Pena-Fernandez, M., Lindoso, A., Entrena, L., Garcia-Valderas, M., Morilla, Y. & Martin-Holgado, P. (2019). Online Error Detection Through Trace Infrastructure in ARM Microprocessors. *IEEE Transactions on Nuclear Science*, 66(7), pp. 1457–1464.

DOI: 10.1109/tns.2019.2921767

Online error detection through trace infrastructure in ARM microprocessors

M. Peña-Fernandez, A. Lindoso, L. Entrena, M. Garcia-Valderas, Y. Morilla, P. Martín-Holgado

***Abstract*—This work presents a solution for error detection in ARM microprocessors based on the use of the trace infrastructure. This approach uses the Program and Instrumentation Trace Macrocells that are part of ARM's CoreSight™ architecture to detect control-flow and data-flow errors, respectively. The proposed approach has been tested with low-energy protons. Experimental results demonstrate high accuracy with up to 95% of observed errors detected in a commercial microprocessor with no hardware modification. In addition, it is shown how the proposed approach can be useful for further analysis and diagnosis of the cause of errors.**

***Index Terms*— ARM, microprocessor trace, fault tolerance, error detection.**

I. Introduction

PROCESSORS for space applications are generally required to be hardened or tolerant to radiation effects. However, the available choices of rad-hard processors are scarce and very expensive. Moreover, rad-hard processors usually lag several generations behind COTS (Components Off-The-Shelf) microprocessors and SoCs (System on-a-Chip) [1], [2]. Thus, there is a growing interest in the use of COTS microprocessors in space applications, particularly for small satellites with tight budget constraints and missions that require high processing power. COTS microprocessors can enable scientific missions and reduce development time and cost, provided that sufficient error detection or mitigation is achieved [3].

Mitigation of radiation effects in COTS microprocessor is difficult because the hardware cannot be modified and there are many internal elements that are not accessible. Error detection and recovery is more generally used, which can be based on software-implemented fault tolerance or processor redundancy (e.g., lock-step techniques). Software techniques are limited and introduce severe penalties in terms of performance. Processor redundancy is costly in terms of hardware resources and power consumption. These approaches generally make use of external hardware units for comparison of the computation results and error checking [4-12].

In this work we propose a solution based on on-chip trace infrastructures for error detection in COTS microprocessors. Debug and trace macrocells are commonly included in COTS microprocessors to support the increasing complexity of software debugging tasks. However, once the application has been developed, they become useless and can be reused for on-line monitoring in an inexpensive way. In particular, trace infrastructures can provide a good deal of information about the instructions executed by a microprocessor in a non-intrusive manner.

The use of the trace interface for error detection and correction has been successfully demonstrated in [6], [7]. Errors are detected or corrected by observing the instruction flow and comparing it among several executions in the same processor or in different processors. Later works based on this idea [8-11] focused on soft cores, where the program-flow trace can be conveniently accessed through a raw or a custom interface. However, the access to the trace in COTS microprocessors is generally provided through hard macrocells that impose specific access protocols and limit the available information. Moreover, the program-flow trace can only detect control-flow errors. To cover data-flow errors, these techniques need to be combined with software-implemented techniques [8-10].

The processor selected for this work is an ARM microprocessor, which is a very popular choice in the commercial market. Debug and trace support is provided in the ARM processors by the CoreSight™ architecture. The ARM Cortex-A9 family of processors includes two CoreSight macrocells: the Program Trace Macrocell (PTM) and the Instrumentation Trace Macrocell (ITM). In the proposed approach, the PTM and the ITM are used to detect control-flow and data-flow errors, respectively. The program-flow trace is monitored through the PTM, while selected computed data are monitored through the ITM. Both traces are checked by an external hardware module developed for this purpose, that we have called Program & Data Trace Checker (PDTC). To the best of our knowledge, this is the first time that a full approach based on trace macrocells, intended to detect both control-flow and data-flow errors, is proposed and tested under radiation.

In addition, the proposed approach can be useful for error diagnosis, as it is able to collect the trace of the processor at the

This work has been supported in part by the projects ESP-2015-68245-C4-1-P and ESP2015-68245-C4-4-P (Spanish MINECO) and by the Community of Madrid under grant IND2017/TIC-7776.

M. Peña-Fernández is with Arquimea Ingenieria SLU., Leganes, Madrid, Spain (email: mpena@arquimea.com)

A. Lindoso, L. Entrena and M. García-Valderas are with the Department of Electronic Technology, Universidad Carlos III de Madrid, Avda. Universidad 30, E-28911 Leganes, Madrid, Spain (e-mail: alindoso@ing.uc3m.es; entrena@ing.uc3m.es; mgvalder@ing.uc3m.es).

Y. Morilla and P. Martín-Holgado are with the Centro Nacional de Aceleradores (Universidad de Sevilla, CSIC, JA). Avda. Tomás Alba Edison nº 7, E-41092 Sevilla, Spain (e-mail: ymorilla@us.es; pmartinholgado@us.es).

.

time an error occurs. To this purpose, a secondary experiment was performed using the Embedded Trace Buffer (ETB), which is another component included in the CoreSight™ architecture. The trace stored in the ETB was further analyzed to reconstruct the processor execution status at the time an error was detected.

The remaining of this paper is as follows. Section II summarizes related work. Section III describes the proposed trace-based error detection approach. Section IV shows the experimental setup and the results using proton irradiation. Finally, section V presents the conclusions of this work.

II. Related work

Conventional fine-grain hardware redundancy techniques, such as Dual Modular Redundancy (DMR) or Triple Modular Redundancy (TMR) on flip-flops, are not suitable for COTS microprocessors because they require modification to the circuit. Thus, fault-tolerant architectures based on COTS microprocessors generally rely on software techniques, external hardware, or hybrid techniques that use a combination of both [12]. Microprocessor errors are usually divided into control-flow errors and data errors. Control-flow errors affect the instruction flow, provoking incorrect jumps. Data errors affect the results of the computations. Each of these types of errors are usually addressed with different error detection or correction techniques.

Software techniques are based on software modifications used to include redundancy in the operations performed by the microprocessor. They are quite convenient for microprocessors because they do not imply hardware modification. From this point of view, they can be considered more manageable than hardware modifications. Their common drawbacks are performance decrease and increased memory size. A large number of works proposing software techniques for both data and control-flow errors can be found in the literature. For data errors the most common and direct approach is data duplication [13], [14]. Data duplication techniques duplicate all data used by the microprocessor and check the consistency between the copies at several points of the executed program. Looking for a trade-off between performance and error coverage, the amount of duplicated data and checkpoints in the program may vary [16]. For control-flow errors, the most common software techniques are signature-based. These techniques assign a signature to each basic block of the executed code. Their goal is to validate the changes in the execution flow. Incorrect jumps in the execution flow can be detected by checking the signatures. Examples of this kind of techniques can be found in [15]. The most important drawbacks of signature-based techniques are that they generally require a large amount of memory to store the correct signatures in the system and signature computation and checking may introduce significant performance penalties.

All software techniques present a common problem due to the intrinsic characteristics of these techniques. Their error coverage is limited to the parts of the architecture that can be accessed from the programmer's side. With this approach, critical registers that are not accessible from software can be completely unprotected.

Alternatively, hardware techniques can be used. In the case of COTS that cannot be modified, hardware techniques are implemented by external hardware connected to the microprocessor. The connection point is an issue itself, and observed information and results can vary depending on its location. Existing techniques are limited by the accessible connections and the kind of information they can provide. Memories storing data or program are a common choice but it must be noted that information is read or written by the microprocessor but it is not processed in the memory. An instruction can be read from memory and checked by external hardware but the error may appear when it is being executed in the microprocessor. The same reasoning can be translated to data stored in memory. The complexity of the external hardware can vary from very small and simple circuits to sophisticated ones with a complexity similar to the microprocessor to be monitored.

A simple approach is to use the very same architecture to replicate the program execution and compare the results. This can be accomplished in several ways, using time redundancy or hardware redundancy. In the first case, the execution of the software is replicated and results are compared for error detection or correction. Error checking can be implemented in the software or in external hardware [4]. The latter option is more robust and contributes to reduce the performance penalties. Alternatively, two or more processors can be used to execute the same software in hardware redundancy. The increasing availability of multicore processors on a chip makes it an appealing approach. However, fault isolation is required to avoid that a fault in any single component leads to the failure of the entire chip [26]. External hardware modules are needed for error checking and management of the architecture. The DMT (Duplex Multiplexed in Time) and DT2 (Dual Duplex Tolerant to Transients) architectures developed by CNES [2] are good examples of the time redundancy and the hardware redundancy approaches, respectively.

Lockstep [17], [18] is a micro-synchronization variant of the processor replication techniques. Lockstep works by executing the same application simultaneously and symmetrically in two identical processors. In an error-free execution, both processors are expected to perform the same operations in every clock cycle. A hardware checker module monitors both processors to detect any discrepancy, in which case the processors are restored to a safe state through a rollback mechanism.

Except for pure software approaches, all fault-tolerant architectures based on COTS are generally hybrid, i.e., they require some software support (e.g., replication of the software or application of software-implemented fault tolerance techniques) as well as some hardware support to monitor the executions and check for errors. Hybrid approaches try to combine all the positive characteristics of both software and hardware techniques. Hardware techniques can help to reduce performance penalties while software techniques are better suited to deal with data errors. Examples of hybrid approaches can be found in [4], [5], [8], [9], [10], [11]. In these cases, the external hardware monitor is typically in charge of checking the instruction addresses to detect control-flow errors.

Modern microprocessors usually provide a trace interface so that software can be debugged during the design cycle. This part of the circuit is not used during normal operation, so that it can be reused for other tasks. The use of the trace interface to observe microprocessor behavior by connecting an external hardware monitor was first proposed in [6]. The trace interface can provide the flow of instructions executed by the processor in a non-intrusive manner. For instance, in [9] this approach was used to monitor a LEON3 microprocessor through its trace interface with good error detection rates. Recent work for ARM processors [19] has proposed also the use of the trace interface to collect trace data and then recreate the control-flow graph off-line.

To the best of our knowledge the trace information has been used so far for detecting control-flow errors [8-12]. However, a complete solution requires the detection of data-flow errors as well. To this purpose, these techniques are usually combined with software techniques for data-flow error detection [8-10]. In the approach proposed in this work, the trace interface is also used for on-line detection of data errors.

III. Trace-based Error Detection Approach

In this paper, a novel error detection technique for COTS microprocessors is presented. The proposed approach is based on retrieving control-flow and data-flow information from the processor and use it to determine if execution is correct or not. The proposed approach has been developed for an ARM CORTEX™-A9 microprocessor and a hardware monitor (PDTC) has been developed as an IP core to observe the microprocessor through its trace interface. The PDTC can be implemented in external hardware or in programmable logic. For convenience, a Zynq-7010 All Programmable SoC [20] was used as a test platform and the PDTC was implemented in the Programmable Logic (PL) of the device.

The CoreSight™ trace subsystem is provided along with the ARM cores. To setup the system, the necessary CoreSight components [21] must be configured and enabled in the microprocessor software initialization. CoreSight trace Subsystem and PDTC are connected through the Extended Multiplexed Input Output (EMIO) available on the Zynq SoC. The main elements involved in the system are displayed in Fig. 1. These elements are described in the following subsections.

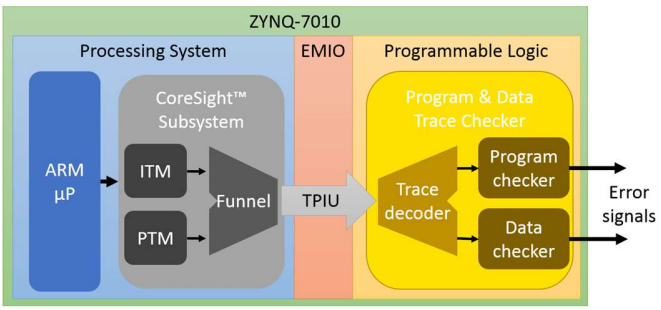

Fig. 1. General trace-based error detection architecture

A. CoreSight subsystem

In our case, we focus on two different CoreSight components: the Instrumentation Trace Macrocell (ITM) [21] and the Program Trace Macrocell (PTM) [22].

The ITM is a software-application driven trace source. It is a CoreSight component of the trace source class. The ITM produces various types of data packets of compressed information. The most interesting packet for our application is the SoftWare Instrumentation Trace (SWIT) packet, which exports through the trace interface any desired 32-bit data value related to the software. To trigger SWIT packet generation, software must write a value in any of the 32 stimulus ports of the ITM, which are mapped in memory as stimulus registers. When any of the stimulus registers is written, the ITM exports a SWIT packet indicating the number of the port (0-31) and the value of the written data in leading-zeroes compression.

The PTM is a real-time module that provides instruction tracing of a processor. It is a CoreSight component of the trace source class based on the ARM Program Flow Trace (PFT) architecture specification [22]. The PTM also generates trace information organized in packets. Among all PTM packets, we focus on the ones that contain Program Counter (PC) address information (I-sync, Branch Address and Waypoint Update).

A waypoint is a point where instruction execution by the processor might involve a change in the program flow. The PTM does not generate PC information unless a waypoint is reached. The PC address is presented in different formats depending on the packet: while I-sync packets always contain the full (32-bit) value of the Program Counter, the Branch Address and Waypoint Update packets are compressed and only contain the bits that have changed since the last PC address information. To maximize PC address available information from the trace port, the Branch Broadcasting option has been enabled on the PTM. With this option, the destination address for all branch instructions is included in the trace. When the waypoint does not provoke a change in the program flow (i.e. when a conditional branch is not taken), this situation is reported by an Atom packet.

The trace information produced by the macrocells is combined by the Funnel, which is a CoreSight component of the trace link class. Every input channel can be enabled or disabled through user programmable configuration registers, and priority can also be selected for each one.

Combined trace information is sent from the Trace Port Interface Unit (TPIU) to the PDTC through the EMIO interface. The TPIU is a CoreSight component of the trace sink class. To manage trace information coming from several sources, the Formatter [23] must be enabled on the TPIU. The Formatter rearranges trace information along with source IDs in a defined structure called frame. To increase flexibility, the TPIU includes FIFO queues and can output trace information synchronized with an independent clock. In our approach, the TPIU runs with the same clock as the PDTC. The TPIU has been configured with 8-bit wide data port in Normal operation mode.

B. Program & Data Trace Checker

The PDTC receives the trace information from the TPIU port and processes it on-line. It has been designed to reconstruct trace information from the trace frames and to identify all trace packets from both sources (ITM and PTM). First, the PDTC decodes relevant packets and extracts useful information about execution flow and software data values. Trace information is then directed to the Program Checker or the Data Checker according to its origin for the corresponding check to be performed. The PDTC is software-programmable via configuration registers, which can be accessed through the system bus (AXI) interface. In addition, its modular implementation enables flexibility and future scalability as more capabilities can be easily introduced without requiring modifications to the actual design.

The retrieved program-flow information processed by the Program Checker is related to PC addresses. This information is used to implement a PC follower capable of updating the last known address executed by the processor, including exceptions. The PC address is then compared with up to eight user-programmable address ranges to determine if execution has reached a forbidden or unexpected region. In such a case, an error signal is raised.

The software data values obtained from the trace interface are related to the state of selected variables during execution, so the Data Checker can determine if their values are valid or not, using two different techniques described below. To this purpose, software is instrumented to write stimulus registers in relevant points of the execution. In this work, we propose to arrange groups among the 32 available stimulus registers and assign each of the groups to different functionalities as it is represented in Fig. 2. Thus, when the PDTC receives a SWIT packet, it extracts the stimulus register number from it, and depending of that number, the required checking is executed using the respective value. If an inconsistency is detected, an error signal is raised.

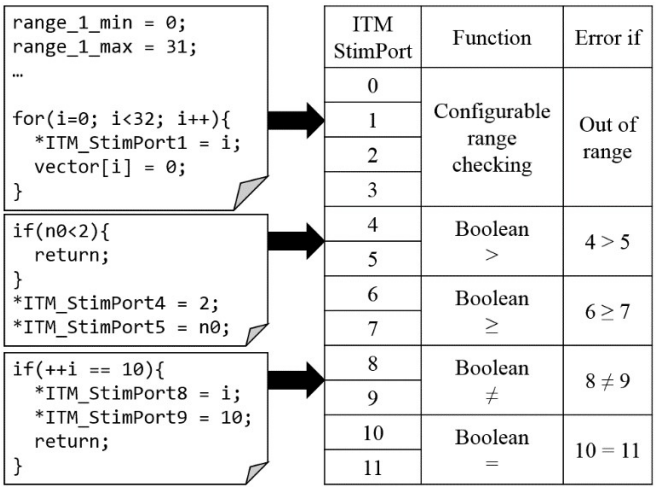

Fig. 2. Code instrumentation examples and Data Checker operation

Related to software data, two different error detection techniques have been developed: range checking and value comparison. Range checking determines if relevant values are within a specified range. Ranges are critical when running loops or indexing arrays. The Data Checker can be configured for up to four programmable ranges. Value comparison is important to ensure correct execution of branch conditions or to check data consistency. To this purpose, some stimulus registers have been grouped by pairs and each pair has been assigned to one boolean operation (equal, not equal, greater than, greater or equal than), so the Data Checker can determine whether the received values satisfy the selected condition.

Table I shows the synthesis results of the PDTC implemented in the programmable logic. The Checker requires a small amount of resources (4.8% and 3.2% of the available LUTs and flip-flops, respectively) and most of them are used for the Trace Decoder. In fact, the AXI interface, which is required to configure the Checker through the system bus, requires a similar amount of resources. Thus, the PDTC can be viewed as a fairly simple peripheral.

TABLE I
SYNTHESIS RESULTS

	# LUTs (% usage)	#FFs (% usage)
AXI Interface	425 (2.4%)	1073 (3.0%)
PDTC	836 (4.8%)	1109 (3.2%)
Total	1261 (7.2%)	2182 (6.2%)

With respect to performance, a major advantage of trace subsystems is that they are implemented as a side channel that does not interfere with the execution of the application. As a matter of fact, trace subsystems are intended to deal with asynchronous events which are difficult to reproduce and debug, so they are designed to provide reliable information with minimal intrusiveness. The PTM does not introduce any time overhead. Compared to other control-flow checking techniques, [27] shows up to 61% performance overhead using signature-based techniques and [4] shows up to 34% performance overhead using assertions. For data checking, the time overhead is proportional to the amount of stimulus register writes. The more stimulus register writes, the lower the error detection latency but the higher the performance overhead. This is a similar trade-off to the case of software implemented fault-tolerance. However, the performance penalty is reduced in the proposed approach because the checking is performed externally and the software only needs to report the values. It also benefits from existing hardware resources to collect trace data on a side channel without affecting the execution. Importantly, the use of the trace interface does not introduce any delay penalty, unlike other approaches that require access to critical interfaces such as the memory bus [2], [4], [5], [8], [17]. Eventually, the ITM can introduce a significant performance overhead if there are many consecutive stimulus register writes. In such a case, we have experimentally observed that the ITM might stall the processor in order to avoid losing information. This problem can be solved, if needed, by computing signatures or compressing the data to be reported

through the stimulus registers.

IV. Experimental results

A proton irradiation campaign was performed at CNA (Centro Nacional de Aceleradores) in Spain to validate the proposed approach. In the following subsections we describe the experimental setup and the radiation results. Finally, we describe how trace information can be analyzed using the proposed approach.

A. Experimental setup

For the radiation campaign we used an external beam line of the 18/9 IBA compact cyclotron. The Device Under Test (DUT) was irradiated in open air with 15 MeV protons. The energy of incident protons in the silicon active area is in the order of 10 MeV. According to previous experiments [25], this energy is sufficient to produce SEEs in the 28 nm technology device without thinning it.

We selected a Xilinx Zynq-7010 All Programmable SoC device for the experiments [20]. Zynq devices integrate hard-core ARM CORTEX™-A9 processors with SRAM-based FPGA. Our experiments were conducted over a basic commercial board (Zybo) with a XC7Z010 device that contains a dual core of ARM CORTEX™-A9. For the experiments only one of the cores was used at the nominal 650 MHz clock frequency. The PDTC was implemented in the Programmable Logic (PL) of the device. The PDTC can only detect errors in the ARM cores. To correct errors in the configuration memory of the PL that may affect the PDTC, we have used the Xilinx Soft Error Mitigation (SEM) Controller IP [24].

We used an SD card to store the boot code, the bitstream and the application software. Upon power up, this information is loaded to the On-Chip Memory (OCM) of the microprocessor to configure the device and start operation.

In order to control the DUT that is exposed to the beam, an external host has been connected to the Zybo board. The external host is in charge of the control of the DUT and the retrieval of information during the experiments. To this purpose, it is connected to the ARM core through a USB interface and to the error signals provided by the PDTC through dedicated pins. When an error is detected, the external host switches off the power of the device and then restarts it again. The system is also restarted in some other cases to ensure the experimental results are fair, as follows. The external host retrieves information from the SEM about errors in the programmable logic. When an unrecoverable error is detected by the SEM, the external host restarts the device. If the communication between the processor and the external host is experiencing a malfunctioning and the host receives corrupted data, the system is restarted as well. Unrecoverable SEM errors and communication errors are not taken into account in the results of the experiments reported in the next section.

The PDTC is able to detect control-flow and data errors thanks to the information provided by the trace subsystem. In order to verify the correctness of the results obtained with the PDTC during the experiments, both error types were double checked by additional means. First, the external host controls the time required for the application and triggers a timeout error when it is exceeded. Second, data checks were implemented in the software as well. Every variable was duplicated and every operation was repeated for the duplicated variable. Consistency checks were implemented immediately after every variable and its copy were updated. At the end of every execution, the results were also compared with a golden reference.

The Zybo board was partially covered, leaving only the DUT exposed to the beam. The external host was placed outside the beam.

B. Radiation results

The experiments were performed with three different software benchmarks: matrix multiplication (MMULT), Advanced Encryption Standard (AES) and a recursive implementation of the sorting algorithm quicksort (QSORT). The benchmarks were run on bare metal, but they could also run in principle on an operating system because the configuration and instrumentation is made at high level. All benchmarks implemented duplicated variables and consistency checks in the software according to the approach described in the previous section. Data values were sent through the ITM ports to be checked by the PDTC. In the current implementation, the checks were configured by hand, but an automatic tool is feasible. Thanks to the combination of error detection in the software and external monitoring we were able to double check all the observed errors reported by the PDTC.

MMULT benchmark is characterized by intensive data-flow operations (multiply-accumulation) and few control decisions. For the experiments we used matrices of 32x32 data size. AES is an encryption algorithm which is characterized by intensive shifting and logical operations. It makes a high usage of XOR operations. The tested implementation uses a key length of 256 bits and 10 iterations.

QSORT is a sorting algorithm which is characterized by intensive use of control decisions and few computational operations. This algorithm could be considered as opposite to MMULT with respect to the structure of the code and the type of executed instructions, because it has a more complex control-flow and uses simpler operations. Our implementation of this algorithm was done in a recursive way to test also how the proposed approach works with intensive function calls. For QSORT algorithm we used vectors of 500 elements.

The performance overhead caused by stimulus register writes depends on the ratio of reported data to the instructions required to compute them. We instrumented the benchmarks to report all computed data and indices of the loops as they are being produced. With this approach, the overhead was 11% for MMULT, 20% for AES, including intermediate results after each iteration, and 51% for QSORT. The latter has the worst ratio between reported data and instructions. For the radiation experiments we included additional data register writes to increase observability of intermediate operations.

All benchmarks were compiled with SDK Xilinx tool and minimum optimization effort (-O0) to preserve data-flow duplication. Table II shows the results of the radiation campaign for all benchmarks. Columns of Table II report the

experimental results for each benchmark, namely MMULT, AES and QSORT. Every row reports the observed errors for a particular error category for all benchmarks. Errors are reported in number and percentage with respect to the total number of observed errors for each benchmark.

The first three rows of Table II report the errors detected by the PDTC, divided in three subcategories (Det. TO, Det. Data and Det. OP). The fourth and fifth rows report the errors undetected by the PDTC, divided in two subcategories (Undet. TO and Undet. Data). And finally the last two rows provide the total number of errors (TOTAL ERRORS) and the total number of errors detected by the PDTC (TOTAL DET).

The error categories used in Table II are defined as follows:

- Det. TO (Detected Timeout error): The PDTC detects an error that is confirmed by a host timeout error.
- Det. Data (Detected Data error): The PDTC detected a data error which is also detected by a software check (discrepancy in duplicated data or in the final result).
- Det. OP (Error detected only by the PDTC): the PDTC raises an error that is not detected by the software checking or the host.
- Undet. TO (Undetected Timeout error): the host detects a timeout condition but the PDTC does not raise an error.
- Undet. Data (Undetected Data error): Data errors that are only detected by software checks.

TABLE II
RADIATION RESULTS: OBSERVED ERRORS

ERRORS	MMULT	AES	QSORT
Det. TO	61 (19.06%)	51 (17.41%)	81 (32.27%)
Det. Data	234 (73.13%)	189 (64.51%)	111 (44.22%)
Det. OP	11 (3.44%)	25 (8.53%)	23 (9.16%)
Undet. TO	7 (2.19%)	22 (7.51%)	24 (9.56%)
Undet. Data	7 (2.19%)	6 (2.05%)	12 (4.78%)
TOTAL ERRORS	320 (100%)	293 (100%)	251 (100%)
TOTAL DET	306 (95.63%)	265 (90.44%)	215 (85.66%)

In the experiments we observed a total of 320 errors for MMULT benchmark, 293 for AES benchmark and 251 for QSORT benchmark. The detection capability of the PDTC varies from 85.66% of detected errors for QSORT to 95.63% for MMULT. The latter result is in line with that reported in [28] for a hybrid approach with a simpler soft core processor (miniMIPS) under neutron radiation. The variations in the error detection capabilities of the PDTC with respect to the different benchmarks are related with the characteristics of the codes. The worst case results are obtained for QSORT, which is the benchmark that has a more complex control flow. Moreover, this benchmark uses recursiveness, which makes the stack pointer a very critical register in this case. However, the stack pointer was not checked by the PDTC in the used implementation.

A few errors were only detected by the PDTC (Det. OP). The causes of these errors may vary. They may be temporary errors that are eventually corrected or errors that remain latent and may cause a malfunction later on. They may also be errors in the Coresight trace subsystem or in the PDTC. It must be noted that the trace subsystem could not be protected because it is part of the microprocessor. The PDTC was partially protected by the SEM. Nevertheless, the amount of errors of Det.OP category is small, ranging from 3.5% to 9.1%. In any case, it is generally advisable to consider them as errors and restart the system in these cases. For the experiments we have considered Det. OP errors as real errors and they have triggered a system restart.

The latency of error detection is very small because the trace information is transmitted as a data stream that is directly captured by the PDTC from the TPIU. As a matter of fact, we have experimentally realized that the error signals provided by the PDTC are generally raised before the error is confirmed by the microprocessor. We have estimated the latency by artificially forcing an error and measuring the time until the processor catches an interrupt produced by the PDTC error signals with a timer. The average measured latency was 225 processor clock cycles (345 ns at the processor clock frequency of 650 MHz). This includes the time used for the Coresight subsystem to encode and transfer the corresponding data packet, and for the PDTC to decode the packet, detect the error and signal it. Note that the TPIU and the PDTC run at a lower clock frequency (150 MHz) than the processor and the TPIU port was configured for a small data width (8 bits). In the current implementation, the PDTC uses 153 ns (23 clock cycles at 150 MHz clock frequency) to detect and signal an error, which is about 45% of the total latency. These choices can be optimized to reduce the error detection latency. It must be noted that previous works using on-line control-flow checking from an external hardware monitor were developed for soft cores. In these cases, the error detection latency is not reported but it is presumably minimal, because the external hardware monitor is connected to the memory bus or to a raw trace interface. However, in a hard core processor, neither of these interfaces is generally available. Nevertheless, the proposed approach can achieve an acceptable error detection latency using the built-in trace subsystem.

Table III shows the fluence and the cross-section for all benchmarks. The cross-section reported in the second row (Cross-section, All errors) takes into account all the observed errors. In the third row (Cross-section, Undetected errors), the reported cross-section has been computed taking only into account the errors that were undetected by the PDTC. The results reported in Table III show the high error detection capabilities of the proposed approach, with a reduction in cross-section up to two orders of magnitude when the PDTC is used. This is a remarkable result because the PDTC is only using some basic features and there is room for improvement with more elaborated checks of the trace information.

TABLE III
RADIATION RESULTS: CROSS SECTION

	MMULT	AES	QSORT
Fluence (p/cm^2)	$1.6 \cdot 10^{12}$	$3 \cdot 10^{12}$	$4.1 \cdot 10^{12}$
Cross-section (cm^2) All errors	$2 \cdot 10^{-10}$	$9.77 \cdot 10^{-11}$	$6.12 \cdot 10^{-11}$
Cross-section (cm^2) Undetected errors	$8.75 \cdot 10^{-12}$	$9.33 \cdot 10^{-12}$	$8.78 \cdot 10^{-12}$

C. *Trace information analysis*

As a proof of concept, a secondary experiment was performed with the same DUT and software benchmarks. Its main purpose is to test the diagnosis capabilities of the trace information. It has been performed preventing any disturbance to the primary experiment about detection capabilities under the radiation conditions explained before. To enable this experiment, the Embedded Trace Buffer (ETB) [21] was used to store and retrieve the trace information.

The ETB is a CoreSight component of the trace sink class. It is provided within the processing system of the Zynq SoC and it is internally connected to the same trace bus than the TPIU, so it receives the very same information that is processed by the PDTC. The ETB has been enabled and its Formatter has also been configured with the same parameters as the TPIU Formatter to produce the same data. The ETB contents can be accessed through the AXI interface using memory mapped registers available from the software application. The ETB buffer size is 4 kilobytes.

The ETB is continuously storing the trace interface in a circular manner, so that, at any time, it contains the most recent trace information. Taking advantage of the detection capabilities of the PDTC, trace capture can be disabled just at the time an error is detected, so a snapshot of the trace remains in the buffer and can be accessed later, either on-line or off-line. In our experiment, the buffer was recorded in the external host for off-line analysis.

When an error is detected, the software first checks for data consistency. Then, just before the system is rebooted, the software enters a function that reads the ETB and sends its contents through a serial port. This is what we have called a trace dump. The external host receives the trace dump and stores it in independent files.

To analyze the trace information, our approach is to simulate the PDTC for the collected trace dumps. This way, we can know exactly how the PDTC has processed the trace information. This simulation approach has also been helpful to identify PDTC design errors in the initial versions of its development. A test bench has been designed to get the data directly from the trace dump files and use it as input stimuli for the PDTC in VHDL simulation, so the evolution of the internal signals can be easily tracked. This approach does not require complete trace information since the beginning of the software execution because the PDTC can get synchronized using synchronization packets. The only requirement is to have enough trace information prior to the error to let the PDTC synchronize before the error appears.

Using the technique explained above, it is possible to reconstruct the processor execution status in the moment of an error. The available information for this analysis can be as rich as needed, since the software can be extensively instrumented by adding instructions to export any variable value in any point of execution. The combination of variable instrumentation with program counter information enables promising novel diagnosis capabilities to evaluate circuit reliability and the effectiveness of fault-tolerance techniques. In addition, obtained diagnostics could be applied to the development of more complex, new detection techniques.

Although the presented work is a first approach, and major improvements can still be done, it has been possible to experimentally confirm the trace information capabilities to reconstruct the execution status. In particular, for the detected timeout errors (Det. TO in Table II), it was verified that an out-of-range PC address was found on the program trace. A wrong PC address can provoke a jump to an invalid code region that causes the processor to hang. The PC addresses can be checked with the original program to locate the wrong instruction that was being executed when the error happened. A similar analysis was performed for data errors to verify that there was incorrect data on the data trace. In all the cases that a trace dump related to a detected error has been analyzed, the information contained in the trace dump confirmed the detected error.

We could also analyze the errors detected by the PDTC that were not detected by the software checks (Det. OP in Table II). These errors can have a variety of sources. They may correspond to benign execution errors, i.e., errors that do not cause malfunctioning, but also to errors inside the Coresight trace subsystem or the PDTC. However, with the current implementation it is not generally possible to know exactly where the error occurred. It must be noted that the Coresight trace subsystem is susceptible to errors and it cannot be hardened as we used a commercial device. The PDTC can be hardened, but the benefits would be marginal due to its relatively small size in comparison with the Coresight trace subsystem. The PDTC used in the experiments was implemented in the PL and we used the SEM IP to correct configuration memory errors. If the collected trace dump contains an error that is verified by simulating it with the PDTC, it may be a benign error or an error in the Coresight subsystem. If the collected trace dump does not trigger the PDTC error detection, it may be an error in the PDTC or an error at the TPIU. Both cases were observed in the collected trace dumps. However, the amount of trace dumps that we were able to collect for this type of errors was small and additional experiments are required in order to reach to a conclusion about the relative sensitivity of each component in the trace processing chain.

V. CONCLUSIONS

In this work we have proposed a new approach for error

detection in ARM microprocessors based on available on-chip trace infrastructures. A hardware module is used to monitor both the program-flow trace and a highly configurable data trace. This approach can be extended to other COTS microprocessors that support program and data tracing.

Experimental results demonstrate that the proposed approach has a high error detection capability, even though only some basic checks were implemented. Control-flow errors are detected in a non-intrusive manner and with no performance penalty. Data can be reported for checking through the trace interface, at the expense of a time overhead that is proportional to the amount of stimulus register writes. Error detection latency is small, despite the fact that trace information has to traverse the trace subsystem. In summary, the trace interface is a viable means to implement error detection in COTS microprocessors. Additionally, traces can be collected for further analysis and diagnosis of the cause of errors.

This work has shown that the checking of trace information has a great potential to detect and diagnose radiation induced errors in complex microprocessors. Future work is oriented to improve the error detection rate, by making a more elaborated use of the rich information that the trace subsystem can provide, and to improve the diagnosis capabilities.

REFERENCES

[1] R. Ginosar, “Survey of processors for space”, Proc. Int. Space System Engineering Conf. (DASIA), 1B, 2012.

[2] M. Pignol, “DMT and DT2: Two Fault-Tolerant Architectures developed by CNES for COTS-based Spacecraft Supercomputers”, Proc. 12th Int. On-Line Testing Symposium (IOLTS), pp. 203-212, 2006.

[3] K. A. LaBel, “NEPP Roadmaps, COTS, and Small Missions”, Presented at NEPP Electronics Technology Workshop (ETW), Goddard Space Flight Center, June, 2016.

[4] J. R. Azambuja, M. Altieri, J. Becker, and F. Lima Kastensmidt, “HETA: Hybrid error-detection technique using assertions,” IEEE Transactions on Nuclear Science, vol. 60, no. 4, pp. 2805–2812, Aug. 2013.

[5] J. R. Azambuja, S. Pagliarini, M. Altieri, F. Lima Kastensmidt, M. J. Becker, G. Foucard, and 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.

[6] 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, pp. 167-172, 2010.

[7] 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.

[8] 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.

[9] L. Parra, A. Lindoso, M. Portela-Garcia, L. Entrena, B. Du, M. Sonza Reorda, L. Sterpone, "A New Hybrid Nonintrusive Error-Detection Technique Using Dual Control-Flow Monitoring", IEEE Transactions on Nuclear Science, vol. 61, no. 6, pp. 3236-3243, Dec. 2014.

[10] A. Lindoso, L. Entrena, M. García-Valderas, L. Parra, "A hybrid fault-tolerant LEON3 soft core processor implemented in low-end SRAM FPGA", IEEE Transactions on Nuclear Science, vol. 64, no. 1, pp. 374-381, Jan. 2017.

[11] B. Du, M. Sonza Reorda, L. Sterpone, L. Parra, M. Portela-García, A. Lindoso, L. Entrena, "On-line Test of Control Flow Errors: A new Debug Interface-based approach", IEEE Transactions on Computers, vol. 65, no. 6, pp. 1846-1855, Jun. 2016.

[12] L. Entrena, A. Lindoso, M. Portela-Garcia, L. Parra, B. Du, M. Sonza-Reorda, L. Sterpone, “Fault-tolerance techniques for soft-core processors using the Trace Interface”, In “FPGAs and Parallel Architectures for Aerospace Applications. Soft Errors and Fault-Tolerant Design”, F. Kastensmidt, P. Rech, Paolo (Eds.), Springer Switzerland, 2016.

[13] 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, pp. 2231–2236, Dec. 2000.

[14] B. Nicolescu, R. Velazco, “Detecting soft errors by a purely software approach: method, tools and experimental results”, Design, Automation and Test in Europe (DATE) Conf., pp. 57 – 62, March 2003.

[15] M. Rebaudengo, M. Sonza Reorda, and M. Violante, “Software level soft-error mitigation techniques,” in “Soft Errors in Modern Electronic Systems”, M. Nicolaidis (Ed.), New York, NY, USA, Springer, 2011.

[16] E. Chielle, J. R. Azambuja, R. S. Barth, F. Almeida, F. L. Kastensmidt, “Evaluating Selective Redundancy in Data-Flow Software-Based Techniques”, IEEE Transactions on Nuclear Science, vol. 60, no. 4, pp. 2768-2775, Aug. 2013.

[17] F. Abate, L. Sterpone, M. Violante, “A New Mitigation Approach for Soft Errors in Embedded Processors”, IEEE Transactions on Nuclear Science, vol. 55, no. 4, pp. 2063-2069, Aug. 2008.

[18] A. B. de Oliveira, G. S. Rodrigues, F. L. Kastensmidt, "Analyzing lockstep dual-core ARM cortex-A9 soft error mitigation in FreeRTOS applications," Proc. 30th Symp. on Integrated Circuits and Systems Design (SBCCI), pp. 84-89, 2017.

[19] A. W. Hoppe, F. L. Kastensmidt, J. Becker, “Control Flow Analysis for Embedded Multi-core Hybrid Systems”, In “Applied Reconfigurable Computing. Architectures, Tools, and Applications. ARC 2018”, N. Voros, M. Huebner, G. Keramidas, D. Goehringer, C. Antonopoulos, P. Diniz (Eds.), Lecture Notes in Computer Science, vol 10824, pp. 485–496, Springer Cham, 2018.

[20] ”Zynq-7000 All Programmable SoC: Technical Reference Manual”, Xilinx Inc., Technical Ref. Manual UG585, Sept. 2016.

[21] “CoreSight Components. Technical Reference Manual”, ARM Ltd., DDI 0314H, 2009.

[22] “CoreSight Program Flow Trace. Architecture Specification”, ARM Ltd., IHI 0035B, 2011.

[23] “CoreSight Architecture Specification v2.0”, ARM Ltd., IHI 0029D, 2013.

[24] “Soft error mitigation controller v4.1 Product guide,” Xilinx Inc., White Paper PG036, Nov. 2014.

[25] A. Lindoso, M. García-Valderas, L. Entrena, Y. Morilla and P. Martín-Holgado, "Evaluation of the Suitability of NEON SIMD Microprocessor Extensions Under Proton Irradiation," IEEE Transactions on Nuclear Science, vol. 65, no. 8, pp. 1835-1842, Aug. 2018.

[26] N. Aggarwal, P. Ranganathan, N. P. Jouppi, J. E. Smith. “Benefits from Isolation in Commodity Multicore Processors”. IEEE Computer, vol. 40, no. 6, pp. 49-59, June 2007.

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

[28] J. R. Azambuja et al., "Evaluating Neutron Induced SEE in SRAM-Based FPGA Protected by Hardware- and Software-Based Fault Tolerant Techniques," IEEE Transactions on Nuclear Science, vol. 60, no. 6, pp. 4243-4250, Dec. 2013.