEE Trans Nucl Sci 58(3):1078–1085
33. Valderas M, Garcia MP, Lopez C, Entrena L (2010) Extensive SEU impact analysis of a PIC microprocessor for selective hardening. IEEE Trans Nucl Sci 57(4):1986–1991
34. Martínez-Álvarez A, Cuenca-Asensi S, Restrepo-Calle F, Palomo Pinto FR, Guzmán-Miranda H, Aguirre MA (2012) Compiler-directed soft error mitigation for embedded systems. IEEE Trans Dependable Secure Comput 9:159–172
35. Reis GA, Chang J, August DI (2007) Automatic instruction-level software-only recovery. IEEE Micro 27:36–47
36. Reinhardt SK, Mukherjee SS (2000) Transient fault detection via simultaneous multithreading. In: Proceedings of the 27th annual international symposium on computer architecture, pp 25–36
37. Napoles J, Guzman H, Aguirre M, Tombs J, Munoz F, Baena V, Torralba A, Franquelo L (2007) Radiation environment emulation for VLSI designs a low cost platform based on xilinx FPGAs. In: Proceedings of the IEEE international symposium on industrial electronics

Chapter 18
Overhead Reduction in Data-Flow Software-Based Fault Tolerance Techniques

Eduardo Chielle, Fernanda Lima Kastensmidt, and Sergio Cuenca-Asensi

Abstract There is an increasing interest in aerospace industry to increment the flexibility of the systems and reduce their cost. In this way, FPGAs offer several advantages as low-cost platform to deploy customized systems. However, the use of sub-micron technologies has increased their sensitivity to radiation-induced transient faults. Therefore, the mitigation of soft errors in systems based on soft-core microprocessors has become a major concern not only in the case of configuration memory protection, but also in the case of data and control-flow maintenance. Software-based fault tolerance techniques represent a valid alternative to improve the reliability in such systems at a reduced cost, but the associated time and memory overheads can limit their applicability. This chapter provides different implementation alternatives of software-based techniques in order to reduce overheads while keeping the reliability at the same level.

18.1 Introduction

FPGAs are becoming increasingly attractive to aerospace applications by offering simplicity, flexibility and low-cost [1]. On the same trend, the use of commercial off-the-shelf (COTS) devices is an interesting low-cost alternative, which increases the range of opportunities and markets in such applications [2, 3]. COTS FPGAs offer the opportunity of design customized microprocessor-based systems at a

E. Chielle (✉)
PGMICRO—Instituto de Informática, Universidade Federal do Rio Grande do Sul (UFRGS), Porto Alegre, Brazil
e-mail: echielle@inf.ufrgs.br

F.L. Kastensmidt
Instituto de Informática, Universidade Federal do Rio Grande do Sul (UFRGS), Porto Alegre, Brazil
e-mail: fglima@inf.ufrgs.br

S. Cuenca-Asensi
Computer Technology Department, University of Alicante, Alicante, Spain
e-mail: sergio@dtic.ua.es

F. Kastensmidt, P. Rech (eds.), *FPGAs and Parallel Architectures for Aerospace Applications*, DOI 10.1007/978-3-319-14352-1_18

fraction of the cost of Rad-Hard devices. They also give the possibility to modify the system after the launch to fix errors not detected during the design phase due to their re-configurability capability [4]. Furthermore, they can achieve significant performance improvements when compared to traditional approaches [5, 6]. However, similarly to other advanced devices, its reliability has decreased due to the miniaturization of technologies [7], thus modern FPGAs are more susceptible to transient faults. Such faults can be caused by energized particles present in space or secondary particles such as alpha particles, generated by the interaction of neutron and materials at ground level [8].

Transient ionization may occur when a single radiation ionizing particle strikes the silicon creating a transient voltage pulse known as Single Event Effect (SEE). This effect affects electronic circuits by modifying values stored in the sequential logic, known as Single Event Upset (SEU), or by changing the function of a circuit in the combinational logic, known as Single Event Transient (SET). Such faults may lead the system to incorrectly execute an application. Consequently, to ensure reliability against SEEs, the use of fault tolerance techniques is mandatory.

A big concern about vulnerabilities of systems implemented with FPGAs relies on the configuration memory. The volatile memories of SRAM-based FPGAs make them sensitive to transient faults. Flash-based FPGAs are more reliable than SRAM-based ones and can be used to avoid the problems of the configuration memory [9]. However, in the case of microprocessor-based systems, other vulnerabilities should be taken into account independently of the underlying technology. Transient faults may affect the data or the control-flow of a running application. To ensure reliability in such cases, two types of fault tolerance techniques can be used. The first one, hardware-based techniques, relies on replicating or adding hardware modules, while the second, software-based techniques, relies on adding instruction redundancy and comparison to detect or correct errors.

Hardware-based techniques usually change the original microprocessor architecture by adding logic redundancy, error correcting codes and majority voters. They can also be based on hardware monitoring devices that exploit special purpose hardware modules, called watchdog processors [10], to monitor memory accesses. However, hardware-based techniques present significant overheads, like reduction in the operating frequency, increase in area and power consumption and high design and manufacture costs [11, 12].

Software-based techniques are a well-known approach to protect systems against SEEs by modifying the program code without having to change the underlying hardware. These techniques are non-intrusive and therefore provide high flexibility and low development time and cost. Although software redundancy brings reliability to the system, it requires extra processing time since more instructions are being executed [13, 14]. Furthermore, a reliable program will require more area in memory since software redundancy is inserted [15, 16].

In this chapter we present a set of data-flow fault tolerance techniques that significantly reduce the overheads and keep the data error detection rate at the same level as state of the art techniques. The techniques are composed following a set of rules.

Thus, it is possible to automatically apply them to the program code. The chapter is divided as follow: Sect. 18.2 presents an overview of the software-based fault tolerance. In this section, the concept of control and data-flow techniques is explained. Section 18.3 illustrates a set of rules for data-flow techniques and some techniques created based on such rules. The execution time and memory footprint as well as the data error detection rate of those techniques are shown in Sect. 18.4. Finally, Sect. 18.5 draws some conclusions.

18.2 Software-Based Fault-Tolerance

There are two types of soft errors that affect microprocessors-based systems: errors in the control-flow and errors in the data-flow. A control-flow error occurs when program flow in incorrectly followed, i.e., the error change the program flow. Data-flow error refers to the soft error caused by a bit-flip in a storage device, such as a register or a memory element. They affect the output of the program, but not its execution. To protect against control and data-flow errors there are, respectively, control-flow techniques [17–19] and data-flow techniques [20–22].

18.2.1 Control-Flow Techniques

Control-flow techniques aim to detect incorrect branches during the program execution. The code is divided in basic blocks, which are branch-free sequences of instructions with no jumps into or out of the block except for the first and last instructions. Figure 18.1 shows the basic blocks and the program flow of the code presented in Table 18.1.

These techniques usually assign a unique signature to each basic block and some protection to the program flow. The signature is assigned to a spare register and checked at the end of the basic block. By doing so, they are able to detect incorrect jumps in the program execution.

18.2.2 Data-Flow Techniques

Data-flow techniques aim to detect faults affecting the data, i.e., the values stored in registers and the memory. In order to do that, such techniques duplicate when detecting and triplicate when correcting all the registers used by the application. By duplicating registers, it is possible to detect data errors by comparing a register with its replica. It is important to notice that every operation performed on a register must also be performed on its replica in order to keep the program consistency.

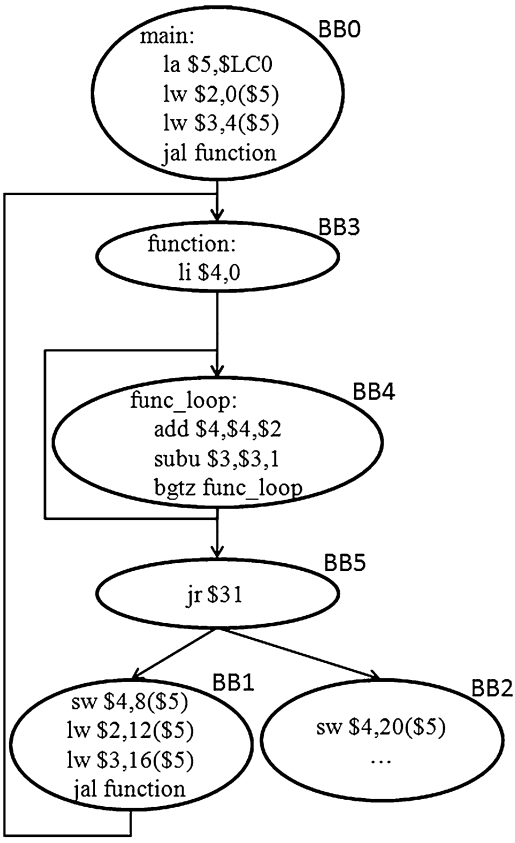


Fig. 18.1 Basic blocks and program flow

Table 18.1 Basic blocks division

BB0	main:
	la $5,$LC0
	lw $2,0($5)
	lw $3,4($5)
	jal function # $31 <- PC + 4
BB1	sw $4,8($5)
	lw $2,12($5)
	lw $3,16($5)
	jal function
BB2	sw $4,20($5)
	…
BB3	function:
	li $4,0
BB4	func_loop
	add $4,$4,$2
	subu $3,$3,1
	bgtz func_loop
BB5	jr $31 # PC <- $31

Table 18.2 Example of a data-flow technique

Original code	Hardened code
1: lw $2,0($4)	1: lw $2,0($4)
	2: lw $12,0($14)
	3: bne $2,$12,error
4: sll $4,$2,1	4: sll $4,$2,1
	5: sll $14,$2,1
	6: bne $4,$12,error
	7: bne $2,$12,error
	8: bne $3,$13,error
9: sw $2,0($3)	9: sw $2,0($3)
	10: sw $12,0($13)
	11: bne $4,$14,error
	12: bne $2,$12,error
13: sw $4,0($2)	13: sw $4,0($2)
	14: sw $14,0($12)
	15: bne $4,$14,error
	16: bne $5,$15,error
17: ble $4,$5,$L2	17: ble $4,$5,$L2

Table 18.2 shows an example of a code hardened by a data-flow technique. In the left side, one can see the original code composed by five instructions, lines 1, 4, 9, 13 and 17. And the right side, shows the same code hardened. Registers $12, $13, $14 and $15 are replicas of registers $2, $3, $4 and $5, respectively. The duplication of the original instructions is presented in lines 2, 5, 10 and 14. In this technique, checkers are inserted before stores and branches, checking the source registers, and after any other instruction, checking the target register. Checkers are inserted in lines 3, 6, 7, 8, 11, 12, 15 and 16.

It is possible to notice by the example that data-flow techniques present significant overheads. The overheads caused by data-flow techniques are higher than the ones caused by control-flow because they duplicate or triplicate the data and the operations performed over them and insert checkers to verify the consistence of the data, while control-flow techniques only insert instructions to change and check the value of signatures. If the application needs fault tolerance, but has performance or energy constraints, data-flow techniques might not be applied. New data-flow techniques with reduced overheads are desirable in such scenarios.

18.3 Methodology and Implementation

A set of rules for data-flow protection is presented in Table 18.3. They are divided in three types: global, duplication and checking rules. There is only one global rule and it is applied for all techniques. It states that every register used by the program has a spare register assign as replica. It makes duplication and checking possible.

Table 18.3 Techniques and rules

Global rules (valid for all techniques)	
G1	Each register used in the program has a spare register assigned as replica
Duplication rules (performing the same operation on the register's replica)	
D1	All instructions except branches
D2	All instructions, except branches and stores
Checking rules (compare the value of a register with its replica)	
C1	Before each read on the register (except load/store and branch instructions)
C2	After each write on the register
C3	Before loads, the register that contains the address
C4	Before stores, the register that contains the datum
C5	Before stores, the register that contains the address
C6	Before branches

The duplication rules regard how the instructions are duplicated. They are applied only when write operations in a register or memory are performed. Thus, branch instructions are never duplicated. There are two possible duplication rules but each technique can only use one. D1 duplicates all instructions except branches. It includes stores which allow the use of unprotected memories since the original value and its replica can be store in different positions in the memory. D2 duplicates all instructions, except branches and stores. It is adequate when the memory is hardened because there is no need for software redundancy in memory since the memory is already hardened. Thus, the overhead caused by the duplication and the number of memory accesses are reduced.

The checking rules indicate when a register and its replica are compared. The aim is to verify if an error has occurred. If the original register and its replica present the different values, an error is reported. The techniques can have any possible combination of checking rules, from zero (no detection) to all. Theoretically, the more checkers are included in one technique, the more reliability is achieved. On the other hand, the overhead is higher. That is the reason why we have not proposed a technique using all the checking rules. The overhead would be bigger than the techniques present in the literature and it would go against the purpose of this work.

Checking rule C1 states that a checker shall be placed before a register is read by an instruction, excluding load/store and branches. The checker compares the values of the register and its replica to detect a possible error. Regarding C2, a checker is inserted right after a write operation is performed on a register. When C3 is used, the register that contains the address in load instructions has to be checked before the load is performed. C4 and C5 insert checkers before stores. C4 checks the register that contains the datum and C5 checks the register that contains the address. Finally, C6 states that the register has to be checked before it is used by a branch instruction.

From the rules, 17 techniques have been implemented and they are showed in Table 18.4. Each technique consists of a combination of rules. Global rule G1 and

Table 18.4 Techniques and rules

Technique	Duplication rule	Checking rules
VAR0	D1	None
VAR0+	D2	None
VAR1	D1	C1, C3, C4, C5, C6
VAR1+	D2	C1, C3, C4, C5, C6
VAR1++	D2	C1, C3, C4, C5
VAR2	D1	C2, C4, C5, C6
VAR2+	D2	C2, C4, C5, C6
VAR2++	D2	C2, C4, C5
VAR3	D1	C3, C4, C5, C6
VAR3+	D2	C3, C4, C5, C6
VAR3++	D2	C3, C4, C5
VAR4	D1	C4, C5, C6
VAR4+	D2	C4, C5, C6
VAR4++	D2	C4, C5
VAR5	D1	C4, C6
VAR5+	D2	C4, C6
VAR5++	D2	C4

one duplication rule (D1 or D2) are mandatory. Only one duplication rule can be used by each technique. The checking rules are optional.

Three of the implemented techniques (VAR1, VAR2 and VAR3) belong to reference [22]. According to the authors, VAR1 is based on [24], VAR4 is equivalent to EDDI [23]. VAR0 and VAR0+ techniques do not detect errors, but they were implemented because they show the minimum overhead possible when all the registers used by the program are duplicated. In these techniques, no checking is done, only duplications. VAR1+ and VAR1++ are variations of VAR1. They use a different duplication rule and VAR1++ implements fewer checking rules. The same can be said about VAR2+ and VAR2++ with relation to VAR2 and VAR3+ and VAR3++ concerning VAR3. They all have a different duplication rule and VAR2++ and VAR3++ uses fewer checking rules than VAR2 and VAR3, respectively. By removing more checking rules we get to VAR4 and VAR5 and applying the same concept stated before we get techniques VAR4+, VAR4++, VAR5+ and VAR5++.

Table 18.5 exemplifies how the different techniques are applied to the program code. In this regard, it was used a piece of code that permits to see the application of all rules. It consists of five instructions: two loads, one add, one store and one branch. They are presented under the original code, formatted as normal text.

Techniques that use D1 have no plus signal in the name and other ones, with one (+) or two (++) plus signals, use D2. Techniques that have D1 as duplication rule, for example VAR0, have all instructions that perform a write operation in a register or memory, i.e., all instructions except branches, replicated using the registers replicas. The techniques using D2, for example VAR0+, only duplicate the instructions that

Table 18.5 Application of the techniques

Original code	VAR0	VAR0+	VAR1	VAR1+	VAR1++
lw $4,0($2)	lw $4,0($2)	lw $4,0($2)	*bne* $2,$*12*,*err*	*bne* $2,$*12*,*err*	*bne* $2,$*12*,*err*
lw $5,4($2)	**lw $14,0($12)**	**lw $14,0($12)**	lw $4,0($2)	lw $4,0($2)	lw $4,0($2)
add $3,$3,1	lw $5,4($2)	lw $5,4($2)	**lw $14,0($12)**	**lw $14,0($12)**	**lw $14,0($12)**
sw $4,0($5)	**lw $15,4($12)**	**lw $15,4($12)**	*bne* $2,$*12*,*err*	*bne* $2,$*12*,*err*	*bne* $2,$*12*,*err*
ble $3,$6,loop	add $3,$3,1	add $3,$3,1	lw $5,4($2)	lw $5,4($2)	lw $5,4($2)
	add $13,$13,1	**add $13,$13,1**	**lw $15,4($12)**	**lw $15,4($12)**	**lw $15,4($12)**
	sw $4,0($5)	sw $4,0($5)	*bne* $*3*,$*13*,*err*	*bne* $*3*,$*13*,*err*	*bne* $*3*,$*13*,*err*
	sw $14,0($15)	ble $3,$6,loop	add $3,$3,1	add $3,$3,1	add $3,$3,1
	ble $3,$6,loop		**add $13,$13,1**	**add $13,$13,1**	**add $13,$13,1**
			bne $*4*,$*14*,*err*	*bne* $*4*,$*14*,*err*	*bne* $*4*,$*14*,*err*
			bne $*5*,$*15*,*err*	*bne* $*5*,$*15*,*err*	*bne* $*5*,$*15*,*err*
			sw $4,0($5)	sw $4,0($5)	sw $4,0($5)
			sw $14,0($15)	*bne* $*3*,$*13*,*err*	ble $3,$6,loop
			bne $*3*,$*13*,*err*	*bne* $*6*,$*16*,*err*	
			bne $*6*,$*16*,*err*	ble $3,$6,loop	
			ble $3,$6,loop		
VAR2	**VAR2+**	**VAR2++**	**VAR3**	**VAR3+**	**VAR3++**
lw $4,0($2)	lw $4,0($2)	lw $4,0($2)	*bne* $2,$*12*,*err*	*bne* $2,$*12*,*err*	*bne* $2,$*12*,*err*
lw $14,0($12)	**lw $14,0($12)**	**lw $14,0($12)**	lw $4,0($2)	lw $4,0($2)	lw $4,0($2)
bne $*4*,$*14*,*err*	*bne* $*4*,$*14*,*err*	*bne* $*4*,$*14*,*err*	**lw $14,0($12)**	**lw $14,0($12)**	**lw $14,0($12)**
lw $5,4($2)	lw $5,4($2)	lw $5,4($2)	*bne* $2,$*12*,*err*	*bne* $2,$*12*,*err*	*bne* $2,$*12*,*err*
lw $15,4($12)	**lw $15,4($12)**	**lw $15,4($12)**	lw $5,4($2)	lw $5,4($2)	lw $5,4($2)
bne $*5*,$*15*,*err*	*bne* $*5*,$*15*,*err*	*bne* $*5*,$*15*,*err*	**lw $15,4($12)**	**lw $15,4($12)**	**lw $15,4($12)**
add $3,$3,1	add $3,$3,1	add $3,$3,1	add $3,$3,1	add $3,$3,1	add $3,$3,1
add $13,$13,1	**add $13,$13,1**	**add $13,$13,1**	**add $13,$13,1**	**add $13,$13,1**	**add $13,$13,1**
bne $*3*,$*13*,*err*	*bne* $*3*,$*13*,*err*	*bne* $*3*,$*13*,*err*	*bne* $*4*,$*14*,*err*	*bne* $*4*,$*14*,*err*	*bne* $*4*,$*14*,*err*
bne $*4*,$*14*,*err*	*bne* $*4*,$*14*,*err*	*bne* $*4*,$*14*,*err*	*bne* $*5*,$*15*,*err*	*bne* $*5*,$*15*,*err*	*bne* $*5*,$*15*,*err*
bne $*5*,$*15*,*err*	*bne* $*5*,$*15*,*err*	*bne* $*5*,$*15*,*err*	sw $4,0($5)	sw $4,0($5)	sw $4,0($5)
sw $4,0($5)	sw $4,0($5)	sw $4,0($5)	**sw $14,0($15)**	*bne* $*3*,$*13*,*err*	ble $3,$6,loop
sw $14,0($15)	*bne* $*3*,$*13*,*err*	ble $3,$6,loop	*bne* $*3*,$*13*,*err*	*bne* $*6*,$*16*,*err*	
bne $*3*,$*13*,*err*	*bne* $*6*,$*16*,*err*		*bne* $*6*,$*16*,*err*	ble $3,$6,loop	
bne $*6*,$*16*,*err*	ble $3,$6,loop		ble $3,$6,loop		
ble $3,$6,loop					
VAR4	**VAR4+**	**VAR4++**	**VAR5**	**VAR5+**	**VAR5++**
lw $4,0($2)	lw $4,0($2)	lw $4,0($2)	lw $4,0($2)	lw $4,0($2)	lw $4,0($2)
lw $14,0($12)	**lw $14,0($12)**	**lw $14,0($12)**	**lw $14,0($12)**	**lw $14,0($12)**	**lw $14,0($12)**
lw $5,4($2)	lw $5,4($2)	lw $5,4($2)	lw $5,4($2)	lw $5,4($2)	lw $5,4($2)
lw $15,4($12)	**lw $15,4($12)**	**lw $15,4($12)**	**lw $15,4($12)**	**lw $15,4($12)**	**lw $15,4($12)**
add $3,$3,1	add $3,$3,1	add $3,$3,1	add $3,$3,1	add $3,$3,1	add $3,$3,1
add $13,$13,1	**add $13,$13,1**	**add $13,$13,1**	**add $13,$13,1**	**add $13,$13,1**	**add $13,$13,1**
bne $*4*,$*14*,*err*	*bne* $*4*,$*14*,*err*	*bne* $*4*,$*14*,*err*	*bne* $*4*,$*14*,*err*	*bne* $*4*,$*14*,*err*	*bne* $*4*,$*14*,*err*
bne $*5*,$*15*,*err*	*bne* $*5*,$*15*,*err*	*bne* $*5*,$*15*,*err*	sw $4,0($5)	sw $4,0($5)	sw $4,0($5)

(continued)

Table 18.5 (continued)

Original code	VAR0	VAR0+	VAR1	VAR1+	VAR1++
sw $4,0($5)	sw $4,0($5)	sw $4,0($5)	**sw $14,0($15)**	*bne $3,$13,err*	ble $3,$6,loop
sw $14,0($15)	*bne $3,$13,err*	ble $3,$6,loop	*bne $3,$13,err*	*bne $6,$16,err*	
bne $3,$13,err	*bne $6,$16,err*		*bne $6,$16,err*	ble $3,$6,loop	
bne $6,$16,err	ble $3,$6,loop		ble $3,$6,loop		
ble $3,$6,loop					

perform a write operation in a register, i.e., all instructions but branches and stores are duplicated. The instructions inserted by the duplication rules are presented in bold.

The instructions inserted by checking rules are presented in italic. In VAR1 and VAR1+ we can see almost all checking rules being used, with the exception of C2. The first and the second checkers are due to C3. The third one is because of C1. Note that the same register is used twice by the instruction but it is only checked once. This optimization is applied because there is no point on checking the same register twice in a row. It would only increase even more the overheads. The fourth and fifth checkers are related to C4 and C5, respectively. And the sixth and the seventh are due to C6. VAR1++ has the all the checking rules that VAR1 and VAR1+ have but C6. VAR2 and VAR2+ use all the checking rules except for C1 and C3. The first three checkers are related to C2. The fourth and the fifth are due to C4 and C5, respectively. And the last two are because of C6. The only difference from VAR2 and VAR2+ to VAR2++ checking rules is that VAR2++ does not implement C6. VAR3 and VAR3+ use C3, C4, C5 and C6. The first and the second checkers are due to C3, the fourth and the fifth are due to C4 and C5, respectively, and the last two checkers are due to C6. VAR3++ implements the same checking rules, except for C6. VAR4 and VAR4+ use checking rules C4, C5, C6 and VAR4++ uses C4 and C5. VAR5 and VAR5+ use C4 and C6 and VAR5++ uses only checking rule C4, which checks the register that contains the datum in stores.

For example, let us see how VAR3 technique is applied. Firstly, we must assign replicas to all registers used by the program. Thus, registers $12, $13, $14, $15 and $16 are assign as replica of registers $2, $3, $4, $5 and $6, respectively. It works the same for all techniques. VAR3 technique uses duplication rule D1. The duplications are inserted in lines 3, 6, 8 and 12 (bold lines). And the checking rules are C3, C4, C5 and C6. So checks have to be inserted before loads, verifying the register that contains the address (C3), before stores checking the registers that contain the datum and the address (C4 and C5) and the registers used by branches (C6). At the first and fourth lines, there are checking instructions regarding C3. At lines 9 and 10, checks are made respecting C4 and C5, respectively. C6 is applied at lines 13 and 14.

Now, if we look at VAR4++. The duplication rule is D2. So all the instructions, except branches and stores are duplicated. It can be seen at lines 2, 4 and 6 (bold). The checking rules consist of C4 and C5. They are applied at lines 7 and 8, respectively (italic). If we compare VAR3 and VAR4++, we can see that VAR4++ clearly presents a lower overhead but at a cost of less checking instructions.

18.4 Results

To evaluate exactly how much the different techniques impact in the overheads and error detection rate, we defined six programs to be used as benchmarks, tested the overheads and submitted them to a fault injection campaign. The benchmarks consist of a bubble sort, the Dijkstra's algorithm, a matrix multiplication, the Run Length Encoding (RLE), a summation and the TETRA Encryption Algorithm (TEA2). They are hardened with all the techniques, totalizing 102 hardened versions. The hardening is done automatically using CFT-tool [25]. This tool modifies the program's assembly code in order to apply a selected technique.

The parameters measured in the tests are: the execution time, the memory footprint and the data error detection. The execution time and the memory footprint are expressed by the relation between the value presented by the hardened program hardened and the unhardened version. The data error detection rate is the percentage of errors affecting the data-flow that are detected. Data errors are the errors that affect the output of the program, but not its execution.

A total of 1,020,000 faults were injected (10,000 per hardened version and only one per execution) by simulation at Register Transfer Level (RTL) using ModelSim in the miniMIPS microprocessor. MiniMIPS is a 32 bits core based on MIPS I architecture. It has a pipeline of five stages and 32 general purpose registers. All miniMIPS instructions take five cycles to be executed and the peak throughput is one instruction per cycle [26].

Faults are injected by forcing a bit-flip in the microprocessor's internal signals. Every signal of the microprocessor is considered. The fault duration was set to one clock cycle in order to force its effect to hit the clock barrier of the flip-flops and therefore increase the probability of an error. A golden execution (with no injected faults) is executed. All the values of the PC during the execution are saved. Also, the portion of the memory that contains the program output is saved. Then, the program is submitted to faults and the values of the PC and the memory results of the program under test are compared to the golden results. As we are using only techniques developed to detect data-flow errors, only the errors that affected the data-flow are considered. Faults causing control-flow errors (errors that changed the program execution flow) are ignored, since they are not in the scope of this work. We also discarded faults that were masked by the microprocessor logic because they do not cause an error. The error is signaled when the result stored in the memory differs from the expected one.

Figure 18.2 shows the averages execution time, memory footprint and error detection rate for all techniques applied to the case-study applications. As one can see, the average minimum overhead is 22 % for the execution time and 24 % for the memory footprint (see VAR0+).

Techniques VAR1, VAR1+, VAR1++, VAR2, VAR2+ and VAR2++ present high error detection rates but very high overheads. Similar error detection rates can be obtained by techniques VAR3, VAR3+, VAR3++, VAR4, VAR4+ and VAR4++ with the advantage of considerable lower overheads. It shows that after certain point, checking instructions get saturated.

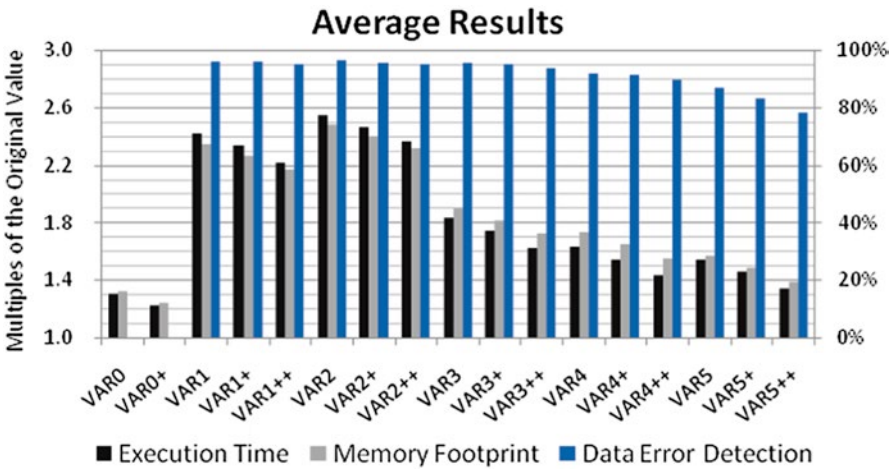


Fig. 18.2 Averages execution time, memory footprint and data error detection for all techniques

Considering only the baseline techniques, VAR3 presents the best results since it has the same error detection rate that VAR1 and VAR2 and present smaller overheads. By changing the duplication rule D1 of the VAR3 technique to D2, we create VAR3+ technique. This technique reduces the execution time overhead from 83 to 74 % and the memory footprint overhead from 90 to 82 % and keep the same error detection rate as VAR3. Comparing VAR3 with VAR4++, we can see a reduction of 40 % in the execution time overhead and of 35 % in the memory footprint overhead with a loss of 5 % in the error detection rate. VAR4++ can be a better solution when constraints are more restrictive or when using the technique combined with a control-flow technique.

18.5 Conclusions

Software-based fault detection techniques are less costly than hardware-based ones but they present time and memory overheads. Several data-flow techniques based on a set of rules designed to search for different tradeoffs between reliability and execution time and between reliability and memory footprint have been presented.

Results show reduction of the overhead in the execution time from 83 to 74 % and from 90 to 82 % in the memory footprint with no degradation of the detection capabilities (VAR3 × VAR3+). With some reduction in the error detection rate the overheads can go down to 34 % in the execution time and 39 % in the memory footprint (VAR5++). In this chapter it was shown that there is still room to reduce overheads without degrading the detection rate. It enables systems with more strict constraints to get the benefits of software protection and also provides performance improvements to systems that already use software-based fault tolerance techniques. Furthermore, the presented techniques can be used together with selective hardening, reducing even more the overheads.

References

1. Li Y, Li D, Wang Z (2000) A new approach to detect-mitigate-correct radiation-induced faults for SRAM-based FPGAs in aerospace application. In: Proceedings of the IEEE national aerospace and electronics conference
2. Reyneri LM, Sansoè C, Passerone C, Speretta S, Tranchero M, Borri M, Del Corso D (2010) Design solutions for modular satellite architectures. In: Aerospace technologies advancements, Intech, Olajnica 19/2, 32000 Vukovar, Croatia (Chapter 9)
3. Grillmayer G, Gsell J, Lepain A, Roser H, Hartling M, Wegmann T, Huber F (2003) ILSE—first laboratory model of the small satellite program at the University of Stuttgart. In: Proceedings of the 54th international astronautical congress, iAC-03-IAA.11.1.09, Bremen
4. Abate F, Sterpone L, Violante M, Kastensmidt FL (2009) A study of the single event effects impact on functional mapping within flash-based FPGAs. In: Proceedings of the design, automation & test in Europe conference & exhibition
5. Smith GL, Torre L (2006) Techniques to enable FPGA based reconfigurable fault tolerant space computing. In: IEEE aerospace conference
6. Cuenca-Asensi S, Martínez-Álvarez A, Restrepo-Calle F, Palomo FR, Guzmán-Miranda H, Aguirre MA (2011) Soft core based embedded systems in critical aerospace applications. J Syst Archit 57(10):886–895
7. Baumann R (2001) Soft errors in advanced semiconductor devices—part I: the three radiation sources. IEEE Trans Device Mater Reliab 1(1):17–22
8. International technology roadmap for semiconductors, 2005 edn, Chapter design, 2005, pp 6–7
9. Costenaro E, Violante M, Alexandrescu D (2011) A new IP core for fast error detection and fault tolerance in COTS-based solid state mass memories. In: Proceedings of the IEEE 17th international on-line testing symposium (IOLTS)
10. Mahmood A, McCluskey E (1988) Concurrent error detection using watchdog processors—a survey. IEEE Trans Comput 37(2):160–174
11. Unsal OS, Koren I, Krishna CM (2002) Towards energy-aware software-based fault tolerance in real-time systems. In: Proceedings of the international symposium on low power electronics and design
12. Asensi SC, Alvarez AM, Calle FR, Palomo FR, Miranda HG, Aguirre MA (2011) A novel co-design approach for soft errors mitigation in embedded systems. IEEE Trans Nucl Sci 58(3):1059–1065
13. Yao T, Zhou H, Fang M, Hu H (2013) low power consumption scheduling based on software fault-tolerance. In: Proceedings of the 9th international conference on natural computation
14. Assayad I, Girault A, Kalla H (2011) Tradeoff exploration between reliability, power consumption and execution time. In: Proceedings of the 30th international conference on computer safety, reliability and security
15. Vogelsang T (2010) Understanding the energy consumption of dynamic random access memories. In: Proceedings of the 43rd annual IEEE/ACM international symposium on microarchitecture
16. Li S, Lai EM-K, Absar MJ (2003) Minimizing embedded software power consumption through reduction of data memory access. In: Proceedings of the 4th international conference on information, communications & signal processing
17. Oh N, Shirvani PP, McCluskey EJ (2002) Control-flow checking by software signatures. IEEE Trans Reliab 51(1):111–122
18. Mcfearin LD, Nair VSS (1995) Control-flow checking using assertions. In: Proceedings of the IFIP international working conference dependable computing for critical applications (DCCA-05), Urbana-Champaign, Sept 1995
19. Alkhalifa Z, Nair VSS, Krishnamurthy N, Abraham JA (1999) Design and evaluation of system-level checks for on-line control flow error detection. IEEE Trans Parallel Distrib Syst 10(6):627–641
20. Azambuja JR, Lapolli A, Rosa L, Kastensmidt FL (2011) Detecting SEEs in microprocessors through a non-intrusive hybrid technique. IEEE Trans Nucl Sci 58:993–1000

21. Oh N, Mitra S, McCluskey E (2002) ED4I: error detection by diverse data and duplicated instructions. IEEE Trans Comput 51(2):180–199
22. Azambuja JR, Lapolli A, Altieri M, Kastensmidt FL (2011) Evaluating the efficiency of software-only techniques to detect SEU and SET in microprocessors. In: Proceedings of the IEEE Latin American symposium on circuits and systems
23. Oh N, Shirvani PP, McCluskey EJ (2002) Error detection by duplicated instructions in super-scalar processors. IEEE Trans Reliab 51(1):63–75
24. Cheynet P, Nicolescu B, Velazco R, Rebaudengo M, Reorda MS, Violante M (2000) Experimentally evaluating an automatic approach for generating safety-critical software with respect to transient errors. IEEE Trans Nucl Sci 47(6 part 3):2231–2236
25. Chielle E, Barth RS, Lapolli AC, Kastensmidt FL (2012) Configurable tool to protect processors against SEE by software-based detection techniques. In: Proceedings of the IEEE Latin American symposium on circuits and systems
26. Hangout LMOSS, Jan S (2010) *The minimips project* [online]. Available http://www.opencores.org/projects.cgi/web/minimips/overview

Chapter 19
Fault-Tolerance Techniques for Soft-Core Processors Using the Trace Interface

Luis Entrena, Almudena Lindoso, Marta Portela-Garcia, Luis Parra, Boyang Du, Matteo Sonza Reorda, and Luca Sterpone

Abstract As microprocessors are increasingly used in safety-critical applications, there is a growing demand for effective fault-tolerance techniques that can mitigate the effects of soft errors while reducing intrusiveness and minimizing the impact on performance and power consumption. To this purpose, approaches that are based on monitoring the microprocessor operation through an external interface in a non-intrusive manner have recently been proposed. In this paper we focus on the use of the trace interface for on-line monitoring. This interface provides detailed information about the instructions executed by the processor and can be reused to support error detection and correction in several ways, including multi-processors in hardware redundancy, time redundancy and control-flow checking.

19.1 Introduction

Microprocessor-based digital systems are ubiquitous today. They are used in a wide variety of applications, including safety-critical ones in sectors such as automotive, aerospace, telecommunications or biomedical. In these fields of application, an error in a microprocessor may produce a wrong computation result or losing the control of a system with catastrophic consequences. At the same time, the evolution of semiconductor technology has enabled the availability of microprocessors at very low costs but has also increased the susceptibility to soft errors even at the ground level. For all these reasons, there is a growing demand for effective fault-tolerance techniques that can provide the required level of robustness for microprocessor-based systems while reducing intrusiveness and minimizing the impact on performance and power consumption.

L. Entrena (✉) • A. Lindoso • M. Portela-Garcia • L. Parra
Electronic Technology Department, Universidad Carlos III de Madrid, Leganés, Spain
e-mail: entrena@ing.uc3m.es; alindoso@ing.uc3m.es; mportela@ing.uc3m.es; lparra@pa.uc3m.es

B. Du • M. Sonza Reorda • L. Sterpone
Control and Computer Engineering Department, Politecnico di Torino, Torino, Italy
e-mail: boyang.du@polito.it; matteo.sonzareorda@polito.it; luca.sterpone@polito.it

F. Kastensmidt, P. Rech (eds.), *FPGAs and Parallel Architectures for Aerospace Applications*, DOI 10.1007/978-3-319-14352-1_19

Developing a microprocessor with the level of quality that is required for real applications is a complex task that involves a large effort in both the hardware design and the associated software tools. For this reason, COTS (Commercial-Off-The-Shelf) components or existing soft-cores are usually preferred. In this case, conventional hardware-based fault tolerance techniques cannot be used because the hardware cannot generally be modified.

Software-based fault tolerance techniques have been widely studied and are commonly used for COTS. They introduce redundancy in the code to detect or correct errors. However, this typically produces a significant performance decrease. On the other hand, software-based approaches are limited because a processor often contains many registers that cannot be directly accessed through software. This limitation is particularly relevant to provide protection for control-flow errors.

Alternatively, there is a growing interest in fault-tolerance techniques that monitor the processor operation from outside in a non-intrusive manner. The monitor is attached to a suitable interface from which it can observe the instruction and data flows coming in and out the processor. Monitor modules are implemented in hardware in order to match the processor speed. In the case of a soft core implemented in a FPGA, the monitor can be included in the same FPGA to provide an integrated solution [1].

The obvious observation interface is the same interface the processor uses to fetch instructions or store data, i.e., the memory buses. Approaches using this kind of interface have been proposed in [2–4]. However, modern processors have other interfaces that can be used for monitoring. As a matter of fact, monitoring capabilities are crucial for system development and software debugging, and are increasingly supported through On-Chip Debug (OCD) interfaces. As these interfaces are useless during normal operation, they can be easily reused for on-line monitoring in an inexpensive way. On the other hand, they can provide internal access to the microprocessor without disturbing it.

In this paper we focus on the use of the trace interface for on-line monitoring and show the possible uses that such interface can have for on-line error detection. The trace interface is a kind of OCD interface that is provided by many processors and it is included in the Nexus standard (class 2, 3 and 4, [5]). As it is intended to obtain traces of the instructions executed by a processor, it generally provides detailed information of the processor operation. While the memory interface provides the instruction flow at the fetch stage, the trace interface reports instructions after they have been executed, so that errors that occur after the instruction has entered the processor can be detected. Moreover, accessing to the instruction flow provided on the fly by the trace interface allows performing control flow checking, and thus detecting possible faults changing the expected sequence of instruction execution.

The remaining of the paper is as follows. Section 19.2 summarizes related work in the field of fault-tolerance for microprocessors. Section 19.3 introduces the trace interface. Sections 19.4 and 19.5 describe several techniques that use the trace interface for error detection or correction. Finally, Sect. 19.6 shows the conclusions of this work.

19.2 Related Work

Techniques that detect and mitigate soft errors in microprocessors are commonly divided in three categories [6]: hardware techniques, software techniques and hybrid techniques. In all of them, two different types of errors are considered: errors that affect the data flow and errors that affect the control flow.

Hardware techniques use hardware modifications to achieve microprocessor's error detection. Due to the complexity and involved costs of microprocessor's architectural changes, the most common approach consists in adding an external module to the microprocessor to monitor its behavior. In the literature, such an external module is referred to as a watchdog processor [7]. The observation capabilities of a watchdog processor mainly depend on the available microprocessor's connections. Watchdog processors have easier access to control-flow information and related work mainly focuses on this type of errors. Data observation depends on the architecture, the application and the available connections and it is quite limited in this scenario.

Watchdog processors can be active or passive. Active watchdog processors execute a program concurrently with the microprocessor. Passive watchdog processors only compute simple operations related with the executed flow and compare the result with the expected one. Passive watchdogs are smaller but require large memory to store the expected result. Active watchdog processors increase error coverage by increasing the processor complexity and the required area [8, 9]. Increasing complexity of an active watchdog processor leads to an external added architecture that could be similar in complexity to the microprocessor under test.

Software techniques modify the software to mitigate errors. In this case, no additional hardware is required but the required software modifications enlarge the code size leading to a performance decrease. Some techniques of this category can be easily implemented because they can be introduced automatically as a set of rules at the compilation step [10].

Data-flow software-based techniques monitor the data correctness. In this category, approaches are commonly split between duplication techniques and assertion techniques. Duplication techniques duplicate computations at four different levels of granularity: instruction, block of instructions, procedure or the entire program [11, 12]. Duplication creates a redundant data flow that can be checked to detect errors. A granularity decrease produces an enlargement of both error latency and execution time but code size is reduced. Data duplication techniques present high error coverage in spite of performance decrease and memory overhead. Several works propose techniques to reduce overheads by selecting for duplication only the most critical information [13, 14]. Assertion techniques introduce additional statements in the code to test the validity and correctness of the data flow. The location and contents of the statements are critical for the error coverage in this kind of methods [15]. Assertion-based techniques are application-dependent, as both the statement content and its location depend on the source code. In this case, the programmer knowledge and ability to find the suitable locations and to check the proper information are important.

Control-Flow software-based techniques evaluate the correctness of the execution flow. An updated overview of the proposed techniques in the literature can be found in [2]. This kind of techniques mostly uses signatures [16, 17] or assertions [18, 19]. Control-flow techniques usually divide the code into branch-free blocks (called Basic Blocks or BBs). When a signature method is utilized, a signature is computed for each BB before execution. When execution takes place, the signature of each BB is computed and compared with the expected one. Whenever a mismatch occurs, an error is detected. Assertions can also be used to check the correct flow of BBs. Implementations of any of the existing methods usually produce large overheads [20].

Hybrid techniques present a trade-off between hardware and software techniques. Several works have proposed different hybrid techniques for microprocessor error detection. Usually, external hardware modules are connected to the memory bus interface to control both data and instructions that are sent through the bus [2]. With this approach instructions are checked just before execution, and techniques make sure the microprocessor is receiving data without errors. However, complex microprocessors with several pipeline stages need additional observation points to certify not only that instructions and data provided by the memories are right but also that nothing have altered the information in the pipeline before the execution stage.

Hybrid techniques usually use software techniques for controlling the data-flow. Even though the observation points can give information about data and instructions separately, it is difficult to merge them into a single observation point. The common approach is based on data storage and instruction re-execution.

Existing hybrid techniques vary considerably. For example, in [2] the proposed hybrid technique consists in a hardware module connected to the microprocessor's memory that monitors the microprocessor's behavior and a hardened assertion-based software technique. In [12] a reconfiguration approach is proposed. In this case, every application program needs its own specific module that will require reconfiguration when using the application. In [20], software is modified to allow an I-IP (Infrastructure IP) monitoring both data and control flow. In this approach, software hardening includes special function calls whenever a basic block starts or ends. In [3], a BB-based approach is proposed where block identifiers or signatures are sent to a watchdog processor that monitors the microprocessor behavior. The watchdog processor computes the signature and checks the correctness of the execution by comparing it with the expected result. Authors report full coverage but this approach presents high overheads in both performance and area. Signatures storage for comparison purposes also presents a drawback which is solved in [11] with an assertion-based approach. However, this approach cannot be applied to architectures with on-chip cache memory. In [14, 17], several hybrid approaches are evaluated which are based on selective hardening. Evaluation compares methods regarding execution time, performance, memory overhead, fault detection capability, etc. In this case, the evaluated hybrid techniques are intrusive because they require architectural changes in the microprocessor.

19.3 The Trace Interface

Nowadays, most complex microprocessors include hardware modules for debugging purposes. They are commonly referred as On-chip Debuggers (OCD). OCDs provide software designers the capability to control the execution flow (step-by-step execution, breakpoints) and data flow (access to internal resources such as register contents or memory contents). The microprocessor's OCD typically stores information of the executed software in a circular buffer called the trace buffer, although the trace interface can be directly accessed as well. The collected information varies with the architecture. Typically, it contains the following data: program counter, instruction register, arithmetic operations results, status flags, signals related to the pipeline and memory accesses, etc.

As an example, the LEON3 provides a 128-bit trace interface [21] that contains the following fields:

- Program Counter (30 bits, word aligned)
- Opcode (32 bits)
- Load/Store parameters (32 bits)
- Time tag (30 bits)
- Control signals: multi-cycle instruction, instruction trap and error mode.

The OCD collected information is sent to a host through a standard bus, such as JTAG. Serial ports are used for simple debugging operations, while parallel ports are used to support data intensive debug operations such as real-time tracing. Embedded cores usually come along with modules for improved support of debugging functions. Examples of these modules are the ARM Embedded Trace Macrocell (ETM) [22], the Xilinx MicroBlaze™ Trace Core (XMTC) [23] or the LEON3 Debug Support Unit (DSU) [24].

A complete set of OCD features are defined in the Nexus 5001 Forum™ standard [25]. Nexus standard is quite popular among microprocessor's manufacturers and their features can also be found in microprocessors that are not Nexus compliant.

19.4 Execution Checking

Fault detection in a given system is essentially a two-stage process: first of all, observing the outputs and/or the value of memory elements, and, after that, comparing those values with the expected ones. Comparing just the outputs increases the latency of fault detection and complicates the implementation of error correction techniques. Therefore, the access to internal resources is desirable, or necessary in many applications, depending on the system requirements. Usually, this task is very difficult or requires of invasive mechanisms that can affect the normal behavior of the circuit under test. However, as it has been explained in previous sections, in modern microprocessors, the trace interface is a non-intrusive mechanism for observing the values stored in internal resources.

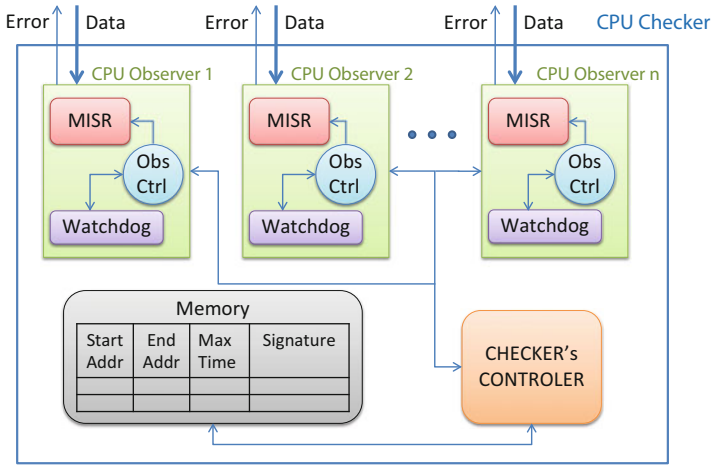


Fig. 19.1 CPU-checker architecture

With respect to the comparison with the golden results, these data must be available and two options are possible:

- Reference data to be compared are stored in some way in the system. This approach is valid only for static applications where the results are always the same. To ensure that no fault can affect the golden values is mandatory, what is a very restrictive condition. Its implementation is limited by the memory capacity of the system, although data compression techniques can be used to soft this restriction.
- Reference data to be compared are also generated in the system. Generally, it can be assumed that only one fault is going to affect the system at a time, and therefore, it is very unlikely that two or more execution replicas suffer a fault during the same run.

Depending on the chosen approach and the way to implement it, different solutions are possible. The most suitable solution depends on the system requirements since each possible approach involves different advantages and disadvantages. In [25], an on-line fault detection technique based on using trace interface and the on-line generation of reference data is presented. In this approach, the behavior of the system during several executions of a critical task is compared by observing data through the trace interface. A hardware module, called CPU Checker, connected to the trace interface calculates, for each critical task, the signature of the generated data. When the task replica has finished, the signatures are compared and a fault is detected when they differ. Comparing only a signature instead of a set of data reduces significantly the memory requirements and the time spent in the comparison step.

The CPU Checker architecture is shown in Fig. 19.1. It contains n CPU Observer modules, which are in charge of generating the signature for each replica of a given task. Thus, n replicas of the task can be executed in parallel. A memory block is used

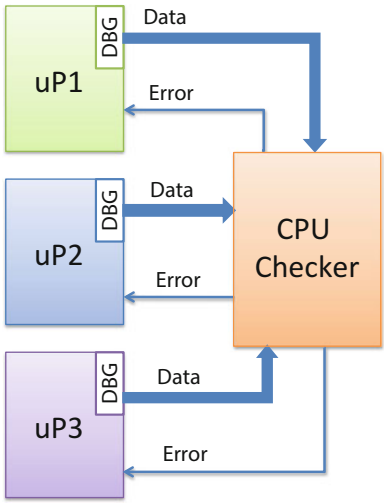


Fig. 19.2 Multi-core architecture with the CPU Checker to perform on-line fault correction

to store the instruction addresses (start and end addresses) of the critical task to be checked, the maximum time needed to execute the task and the calculated signatures. And finally, a Checker's controller module manages the complete process. The instruction addresses of the critical task are used to know when that task starts and finishes, and thus, when the signature generation should start and end. The maximum time parameter is used by a watchdog to detect a loss of sequence in the execution.

This technique can be applied along to different system architectures and hardening techniques:

- Time redundancy. The critical task is repeated at different instants of time by the same microprocessor. This solution allows the detection of permanent and transient faults. In this case, the CPU Checker only requires one CPU Observer module, since tasks are not executed in parallel.
- Hardware redundancy. It consists in using several microprocessors to repeat the critical tasks. The number of CPU Observers depends on the numbers of extra microprocessors. Typically one extra processor is enough for error detection, while two extra processors are required for error correction. This method does not degrade the performance as time redundancy but introduces high area overhead. However, in multi-core systems, this technique is applicable without adding area overhead (see Fig. 19.2).

19.4.1 Experimental Results

Fault injection campaigns over a LEON3 microprocessor have been performed to study the effectiveness of this fault detection technique. In [25, 26], stuck-at faults and SEUs are evaluated. The experiments allow the analysis of the CPU Checker in terms of necessary logic resources, latency and fault coverage.

With respect to the logic resources, the area of the CPU Checker depends on the number of critical tasks that could be executed simultaneously (number of required CPU Observers), and on the number of critical tasks to observe, which directly affects to the memory size. For example, for the case of two CPU Observers and five different critical tasks in a Virtex5 FPGA from Xilinx, the CPU Checker requires 416 FFs and 467 Look-up Tables (LUTs), what are minimal resources compared to those required to implement a microprocessor.

Latency depends on the granularity of the critical tasks to observe. The latency would be minimal if every instruction is considered as a critical task. However, this involves a high area overhead. The latency would be maximum if the complete application is considered as the critical task, since then the faults are detected after the second run. Furthermore, the hardening technique and chosen architecture also affect the fault detection latency. Time redundancy involves high latencies, while hardware and multicore architectures can be used in lockstep to reduce as much as possible the latency.

On the other hand, results are used to measure the fault detection capabilities depending on the part of the trace data that is observed. Fibonacci an elliptic filter applications have been used as benchmarks. The experiments show the following results:

- Case 1: When the complete trace interface is taken into account for calculating the signature, only 0.05 % of the errors are kept undetectable. However, false detected errors are higher with respect to other options (~30 %). Besides, logic resources are also higher.
- Case 2: When the information to observe consists on the opcode, the undetected data increase up to 0.2–0.3 %, but the false detected errors decrease down to ~10 %.
- Case 3: When the information to observe consists on the opcode and the load/store parameters the percentage of undetectable errors is on the same order as in case 1 and the false detected errors is in the range of 15–20 %.

False detected errors do not affect the system reliability although it may affect system performance, depending on the action to be performed after fault detection. For low error rates, the impact of some sporadic error recovery action is negligible. For this reason, false detected errors are not considered as a main concern, being the main objective to reduce as much as possible the number of undetected faults, which does affect directly into the system reliability. Otherwise, the hardware module can be hardened to reduce the chance of false errors.

19.5 Control-Flow Checking

Control-flow checking targets the detection of errors that affect the control flow. Control flow errors are generally very critical, as they may cause the processor to hang indefinitely. In this section, we describe specific techniques for control-flow checking using the trace interface.

19.5.1 PC Prediction

Errors in the Program Counter (PC) are generally critical, as they change the instruction execution flow. The PC prediction technique consists in predicting the next PC value and comparing it with the actual PC value obtained at the trace interface for the next executed instruction [27]. The next PC value can be predicted using the address and the opcode of the current instruction. For a non-branch instruction, the PC must be incremented by the size of the instruction. For an unconditional branch instruction, the PC must be incremented by the branch offset. Finally, for a conditional branch instruction, the PC must be incremented by the branch offset if the branch is taken or by the size of the instruction if the branch is not taken. If these conditions are not met, an error in the execution flow is detected.

Subroutine calls can also be considered by implementing a Stack Replica, which can be limited to few levels to save resources [28]. On a subroutine call, the return address is stored in the Stack Replica. Then, when a return instruction is observed, the predicted next PC value can be recovered from the Stack Replica and checked.

This technique can only detect a subset of errors, such as those affecting the program counter. However, a major property of this technique is that it does not require reference data to compare with. Therefore, it can be implemented with very few hardware resources.

19.5.2 Signature-Based Checking

Several Control Flow Checking approaches have been proposed, which are based on *signature monitoring* [6]. The basic idea is to divide the program into a set of blocks (named *basic blocks*, or BBs), having only one entry-point and only one exit-point: hence, all instructions in a BB are necessarily executed together, in their order. Each basic block has an associated signature that is calculated at compile time and stored in the system. During the execution phase, a run-time signature is calculated and (at the end of the block execution) compared with the reference signature, thus allowing to detect any error affecting the block execution flow. Signature computation and comparison can be performed in different ways (e.g., in hardware or software), characterized by different costs and invasiveness, as well as detection capabilities.

In [29] a method was proposed, which combines signature-based checking with the usage of Debug and Trace features: control flow checking can be performed by an external hardware module that monitors the sequence of instructions executed by the processor through the trace interface and compares it with the reference sequence of instructions to detect possible control flow errors. Experiments showed that a few checks performed when a branch is executed and at the end of each basic block can detect a high percentage of control flow errors.

In particular, the following two checks can be implemented:

- *Check #1*: For every instruction, the external module checks whether the instruction is a (conditional or unconditional) jump instruction or not; in the former case, the new value of the PC must be either the address of the following instruction, or the target one (for conditional branches; for unconditional ones, it must be the target address); in the latter case, the new value of the PC must be the address of the following instruction;
- *Check #2*: Each time the end of a BB is reached, the external module checks whether the signature computed out of the machine codes of the instructions executed during the BB matches a pre-computed one, which is stored in a table inside the module itself.

These checks can be effectively performed by just tracing the values hold by the Program Counter and the Instruction Register.

The major advantages of the proposed method lie first of all in its low intrusiveness, since it does not require any change in the processor or in the software it runs; moreover, the experimental results show that the method can detect a high percentage of control flow errors caused by bit flips in the processor internal flip flops. Latency of fault detection is also very limited. The required external hardware module is relatively small and is independent on the application software. Finally, the method is effective even if the processor uses caches. The major limitation of the method lies in the size of the table storing the signatures associated to the different BBs composing the application code. Clearly, a trade-off can be made between the size of this table and the achievable fault coverage, assuming that no check is performed for BBs whose signature is not stored in the table.

In [30] the same method is improved by allowing it to self-learn the required information about the organization of the application code in BBs and the value of the related signatures. In practice, the method does not require any signature precomputation out of the considered code: when the application is run, the monitor starts tracing the instructions and each time it finds a BB, checks whether the processor already executed it, or not. In the former case, no check is performed, and the signature of the BB is computed and stored in the table; in the latter case, the table is accessed and the usual check is performed. Moreover, the external module may be instructed to only store the signatures of a subset of the BBs; in this case it automatically and dynamically selects the BBs to be stored in the table, e.g., according to the frequency of their execution. In this way, no pre-processing of the application code is required to compute the BB signatures, an optimal trade-off is achieved between the table size and the obtained results and a higher fault coverage can be achieved.

19.5.3 Dual Control-Flow Monitoring

An approach to check the control-flow without the need of additional information for comparison consists in observing the control-flow at two different points. This technique is called dual control-flow monitoring [31]. In this approach, the instruction

flow of the microprocessor is captured both upstream at the bus between the memory and the microprocessor and downstream at the trace interface. The upstream interface observes the address and opcode of each instruction at the cache bus or at the system bus if no cache is used. The downstream interface observes the address and opcode of each instruction at the trace interface, right after execution. If an error corrupts the instruction flow within the processor, it can detected by comparing the downstream instruction flow with the upstream instruction flow.

Dual control-flow monitoring is quite effective in highly pipelined processors, where instructions travel through the pipeline. An error which occurs in the PC or the Instruction Register (IR) at any stage in the pipeline will finally be observed at the trace interface and can be detected by comparing the trace interface output with the memory address and instruction collected at the fetch stage.

Note that errors in the PC at the fetch stage may not be detected with this approach, as the fetch PC is issued by the processor. However, these errors can be covered by using the PC prediction technique in combination with dual control-flow monitoring.

The monitor can also check the time tag and the trap and error flags provided by the trace interface. The time tag is used as a watchdog to detect hang errors. The trap and error flags can also be used to detect illegal traps caused by invalid instruction or invalid memory addresses.

The dual control-flow monitoring technique does not require additional information for checking, because it compares incoming instructions with outcoming instructions. Additionally, it does not affect performance. The area overhead due to the monitor circuit may vary depending on the processor and the characteristics of the interfaces.

19.5.4 Experimental Results

The effectiveness of control-flow checking using an external hardware module attached to the trace interface has been tested for several soft-core processors. In [28], the PC prediction technique was tested for PicoBlaze processor. The results show that this technique can detect between 40 and 50 % of the total errors, including both SEUs and SETs. In particular, PC prediction makes a very good job at detecting hang errors, i.e., errors that provoke abnormal program termination or an infinite loop. By combining this approach with software hardening, the percentage of detected errors can typically reach 99 %.

For more complex processors, the dual control-flow monitoring technique is very effective. An extensive fault injection campaign has been performed for LEON3 microprocessor using several code benchmarks. A hardened version of the code was implemented with a duplication approach based in [11]. Injection of SEU and SET were performed with AMUSE tool [32], which enables large fault injection campaigns. Additional results can be found in [31].

Table 19.1 Fault-injection results using unhardened software

Elements	Faults injected (M)	Errors observed	SDC	Hang	Errors detected
PC & IR	4.8	1,643,534	0	0	1,643,534 (100 %)
Other registers	20.9	1,571,582	752,826	98,816	719,940 (45.8 %)
All registers	25.8	3,215,116	752,826	98,816	2,363,474 (73.5 %)
SET	329.9	3,290,266	1,019,130	51,517	2,219,619 (67.5 %)

Table 19.2 Fault-injection results using hardened software

Elements	Faults injected (M)	Errors observed	SDC	Hang	Errors detected
PC & IR	10.4	3,092,329	0	0	3,092,329 (100 %)
Other registers	45.2	3,161,890	254,560	192,728	2,714,918 (85.9 %)
All registers	55.6	6,254,219	254,560	192,728	5,806,931 (92.9 %)
SET	711	7,138,702	226,966	64,080	6,847,656 (95.2 %)

Tables 19.1 and 19.2 show the results for the fault injection campaigns of the unhardened version and the hardened version respectively. For Tables 19.1 and 19.2, column 2 reports the number of injected faults, column 3 reports the number of observed errors, columns 4 and 5 report SDC (Silent Data Corruption) and Hang errors, respectively, and column 6 reports the detected errors. Both Tables 19.1 and 19.2 report results regarding where the faults are located: row 2 reports results for PC (Program Counter) and IR (Instruction Register), row 3 reports results for any other register, and row 4 reports results for all registers (including PC and IR). Finally, The last row reports the results for the SET experiments.

These results demonstrate that this technique can detect all errors produced by faults injected in the PC and IR registers for all stages. Even though they form just a small portion of the total internal flip-flops in the processor (less than 20 % in the case of a LEON3 using a minimal configuration), they are usually very critical. In fact, the results in [31] show that about 50 % of observable errors are produced in these registers. On the other hand, these include all control-flow errors, according to [16]. The dual control-flow monitor can also indirectly detect errors injected in other registers, covering more than 70 % of the total observable errors.

The signature monitoring approach described in Sect. 19.5.2 was experimentally evaluated on a miniMIPS processor, assuming that a debug interface similar to the one available in the LEON3 processor is available. The fault models defined in [18], which specifically focus on the Control Flow Errors, were considered. Results show that the method achieves 100 % fault coverage when the signature table is large enough to store all BB signatures, and that the fault coverage decreases quite slowly when the table size reduces. On the other side, the size of the external hardware module performing the checks (not including the table) is limited to about 2 % of the total size of the processor. Its complexity only slightly increases when the improved method is implemented, while in this way very high fault coverage figures can be achieved even with a table whose size allows storing the signatures of a fraction of the total set of BBs.

19.6 Conclusions

This work presents a detailed analysis of the capabilities of the microprocessor's trace interface as a microprocessor observation point. The trace interface, which is commonly found in many microprocessors, is a non-intrusive observation point that provides quite useful information for microprocessor error detection. In addition, observation is performed after instructions are executed and errors that occur in the pipeline can be detected.

Several strategies can be used to detect errors with the information provided by the trace interface. All of them can provide high error detection coverage with no intrusiveness. Techniques have been tested on different microprocessors (miniMIPS, PicoBlaze and LEON3) which also demonstrate that trace interface can be used effectively in a broad range of microprocessors.

Acknowledgment This work was supported in part by the Spanish Government under project PHB2012-0158-PC.

References

1. Aleksejev I, Jutman A, Devadze S, Odintsov S, Wenzel T (2012) FPGA-based synthetic instrumentation for board test. In: Proceedings of IEEE international test conference, Austin
2. Azambuja JR, Altieri M, Becker J, Kastensmidt FL (2013) HETA: hybrid error-detection technique using assertions. IEEE Trans Nucl Sci 60(4):2805–2812
3. Azambuja JR, Pagliarini S, Altieri M, Kastensmidt FL, Hubner M, Becker J, Foucard G, Velazco R (2012) A fault tolerant approach to detect transient faults in microprocessors based on a non-intrusive reconfigurable hardware. IEEE Trans Nucl Sci 59(4):1117–1124
4. Azambuja JR, Lapolli A, Rosa L, Kastensmidt FL (2011) Detecting SEEs in microprocessors through a non-intrusive hybrid technique. IEEE Trans Nucl Sci 58(3):993–1000
5. The Nexus 5001 forum standard for a global embedded processor debug interface, IEEE-ISTO 5001–2003, 2003, ver. 2.0
6. Nicolaidis M (ed) (2011) Soft errors in modern electronic systems. Springer, New York
7. Mahmood A, McCluskey E (1988) Concurrent error-detection using watchdog processors. IEEE Trans Comput 37(2):160–174
8. Michel T, Leveugle R, Saucier G (1991) A new approach to control flow checking without program modification. In: Proceedings of the 21st FTCS-21, June 1991, pp 334–341
9. Bergaoui S, Leveugle R (2009) IDSM: an improved control flow checking approach with disjoint signature monitoring. In: Proceedings of the Conference on DCIS, Nov 2009, pp 249–254
10. Rebaudengo M, Reorda MS, Torchiano M, Violante M (1999) Soft-error detection through software fault-tolerance techniques. In: International symposium on defect and fault tolerance in VLSI systems, pp 210–218
11. Cheynet P, Nicolescu B, Velazco R, Rebaudengo M, Sonza Reorda M, Violante M (2000) Experimentally evaluating an automatic approach for generating safety-critical software with respect to transient errors. IEEE Trans Nucl Sci 47(6):2231–2236
12. Engel H (1997) Data flow transformations to detect results which are corrupted by hardware faults. In: Proceedings of the IEEE high-assurance system engineering workshop, pp 279–285

13. Benso A, Chiusano S, Prinetto P, Tagliaferri L (2000) A C/C++ source-to source compiler for dependable applications. In: Proceedings of the IEEE international conference on dependable systems and networks, pp 71–78
14. Nicolescu B, Velazco R (2003) Detecting soft errors by a purely software approach: method, tools and experimental results. In: Design, automation and test in Europe conference and exhibition, pp 57–62
15. Hiller M (2000) Executable assertions for detecting data errors in embedded control systems. In: Proceedings of the IEEE international conference on dependable systems and networks, pp 24–33
16. Vemu R, Gurumurthy S, Abraham JA (2007) ACCE: automatic correction of control-flow errors. In: Proceedings of the international test conference, pp 1–10
17. Chielle E, Azambuja JR, Barth RS, Almeida F, Kastensmidt FL (2013) Evaluating selective redundancy in data-flow software-based technique. IEEE Trans Nucl Sci 60(4):2768–2775
18. Alkhalifa Z, Nair VSS, Krishnamurthy N, Abraham JA (1999) Design and evaluation of system-level checks for on-line control flow error detection. IEEE Trans Parallel Distrib Syst 10(6):627–641
19. Vemu R, Abraham JA (2006) CEDA: control-flow error detection through assertions. In: Proceedings of the 12th IEEE international on-line testing symposium, pp 151–158
20. Mukherjee S, Weaver C, Emer J, Reinhardt S, Austin T (2003) A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In: Proceedings of the 36th international symposium microarchitecture, Dec 2003, pp 29–40
21. GRLIB IP core usefs manual. ver. 1.0.22, Aeroflex Gaisler, Jan 2010
22. Embedded Trace Macrocell, ETMv1.0 to ETMv3.4, architecture specification. ARM Limited, 2007
23. Xilinx MicroBlaze™ Trace Core (XMTC) (v1.00c), Xilinx, 2009
24. www.gaisler.com
25. Portela-Garcia M, Grosso M, Gallardo-Campos M, Sonza Reorda M, Entrena L, Garcia-Valderas M, Lopez-Ongil C (2012) On the use of embedded debug features for permanent and transient fault resilience in microprocessors. Microprocess Microsyst 36(5):334–343
26. Grosso M, Reorda MS, Portela-Garcia M, Garcia-Valderas M, Lopez-Ongil C, Entrena L (2010) An on-line fault detection technique based on embedded debug features. In: Proceedings of the 16th IEEE on-line testing symposium, pp 167–172
27. Parra L, Lindoso A, Portela M, Entrena L, Grosso M, Reorda MS (2011) Control flow checking through embedded debug interface. In: Proceedings of the 26th conference on design of circuits and integrated systems, pp 339–343
28. Parra L, Lindoso A, Portela M, Entrena L, Restrepo-Calle F, Cuenca-Asensi S, Martinez-Alvarez A (2014) Efficient mitigation of data and control flow errors in microprocessors. IEEE Trans Nucl Sci 61(4):1590–1596
29. Du B, Reorda MS, Sterpone L, Parra L, Lindoso A, Portela-Garcia M, Entrena L (2013) Exploiting the debug interface to support on-line test of control flow errors. In: Proceedings of the 19th IEEE on-line testing symposium (IOLTS), July 2013, pp 98–103
30. Du B, Reorda MS, Sterpone L, Parra L, Lindoso A, Portela-Garcia M, Entrena L (2014) A new solution to on-line detection of control flow errors. In: Proceedings of the 20th IEEE on-line testing symposium (IOLTS), July 2014, pp 105–110
31. Parra L, Lindoso A, Portela-Garcia M, Entrena L, Du B, Reorda MS, Sterpone L (2014) A new hybrid nonintrusive error-detection technique using dual control-flow monitoring. IEEE Trans Nucl Sci 61(6):3236–3243
32. Entrena L, Garcia-Valderas M, Fernandez-Cardenal R, Lindoso A, Portela Garcia M, Lopez-Ongil C (2012) Soft error sensitivity evaluation of microprocessors by multilevel emulation-based fault injection. IEEE Trans Comput 61(3):313–322

Part VI
Parallel Architectures and GPUs

Chapter 20
Soft-Error Effects on Graphics Processing Units

Paolo Rech, Daniel Oliveira, Philippe Navaux, and Luigi Carro

Abstract Graphics Processing Units (GPUs) evolved from graphics-specific devices to general-purpose computing accelerators that scientists use to run large-scale simulations. Additionally, GPUs are very attractive for safety-critical applications that extensively use signal or image processing.

Unfortunately, while the performance and efficiency of GPUs are well established, their resilience characteristics in a large-scale computing system and safety critical-application have not been fully evaluated. The presence of complex scheduling circuitry, for instance, may significantly increase the parallel code error rate. Moreover, the parallel architecture of GPUs introduces novel radiation experiment challenges that need to be solved.

In this Chapter we present a detailed radiation test setup for GPUs, including some recommendations for parallel devices experiments. We also present some experimental results on the radiation sensitivity of modern GPUs, considering both low-level static analysis and typical parallel application behaviors under radiation.

20.1 Introduction

Graphics Processing Units (GPUs) are electronic devices designed to perform high-performance stream processing and provide very high computational power combined with low cost, reduced power consumption, and flexible development platforms.

In order to achieve the proposed objective, GPUs manipulate a large number of memory locations, and are typically able to execute several elementary tasks in parallel at high speeds [1, 2]. Due to their highly parallel structure, GPUs are more effective than general-purpose CPUs when large blocks of data need to be processed in parallel. GPUs have then recently become popular not only for graphical applications, but also in the High Performance Computing (HPC) market [3, 4].

P. Rech (✉) • D. Oliveira • P. Navaux • L. Carro
Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre, Brazil
e-mail: prech@inf.ufrgs.br; dagoliveira@inf.ufrgs.br; navaux@inf.ufrgs.br; carro@inf.ufrgs.br

F. Kastensmidt, P. Rech (eds.), *FPGAs and Parallel Architectures for Aerospace Applications*, DOI 10.1007/978-3-319-14352-1_20

Scientists have begun to take advantage of the unprecedented amount of parallelism available in GPUs to expedite their scientific simulations and to derive scientific insights more quickly. For example, Titan, the world's second fastest supercomputer for open science in 2014, consists of 18,688 GPUs that scientists from various domains such as astrophysics, fusion, climate, and combustion use routinely to run large-scale simulations. Moreover, in some safety-critical applications, such as automotive, avionics, space and biomedical, GPUs would be very suitable. As an example, the *Advanced Driver Assistance Systems* (ADAS), which are increasingly common in cars, make an extensive usage of images (or radar signals) coming from external cameras and sensors to detect possible obstacles, triggering the breaks automatically if necessary. Starting in 2015, only vehicles equipped with ADAS will be eligible to receive the highest security level from Euro-NCAP [5], one of the most authoritative car evaluation agencies in Europe. A Low-power System on Chip including a GPU core, like the NVIDIA Tegra, is likely to be the computational core of ADAS. Airbus is finalizing the ARAMIS project, aimed at integrating of all the electronics required to implement the collision avoidance system into a single board including a GPUs core [6]. Unfortunately, the European Aviation Security Agency (EASA) does not accept multicores chips with more than two cores on an aircraft, yet. The main reason for such a limitation on parallelism from EASA is that a standardize reliability evaluation protocol has not yet been developed. Our paper moves on the direction of understanding the reliability of GPUs, giving novel insights on their behaviors when exposed to ionizing radiation.

In both application scenarios (HPC and safety-critical embedded applications), GPUs reliability is a major concern. As the newest GPUs are built with cutting-edge technologies, offer a great amount of resources, and operate at extremely high frequencies, they may be particularly susceptible to experience radiation-induced errors, including those originating from the terrestrial neutron radiation environment [7, 8]. On safety-critical applications, the reliability qualification of GPUs is essential to evaluate if the device is compliant with the project specifications. Hardening techniques like Error Correction Codes (ECC), duplication with comparison, triplication, or Algorithm Based Fault Tolerance [9, 10] could eventually be applied if the error rate of GPUs is found to exceed the reliability requirement. Supercomputers are composed of thousands of devices that work in parallel and, thus, the probability of having at least one radiation-induced corruption is very high. Hardening strategies become mandatory even for HPC application with the specific constraint to avoid the introduction of useless overhead. Evaluating precisely the radiation-induced error rate of a code executed on a GPU is then of extreme importance as it allows to evaluate the trade-off between the hardening strategy detection/correction capabilities and the introduced computational overhead.

An intense research discussion on GPUs radiation sensitivity has recently started [11], focusing on the probability of caches and registers failures, tracking errors propagation to the output [12–14] as well as devising software and architectural techniques to harden GPU-based systems [15]. Most of the research done on GPU reliability is based on fault-injection simulations [12, 13, 16], on field tests [14], or radiation experiments [17, 18]. Experimental data presented in the later highlights

that the corruption of resources shared among parallel threads like caches or critical resources, such as the scheduler, may reduce the GPU reliability and generate a large number of multiple errors in the output.

20.2 GPUs Architecture and Radiation Vulnerability

Modern GPUs are divided into various computing units, named *Streaming Multiprocessors* (SM), each of which has the ability to executing several threads in parallel (see Fig. 20.1). Each basic computing unit (named *CUDA core* in NVIDIA devices) in the SM executes one thread with dedicated registers, avoiding complex resource sharing or the need of long pipelines [2].

It is the programmer's task to divide the instantiated threads into a *grid of blocks* when designing a *kernel* to be executed on a GPU. It is easy to modify the thread distribution, as the block size and the grid size are both parameters that have to be specified when launching a CUDA kernel to be executed on a GPU.

The number of blocks assigned to a Streaming Multiprocessor in the GPU will depend on the number of registers, on the amount of shared memory available in the SM, and on the resources required by each block to be executed. On GPUs built with the *Fermi* architecture, like the ones used in the presented study, the number of blocks assigned to a SM cannot exceed 8 while in *Kepler* devices up to 16 blocks can be assigned to a SM.

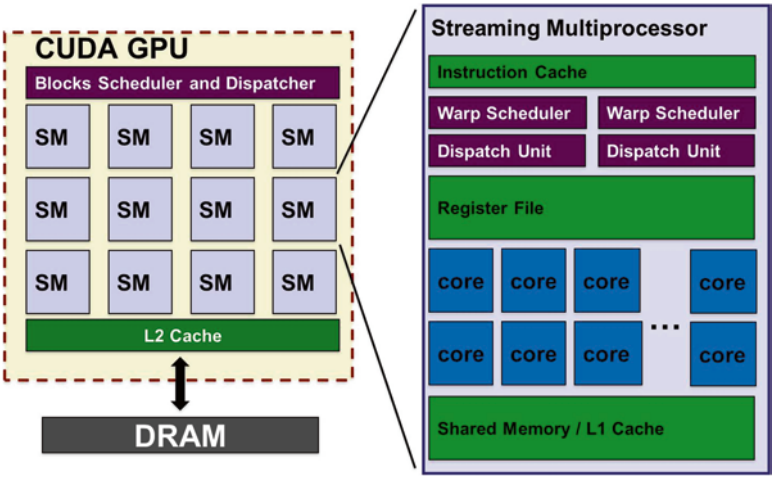


Fig. 20.1 A representative CUDA-based GPU architecture, composed of an array of SMs that share L2 cache and external DRAM. In the SM, warps are assigned to CUDA cores by two schedulers. A thread has dedicated register files and shares with threads in the same SM a shared memory, L1 and instruction cache

Some blocks will be queued for later computation if the grid size exceeds the number of blocks that can be dispatched among the SMs available in the GPU. Before dispatching a queued block to the first SM that becomes available, the GPU's block scheduler needs to check if some SM completed the current block execution and, if so, it transfers the results to the on-board DDR memories. The queued block is then assigned to the SM, the input data is eventually read from the DDR, and, finally, the queued block execution is triggered and synchronized [19].

GPUs with CUDA capabilities 2.0 or 3.5, as the vectors of this study, can execute up to 64 and 192 parallel threads in an SM in a computing cycle, respectively. If the block size exceeds 64 or 192, the execution of some threads will be delayed until the computation of the preceding warps of the block has been completed. It is worth noting that the next block to be treated will be assigned to the SM only when all threads in the current block have been processed. Therefore, if the number of threads in a block is not a multiple of 64, in the last cycle the SM will execute less than the maximum amount of threads, wasting parallel capabilities.

Each SM disposes of two schedulers (see Fig. 20.1). At every instruction issue time, the first scheduler issues one instruction for some warp with an odd ID and the second scheduler issues one instruction for those with an even ID. When double-precision floating-point instructions have to be executed, like in the codes analyzed in this paper, the second scheduler cannot issue any instruction.

A parallel code to be executed on a GPU is typically composed of several independent threads, all executing the same set of instructions on dedicated memory location. Increasing the amount of threads brings then higher throughput to the application. To do so, the programmer can choose either to increase the block size, which will require more computational effort in each SM and delay the assignment of the next blocks, or to increase the grid size, thus having more blocks to be dispatched. The GPU parallel management is strictly related to the chosen thread distribution. The scheduling and computational load required for blocks and warps assignment, as well as resources distribution, are strictly related to the chosen grid and block sizes, which is then likely to influence also the GPU radiation response.

When evaluating the radiation reliability of GPUs it essential to consider and analyze the effects of different thread distributions in the GPU parallel management and the consequent variation on the device cross section. Such an evaluation will detect the distribution, in terms of grid size and block size, which offers lower cross section and higher probability of completing computation correctly. For instance, reducing the number of threads available while increasing the workload of each thread lowers the dispatcher load, which is likely to reduce the GPU cross section, but the recourses distribution, caches requirements, memory access latencies will be affected by the changed threads complexity, with non-obvious effects on the radiation-induced error rate.

There are a number of ways that neutron or ionizing particle in general strikes perturb GPUs. A neutron may induce bit flips in memory elements as well as transient voltage spikes in logic computing resources or control circuitry. GPUs use large caches and complex task schedulers to manage the parallelism on the system. While task scheduling is performed in software as part of the operating system tasks

on CPUs, GPUs have dedicated, hardware-based task schedulers internally. The caches and schedulers are particularly critical for parallel processors and failures in these areas can lead to multiple output errors or functional interruptions [17, 19].

A radiation strike leads to one of the following outcomes: (1) no effect on the program output (the error is masked or corrupted data is not used), (2) Silent Data Corruption (incorrect program output), (3) program crash, (4) system hang (the GPU has to be rebooted to restore its functionality). Out of these outcomes, (2) is harmful as it remains undetected and unpredictable, while (3) and (4) are to be strictly avoided in safety-critical applications and in HPC, as they lead to loss of functionality, performance penalties, and possible data loss.

From a radiation test point of view, the CUDA cores are isolated such that a single radiation-induced event in one of them will only corrupt the thread assigned to it. Threads that follow the corrupted one or assigned to CUDA cores near the struck one will not be affected. Nevertheless, errors in the L1 cache or shared memory are likely to affect several threads in the SM, as all threads can access that data. Similarly, errors in the L2 cache, shared among all SMs, are likely to affect several blocks of threads. A radiation strike in one of the schedulers may lead to wrong task assignments forcing threads to work on wrong data, to synchronization issues leading to incomplete results, or to conflicts or control flow errors that induce kernel panic or crashes. Instantiating a higher number of parallel threads typically reduces the code execution time but increases the scheduler strain required to manage execution and resource sharing. Imposing a higher scheduler strain (either on the warp or block scheduler) has the drawback of increasing the probability of having the scheduler affected by radiation [19].

Only the major storage structures of GPUs for HPC applications are protected with Single Error Correction Double Error Detection (SECDED) Error-Correcting Code (ECC) including device memory, L2 cache, instruction cache, register files, shared memory, and L1 cache. However, some resources are left uncovered, e.g., logic, queues, the thread block scheduler, warp schedulers, instruction dispatch units, and interconnect network. Unfortunately, the details of resilience support for these structures are considered business-sensitive by vendors and, hence, unavailable. It's worth noting that GPUs for embedded systems typically do not include any reliability system.

20.3 Experimental Setup

Radiation experiments were performed in the VESUVIO neutron facility at ISIS, Rutherford Appleton Laboratories (RAL) in Didcot, UK and at LANSCE, Los Alamos National Laboratory, Los Alamos, NM, USA (Fig. 20.2). Both these facilities provide a neutron spectrum that has been demonstrated to be suitable for emulating the atmospheric neutron flux [20]. The available neutron flux was of about 5×10^4n/(cm^2 s) in VESUVIO and 5×10^6n/(cm^2 s) for energies above 10 MeV. Irradiation was performed at room temperature with normal incidence.

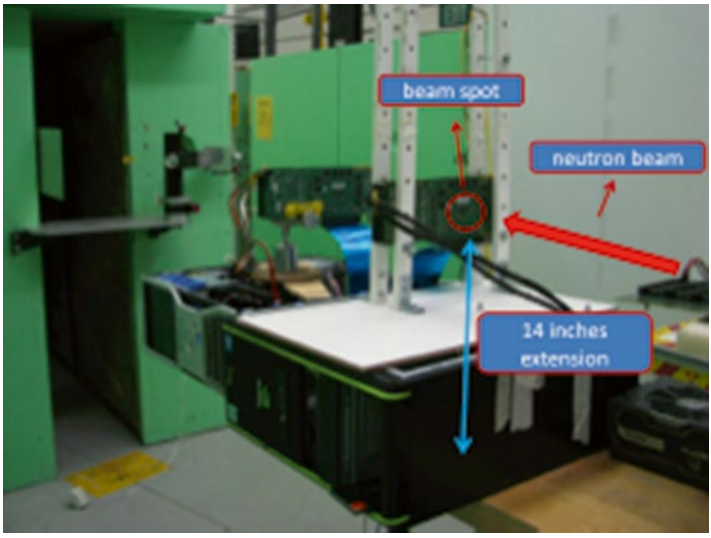


Fig. 20.2 GPU radiation test setup inside the ICE House II at LANSCE, Los Alamos National Laboratory, Los Alamos, NM, USA

It is worth noting that the neutron flux the GPUs receive during radiation experiments is 10 orders of magnitude higher than the atmospheric neutron flux (which, according to the JEDEC standard, is of about 13 n/(cm^2 h) at sea level). Experiments should then be carefully designed to ensure that the probability of more than one neutron generating a failure in a single code execution remains practically negligible. As a general advice, the observed error rates should be lower than 10^{-2} errors/execution. The error rate can be lowered either by reducing the flux that reach the device under test (for instance installing the GPU farther from the particles source) or reducing the amount of data elaborated and workload. On a GPU it is typically easy to design scalable tests. In fact, the homogenous structure of the device and code allow the programmer to parametrize the number of instantiated parallel processes and, consequently the workload.

Since a much lower neutron flux may hit a GPU in a realistic environment, it is highly likely to not have more than one corruption during a single execution. We can, therefore, scale the experimental data in the natural radioactive environment without introducing artificial behaviors.

The beam was focused on a spot with a diameter of 2 cm plus 1 cm of penumbra. The size of the spot is sufficient to uniformly irradiate the whole GPU chip, leaving the on-board DDR and power circuitry of the GPU out of the beam. This is essential for preventing neutron-induced errors on power switches to compromise the experiment. Moreover, having the DDR memory out of the beam allowed us to use it as a safe temporary storage for test results, as we will detail in the following.

The GPU can be fully controlled by a normal desktop-PC through a 2.5 GHz PCI-Express bus. We put an extension of 20 cm to the PCI-Express so to prevent

scattering neutrons to affect the PC functionalities (Fig. 20.2). The extension was provided with fuses to prevent current spikes from the GPU to reach the PC motherboard. Power was given to both the GPU and bus with current-controlled supplies to further prevent neutrons-induced latchups from destroying the device. The described test setup is low-cost, but very effective and gives precise data on the radiation sensitivity of the GPU.

NVIDIA CUDA programming strategy allows integrating in a single application both CPU and GPU codes. The key CUDA operations are *thread synchronize*, *cudamalloc*, and *cudamemorycopy*. The former is used to trigger the start of GPU execution, while the others are used to exchange data, allowing the CPU to access the DDR available on the GPU board. The GPU can then be treated as a stand-alone device that, once initialized, executes the provided instructions without the need of external stimuli.

The role of the PC in this kind of test is just to initialize the board under test, download the results, and check for mismatches when the test is finished. The sequence of a generic test on a GPU can be detailed as follows:

1. **Initialization**: the PC loads instructions and/or data on the GPU;
2. **Test**: the PC triggers the GPU with the *thread synchronize* command. The GPU actually executes the code while the PC is in idle state. When the test finishes, the GPU loads the results in the DDR. In this step the GPU simply maintains data when performing a static test for measuring the sensitivity of memory elements;
3. **Readback**: the PC, using *cudamemorycopy* operation, downloads from the GPU DDR the experimental data and checks for mismatches.

Thanks to the extreme high frequency of both the PC and PCI-Express, steps 1 and 3 can be performed very quickly (order of milliseconds), making it very unlikely for a neutron to generate an error during their execution. It is then possible to perform steps 1–3 continuously under radiation to gain a statistically significant amount of data. This is particularly useful in neutron radiation test, as normally the beam cannot be switched off easily. In fact, in most of the neutron accelerator facilities the beam opening/closing procedure takes several seconds to be fully completed. In the ISIS particular case, two concrete shutters are used to block the beam and the opening or closing process takes about 1 min to be accomplished. It would be then rather impractical to stop the beam before each test initialization or readback.

A software and a hardware watchdog were included in the setup. The software watchdog monitors a time-stamp written by the application running on the GPU. If the time-stamp is not updated in 10 s the GPU application is killed and launched again. Such a watchdog is required to detect and manage radiation-induced program crashes or control flow errors that prevent the GPU from completing the assigned tasks (e.g., the GPU enters an infinite loop). The triggering of the software watchdog is counted as a *Functional Interruption*. The hardware watchdog is an Ethernet controlled switch that performs a power cycle of the host computer if the host computer itself does not acknowledge any ping requests in 10 min. The hardware watchdog is necessary as radiation can corrupt the PCIe controller on the GPU

board as well, possibly causing the host computer to hang. Finally, it is worth noting that the irradiated GPU should not be set as the primary graphic card of the controlling PC. This is because the operating system running on the PC will probably crash if the primary graphic card experiences a latchup or a functional interruption, and a manual power cycle of the PC will be necessary.

20.4 Experimental Results

20.4.1 Basic Structures Test

As a first characterization of the tested devices, the baseline tests to conduct are the characterization of neutron-induced errors in the L1/shared memory, the L2 cache when used with L1/shared, and also an adder that fully exercised the carry functionality.

The Devices Under Test (DUTs) are three commercial-off-the-shelves *Kepler* K20 GPUs designed by NVIDIA in a 28 nm technology node. The K20 is among the current highest performing NVIDIA GPUs, and acts as an accelerator in two of the ten fastest supercomputers, including Titan. The DUT is composed of 15 SM, each of which is divided in 192 CUDA cores. The K20 features a 706 MHz SM core clock, 1.25 MB L2 cache, a total of 832 KB in L1 cache, and a total of 3.25 MB of register file storage. The K20 is equipped with a Single Error Correction Double Error Detection (SECDED) ECC mechanism. Tests should be performed both having the GPU Error Correction Code (ECC) disabled and enabled. By disabling error correction one is able to determine the base sensitivity of the cells without error correction.

As GPUs are used for massively parallel operations, it is necessary to implement the test code so that it will properly distributed across the GPU, including the memory and compute infrastructure. The 64 KB of on-chip memory was configured for all tests to maximize the amount of shared memory per SM. There is 48 KB of shared memory and 16 KB of L1 cache in the on-chip memory on each SM. The grid size is maximized to the number of SMs, which is 15 for the K20. The block size is maximized to the number of CUDA cores, which is 192 for the K20. We sized the test data so that memory was utilized to the maximum extent possible. In practice, this meant almost filling the shared memory portion of on-chip memory, but only filling about half of L2 cache. By using only half of L2, one is able to maintain one thread per core per kernel.

The configurable L1/shared memory cache test set an array of elements in shared memory at the beginning of test execution. The instrumentation code that checks for errors forces the array to remain resident in shared memory over a reasonably long time and minimizes rewrites over possible errors in shared memory before they were recorded.

The L2 cache test also uses L1/shared memory. The data in L2 were persistent throughout the test, but the data in L1/shared were overwritten constantly. The test read the array in L2 into shared, where the instrumentation code would check for

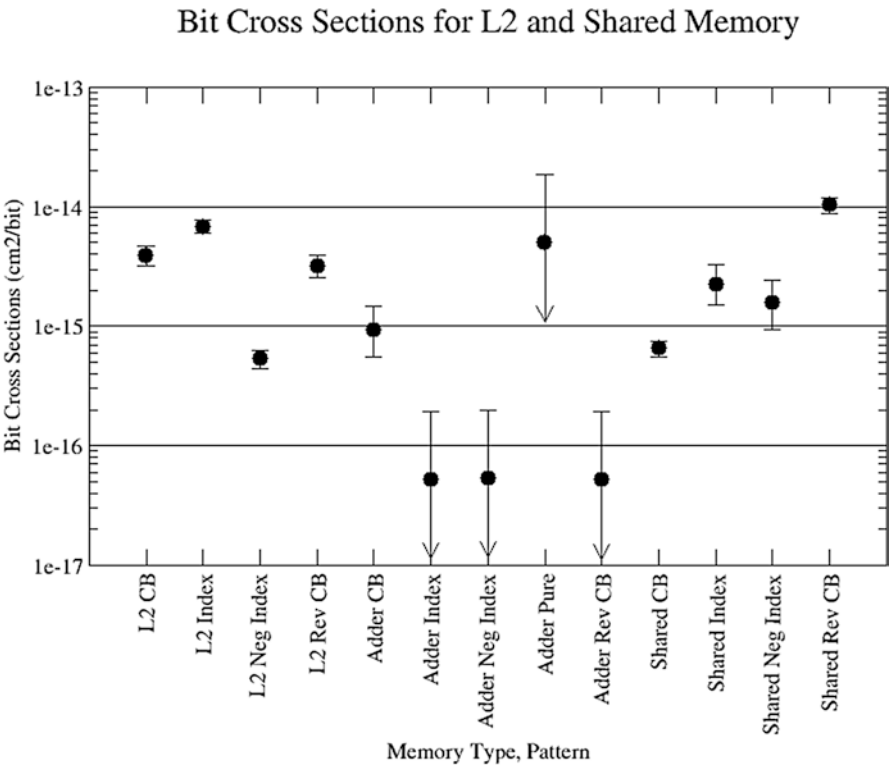


Fig. 20.3 Bit cross sections for three different test codes with four different test patterns [21]

correctness and correct errors. The read/check/correction cycle iterated many times before the kernel ended. This keeps the array resident in L2 cache over a reasonably long time and minimizes rewrites in L2.

Finally, the adder test is used to evaluate the add-carry circuit sensitivity. Only add-carry, an increment, bit shifts and a NOT were used in this test.

Figure 20.3, taken from [21], shows the bit cross-sections for the caches (L2 and Shared Memory/L1) and the adder. Only one error was observed in the adder test, possibly not enough to build a significant statistic. On the contrary, all the other experiments provided more than 100 errors each, which result in a good statistic. While there are differences in sensitivities based on the type of memory and test pattern, the differences are not large.

Other experiments, like the one proposed in [22] show a not negligible pattern dependence on GPU memory structures. In particular, the L2 cross section for the K20 depends on the written pattern. For the 0s pattern, the L2 cross section was found to be approximately 40 % higher than that of the 1s pattern. This means that L2 bits set to 0 are more likely to be corrupted by high-energy neutrons than bits set to 1. The observed dependence on test pattern is due to the asymmetries intrinsic in

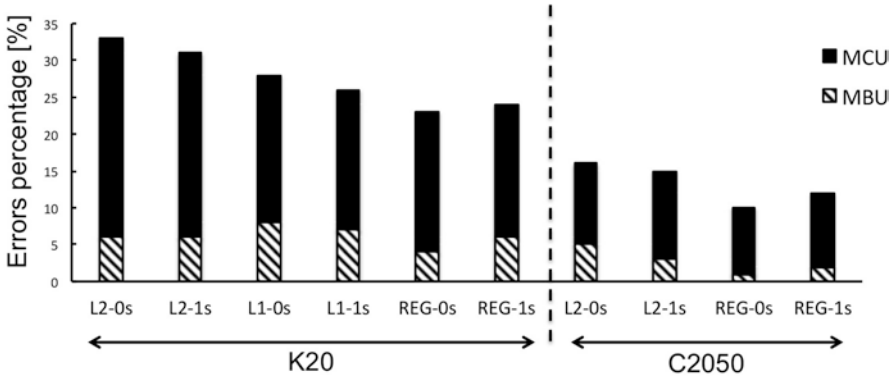


Fig. 20.4 Percentage of errors found to be MCU in the memory structures of K20 and C2050. MBUs are those MCUs with more than one bit corrupted in the same word. C2050 L1 test (not included in the picture) did not provide a statistically significant amount of failure [22]

the cache cell design. This specific result can be achieved only through radiation experiments, and is fundamental to precisely evaluating the resilience of GPUs.

The test procedure described in the previous section allows also distinguishing between Multiple Cells Upset (i.e., multiple errors generated by a single impinging particle) and Multiple Bit Upset (i.e. multiple errors belonging to the same word generated by a single impinging particle).

For this test two different devices were used: the K20 and the C2050. The C2050 belongs to the *Fermi* family, and was released 2 years earlier than the K20. As K20 is built in a 28 nm technology node while the C2050 in a 40 nm node the comparison among the two devices sensitivity to multiple errors is of particular interest. In fact, we will check wherever the shrink of transistors dimension effectively increases the probability of having multiple failures.

Figure 20.4 shows the percentage of events that were found to be MCU and MBU (L1 test did not provide a statistically significant amount of multiple events on the C2050 and is not included). Whenever more than one bit was found corrupted during a test, an MCU was detected. If the corrupted bits belonged to the same word, an MBU was counted. K20's memory structures are about two times more prone to experience multiple events than C2050's. These results are very reasonable given the small feature size, and many other microprocessor components have higher MCU and MBU rates. For both generations of GPUs, the L2 cache is more likely to experience multiple events, probably because of its dense and compact design. There is no significant pattern dependence on multiple events probability. The reported results are of extreme importance for the tuning of fault injectors as they give the correct probability for multiple events occurrences.

The distinction between MBUs and MCUs is fundamental as it categorizes whether a radiation-induced event could be corrected with the Single Error Correction Double Error Detect (SECDED) ECC mechanism included in the K20 and C2050 devices. MCUs could occur as multiple single errors that would be correctable with

a SECDED ECC, whereas an MBU would be incorrigible. Then, even if about 33 % of neutron-induced events are multiple events in the K20 L2 cache, only 6 % are incorrigible MBUs. It is worth noting that no MBU with more than two bits corrupted was detected in the experimental campaign. For the current GPU generation it is reasonable to believe that the great majority of radiation-induced failures in the memory structures of GPUs can be detected or corrected by the included ECC mechanism. Nevertheless the SECDED ECC may become insufficient if the observed trend of increasing MBU occurrences from a GPU generation to the new one is maintained.

20.4.2 Dynamic Test

The memory structures and adder test discussed in the previous section give important information on the static radiation response of GPUs. Nevertheless, to fully evaluate the device reliability it is also essential to measure its dynamic behavior under radiation. To do so, a code must me run on the device under test, and the code results should be monitored. For the GPU dynamic test we provide a known input to the device and, once computation is completed, results are compared to a precomputed golden copy. When a mismatch is detected a Silent Data Corruption (SDC) occurred while when the GPU fails in providing an output a Functional Interruption (FI) is counted.

There are several benchmarks that can be used for radiation test of GPUs. Most of them are part of HPC code suites that are available on line, like Rodinia or NAS. Here we present results obtained with Matrix Multiplication and FFT, which are typical workloads for parallel devices.

As we will show in the following, the parallel architecture of GPUs is likely to increase the number of elements found to be corrupted at the end of one computation. The observed multiple output errors have a significantly different origin than the ones discussed for memory elements in the static test. When a parallel code is executed, in fact, even a neutron-induced single failure may spread, especially when it affects shared or critical resources.

Matrix Multiplication performs the multiplication of two $2{,}048 \times 2{,}048$ random matrices (A and B) executing $2{,}048 \times 2{,}048$ parallel threads, each in charge of calculating a single element of the resulting matrix following Eq. 20.1.

$$M[i,j] = \sum_{k=1}^{2048} A[i,k] \bullet B[k,j] \tag{20.1}$$

The experimentally obtained neutron-induced error rate of matrix multiplication is 2.75×10^{-2} errors/execution. It's worth noticing that input matrices where stored in the DDR available on the GPU board, which were not irradiated. Output errors are then produced by the corruption of GPU internal memory and logic resources.

We can further study experimental data analyzing the corrupted resulting matrix. Figure 20.5 shows the percentage of faulty executions in which a single error or

Fig. 20.5 Percentage of single and multiple output errors at the output of 2,048×2,048 matrices multiplication executed on a GPU [17]

SINGLE VS MULTIPLE ERRORS	
Single Error [%]	42.29
Errors in a Row [%]	22.86
Errors in a Column [%]	26.85
Random Errors [%]	8.00

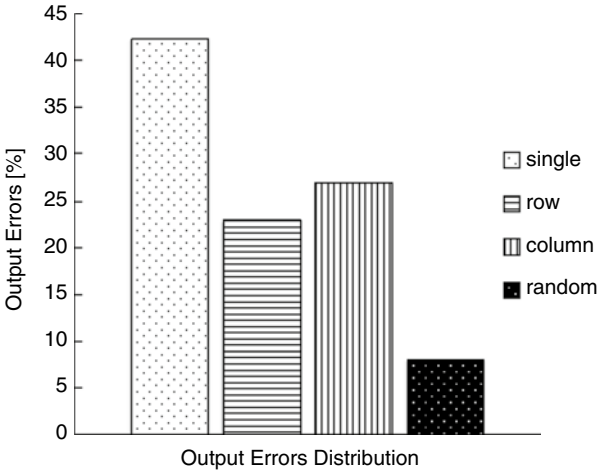
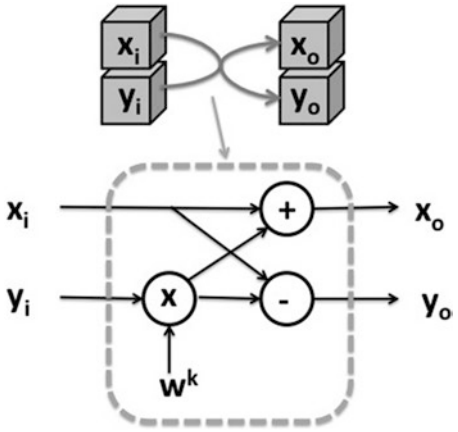


Fig. 20.6 A basic butterfly module used to update two-by-two all the 64 elements composing the FFT. W^k denotes a suitable Nth root of unity [18]



multiple errors were detected on the output matrix. As it can be seen, single output errors are detected in less than 43 % of the cases. This result is of extreme importance as it demonstrates that for modern GPUs the accredited assumption of having just single radiation-induced output errors is no longer valid.

Figure 20.5 shows also the different error patterns we detected when multiple errors affect the output matrix. In most of the cases, multiple errors are distributed on a single row or column, while just in 8 % of the cases errors are randomly distributed (Fig. 20.6).

Errors on single row or column may be due to cache bits corruption. In fact, all the threads in charge of calculating a row of matrix M (similar considerations can be applied to column) take the same row of matrix A but different columns of matrix B as input. To improve the code parallelism, the row of A is stored on the cache of the multiprocessors where the considered threads are executed. Thus, if a bit of that row is corrupted, all the correspondent elements in the row of M will be erroneous. It is worth noticing that the threads in charge of calculating a row of M are not all destined to the same multiprocessor, on the contrary, they are likely to be distributed homogeneously among the 15 microprocessors to maintain a high level of parallelism. During our experiments, in fact, we never observed a whole row of M corrupted. Just some locations in some random locations inside the row were found to be erroneous.

Randomly distributed errors are probably caused by scheduler failure. The scheduler is in charge of designating the group of threads that has to be executed per multiprocessor and of detecting if all the threads have completed computation after the execution. If so, results are presented at the output and another group of threads is executed in the correspondent multiprocessor. In the case of scheduler corruption, the results may be presented even if some threads have not completed computation, leading to wrong results. As shown in Fig. 20.3, just two locations of M were corrupted in the majority of the cases in which randomly distributed errors occurred, and it is very unlikely to have three or four wrong random locations, as this happens on about 1.14 % and 0.57 % of the faulty computations, respectively. As we will detail in the following sections, this information is essential to optimize the proposed hardening strategy and tune its correction capability.

The Fast Fourier Transform parallel code tested implements 512×512 1D-FFTs of 64-points each. The FFT input is composed of a $64 \times 512 \times 512$ double precision floating-point matrix for the real part and a $64 \times 512 \times 512$ matrix for the imaginary part. We choose to test relatively small FFTs (64-points) to limit the number of iterations and ease the study of error propagation, while having 512×512 1D-FFTs eases the gathering of a statistically significant amount of errors.

A thread acts like a butterfly module [23] updating the values of two floating-point elements in the complex matrix using the values of two elements computed in the previous iteration as inputs (see Fig. 20.1). The implemented algorithm is based on the FT kernel of the NAS Parallel Benchmarks [24] implemented in C and ported to the GPU architecture using CUDA. As represented in Fig. 20.2, each 64-points 1D FFT kernel is composed of six sequential iterations ($\log_2 64 = 6$) of a variant of the Stockham FFT algorithm [25].

For all iterations, the GPU instantiates 512×512 parallel threads, grouped in blocks of 512 threads each. A thread is in charge of evaluating the intermediate FFT values on the assigned complex vector of size 64.

As a thread is in charge of updating two complex values, a radiation induced error that prevents the thread from completing its execution or corrupts the thread input data produces at least two output errors. Nevertheless, a single error in a thread can be generated by the corruption of the internal register that stores the value of just one of the two elements to update, or disturbing just one of the operations

needed to calculate the FFT. The thread can then complete its execution, allowing the correct calculation of the second complex number. Single output errors occur in the FFT only if such a single thread error occurs in the last iteration. This occurred in just 1.63 % of the faulty executions for the real and in 4 % of the faulty executions for the imaginary part [18].

The experimentally observed multiple error distributions are shown in Fig. 20.8. It is worth noting that in most of the cases 64 or less output values were found corrupted, and those locations belong to the same 64-point FFT. These errors patterns are caused by error propagation from one iteration to the following ones in the same 64-points FFT, as represented in Fig. 20.7. As said, the amount of errors is likely to double at each iteration, thus it is very unlikely to have an odd number of errors in the output, and this is in agreement with experimental data (see Fig. 20.8).

The worst case for a 64-points FFT occurs when radiation affects a thread in its first iteration. If a single error is produced in one thread in the first iteration, at each of the following five iterations (there are six iterations in total) the number of errors is doubled, and $25 = 32$ errors appear in the output. A double thread error is produced when radiation prevents the thread from completing its execution generating a functional interruption or corrupting the thread input. In this situation 64 output errors are to be expected in the FFT. It is improbable to have between 32 and 64 errors in the output vector. In fact, as it is very unlikely to have two neutrons corrupting the GPU in a single FFT execution, the only way of having more than 32 errors is to have a thread in the first iteration which generates two errors that spread to 64 errors in the output.

Finally, only few executions experienced more than 64 errors in the output. This rare situation occurs when radiation leads a SM to experience a functional interruption preventing a whole warp of 32 threads or even a whole block of 512 threads from

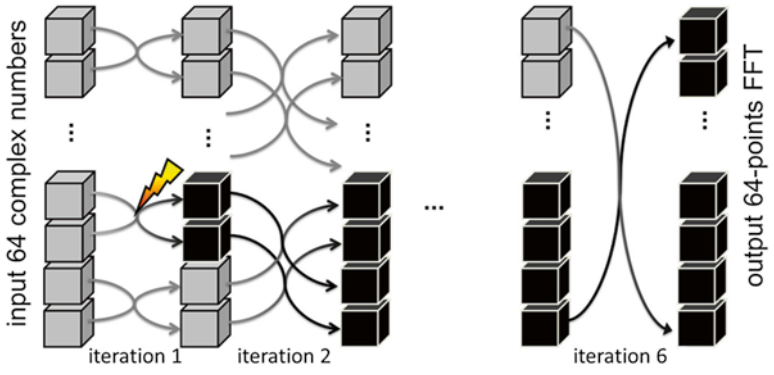


Fig. 20.7 In each iteration a thread updates two-by-two all the 64 values of the FFT using the basic butterfly module. Six iterations are necessary to complete the execution. If an operation in one iteration is corrupted by radiation, two (or more) values will be wrongly updated, and the number of errors doubles in the following iteration [18]

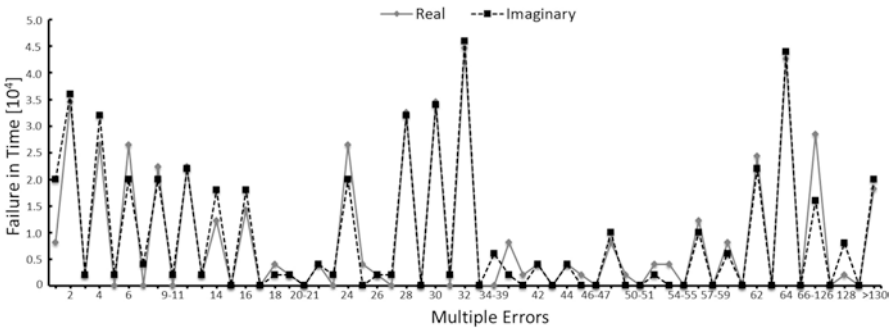


Fig. 20.8 FFT real and imaginary multiple output errors FIT. Consequent distributions that were never experimentally observed are grouped in the picture (it is the case of 9 to 11 errors, 20 and 21, etc.) [18]

completing their execution, possibly affecting more than one 64-points FFT outputs. Those errors will then spread and a huge amount of errors are expected at the output.

A parallel code to be executed on a GPU is typically composed of several independent threads, all executing the same set of instructions on dedicated memory location. Increasing the amount of threads brings then higher throughput to the application. To do so, the programmer can choose either to increase the block size, which will require more computational effort in each SM and delay the assignment of the next blocks, or to increase the grid size, thus having more blocks to be dispatched. The GPU parallel management is strictly related to the chosen thread distribution. The scheduling and computational load required for blocks and warps assignment, as well as resources distribution, are strictly related to the chosen grid and block sizes, which is then likely to influence also the GPU radiation response. So, the threads distributions as well as the number of instantiated thread significantly impact the radiation response of parallel devices. A detailed discussion on GPU parallel management reliability is presented in [19].

20.5 Conclusions

The spread of Graphics Processing Units in High Performance Computing and Safety-Critical applications arises new radiation test challenges. Unlike programmable logic devices or traditional sequential CPUs, GPUs requires complex scheduling and parallel processes management. Those resources corruption is critical, as various processes could be affected. Moreover, caches are shared among parallel tasks to reduce memory latencies. An error in the cache becomes, then, even more critical than in CPUs, as all the processes using the corrupted value are likely to produce a wrong result.

GPU vendors and designers are putting a lot of effort to reduce the radiation sensitivity of their devices, mostly focusing on the main resources physical implementation. Nevertheless, the inner GPU structure makes the device very prone to be corrupted. Moreover, as traditionally GPUs were employed in graphical or video editing applications, their architecture is voted to performances and not to fault tolerance. It becomes than hard, in the present moment, to introduce architectural solution to reduce the impact of radiation on GPUs. It is more likely that novel software-based hardening strategy will be designed to detect and, eventually, correct radiation induced failures without requiring hardware changes.

References

1. Owens JD, Houston M, Luebke D, Green S, Stone JE, Phillips JC (2008) GPU computing. Proc IEEE 96(5):879–899
2. Lindholm E, Nickolls J, Oberman S, Montrym J (2008) NVIDIA tesla: a unified graphics and computing architecture. IEEE MICRO 28(2):39–55
3. Kruger J, Westermann R (2003) Linear algebra operators for GPU implementation of numerical algorithms. ACM Trans Graph 22(3):908–916
4. Liepe J, Barnes C, Cule E, Erguler K, Kirk P, Toni T, Stumpf MPH (2012) ABC-SysBio—approximate Bayesian computation in Python with GPU support. Bioinformatics 26(14): 1797–1799
5. Euro NCAP rating review, Report from the Ratings Group, June 2012. Available: http://www.euroncap.com
6. Bender O (2014) ARAMIS—concepts to validate the safe application of multicore architectures in the avionics domain, HiPEAC 2014. Available [online] http://www.across-project.eu/workshop2013/121108_ARAMIS_Introduction_HiPEAC_WS_V3.pdf
7. Seifert N, Zhu X, Massengill LW (2002) Impact of scaling on soft-error rates in commercial microprocessors. IEEE Trans Nucl Sci 46(6):3100–3106
8. Nguyen HT, Yagil Y, Seifert N, Reitsma M (2005) Chip-level soft error estimation method. IEEE Trans Device Mater Reliab 5(3):365–381
9. Lerner MD (1988) Algorithm based fault tolerance in massively parallel systems. Department of Computer Science, Columbia University, Tech. Rep., 1988
10. Mitra S (2012) System-level single-event effects. IEEE nuclear and space radiation effects conference, NSREC 2012 short course
11. Bautista-Gomez L, Cappello F, Carro L, DeBardeleben N, Fang B, Gurumurthi S, Pattabiraman K, Rech P, Reorda MS (2014) GPGPUs: how to combine high computational power with high reliability. In: Proceedings of the IEEE design, automation and test in Europe (DATE), 2014, Dresden
12. Shi G, Enos J, Showerman M, Kindratenko V (2009) On testing GPU memory for hard and soft errors. In: Proceedings of the symposium on application accelerators in high-performance computing (SAAHPC), 2009
13. Wang NJ, Quek J, Rafacz TM, Patel SJ (2004) Characterizing the effects of transient faults on a high-performance processor pipeline. In: Proceedings of the IEEE international conference on dependable systems and networks (DSN), 2004, pp 61–70
14. Haque IS, Pande VS (2010) Hard data on soft errors: a large-scale assessment of real-world error rates in GPGPU. In: Proceedings of the IEEE/ACM international conference on cluster, cloud and grid computing, 2010, pp 691–696
15. Sheaffer JW, Luebke DP, Skadron K (2007) A hardware redundancy and recovery mechanism for reliable scientific computation on graphics processors. In: Proceedings of the ACM SIGGRAPH symposium on graphics hardware (GH), 2007, pp 55–64

16. Fang B, Pattabiraman K, Ripeanu M, Gurumurthi S (2014) GPU-Qin: a methodology for evaluating the error resilience of GPGPU applications. In: Proceedings of the IEEE international symposium on performance analysis of systems and software (ISPASS), 2014
17. Rech P, Aguiar C, Frost C, Carro L (2013) An efficient and experimentally tuned software-based hardening strategy for matrix multiplication on GPUs. IEEE Trans Nucl Sci 60(4): 2797–2804
18. Pilla LL, Rech P, Silvestri F, Frost C, Navaux POA, Sonza Reorda M, Carro L (2014) Software-based hardening strategies for neutron sensitive FFT algorithms on GPUs. IEEE Trans Nucl Sci 61(4):1874–1880
19. Rech P, Pilla L, Navaux POA, Carro L (2014) Impact of GPUs parallelism management on safety-critical and HPC applications reliability. In: Proceeding IEEE international conference on dependable systems and networks (DSN), June 2014, pp 455–466
20. Violante M, Sterpone L, Manuzzato A, Gerardin S, Rech P, Bagatin M, Paccagnella A, Andreani C, Gorini G, Pietropaolo A, Cargarilli G, Pontarelli S, Frost C (2007) A new hardware/software platform and a new 1/e neutron source for soft error studies: testing FPGAs at the ISIS facility. IEEE Trans Nucl Sci 54(4):1184–1189
21. Oliveira DAG, Rech P, Quinn HM, Fairbanks TD, Monroe L, Michalak SE, Anderson-Cook C, Navaux POA, Carro L (2014) Modern GPUs radiation sensitivity evaluation and mitigation through duplication with comparison. IEEE Trans Nucl Sci 61(6):3115–3123
22. Rech P, Carro L, Wang N, Tsai T, Hari SKS, Keckler SW (2014) Measuring the radiation reliability of SRAM structures in GPUs designed for HPC. In: Proceedings of the IEEE SELSE 2014
23. Jou J-Y, Abraham JA (1988) Fault-tolerant FFT networks. IEEE Trans Comput 37(5): 548–561
24. Bailey D et al (1994) The NAS parallel benchmarks. RNR technical report RNR-94-007, March 1994
25. Stockham TG (1966) High-speed convolution and correlation. Proceedings of the Spring Joint Computer Conference, 1966, pp 229–233