2014 9th International Design and Test Symposium

Reliability Assessment of Backward Error Recovery for SRAM-based FPGAs

Sahraoui Fouad*, Fakhreddine Ghaffari*, Mohamed El Amine Benkhelifa* and Bertrand Granado‡
*ETIS, CNRS UMR 8051, ENSEA, UCP; 6 avenue du Ponceau, 95000 Cergy-Pontoise, FRANCE
‡ LIP6, UPMC, CNRS UMR 7606; 4 Place Jussieu, 75252 PARIS Cedex 05, FRANCE

***Abstract*—Reliability is a major concern for embedded systems. Semiconductor devices used to implement them can suffer from various environmental perturbations. This is more evident when considering SRAM-based FPGA. Perturbations are very frequent and they can limit FPGA's usability. In this paper, a new fault tolerance approach is presented which try to take advantage of partial dynamic reconfiguration provided by SRAM-based FPGAs. The approach is based on the Backward Error Recovery to mitigate faults on the configuration layer by restoring the correct behavior of the application. Fault injection using emulation is used to evaluate the reliability of the proposed fault mitigation technique and its results are compared to those obtained when configuration scrubbing is used. An improvement of up to 12% for reliability and availability of the Design Under Test is observed.**

Keywords: Fault mitigation, Backward Error Recovery, SRAM-based FPGA, Reliability, Configuration Scrubbing.

I. Introduction

Modern Field Programmable Gate Arrays (FPGAs) have gained an important place over the past decade on various application domains, like: transportation, aeronautic, and even aerospace [1]. Such widespread utilization is mainly due to certain number of benefits that FPGAs offer, like : short time to market, affordable engineering costs, in-field (re)configurable and high resources availability, high execution speed and low energy consumption.

Most used FPGAs are structured into two layers : (a) *configuration layer* and (b) *resources layer*.
Configuration layer is based on SRAM technology and it is divided into a set of SRAM-based cells where each one is used to hold one bit, this latter configures a specific element from the resource layer. This set of bits is known as "bitsream". Resource layer is also divided into cells called *tiles*, each one contains a fixed number of Look-Up Table (LUT) and Flip-Flop (FF) that can be used to implement a small boolean function. A complex system is build on FPGA by dividing it to small boolean functions, map each one to a specific tile, and then interconnect each tile by a set of routing wires/matrices.

By writing a new bitsream to the configuration layer, a new functionality is performed by the resources layer elements, this is known as *(Re-)Configuration*. Such ability brings new paradigms to how applications are designed and executed on FPGAs. Indeed, instead of having all the application loaded at the same time into the FPGA where only some portions of this application is processing data, it is possible to split the whole application to different modules, and load/unload modules on-demand following execution conditions [2].

However, as a result of the huge number of SRAM memory used on FPGAs to achieve reconfiguration, FPGAs are very sensitive to faults caused by the interactions of high energy particles with their substrate [3]. Single-Event Effects (SEE) regroups those types of interactions, where the most recurrent faults on SRAM-based FPGAs are Single Event Upset (SEU) and Single Event Transient (SET), both are soft errors. A soft error is a temporarily disruption of the application execution and can be corrected by a simple reset/reconfiguration. Hard errors are faults that cause permanent damage to the FPGA and involve its replacement. Hard errors are out of the scope of this work.

Single Event Transient (SET) is a voltage glitch in one electronic element of the FPGA. If this glitch is used by a combinational logic then wrong values could be captured and computed. The SET can also appear and vanish without any impact. Single Event Upset (SEU) is an inversion of value saved by a sequential logic of FPGA. SEUs can occur either in the configuration memory or in the application memory like for example a flip-flop. The application-SEU is less frequent then the configuration-SEU because the first memory is an order of magnitude smaller than the configuration memory. In the rest of these work, SEU is used to refer to SEU occurring on the configuration memory of an FPGA.

A well-known approach to mitigate faults on FPGAs is the use of Triple Modular Redundancy (TMR) [4]. TMR mitigates faults by performing a majority voting upon the results of three identical instances. While there is only one faulty instance, TMR produces a high rate of reliability. However, it is very resources/power consuming and can even degrade the execution speed of the hardened application. Moreover, TMR can be inefficient on FPGAs due to SEU accumulation on the configuration layer or when multiple bit upsets are considered [5], [6].

To overcome such weakness with less overhead, new fault tolerant approaches based exclusively on partial dynamic reconfiguration (PDR) are proposed in the literature, those approaches try to take advantage of FPGA abilities. In this work, a reliability evaluation is performed for both *configuration scrubbing* and *Backward Error Recovery* using PDR through fault injection campaigns.

The rest of this paper is organized as follows: Section II

978-1-4799-8200-4/14/$31.00

gives an overview of configuration scrubbing method used to mitigate transient faults on FPGA's configuration. In Section III, we present our fault mitigation approach based on Backward Error Recovery (BER) and Partial Dynamic Reconfiguration. Section IV discusses the architecture used to perform fault injection emulation on those mitigation approaches and the adopted fault generation model. Section V gives reliability performances for both configuration scrubbing and BER approaches. Finally, section VI summarizes our contribution and outlines perspectives.

II. Configuration Scrubbing

A convenient way to eliminate SEU from the configuration memory of an SRAM-based FPGA is Configuration Scrubbing [7]. It is a technique based on the rewrite of initial bistream into the SRAM memory. The scrubber operates continuously sequencing through each frame and reloading its content following two possible strategies [8].

An efficient and simplest strategy to implement it is *"Blind Scrubbing"*. The scrubber uses a golden copy of the bitstream and periodically refresh the configuration layer with it. Usually, such strategy is performed with an external controller, which means that additional circuitry must be added to the initial FPGA-based system. Moreover, the controller must be fault tolerant to SEUs and a RadHard memory must be used to save the golden copy of the the initial bitstream.

Alternatively to blind scrubber, *"Readback Scrubber"* performs a detection of upsets on each frame before triggering any rewrite of the initial values.

Readback scrubbers are based on the ability to readback configuration memory from the SRAM [9] and to check if upsets are present or not. Detection of upsets can be performed either by comparing the retrieved frames with the initial ones or by using error detection and correction codes (EDAC). First version of readback scrubber [7] was based on cyclic redundancy check (CRC) to evaluate the integrity frames and it uses the external configuration interface SelectMap for "Read/Write" operations. Later, Extended Hamming code [10] (ECC) was used to carry detection/correction (SECDED) and the Internal Configuration Access Port is used to frame read/write. According to [7], readback srubbers can be divided based on their detection and repair approach as follow :

- Readback & Repair with Golden copy
- Readback with ECC & Repair with Golden copy
- Readback & Repair with ECC

The first one is exclusively based on the comparison of readback frames with the Golden copy of bitsream. For correction, the initial frame from the golden memory is rewritten into the configuration memory. The second and the third types use EDAC information available inside each frame to seek for any upsets occurring on the frame. Furthermore, in modern FPGAs, a hardwired circuitry is available to perform automatic detection after each readback of a frame [11].

For correction, the second type of scrubber uses a golden copy to reload correct values into the configuration memory, where the third one uses directly the location capability of used code to identify the bit to correct.

Improvement to scrubbing based on ECC can be achieved considering the fact that not all configuration bit upsets lead automatically to an application error [12]. It is possible to define a subset of configuration bits, where, if one of those bits is flipped then a functional change of the application will imperatively appear and the application would be incorrect (erroneous behavior or failure). By identifying those critical bits offline, it is possible to trigger correction only upon their change. Critical bits can be used to enhance the rate of scrubbing [12].

III. Backward Error Recovery Using Partial Dynamic Reconfiguration

Backward Error Recovery (BER) is the ability to bring back an application to a correct state upon the detection of an error. We already proposed in [13] a fault mitigation approach based on BER for SRAM-based FPGAs using Partial Dynamic Reconfiguration.

To protect a hardware IP core against transient faults, the IP is placed into an Enhanced Reliability Region (ERR) and a Reliability Controller (RC) is added to the final system. The RC is hosted into a static region [13]. An ERR is a normal partial dynamic region protected by the RC. This latter can be assigned to protect multiple ERRs by performing the following operations on each one separately using a frame masking mechanism [13], [14].

A. Fault Detection (FD)

Using Extended Hamming Codes, only one fault can be corrected and up to two faults can be detected per frame [11]. Each frame contains a parity bits calculated at the bitstream generation step. At run-time, those bits are used together with a hardwired circuit *"FRAME_ECC"* to check the correctness of the data stored in that frame at each read cycle. In our reliability controller, we only use detection capability of the available EDAC to activate the Recovery action. We avoid its correction capability due to the potential risk of fault introduction [15]. The fault detection is performed continuously in multitasking with the tasks execution until a fault is detected or a hardware checkpoint is required, hence no delay is introduced in the IP workload.

B. Hardware Checkpoint (HC)

Backward Error Recovery (BER) tends to return an IP to a free-fault state when a fault is detected. To achieve this, a periodic/sporadic creation of checkpoint is used.

Based on the nature of FPGA-based application, HC uses partial dynamic reconfiguration [11] for the checkpoint creation. The context of each IP is retrieved from the configuration layer of the FPGA by performing a readback capture of the corresponding ERR. A readback capture is a normal readback of bitstream [11] using PDR-feature preceded by a capture sequence (GCAPTURE) of FFs/LUTRAMs contents belonging to the target ERR. This distributed information is

localized on a set of frames and identified from the produced files after the bitstream generation (Xilinx file with extension .ll) [16].

Hardware checkpoint is also performed by the RC in parallel with the execution of tasks. Like configuration scrubbing, no preemption of IPs is needed to trigger the capture sequence of FFs or LUTRAMs nor the readback of frames.

C. Hardware Recovery (HR)

After a fault detection, RC must halt the execution of the corresponding ERR and performs a hardware recovery using partial dynamic reconfiguration. The HR is a set of two actions, first a rewrite of IP's context frames from the last checkpoint and the correct frame which was detected as faulty is performed, then a restore sequence (GRESTORE) [11] is activated to re-initialize the content of FFs and LUTRAMs to the previously saved context of the task.

Unlike previous operations, the execution of HR induces a time overhead for the recovered task, this overhead $T_{recovry_i}$ is expressed by:

$$T_{recovry_i} = t_{frame_{write}} \times (nf_{context_i} + 1), \qquad (1)$$

where $t_{frame_{write}}$ is the required time to write one frame to the configuration layer by the reconfiguration controller and $nf_{context_i}$ is the number of context frame plus 1 for the erroneous frame.

Figure 1 depicts two scenarios of a task execution with our reliability controller: (a) shows a normal execution where no fault is detected by FD and no overhead introduced and (b) shows the hardware recovery in action upon a fault detection. The execution of the IP is stopped during the HR execution.

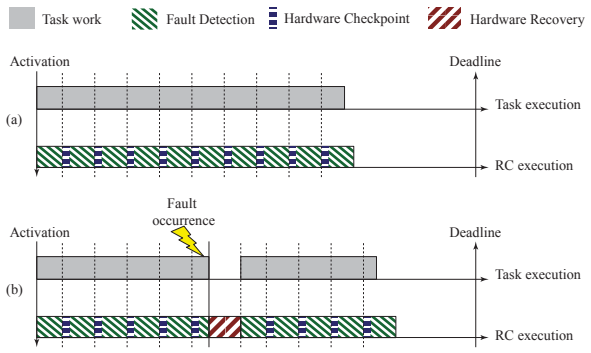

Fig. 1: Execution scenario of a task with reliability controller (a) without and (b) with fault occurrence.

IV. Hardware Platform for Fault Emulation

A. Test Application

Advanced encryption standard (AES) is used as a design under test (DUT). By nature, the AES algorithm is very sensitive to any changes, for example a small variation of its inputs can produce significant variations on the output or completely new one [17].

In our DUT, an implementation based on a KCPSM3 [18] (KCPSM3 is an embedded 8-bit RISC microcontroller for xilinx FPGAs) was used to perform an AES-128 computation. Also, a hardware version of substitution box (S-BOX) was used to accelerate the computation time of AES, from the processor point of view, the SBOX is considered as an I/O device. The PicoBlaze also uses an I/O controller to communicate with its RS232 controller and SBOX module as illustrated in Figure 2. The use of a processor based implementation of AES is motivated by having inside the DUT a design with both hardware and software computations.

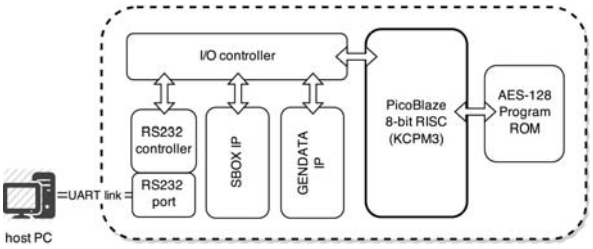

Fig. 2: Design under test architecture.

The *plaintext* and the encryption key are generated by a small IP core named *"DATAGEN"* located inside the DUT, see Figure 2. First, the processor loads two random arrays of 128-bits length each, and then it computes the encryption by executing the AES algorithm. Finally, the *plaintext*, the encryption *key* and the *ciphertext* are sent through the RS232 port to a host PC to be logged for later analyses.

We use the architecture presented in Figure 3 to implement the reliability controller of BER approach. And, to implement blind scrubbing with golden copy, we reuse this same architecture. We only change the application executed on the mircoblaze processor. This guarantees the same frequency of configuration memory read/write operation for both mitigation approaches.

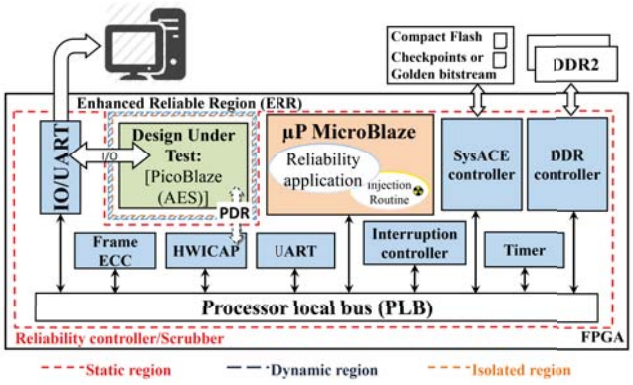

Fig. 3: Architecture of evaluation platform with Fault Emulation.

B. Injection Flow

The reliability of each mitigation method is evaluated using fault emulation approach [19]. For each mitigation approach, the DUT was placed on an isolated partial reconfigurable region [20]. This insure that injected faults via configuration layer will only target the DUT's logic/routing, the fault injector and the RC/scrubber remain correct throughout the fault injection campaigns.

The fault injector is implemented as a routine interruption launched by the Timer. Using the information of the characteristics of a fault to inject (type, size, shape), the routine halts the Reliability application executed on the microblaze, it selects a frame address following a probabilistic law, read the frame from the configuration layer invert the bit and rewrite the frame to the configuration layer. following the type of the fault and its shape, one or more frame/bit can be involved in one fault injection.

The probabilistic law used to generate faults characteristics and frames address is generated based on the ground experimentation reported in [21]. This choice is motivated by reproducing real behavior of the FPGA toward heavy ion perturbations. For the power LET, we use the information reported for the characterization using heavy ion with two Fluence (38.1 and 68.3 $MeV - cm^2/mg$). We also take into account the variation of incident angles (θ, ϕ), we reproduce two different values for each angle, $\theta = (0°, 90°)$ and $\phi = (0°, 60°)$ [21]. Table I gives an example of faults distribution law considering SEU/MBU.

TABLE I: Example of fault distribution law deduced from [21] for $(\theta = 0°, \phi = 0°)$

LETs & Angles		38.1 $MeV - cm^2/mg$		68.3 $MeV - cm^2/mg$	
SBU	1-bit	54%		41%	
MBU	2-bits	46%	39%	59%	34%
	3-bits		6%		13%
	4-bits		1%		12%

Percentage of Table I are used to compute the among of gault that should be injected into DUT memory and to chose their characteristics (size & shape).

V. Experimental results

Our experiments have been conducted on an ML506 board which incorporates a Virtex-5 FPGA. The fault injection was repeated, for each combination of LET fluence and incident angles, 50 times to produce significant data set to analyzed. A fault injection campaign finishes after the injection of 1000 faults, and the injection routine send a signal to the host PC to halt the FPGA, refresh the configuration layer with correct bitstream and launches a next injection campaign.
Each campaign results are stored on the host PC, later this information are parsed by a script on the host pc. The script recomputes the ciphertext using the saved plaintext and the encryption key and it compares it with the one produced by the DUT.

The process of fault injection is performed in parallel with the execution of configuration scrubbing or BER approach. The histogram presented in Figure 4 summarizes the percentage of correct output data produced by the DUT when protected by BER approach and when protected by scrubber. The fault injection frequency was set to 10 events by second. An event can produces a single fault or multiple.

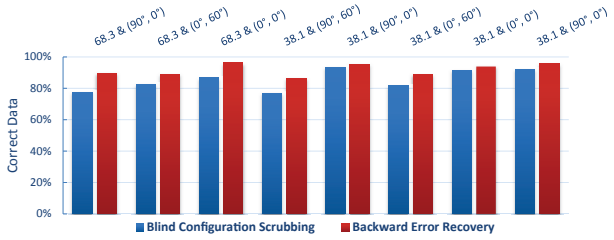

Fig. 4: Percentage of correct encrypted data produced by the DUT.

We can notice, from Figure 4, that the BER approach produces a slightly better protection against faults for each fault injection campaign. Improvement varies between 2% and 12% depending on the type of radiation (environmental characteristics).

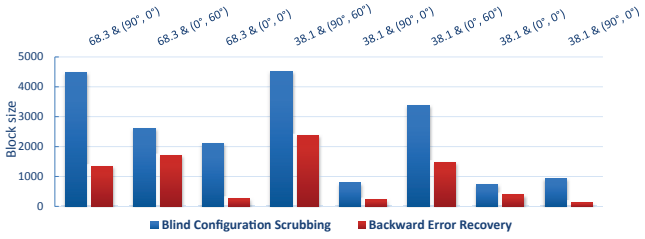

Fig. 5: Maximum size of erroneous block of data.

Moreover, when considering the biggest size of erroneous block of data, given in Figure 5, configuration scrubber produces longer erroneous behavior upon time execution. This means that the availability of the protected system was lowered when using configuration scrubbing.
Such behavior is explained by the fact that scrubbing tends to correct the configuration layer without taking into account the current state of the DUT (correct/erroneous state). Evermore, when using TMR with scrubbing, in [22], authors point out that scrubbing failed to correct faults. Experiments reported in [23] show that when inverting some configuration bits (occurrence of upsets) and then correcting them, the application (DUT) continued to show an erroneous behavior due to persistent errors.

Using BER approach shows that it is possible to avoid such a behavior by triggering recovery of the DUT's context to go back to a correct state. The same results are obtained when we vary the values of fault injection frequency, as shown in Figure 6. Both configuration scrubbing and BER approach perform better protection when fault arrival is relaxed.

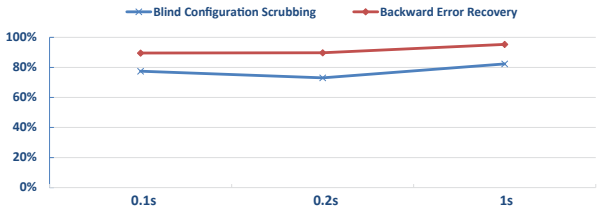

Fig. 6: Fault injection frequency variation for heavy ion emulation with 68.3 & $(\theta = 0°, \phi = 0°)$.

VI. Conclusion and Future work

In this work, we conducted fault emulation against Configuration Scrubbing approach and Backward Error Recovery to evaluate their reliability performance. A software/hardware implementation of AES algorithm was used as a design under test. Faults information was generated following a probabilistic distribution law on Virtex-5. Obtained results show an improvement on both reliability and availability of the tested system.

In our future work, we will conduct a deep evaluation of BER overhead upon DUT need to be conducted and minimization must be performed to enhance the overall reliability of our approach.

References

[1] B. Osterloh, H. Michalik, S. A. Habinc, and B. Fiethe. Dynamic partial reconfiguration in space applications. In *Adaptive Hardware and Systems, 2009. AHS 2009. NASA/ESA Conference on*, pages 336–343.

[2] E. J. McDonald. Runtime fpga partial reconfiguration. In *Aerospace Conference, 2008 IEEE*, pages 1–7.

[3] R. D. Schrimpf. Radiation effects in microelectronics. In *Radiation Effects on Embedded Systems*, pages 11–29. Springer Netherlands, 2007.

[4] F.L. Kastensmidt, L. Carro, and R.A. da Luz Reis. *Fault-tolerance Techniques for SRAM-based FPGAs*, volume 32. Springer Verlag, 2006.

[5] H. Quinn, K. Morgan, P. Graham, J. Krone, M. Caffrey, and K. Lundgreen. Domain crossing errors: Limitations on single device triple-modular redundancy circuits in xilinx fpgas. *Nuclear Science, IEEE Transactions on*, 54(6):2037–2043, 2007.

[6] G. Foucard, P. Peronnard, and R. Velazco. Reliability limits of tmr implemented in a sram-based fpga: Heavy ion measures vs. fault injection predictions. In *Test Workshop (LATW), 2010 11th Latin American*, pages 1–5.

[7] Xilinx Application Notes XAPP216. Correcting single-event upset through virtex partial reconfiguration, 2000.

[8] Jonathan Heiner, Benjamin Sellers, Michael Wirthlin, and Jeff Kalb. Fpga partial reconfiguration via configuration scrubbing. In *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, pages 99–104. IEEE.

[9] Ken Chapman. Seu strategies for virtex-5 devices, 2010.

[10] B. Dutton and C. Stroud. Built-in self-test of embedded seu detection cores in virtex-4 and virtex-5 fpgas. In *International Conference on Embedded Systems and Applications*, pages 149–155.

[11] Xilinx. Virtex-5 fpga configuration user guide ug191 (v3.11).

[12] Ken Chapman. Virtex-5 seu critical bit information. *XAPP864*, 2010.

[13] F. Sahraoui, F. Ghaffari, M. El Amine Benkhelifa, and B. Granado. An efficient ber-based reliability method for sram-based fpga. In *Design and Test Symposium (IDT), 2013 8th International*, pages 1–6.

[14] F. Sahraoui, F. Ghaffari, M. El Amine Benkhelifa, and B. Granado. Context-aware resources placement for sram-based fpga to minimize checkpoint/recovery overhead. In *Reconfigurable Computing and FPGAs (ReConFig), 2014 International Conference on*.

[15] M. Berg, C. Poivey, D. Petrick, D. Espinosa, A. Lesea, K. A. LaBel, M. Friendlich, H. Kim, and A. Phan. Effectiveness of internal versus external seu scrubbing mitigation strategies in a xilinx fpga: Design, test, and analysis. *Nuclear Science, IEEE Transactions on*, 55(4):2259–2266, 2008.

[16] Xilinx. Command line tools user guide (ug628 v14.2), 2012.

[17] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES-the advanced encryption standard*. Springer, 2002.

[18] Xilinx. Picoblaze 8-bit microcontroller.

[19] G. L. Nazar and L. Carro. Fast single-fpga fault injection platform. In *Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), 2012 IEEE International Symposium on*, pages 152–157.

[20] John D. Corbett. The xilinx isolation design flow for fault-tolerant systems, 2012.

[21] H. Quinn, K. Morgan, P. Graham, J. Krone, and M. Caffrey. Static proton and heavy ion testing of the xilinx virtex-5 device. In *Radiation Effects Data Workshop, 2007 IEEE*, volume 0, pages 177–184.

[22] B. Pratt, M. Caffrey, P. Graham, K. Morgan, and M. Wirthlin. Improving fpga design robustness with partial tmr. In *Reliability Physics Symposium Proceedings, 2006. 44th Annual., IEEE International*, pages 226–232.

[23] K. Morgan, M. Caffrey, P. Graham, E. Johnson, B. Pratt, and M. Wirthlin. Seu-induced persistent error propagation in fpgas. *Nuclear Science, IEEE Transactions on*, 52(6):2438–2445, 2005.