A New Hybrid Non-Intrusive Error-Detection Technique using Dual Control-Flow Monitoring

L. Parra, A. Lindoso, M. Portela-Garcia, L. Entrena, B. Du, M. Sonza Reorda, L. Sterpone

***Abstract*—Hybrid error-detection techniques combine software techniques with an external hardware module that monitors the execution of a microprocessor. The external hardware module typically observes the control flow at the input or at the output of the microprocessor and compares it with the expected one. This paper proposes a new hybrid technique that monitors the control flow at both points and compares them to detect possible errors. The proposed approach does not require any software modification to detect control-flow errors. Fault injection campaigns have been performed on a LEON3 microprocessor. The results show full control-flow error detection with no performance degradation and a small area overhead. A complete solution can be obtained by complementing the proposed approach with software fault-tolerance techniques for data errors.**

***Index Terms*—**Microprocessors, SEEs, soft errors, fault tolerance, hybrid fault tolerance techniques

I. Introduction

As the semiconductor technology moves to smaller transistors and higher integration densities, more complex systems can be implemented in a cost-effective manner. At the same time, transistors become more susceptible to faults caused by highly energetic particles present in space or secondary particles such as alpha particles, generated by the interaction of neutron and materials at ground level. Non-destructive Single-Event Effects (SEEs), also known as soft errors, are an increasing concern for the reliability of complex digital systems. They occur when a particle strikes a node in a circuit and generates a transient voltage pulse that can propagate within the circuit [1]. When the transient pulse occurs in a memory element, such as a register, it is known as a Single-Event Upset (SEU). When the transient pulse occurs in a combinational element, the effect is known as a Single-Event Transient (SET).

Microprocessor-based systems are increasingly used in many applications, including safety-critical and high availability ones in sectors such as automotive, biomedical and aerospace. In these applications, the use of fault tolerance techniques is mandatory to detect or correct errors caused by SEEs. This requirement must be satisfied with minimum overheads in area, performance and power consumption. Moreover, due to the high effort involved in developing and qualifying a microprocessor and its associated tools, there is a growing interest in COTS (Commercial-Off-The-Shelf) microprocessors even though they may not have RadHard versions. In this case, conventional hardware-based fault tolerance techniques cannot be used and hardening must be implemented using software or hybrid techniques.

Errors produced by SEEs in a microprocessor are usually divided into data errors and control-flow errors. If an error occurs in a register or memory position storing data, a wrong computation result may be obtained. If an error occurs in a control register, such as the program counter or the instruction register, the instruction flow may be corrupted and a wrong result may be produced or the processor may lose control and enter an infinite erroneous loop. Both types of errors can be detected using software techniques. Fault tolerance techniques based on software rely on adding extra instructions to the original program code to detect or correct faults [2]. Software-based techniques provide high flexibility, low development time and low cost, since they can be implemented without modifying the hardware. However, software-based techniques cannot achieve full system protection against soft errors [3] and may produce large overheads in processing time and storage needs, particularly when designed to protect the microprocessor against control-flow errors [4],[5].

Hybrid techniques [4]-[9] combine hardware and software fault-tolerance techniques in order to improve error detection and reduce the performance degradation that software techniques entail. Hybrid fault tolerance techniques typically consist in adding an external module, often known as a watchdog processor, to monitor the execution of instructions in the processor. To this purpose, the hardware monitor is connected to the bus between the memory and the microprocessor [4],[5],[8] or to the trace interface [7],[9],[10]. In the first case, the control flow is monitored at the first pipeline stage (instruction fetch), while in the second case the control flow is monitored at the last pipeline stage after the instruction has been executed. In order to check for the correctness of the control flow, additional information must be stored in the system, usually in the form of signatures or assertions which are embedded in the software and provided to the external monitor for checking [4]-[6]. Although these

This work was supported in part by the Spanish Government under contract TEC2010-22095-C03-03.

L. Parra, A. Lindoso, M. Portela-Garcia and L. Entrena are with the University Carlos III of Madrid, Electronic Technology Department, Avda. Universidad, 30, Leganes (Madrid), Spain. (e-mails: lparra@pa.uc3m.es, alindoso@ing.uc3m.es, mportela@ ing.uc3m.es and entrena@ing.uc3m.es)
B. Du, M. Sonza Reorda and L. Sterpone are with Politecnico di Torino, 10129 Torino, Italy (e-mail: boyang.du@polito.it, matteo.sonzareorda@polito.it and luca.sterpone@polito.it)

techniques are effective, they usually introduce significant memory and performance overheads. For instance, [4] shows up to 61% performance overhead using signature-based techniques and [5] shows up to 34% performance overhead using assertions. The Program Counter (PC) Prediction technique [11] can perform some control-flow checks with no performance overhead, but it may result in low error detection for highly pipelined processors, such as LEON3.

In this work we propose a new hybrid technique that uses a dual control-flow monitoring approach. The instruction flow of the microprocessor is captured both upstream at the bus between the memory and the microprocessor and downstream at the trace interface. If an error corrupts the instruction flow at any stage, it is detected by comparing the downstream instruction flow with the upstream instruction flow. On the other hand, errors in the generation of fetch addresses are detected using a PC Prediction technique. The proposed approach can detect all errors in the program counter and the instruction register at any of the pipeline stages. These include all control-flow errors, as described in [12]. It can also detect some data errors produced by corrupted instructions that generate wrong data or data addresses without affecting the control flow.

Modern microprocessors have a pipeline of several stages. SEEs can occur at any stage provoking both data and control-flow errors that cannot be easily detected with previously proposed techniques using a single monitoring point. Error detection is enhanced by using dual monitoring at the input and at the output of the instruction stream. A major advantage of the proposed approach is that it does not require any software modification and therefore it produces no performance degradation. By complementing it with software fault-tolerance techniques to cover data errors, a complete solution against SEEs with reduced performance degradation and low memory overhead is obtained. The proposed approach has been validated by fault injection using a LEON3 processor as a case study.

The remaining of the paper is as follows. Section II summarizes related work. Section III describes the proposed dual control-flow monitoring approach. Section IV describes the software hardening approach used in this work to improve protection for data errors. Section V shows the fault injection experimental results obtained to validate the proposed approach. Finally, section VI presents the conclusions of this work.

II. Related Work

Error detection in microprocessor-based systems has been studied thoroughly in the literature. Errors produced by SEEs in a microprocessor are usually divided into data errors and control-flow errors [13]. Techniques that deal with data errors check the data consistency in the system. The most common approaches are assertion-based techniques and duplication techniques.

Assertion-based techniques insert additional statements in the code to check data correctness. These techniques are application-dependent and must be skillfully used, because the results are very sensitive to the contents and the location of the assertions in the program code.

Duplication techniques use redundant computations with different levels of granularity: instruction, block of instructions, procedure or even the entire program [14],[15]. Instructions and data are commonly duplicated to create a redundant data flow. Error detection is accomplished by comparing the results of the two data flows as often as the selected level of granularity permits. These techniques can achieve high fault coverage. However, duplicated instructions increase the code size and decrease performance. Reducing the granularity of duplication typically contributes to reduce the overheads but it increases error detection latency.

An example of duplication techniques can be found in [16]. This work proposes a set of transformation rules that can harden any high-level source code. Rules are split into data hardening and control-flow hardening rules. Data hardening rules duplicate all variables and check both copies after each write operation. Overheads can be reduced, as proposed in [17], [18], by applying duplication only to certain parts of the code.

Techniques that detect control-flow errors check the consistency of the execution flow. In this field, a broad range of solutions can be found in the literature. An updated overview of these techniques can be found in [5]. The most common control-flow techniques utilize signatures [12],[19] or assertions [20],[21] which are embedded in the software.

A common approach among control-flow checking techniques is based in dividing the program into Branch-free Blocks (BBs) and computing a signature for each executed BB. To detect errors, this signature is compared with the reference signature, which must be stored in the system for each possible BB. Alternatively, special instructions are inserted in the code to assert the beginning and end of each BB. The computation and checking of signatures or assertions usually introduce large overheads. On the other hand, control-flow checking techniques have not yet achieved full fault tolerance [3], [5].

Hardware-based techniques use hardware modifications or hardware extensions for error detection. Non-intrusive approaches usually observe the microprocessor behavior from an available bus. Approaches in this field can vary substantially. Several works rely on a watchdog processor that executes a program concurrently with the main processor (active watchdog processor) [22], [23]. Other approaches use a passive watchdog processor that verifies the signatures or assertions stored internally or produced by the main processor. Active watchdog processors produce a higher overhead than passive ones. On the other hand, passive watchdog processors require complex software modifications and larger memory.

Hybrid approaches combine the advantages of hardware and software approaches. In such scenario, the goal is to achieve good error coverage with the smallest additional hardware and a reduced performance decrease. Many different approaches can be found in the literature [4]-[9].

The use of the trace interface has recently been proposed as an alternative way to observe microprocessor execution

[7],[8],[24]. Modern microprocessors usually provide this type of interface for debugging purposes (Standard Nexus, class 2, 3 and 4, [31]). As they are useless during normal operation, they can be easily reused for on-line monitoring in an inexpensive way. On the other hand, they can provide internal access to the microprocessor without disturbing it.

III. Dual Control-Flow Monitoring

So far, hybrid approaches have been based on observing the microprocessor execution from a single suitable observation interface, which can be the memory bus or the trace interface. To detect control-flow errors, the observed control-flow information must be compared with the expected one. Storing and checking control-flow information usually introduces large overheads.

The approach proposed in this work seeks to ensure efficient error mitigation with the smallest possible overhead. To achieve this goal, the observability of the system has been incremented by observing the system behavior in two different points of the instruction stream. One of the observation points is the memory or cache bus. This bus provides information of the program counter (PC) and the instruction code (opcode and operands) at the fetch stage, just when the instruction is loaded in the microprocessor.

The second observation point is the instruction trace interface, which provides the most relevant information of each executed instruction, including the program counter (PC), instruction code (opcode and operands), time tag, and trap and error flags [25]. This information is provided just after the instruction is executed. In the case of LEON3, which has 7 pipeline stages (fetch, decode, register access, execute, memory, exception and write back), the information in the trace interface corresponds to the exception stage. The trace interface can be accessed without affecting the normal operation of the processor or adding any performance penalties. Moreover, the use of the trace interface as an observation point does not interfere with the possible use of the trace interface for debugging purposes.

Once an instruction is loaded from the memory, the instruction information travels along the microprocessor data path and is used in each stage to drive the operation of the microprocessor. An error which occurs in the PC or the instruction register (IR) at any stage will be finally observed at the trace interface and can be detected by comparing the trace interface output with the upstream information collected at the fetch stage. Notwithstanding, an error in the PC at the fetch stage may not be detected because it is issued by the microprocessor. When such an error occurs, both observation points (memory bus and trace interface) provide the very same information, but it is erroneous. To improve the detection capabilities including those errors, a PC prediction technique is used [11].

PC prediction is a control-flow checking approach that consists on predicting the next PC value by checking the opcode and the present PC value. The predicted PC value is then compared with the PC value of the next executed instruction. If there is any difference between both PCs, an error in the program flow is detected. The opcode of every new instruction executed is checked. If the opcode corresponds with a branch instruction, the PC must be incremented either by the branch offset if the branch is taken, or by the instruction size if the branch is not taken. For a non-branch instruction, the PC must be incremented by the instruction size. The trace interface provides the information required by the PC Prediction technique just after the instruction is executed.

A dedicated Hardware Monitor (HM) module has been designed to implement the proposed approach. Fig. 1 shows the HM observation points and Fig. 2 shows the internal structure of the HM. There is an interface for each observation point and a block for each implemented technique. The control block is responsible for the correct behavior of the different blocks and interfaces.

The information provided at the two observation points must be synchronized for a correct comparison. To this purpose, upstream data are stored in an input buffer with a size equal to the number of pipeline stages of the processor. When a new instruction appears in the memory bus at the fetch stage, the HM catches the opcode and the PC of the instruction and stores them in the input buffer. Then, every new instruction provided by the trace interface is compared with the instruction stored in the input buffer in order of appearance. It must be noted that the error detection latency is minimal, because an error can be detected as soon as the executed instruction appears at the trace interface.

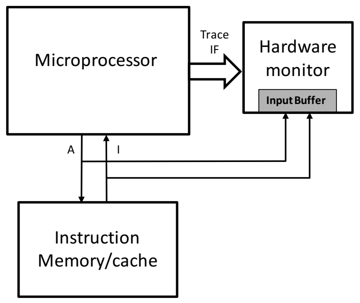

Figure 1. Hardware monitor observation points

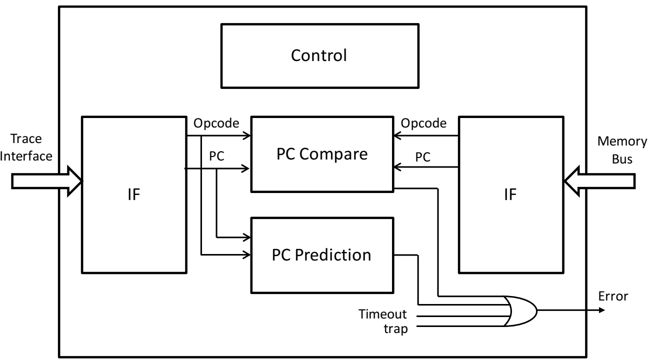

Figure 2. Internal architecture of the Hardware Monitor.

In addition to instruction comparison and PC prediction, the control module of the hardware monitor also checks the time tag and the trap and error flags provided in the trace interface. The trace interface time tag is the output of an internal counter inside the processor that is incremented each clock cycle as long as the processor is running. A timeout condition is set to cope with the case the processor hangs in a particular instruction. The timeout condition activates the error signal if the time tag advances without issuing new instructions for a long period of time.

The trap and error flags provided by the trace interface are used for exception handling. To implement exception handling it is important to differentiate between fault-induced exceptions, which may be caused by an SEE, and implemented exceptions, which are expected to occur under normal execution. The HM uses the trace interface flags to detect fault-induced exceptions that cause an unexpected trap or the processor entering error mode. Unexpected traps can be caused in several ways, such as invalid instructions or invalid memory addresses. They can be differentiated from implemented traps by checking the next instruction provided by the trace interface. The trap signal of Fig. 2 triggers when an unexpected trap occurs or the processor enters error mode.

The HM can be implemented with small hardware overhead since it does not require storing information obtained at compilation time. The proposed module can work with the observed information without disturbing the normal microprocessor behavior. Moreover, the HM can detect control-flow errors without any specific support from the application software.

IV. Data Hardening

Section III proposes a Hardware Monitor (HM) that is able to detect control-flow errors by comparing the information provided by the memory bus and the trace interface. However, data errors are not covered by the proposed approach. To achieve full error coverage, it is necessary to protect data as well as control-flow.

We have used software-based data hardening to lessen the impact of data-flow hardening in the HM. Hardware-based data-flow hardening requires additional connections to the microprocessor architecture that are not easily available. Another drawback of the hardware implementations is the area increase. Data hardening techniques require instruction re-execution with the corresponding additional storage.

Several software data hardening techniques have been proposed in the literature [13]. For complementing the proposed HM with software data hardening techniques we have selected a combination of two techniques:

- Total data-flow duplication, based in [14] and [16].
- Inverted branches, based in [8].

Total data-flow duplication duplicates all the software data and compares both data flows whenever a write operation is performed. When a discrepancy between the two data flows appears, an error is detected. This method achieves good data error coverage but increments code size and decreases performance. In order to reduce the performance penalty as well as the code size, the checking points of the code were reduced as proposed by [18]. In [18] the whole data flow is duplicated but only certain variables (final variables) are checked. A similar approach can be found in [8] where the number of checks is varied depending on the system requirements. This approach maintains the data integrity as every operation is performed twice but checking instructions are considerably reduced, at the expense of some acceptable increase in the error detection latency, i.e., the time between an error occurs and it is detected.

Another software hardening technique has been applied to our code in order to detect errors in conditional branches. In a conditional branch, errors may appear in the evaluation of the condition codes or in the condition codes themselves, resulting in the branch incorrectly taken or not taken. The technique called “inverted branches” [8] was used to detect errors in conditional branches. This technique reevaluates the branch condition in two locations. When the branch is taken, the branch instruction is repeated with an inverted condition. Additionally, when the branch is not taken the branch instruction is simply repeated. The objective of this technique is to repeat the evaluation of the condition codes. If the repeated evaluation does not produce the same result, an error is detected.

In our work all the software hardening techniques have been applied directly in high-level source code.

V. Fault Injection Experimental Results

The proposed technique has been implemented and validated on a system based on the LEON3 processor. The LEON3 processor implements the full SPARC V8 standard and it is widely used in space applications. The LEON3 core has the following main features [25]: 7-stage pipeline with Harvard architecture, separate instruction and data caches, hardware multiplier and divider, on-chip debug support and multiprocessor extensions. The register file is divided in a configurable number of register windows, so that at any one instant a program sees 8 global integer registers plus a 24-register window. The number of register windows is implementation-dependent and can be configured within the limit of the SPARC standard (2-32), with a default setting of 8.

A basic configuration has been built to perform a fault injection campaign. The system configuration includes one LEON3 integer unit with 8 register windows, instruction and data caches (2 kB each), instruction trace interface, interrupt controller, system bus (AMBA), memory controller and general purpose input/output. The memory controller can drive external RAM and ROM where code and data are stored. It must be noted that this system includes several components, besides the LEON3 processor core, that are typically needed to interface the processor. Hardening these components is beyond the scope of this work. However, they have been kept in the system because in practice it is very difficult to clearly distinguish them from the LEON3 processor hardware.

The HM logic area is about 22% the LEON3 logic area, excluding the memories that implement the register file. Table

I shows the synthesis results for a 90 nm technology. It must be noted that the utilized LEON3 configuration is minimal. Upgrading LEON3 with additional modules does not modify the HM architecture and does not increase HM area.

TABLE I. SYNTHESIS RESULTS

	#Gates	#FFs	Area(um^2)	Memory
LEON3	7,185	1,851	116,881	16Kb
HM	1,230	399	27,613	512b

Three software applications have been used for testing. The first one, (BBS), implements the Bubble Sort algorithm for a vector of 15 values. The second one, (Mmult), implements a 5x5 matrix multiplication. The third one, (AES), implements the AES encryption algorithm. In all cases, intermediate computation results are frequently sent to a parallel output port, where they can be checked during the fault injection process. All algorithms were developed in C and compiled with GCC using –O2 optimization option. To better demonstrate the capabilities of our approach, the experiments were first conducted with an unhardened version of the application software, as given by the compiler with no further manipulation to harden it. Then, the experiments were repeated with a software version that is hardened for data errors as described in section IV. The hardened software version was also developed in C and compiled with the same options.

In the experiments, we adopted the same approach as in [24] and [26] to evaluate the error detection capabilities. We estimate the global error rate using fault injection. The dynamic cross-section can then be calculated as the product of the static cross-section and the estimated global error rate. Because the static cross-section is the same for the hardened and unhardened versions of the circuit, relative comparisons can be made in terms of the global error rate. Moreover, fault injection allows us to perform a more detailed error analysis.

To obtain the global error rate, we used the AMUSE tool [27], [28]. This tool is an emulation-based fault injection system that can cover both SEU and SET, including logical, latch-window and electrical masking effects. It also provides very high performance, which enables very large fault injection campaigns to be executed in a short time. With respect to test coverage, as described in [29], AMUSE typically provides 100% coverage of expected radiation test results with respect to fault locations, input vectors and clock cycles of operation for small or medium-size test cases.

Fault injection campaigns were conducted for SEUs and SETs. For SEU experiments, we injected SEUs in every flip-flop and clock cycle, covering the full SEU space of the application. For SET experiments, we injected faults at several random instants within every clock cycle for every gate and with a pulse width of 10% of the clock period, using the approach described in [28].

In the experiments, errors were classified in several categories, following the terminology proposed in [30]. Errors that are not detected by either the HM or hardened software are classified as Silent Data Corruption (SDC) or Hang. An error is classified as SDC as soon as an erroneous output is observed at the output port. An error is classified as Hang if no new values are observed at the output port for a long time, which indicates the processor may be lost. To this purpose, we have established a timeout condition with some extra clock cycles that allows for the correct completion of the computation. An error is classified as Hang if the timeout condition is overtaken. Note that a Hang error can be produced by a control-flow error (e.g., an incorrect jump) or by a data error (e.g., an error in the index of a loop that prevents the program from finishing in due time).

Tables II to VII summarize the results of the fault injection campaigns with the HM for the three selected software applications with unhardened and hardened software versions. The internal registers of the LEON3 have been divided in two sets: Set I includes the PC & IR for all stages (346 FFs) and Set II includes the remaining registers (1,505 FFs). The first three rows in the tables show the results of SEU fault injection for the sets I and II, and all the registers, respectively. The last row shows the results of SET fault injection. From left to right, each table shows the number of injected faults, the total amount of observed errors and the classification of errors as SDC, Hang or Detected by the HM. The percentage of errors in each category with respect to the total amount of observed errors is provided in brackets.

Errors reported in the tables are true errors, i.e., errors that produce a wrong observable behavior. False errors, such as those that can happen in the hardware monitor, have not been included. The effect of a false error is to trigger an unnecessary error recovery action. For low error rates, the impact of some sporadic error recovery action is negligible. Otherwise, the hardware module can be hardened to reduce the chance of false errors.

As shown in Table II, the unhardened Bubble Sort algorithm takes 3,404 clock cycles. Therefore, we have injected 3,404 SEUs per flip-flop, up to 6,3 million SEUs in total. We have also injected 10,234 SETs per combinational node, up to 80,6 million SETs in total. Taking into account the large amount of injected faults, the error margin is smaller than 0.1% with 95% confidence [31].

TABLE II. FAULT INJECTION RESULTS (BBS, UNHARDENED SW)

Elements	Faults injected	Errors observed	SDC	Hang	Errors detected
PC & IR (I)	1.177 M	343,278	0	0	343,278 (100%)
Other Regs (II)	5.123 M	361,307	167,192 (46.3%)	36,764 (10.2%)	157,351 (43.6%)
All Regs	6.301 M	704,585	167,192 (23.7%)	36,764 (5.2%)	500,629 (71.1%)
Comb. logic (SETs)	80.649 M	777,634	258,768 (33.3%)	24,579 (3.2%)	494,287 (63.6%)

The proposed approach is able to detect 100% of the errors in Set I and many of the errors in Set II. Although Set I is much smaller than Set II, it accounts for about half of the total observed errors. This is because the PC and IR registers are very critical. In particular, Set I accounts for all control-flow errors [12]. Errors in other registers (Set II) may produce a wide variety of effects, but they can also be detected by the HM if they eventually produce a control-flow error, invalid addresses, infinite loops, etc. The HM is also able to detect a similar percentage of errors caused by SETs.

Table III summarizes the results of the fault injection campaigns using the hardened BBS application software. In this case, the application takes 8663 clock cycles, and the amount of injected faults goes up to 16 million SEUs and 205 million SETs. Again, all errors in Set I are detected. Some of these errors may be detected by software, if the software error detection triggers earlier than the HM. By combining the HM with software hardening for data errors, 92.0% of SEUs and 95.2 % of SETs are detected. The majority of the remaining undetected errors correspond to faults injected outside of the processor core, which are not covered by the proposed approach. For instance, an error in the memory controller or the bus controller may affect in a common way to duplicated variables and therefore may not be detected by the hardened software. Protection against these errors should be provided by other means, which are outside the scope of this work.

TABLE III. FAULT INJECTION RESULTS (BBS WITH SW HARDENING)

Elements	Faults injected	Errors observed	SDC	Hang	Errors detected
PC & IR (I)	2.997 M	767,712	0	0	767,712 (100%)
Other Regs (II)	13.038 M	813,107	81,996 (10.1%)	45,195 (5.6%)	685,916 (84.4%)
All Regs	16.035 M	1,580,819	81,996 (5.2%)	45,195 (2.9%)	1,453,628 (92.0%)
Comb. logic (SETs)	205.233 M	1,881,373	78,992 (4.2%)	10,448 (0.6%)	1,791,933 (95.2%)

The fault injection results for the Mmult application using unhardened and hardened software are shown in Tables IV and V, respectively. The Mmult application is more complex and requires 6,143 clock cycles to complete for the unhardened software case, and 14,788 clock cycles for the hardened software case. Therefore, the amount of injected faults increases to provide the same test coverage. The error detection capabilities are very similar to the BBS application. Again, the HM detects all errors in Set I and many of the errors in Set II. For this software application, 93.4% of SEUs and 96.9% of SETs were detected with a combination of the HM and software hardening for data errors.

For the third application (AES), the fault injection results are shown in Tables VI and VII using unhardened and hardened software, respectively. The AES application has a larger code, although it executes in less clock cycles, namely 4,377 clock cycles using unhardened software and 6,564 clock cycles for the hardened software version. The error detection capabilities are again similar to the other applications.

TABLE IV. FAULT INJECTION RESULTS (MMULT, UNHARDENED SW)

Elements	Faults injected	Errors observed	SDC	Hang	Errors detected
PC & IR (I)	2,125 M	466,416	0	0	466,416 (100%)
Other Regs (II)	9,245 M	599,949	264,051 (44.0%)	50,084 (8.3%)	285,814 (47.6%)
All Regs	11,371 M	1,066,365	264,051 (24.8%)	50,084 (4.7%)	752,230 (70.5%)
Comb. logic (SETs)	145,533 M	1,495,468	436,119 (29.2%)	25,499 (1.7%)	1,033,850 (69.1%)

TABLE V. FAULT INJECTION RESULTS (MMULT WITH SW HARDENING)

Elements	Faults injected	Errors observed	SDC	Hang	Errors detected
PC & IR (I)	5.11 M	1,173,644	0	0	1,173,328 (100%)
Other Regs (II)	22.3 M	1,586,301	65,174 (4.1%)	117,046 (7.4%)	1,404,397 (88.5%)
All Regs	27.4 M	2,759,945	65,174 (2.4%)	117,046 (4.2%)	2,577,725 (93.4%)
Comb. logic (SETs)	350.3 M	3,921,304	69,807 (1.8%)	50,633 (1.3%)	3,800,864 (96.9%)

TABLE VI. FAULT INJECTION RESULTS (AES, UNHARDENED SW)

Elements	Faults injected	Errors observed	SDC	Hang	Errors detected
PC & IR (I)	1.514 M	833,840	0	0	833,840 (100%)
Other Regs (II)	6,587M	610,326	321,583 (52.7%)	11,968 (2.0%)	276,775 (45.31%)
All Regs	8,102 M	1,444,166	321,583 (22.3%)	11,968 (0.8%)	1,110,615 (76.9%)
Comb. logic (SETs)	103.701 M	1,017,164	324,243 (31.9%)	1,439 (0.1%)	691,482 (68.0%)

TABLE VII. FAULT INJECTION RESULTS (AES WITH SW HARDENING)

Elements	Faults injected	Errors observed	SDC	Hang	Errors detected
PC & IR (I)	2,271 M	1,150,973	0	0	1,150,973 (100%)
Other Regs (II)	9,879 M	762,482	107,390 (14.1%)	30,487 (4.0%)	624,605 (81.9%)
All Regs	12,150 M	1,913,455	107,390 (5.6%)	30,487 (1.6%)	1,775,578 (92.8%)
Comb. logic (SETs)	155,511 M	1,336,025	78,167 (5.9%)	2,999 (0.2%)	1,254,859 (93.9%)

Finally, we performed several fault injection campaigns on the register file. The purpose of these campaigns is to evaluate the capability of the HM for error detection, even though errors in the register file are data errors. The register file of LEON3 consists of two RAM modules that implement the 8 register windows and the 8 global registers. The RAM modules are commonly protected by using radiation-hardened memory or using EDAC (Error Detection And Correction codes). Otherwise, software fault tolerance techniques can be used. For these campaigns we assumed that the RAM modules are not protected except by the error detection mechanisms of the HM and the implemented hardened software.

The results of the fault injection campaigns on the register files for several software applications are summarized in Table VIII. Again, we covered the full SEU space and injected SEUs in every RAM bit and clock cycle. The first three rows show the results for the BBS, Mmult and AES applications, respectively, using the HM with unhardened software. Although the injected faults produce data errors, the HM is able to detect 30.8%, 42.8% and 38.0% of them, respectively. These errors are mainly detected by the timeout and exception handling features of the HM. The next three rows in Table VIII show the results using hardened software versions. In these cases, the error detection rate rises to 93.3%, 92.4% and 97.3, respectively. The reason why no full error detection is achieved is that the compiler optimizes away some of the redundant code used for error detection. Error detection can be improved by reducing the compiler optimization level at the expense of increasing the execution time. For instance, the error detection rate in the Mmult application rises to 99.3% by reducing the compiler optimization level to –O0, as shown in the last row of Table VIII.

TABLE VIII. FAULT INJECTION RESULTS (REGISTER FILE)

Case	Faults injected	Errors observed	SDC	Hang	Errors detected
BBS HM	29.628 M	509,955	310,197 (60.8%)	42,467 (8.3%)	157,291 (30.8%)
Mmult HM	53.469 M	1,413,124	580,597 (41.1%)	227,806 (16.1%)	604,721 (42.8%)
AES HM	71.708 M	1,414,901	844,203 (59.7%)	32,470 (2.3%)	538,228 (38.0%)
BBS HM+SW	75.403 M	2,626,626	175,623 (6.7%)	0 (0%)	2,451,003 (93.3%)
Mmult HM+SW	128.715 M	1,510,704	49,016 (3.2%)	65,885 (4.4%)	1,395,803 (92.4%)
AES HM+SW	107.538 M	3,479,705	22,668 (0.7%)	72,626 (2.1%)	3,384,411 (97.3%)
Mmult HM+SW –O0	284.621 M	1,976,912	7,864 (0.4%)	6,040 (0.3%)	1,963,008 (99.3%)

When comparing the proposed approach with other related works, several aspects must be taken into account. First of all, many works usually only inject faults in some selected locations and evaluate error detection in a limited part of the microprocessor registers (typically some special registers such as PC and IR, register file, and program and data memories) [2], [6], [14]. However, this is not generally enough for complex microprocessors, which usually have many additional internal registers. On the other hand, most works provide very small test coverage or use higher-level models of the microprocessor in order to reduce the test effort.

An updated comparative between different hybrid hardening techniques is shown in [26]. Only one approach in this table can be fairly compared with this work in terms of microprocessor complexity [10], because it shows results for LEON3 and ARM microprocessors. Our work presents more complete experimental results including SETs. In [10], detection is accomplished by using two different microprocessors running the very same software or only one microprocessor using entire program duplication. Both possibilities considerably increase detection latency and either performance or area overheads. Although that approach presents a good error detection rate, the large latency and overheads that are obtained may not be acceptable in many cases.

The extension of the injection campaigns is comparable to [26]. However, the microprocessor used in that work (PicoBlaze) is much simpler. As a matter of fact, the approach used in [26] would produce poor results in the case of a strongly pipelined processor such as LEON3. The differences between the detection rate presented in this work and other works are due to the implementation of the software hardening techniques and the microprocessor architecture itself.

VI. CONCLUSIONS

This paper presents a novel hybrid approach for error detection in microprocessors which is based on monitoring and comparing the instruction flow at the input and at the output of the microprocessor. The proposed technique is intended for complex microprocessors with several pipeline stages in which instructions can be corrupted as they move into the pipeline of the processor. This technique has several advantages with respect to previous approaches that use a single observation point. Firstly, it does not require software modifications or additional information to compare with. Secondly, as the control-flow is observed at two different points, just before and after instruction execution, it can detect any error that happens in between.

Experimental results with LEON3 microprocessor demonstrate that the proposed approach can achieve 100% control-flow error detection. On the other hand, control-flow errors account for the majority of errors. By complementing it with software-based fault tolerance techniques, which are only required for protection against data errors, a complete solution against SEEs with reduced performance degradation and low memory overhead can be obtained.

REFERENCES

[1] 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, pp. 583–602, Jun. 2003.

[2] B. Nicolescu, Y. Savaria, and R. Velazco, "Software detection mechanisms providing full coverage against single bit-flip faults," IEEE Transactions on Nuclear Science, vol. 51, no. 6, pp. 3510–3518, Dec. 2004.

[3] J. R. Azambuja, S. Pagliarini, L. Rosa, and F. L. Kastensmidt, "Exploring the limitations of software-only techniques in SEE detection coverage," J. Electron. Test., no. 27, pp. 541–550, 2011.

[4] J. R. Azambuja, A. Lapolli, L. Rosa, F. L. Kastensmidt, "Detecting SEEs in microprocessors through a non-intrusive hybrid technique" IEEE Transactions on Nuclear Science, Vol. 58, no. 3, pp. 993-1000, June 2011.

[5] J. R. Azambuja, M. Altieri, J. Becker, F. L. Kastensmidt. "HETA: Hybrid Error-Detection Technique Using Assertions". IEEE Transactions on Nuclear Science, vol. 60, no. 4, pp. 2805-2812, Aug. 2013.

[6] Bernardi, P.; Sterpone, L.; Violante, M.; Portela-Garcia, M., "Hybrid Fault Detection Technique: A Case Study on Virtex-II Pro's PowerPC 405," Nuclear Science, IEEE Transactions on , vol.53, no.6, pp.3550,3557, Dec. 2006

[7] M. Grosso, M. Sonza Reorda, M. Portela-Garcia, M. Garcia-Valderas, C. Lopez-Ongil, L. Entrena "An on-line fault detection technique based on embedded debug features", Proc. 16th IEEE On-Line Testing Symposium, 2010, pp. 167-172.

[8] J. R. Azambuja, S. Pagliarini, M. Altieri, F.L. Kastensmidt, M. Hubner, J. Becker, G. Foucard, R. Velazco. "A Fault Tolerant Approach to Detect Transient Faults in Microprocessors Based on a Non-Intrusive Reconfigurable Hardware". IEEE Transactions on Nuclear Science, vol. 59, no. 4, pp. 1117-1124, Aug. 2012.

[9] Parra L., Lindoso A., Portela M., Entrena L., Restrepo-Calle F., Cuenca-Asensi S., Martínez-Álvarez A.,"Efficient Mitigation of Data and Control Flow Errors in Microprocessors", 14th European Conference on Radiation and Its Effects on Components and Systems (RADECS), 2013

[10] M. Portela-Garcia, M. Grosso, M. Gallardo-Campos, M. Sonza Reorda, L. Entrena M. Garcia-Valderas, C. Lopez-Ongil. "On the use of embedded debug features for permanent and transient fault resilience in microprocessors", Microprocessors and Microsystems, vol. 36, no. 5, pp. 334-343. July, 2012.

[11] L. Parra, A. Lindoso, M. Portela, L. Entrena, M. Grosso, M. Sonza Reorda, "Control Flow Checking through Embedded Debug Interface", Proc. 26th Conference on Design of Circuits and Integrated Systems, pp. 339-343, 2011.

[12] R. Vemu, S. Gurumurthy and J. A. Abraham. "ACCE: Automatic Correction of Control-flow Errors". Proc. Int. Test Conf. (ITC), pp. 27.1-10, 2007.

[13] M. Nicolaidis, "Soft errors in modern electronic systems" Springer 2011.

[14] P. Cheynet, B. Nicolescu, R. Velazco, M. Rebaudengo, M. Sonza Reorda, M. Violante, "Experimentally evaluating an automatic approach for generating safety-critical software with respect to transient errors", IEEE Transactions on Nuclear Science, Vol. 47, No. 6, 2000, pp. 2231-2236.

[15] H. Engel, "Data Flow transformations to Detect Results which are corrupted by hardware faults", Proc. IEEE High-Assurance System Engineering Workshop, 1997, pp. 279-285.

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

[17] A. Benso, S. Chiusano, P. Prinetto, L. Tagliaferri, "A C/C++ source-tosource compiler for dependable applications", Proceedings of the IEEE International Conference on Dependable Systems and Networks, 2000, pp. 71-78.

[18] B. Nicolescu and R. Velazco, "Detecting soft errors by a purely software approach: Method, tools and experimental results," Proc. Design, Automation and Test in Europe (DATE), pp. 57-63, 2003.

[19] E. Chielle, J. R. Azambuja, R. S. Barth, F. Almeida, F. L. Kastensmidt. "Evaluating Selective Redundancy in Data-flow Software-based Technique". IEEE Transactions on Nuclear Science, vol. 60, no. 4, pp. 2768-2775, Aug. 2013.

[20] Z. Alkhalifa, V. S.S. Nair, N. Krishnamurthy, J.A. Abraham, "Design and evaluation of System-Level Checks for On-line Control Flow Error Detection", IEEE Transactions on Parallel and Distributed Systems, vol 10, No. 6, 1999, pp.627–641.

[21] R. Vemu, J.A. Abraham, "CEDA: Control-Flow Error Detection through Assertions", Proc. 12th IEEE International On-Line Testing Symposium (IOLTS), pp. 151-158, 2006.

[22] T. Michel, R. Leveugle, G. Saucier, "A New Approach to Control Flow Checking Without Program Modification", 21th International Symposium on Fault-Tolerant Computing (FTCS-21), pp. 334-341, 1991

[23] S. Bergaoui and R. Leveugle, "IDSM: An improved control flow checking approach with disjoint signature monitoring," in Proc. Conf. on Design of Circuits and Integrated Systems (DCIS) , 2009, pp. 249–254W.

[24] Mansour, R. Velazco. "An Automated SEU Fault-Injection Method and Tool for HDL-Based Designs". IEEE Transactions on Nuclear Science, vol. 60, no. 4, pp. 2728-2733, Aug. 2013.

[25] "GRLIB IP Core User´s Manual". Version 1.0.22. Aeroflex Gaisler. January 2010

[26] L. Parra, A. Lindoso, M. Portela, L. Entrena, F. Restrepo-Calle, S. Cuenca-Asensi, A. Martinez-Alvarez. "Efficient Mitigation of Data and Control Flow Errors in Microprocessors". IEEE Transactions on Nuclear Science, vol. 61, no.4, pp. 1590-1596, Aug. 2014.

[27] L. Entrena, M. Garcia-Valderas, R. Fernandez-Cardenal, A. Lindoso, M. Portela Garcia, C. Lopez-Ongil, "Soft Error Sensitivity Evaluation of Microprocessors by Multilevel Emulation-Based Fault Injection," IEEE Transactions on Computers, pp. 313-322, March, 2012.

[28] L. Entrena, M. García-Valderas, R. Fernández-Cardenal, M. Portela, C. López-Ongil. "SET Emulation Considering Electrical Masking Effects". IEEE Transactions on Nuclear Science, vol. 56, no. 4, pp. 2021-2025, Aug. 2009.

[29] H. M. Quinn; D.A. Black, W.H. Robinson, S.P. Buchner. "Fault Simulation and Emulation Tools to Augment Radiation-Hardness Assurance Testing". IEEE Transactions on Nuclear Science, vol. 60, no. 3, pp. 2119-2142, June 2013.

[30] S. Mukherjee, C. Weaver, J. Emer, S. Reinhardt, and T. Austin, "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor", 36th Proc. International Symposium on Microarchitecture, pp. 29–40, Dec. 2003.

[31] IEEE-ISTO 5001–2003, ''The Nexus 5001 Forum™ Standard for a Global Embedded Processor Debug Interface'', Version 2.0, 2003.

[32] R. Leveugle, A. Calvez, P. Maistri, P. Vanhauwaert, "Statistical fault injection: quantified error and confidence". Proc. Design, Automation & Test in Europe (DATE'09), pp. 502-506, Apr. 2009.