Non-Intrusive Reconfigurable HW/SW Fault Tolerance Approach to Detect Transient Faults in Microprocessor Systems

José Rodrigo Azambuja, Samuel Pagliarini, Mauricio Altieri, Fernanda Lima Kastensmidt, Michael Hübner, Jürgen Becker, Gilles Foucard, 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 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 minimal memory area and execution time overhead. Fault injection results show the efficiency of this method on 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 tolerant techniques capable of recovering the system from a fault with minimum implementation and performance overhead.

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 in drain PN junction of the off-state transistors that causes a temporary charge or discharge in a node of the circuit. Such phenomenon generate transient voltage pulses that can be interpreted by the circuit as internal signals, thus provoking an erroneous result [3]. The consequence can be transient pulses (SET) in the combinational logic or bit-flips (SEU) in the memory elements In microprocessors, the consequences of SEUs or SET can be errors in the data and control flow [4]. The use of fault tolerance techniques, which can be based either on software or hardware redundancy or, in the case of microprocessors, in a hybrid approach, 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, not requiring 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 in protecting all the possible control-flow effects that can 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 can have access to the data and code memory connections. Since they only have access to the memory, watchdogs do not detect faults that are latent inside the microprocessor, as well as faults in the register bank.

Aiming at reducing the overheads in execution time and memory usage, this paper combines hardware-based and software-based fault tolerance techniques in order to present a non-intrusive hybrid technique to detect SET and SEU in microprocessors. This proposed technique is based on the use of on-line control flow checker module (OCFCM) to detect

This work was supported in part by CNPq and CAPES Brazilian Agencies.

José Rodrigo Azambuja, Samuel Pagliarini, Mauricio Altieri and Fernanda Lima Kastensmidt are with the Instituto de Informática, PPGC and PGMICRO at Universidade Federal do Rio Grande do Sul (UFRGS), Porto Alegre, Brazil. (phone: +55 (51) 3308 7036) e-mail: {jrazambuja, snpagliarini, mascarpato, fglima}@inf.ufrgs.br.

Michael Hübner and Jürgen Becker are with the Institut für Technik der Informationsverarbeitung (ITIV) at Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany. (phone: +49 721 608 42504) e-mail: {michael.huebner, becker}@kit.edu.

Gilles Foucard and Raoul Velazco are with the TIMA laboratory at Institut Polytechnique de Grenoble (INP), Grenoble, France. (phone: +33 4 76 57 49 88) e-mail: {gilles.foucard, raoul.velazco}@imag.fr.

978-1-4577-0587-8/11/$26.00 ©2011 IEEE

faults that affect the control flow of the program and duplication of variables to detect faults affecting the data.

Fault injection campaigns show the efficiency of this technique to detect 100% of the faults, while leading to 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).

The paper is organized as follows. Section 2 presents previous work. Section 3 describes the proposed hybrid technique. Section 4 describes the implementation flow of the studied techniques. Section 5 presents the fault injection campaign and results. Section 6 presents main conclusions and future work.

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.

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. One of the main problems is the detection of faults that affect the program control flow [5]. Among representative solutions can be mentioned: Structural Integrity Checking (SIC) [8], Control-Flow Checking by Software Signatures (CFCSS) [9], Control-Flow Checking using Assertions (CCA) [10] and Control-Flow Checking using Assertions (ECCA) [11]. None of them can achieve full fault tolerance against soft errors. A technique called Control-flow Error Detection through Assertions (CEDA) [12] achieved up to 98.9% fault detection for control flow effect.

Hardware-based techniques consist of checking the memory accesses performed by the microprocessor, such as the module presented in [13], that knows the part of the memory that can be accessed by the microprocessor; checking the consistency of variables, such as [14] that exploits known relationships among variables; or checking control-flow checkpoints, such as [15] that checks if the branches are consistent with the program flow. As software-based techniques, the hardware-based ones cannot achieve full fault tolerance against SEEs.

In order to achieve full fault detection, hybrid solutions must be used. In [16], some hybrid solutions are evaluated in terms of execution time, memory overhead and fault detection capability. However, the presented hybrid solutions are intrusive. This means that modifications in the microprocessor design are required.

Previous work has proposed a non-intrusive technique composed of watchdog and signatures to improve the detection coverage of these types of faults [17]. This solution can achieve 100% of fault detection. However, this solution presents an increase of up to 2.52 times in execution time and an increase of up to 3.26 times in program memory. Aiming at reducing the overhead in execution time and memory sizes with high flexibility in the fault tolerance an innovative improvement in the technique proposed in [17] was developed and is presented in this paper.

III. Proposed HW/SW Fault Tolerant Technique

The proposed fault tolerant technique combines a hardware-based module with two software-based program transformation techniques. The hardware module, called online control-flow checker module (OCFCM) 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 referred one is called *Inverted Branches* and is combined with the hardware module to detect faults affecting the branch instructions in the program code. The second one, named *Variable*s, aims at protecting the microprocessor system against faults that cause errors in the data.

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

A. Hardware-based Technique: OCFCM

The first technique implemented in this work is called OCFCM. The main objective of the OCFCM is to detect faults causing incorrect jumps in the program flow. In order to do that, this module combines most of the non-intrusive hardware-based techniques characteristics, 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 that by storing some application oriented information and reading the address and data buses between the microprocessor and its program memory.

The OCFCM is an application specific module and therefore knows 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 program. 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 is also capable of detecting loops in the program execution by checking the number of clock cycles spent on a single instruction. Such detection is very important in microprocessor 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. In order 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.

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 localize their destination address. The proposed hardware module, instead of implementing a full generic decoder, implements a reduced specific decoder composed of a list of physical memory positions of all the branch instructions in the program. This decoding can be automatically generated during compilation time and allows the OCFCM to calculate each instruction's consistent destination address based only on the address bus and the data bus. This leads to a great 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 the 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.

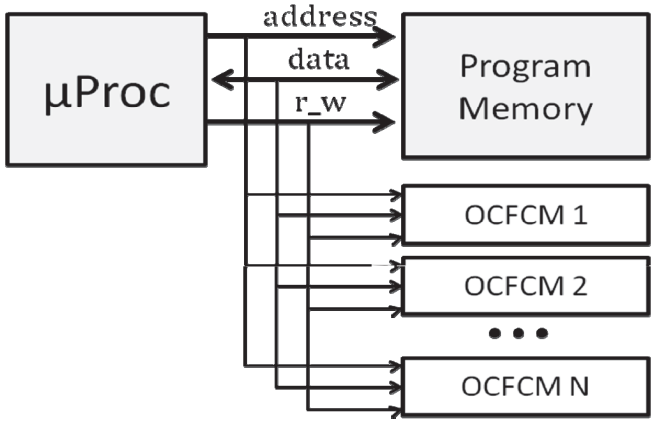

Fig. 1: Block diagram of the proposed technique.

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 whenever a branch instruction is performed, two possible paths arise (branch taken or not taken), and both are accepted by the OCFCM. In order to detect such errors, a software-based technique called *Inverted Branch* will be used.

The main objective 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, since either the branch condition is 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

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

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, will be used a software-based technique called *Variables*. As well as the *Inverted Branches*, this technique transforms the original program code, inserting new instructions.

The *Variables* technique replicates the variables used by the program code, all 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. By doing so, it 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. However, this may decrease the fault detection rate.

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

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.

In order to automate the software transformation, we used a tool called Hardening Post Compile Translator (HPC-Translator) [17]. 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.

The OCFCM also requires some information from the running application. In this paper, a new function was developed to read the program flow extracted by the HPCT tool and automatically 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 fully automatic.

The complete program transformation and hardware module generation flow can be seen in Fig. 4. As one can see, the input is a C code, which is compiled into an architecture-dependent machine code file. This file is then 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 input in a synthesis tool to generate the final FPGA configuration bitstream.

In order to evaluate both the effectiveness and the feasibility of the presented approaches, three applications were chosen as case-studies: a 6x6 matrix multiplication, a bubble sort and a bit count. The matrix multiplication requires a large data processing with only a few loops and therefore uses mostly the

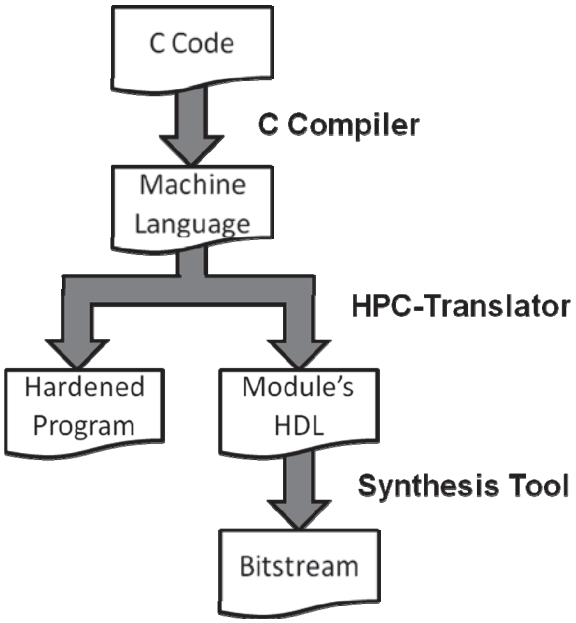

Fig. 4: Automatic hardware generation flow.

datapath from the microprocessor. The bubble sort algorithm, on the other hand, uses a large number of loops, control registers and branch instructions and therefore uses mostly the controlpath. Finally, the bit count is a small simple algorithm that combines successive sums inside a large loop.

In the following is described the implementation of each technique as well as the implementation results of the case-study applications.

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 finally some microprocessor definitions, such as the maximum number of clock cycles allowed to execute the same instruction.

OCFCM's were both generated for an SRAM-based FPGA and a Flash-based FPGA, considering all three 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 are up 213 look-up tables and 33 flip-flops, or 4.2% of the SRAM-based FPGA board, and up to 516 VersaTiles, or 2.1% of the flash-based FPGA board. It is important to notice that each OFCM depends only on the application, meaning that a bigger microprocessor would require the same area for each OCFCM and therefore a minor percentage of its total area.

TABLE I
OCFCM TECHNIQUE AREA RESULTS

	SRAM-based FPGA (Virtex-II XC2V1000-4FF896)		Flash-based FPGA (ProAsic3 1000)
OCFCMs	Look-up Tables (LUTs)	Flip-flops (FFs)	VersaTiles
FPGA Board	5120	5120	24576
Matrix Multiplication	200 (3.9%)	33 (0.6%)	349 (1.4%)
Bubble Sort	213 (4.2%)	33 (0.6%)	516 (2.1%)
Bit Count	192 (3.7%)	33 (0.6%)	331 (1.3%)

Considering the possibility to dynamically reprogram the SRAM-based FPGA, we also verified the timing 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. As shown in Table II, the maximum time required to partially program an OCFCM on a SRAM-based FPGA was 9.2ms, which can be considered a small value when compared to the execution time of the applications, up to 3.99ms. The ProASIC3 FPGA has no partial reconfiguration, so the entire system must be reconfigured in the FPGA.

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

Source	Reconfiguration Time
Matrix Multiplication	8.5 ms
Bubble Sort	9.0 ms
Bit count	8.1 ms

B. Program Code Transformation

The code transformation was performed using the HPC-Translator, which automatically hardened the three 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. The hardened versions implement the *Variables* and *Inverted Branches* techniques, described in machine language. The flow used is the same as the one shown in Fig. 4.

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. The overhead in program memory usage is introduced due to the hardening program transformation performed by HPC-Translator.

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

Source	Original Program Code	Hardened Program Code
Matrix Multiplication	1.19 ms	1.79ms (1.5x)
Bubble Sort	0.23 ms	0.40ms (1.73x)
Bit count	3.99ms	7.51ms (1.88x)

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)

As shown in Table III, there was an increase in execution time up to 1.88x the original (bit count algorithm). This happens due 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 in 1.55x.

Table IV shows an increase up to 2.92x in program code. Flash memories can be used and protected by error-correcting code techniques, such as hamming codes.

V. Fault Injection Experimental Results

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 [18].

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. SEUs were injected in registered signals, while SETs were injected in combinational signals, both during one and a half clock cycle. The fault injection campaign is performed automatically by simulation. At the end of each execution, the results stored in memory are 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 IV. It shows the number of injected faults (*Faults Injected*) for each application, the number of faults that caused an error in the microprocessor (*Wrong Result*) and the detection rate achieved by the proposed solution (*Faults Detected*). 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 IV
PRELIMINARY FAULT INJECTION RESULTS FROM SIMULATION

Source	Number of Injected Faults	Number of Errors	Number of Detected Errors
Matrix Multiplication	40,000	8,021	100%
Bubble Sort	40,000	8,746	100%
Bubble Sort	40,000	8,960	100%

Table IV shows an injection of 40,000 faults for each application. From the total 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. 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.

When the proposed technique is compared with the method proposed in [17], which also has 100% detection capability, a significant improvement in the execution time is observed. The overhead in execution time has dropped from 2.5 times to less than 2 times. The same has occurred in the memory size, where the overhead dropped from 3.2 times to less than 3 times. Other related works, such as CEDA [12], 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 technique, full fault detection against transient errors. Results have shown the efficiency of the proposed approach, while presenting low overhead in execution time and in memory size.

We are currently working on decreasing the impact on execution time and memory overhead of both studied techniques, while keeping the same fault detection rates. As future work, we intend to perform a fault injection campaign in the bitstream of SRAM-based FPGAs. As a final goal, radiation ground testing campaigns will be performed to compare the simulation results to those issued from the exposure of the circuit to radiation beams.

References

[1] R. C. Baumann, "Soft errors in advanced semiconductor devices-part I: the three radiation sources," in *IEEE Transactions on Device and Materials Reliability*, March 2001.

[2] F. Sexton, "Destructive single-event effects in seminconductor devices and ics," in IEEE Transactions on Nuclear Science, Vol. 50, Issue: 3, 2003.

[3] P. E. Dodd, L. W. Massengill. "Basic Mechanism and Modeling of Single-Event Upset in Digital Microelectronics". IEEE Transactions on Nuclear Science, vol. 50, 2003, pp. 583-602.

[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. on Defect and Fault Tolerance in VLSI Systems*, p. 334–342, 2005.

[5] J. R. Azambuja, A. Lapolli, L. Rosa, F. L. Kastensmidt, "Evaluating the Efficiency of Data-flow Software-based Techniques to Detect SEEs in Microprocessors," in Proc. IEEE Latin-American Test Workshop, 2011.

[6] A. Mahmood and E. McCluskey, "Concurrent error detection using watchdog processors – a survey", IEEE Trans on Computers, 1988.

[7] S. Cuenca-Asensi, A. Martinez-Alvarez, F. Restrepo-Calle, F. Palomo, H. Guzman-Miranda, M, Aguirre, "A Novel Co-Design Approach for Soft Errors Mitigation in Embedded Systems," in IEEE Transactions on Nuclear Science, Vol. PP. Issue: 99, 2011.

[8] D. Lu, "Watchdog processors and structural integrity checking," in IEEE Transactions on Computers, vol. C-31, no. 7, pp. 681–685, Jul. 1982.

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

[10] L. Mcfearin, V. Nair, "Control-flow checking using assertions," in Proc. IFIP Int. Working Conf. Dependable Computing for Critical Applications (DCCA-05), Urbana-Champaign, IL, Sep. 1995.

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

[12] R. Vemu, J. Abraham, "CEDA: Control-flow error detection through assertions," in Proc. of IEEE Int. On-Line Testing Symposium, 2006.

[13] M. Namjoo, E. J. McCluskey, "Watchdog processors and capability checking". In Proceedings of the 12th international symposium on fault-tolerant computing (FTCS-12), pp 245–248, 1982.

[14] A. Mahmood, D. J, Lu, E. J. McCluskey, "Concurrent fault detection using a watchdog processor and assertions". In Proceedings of the IEEE International Test Conference (ITC), pp. 622–628, 1983.

[15] M. A. Schuette, J. P. Shen, "Processor control flow monitoring using signatured instruction streams". In IEEE Trans Comput 36, 1987.

[16] J. R. Azambuja, A. Lapolli, L. Rosa, F. L. Kastensmidt, "Detecting SEEs in Microprocessors Through a Non-Intrusive Hybrid Technique," in *IEEE Transactions on Nuclear Science*, Vol. PP , Issue: 99, 2011.

[17] E. Rhod, C. Lisboa, L. Carro, M. S. Reorda, M. Violante, "Hardware and software transparency in the protection of programs against SEUs and SETs," in *Journal of Electron Test*, no. 24, pp. 45–56, 2008.

[18] L. M. O. S. S. Hangout, S. Jan. "The minimips project". Available at opencores.org/projects.cgi/web/minimips/overview, 2009.