www.czasopisma.pan.pl PAN www.journals.pan.pl
INTL JOURNAL OF ELECTRONICS AND TELECOMMUNICATIONS, 2014, VOL. 60, NO. 1, PP. 103–108
Manuscript received January 10, 2014; revised March, 2014. DOI: 10.2478/eletel-2014-0012

Tracing Fault Effects in FPGA Systems

Mariusz Węgrzyn and Janusz Sosnowski

***Abstract*—The paper presents the extent of fault effects in FPGA based systems and concentrates on transient faults (induced by single event upsets – SEUs) within the configuration memory of FPGA. An original method of detailed analysis of fault effect propagation is presented. It is targeted at microprocessor based FPGA systems using the developed fault injection technique. The fault injection is performed at HDL description level of the microprocessor using special simulators and developed supplementary programs. The proposed methodology is illustrated for soft PicoBlaze microprocessor running 3 programs. The presented results reveal some problems with fault handling at the software level.**

***Keywords*—FPGA testing, application based testing, fault injection, SEUs**

I. Introduction

FPGA based systems become very popular in many technical domains, including dependable applications where fault effects may have critical consequences. Hence, various fault tolerance schemes are proposed in the literature, e.g. partial reconfiguration, massive redundancy, error scrubbing [1]–[3]. As opposed to classical systems based on ASICs and microprocessors (with fixed logical structure) in FPGAs we face some additional problems related to configuration faults which have severe impact on the system operation. Developing fault detection and fault tolerance techniques we have to analyze propagation of fault effects taking into account the specificity of FPGA structures. A special interest is targeted at transient faults, due to the fact that permanent faults result in similar effects as in classical fixed logic systems. Transient faults relate to SEUs (single event upsets) caused by cosmic radiation, electromagnetic disturbances, power problems, etc. In the literature we observe a strong interest to analyze transient fault impact on FPGAs exposed to radiation (e.g. [4]–[7]). These experiments confirm critical effects at the level of implemented application within FPGAs. Unfortunately, the controllability and observability of these experiments is low. They give only some general view of the problem.

The drawbacks of radiation experiments can be alleviated in simulation based fault injections. For this purpose we have developed fault injection scenarios at HDL level. We have concentrated on SEUs within FPGA configuration memory, due to the fact that related faults may have significant impact on the system operation, however the analysis of their effects is still neglected. Moreover, we concentrate on investigating these effects in correlation with implemented applications. An important and original contribution of this paper is tracing fault effects related to three levels: logical, microprocessor and application. We take into account microprocessor model PicoBlaze which is used to run various programs. This is quite typical model of using FPGAs in many systems (e.g. [3], [8], [9]), however in practice we can use more sophisticated microprocessors e.g. Power PC [8] (for many of them configuration files for FPGAs are available).

Various fault handling techniques have been developed for classical fixed logic microprocessor systems ([10], [11] and references therein). We have found that using them directly in FPGA based microprocessor systems is not so efficient as in classical fixed logic and some supplementary techniques have to be added. We deal with this problem in the paper. Another important issue in fault handling is testing the hardware platform. FPGAs create here more problems than fixed logic systems. This results from functional block universality and a wide scope of configurations. So, complete testing of the available logical and interconnection resources is cumbersome, it involves several chip reconfigurations (high time overhead) including programming various BIST schemes (compare [12]–[14]). In practice, we are restricted to a single application implemented on FPGA. Hence, application based testing can be a reasonable solution [9], [15]–[17]. Checking the effectiveness of such tests we can use fault injection techniques.

The outline of the paper is as follows. Section 2 gives some general view on fault models and fault injection techniques in FPGAs. Section 3 describes the developed fault injection environment and test scenarios. Experimental results are presented in section 4. Final conclusions are given in section 5.

II. Faults in FPGAs

SRAM based FPGAs are composed of configurable logic blocks (CLBs), routing circuitry which connects these blocks and configuration memory which defines performed functions of CLBs and their interconnections (performed by switch boxes and wiring segments). Typically, CLBs comprise some look up tables, flip-flops and internal routing (e.g. implemented with multiplexers controlled by the configuration memory). Look up tables (LUTs) define logical functions (truth table) of n inputs and m outputs. Sometimes special data RAM memories and specialized logical blocks (e.g. arithmetic logic) are included also. The functionality of FPGA is determined by the configuration memory, its content is programmed externally by loading appropriate stream of bits. This stream usually is organized in frames (e.g. 32 bits in Virtex 4). Frames can be attributed to CLBs, and other programmable components in FPGA (e.g. switches).

In general, we can distinguish permanent and transient faults. Permanent faults result from physical damages such as stuck at faults caused by shorts, opens of connections to inputs or outputs of logic circuits, etc. Transient faults relate to temporary disturbances which change the state of logical components (e.g. flip-flop state change, gate input or output pulse glitch, RAM cell state change). These disturbances

M. Węgrzyn and J. Sosnowski are with the Institute of Computer Science, Warsaw University of Technology, ul. Nowowiejska 15/19, Warsaw 00-665, Poland (e-mails: marioweg@o2.pl; jss@ii.pw.edu.pl).

can result from electromagnetic interference, power supply noise, cosmic, artificial and natural radiation (e.g. due to alpha particles which are emitted by radiating impurities in the chip packaging materials). There are also intermittent faults which have the nature of reoccurring transient faults of short or long duration. In fact, they relate to physical damages of more subtle nature, e.g. degradation of some electrical parameters (revealing as logical faults under specific operational circumstances), crosstalk, etc. Dealing with permanent and intermittent faults in FPGAs practically is very similar to classical fixed logic circuitry [10], [11]. Similarly we can treat transient faults related to fixed logic components of FPGAs (e.g. CLBs, data RAMs). The most critical are transient faults within the configuration memory. These faults may change the function of FPGA blocks and connections. As opposed to fixed logic systems many transient fault mitigation techniques, e.g. based on time or software redundancy are not effective in the case of configuration faults. FPGAs are becoming more prone to transient faults due to the increasing integration density [5], [6]. In most cases they are modeled as single event upsets. They are more probable in space and avionics systems exposed to higher level of cosmic radiation, nevertheless neutron particles present in the atmosphere can also produce SEUs.

In practice, an important issue is transient fault risk assessment in relevance to configuration memory. For this purpose various experiments have been performed with artificial and natural radiation. These experiments have low controllability and observability, which is important to get a deeper view on SEUs consequences. Moreover, in practice we deal with FPGA based systems performing a preprogrammed function (application), so it is reasonable to analyze fault effects concerning this application. Hence, we have developed a test bench for experiments with fault injection at hardware description level. In general, FPGA susceptibility to SEUs can be evaluated in four ways:

1) *Exposing a large number of FPGA circuits to cosmic rays at specified conditions on the earth for a long time.* This approach has been used in Rosetta experiment [4] which assured total testing time in the range from hundred thousand to millions device hours. FPGA configuration bit streams were continuously checked by read out operations (compared with the reference stored patterns). In this way average failure rates have been derived for 4200 m above sea level and 550 meters below ground.
2) *Exposing FPGAs to an accelerated neutron flux.* Special expensive equipment is needed here [6], [7], it assures also specification of particle energies and intensity. Knowing the space radiation energies and intensity on the earth or satellite orbits (for different places and altitudes) we can recalculate the operation time in the experiment (with the accelerator) to the equivalent time in the considered FPGA operation target environment, e.g. [3].
3) *Fault emulation in FPGAs.* Faults can be emulated by disturbing configuration bit streams and loading them through the configuration interface (e.g. JTAG) to the tested chip and monitoring its behavior. Here, some supporting equipment is needed (compare [3], [8], [18]). This can be extended for flip-flops and block RAMs in more sophisticated fault injectors. In [19] a more complex fault emulation scheme is integrated with the modified VHDL source code (additional gates, multiplexers, wires, etc.) of the analyzed system, this approach is not useful for the planned experiments with SEUs in LUTs.
4) *Fault injections into FPGA simulators* – this approach has been used in our experiments (section 3).

The first approach gives some general characterization of FPGA susceptibility to SEUs, which can be a basis of some scaling adapted to the considered implemented systems (e.g. taking into account resource usage). Typically, SEU error rate is specified in FIT units (the number of failures that can occur in 10^9 hours) referred to the memory size in bits. In [3], [4], [20] we have quite interesting results related to radiation at the level of the earth. They cover not only susceptibility of configuration memory to SEUs but also data RAMs incorporated in logical units (block RAM memory). Depending upon the technology (150nm to 65 nm in Virtex5) failure rate for configuration memory was in the range 401-151 FIT/Mbs (with 95% confidence intervals: [367,435]-[101,215]), it decreased with higher integration densities (however, higher dispersion was observed over the tested chips here) due to additional new fault hardening techniques used in these memories. In the case of data RAM block failure rate was in the range 397-635 FIT/Mb (with 95% confidence intervals: [317,491]-[428-907]) and it increased with the integration density. The Rosetta experiment [4] confirmed that most probable are single bit changes, multi bit upsets were negligible. Configuration memory is less susceptible to SEUs than data memory. Moreover, the susceptibility of flip-flops in CLBs was very low (for Virtex5 approximately 0.06 FIT/Mb). The presented results explain why in newly developed FPGAs we can encounter ECC codes combined with configuration frames.

We should be conscious that the probability of SEUs increases with the altitudes above the sea level, hence it can be several times higher in the mountains, for airplane flights the increase can be several hundred times higher, in the space environment (satellites and other cosmic equipment) this can be even higher [3]. Moreover, in these cases multi bit upsets are also more probable.

Experiments of the second type allow us to characterize fault susceptibility of real applications. In practice, implementing some application (circuit) over FPGA we do not use all chip resources, moreover in this case many configuration bits do not effect this implementation. Hence, the derived fault susceptibility in experiments of type 2 have to be scaled down taking into account radiation characteristics of the operational environment. However, experiments of type 2 can be performed in short time using a single or a few FPGA chips. This is critical taking into account a wide scope of possible application's and implementations.

Experiments of type 3 are more flexible and cheaper, they can be performed in short time with single FPGA, moreover the required test environment can be relatively cheap. However, it is targeted at checking fault susceptibility in relevance to artificially injected SEUs, so the equivalent fault rate in FITs needs to be recalculated taking into account overall

SEUs susceptibility of FPGAs (obtained from the first group of experiments). For this purpose we can use the following formula to scale the results:

$$\mathrm{FS_A} = \mathrm{FS_{FPGA}} * \mathrm{CU_A} * \mathrm{NFS_A}$$

where $\mathrm{FS_A}$ and $\mathrm{FS_{FPGA}}$ are fault rates in FITs related to the considered preprogrammed application and overall FPGA structure (we consider here only configuration memory bits), $\mathrm{CU_A}$ is configuration memory usage ratio for the implemented application, $\mathrm{NFS_A}$ is natural fault masking of the application (e.g. algorithm fault tolerance, partial redundancy). $\mathrm{CU_A}$ relates to the limited usage of logical blocks and potentially critical configuration bits for the application (compare [3]). For Virtex 5 (xc5v1x50t) the nominal fault rate is 151 FIT/Mb and the configuration memory is 11.37 Mb, hence for this chip $\mathrm{FS_{FPGA}} = 1717$ FIT, which is equivalent to about 66 years of Mean Time Between Failures (MTBF). FISA relates to natural fault tolerance of the application. In [3] an application with dual microprocessors (PicoBlaze) has been considered which used 82% of chip logical resources and the estimated critical configuration bits resulted in $\mathrm{CU_A} = 16\%$. Fault injection experiment (over 5000 faults injected randomly in configuration bits) resulted in 4.9% failures, hence $\mathrm{FS_A} = 84$ FIT ($\mathrm{NFS_A}$ was about 30%). Using an error recovery technique (based on an additional PicoBlaze processor) $\mathrm{FS_A}$ has been reduced to 20 FIT [3].

In experiment 3 it is difficult to trace fault effects, moreover correlation of configuration bits with used functional blocks is quite complex, so significant number of fault injections is needed even if the application uses only a small part of the chip. Deeper fault propagation analysis is useful to optimize fault mitigation mechanisms. This can be achieved in experiment 4. Here we can disturb only used resources. However, a good simulator is needed (e.g. Cadance).

In the literature various error detection, error recovery and fault tolerance techniques have been proposed for FPGA based systems. The most complex ones use massive redundancy e.g. triple modular redundancy, which is very expensive and acceptable in the case of critical applications ([1], [3], [19] and references). Targeting at configuration SEUs we can detect faults by periodical read backs and if needed recoding configuration bit stream [21]. This results in high time overhead unacceptable in many applications. Some FPGAs provide the capability of partial reconfiguration, which can be used to optimize this technique. Checking the correctness of the configuration frames can be done by some external circuitry or internally by built in checker in FPGA (e.g. based on some simple microprocessor with self-testing capability). The latter solution is possible in FPGAs which provide internal configuration port and embedded ECC codes in the configuration frames stored on chip. This is available in Virtex 4 and 5 devices, where ECC code detects double errors, locates single bit errors. This capabilities can be used to mitigate SEUs (compare [3], where FSA has been reduced from 84 FIT to 20 FIT).

In classical fixed logic microprocessor based systems various software techniques can be used to mitigate SEUs (e.g. recalculation, checksums, exception handling, software redundancy [10]). Recently, many applications implemented in FPGAs are based on embedded (programmed in FPGA) microprocessors. Hence, arises the problem of checking their fault robustness in FPGA, as well as checking the effectiveness of software tests for microprocessors [9]. These techniques can be targeted at the considered applications to make them more effective and cheaper. The robustness of these solutions can be checked in experiments of type 4. In particular, they allow us to achieve high degree of stressing the tested application as compared with other experiments.

III. Testing Schemes

The developed experiments are targeted at testing fault susceptibility of application programs running on a microprocessor implemented within FPGA. The general idea is to use appropriate microprocessor simulator which accepts its specification in HDL language, correlates it with the targeted FPGA, performs simulations of executing provided programs (in assembler) and allows analyzing the behavior of the tested application (e.g. program results) in this environment. These assumptions are fulfilled by two simulators: Cadence NC VHDL and Mentor Graphics ModelSim. Fault injection is performed at microprocessor HDL description level, which reflects FPGA implementation.

There are available HDL descriptions of soft processors for FPGA implementation, e.g. PicoBlaze for Xilinx FPGAs [22]. Their HDL descriptions reflect the FPGA structure in order to efficiently use the FPGA resources that allow precise modeling of the faults and their automated fault injection. Each simulated fault is represented by an appropriate HDL file, however some additional scripts are needed to perform fault injection campaigns and analyze their effects in an automatic way. The faults in an HDL description of the processor are simulated by modifying the individual functional blocks. For each functional block a HDL model describing behavior of SEU-induced faults is developed. The HDL model should actually reflect the change of configuration as a consequence of the SEU effect.

In the performed experiments we use PicoBlaze processor specified in VHDL [22]. The basic VHDL entities (Xilinx basic primitive subcircuits) in Xilinx PicoBlaze processor core description are: RAMs, LUTs, multiplexers and flip-flops. RAM and flip-flop state changes are of a transient nature and can be modified (i.e. restored to a fault-free value) during normal system operation. These faults are detected with an online functional test, specific to the target application and hence they are not the subject of this investigation. Our goal is to analyze fault effects in the configuration of the processor core (in particular, faults in LUTs).

The generation of the fault descriptions was implemented as a perl script. All the instances of LUTs related to functional blocks are located (specified) in the VHDL description of the processor core. For each LUT instance its initialization parameter is investigated and the list of the initialization parameters describing all the SEU-induced faults as well as all the stuck-at faults at the LUT inputs and outputs are generated. For some LUT instances it is possible that a single bit change of a LUT content manifests itself as a stuck-at fault. In such a case a duplicated stuck-at fault description is excluded. In a similar way the stuck-at faults at the LUT inputs as well as

a)

```
move_group_LUT: LUT4
generic map (INIT => X"7400")
port map( I0 => instruction(14),
          I1 => instruction(15),
          I2 => instruction(16),
          I3 => instruction(17),
          O => move_group );
```

b)

Inputs			INIT => X"7400"		INIT => X"7480"	
I2	I1	I0	O(I3=0)	O(I3=1)	O(I3=0)	O(I3=1)
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	0	1	0	1
0	1	1	0	0	0	0
1	0	0	0	1	0	1
1	0	1	0	1	0	1
1	1	0	0	1	0	1
1	1	1	0	0	**1**	0

Fig. 1. Fault effect related to the change of one bit (X"7400" → X"7480") in LUT4: a) VHDL description of fault-free four-input circuit, b) truth tables for fault free and faulty LUT.

the stuck-at faults at the LUT output can also be modeled by modifying the contents of the LUT configuration.

An example of a modeled fault is shown in Fig. 1. The VHDL description of a LUT implementing a circuit generating an internal processor signal *move-group* is shown in Fig. 1a and the corresponding truth table in Fig. 1b. The input signals I0-I3 relate to the specified bits (in brackets) of the instruction code (in the instruction register). The implemented logic function is defined by the initialization parameter (INIT) assumed as X"7400", i.e. hexadecimal code related to a vector comprising bits of concatenated columns O(I3=1) and O(I3=0). The most significant bits of O(I3=1) and O(I3=0) bytes relate to the last row of the table. The SEU-induced fault of a LUT typically manifests itself as a change of one bit of the LUT, thus modifying the Boolean function it implements. Let us assume that the 8^{th} bit of the LUT column O(I3=0) has been changed (truth table in Fig. 1b with marked false value as bold underlined 1). This fault can be modeled in VHDL description changing the initialization parameter (INIT) from X"7400" to X"7480". Similarly, we can model stuck-at faults on inputs or outputs. For example a stuck-at-1 fault at input I3 in the considered LUT is modeled by INIT = X"7474", i.e. column O(I3=0) assumes the value of column O(I3=1). Stuck-at-1 fault at output O is modelled by INIT = X"FFFF" (all LUT memory entries equal to 1).

The considered LUT (LUT4 in VHDL description [22]) in Fig. 1a relates to the decoder for the control of the program counter and CALL/RETURN stack. Having analyzed the effect of the simulated fault (INIT = X"7480") we have found that it resulted in erroneous decoding of SUB or SUBCY instructions as RETURN or JUMP with unknown address locations, so the program did not terminate correctly.

During the fault simulation the generated "faulty" initialization parameters were applied one by one to the VHDL description of the Xilinx PicoBlaze processor core [22]. A modified VHDL description is then used by the simulator to run a tested application (program). Taking into account a large number of considered faults we use special scripts which automatize the processes of loading new configuration, running the application and storing results. Having injected the specified number of fault injections we check the results with an additional script which qualifies fault effects and generates summarized statistics. It is also possible to trace effects of individual faults even at the signal levels within selected internal logic circuits, e.g. an output of some flip-flop. For PicoBlaze processor we have identified 1804 single bit faults related to used LUTs. The Xilinx PicoBlaze processor is a small 8-bit microprocessor, used mainly for training purposes. It has 1K of program space, 16 8-bit registers, 256 input and 256 output ports, a 64-byte internal scratchpad RAM and a 31-location stack. The original VHDL description of the processor core consists of about 1500 lines of code.

We needed to modify the original VHDL description to enable the perl script fault injection. The PicoBlaze hardware is generated using VHDL loops "for" which create more instances of the same sub-circuits (most loops replicate 8 times bit slices of some logical blocks). To make accessible all these instances to the fault injector, we have "unrolled" all "for" loops (explicit code blocks embedded in VHDL description). The unrolled VHDL description results in about 3000 lines of code. In this way the perl script has direct access to every line of the hardware description. The fault injection is implemented by reading line by line this unrolled VHDL description by the perl script. The script looks for proper strings (description of INIT parameters) in the code and then changes values of these parameters. After every change was completed, simulation is started by the same script.

In the VHDL description we can distinguish 14 modules, we give functions of these modules, related VHDL description lines (beyond these lines there are some initialization, comment and control lines) and the number of functional FPGA elements used in these modules (CLB – logical blocks, LUT – configuration tables, FF – flip-flops, MUX – multiplexers and XOR circuits). The presented parameters give some view on the PicoBlaze microprocessor complexity:

- Basic control unit (lines 305-326); CLB = 1, LUT = 1, FF = 3
- Interrupt logic (lines 343 to 392); CLB = 2, LUT = 3, FF = 6
- Decoder for the control of the program counter and CALL/RETURN stack circuitry (lines 414 to 463); CLB = 3
- The ZERO and CARRY flags circuitry (lines 479 to 625); CLB = 6, LUT = 11, FF = 3, MUX = 8, XOR = 3
- The program counter (lines 693 to 944); CLB = 10, LUT = 20, FF = 10, MUX = 20, XOR = 29
- Register bank and the second operand selection (lines 1208 to 1467); CLB = 5, LUT = 10, FF = 1
- Memory storing function (lines 1493 to 1508); CLB = 5, LUT = 2, FF = 10
- Logical operations combined with the pipeline stage to form ALU multiplexer and decoding circuitry (lines 1713 to 1856); CLB = 5, LUT = 9, FF = 8

- Shift and Rotate operations combined with the pipeline stage (lines 1888 to 2079); CLB = 6, LUT = 11, FF = 9, MUX = 1
- Arithmetic operations combined with the pipeline stage (lines 2107 to 2339); CLB = 5, LUT = 9, FF = 8, MUX = 8, XOR = 8
- Generation of the most significant bit in ALU (lines 2376 to 2396); this function is implemented within the presented above 3 functional modules related to ALU
- ALU multiplexer (Lines 2418 to 2641); CLB = 9, LUT = 17, FF = 1, MUX = 1, XOR = 8
- Read and Write strobes (lines 2674 to 2691); CLB = 2, LUT = 3, FF = 2
- CALL/RETURN stack control (lines 2964 to 3075); CLB = 6, LUT = 5, FF = 11, MUX = 4, XOR = 5

We have found the need of checking FPGA fault susceptibility at the application level, which runs on the preprogrammed processor soft core PicoBlaze. Here, we have to take into account the fact that the processor resources can be used partially or in a limited way. Moreover, in this case we may encounter natural fault masking capability of the application as well as we can introduce some additional fault tolerance mechanisms at the software level (compare [10]). For this purpose we have developed 3 matrix multiplication programs (MM1-MM3). The basic program MM1 comprises 133 instructions, only 16 different instructions from the PicoBlaze instruction set are used. The static distribution of the used instructions was as follows (instruction occurrences in the program are shown in brackets):

ADD(40), CALL(1), COMPARE(8), FETCH(3), JUMP[C,NC,Z,NZ](20), RETURN(1), SRA(2), SR0(1), STORE[kk,(sY)](53), SUB(4), TEST(1).

Program MM2 is an enhanced version of MM1 by adding control sums in columns and rows of the first and second argument matrix, respectively. This assures control sums (columns and rows) in the resulting matrix. MM2 program comprises 226 assembler instructions, using 17 instruction codes from the processor list with the following distribution:

ADD(53), ADDCY(2), CALL(1), COMPARE(21), FETCH(16), JUMP[C,NC,Z,NZ](46), RETURN(2), SRA(2), SR0(1), STORE[kk,(sY)](78), SUB(5), TEST(1).

Program MM3 is an enhanced version of MM2 by adding exception handling. It comprises 386 instructions, using 21 instruction codes from the processor list with the following distribution:

ADD(59), ADDCY(4), AND, CALL(1), COMPARE(23), FETCH(17), JUMP[C,NC,Z,NZ](53), RETURN(2), SRA(2), SR0(1), STORE[kk,(sY)](85), SUB(5), TEST(2).

The mnemonics of instructions are self-explanatory (compare [22]), please note that JUMP and STORE relate to 4 (different condition tags) and 2 (memory address immediate or in a register) types of instructions, respectively.

In the performed experiments configuration faults have been injected at VHDL description level and for each fault the analyzed program has been executed in the simulator. Fault impact has been checked at the application level. For this purpose we have developed a special script which compared the resulting matrix with the reference correct one and checked program termination. Fault effects are summarized in Tab. I.

TABLE I
FAULT INJECTION EFFECTS FOR MATRIX MULTIPLICATION.

Program version	Fault effects			Number of new faults
	No result	Incorrect result	Correct result	
MM1	28.62%	21.23%	50.16%	12
MM2	28.62%	21.22%	48.84%	13
MM3	33.11%	19.94%	53.05%	13

We have distinguished 3 classes of fault effects: no result – the program does not terminate (infinite loop) or does not produce the resulting matrix, incorrect result – the generated result is erroneous (typically many result matrix entries are incorrect), correct result – many injected faults disturb operation of microprocessor logic blocks which are not used during the program execution (e.g. related to not used instructions). The input data for the tested applications assures 100% coverage of executed instructions in the program.

In [9] the first author checked fault effect propagation for the developed self-test program of PicoBlaze microprocessor. This test used up to 256 test vectors and covered over 90% of injected faults. Each test iteration (related to the specified initial test vector) involved execution of about 300 processor instructions. There were 41 hard to detect faults of the first order, i.e. detected by only one test vector. It was interesting that the matrix multiplication program allowed us to reveal (detect) some additional faults (shown in the last column). Tracing the propagation of these faults in these programs we could identify the reasons of their masking in the self-test program. Hence, it was possible to improve some instruction sequences to cover these faults. We have also noticed that the considered programs detected 16, 16 and 21 hard to detect faults, respectively. An interesting issue was that these 3 programs detected almost the same new faults.

It was interesting to compare this with fault effects in the developed program of matrix multiplication for the fixed logic microprocessor (compatible with Intel x86) [10]. Here, we have got more fault detections and corrections due to the control sums. In particular incorrect results appeared only in 0-1% cases, depending upon fault injections (single bit-flips in data, registers or program code). Correct results contributed 50-80% in the case of program handling exceptions. The program handling only checksums assured 30-36% correct results for faults injected into registers and data area (however 2.1% correct results appeared for faults injected into program code). This relative high fault robustness (small percentage of incorrect undetected results) did not appear in FPGA based microprocessor, due to the fact that configuration faults introduce more permanent disturbances as opposed to more transient effects in fixed logic microprocessors. Moreover, system exceptions in Intel x86 platform are more efficient than in PicoBlaze. In the FPGA experiment incorrect result matrices comprised many erroneous entries, quite often these entries were repeated. Hence, corrections practically were not possible.

Performing a similar experiment with a simpler application (calculation of Fibonacci series – about 30 instructions) we have found over 65% of correct results and no newly detected fault as compared with the basic testing program.

The performed experiments confirm that software techniques used to detect or correct errors in fixed microprocessors are not sufficient to deal with configuration faults in FPGA based systems. They can mitigate fault effects beyond the configuration memory, e.g. SEUs effects within data flow, flip-flops. Hence, we have to use more expensive redundant systems, doubled microprocessor with comparison [3], [8] or check periodically the correctness of the configured system either by a self-test procedure or checking configuration read backs [3], [14]. Both approaches can be optimized to the implemented application. Configuration read back can be performed externally (comparison of the whole configuration file with the reference one or checking its compacted signature) which is time consuming. In the case of FPGAs with stored ECC codes for each configuration frame a faster internal checking can be performed, e.g. internally read back frames are verified with ECC code checker. Adapting this approach to the configured application is not trivial due to the difficulty of localizing and selecting frames related to the used blocks. It is also worth noting that even in the case of double modular redundancy checking configuration bits is needed to restore redundant structure after detection of the faulty processor. In the opposite case the degraded simplex structure becomes susceptible to the next fault. In critical applications we can use more expensive flash based FPGAs, which can be considered as SEUs resistant.

IV. Conclusions

In many practical FPGA based systems we should use SEU mitigation techniques, in particular, they should cover configuration memory. This is a challenging problem in the case of microprocessor based systems where SEU effects from logical level propagate to microprocessor and application level. Each of these levels can introduce various fault barriers, moreover it comprises some natural fault robustness, which is characteristic to the implemented application. Hence, application oriented evaluation techniques are of great importance in the case of FPGAs. This is assured in the developed testing scheme. An interesting result is that some classical software fault mitigation techniques used in fixed logic microprocessor systems are not effective in FPGA based systems. Hence, efficient reconfiguration combined with application based microprocessor self-tests is of great importance. Developing these tests we should also verify them in fault injection experiments. This technique allowed us to identify and correct some imperfections in the previously developed (by the first author) PicoBlaze processor test used in [9]. In critical problems flash configuration memories can be used.

Further research is targeted at checking various classes of applications and improving microprocessor tests. We plan to develop application based tests for programs with more complex control flow. In particular, we can use here various structural coverage metrics (compare [23]) to select representative input data. This can be also combined with structural and pseudorandom testing schemes [23].

References

[1] C. Bolchini, A. Miele, and C. Sandionigi, "TMR and partial dynamic reconfiguration to mitigate SEU faults in FPGA," in *IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, 2007, pp. 87–95.

[2] F. L. Kastensmidt, L. Carro, and R. Reis, *Fault-tolerance techniques for SRAM-based FPGAs*. Springer, 2006, ISBN-10 0-387-31068-1.

[3] U. Legat, A. Biasizzo, and F. Nowak, "On-line self-recovery of embedded multi-processor SOC on FPGA using dynamic partial reconfiguration," *Information Technology and Control*, vol. 41, no. 2, pp. 116–124, 2012.

[4] A. Lesea *et al.*, "The rosetta experiment: atmospheric soft error rate testing in differing technology FPGAs," *IEEE Transactions on Materials Reliability*, September 2005.

[5] "Device Reliability Report," Xilinx Corporation, November 2013, UG 116 (v. 9.6).

[6] Neutron induced Single Event Upsets FAQ, Microsemi 55800021-0/8.11, August 2011.

[7] Overview of iRoC Technologies Report, "Radiation results of the SER tests of Alcatel FPGA," December 2005, 55900061-0/8.06.

[8] J. S. Monson, M. Wirthlin, and B. Hutchings, "A fault injection analysis of Linux operating on an FPGA-embedded platform," *International Journal of Reconfigurable Computing*, vol. 2012, p. 11, 2012, Article ID 850487, doi:10.1155/2012/850487.

[9] M. Wegrzyn, F. Novak, A. Biasizio, and M. Renovell, "Functional testing of processor cores in FPGA based applications," *Computing and Informatics*, vol. 28, no. 1, pp. 97–113, 2009.

[10] P. Gawkowski and J. Sosnowski, "Software implemented fault detection and fault tolerance mechanisms, part II," *Electronics and Telecommunications Quarterly*, vol. 51, no. 3, pp. 495–508, 2005.

[11] I. Koren and C. M. Krishna, *Fault tolerant systems*. Elsevier, Inc., 2007.

[12] A. R. Pandey and H. J. Patel, "Reconfiguration technique for reducing test time and test data volume in Illinois Scan Architecture Based Designs," in *IEEE VLSI Test Symposium*, 2002, pp. 9–15.

[13] M. Renovell, J. M. Portal, J. Figueras, and Y. Zorian, "Testing the interconnect of RAM based FPGAs," *IEEE Design and Test of Computers*, vol. 15, no. 1, pp. 45–50, 1998.

[14] J. Sosnowski and M. Pawłowski, *Universal and application dependent testing of FPGAs, EDCC-2 Companion Workshop on Dependable Computing*. AMK Press, 1996, pp. 111–120, ISBN 83-906582-0-8.

[15] M. Rozkovec, J. Jeníček, and O. Novák, "Application dependent FPGA testing method," in *13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools*, 2010.

[16] J. Sosnowski and M. Pawłowski, "Improving Testability in systems with FPGAs," in *22nd Euromicro Conference Beyond 2000: Hardware/Software design Strategies Short Contributions*, W. Bob, Ed. IEEE Computer Society Press, 1996, pp. pp. 236–241, ISBN 0-8186-7703-1.

[17] M. B. Tahoori, E. J. McCluskey, M. Renovell, and P. Faure, "A multi-configuration strategy for an application dependent testing of FPGAs," in *22nd IEEE VLSI Test Symposium*, 2004, pp. 154–159.

[18] G. G. Cieslewski, A. D. George, and A. M. Jacobs, "Acceleration of FPGA fault injection through multi-bit testing," in *Engineering of Reconfigurable systems and Algorithms*, July 2010.

[19] S. Rudrakshi, V. Midasala, and S. N. Bhavanam, "Implementation of FPGA based fault injection Tool (FITO) for testing fault tolerant designs," *IACSIT International Journal of Engineering and Technology*, vol. 4, no. 5, pp. 522–526, October 2012.

[20] A. Lesea, *Continuing experiments of atmospheric neutron effects on deep submicron integrated circuits*. Xilinx Corporation, October 2011, WP286.

[21] K. Chapman, *SEU strategies for Virtex – 5 devices*. Xilinx Corporation, 2010, XAP864 (v.2).

[22] PicoBlaze 8-bit Embedded Microcontroller, "User Guide for Spartan-3, Virtex-II, andVirtex-II Pro FPGAs," November 21 2005, www.xilinx.com, 1-800-255-7778 UG129 (v1.1.1).

[23] J. Sosnowski, "Software-based self-testing of microprocessors," *Journal of Systems Architecture*, vol. 52, pp. 257–271, 2006.

[24] B. Dutton and C. Stroud, "Soft core embedded processor based built-in self-test of FPGAs," in *24th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, 2009.

[25] ——, "Built-in self-test of programmable input/output tiles in Virtex-5, FPGAs," in *IEEE Southeastern Symposium on System Theory*, 2009, pp. 235–239.

[26] S. K. Venishetti, A. Akoglu, and R. Kalra, "Huerarchical built-in self-testing and FPGA based healing methodology for system on chip," in *IEEE 2nd NASA/ESA Conference on Adaptive Hardware and Systems*, 2007.