

Efficient Mitigation of Data and Control Flow Errors in Microprocessors

Luis Parra, Almudena Lindoso, Marta Portela, Luis Entrena,
Felipe Restrepo-Calle, Sergio Cuenca-Asensi, Antonio Martínez-Álvarez

Abstract— The use of microprocessor-based systems is gaining importance in application domains where safety is a must. For this reason, there is a growing concern about the mitigation of SEU and SET effects. This paper presents a new hybrid technique aimed to protect both the data and the control-flow of embedded applications running on microprocessors. On one hand, the approach is based on software redundancy techniques for correcting errors produced in the data. On the other hand, control-flow errors can be detected by reusing the on-chip debug interface, existing in most modern microprocessors. Experimental results show an important increase in the system reliability even superior to two orders of magnitude, in terms of mitigation of both SEUs and SETs. Furthermore, the overheads incurred by our technique can be perfectly assumable in low-cost systems.

Index Terms— Single Event Transient, Single Event Upset, microprocessor, fault tolerance, soft error.

I. INTRODUCTION

THE use of microprocessor-based systems is gaining importance in application domains where safety is a must. In this case, errors induced by radiation in the microprocessor may cause wrong computations or even losing control of the entire system. Therefore, mitigation of Single-Event Effects (SEE) is mandatory in safety- or mission-critical applications.

SEEs, such as Single-Event Upsets (SEUs) or Single-Event Transients (SETs), may affect microprocessors in several ways. 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 stack pointer, the instruction flow may be corrupted and a wrong result may be produced or the processor may lose control and enter an infinite loop. Both data and control-flow errors need to be carefully addressed by software and hardware error mitigation techniques.

Software-based approaches have been proposed for both

data and control-flow errors. For data errors, software approaches apply redundancy at low level (assembly code) [1], [2], [3] or high-level source code by means of automatic transformation rules [4]. Also, multithreading has been applied to implement software detection and recovery solutions to mitigate faults [5]. Control-flow checking techniques are typically based on signature monitoring [6]-[11]. The program is divided into a set of branch-free blocks, where each block is a set of consecutive instructions with no branches except for possibly the last one. A reference signature is calculated at compile time and stored in the system for each block. During operation, a run-time signature is calculated and compared with the reference signature to detect control-flow errors.

Hardware-based approaches can be used for microprocessors at several abstraction levels as for any other digital device. Microprocessor specific techniques introduce system level redundancy by using multiple processors, coprocessors or specialized system modules [12]-[15]. In particular, the reuse of debug infrastructures has recently been proposed [16]. Debug infrastructures are intended to support debugging during the development phase, and are very common in modern microprocessors. As they are useless during normal operation, they can be easily reused for on-line monitoring in an inexpensive way [17]. On the other hand, they can provide internal access to the microprocessor without disturbing it, and require neither processor nor software modifications.

Both hardware and software-based techniques have advantages and disadvantages. The software-based approaches are very flexible and can be used with Commercial Off-The-Shelf (COTS) microprocessors, because no internal modifications to the microprocessor are required. However, they may produce large overheads in processing time and storage needs [18], [19]. These can be particularly very large for control-flow checking, because a large amount of signatures need to be stored and checked very often. To the contrary, hardware-based techniques can be quite effective but they usually introduce large area overheads and generally require modifications on the microprocessor, which are not feasible in COTS. However, these drawbacks can be overcome by reusing the debug infrastructures, as they use existing hardware interfaces in a non-intrusive manner.

This paper presents a new hybrid technique aimed to protect both the data and the control-flow of embedded applications running on microprocessors. On one hand, the approach is

This work was supported in part by the Spanish Government under contracts TEC2010-22095-C03-03 and PHB2012-0158-PC.

L. Parra, A. Lindoso, M. Portela 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)

F. Restrepo-Calle, S. Cuenca-Asensi and A. Martínez-Álvarez are with the University of Alicante, Computer Technology Department, Carretera San Vicente del Raspeig s/n, 03690 Alicante, Spain (e-mails: frestrepo@dtic.ua.es, sergio@dtic.ua.es and amartinez@dtic.ua.es).

based on software redundancy techniques for correcting the errors produced in the data. On the other hand, control-flow is checked by a small hardware module that monitors the sequence of instructions executed by the processor through the debug interface and detects illegal changes in the control flow. Contrarily to other hybrid approaches, the proposed approach does not require additional information to be stored internally or to be sent to the external hardware module in order to detect control-flow errors. Thus, the overheads incurred by our technique are only caused by data error correction and can be perfectly assumable in low-cost systems. The experimental results show an important increase in the system reliability even superior to two orders of magnitude, in terms of mitigation of both SEU and SET effects.

This paper is organized as follows. Section II reviews other related works. Section III describes the proposed hybrid hardening approach. Section IV shows the experimental results. Finally, Section V summarizes the conclusions of this work.

II. BACKGROUND AND RELATED WORKS

Error detection techniques for microprocessor-based systems can be classified into three categories [20]: software-based techniques, hardware-based techniques and hybrid techniques.

Software-based techniques exploit the concepts of information, operation and time redundancy to detect or correct errors during program execution. These techniques are usually implemented by automatically applying a set of transformation rules to the source code [21]. Two types of errors are generally distinguished: data-flow errors, which affect data computations, and control-flow errors, which affect the correct order of instruction execution. Techniques for data-flow error detection are typically based on instruction and data duplication and comparison [22]. For control-flow errors, there is variety of techniques which are based on assertions [9], [10] or signatures [3], [11].

Control-flow checking techniques usually divide the program into Branch-free Blocks (BBs), where each block is a set of consecutive instructions with no branches except for possibly the last one. A reference signature is calculated at compile time and stored in the system for each block. During operation, a run-time signature is calculated and compared with the reference signature to detect control-flow errors. Alternatively, special instructions are introduced 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 [14].

Hardware-based techniques use hardware modifications or hardware extensions for error detection. Because modifying the microprocessor is generally very costly, the preferred approach consists in adding an external module to monitor the control-flow of the execution. This approach is compatible with the use of COTS. The external module is often called a *watchdog processor* [12].

A watchdog processor can detect control-flow errors by

either executing a program concurrently with the main processor (active watchdog processor) or by computing the signatures of the BBs and comparing them with the expected ones (passive watchdog processor). In the active case, the overhead is very large because the watchdog processor is almost a real processor [23], [24]. In the passive case, the watchdog processor is small, but it requires a large amount of memory to store the signatures of the BBs. In addition, the code must be modified to let the watchdog processor know when the program reaches or leaves a block. Thus, code size and performance overheads increase.

For data errors, a watchdog processor can only perform some limited checks, such as checks for access to unexpected data addresses [25] or range checks for some critical variables [26]. As variable checking increases the complexity of the watchdog processor, algorithms to select the most critical variables must be used, such as [27] or [28].

The watchdog processor must be connected to the bus between the memory and the microprocessor in order to monitor the instruction and data flows. This may pose some difficulties, particularly if cache memories are used, because the cache interface is usually critical or may not be available. The use of debug infrastructures has recently been proposed as an alternative way to observe microprocessor execution [16]. Debug infrastructures are intended to support debugging during the development phase, and are very common in modern microprocessors. As they are useless during normal operation, they can be easily reused for on-line monitoring in an inexpensive way [17]. On the other hand, they can provide internal access to the microprocessor without disturbing it. In [16] the debug infrastructure is used to get the most relevant information of the execution combined with time or hardware redundancy. However, this approach requires at least a full duplication of the execution of the program. This provides full error detection but the error detection latency is very high, because errors can only be detected after the full program has been executed twice. Control-flow checking through the debug interface has been preliminarily explored in [29]. This work does not deal with data errors and results are reported only for SEUs with small fault injection campaigns.

Hybrid techniques combine both software and hardware fault tolerance techniques in order to take advantage of the benefits that each technique can provide. A reconfiguration-based approach is proposed in [15]. This approach requires switching to an application-specific module for every application program. In [30] a hybrid approach is proposed which uses an Infrastructure IP (I-IP). The I-IP checks both the control-flow and the data-flow, but the original source code must be modified to support both types of checks. In addition to data instruction duplication, special function calls are inserted at the beginning and at the end of each BB. A similar approach is proposed in [8], where the original code is extended to send either block identifiers or block signatures to the watchdog processor. The watchdog processor manages a queue of block identifiers, computes the on-line block signatures and checks for any inconsistency with the correct expected result. This approach can reach 100% fault detection

in the fault injection campaign, but at high costs of performance and area occupation. Moreover, this technique has limitations concerning its scalability to applications with large quantities of jump instructions [14]. To solve this problem, the authors propose an improved control-flow checking approach based in assertions in [14]. This approach presents the limitation that it cannot be applied with on-chip cache memories. Finally, different hybrid approaches, based on selective hardening, are evaluated in terms of execution time and memory overheads and fault detection capability in [18] and [19]. But the presented hybrid solutions are intrusive, meaning that the microprocessor used must be redesigned.

III. HYBRID HARDENING APPROACH

The approach proposed in this work relies on the use of a Control-Flow Checking (CFC) module that monitors the execution of the microprocessor without disturbing it, while data errors are corrected by using software techniques. The CFC module observes the microprocessor execution through the *trace interface*, which provides improved observability of the control-flow.

This interface is commonly directly accessible in modern processors (Standard Nexus, class 2, 3 and 4, [31]). It allows designers to control execution and access internal resources from outside the system, as well as to trace program and data. The trace interface can be accessed directly on-chip or off-chip through some standard port such as JTAG. In our case, we use a direct access. The observation of internal data is done in a non-intrusive way and it does not affect the performance of the processor. The information that can be obtained from the trace interface usually includes the value of the program counter, the operation code (opcode), instruction results, load data and store data. A detailed analysis of the trace interface for several processors can be found in [16]. In comparison, CFC techniques that use a watchdog processor observe the processor operation through the microprocessor or system bus. In this case, the address and opcode of the instructions are observed at the fetch stage, i.e., just before the instruction is executed.

Contrarily to other hybrid approaches, the proposed approach use neither signatures nor assertions to detect control-flow errors. Instead, a program counter prediction technique is used which does not require additional information to be stored internally or to be sent to the external hardware module. Therefore, control-flow error detection adds neither memory nor performance overhead.

Hardware-based control-flow checking is combined with software-based data error correction to provide a complete approach. Experimental results include both SEU and SET fault injection. The proposed approach is described in the following sections.

A. Data Hardening

Two software-based techniques are proposed in this work to be applied in different scenarios according to the maximum response time allowed, and the performance and code size constraints. Both of them are applied at low-level code

(assembly) but with different granularity levels.

The first one is an adaptation of the SWIFT-R technique proposed by Reis et al. [32]. SWIFT-R is an overall method aimed to recover faults from the data section, mainly related to the register file of the microprocessor. Similar to hardware TMR, the idea is to keep two copies of any data that come into the software protected area (also called Sphere of Replication or SoR). In our case, the borders of SoR include the entire microprocessor data-path excluding the memory and ports. Every instruction that operates with the data is replicated too. Finally, to check the consistency of the data, software majority voters and recovery procedures are inserted before any instruction that implies the data leaves the SoR (e.g., store into a memory location or write to an output port), and also before any conditional branch.

Fig. 1 presents an example of a basic program hardened using SWIFT-R (assembly code). Notice that register copies ($s0'$, $s0''$, ...) are stored in other available registers from the microprocessor register file, i.e., unused registers in the original program. Furthermore, majority voters are recovery procedures that compare if at least two versions of a register have the same value, correcting a possible corrupted data using the third copy.

Any soft-error affecting the program data within the microprocessor is masked by the copies and corrected in a short term. Data correction lasts the number of clock cycles necessary to execute the instructions of the voter and the recovery of the affected register. However, because of its fine replication granularity (instruction level) the code size and the execution time can be increased from 2.5 to 3 times compared to the non-hardened program [33]. Therefore, this method represents a suitable solution when a quick recovery time is needed and there are no severe overhead limitations.

#	Non-hardened code	SWIFT-R code
1	main: LOAD s0, 00	main: LOAD s0, 00
2		Create s0 copies
3	LOAD s1, 2A	LOAD s1, 2A
4		Create s1 copies
5	ADD s0, s1	ADD s0, s1
6		ADD s0', s1'
7		ADD s0'', s1''
8	CALL incr	CALL incr
9		Majority voter for s0
10	STORE s0, 00	STORE s0, 00
11	RETURN	RETURN
12		
13	incr: LOAD s2, 0F	incr: LOAD s2, 0F
14		Create s2 copies
15	ADD s0, s2	ADD s0, s2
16		ADD s0', s2'
17		ADD s0'', s2''
18	RETURN	RETURN

Figure 1. Code example of data hardening using SWIFT-R

The second method is based on Procedural Replication (PR) instead of instruction replication. The replication unit is the procedure (function), which is a block of code that performs a single task and returns some values. Every procedure is computed twice and recomputed a third time if a discrepancy

between the previous two computations occurs [34]. Fig. 2 illustrates this approach.

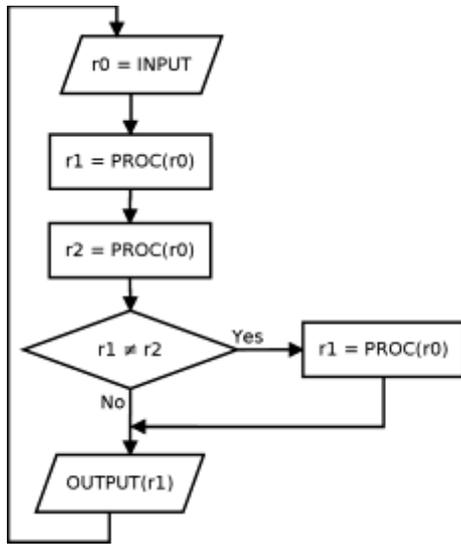


Figure 2. Data hardening using Procedural Replication (PR)

To obtain this behavior, a few code transformations are needed in the original code, involving conditional jumps and consistency checkers (inserted after procedure calls). Thus, the impact in code size is, a priori, small. In contrast, the recovery time is much longer for this second technique than for SWIFT-R. It depends on the number of instructions included in the procedure's duplicated call. Usually, the recovery time is equal to the total execution time of the original procedure plus a few additional comparisons. The recovery time is, therefore, a significant issue that must be taken into account with this method. Nonetheless, as the third procedure call only occurs in case of error detection, the execution time overhead factor during normal operation of the system (error-free state) is only 2 times. Otherwise, in the unlikely event of error recovery, this overhead is up to 3 times.

B. Control Flow Checking

The hybrid approach presented in this paper performs Control-Flow Checking (CFC) by adding a dedicated hardware module (CFC module). The module accesses to internal resources by means of the debug infrastructure available in modern microprocessors (Fig. 3). Debug Infrastructures support software debugging in embedded system development. As they are useless during normal operation, they can be easily reused for online monitoring in an inexpensive way. In addition, they can provide internal access to the processor without disturbing it and do not require any modification neither to the hardware nor to the software (i.e. no performance penalties are experimented).

The debug interface provides access to the trace buffer, which stores the most relevant information of the executed instruction, such as the instruction address, program counter (PC), instruction code (opcode), trap and error flags, etc. The trace buffer is located in the processor and read out through the trace interface. Generally, CFC techniques only observe

the opcode and the program counter of the instruction just before the instruction is executed, because they normally observe the processor operation through the cache or system bus. With this approach, internal errors cannot be easily detected unless they produce a wrong sequence of instructions, because the information is observed in the fetch stage. However, it is possible to observe the system behavior completely with the information provided by the trace interface right after the instruction is executed without modifying the normal microprocessor operation or adding any performance penalties.

The microprocessor utilized in this work does not provide a debug infrastructure. However, an implementation of a debug infrastructure similar to [35] has been accomplished to validate the proposed approach. The implemented instruction trace interface provides the program counter (PC), instruction code (opcode) and the trap flag. This information is monitored on-line by the CFC module for every executed instruction.

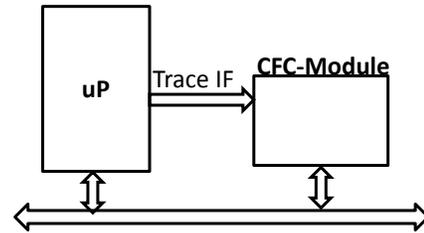


Figure 3. System structure hardened with a CFC-Module

The operation of the proposed CFC module consists on predicting the next Program Counter (PC) value, starting from the opcode of the executed instruction, and comparing it with the actual PC value for the next executed instruction. If there is any difference, an error in the execution flow is detected. Fig. 4 shows the prediction algorithm for the next program counter within the CFC module. When a non-branch instruction is considered, the program counter must be incremented by the size of the instruction. When a branch instruction is considered, the program counter must either be incremented by the branch offset if the branch is taken, or be set to the next address in the sequence if the branch is not taken.

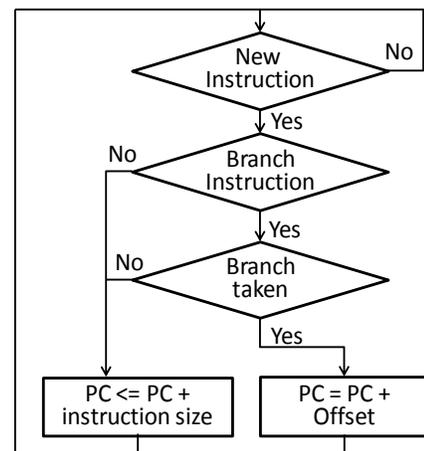


Figure 4. Next program counter prediction

This technique has preliminarily been explored in [29], obtaining a good trade-off between the fault detection coverage and area overhead. In order to apply this technique, the value of the PC and operation code of the current executed instruction must be accessible. These values are generally present in trace interfaces, and therefore, this technique is applicable to those processors that contain a trace bus. Fig. 5 shows the structure of the CFC module.

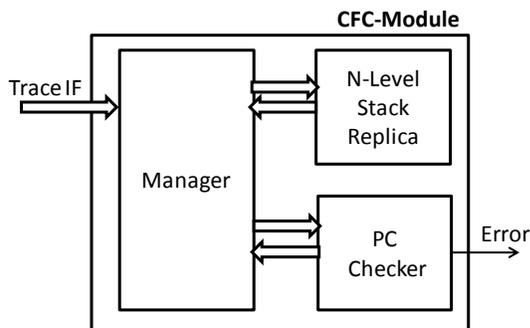


Figure 5. Block diagram to show the CFC module structure

The CFC module consists of three blocks:

- PC Checker: it compares the predicted PC value with the actual one.
- N-level Stack Replica: this block replicates N stack positions in order to check the control flow in case of call and return from subroutines.
- Manager: this block is in charge of three main tasks: decoding the executed instruction, predicting the PC value for the next instruction and managing the other blocks.

The PC prediction value of the next instruction to be executed takes into account branch and non-branch instructions, distinguishing between conditional and unconditional branches. Furthermore, when a subroutine is called, the return PC value is stored in the N-level Stack Replica. Thus, when a return instruction is executed, the predicted PC value for the next instruction can be recovered from that block and checked. A suitable trade-off between necessary hardware resources and error coverage can be achieved by selecting the number of stack positions to be replicated. For applications with low level of nested subroutines, a few stack positions should be replicated. In this paper, the experimental results have been performed with 3 replicated positions (3-Level Stack Replica).

IV. EXPERIMENTAL RESULTS AND DISCUSSION

To assess the effectiveness of our approach, we have performed extensive SEU and SET experiments in a PicoBlaze microprocessor considering several software applications and different combinations of software and hardware hardening techniques.

In the experiments, we adopted the approach proposed in [36]. This approach demonstrates that the dynamic cross-section (τ_{SEU}) can be calculated as the product of the static cross-section (σ_{SEU}) and the global error rate (τ_{NETFI}):

$$\tau_{SEU} = \sigma_{SEU} * \tau_{NETFI}$$

where τ_{NETFI} is obtained by emulation-based fault injection. Because the static cross-section σ_{SEU} is the same for all the hardened versions of the circuit, relative comparisons can be made in terms of the global error rate τ_{NETFI} . To obtain the global error rate, we used the AMUSE tool [37], which is an evolved version of the NETFI tool [36]. In relation to test coverage, as described in [38], 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. On the other hand, fault injection allows us to perform a more detailed analysis of the hardening approaches.

We worked with a compiler front-end and back-end for PicoBlaze microprocessor in order to generate the hardened software versions. PicoBlaze is a soft-microprocessor based on a RISC architecture of 8 bits, with severe limitations in performance and resources, but widely used in FPGA-based embedded systems. These facts make it especially appropriate for our case study, taking into account that PicoBlaze is mainly found in cost sensitive applications.

The PicoBlaze architecture contains 16 general-purpose 8-bit registers, and separate memories for data (internal 64bytes scratchpad RAM) and code (1024words ROM). It also includes a stack memory (31 words) to support subroutine calls and up to 256 input and output ports. In the multi-cycle implementation of PicoBlaze all the instructions take the same number of cycles to be executed, namely two cycles. In addition to the general-purpose register file, PicoBlaze has other internal register such as the Program Counter (PC) and the Stack Pointer (SP). Because these internal registers are not accessible by software instructions, control-flow error detection is difficult to perform by software-based techniques.

In this case study, a cycle accurate and RTL equivalent clone of the original PicoBlaze-3 version (RTL PicoBlaze) has been used. The PicoBlaze design has been extended with a trace interface, similar to the one that can be found for LEON3 [35], and the CFC module has been connected to it. Table I summarizes the synthesis results. The CFC module implies an overhead of 435 gates and 119 FFs. This corresponds to an area overhead of about 40% because PicoBlaze is a very small microprocessor. For more complex microprocessors, the overhead can be expected to be much smaller. The CFC module does not affect the maximum operation frequency of the processor.

TABLE I. SYNTHESIS RESULTS

	#Gates	#FFs	Max Freq (MHz)
Microprocessor	1066	194	350
CFC	435	119	670

The architecture was synthesized for a 90 nm technology using the SAED90 nm library provided by Synopsys [39]. For SEU experiments, we injected SEUs at every FF and clock cycle. For SETs experiments, we injected faults at several

random instants within every clock cycle for every gate and with a pulse width of 500 ps (approximately 20% of the clock period), using the approach described in [40]. The number of injected faults varies with the software application and the selected hardening techniques. It ranges from 17,483 faults/node to 105,344 faults/node (up to 111 million faults in total for SETs and up to 20 million faults in total for SEUs). All nodes are fault injected. Then, for each node, we generate a list of fault injection instants with a random time interval between two consecutive injection instants. The mean of this random time interval is 500 ps (equal to the pulse width). The clock period is 2.67 ns. Therefore, 5.3 faults are injected in every clock cycle on average and we can ensure that all clock cycles are covered for SETs.

In the experiments, faults were classified according to their effect on the program behavior as proposed in [41]. Silent Data Corruption (SDC) failures are faults that have not been detected or corrected and make the program finish with an erroneous output. Hang failures are the ones that provoke abnormal program termination or an infinite loop. To detect this type of failures, we established a timeout condition with some allowed extra clock cycles for the computation to complete correctly (the timeout value depends on the application and the software hardening technique, ranging from 100 to 3300 clock cycles for the performed experiments).

Three different software applications were used for the experiments: matrix multiplication (Mmult), a Proportional-Integral-Derivative Controller (PID) and a Finite Impulse Response filter (FIR). In the experiments, we tested every application with no hardening at all (NH) and hardened with SWIFT-R (SR), Procedural Replication (PR) and all of the above combined with CFC (NH+CFC, SR+CFC and PR+CFC).

Tables III and IV summarize the results for SEUs and SETs, respectively. The percentage of SDC and HANG failures are reported for all the software versions: 3 benchmarks (Mmult, PID and FIR), non hardened versions (NH) and 2 different software hardening techniques (SR and PR), with and without CFC module.

The experimental results demonstrate that the combination of SW hardening for data errors and HW hardening for control-flow errors is able to mitigate almost all errors. Generally, the CFC module removes most Hang errors and the SR or PR techniques remove most SDC errors, but both techniques are required to produce a relevant mitigation. The best results are achieved with the combination of PR and CFC module. The last column in Tables III and IV shows the relative error rate reduction that is obtained in this case with respect to the NH version. This reduction can be as large as 114 times, i.e., more than 2 orders of magnitude, in the FIR case.

When selecting a software technique, it must be taken into account that the studied software techniques require different recovery times and memory overheads. SWIFT-R is less effective than PR and it also introduces higher code and execution time overheads, as reported in Table II.

TABLE II. STATIC CODE SIZE AND EXECUTION TIME OVERHEADS

Program	Size (#instructions)	Version	Code size Overhead	Execution time Overhead
<i>Mmult</i>	3304	SR	3.02×	2.55×
		PR	1.65×	2.03×
<i>PID</i>	7390	SR	2.37×	2.65×
		PR	1.24×	2.02×
<i>FIR</i>	7374	SR	2.85×	2.68×
		PR	1.07×	1.96×

In contrast, PR has significant recovery times (ranging from 590 to 3300 clock cycles for the considered benchmarks), while the recovery time for SWIFT-R is negligible.

Finally, Table V shows a summary comparison with some of the hybrid techniques reported in Section II. The results shown in Table V are taken from the papers referenced in the first column. The proposed technique does not require software modifications for control-flow error detection. Thus, the execution time overhead is only due to data hardening. When the SR technique is used, the overhead is slightly larger because it includes both data error detection and recovery. Code size overhead is smaller or similar. The approaches differ in the fault injection techniques: [30] uses software fault injection, while [8], [14] and [16] inject faults directly in the signals of the microprocessor VHDL code. In contrast, we inject into every node of the final synthesized netlist and consider the real delays as estimated by the synthesis tool. With respect to error detection, our results are close to 100%, but the fault injection experiments are much larger and more accurate.

V. CONCLUSIONS

An efficient SEU and SET mitigation approach for embedded applications running on microprocessors has been proposed that combines software-based techniques for data errors and non-intrusive hardware-based techniques for control-flow errors. The results demonstrate that up to two orders of magnitude of improvement in the soft error rate can be achieved with this approach.

REFERENCES

- [1] N. Oh, S. Mitra, and E. J. McCluskey, "(EDI)-I-4: error detection by diverse data and duplicated instructions," *IEEE Transactions on Computers*, vol. 51, no. 2, pp. 180–199, 2002.
- [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] 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
- [4] M. Rebaudengo, M. Reorda, and M. Violante, "A new approach to software-implemented fault tolerance," *Journal of Electronic Testing-Theory and Applications*, vol. 20, no. 4, pp. 433–437, Aug 2004.
- [5] T. Vijaykumar, I. Pomeranz, and K. Cheng, "Transient-fault recovery using simultaneous multithreading". *Proc. 29th Annual Int. Symposium on Computer Architecture*, 2002, pp. 87–98, , 2002.

- [6] M. Sonza Reorda. "Software-level soft-error mitigation techniques". In "Soft errors in modern electronic systems", M. Nicolaidis (Ed.), Springer, 2011.
- [7] N. Oh, P.P. Shirvani, E.J. McCluskey, "Control-Flow Checking by Software Signatures", IEEE Transactions on Reliability, vol. 51, No. 2, pp. 111-122, 2002,
- [8] 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.
- [9] 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.
- [10] 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.
- [11] R. Vemu, S. Gurumurthy, and J. A. Abraham, "ACCE: Automatic correction of control-flow errors," Proc. Int. Test Conference, pp. 1-10, 2007
- [12] A. Mahmood and E. McCluskey, "Concurrent error-detection using watchdog processors," IEEE Transactions on Computers, vol. 37, no. 2, pp. 160-174, Feb 1988.
- [13] S. Mukherjee, M. Kontz, and S. Reinhardt, "Detailed design and evaluation of Redundant Multithreading alternatives," Proc. 29th Int. Symposium on Computer Architecture, pp. 99-110, May 2002.
- [14] 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.
- [15] 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.
- [16] 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
- [17] 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.
- [18] Cuenca-Asensi, S.; Martinez-Alvarez, A.; Restrepo-Calle, F.; Palomo, F. R.; Guzman-Miranda, H.; Aguirre, M. A.; "A Novel Co-Design Approach for Soft Errors Mitigation in Embedded Systems," IEEE Transactions on Nuclear Science, vol. 58, no. 3, pp. 1059-1065, June 2011
- [19] A. Lindoso, L. Entrena, E. San Millán, S. Cuenca-Asensi, A. Martínez-Alvarez, F. Restrepo-Calle. "A Co-Design Approach for SET Mitigation in Embedded Systems". IEEE Transactions on Nuclear Science, vol. 59, no. 4, pp. 1034-1039, Aug. 2012
- [20] 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.
- [21] 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.
- [22] 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.
- [23] 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
- [24] 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. 249-254
- [25] M. Namjao and E. J. McCluskey, "Watchdog processors and capability checking," in Proc. Int. Symp. Fault Tolerant Computing, 1982, pp. 245-248.
- [26] 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.
- [27] S. Bergaoui, P. Vanhauwaert and R. Leveugle. "A New Critical Variable Analysis in Processor-Based Systems". IEEE Transactions on Nuclear Science, vol. 57, no. 4, pp. 1992-1999, Aug. 2010.
- [28] A. Benso, S. Di Carlo, G. Di Natale, P. Prinetto, "A Watchdog Processor to Detect Data and Control Flow Errors", 9th IEEE International On-Line Testing symposium (IOLTS), pp. 144-148, July 2003.
- [29] 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.
- [30] 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
- [31] IEEE-ISTO 5001-2003, "The Nexus 5001 Forum™ Standard for a Global Embedded Processor Debug Interface", Version 2.0, 2003.
- [32] G. A. Reis, J. Chang, and D. I. August, "Automatic instruction-level software-only recovery," IEEE Micro, vol. 27, no. 1, pp. 36-47, 2007.
- [33] A. Martínez-Álvarez, S. Cuenca-Asensi, F. Restrepo-Calle, F.R. Palomo Pinto, H. Guzmán-Miranda, M.A. Aguirre, "Compiler-Directed Soft Error Mitigation for Embedded Systems," IEEE Trans. Dependable and Secure Computing, vol. 9, no. 2, pp. 159-172, March/April, 2012.
- [34] N. Oh, and E. McCluskey, "Error detection by selective procedure call duplication for low energy consumption," IEEE Transactions on Reliability, vol. 51, no. 4, pp. 392-402, Dec., 2002.
- [35] "GRLIB IP Core User's Manual". Version 1.0.22. Aeroflex Gaisler. January 2010.
- [36] W. 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
- [37] 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.
- [38] 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.
- [39] SAED 90nm Generic Library, Synopsys Armenia Educational Department, [Online]. Available: <http://www.synopsys.com/Community/UniversityProgram>
- [40] 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.
- [41] 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.

TABLE III. SEU FAULT INJECTION RESULTS

Benchmark	Error Type	NH	SR	PR	NH+CFC	SR+CFC	PR+CFC	Rel. Red.
Mmult	SDC (%)	9.65%	2.92%	0.30%	8.43%	2.18%	0.15%	113 ×
	Hang (%)	6.94%	4.74%	4.98%	0.00%	0.00%	0.00%	
	TOTAL (%)	16.59%	7.66%	5.28%	8.43%	2.18%	0.15%	
PID	SDC (%)	12.97%	4.73%	0.80%	10.51%	2.18%	0.15%	15 ×
	Hang (%)	7.77%	5.27%	5.73%	1.95%	0.09%	1.25%	
	TOTAL (%)	20.74%	10.00%	6.53%	12.47%	2.28%	1.40%	
FIR	SDC (%)	12.39%	3.21%	0.88%	9.91%	0.69%	0.15%	114 ×
	Hang (%)	5.55%	5.20%	4.79%	0.01%	0.01%	0.01%	
	TOTAL (%)	17.94%	8.41%	5.67%	9.91%	0.69%	0.16%	

TABLE IV. SET FAULT INJECTION RESULTS

Benchmark	Error Type	NH	SR	PR	NH+CFC	SR+CFC	PR+CFC	Rel. Red.
Mmult	SDC (%)	0.83%	0.30%	0.05%	0.17%	0.16%	0.02%	78 ×
	Hang (%)	0.90%	0.61%	0.76%	0.00%	0.00%	0.00%	
	TOTAL (%)	1.73%	0.91%	0.80%	0.17%	0.16%	0.02%	
PID	SDC (%)	1.30%	0.64%	0.05%	0.85%	0.16%	0.01%	22 ×
	Hang (%)	1.18%	0.93%	0.95%	0.14%	0.01%	0.10%	
	TOTAL (%)	2.48%	1.56%	1.00%	0.98%	0.17%	0.11%	
FIR	SDC (%)	1.20%	0.56%	0.12%	0.74%	0.07%	0.02%	114 ×
	Hang (%)	1.04%	0.92%	0.92%	0.00%	0.00%	0.00%	
	TOTAL (%)	2.24%	1.48%	1.04%	0.74%	0.07%	0.02%	

TABLE V. COMPARISON WITH OTHER APPROACHES

Approach	SW Support	Monitored bus	Latency	Case Study		Overheads			Faults Injected	Error detection
				Processor	Benchmarks	Area	Time	Code		
[30]	data & ctrl	memory buses	low	PowerPC	MMult, Ellip-filter, LZW, Viterbi	366 slices	2.06-2.81	2.07-3.09	SEUs 100K	SEU: 90.6-95.6%
[8]	data & ctrl	memory buses	low	miniMIPS	BubbleSort, MMult	15% (128 FFs)	2.33-2.53 (ctrl. only 1.33-1.60)	3.26-3.60	SEEs 50K	SEU: 100% SET: 100%
[14]	data & ctrl	memory buses	low	miniMIPS	BubbleSort, MMult	11%	(ctrl. only 1.08-1.34)	2.79-2.91	SEEs 100K	SEU: 100% SET: 100%
[16]	data & ctrl	Dbg./Trace Interface	high	LEON, ARM7	Fibonacci, Ellip-filter	16%	2.00	-	SEUs 10K	SEU: 99%
SR+CFC	data only	Dbg./Trace Interface	low	Picoblaze	MMult, PID, FIR filter	435 gates 119 FFs	2.55-2.68 (ctrl. only 1)	2.37-3.02	SEEs 130M	SEU: 97.72-99.31% SET 99.83-99.94%
PR+CFC	data only	Dbg./Trace Interface	med (data) low (ctrl)	Picoblaze	MMult, PID, FIR filter	435 gates 119 FFs	1.96-2.03 (ctrl. only 1)	1.07-1.65	SEEs 130M	SEU 98.60-99.85% SET 99.89-99.98%