uc3m | Universidad Carlos III de Madrid

e-Archivo

This is a postprint version of the following published document:

Pena-Fernandez, M., Lindoso, A., Entrena, L. & Garcia-Valderas, M. (2020). The Use of Microprocessor Trace Infrastructures for Radiation-Induced Fault Diagnosis. *IEEE Transactions on Nuclear Science*, 67(1), pp. 126–134.

DOI: 10.1109/tns.2019.2956204

© 2020, IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

The Use of Microprocessor Trace Infrastructures for Radiation-Induced Fault Diagnosis

M. Peña-Fernandez, A. Lindoso, L. Entrena, M. Garcia-Valderas

Abstract— **This work proposes a methodology to diagnose radiation-induced faults in a microprocessor using the hardware trace infrastructure. The diagnosis capabilities of this approach are demonstrated for an ARM microprocessor under neutron and proton irradiation campaigns. Experimental results demonstrate that the execution status in the precise moment that the error occurred can be reconstructed, so that error diagnosis can be achieved.**

Index Terms— **ARM, microprocessor trace, single event effects, fault tolerance, fault diagnosis.**

I. Introduction

THE increasing use of electronic systems in safety-critical and mission-critical applications has raised the concern about radiation effects. Fault tolerance and radiation hardening techniques are applied to cope with these effects, and radiation testing is then used to validate the error rates. However, radiation testing does not generally provide information about the causes of errors and the vulnerabilities of circuits. Gaining insight into the sources of errors is crucial to understand radiation effects and eventually protect the circuits in an effective manner. Fault diagnosis attempts to provide knowledge about the cause of errors that may enable the identification of the most critical parts of circuits and specific countermeasures to be taken to mitigate errors closer to their origin or before they provoke a catastrophic failure. Error diagnosis can also help system housekeeping. For safety-critical applications, it is important to know if an error is tolerable, if the system can still partially operate or if the system must be put out of service immediately. The case of microprocessors is particularly relevant, because they are ubiquitous and, at the same time, they show complex and varied failure modes that need to be tackled in different ways. Analyzing the vulnerabilities of microprocessor-based applications to radiation effects is a difficult task that has long been attempted by researchers [1-6].

In radiation testing, errors are typically detected by comparing the output of a circuit with a known correct output. The goal of fault diagnosis is to determine when and where the errors were originated. One common approach is to force faults at known times and locations, so that the observed fault effects can be easily associated to each fault. Focused laser [7] and micro-beam [8], [9] testing can be used for this purpose. However, the results produced by these techniques may be biased, because faults are not randomly produced. Besides, the beam spot and step are generally larger than the current transistor dimensions [7], [9]. Alternatively, fault injection has been used to analyze the vulnerability of the internal structures of a microprocessor [10]. However, fault injection does not accurately emulate real radiation-induced faults. In addition, all these approaches require an in-depth knowledge about the internal structure of the circuit under test, in order to focus the beam or to inject faults in particular locations. Finally, another approach consists in testing the processor while running a specific code intended to facilitate diagnosis or to perform self-inspection. For instance, in [16] a PowerPC750 microprocessor is tested with a "do nothing" loop, to reduce the number of susceptible locations, and a "pin wiggler" method that inspects the internal registers and toggles a pin if an error is found. Obviously, this approach cannot be used to diagnose faults in a real application.

The objective of this work is to provide a means to diagnose the faults that may happen in a microprocessor running a real application under radiation testing. To this purpose, we propose the use of microprocessor trace infrastructures. Trace infrastructures are commonly found in modern microprocessors to support software debugging. Among the different utilities for debugging, tracing is particularly useful to deal with asynchronous events, such as interrupts. Trace infrastructures have been used in the past for error detection [17]. In this work, we focus on microprocessor fault diagnosis. After an error is detected, using the approach proposed in [17], trace information and context data are collected and analyzed to diagnose the cause of the error.

Software debugging is typically performed through breakpoints. When a breakpoint is triggered, the processor jumps to a particular application that supports user interaction. From this application, the user can get the contents of registers and memories to check the execution. This approach requires

This work has been supported in part by project ESP-2015-68245-C4-1-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).

knowing the control flow of the application in order to set appropriate breakpoints.

Hardware trace is provided to deal with asynchronous events, because they are difficult to reproduce and may make the processor lose control. A hardware trace unit provides a log of the instructions executed by the processor as well as information about context switching and interrupts. It can also log selected data. This information is continuously and automatically logged in a circular buffer, so that the most recent history of the processor can be retrieved at any time, particularly when the processor crashes. The trace data let the user reconstruct the program flow and the status of the processor when an event happened. Contrary to debugging, which is an intrusive operation, hardware tracing is implemented by the processor without interfering with normal execution. As radiation effects are asynchronous by nature, hardware tracing is well-suited to collect information that can be analyzed for microprocessor fault diagnosis.

Hardware trace infrastructure and protocols are microprocessor-specific. However, most microprocessors today support tracing and provide a trace interface to access the trace information. Without loss of generality, the proposed approach has been developed and tested for an ARM Cortex-A9 microprocessor. To evaluate the diagnosis capabilities of the proposed approach, a first experiment was performed with neutrons at LANSCE. Then, a second experiment with low-energy protons was performed at CNA (Centro Nacional de Aceleradores) in Spain. The results of these experiments demonstrate that trace dump information can be used to reconstruct execution status in the precise moment that the error occurred, so that error diagnosis can be achieved. The proposed approach can also be used online to diagnose errors and implement tailored recovery actions for each type of error while in operation.

The remaining of this paper is as follows. Section II introduces the problem of microprocessor fault diagnosis and summarizes related work. Section III describes the proposed trace-based fault diagnosis approach. Section IV discusses the experimental results. Finally, section V presents the conclusions of this work.

II. Microprocessor Fault Diagnosis

Analyzing the causes of errors and the vulnerabilities in microprocessors working in harsh environments is a difficult problem, mainly due to the limited observability of the internal state of the processor during a test. Most existing approaches are based on evaluating the effects of many faults and then use this information to perform some sort of cause-effect analysis.

Systematic approaches for architectural vulnerability analysis are generally based on fault injection [10]. The goal of this analysis is to compute the Architectural Vulnerability Factor (AVF) of each internal structure, which expresses the probability that a visible system error will occur given a bit flip in a storage cell. To this purpose, a large fault injection campaign is performed and the effects of injected faults are classified. However, note that AVF is a conditional probability and it does not include the probability that a bit flip occurs. Despite this issue, fault injection results can be used to predict a cross-section by calculating the product of the static cross-section and the error rate obtained by fault injection [11], [12]. Nevertheless, cause-effect analysis is difficult because many faults produce similar effects that cannot be easily distinguished. Some common effects, such as a processor crash, may have very different causes.

In [5] and [6], fault injection was used to induce errors. The erroneous outputs were collected in a fault dictionary, which were later used to try to diagnose errors during radiation testing by correlating the responses of the circuit with those stored in the dictionary. However, the diagnosis capabilities of this method are limited by the aliasing of the responses, because there can be many faults that produce the same response. Artificial intelligence methods have been proposed to improve cause-effect analysis, such as expert systems, neural networks and fuzzy logic. A brief review of these methods is provided in [15].

In [13] an analytical model for the vulnerability of L2 caches is proposed. This model was later validated with data collected from error logs of information systems present globally in the field [14].

In general, fault diagnosis is highly dependent upon the quality and completeness of the information that can be collected when an event happens. It is also of the outmost importance for an accurate diagnosis that this information is collected immediately after the event. A hardware trace unit can collect enough information to reconstruct the program flow and the status of the processor in a non-intrusive manner. The proposed fault diagnosis approach is described in the following section.

III. Fault Diagnosis Approach

The proposed fault diagnosis approach has been developed and tested for the ARM Cortex-A9 microprocessor, based on the ARM’s CoreSight trace infrastructure [18]. CoreSight includes a wide set of modules developed by ARM to support debugging and tracing in high-complexity SoC designs. The CoreSight architecture is modular, flexible and compatible with almost every microprocessor developed by ARM. CoreSight components can be classified into three main groups: trace source, trace link and trace sink. Trace information is originated in the processor and collected by a trace source component, which generates normalized packets containing trace information. Trace link components are in charge of transmitting the packets. They can also include additional information, typically related to the source that generates each packet. Trace packets are received by trace sink components, which enable trace information to be accessed from outside CoreSight.

The device under test was a Z7010 SoC, belonging to the ZYNQ-7000 APSoC family supplied by Xilinx. This system includes a Processing System (PS) featuring a dual-core ARM Cortex-A9 processor along with Programmable Logic (PL). Signals from PS and PL can be connected to external pins through the Multiplexed Input Output (MIO) interface. It is also possible to route signals from PS to PL and vice versa using the

Extended Multiplexed Input Output (EMIO) interface. The CoreSight subsystem included in the PS of the Z7010 SoC contains several components. Among them, two trace sources have been used in this work, as they provide information that can be useful regarding error diagnosis: the Program Trace Macrocell (PTM) and the Instrumentation Trace Macrocell (ITM).

The PTM [19] traces instructions executed by the processor in real time. It identifies certain instructions and events inside the processor as waypoints. A waypoint is defined to be any point in execution where an instruction can make a change in the sequential program flow. The PTM generates packets containing information related to execution flow, and others containing auxiliary information as synchronization or context. Three packet types have been used for diagnosis purposes as they contain PC (Program Counter) address information. These packet types are I-Sync, Branch Address and Waypoint Update. While the I-Sync and Branch Address packets contain information of waypoints that can be directly inferred from executed code, the Waypoint Update packet is used to upgrade any instruction to the range of a waypoint when an asynchronous branch occurs, such as an interrupt or an exception. In such a case, the program execution flow could not be reconstructed just inferring it from the code. To handle this situation, the address contained in the Waypoint Update packet represents the last instruction successfully executed before the branch was taken, allowing to reconstruct the program execution flow in a correct manner. To maximize the available PC address information, the PTM has been configured with option Branch Broadcasting enabled.

The ITM [18] generates trace information related to program data. Among all ITM packet types, the SWIT (SoftWare Instrumentation Trace) packet is the most interesting one for diagnosis purposes. This packet can record the value of any 8-bit, 16-bit or 32-bit program data. To this end, software must be instrumented, and new instructions must be added to write the values of interest in one of the 32 stimulus ports available in a region of memory reserved for this purpose. When a new value is written to a stimulus port, a SWIT packet is generated, containing both the data value, up to 32 bits, and the stimulus port number.

Information produced by both the PTM and the ITM is transmitted through the Funnel and the Replicator to reach the Trace Port Interface Unit (TPIU) and the Embedded Trace Buffer (ETB), which are CoreSight trace sinks. The ETB is a 4kB circular buffer which always stores the most recent trace information. The TPIU is a standard port which outputs trace information. In the current implementation, it has been configured in normal mode with 8-bit port data width and has been driven to the PL through EMIO.

A custom IP module, called Program & Data Trace Checker (PDTC), was implemented in the PL to decode and check the trace packets. The error detection capabilities of the PDTC have been shown in a previous work [17]. The PDTC uses the instruction trace provided by the PTM to detect control-flow errors. To this purpose, it tracks the addresses of the instructions executed by the processor and raises an error signal if execution reaches a forbidden or unexpected region. For data errors, the PDTC checks the range and the values of selected variables provided by the ITM. The trace is also stored in a custom buffer of 2kB implemented in the PL. Because of the low latency error detection of the PDTC, it is possible to freeze trace acquisition on both buffers and capture a snapshot of the trace when an error occurs.

The trace information can be downloaded from the buffers for offline or online analysis. For testing purposes we designed a utility software to read the PDTC status and the contents of the buffers, and send all this information through the serial port, which is called a trace dump. The trace dump can be obtained for every error detected by the PDTC. Once received, the trace dump is timestamped and stored by an external host to enable further analysis. After collecting the trace dump, the external host power cycles the DUT to capture new errors.

Trace information is almost limited to branch and exception PC addresses, CPU context and data values. The PDTC can detect errors and activate error signals with very low latency. These signals can optionally be used to force an exception on the processor to download any further information about its internal status after the error has been detected by the PDTC and before the power cycle.

The system architecture is summarized in Fig. 1, where black arrows represent trace data, green arrows represent the AXI interface and the brown arrow represents a serial port. Dotted arrows represent PDTC error signals, which are connected to several points in the system: 1) to the custom buffer to freeze trace acquisition; 2) to the processor to trigger the exception that forces downloading of the trace dump; and 3) to the external host to register it.

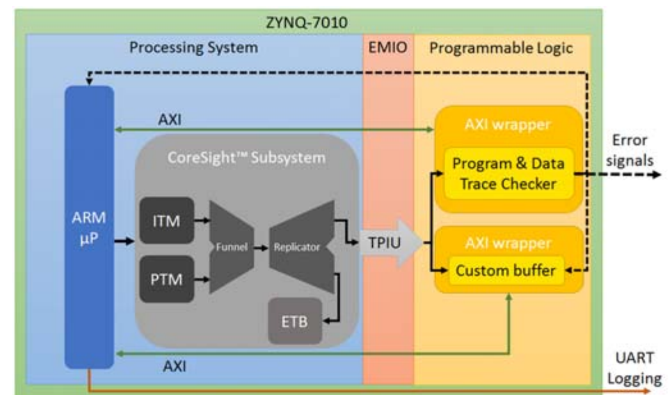

Fig. 1. Trace generation and processing architecture

A common fault effect is that the processor becomes unresponsive, which is often called a hang. There are many locations that can cause this effect when struck by a particle. In the ARM architecture, one of the most common causes is an attempt to access to a protected memory position. In such a case, the processor raises an exception and the control is transferred to the associated interrupt handler. These exceptions are catched by PDTC as a control-flow error. However, unless an appropriate interrupt handler is provided, the default implementation of each interrupt handler basically ends up in

an infinite loop and the processor hangs. To make sure the trace dump is properly collected in these cases, we designed a generic interrupt handler that is associated to all unexpected exceptions. This interrupt handler dumps the trace to the host before the processor is reset. In addition, we also took care to configure all unused memory areas as protected areas in the MMU (Memory Management Unit), so that we can catch any attempt to access to these areas.

A. Trace analysis

The recorded trace dump contains relevant information about the history of the processor at the moment of the error. A simple way to analyze this information, and exactly determine the trace value that triggered the error to diagnose it, consists in simulating the PDTC with the trace dump. Alternatively, the diagnosis could be performed online in the PDTC, but that would increase its complexity. A simulation test bench has been designed to gather PDTC configuration and trace information from the trace dumps and use it as input stimuli for the PDTC as shown in Fig. 2. A script-based tool has been developed to automate the simulations for each trace dump and assert conditions have been added to automatically obtain the report of each simulation.

For every trace dump, it is assumed that the PDTC must detect an error in the last clock cycles. When a program flow error is detected, the address of the last correctly executed instruction can be identified, and also the previous ones. When a data error is detected, the faulty data value can be obtained. Wrong data can be identified by comparing it with redundant data, if the code contains redundant data, or by reproducing the execution. Having two different buffers is very useful to further investigate whether the stored information is valid or not. If simulation with ETB ends without errors, but simulation with custom buffer results with an error, it means that the error occurred in a point between the ETB and the PDTC, probably in the TPIU, provoking an erroneous trace information that has been captured by the custom buffer and the PDTC. If neither the ETB nor the custom buffer simulations present errors, then the error must have occurred inside the PDTC.

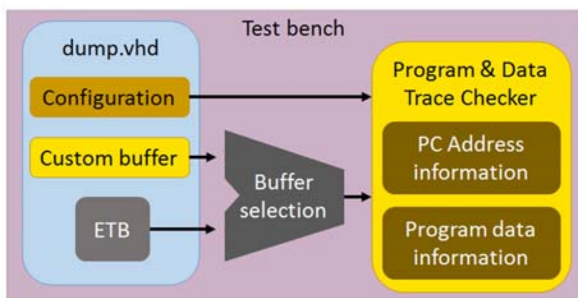

Fig. 2. Trace simulation test bench

B. Memory and stack analysis

Trace dump analysis determines the reason why the PDTC raised the error signal but may not represent the exact source of the error. To improve the diagnosis capabilities, additional information can be downloaded when an error is detected. In particular, the following information has been selected to be downloaded by the system in the case of an error:

- Stack dump. When an exception occurs, register contents in the moment of the exception are saved in the stack. Then, the exception handler has been provided with a function to read the stack contents and export it through the serial port.
- Memory dump. In the case of a data error, every data in the program is exported through the serial port.

Offline analysis of memory and stack dumps can complement trace dump information, providing a richer knowledge which ends up in a more precise error diagnosis.

IV. Experimental Results

A. Common experimental setup

Two different experiments are described within this work, presenting particular aspects. However, both of them have a common setup, which is described beforehand.

The Device Under Test (DUT) was a Xilinx Zynq-7010 All Programmable SoC device [20]. Zynq devices integrate SRAM-based FPGA fabric with hard-core ARM processors. We used a commercial board (Zybo) featuring a XC7Z010 device for the experiments, which contains a dual core ARM Cortex-A9 processor, and 512 MB DDR memory. Only one core was used for the experiments, at the nominal 650 MHz clock frequency. The PDTC is implemented in the FPGA fabric of the device and connected to the trace interface as described in previous section. We have used the Xilinx Soft Error Mitigation (SEM) Controller IP [20] to correct errors affecting the PDTC implemented in programmable logic.

An SD card was used to store the boot code, the FPGA configuration file and the application. The device automatically loads this information to the FPGA configuration memory and to the On-Chip Memory (OCM) of the microprocessor to start operation after a power-on reset.

The DUT is controlled from an external host which is connected to the Zybo board. The external host is placed outside the beam. It retrieves information during the experiments from the microprocessor USB serial interface and also from dedicated pins providing the error signals by the PDTC. The external host has the capability to switch off the power of the DUT and restart again when an error is detected. The system is also restarted if the external host detects a timeout or if the communication with the microprocessor is receiving corrupted data because of malfunctioning.

We used a matrix multiplication benchmark executed in a single core. Two input matrices, A and B, are multiplied to obtain the result matrix C. Matrices A and B are initialized from a constant memory and remain unchanged. A golden matrix (Gold) is calculated at the very beginning of the code after A and B initialization and then remains unchanged as well. Matrix C is re-calculated in every iteration of the code, so it is constantly being updated. The source code was implemented in C++. To double check PDTC reported errors, we implemented triplicated data and checked it for consistency inside the

software. Matrices and variables were triplicated, and every operation was repeated for the triplicated variables. At the end of every execution, the results are checked by comparing them with the golden matrix Gold. It must be noted that this redundant implementation is not required by the proposed approach and it was only used as a means to confirm data errors detected by the PDTC.

B. Neutron irradiation results

The proposed approach was tested at Los Alamos National Laboratory's (LANL) Los Alamos Neutron Science Center (LANSCE) in August 2018. LANSCE provides a white neutron source that emulates the energy spectrum of the atmospheric neutron flux. We used 16 Zybo boards and thousands of faults were observed and analysed during the tests, obtained from the irradiation of two versions of the same benchmark:

- B1: 32x32 matrix multiplication, maximum optimization effort, all caches enabled.
- B2: 128x128 matrix multiplication, maximum optimization effort, all caches enabled.

Neutron test results for B1 and B2 show that over 84% and 86% of the faults, respectively, were associated to data abort exceptions. This result makes sense as the matrix multiplication benchmark is very data-intensive, involving many memory accesses to load and store matrix values. Therefore, memory access is one of the most common operations. It also indicates that the vast majority of errors happened in the registers or in the cache controller, but not in the memory itself, because the latter usually produce Silent Data Corruption (SDC) errors which correspond to about 10% of the total in both cases. As mentioned before, unused memory areas were protected in the MMU configuration, so that wrong memory accesses are handled by a custom exception handler that produces a trace dump. The unused memory space is 85% of the total memory address space. From the trace data we can extract the address of the last completed instruction before an error.

The bar graph on Fig. 3 shows the number of errors obtained at each address for 3374 trace dumps in B2. Addresses with no errors are not included in the bar graph.

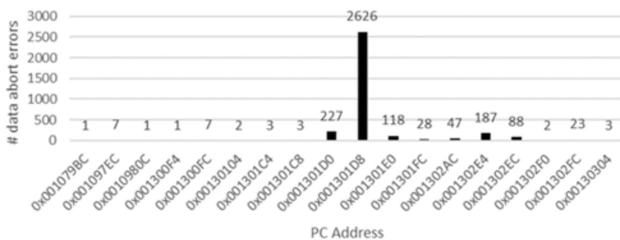

Fig. 3. Bar graph of diagnosed addresses for 128x128 matrix size under neutron irradiation

Remarkably, there are a few addresses which seem to be much more related with data abort errors than the rest of the code, namely the addresses 0x001301D0, 0x001301D8 and 0x001301E0. When correlating the error distribution with the code, it turns out that these instructions correspond to the inner loop of the matrix multiplication algorithm, where the application spends most of the execution time and data memory accesses are the most intensive. Going deeper in the code, we saw that the addresses in the bar graph correspond to instructions immediately preceding a memory access instruction using indirect addressing mode (Table I). However, it can also be noticed that, while all addresses involved in Table I are executed in the same loop, not all memory accesses have the same error sensitivity, being the most relevant the instruction ldr r5, [r2, #-8].

TABLE I
DETAILED ADDRESS CONTENTS

0x001301C4	ldr	r8, [r3, #4]
0x001301C8	add	r3, r3, #2048
0x001301CC	ldr	r6, [r3, #-2040]
0x001301D0	add	r2, r2, #16
0x001301D4	**ldr**	**r4, [r3, #-2036]**
0x001301D8	cmp	r3, r9
0x001301DC	**ldr**	**r5, [r2, #-8]**
0x001301E0	ldr	lr, [r2, #-4]
0x001301E4	**ldr**	**r7, [r2, #-12]**
0x001301E8	mla	r0, r6, r5, r0
0x001301EC	mla	ip, r8, r7, ip
0x001301F0	mla	r1, r4, lr, r1
0x001301F4	bne	1301c4

The bar graph in Fig. 4 shows the number of errors obtained at each address for 984 trace dumps in B1.

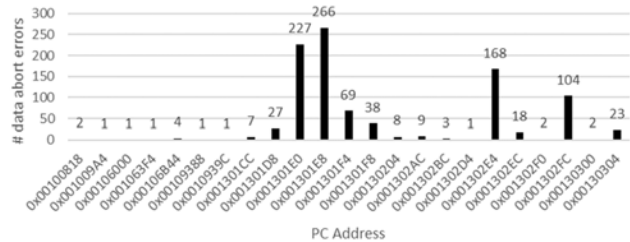

Fig. 4. Bar graph of diagnosed addresses for 32x32 matrix size under neutron irradiation

In the case of Fig. 4, there are several addresses that show high sensitivity, but it also can be observed that they lie in two main error zones. In this case, the addresses 0x001301E0, 0x001301E8 and 0x001301F4 are also related with the inner loop, but there is a second zone with addresses 0x001302E4 and 0x001301FC, which corresponds to the checking loop, which runs along all the matrices to check their values. Results from B1 and B2 demonstrate that the maximum error rates are associated to the most executed memory access instructions, which is the expected result.

The case of SDC errors is the opposite of the previously discussed one: while the pointer used when accessing to a specific memory address can be assumed as correct, the value obtained is corrupted. This can be associated to a fault in the memory cell or in the register storing the value. Some examples that illustrate the diagnosis capabilities are displayed in Table II. In these examples, the temporal evolution of data captured by the ITM is shown. A mismatch can be observed at instant t, which has been normalized to the error instant. In these cases, the erroneous datum differs from the correct one in only one bit, i.e., it is a bit flip. Moreover, it can be observed that the values captured after the error are correct, fitting with the idea that the error is restricted to a particular data value. Furthermore, this

analysis can be correlated with the instruction addresses which are simultaneously obtained to determine the region of code in which the data error appeared. These results prove that trace dump information can be used to reconstruct execution status in the precise moment that the error occurred, so error diagnosis can be achieved.

TABLE II
DATA ERROR DIAGNOSIS

	t-1	t	t+1
a)	0xF038F057	0x270E04C2	0xA35B81D5
	0xF038F057	0x270E14C2	0xA35B81D5
b)	0xC945A391	0xF1BAD7EE	0x5F8F5440
	0xC945A391	0xF1BAD76E	0x5F8F5440

C. *Proton irradiation results*

A proton irradiation campaign was performed using the external beam line of a 18/9 IBA compact cyclotron located at Centro Nacional de Aceleradores (CNA), Sevilla, Spain. The DUT was irradiated in open air with 15 MeV protons. The proton energy in the active area is estimated in the order of 10 MeV. It has been demonstrated in previous experiments [21] that this energy is enough to produce SEEs in a 28 nm technology device without thinning it.

To better take into account errors in the FPGA fabric, the PDTC was duplicated in this case and both instances were implemented in the Programmable Logic (PL) of the device. Moreover, in this experiment the external host retrieves SEM information to register FPGA events that may enable further investigation. The system is also restarted in the case of a mismatch between the outputs of the two PDTC instances, or if the SEM signals an unrecoverable FPGA error. Notwithstanding, unrecoverable SEM errors and communication errors are not taken into account in the results of the experiments reported in this section.

Diagnosis of addresses and data worked as in the neutron experiment. However, the goal of this new experiment was to try additional versions of the matrix multiplication benchmark and collect more detailed results, learning from the previous experience and taking advantage of the higher amount of errors that can be obtained with proton irradiation.

Seven variations of the 32x32 matrix multiplication benchmark Code Under Test (CUT) were evaluated under proton irradiation. The goal is also analyzing the differences in error types among code versions, which could be considered as an additional dimension for error diagnosis. The main characteristics of the CUT versions are the following:

- CUT1: all caches enabled, maximum optimization effort, matrices and processor stack are stored in DRAM.
- CUT2: all caches disabled, maximum optimization effort, matrices and processor stack are stored in DRAM.
- CUT3: all caches enabled, no optimization effort, matrices and processor stack are stored in DRAM.
- CUT4: all caches enabled, maximum optimization effort, matrices and processor stack are stored in OCM instead of DRAM.
- CUT5: only L1 data cache enabled, maximum optimization effort, matrices and processor stack are stored in DRAM.
- CUT6: only L1 instruction cache enabled, maximum optimization effort, matrices and processor stack are stored in DRAM.
- CUT7: only L2 cache enabled, maximum optimization effort, matrices and processor stack are stored in DRAM.

In this experiment, memory and stack dumps were retrieved for additional analysis. Memory dumps exported all matrix values in the case of a data error and stack dumps were triggered in the case of exception errors. Exception errors (EE) are defined as faults that provoke an unexpected state in the processor, so that the main program loses control of the execution, which is transferred to a predefined exception handler. In the case of ARM Cortex-A9 processor, there are the following exceptions:

- Prefetch Abort. Forbidden memory access when an instruction is fetched.
- Data Abort. Forbidden data access attempt.
- Undefined Instruction. Unrecognized instruction code.

Starting from CUT1, which could be considered as the baseline version, CUT2, CUT5, CUT6 and CUT7 evaluate the impact of using caches in the system. It is important to note that DRAM is off-chip, and thus it is also outside of the proton beam. However, caches and OCM are on-chip, so they are prone to errors caused by radiation. CUT3 may give a figure of optimization impact on errors. Finally, CUT4 can be representative of OCM error susceptibility and enables comparison with caches, which are also on-chip.

CUT1 is almost the same software as B1 used in neutron experiment, so it is interesting to evaluate the similarities between these two experiments. In this case, CUT1 showed 86% data abort exceptions and around 10% SDC errors, which is very similar to the behavior found in the neutron experiment. Moreover, it is also worth comparing the error distribution bar graph shown in Fig. 5, which presents the number of errors obtained at each address for 242 trace dumps. This bar graph has a very similar shape to the one obtained for B1 (Fig. 4).

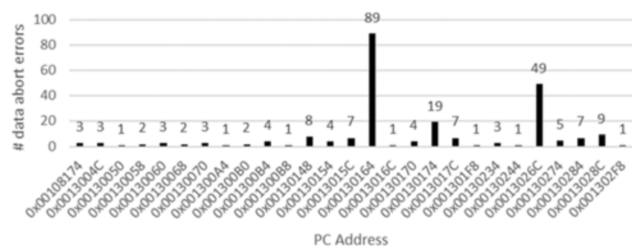

Fig. 5. Bar graph of diagnosed addresses for 32x32 matrix size under proton irradiation

As discussed for the neutron experiment, some data abort errors can be provoked by an erroneous pointer to memory. In the neutron experiments this was only a hypothesis, but in the proton experiments the stack dump functionality was implemented, so we obtained register values in the moment of an exception. We also studied the utility of each register. In optimized code, each register has a very specific function during the matrix multiplication loop. Some registers store A and B values, some others contain intermediate results and

finally others serve as pointers to access all the values. Register values can be obtained from the stack dump. The hypothesis was confirmed for CUT2, where incompatible register values were found in some cases. To give an example, r2 value, which stores the pointer to matrix A, was found with value 0x7a13dad8, which is out of the range of matrix A (0x0013c2c8 to 0x001402c4) thus provoking an access violation which ends up in a data abort exception. In another example for CUT6, r3 value was also found to be outside B matrix (0x001402c8 to 0x001442c4) with a value of 0x40148298. A bit-flip on bit 30 is likely in this case, but r3 value would still be outside B bounds; an MBU in bits 15 and 30 would be plausible as it results in an original value inside B bounds.

Error distribution can also be analyzed, plotting error causes for different CUTs. In Fig. 6, errors are classified into exception errors (EE) and data errors (DE) and are plotted in a normalized way. The cross-section (σ) obtained for the total number of errors is also plotted against the secondary y-axis, and exact values can be found in parentheses below each CUT name in the horizontal axis.

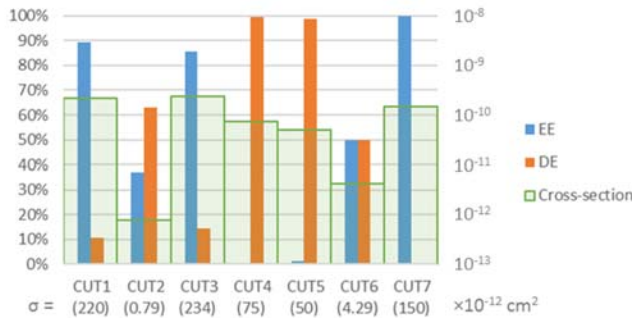

Fig. 6. Exception Errors (EE) vs Data Errors (DE)

It can be noted that in Fig. 6, those CUTs storing data on chip (CUT4 uses OCM and CUT5 uses L1 data cache) are clearly dominated by data errors, while those CUTs using DRAM to store data (which is not under the beam) are not. On the other hand, exception errors dominate when the L2 cache is used (CUT1 and CUT3 use all caches, CUT7 uses only L2 cache). This result solves the hypotheses made on neutron irradiation tests about the origin of data abort exception errors, which are demonstrated to be mostly related with the use of L2 cache. The lowest cross-section is obtained by CUT2, which uses no caches. In addition, CUT3, which is a non-optimized version of CUT1, has a higher cross-section, so no benefit is obtained by removing code optimization.

Apart from Data Abort errors, more types of exceptions can be diagnosed using the trace information: prefetch abort exception (seven occurrences were found in CUT7) and undefined instruction exception (one occurrence was found in CUT4 and another one in CUT6).

Regarding data errors, they can have diverse origin, as different resources (processor core, registers, memory and caches) are involved in the calculations. However, obtained data can be used to extract useful results. In the sequel, we use several categories for data errors. If a matrix has few wrong values (we set a threshold of 25 wrong values), it can be assumed to be a localized error, and the number of erroneous bits is evaluated into the following error types:

- SEU: only one bit is wrong.
- MBU2: two bits are wrong.
- MBU3: three bits are wrong.
- MANY BITS: more than three bits are wrong.

If a matrix has more than 25 wrong values, the number of wrong bits is not evaluated and all cases are labeled as MANY VALUES error type.

Matrices A, B and Gold remain unchanged once they are initialized, so any error found in any of them can be assumed as an error in the memory, or in the initialization process. In the former, few bits would be wrong, in the latter many bits or even many values would be wrong. Fig. 7 represents the distribution of error types for A, B, and Gold extracted from memory dumps.

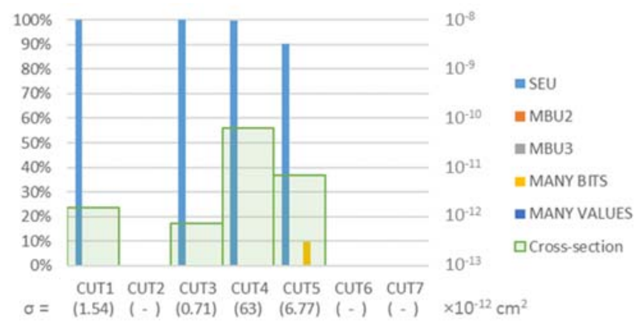

Fig. 7. Memory errors causing data errors

It can be observed that most errors are mainly caused by SEU faults in the memory. CUT4, which uses only on-chip memories, shows the highest cross-section for memory errors. For those CUTs not displaying any cross-section value (CUT2, CUT6 and CUT7) we did not observe any memory error during the tests. The contribution of MANY BITS errors in CUT5 can be explained as an error in the cache controller, returning a wrong value.

Errors observed in the result matrix C may be caused by the propagation of a memory error in the input matrices A or B, by computation errors inside the processor or by an error in the memory after the values are stored. The latter is much less likely than the former as the lifetime of C values in memory is much shorter than in the case of A, B and Gold.

Matrix C values are the only ones included in the trace information and checked by the PDTC. The case of a memory error that propagates to the result can be diagnosed by checking for errors in the memory when a wrong result is detected in the trace. Fig. 8 illustrates the type of errors found by the trace in the cases that memory errors were found, normalized over all data errors found by the trace.

In Fig.8 it can be observed that CUT4 and CUT5 are the most vulnerable to memory errors affecting trace values as they have the highest cross-section values. This fits with the results in Fig. 7 which demonstrated these versions had the highest cross section to memory errors. In addition, it is foreseeable that a single wrong bit (SEU) in memory can produce MANY BITS errors in results, as it is demonstrated for CUT4.

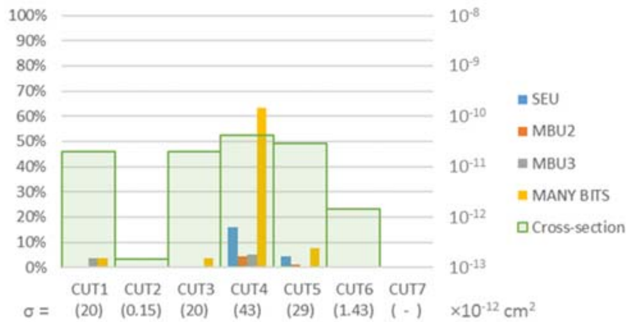

Fig. 8. Memory errors captured by the trace

Fig. 9 illustrates the type of errors found by the trace in the cases that no memory errors were found, normalized over all data errors found by the trace. In this case, the error must have happened inside the processor (register, pipeline, etc.).

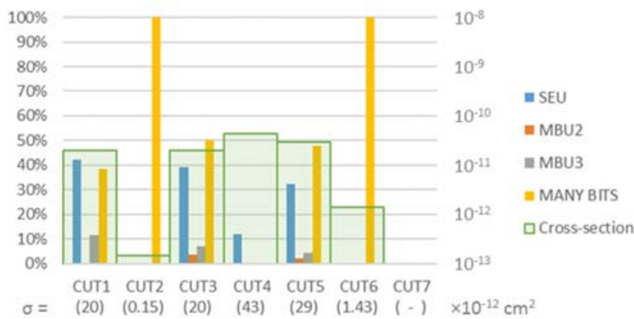

Fig. 9. Computation errors captured by the trace

Information given by Fig. 9 is very rich. Firstly, it can be observed that in CUT4, with all the data in OCM, only about 12% contribution to data errors is found outside the memory. In contrast, those CUTs not using any on-chip data memory (OCM nor L1 data cache), which are CUT2 and CUT6, show a 100% error contribution from the processor and none from the memory. CUT1 and CUT3 give a similar figure in cross-section and error distribution, demonstrating again that no improvement is found in removing optimization. CUT5 gives almost the same figure as CUT1 and CUT3, but with a slightly higher cross-section. This result could mean that CUT1 and CUT3 computation errors are dominated by L1 Data Cache, which becomes more relevant on CUT 5.

It is remarkable that, no matter if the information obtained is from the trace, the memory or the stack dumps, diagnosis is possible because of the low latency error detection of the PDTC, which allows capturing the moment when the application has an error to export all relevant information about it.

Finally, errors in the trace interface and the PDTC have also been analyzed. In some cases, a false positive (dump with no error) is reported by the system. In most cases, it is caused by an error in the FPGA, and PDTC duplication enables to detect it. But there are cases in which the PDTC seems to work properly and even so, a false error is detected. Taking advantage of the double buffer architecture displayed on Fig. 1, we can store trace information in two points of the trace interface chain, namely the ETB and the custom buffer. It can be observed that in some cases, the information contained in each buffer differs in only one bit in one word. This can be associated to an error in the trace interface.

V. Conclusions

Fault diagnosis is highly dependent upon the quality and completeness of the information that can be collected when an event happens. Hardware trace infrastructures, which are commonly found in modern microprocessors, can be conveniently used for this purpose. In the approach proposed in this work, the trace is accessed online to detect errors and retrieve a large amount of information about the internal state of the processor when an error happens. Experimental results demonstrated that with this information it is generally possible to reconstruct the execution status in the precise moment that the error occurred and determine many interesting aspects about the cause of errors, such as the address of the last completed instruction before an error, the type of error and the location of the error. The proposed approach is not intrusive and can diagnose the faults that may happen in a microprocessor running a real application under radiation testing. It can be used offline, to precisely analyze the vulnerability of processor resources, or online, to diagnose errors and implement tailored recovery actions for each type of error while in operation.

The proposed approach relies on the capability of the PDTC to detect errors and thus obtain error evidence information. Because of that, an error which is not detected by the PDTC could not be diagnosed with this approach. However, the PDTC has been proven to detect most observable errors in previous work [17], so very few errors are expected to remain undiagnosed. In the future, we plan to improve the implementation of the PDTC to increase error detection capabilities.

Acknowledgment

The authors would like to acknowledge the support of Los Alamos Neutron Science Center (LANSCE) at Los Alamos National Laboratory (LANL) to perform the neutron irradiation experiment.

References

[1] N. Naber, T. Getz, Y. Kim, and J. Petrosky. “Real-time fault detection and diagnostics using FPGA-based architectures”. Proc. Intl. Conf on Field Programmable Logic and Applications, pp. 346-351, Sept. 2010.

[2] S. Wei, F. Tongshun, and D. Mingfang. “Research for digital circuit fault testing and diagnosis techniques”. Proc. Intl. Conf. on Test and Measurement, vol. 1, pp. 330-333, Dec. 2009.

[3] F. A. Bower, D. J. Sorin, and S. Ozev. “Online diagnosis of hard faults in microprocessors”. ACM Trans. on Architecture and Code Optimization (TACO), vol. 4, no. 2, Article 8, June 2007.

[4] B.K. Sikdar, N. Ganguly, and P.P. Chaudhuri. “Fault diagnosis of VLSI circuits with cellular automata based pattern classifier”. IEEE Trans. on CAD of Integrated Cicuits and Systems, vol. 24, no. 7, pp. 1115-1131, July 2005.

[5] J. M. Mogollon, J. Napoles, H. Guzman-Miranda, and M. A. Aguirre. “Real Time SEU Detection and Diagnosis for Safety or Mission-Critical ICs Using HASH Library-Based Fault Dictionaries”. Proc. RADECS, paper J-3, pp.705-710, Sept. 2011.

[6] J. M. Mogollon, J. Napoles, H. Guzman-Miranda, and M. A. Aguirre. “Metrics for the Measurement of the Quality of Stimuli in Radiation Testing Using Fast Hardware Emulation”. IEEE Trans. on Nuclear Science, vol. 60, no. 4, pp. 2456–2460, Aug. 2013.

[7] S. P. Buchner, F. Miller, V. Pouget, and D. P. McMorrow. “Pulsed-Laser Testing for Single-Event Effects Investigations”. IEEE Trans. on Nuclear Science, vol. 60, no. 3, pp. 1852–1875, June 2013.

[8] F. W. Sexton. “Microbeam Studies of Single-Event Effects”. IEEE Trans. on Nuclear Science, vol. 43, no. 2, pp. 687–695, Apr. 1996.

[9] Y. Xu1 et al. “An Accelerator-Based Neutron Microbeam System for Studies of Radiation Effects”. Radiation Protection Dosimetry, vol. 145, no. 4, pp. 373–376, Dec. 2011.

[10] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt and T. Austin, "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor". Proc. 36th Annual IEEE/ACM Intl. Symp. on Microarchitecture (MICRO-36), pp. 29-40, Dec. 2003.

[11] R. Velazco, S. Rezgui and R. Ecoffet, "Predicting error rate for microprocessor-based digital architectures through C.E.U. (Code Emulating Upsets) injection". IEEE Trans. on Nuclear Science, vol. 47, no. 6, pp. 2405-2411, Dec. 2000.

[12] R. Mansour and Velazco, “An automated SEU fault-injection method and tool for HDL-based Designs”. IEEE Trans. on Nuclear Science, vol. 60, no. 4, pp. 2728–2733, Aug. 2013.

[13] H. Asadi, V. Sridharan, M. B. Tahoori and D. Kaeli, "Vulnerability Analysis of L2 Cache Elements to Single Event Upsets". Proc. Design Automation & Test in Europe Conf, pp. 1276-1281, March 2006.

[14] S. Z. Shazli, M. Abdul-Aziz, M. B. Tahoori and D. R. Kaeli, "A Field Analysis of System-level Effects of Soft Errors Occurring in Microprocessors used in Information System”, IEEE Intl. Test Conf., paper 24.3, Oct. 2008.

[15] D.V. Kodavade, S.D.Apte, “Troubleshooting Microprocessor Based System using An Object Oriented Expert System”. Intl. Journal of Adv. Computer Science and Applications, vol. 3, no.5, pp. 111-116, 2011.

[16] G. M. Swift, F. F. Fannanesh, S. M. Guertin, F. Irom and D. G. Millward, "Single-event upset in the PowerPC750 microprocessor". IEEE Trans. on Nuclear Science, vol. 48, no. 6, pp. 1822-1827, Dec. 2001.

[17] M. Peña-Fernandez, A. Lindoso, L. Entrena, M. Garcia-Valderas, Y. Morilla and P. Martín-Holgado, "Online error detection through trace infrastructure in ARM microprocessors". IEEE Trans. on Nuclear Science, vol. 66, no. 7, pp. 1457-1464, July 2019.

[18] “CoreSight Components. Technical Reference Manual”, ARM Ltd., DDI0314H, 2009.

[19] “CoreSight Program Flow Trace. Architecture Specification”, ARM Ltd., IHI0035B, 2011.

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

[21] 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 Trans. on Nuclear Science, vol. 65, no. 8, pp. 1835-1842, Aug. 2018.