A Fault Tolerant Approach to Detect Transient Faults in Microprocessors Based on a Non-Intrusive Reconfigurable Hardware

José Rodrigo Azambuja, Samuel Pagliarini, Mauricio Altieri, Fernanda Lima Kastensmidt, Michael Hübner, Jürgen Becker, Gilles Foucard, and Raoul Velazco, *Member, IEEE*

***Abstract*—This paper presents a non-intrusive hybrid fault detection approach that combines hardware and software techniques to detect transient faults in microprocessors. Such faults have a major influence in microprocessor-based systems, affecting both data and control flow. In order to protect the system, an application-oriented hardware module is automatically generated and reconfigured on the system during runtime. When combined with fault tolerance techniques based on software, this solution offers full system protection against transient faults. A fault injection campaign is performed using a MIPS microprocessor executing a set of applications. HW/SW implementation in a reprogrammable platform shows smaller memory area and execution time overhead when compared to related works. Fault injection results show the efficiency of this method by detecting 100% of faults.**

***Index Terms*—Fault tolerance, microprocessors, reconfigurable, single event effects.**

I. Introduction

ADVANCES in the semiconductor industry, such as low voltage supply, transistor dimensions and high density integration, have dropped threshold voltages boundaries, node capacitances and tightened the noise margins, reducing the transistor reliability. With nanometer dimension technologies, transistors have become more susceptible to faults caused by radiation interference, which can be caused by energized particles present in space or secondary particles such as alpha particles, generated by the interaction of neutrons and materials at ground level [1]. Thus, high reliable applications, such as space applications or avionics, demand fault tolerance techniques capable of recovering the system from a fault with minimum implementation and performance overhead.

Manuscript received September 16, 2011; revised January 08, 2012, April 05, 2012 and May 21, 2012; accepted May 21, 2012. Date of publication July 10, 2012; date of current version August 14, 2012. This work was supported in part by CNPq and CAPES Brazilian Agencies.

J. R. Azambuja, S. Pagliarini, M. Altieri, and F. L. Kastensmidt are with the Instituto de Informática, PPGC and PGMICRO, Universidade Federal do Rio Grande do Sul (UFRGS), 91501970 Porto Alegre, Brazil (e-mail: jrazambuja@inf.ufrgs.br; snpagliarini@inf.ufrgs.br; mascarpato@inf.ufrgs.br; fglima@inf.ufrgs.br).

M. Hübner and J. Becker are with the Institut für Technik der Informationsverarbeitung (ITIV), Karlsruhe Institute of Technology (KIT), 76131 Karlsruhe, Germany (e-mail: michael.huebner@kit.edu; becker@kit.edu).

G. Foucard and R. Velazco are with the TIMA Laboratory, Institut Polytechnique de Grenoble (INP), 38031 Grenoble, France (e-mail: gilles.foucard@imag.fr; raoul.velazco@imag.fr).

Digital Object Identifier 10.1109/TNS.2012.2201750

Single event effects (SEE) faults are one of the major effects that may occur when a single radiation ionizing particle strikes the silicon. An SEE may be destructive or non-destructive. The first is defined as a permanent degradation or destruction of the device, while the second is defined as an effect that causes no permanent damage and can be reset by applying the correct signals to the device [2].

Non-destructive effects are provoked by the interaction of a single energized particle with the drain of the PN junction of the off-state transistors that causes a temporary charge or discharge in a node of the circuit. Such phenomenon generates a transient voltage pulse that can be interpreted by the circuit as an internal signal, thus provoking an erroneous result [3]. The consequence can be transient pulses in the combinational logic (SET) or bit-flips in the memory elements (SEU). In microprocessors, the consequences of SEUs or SET are errors in the data or control path [4]. The use of fault tolerance techniques, which can be based on software, hardware redundancy or a hybrid solution for microprocessors, is mandatory to detect or correct these types of transient faults.

Software-based techniques rely on adding instruction redundancy and comparison to detect or correct faults. They provide high flexibility, low development time and cost, and do not require modifications on the hardware. In addition, new generations of microprocessors that do not have RadHard versions can be used. As a result, aerospace applications may use commercial off-the-shelf (COTS) microprocessors with RadHard software. However, software-based techniques cannot achieve full system protection against soft errors [4], due to control-flow errors [5]. This limitation is due to the inability of the software to protect all the possible control-flow effects that may occur in the microprocessor.

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 and in this case are non-intrusive. They exploit special purpose hardware modules, called watchdog processors [6], to monitor memory accesses. Watchdogs usually have access to data and code stored in memory. Since they only have access to the memory, watchdogs do not detect faults that are latent inside the microprocessor or faults in the register bank.

Aiming at reducing the overheads in execution time and memory usage, the approach described in this paper combines hardware-based and software-based fault tolerance techniques

in order to detect SETs and SEUs in microprocessors. This proposed technique is based on the use of on-line control flow checker module (OCFCM) to detect faults that affect the control flow of the program and duplication of variables to detect faults affecting the data.

Fault injection campaigns show that this technique can detect 100% of the faults, with minimal memory area and execution time overheads. This technique is evaluated in a case-study system composed of a MIPS microprocessor soft-IP and OCFCM embedded in a reconfigurable field programmable gate array (FPGA).

II. Previous Work

Error detection techniques for software-based systems can be classified in three categories [7]: software-based techniques, that exploit the concepts of information, operation and time redundancy to detect errors during program execution; hardware-based techniques, which exploit area redundancy, through additional hardware modules; and hybrid techniques, that combine software- and hardware-based techniques, by adding additional hardware modules capable of analyzing additional instructions inserted in the program code.

Software-based fault tolerance techniques have been proposed in the past years. They can be automatically applied to the source code of a program, thus simplifying the task for software developers: by protecting the system during software construction, the development costs can be reduced significantly [7]. Fault tolerance techniques based on software can provide a high flexibility, low development time and cost for computer-based dependable systems. However, they usually present high overhead in memory and the degradation in execution time.

Most software-based techniques focus on protecting the system against faults affecting the data flow (errors affecting the system's data variables) or the control flow (errors affecting the execution of the running application). Techniques aiming at data flow errors, such as [8], [9], can achieve full fault detection. On the other hand, techniques aiming at control flow protection cannot achieve full fault detection. Among representative solutions can be mentioned: structural integrity checking (SIC) [10], control-flow checking by software signatures (CFCSS) [11], control-flow checking using assertions (CCA) [12] and enhanced control-flow checking using assertions (ECCA) [13]. None of these techniques can achieve full fault tolerance against soft errors. A technique called control-flow error detection through assertions (CEDA) [14] achieved up to 98.9% fault detection for control flow effect.

Hardware-based techniques exploit special purpose hardware modules, called watchdog processors [6], to monitor the control-flow of programs, as well as memory accesses. There are three types of operations which monitor the behavior of the main processor while running the application. Such operations consist of 1) checking the memory accesses performed by the microprocessor, such as the module presented in [15], that knows the portion of the memory that can be accessed by the microprocessor; 2) checking the consistency of variables, such as [16] that exploits known relationships among variables; or 3) checking control-flow checkpoints, such as [17]–[20] that check if the branches are consistent with the program flow.

There are mainly two types of watchdog processors: active and passive. The active watchdog processors execute a program concurrently with the main processor checking whether their program evolves accordingly with the one executed by the main processor. The passive watchdog processors do not run any program, they compute a signature by observing the main processor's bus then perform consistency checks. As software-based techniques, the hardware-based ones cannot achieve full fault tolerance against SEEs.

Hybrid techniques adopt software-based techniques along with a hardware module into the microprocessor system [21]. The software running on the processor core is modified to implement instruction duplication and information redundancy. Besides this regular software-based code transformation, instructions to communicate to the hardware module are inserted. By doing so, the software sends information to the hardware module about the running application. The hardware module works concurrently with the main processor doing consistency checks among duplicated instructions and verifying the correctness of the program flow by monitoring the microprocessor's accesses to the memory through the data and address buses.

Hybrid techniques are effective considering they provide a high level of tolerance while minimizing memory use and performance degradation. On the other hand, in order to be adopted, demand the source code of the application running on the processor, which is not always available.

An approach was proposed in [22], where a hardware module is introduced between a processor and its instruction memory, and charged with substituting on-the-fly the fetched code with a hardened one. However, the module did not include an ALU or a control unit and was not supported by a suitable design flow environment. In addition, the method had a significant performance overhead and could not cope with permanent faults. In [23], some hybrid solutions are evaluated in terms of execution time, memory overhead and fault detection capability. However, the presented hybrid solutions are intrusive, i.e., modifications in the microprocessor design are required.

Previously we have proposed a non-intrusive technique composed of a watchdog and signatures to improve the detection coverage of transient faults [24]. This solution achieves 100% fault detection, but with an increase of up to 2.52 times in execution time. Besides the performance degradation, the approach has scalability issues. The approach assigns a prime number and the result of the multiplication of the predecessor basic block's prime numbers to each basic block. By doing so, this approach becomes unfeasible for program codes with a large number of basic blocks or transitions between them. In other words, it is not recommended for applications with intense control-flow.

Aiming at reducing the overhead in execution time and memory size, while providing high flexibility to fault tolerant systems, an innovative fault tolerance hybrid technique is developed and demonstrated in this paper.

III. Proposed HW/SW Fault Tolerance Technique

The proposed hybrid fault tolerance technique combines a hardware-based module with two software-based program

transformation techniques. The hardware module, called online control-flow checker module monitors the address and data buses between the microprocessor and its program memory checking the memory accesses, branches and control flow checkpoints performed by the microprocessor.

As stated before, such checks are not enough to reach 100% fault tolerance and, therefore, in order to fully protect the system against faults in the microprocessor, two software-based techniques, presented in [5] are also used. The first one is referred as *Inverted Branches* and is combined with the hardware module to detect faults affecting the branch instructions in the program code. The second one, named *Variables*, aims at protecting the microprocessor system against faults that cause errors in the data.

In the following, each technique will be described in detail.

A. Hardware-Based Technique: OCFCM

The main objective of the OCFCM is to detect faults that cause incorrect jumps in the program flow. In order to do that, this module combines most of the non-intrusive hardware-based techniques, such as checking whether the microprocessor is accessing correct memory areas for data and program, the consistency of some variables, control flow checkpoints and also the program counter (PC) evolution during runtime. OCFCM is capable of doing that by storing some application oriented information and reading the address and data buses between the microprocessor and its program memory. In order to work properly, the module requires access to the memory buses. Microprocessors with instruction cache or on-chip memory may not have accessible buses and therefore cannot be protected by this technique.

The OCFCM is an application specific module and has information about the portion of memory that the application is allowed to access. By doing so, it is capable of detecting incorrect memory accesses, both for data and instructions. Some variables, such as the PC and the stack pointer (SP) are also checked through the data and address buses during runtime.

By checking the PC evolution during runtime, the OCFCM can detect if the software execution is stuck at the same memory address, by checking the number of clock cycles spent on a single instruction. Such detection is very important in microprocessor-based systems, because software-based techniques cannot achieve such detection (in order to detect errors, redundant instructions must be executed, and a loop may hold the microprocessor in a single instruction).

In addition to these fault detection capabilities, the OCFCM has the ability of checking branch instructions during runtime and verify if they performed a correct branch in the program flow. To do that, the OCFCM checks the program counter (PC) evolution through the address bus. The program executes the instruction stored in program memory sequentially until a branch instruction is found. When performing a branch instruction, a new path becomes possible, other than the normal sequential execution. In this case, the OCFCM decodes the new possible path and checks if the microprocessor is still following the program graph. It is important to mention that the OCFCM can only check branch instructions with fixed target addresses. Branch instructions with dynamic addresses (Jump to Register, for example) must be replaced (Jump to Address, for example).

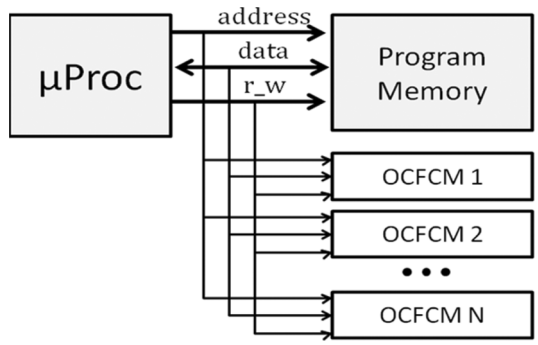

Fig. 1. Block diagram of the proposed technique.

In order to detect an inconsistency in the program flow, the instructions fetched by the microprocessor must also be fetched and decoded by the hardware module, to identify branch instructions and locate their destination address. Instead of implementing a full generic decoder, the proposed hardware module, implements a reduced decoder composed of a list of physical memory positions of all the branch instructions in the program. The decoder can be automatically generated during compilation time and allows the OCFCM to calculate each instruction's consistent destination address based only on the address and the data buses. This leads to a significant area reduction, as well as the maintenance of the original microprocessor's timing characteristics, such as the clock frequency.

On the other hand, this approach is application-specific, since it relies on a list of branch addresses. In order to adapt the proposed technique to general-purpose microprocessors, reconfiguration must be used, so that the system can reconfigure the same area with different modules, each one specifically designed for each application. Depending on the area available on chip, designers may build more than one module on the FPGA. In this case, the system can have a set of pre-defined OCFCMs, which can be switched between different programs without affecting the overall final computation time. This architecture is shown in Fig. 1.

The OCFCM is expected to detect control flow faults that either cause the PC to freeze at the same memory address (through the watchdog) or the ones that break the sequential evolution of the PC with an inconsistent destination address. Unfortunately, these two cases do not comprehend all types of control flow errors. An incorrect decision, whether to take or not the branch, cannot be detected by the module. Therefore, a software-based technique is required.

B. Software-Based Technique: Inverted Branches

Although the OCFCM technique is able to detect most control flow errors, it cannot guarantee whether a branch instruction was correctly executed or not. This happens because when a branch instruction is performed, two possible paths arise (branch taken or not taken) and both are legal from the OCFCM's point of view. In order to detect such errors, a software-based technique called *Inverted Branch* will be used.

Original Code	Hardened Code
1: beq r1, r2, 6	1: beq r1, r2, 5 **2: beq r1, r2, error**
3: add r2, r3, 1	3: add r2, r3, 1
	4: jmp 6 **5: bne r1, r2, error**
6: add r2, r3, 9 7: jmp end	6: add r2, r3, 9 7: jmp end

Fig. 2. *Inverted* Branches technique transformation.

Original Code	Hardened Code
2: ld r1, [r4]	**1: bne r4, r4', error** 2: ld r1, [r4] **3: ld r1', [r4' + offset]**
6: add r1, r2, r4	6: add r1, r2, r4 **7: add r1', r2', r4'**
10: st [r1], r2	**8: bne r1, r1', error** **9: bne r2, r2', error** 10: st [r1], r2 **11: st [r1' + offset], r2'**

Fig. 3. *Variables* technique transformation.

The main goal of this technique is to replicate the branch instructions, which are more difficult to duplicate than non-branch instructions. This is due to the fact that they have two possible paths; the branch condition is either true or false. When the condition is false (branch not taken) the branch can be simply replicated. The new instruction is added right after the original branch. Otherwise, when the condition is true (branch taken), the duplicated branch instruction must be inverted and inserted on the branch taken address.

Fig. 2 illustrates a transformation rule being applied to a piece of code protected by *Inverted Branch* technique. The conditional branch instruction *Branch if EQual* located in position 1 $(beq\ r1, r2, 6)$ will jump to position 6 if registers r1 and r2 contain the same value. Initially, the branch will be replicated and inserted right after the original instruction, in position 2. The original branch instruction is then inverted and inserted in the original branch destination address position (5). This is achieved by using the *Branch if Not Equal* instruction $(bne\ r1, r2, error)$. In this process, the original branch instruction destination address must be adjusted to the new address (5, in the hardened code).

The insertion of the replicated inverted branch instruction may affect other execution flows. For example, the instruction in position 5 cannot be executed after the add instruction located in position 3 $(add\ r2, r3, 1)$. The reason is that it could modify the value stored in the $r2$ register and cause an erroneous fault detection. In order to protect the other execution flows, the inverted branch must be protected with the instruction in position 4. Such unconditional branch does not allow the instruction in position 5 to be executed after instruction 3, but only after a branch instruction with destination address pointing to its actual position.

C. Software-Based Technique: Variables

The OCFCM technique was designed to detect faults in the control flow, while the *Inverted Branches* technique was used to complement OCFCM. Even though they should detect all the faults affecting the program flow, they are not able to detect errors on the data path, such as errors in the register file or the data memory. In order to provide a full solution against SEEs, a software-based technique called *Variables* is used. This technique, as well as the *Inverted Branches*, transforms the original program code.

The *Variables* technique replicates the variables used by the program code, replicates the write operations performed on the variables and also performs consistency checks between the variables and their replicas when a memory access is performed or a branch instruction is executed. With the consistency checks, the technique guarantees that the data being stored and read from memory are correct, as well as the data being used by branch instructions.

Fig. 3 illustrates a piece of code protected by *Variables* technique. The original code has three instructions that operate with registers and memory elements. Instructions 1 and 3 are inserted to protect the load instruction located in position 2 $(ld\ r1, [r4])$, where the first instruction verifies the register containing the base address for the load instruction $(r4)$ and its replica $(r4')$. The second instruction replicates the load instruction, using the replicated memory position $(r4' + offset)$ and loads the value into the replicated register $(r1')$. Instructions 8, 9 and 11 are inserted to protect the store instruction. While instructions 8 and 9 verify values stored in the base and data registers ($r1$ and $r2$, respectively) against their replicas ($r1'$ and $r2'$, respectively). Instruction 11 replicates the original store instruction located in position 10 $(st\ [r1], r2)$ using the replicated registers $r1'$ and $r2'$ over a replicated memory address $(r1' + offset)$.

The original add instruction located in position 6 $(add\ r1, r2, r4)$ operates only over registers and therefore does not need any offset. In order to protect this instruction, instruction 7 is inserted, which performs the original instruction, but using the replicated registers ($r2'$ and $r4'$) and writing over the replicated destination register $(r1')$.

This transformation duplicates the data being stored, i.e., the number of registers and memory addresses. Consequently, the applications are limited to a portion of the available registers and memory address. In some cases, compilers can restrict the application to a small set of registers and memory addresses, allowing the duplication. In other cases, the rules can be applied to a subset of the used registers and memory positions, although it may compromise the fault detection rate.

IV. Implementation Flow

The implementation of the technique is divided in the OCFCM implementation and the program transformation, described in the previous section. Both tasks can be performed automatically during compilation time.

The chosen case-study microprocessor is a five-stage pipeline microprocessor based on the MIPS architecture, but with a reduced instruction set. The miniMIPS microprocessor is described in [25].

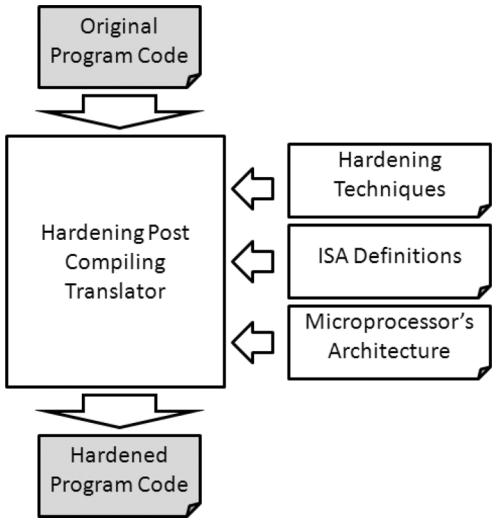

Fig. 4. HPC-Translator work flow.

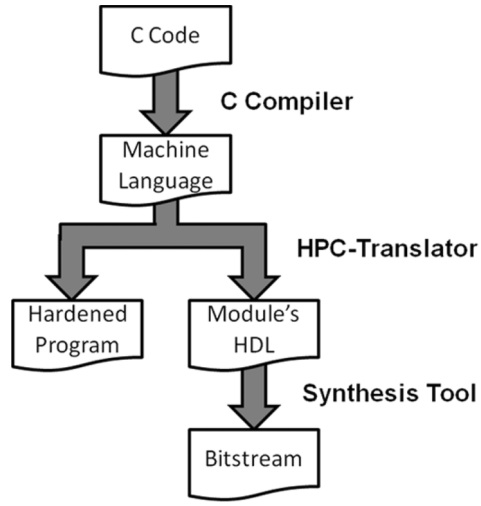

Fig. 5. Automatic hardware generation flow.

In order to automate the software transformation, we used a tool called Hardening Post Compile Translator (HPC-Translator) [24]. Implemented in Java, the HPC-Translator tool is able to automatically parse a machine code program, extract the basic blocks information and build the program flow. After extracting the program flow, this tool can insert new instructions and modify existing ones without affecting the original program tasks. The transformations to harden the original program described before can then be implemented, as well as subroutines to inform the system of an error.

Fig. 4 shows the HPC-Translator's workflow. The tool receives four distinct inputs: the original program code, the user choices of protection techniques, the instruction set architecture (ISA) definition and a file describing the microprocessor's architecture. Using these inputs, the HPC-Translator is able to generate a hardened program code.

The OCFCM also requires some information from the running application. The HPCT tool was extended to automatically read the program flow and create a Verilog file describing the customized hardware module. This function was then added to HPCT and inserted on the hardening flow in order to make it fully automatic.

The complete program transformation and hardware module generation flow is shown in Fig. 5. The input is a C code, which is compiled into an architecture-dependent machine code file and submitted to HPC-Translator, which generates a hardened program code (to be executed by the microprocessor) and a Verilog file describing the customized hardware module. The Verilog description is then synthesized to generate the final FPGA configuration bitstream.

In order to evaluate both the effectiveness and the feasibility of the presented approach, a benchmark consisting of six applications was created. As case-studies applications, we chose the following six applications: a 6×6 matrix multiplication, a bubble sort, a Dijkstra, a short encryption, a run length encoding and a bit count. The matrix multiplication requires a large amount of data processing with only a few loops and therefore uses mostly the datapath from the microprocessor. The bubble sort algorithm, on the other hand, has a large amount of loops and branch instructions and therefore uses mostly the controlpath. The Dijkstra algorithm is able to find the shortest distance between two nodes in a network and therefore is used in communication of systems on chip. The run length encoding and the short encryption are algorithms normally used in satellites in order to reduce the size of transmitted data by compressing it and to secure the communication by encrypting the transmitted data, respectively. The bit count is a small application that counts the number of bits set to '1' in a configurable fixed loop.

In the following we describe the implementation of each technique.

A. OCFCM Implementation

As mentioned before, OCFCMs are generated automatically by the HPC-Translator. In order to generate them, we input the tool with each case-study application and receive as output a Verilog description of the module. The Verilog description mainly consists of a list of every branch address in the code, application definitions (such as which memory area is allowed for data and program access) and some microprocessor definitions (such as the maximum number of clock cycles allowed to execute the same instruction).

OCFCM's were generated for both an SRAM-based FPGA and a Flash-based FPGA, considering all six case-study applications. The area required for each of them is presented in Table I, represented by the number of look-up tables and flip-flops (SRAM-based FPGAs) or VersaTiles (Flash-based FPGA). The area required for the modules is up 240 look-up tables and 33 flip-flops, or 7.8% of the miniMIPS implemented in the SRAM-based FPGA board, and up to 528 VersaTiles, or 2.9% of minimIPS implemented in the flash-based FPGA board.

Table I compares the size of the OCFCM modules to the size of the miniMIPS microprocessor. It is important to notice that each OCFCM depends only on the application and the microprocessor's Instruction Set Architecture (ISA) and therefore is

TABLE I
OCFCM TECHNIQUE AREA RESULTS COMPARED TO THE MINIMIPS MICROPROCESSOR

	SRAM-based FPGA (Virtex 4 xc4vlx80-12ff1148)		Flash-based FPGA (ProAsic3 1500)
OCFCMs	Look-up Tables (LUTs)	Flip-flops (FFs)	VersaTiles
miniMIPS	3084	1485	17978
Matrix Multiplication	200 (6.5%)	33 (2.2%)	349 (1.9%)
Bubble Sort	213 (6.9%)	33 (2.2%)	516 (2.9%)
Bit Count	192 (6.2%)	33 (2.2%)	331 (1.8%)
Dijkstra	216 (7.1%)	33 (2.2%)	424 (2.4%)
Encryption	197 (6.4%)	33 (2.2%)	340 (1.9%)
Encoding	240 (7.8%)	33 (2.2%)	528 (2.9%)

TABLE II
PARTIAL RECONFIGURATION TIME FOR SRAM-BASED FPGA (TIME IN MILLISECONDS)

Source	Reconfiguration Time
Full FPGA (2vp30ff896-7)	960 ms
Matrix Multiplication	8.5 ms
Bubble Sort	9.0 ms
Bit count	8.1 ms
Dijkstra	9.2 ms
Encryption	8.3 ms
Encoding	9.9 ms

microprocessor independent. It means that more complex microprocessors would require the same resource usage for each OCFCM and therefore a smaller percentage of its total area.

Considering the possibility to dynamically reprogram the SRAM-based FPGA, we also verified the reconfiguration time required for each module, by loading the partial bitstream from the external memory and writing it into the ICAP port. The time consumed to reconfigure the modules was measured by software, using the XTime.h library, based on a Virtex-II Pro (2vp30ff896-7) platform. In the case of the ProASIC3 FPGA, there is no partial reconfiguration, so the entire system must be reconfigured in the FPGA.

As shown in Table II, the time required to partially reprogram an OCFCM on a SRAM-based FPGA varied from 8.1 ms to 9.9 ms. This value is directly proportional to the size of each OCFCM. The reconfiguration is only necessary when the module is not implemented on the board, meaning that it is not required when using the architecture shown in Fig. 1.

B. Program Code Transformation

The code transformation was performed using the HPC-Translator, which automatically hardened the six case-study applications. We input the tool with each program's binary code and configured it to implement the software-based techniques mentioned in the previous sessions. As output, we received, for each application, a hardened version of the original program code. They implement the *Variables* and *Inverted Branches* techniques, described in machine language. The flow used was the same as the one shown Fig. 5.

TABLE III
SOFTWARE-BASED TECHNIQUE IMPLEMENTATION RESULTS (EXECUTION TIME IN MILLISECONDS)

Source	Original Program Code	Hardened Program Code
Matrix Multiplication	1.19 ms	1.79 ms (1.5x)
Bubble Sort	0.23 ms	0.40 ms (1.73x)
Bit count	3.99 ms	7.51 ms (1.88x)
Dijkstra	1.78 ms	3.15 ms (1.77x)
Encryption	0.15 ms	0.34 ms (2.26x)
Encoding	1.37 ms	2.89 ms (2.11x)

TABLE IV
SOFTWARE-BASED TECHNIQUE IMPLEMENTATION OVERHEAD (MEMORY OCCUPATION IN BYTES)

Source	Original Program Code	Hardened Program Code
Matrix Multiplication	460 bytes	1192 bytes (2.59x)
Bubble Sort	772 bytes	2200 bytes (2.84x)
Bit count	204 bytes	596 bytes (2.92x)
Dijkstra	1784 bytes	4978 bytes (2.79x)
Encryption	600 bytes	1636 bytes (2.73x)
Encoding	2736 bytes	7388 bytes (2.71x)

Tables III and IV show the overhead results for the hardened program code. Table III compares the execution time of the hardened version with the original one, while Table IV compares the program memory overhead of the hardened to their original version. Both overheads (execution time and program memory) are due to the hardening program transformation performed by HPC-Translator, which adds new instructions to the original program code, affecting directly the program memory and indirectly the execution time.

As shown in Table III, there was an increase in execution time up to $2.26\times$ the original (encryption algorithm). This is duo to redundant instructions, but mostly because of the *Variables* technique, which duplicates all variables of the program code. On the other hand, the matrix multiplication algorithm increased the execution time by $1.5\times$, showing that the overheads vary according to each application.

Table IV shows an increase of the program code that ranges from $2.59\times$ to $2.92\times$ the original size. These results mean that the original application will require at least $2.59\times$ of the original program memory size when protected. On the other hand, this increase in memory should not affect the overall system fault

TABLE V
PRELIMINARY FAULT INJECTION RESULTS FROM SIMULATION

Source	Faults Injected	Incorrect Results	Errors Detected
Matrix Multiplication	40,000	8,021	100%
Bubble Sort	40,000	8,746	100%
Bit Count	40,000	8,960	100%
Dijkstra	40,000	8,312	100%
Encryption	40,000	8,995	100%
Encoding	40,000	8,712	100%

tolerance, since memories more tolerant to radiation could be used to store the program code, such as memories hardened with information redundancy techniques.

V. FAULT INJECTION EXPERIMENTAL RESULTS

In order to start the fault injection campaign, faults were injected in all signals of the non-protected microprocessor, one per program execution. The SEU and SET types of faults were injected directly in the microprocessor VHDL code by using ModelSim SE 6.6b [26]. SEUs were injected in registered signals, while SETs were injected in combinational signals. Faults remained on the system during one and a half clock cycle, so that SETs would hit both rising and falling clock edges. The fault injection campaign was performed automatically by simulation. At the end of each execution, the results stored in memory were compared with the expected correct values.

The experiment continuously compared the Program Counter (PC) of a golden microprocessor with the PC of the faulty microprocessor and the generated data results. Fault injection results are presented in Table V. It shows the number of injected faults (*Faults Injected*) for each application, the number of faults that caused an error in the microprocessor (*Incorrect Result*) and the detection rate achieved by the proposed solution (*Errors Detected*). The system was simulated with a clock period of 42 ns and a total of 2459 signals describing it. 40 000 faults represent 16 times the number of signals, but only 0.4% of the extensive possibilities of faults for the encryption algorithm.

This fault injection campaign simulates the effects of transient faults in the case-study system is implemented in a Flash-based FPGA, the ProASIC3 from Actel, where the user's logic (VersaTiles) can be upset by SEU and SET.

Table V shows a fault injection campaign of 40 000 faults for each application. From the total amount of faults injected, around 20% affected the system and caused an error in the final result. When protected by the proposed techniques, 100% of the faults were detected. In order to confirm these results, we injected more 140 000 faults in the PC (which is the most sensitive area of the microprocessor with respect to control-flow errors) of the bubble sort application, due to its low execution time and got 100% fault detection. These results mean that the studied hardening approach was able to fully protect the microprocessor system, by detecting every transient fault injected in the case-study applications. Aside from these results, an average of 1% faults with no errors per application was detected.

When the proposed technique is compared with the method proposed in [24], which also has 100% detection capability, a significant improvement in the execution time is observed. The overhead in time has dropped from 2.5 times to less than 2 times. The same has occurred in the memory size, the overhead dropped from 3.2 times to less than 3 times. Other related works, such as CEDA [14], a pure software-based technique and [7], a hybrid intrusive technique, could not achieve such results, being able to detect up to 98% with an overhead in the execution time of up to 3 times the original.

VI. CONCLUSION AND FUTURE WORK

This paper presented a non-intrusive hybrid approach based on reconfiguration that achieved, by combining the proposed OCFCM with the software-based techniques, full fault detection against transient errors. A tool was used to automatically harden six case-study applications and a fault injection campaign was performed by injecting 240 000 faults.

Results have proved an advance of the state of the art by showing the efficiency of the proposed approach, while presenting low overhead in execution time and in memory size. The proposed hybrid technique achieved 100% fault detection for transients errors, while presenting an overhead in execution time ranging from $1.5\times$ to $2.26\times$ the original one and $2.59\times$ to $2.92\times$ overhead in program memory.

We are currently working on decreasing the impact on execution time and memory overhead of both studied techniques, while keeping the same 100% fault detection rate. In future work, we also intend to verify the feasibility and efficiency of the proposed technique applied to a real time operating system. We also intend to perform a fault injection campaign in the bitstream of the Virtex II and analyze the detection capability of the method in SRAM-based FPGAs. As a final goal, we would like to perform a radiation ground testing campaign to compare the fault injection results to those issued from the exposure to radiation beams.

REFERENCES

[1] R. C. Baumann, "Soft errors in advanced semiconductor devices-part I: The three radiation sources," *IEEE Trans. Device Mater. Rel.*, vol. 1, no. 1, pp. 17–22, Mar. 2001.
[2] F. Sexton, "Destructive single-event effects in seminconductor devices and ICS," *IEEE Trans. Nucl. Sci.*, vol. 50, no. 3, pt. 3, pp. 603–621, Jun. 2003.
[3] P. E. Dodd and L. W. Massengill, "Basic mechanism and modeling of single-event upset in digital microelectronics," *IEEE Trans. Nucl. Sci.*, vol. 50, no. 3, pt. 3, pp. 583–602, Jun. 2003.
[4] C. Bolchini, A. Miele, F. Salice, and D. Sciuto, "A model of soft error effects in generic IP processors," in *Proc. 20th IEEE Int. Symp. Defect Fault Tolerance in VLSI Syst.*, 2005, pp. 334–342.
[5] J. R. Azambuja, A. Lapolli, L. Rosa, and F. L. Kastensmidt, "Evaluating the efficiency of data-flow software-based techniques to detect SEEs in microprocessors," in *Proc. IEEE Latin-Amer. Test Workshop*, 2011, pp. 1–6.
[6] A. Mahmood and E. McCluskey, "Concurrent error detection using watchdog processors—A survey," *IEEE Trans. Comput.*, vol. 37, no. 2, pp. 160–174, Feb. 1988.
[7] E. Rhod, C. Lisboa, L. Carro, M. S. Reorda, and M. Violante, "Hardware and software transparency in the protection of programs against SEUs and SETs," *J. Electron Test*, no. 24, pp. 45–56, 2008.

[8] M. Rebaudengo, M. S. Reorda, M. Torchiano, and M. Violante, "Soft-error detection through software fault-tolerance techniques," in *Proc. IEEE Int. Symp. Defect Fault Tolerance VLSI Syst.*, 1999, pp. 210–218.

[9] C. Bolchini, L. Pomante, F. Salice, and D. Sciuto, "Reliable system specification for self-checking datapaths," in *Proc. Conf. Design, Autom. and Test Eur.*, Washington, DC, 2005, pp. 1278–1283.

[10] D. Lu, "Watchdog processors and structural integrity checking," *IEEE Trans. Comput.*, vol. C-31, no. 7, pp. 681–685, Jul. 1982.

[11] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Control-flow checking by software signatures," *IEEE Trans. Reliability*, vol. 51, no. 1, pp. 111–122, Mar. 2002.

[12] L. Mcfearin and V. Nair, "Control-flow checking using assertions," in *Proc. IFIP Int. Working Conf. DCCA-05*, Urbana-Champaign, IL, Sep. 1995, pp. 103–112.

[13] Z. Alkhalifa, V. Nair, N. Krishnamurthy, and J. Abraham, "Design and evaluation of system-level checks for on-line control flow error detection," *IEEE Trans. Parallel Distrib. Syst.*, vol. 10, no. 6, pp. 627–641, Jun. 1999.

[14] R. Vemu and J. Abraham, "CEDA: Control-flow error detection through assertions," *IEEE Trans. Comput.*, vol. 60, no. 9, pp. 1233–1245, Sep. 2011.

[15] M. Namjoo and E. J. McCluskey, "Watchdog processors and capability checking," in *Proc. 12th Int. Symp. Fault-Tolerant Comput.*, 1982, pp. 245–248.

[16] A. Mahmood, D. J. Lu, and E. J. McCluskey, "Concurrent fault detection using a watchdog processor and assertions," in *Proc. IEEE Int. Test Conf.*, 1983, pp. 622–628.

[17] M. A. Schuette and J. P. Shen, "Processor control flow monitoring using signatured instruction streams," *IEEE Trans. Comput.*, vol. 36, no. 3, pp. 264–276, Mar. 1987.

[18] M. Namjoo, "CERBERUS-16: An architecture for a general purpose watchdog processor," in *Proc. 13th Int. Symp. Fault-Tolerant Comput.*, 1983, pp. 216–219.

[19] J. Ohlsson and M. Rimen, "Implicit signature checking," in *Proc. Digest 25th Int. Symp. Fault Tolerant Comput.*, 1995, pp. 218–227.

[20] K. Wilken and J. P. Shen, "Continuous signature monitoring: Lowcost concurrent detection of processor control errors," *IEEE Trans. Comput-Aided Des. Integr. Circuits Syst.*, vol. 9, no. 6, pp. 629–641, Jun. 1990.

[21] P. Bernardi, L. M. V. Bolzani, M. Rebaudengo, M. S. Reorda, F. L. Vargas, and M. Violante, "A new hybrid fault detection technique for systems-on-a-chip," *IEEE Trans. Comput.*, vol. 55, no. 2, pp. 185–198, Feb. 2006.

[22] M. Schillaci, M. S. Reorda, and M. Violante, "A new approach to cope with single event upsets in processor-based systems," in *Proc. 7th IEEE Latin-American Test Workshop*, Mar. 2006, pp. 145–150.

[23] S. Cuenca-Asensi, A. Martinez-Alvarez, F. Restrepo-Calle, F. Palomo, H. Guzman-Miranda, and M. Aguirre, "A novel co-design approach for soft errors mitigation in embedded systems," *IEEE Trans. Nucl. Sci.*, vol. 58, no. 3, pt. 2, pp. 1059–1065, Jun. 2011.

[24] J. R. Azambuja, A. Lapolli, L. Rosa, and F. L. Kastensmidt, "Detecting SEEs in microprocessors through a non-intrusive hybrid technique," *IEEE Trans. Nucl. Sci.*, vol. 58, no. 3, pt. 2, pp. 993–1000, Jun. 2011.

[25] L. M. O. S. S. Hangout and S. Jan, "The minimips project," [Online]. Available: http://opencores.org/projects.cgi/web/minimips/overview, 2009

[26] Mentor Graphics [Online]. Available: http://www.model.com/content/modelsim-support, 2009