Pena-Fernandez, M., Lindoso, A., Entrena, L. & Garcia-Valderas, M. (2020). Error Detection and Mitigation of Data-Intensive Microprocessor Applications Using SIMD and Trace Monitoring. *IEEE Transactions on Nuclear Science*, 67(7), pp. 1452–1460.

DOI: 10.1109/tns.2020.2992299

Error Detection and Mitigation of Data-Intensive Microprocessor Applications using SIMD and Trace Monitoring

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

Abstract— **This work proposes a software error mitigation approach that uses the SIMD coprocessor to accelerate computation over redundant data. In addition, an external IP connected to the microprocessor's trace interface is used to detect errors that are difficult to cover with software-implemented techniques. The proposed approach has been implemented in an ARM microprocessor and an irradiation campaign with neutrons has been carried out at Los Alamos National Laboratory. Experimental results demonstrate the high error coverage (more than 99.9%) of the proposed approach. The neutron cross section of errors that were not corrected nor detected was reduced by more than three orders of magnitude.**

***Index Terms*— ARM, SIMD, NEON, microprocessor, microprocessor trace, error mitigation, error detection.**

I. Introduction

The progress of semiconductor manufacturing technology has made microprocessors cheap and affordable for a huge variety of applications, including safety-critical and high availability ones. For these applications, radiation-induced soft errors are becoming an increasing concern. As a consequence, there is a demand for error mitigation techniques that can be adapted to the requirements of different types of systems. High-end systems have often used redundant hardware to detect or correct errors [1]. However, hardware solutions may be prohibitively expensive and difficult to justify in many cases. Alternatively, Software Implemented Fault-Tolerance (SWIFT) techniques are often seen as a viable solution.

Software error detection and correction techniques are generally based on duplicating or triplicating program instructions and data [2]. The goal is to create redundant execution streams or threads, so that errors can be detected or corrected by comparing or voting, respectively, the results of the redundant execution streams. This approach may effectively protect against data errors, but it introduces significant performance penalties. Furthermore, replicating data is not sufficient, because errors may also corrupt the control flow. Pure software control-flow checking techniques are typically based on dividing the code into basic blocks and computing signatures for each basic block [3], which are checked whenever there is a change in the control flow. The computation and checking of signatures is another overhead that negatively affects performance. To sum up, reducing the performance overhead of software-implemented fault-tolerance techniques without compromising error detection and mitigation capabilities is a challenge.

From a general point of view, the drawbacks of software-based fault-tolerant techniques can be reduced by making a smart use of existing microprocessor resources. Building upon this idea, in this work we propose a solution based on the use of SIMD (Single Instruction Multiple Data) co-processor to accelerate the computation over replicated data. Control-flow errors are also covered by including control signatures in the execution flow. However, covering all possible types of errors with a pure software approach is generally very difficult. Therefore, the proposed approach is combined with Trace Monitoring to additionally support the detection of control-flow errors. Without loss of generality, the proposed approach has been developed and evaluated for an ARM Cortex-A9 processor [4].

The suitability of SIMD microprocessor extensions in radiation environments has been analyzed in [5]. Experimental results have shown that the use of the ARM SIMD co-processor, known as NEON™, notably improves performance but can also increase cross section. However, the performance increase is generally higher than the cross section increase, so that the Mean Work To Failure improves when NEON™ coprocessor is used. SIMD co-processors are very well suited for data-intensive applications, as they can accelerate processing when operations are repeated over large data sets.

Trace Monitoring reuses the debug infrastructure that is commonly included in modern microprocessors for on-line error detection. The use of Trace Monitoring for error detection in an ARM microprocessor has recently been demonstrated in [6]. This approach uses two trace macrocells that are included by default in the commercial version of the microprocessor: the Program Trace Macrocell (PTM) and the Instrumentation Trace

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).

Macrocell (ITM). The PTM reports the flow of instructions executed by the microprocessor while the ITM can check selected data.

In this work, we propose an error detection and mitigation approach that combines an SIMD-based data hardening technique with the detection capabilities of an external IP connected to the trace interface. The proposed approach has been tested with neutrons at Los Alamos National Laboratory. Experimental results show that the combination of both techniques achieves a high error coverage.

The remaining of this paper is as follows. Section II summarizes related work. Section III describes the proposed approach. Section IV shows the experimental results. Finally, Section V summarizes the conclusions of this work.

II. Background and Related Work

Microprocessor hardening techniques are usually divided into software and hardware techniques. Software techniques modify the software to harden the microprocessor while hardware techniques modify the hardware [2]. Hybrid techniques are a combination of the benefits of both software and hardware techniques. Errors produced in a microprocessor are generally classified into two categories: control-flow errors and data errors.

Control-flow errors affect to the correct order of code execution and data errors affect to the data handled by the executed software. Even if errors are usually classified this way, some data errors can trigger control-flow errors and vice versa. For example, if wrong data affects the condition of a conditional instruction, it could trigger a control-flow error.

Traditionally, software-based control-flow error detection techniques are mainly based on signatures [3] or assertions [7]. Signature-based techniques assign unique signatures to parts of the code so that it is possible to check them to detect control-flow errors. The code fragments are named Basic Blocks (BBs). A BB is a set of consecutive instructions with no branches except for possibly the last instruction. A unique signature is assigned to each BB. When a BB is executed, its signature is computed on-line and compared with the assigned signature. Errors are triggered whenever a discrepancy is found. Signature techniques usually present large overheads in terms of storage and performance. Not only do they require additional memory to store signatures but also additional computations have to be inserted in the code to compute signatures and to check them.

Assertion techniques modify software by inserting additional instructions that check the correctness of the executed code. Their success depends on the application designer ability to locate them in the code and the data that is checked. Assertions are application-dependent. Assertions can be combined with other techniques to increase the fault coverage. In [7], the CEDA technique is proposed which combines assertions and signatures.

Control-flow error techniques cannot achieve full error coverage because data errors are not covered. However, it must be noted that control-flow errors frequently lead to execution misbehavior that can trigger a microprocessor hang (microprocessor cannot return to normal execution state).

Software hardening techniques to detect or correct data errors are commonly based on introducing data and code redundancy with different levels of granularity (instruction, blocks, function, program, etc.) [2], [8], [9], [10], [11]. The deeper the replication level applied, the smaller the error detection latency but the higher the overhead in terms of memory usage and performance decrease. Duplication is used when error detection is sufficient. However, error mitigation requires triplication in order to restore the erroneous copy. The concept of Sphere of Replication (SoR) [1] is often used to compare replication approaches. The SoR defines the logical replication boundary inside which faults can be detected. Data are replicated when entering the SoR and are checked before leaving the SoR.

EDDI [9] and SWIFT-R [11] are representative examples of data replication techniques. EDDI duplicates instructions and data to create two redundant, intertwined execution streams. Each redundant stream uses separated registers and memory addresses to avoid interfering with each other. Thus, the memory is part of the SoR. SWIFT-R extends this idea to error correction. The SWIFT-R transformation can be seen as TMR implemented in software at the instruction level. Two redundant copies of all data are created and processed by replicated instructions along with the original data and instructions in the same execution thread. SWIFT-R thus creates three redundant, intertwined execution streams. At certain synchronization points, software majority voting is inserted in the program to correct errors.

At higher levels of granularity, two or more identical copies of the same program can be run as independent threads and their outputs are compared. This type of techniques is named Redundant MultiThreading (RMT) [1]. RMT can be efficiently implemented on top of a simultaneous multithreaded (SMT) processor or on a multicore processor (Chip-level Multiprocessing, CMP). However, these techniques need a processor that specifically supports multithreading.

To achieve a complete solution with high coverage, it is necessary to address both data and control-flow errors. Software approaches can be used for both types of errors, but it has been demonstrated that external hardware approaches can be more effective for control-flow errors [13]. The trace interface is an infrastructure that is available in most modern microprocessors and its use is only intended for debugging purposes. Once the design cycle is completed, it is no longer in use. Some works have proposed the use of this infrastructure for error detection [14]. A hardened LEON3 soft-core microprocessor is proposed in [15]. This hardened LEON3 uses a hybrid technique based on software duplication for data errors and an external IP connected to the trace interface for control-flow errors. The use of the trace interface for error detection in ARM microprocessors has recently been proposed and demonstrated, showing good error detection [6]. The capabilities of the trace interface as an observation point for error detection have also been adapted to multicore systems in [12].

The main contribution of the present work is to combine the use of the SIMD coprocessor and the trace interface for error detection and mitigation. The SIMD coprocessor is used to accelerate computation over redundant data by parallel processing. In addition, the SIMD coprocessor is also used to compute control signatures. Trace Monitoring is used to detect control-flow errors that are difficult to handle from a pure software approach.

III. Proposed Approach

In this work we propose an approach that can mitigate data errors and detect control-flow errors. Our software-based data-flow hardening technique relies on the use of the SIMD co-processor, which is used to perform operations over replicated data. This technique is complemented by Trace Monitoring to detect control-flow errors that are difficult to handle from a pure software approach, such as memory exceptions. To this purpose, we use an external IP connected to the trace interface that observes the program execution. The Trace Monitor is also used during the experiments to check the results of the computations and validate the executions. Each of these techniques is described in the following subsections.

A. Data hardening with NEON SIMD co-processor

As SWIFT-R, the proposed data error mitigation approach uses three redundant copies of data to correct errors. However, it must be noted that in SWIFT-R the SoR does not include the memory. Data are loaded once from memory and the redundant copies are created in the processor. This approach may benefit performance but it increases vulnerability, because a load error may propagate to all copies [11]. For this reason, in our implementation the copies are created in memory.

The performance overhead produced by data replication techniques can be reduced by exploiting the parallel processing capabilities of SIMD co-processors, which are commonly found in modern microprocessors. SIMD acceleration for replicated data computation has been proposed previously in [16]. An SIMD co-processor has its own register file made of wide registers that can store multiple data to be processed in parallel. To this purpose, the instruction set is also extended with parallel instructions that perform the very same operation over several data stored in the co-processor register file. In particular, the ARM SIMD coprocessor, NEON^{TM} [17], has sixteen 128-bit wide registers (Q0-Q15). Each of these registers can be used as a vector of elements of the same data type, also called lanes. Supported data types include 8, 16, 32 and 64 bit signed and unsigned integer as well as single precision floating point. A NEON instruction performs the same operation over all lanes in parallel. Fig. 1 shows the NEON register file and also an example that illustrates the behavior of SIMD instructions. The operation shown performs four 32-bit additions simultaneously (every NEON register is able to allocate four 32-bit lanes).

In our approach, data are replicated into the lanes of NEON registers. One single NEON register stores the original data and its two copies, and the required operation over replicated data can be implemented with a single NEON instruction. However, the executed parallel instruction can be a single point of failure, because an error in the instruction can provoke a wrong result in all lanes. To solve this problem, an additional lane is used for control data. Thus, in the proposed approach, each scalar variable is transformed into a vector of 4 elements. Element 0 contains the control data, while elements 1 to 3 contain redundant data. All elements are processed in parallel by NEON instructions.

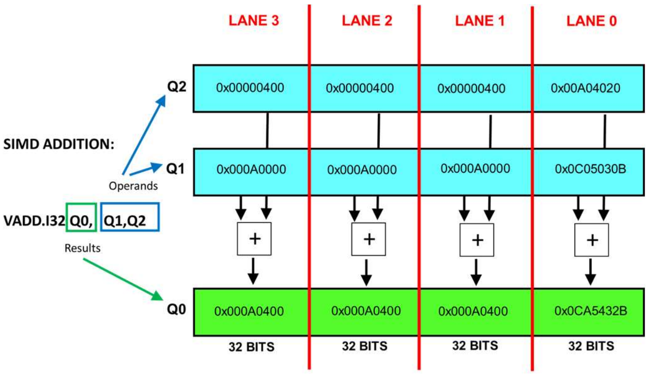


Fig. 1. NEON register file and instruction example

Fig. 2 shows the SIMD-based hardened data structure used in this work. All lanes from one SIMD register are devoted to one single data: 3 lanes for original data and its two copies, and one lane for control data. An example of the computation with this data structure is shown in Fig. 1. Lanes 1, 2 and 3 perform the computation with the same data (original data and two copies) and lane 0 performs the computation with the control data value.

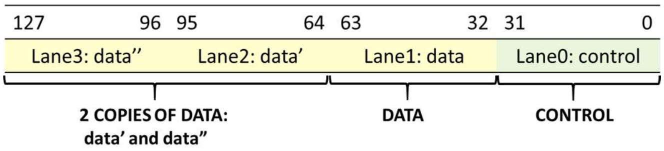


Fig. 2. SIMD-based hardened data.

Control data are subject to the same operations as regular data. However, control data are static and the final results on the control data lanes can be predicted. From this point of view, control data work as control signatures of the sequence of instructions executed on the data. At a synchronization point, data errors are corrected by majority voting and control errors are detected by comparing the control signature with the expected one.

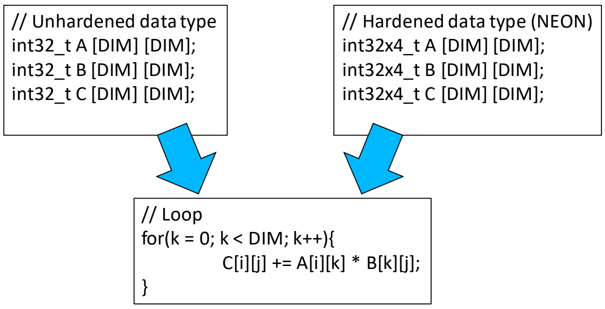
A correct control signature indicates that the sequence of executed instructions was correct. Control data checking must be performed at least before any dynamic branch, i.e., when the program flow may vary depending on the values of the variables. The concept of Basic Block (BB) has been used in the past for this purpose [3]. In our approach, we extend this concept to Static Blocks.

A Static Block (SB) is a set of instructions that are executed in a sequence that can be determined at compile time. For instance, consider a loop with no branches inside the loop body. The loop body can be considered as a BB. The loop iterates over this BB and the signature is checked at the end of every iteration. However, if the number of iterations in the loop is static, the signature can be computed for the entire loop. In this case, the loop, including all iterations, is considered a SB.

Most data-flow hardening approaches work at instruction level and require specialized compilers to transform the original code into the hardened code. This is troublesome from a practical point of view. On the contrary, the proposed NEON-based mitigation approach can be easily implemented by using object-oriented programming (C++). The NEON coprocessor can be programmed in three different ways [18]:

- Assembly. This is the recommended mode for high performance, but it is complex to handle. With assembly, application developers can completely control the implementation: SIMD instructions, registers, load and store, etc.
- Automatic NEON code generation. Compiler flags and data structures can be selected so that the compiler automatically detects parallelization in the code. It must be noted that compilers will not be able to detect all possible parallelization unless application developers help them by slightly adapting their code. Even then, ARM recommends to carefully check the generated assembly code to make sure the compiler has used SIMD instructions correctly.
- Intrinsics. Intrinsics can be seen as special functions that implement NEON instructions. They can be considered as an intermediate level of detail between high level language and assembly. For the application developer, they can be considered as regular functions that are called in the application. However, when they are executed, no function call takes place. The compiler will directly translate intrinsics to NEON instructions and insert them in the generated assembly code. The main advantage of intrinsics is that the developer can control the SIMD instructions that are in use, but some low level decisions are handled by the compiler to reduce the complexity. For instance, register allocation is performed automatically when intrinsics are used. Also, the compiler can reorder instructions to achieve higher performance. The main disadvantage is that there is not complete control of the generated code and the performance may be reduced with respect to assembly. Intrinsics represent a good tradeoff between controlling the SIMD instructions usage and the required abstraction level.

In this work, we have mostly used automatic NEON code generation, complemented by intrinsics in some particular cases. Using C++, data and operations can be easily replicated by defining appropriate data types that can encapsulate the redundant data and the control data. In fact, some data types already exist that can support this approach in a transparent manner. NEON libraries include NEON extensions of most common data types. For instance, int32x4_t data type is the extension of int32_t to 4 data, and it includes the definition of typical integer operations extended to 4 lanes, such as addition and multiplication. Fig. 3 illustrates this concept for a matrix multiplication example. The original declarations of the unhardened matrices are shown on the left. In the hardened version, these declarations are substituted by the ones shown on the right, which use the NEON data types. The code shown at the bottom of Fig. 3 works in both cases and it does not require any particular adaptation for the hardened case. However, in the hardened case each operation is executed in parallel for all lanes. This way, the extension of most operations to the x4 data type is transparent to the user.



```
// Unhardened data type
int32_t A [DIM] [DIM];
int32_t B [DIM] [DIM];
int32_t C [DIM] [DIM];
```

```
// Hardened data type (NEON)
int32x4_t A [DIM] [DIM];
int32x4_t B [DIM] [DIM];
int32x4_t C [DIM] [DIM];
```

```
// Loop
for(k = 0; k < DIM; k++){
        C[i][j] += A[i][k] * B[k][j];
}
```

Fig. 3. Code example of data hardening using NEON data types

For a given SoR, the implementation of the proposed approach is summarized in the following steps. First, the SoR input variables must be created and initialized with redundant data. Then, the data types of variables in the SoR must be substituted by their respective hardened data types. This step just affects the declaration of the variables, as shown in Fig. 3. The control data lanes must be assigned when entering a SB and checked when leaving the SB. Finally, the SoR output variables must be checked when leaving the SoR. Checking can be implemented in the program or using the trace monitor, as described in the next section. With the use of object oriented programming (C++), initialization and checking are supported by appropriate functions that are defined for each data type, so the designer just needs to call them where applicable in the code.

The NEON-based approach described in this section can mitigate data and control-flow errors. However, software techniques are limited to the accessible microprocessor's parts in the programmers' model, thus limiting error coverage, especially in the case of control-flow errors. To broaden the error scope, we have combined the proposed technique with the trace monitoring technique described in the next section.

B. Trace monitoring

The second microprocessor built-in feature that has been reused in the scope of this work is the trace interface. Trace resources are generally conceived to provide application development support like code coverage, performance analysis, timing requirements supervision and execution profiling. For that purpose, trace components provide execution- and data-related information at low level. This information is usually gathered by specific equipment along with computer-based software to provide useful statistics to the developer. Depending on the specific equipment and the system overall design, some analysis can be done online during processor execution while others must be processed offline. With that in mind, it is possible to extend the use of the trace information to different purposes by using the same low-level information. In the case of this work, trace information has been used to detect radiation-induced processor errors.

The use of the trace interface for error detection is an extension that is not specifically supported by the processor providers. In addition, the use of computer-based tools may not be suitable for detecting errors caused by radiation in an embedded system. For that reason, the Trace Monitoring technique proposes to design a custom IP based on the trace interface protocol specification, putting special attention in those features of the trace interface which are helpful to detect errors. Low power and small area are requirements for this IP, so that it can be embedded in an application with minimum penalties. With respect to performance, the IP is specifically optimized to process trace information at high speed with the aim to reduce error detection latency.

The microprocessor used in this work includes a trace subsystem based on the ARM CoreSight technology [19]. CoreSight is a family of components which are commonly found in almost all ARM microprocessor implementations, providing tracing and debug functionalities. Particularly, for this application, two modules have been used: the PTM (Program Trace Macrocell) [20], to obtain trace information related with program execution flow, and the ITM (Instrumentation Trace Macrocell) [21], to obtain trace information related with program data. Trace information is managed by CoreSight components and transferred to an external IP to be checked.

Trace monitoring with a custom external IP for ARM microprocessors was introduced in [22] and extended in [6]. With this technique, control-flow errors are detected by an external IP called Program & Data Trace Checker (PDTC) that observes and analyzes the trace information provided by the PTM. For instance, the PDTC can detect errors in memory addresses that are difficult to detect with a pure software approach. Even if an address is checked before issuing a memory access instruction, the address can get corrupted when the instruction is executed, resulting in a memory exception. Errors concerning execution out of expected code regions can be detected with very low latency, as Program Counter (PC) addresses are observed through the trace interface just a few clock cycles after they are executed [6]. In addition to the PDTC error detection capabilities presented in [22], a new feature has been implemented to enhance control-flow error detection, which we called loop watchdog. Many applications execute a main loop running indefinitely, so it is possible to compute the maximum time for that loop to be executed. The PDTC can be configured with the Program Counter value of the first instruction of the loop and with the maximum execution time. If the configured address is not received within the expected time, the PDTC will raise an error signal. With this approach, the maximum error detection latency of the PDTC for control-flow errors is one loop cycle. The PDTC can detect these situations and could trigger rollback actions in a more reliable manner and with lower performance penalty than software techniques.

To provide a valuable feedback of the proposed data mitigation technique, a double check was performed externally through the trace interface. Replicated data and control data are checked in the program, but they are also checked in the PDTC and reported during the experiments. This external double check is feasible because data values can also be traced through the ITM . The PDTC data checking capabilities presented in [6] have been extended to handle triplicated data. In the case the three values received are not equal, the PDTC raises an error.

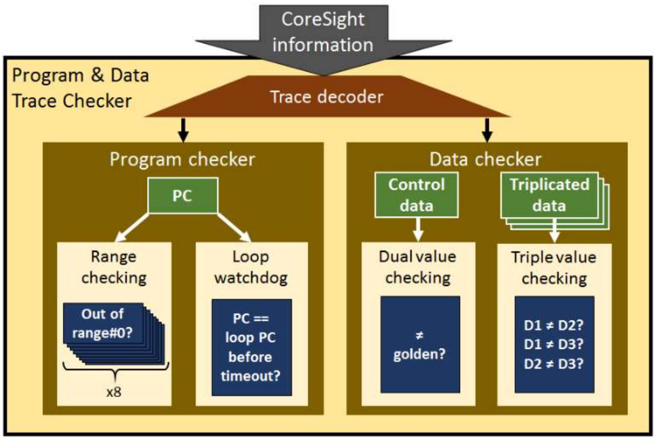


Fig. 4. Program and Data Trace Checker (PDTC) architecture

Fig. 4 illustrates the architecture of the PDTC showing the modules in use: Program Checker and Data Checker. The Program Checker contains configurable address range registers and watchdog registers. These registers can be configured by the user to specify the valid code regions and the expected loop start address and execution time, respectively. Then, the PC addresses provided by the PTM are compared with the configured address range registers to determine if execution has reached a forbidden or unexpected region. The PC address is also compared with the watchdog start address to clear the watchdog. The Data Checker contains a set of data registers that are associated to specific checks, including dual and triple equal comparison. To check hardened output data, the user just needs to write the data into the appropriate registers and the checker performs the comparison. To use trace monitoring in a particular application, the following actions are generally required: (1) configure and enable the microprocessor trace port [6], [22]; (2) configure the PDTC with the allowed instruction address ranges [22] and the address of the main loop(s); and (3) export desired data values to be checked by the PDTC [6]. These actions are supported by the PDTC driver in high-level code (C++), so they can be implemented with simple function calls. Instruction addresses are referenced by tags that can be added to the code. Data to be checked are referenced by variable names. More detailed information about the configuration of the trace components can be found in [6] and [22].

The PDTC is a small piece of hardware (about 6% of a rather small device, XC7Z010, [6]). Most of it is actually used for the configuration and trace interfaces with the microprocessor. The impact of the enhancements introduced in this work in the size of the trace monitor is negligible. With respect to performance, the use of the trace interface does not introduce any delay penalty. The only overhead is caused by exporting data values to be checked, but the checks are performed by the trace monitor in parallel with the processor.

IV. Experimental Results

The proposed approach was tested with neutrons at Los Alamos National Laboratory in 2018.

A matrix multiplication benchmark was selected to perform the tests. This is a very data-intensive benchmark, so the data hardening is suitable to be accelerated with the proposed technique. Matrices were composed of 32-bit integer elements. On every power cycle, both input matrices, A and B, were initialized with random values. For the control data, we used randomly selected values that were also initialized on every power cycle. After the initialization of the matrices, a first matrix multiplication computation is performed to obtain the golden matrix, Gold. Then, the code enters an infinite loop where three steps are performed sequentially: 1) matrix A and B are multiplied to obtain result matrix, C; 2) redundant values, if present, in matrix C are checked between them and the control value, if present, is checked against the Gold control value; and 3) matrices C and Gold are compared value by value. Three error signals were provided to report correctable data error at step two, or control data error at step two, or Silent Data Corruption (SDC) error at step three, respectively. The code was optimized for performance using maximum optimization option (-O3) in the compiler settings. The benchmark was run on bare metal without operating system.

The described benchmark was running on one of the cores of a dual core ARM CORTEX™-A9 processor. The Device Under Test (DUT) was a Zynq XC7Z010 from Xilinx [23] mounted on a commercial board (Zybo) [24]. The PDTC was implemented in the programmable logic of the device and connected to the trace interface of the processor via Trace Port Interface Unit (TPIU) through Extended Multiplexed Input Output (EMIO) interface. Errors in the programmable logic that may affect the PDTC are corrected or detected by the Soft Error Mitigation (SEM) Controller IP [26], which is included in the system for this purpose. The SEM is monitored by the external host to record errors in the programmable logic.

A picture of the experimental setup is shown in Fig. 5. In order to maximize the number of events gathered during the experiment, we used 16 Zybo boards (on the left of Fig. 5) that were exposed simultaneously to the neutron beam. The fluence per board was 1.08×10^{11} n/cm^2. The boards are controlled by 4 external hosts (right side of Fig. 5) that are able to collect the results and can reset each of the DUTs in the case of a misbehavior during the experiment. Particularly, the external host can power cycle each DUT in the case any error is reported by a DUT. In addition, the external host has a watchdog timer for each DUT, so if no status information is received from the DUT in a specified period of time, a timeout is logged and a power cycle is triggered. Every external host controls four DUTs. The controllers are located near the DUTs but they are not exposed to the neutron beam. The results gathered by the external hosts are sent to a computer located outside the beam influence and processed offline.

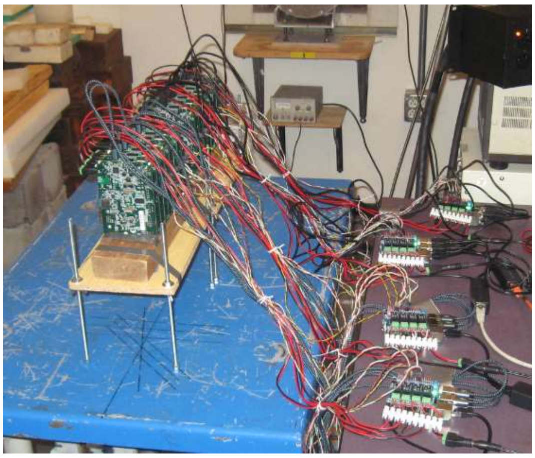


Fig. 5. Experimental setup

A. *Performance comparison*

To analyze the performance of the proposed technique, we measured the time required to execute the matrix multiplication benchmark with a timer, running the benchmark several times until the measure stabilizes. Table I shows the performance overhead of a conventional software-implemented fault tolerance (SWIFT) approach and the implementation of the proposed approach using NEON (NEON-SWIFT) for several matrix sizes and cache options. The results are normalized to the execution of the original unhardened code for each case, so that each value shown in the table is the number of times the execution of the unhardened code using the same cache option.

For the proposed approach, the performance overhead is around 4 in all cases. This result is explained by the transformation of each original data into 4 data (3 redundant copies plus control data). Memory access dominates and the arithmetic computational effort is similar to the original code. The peak overhead for cache-enabled, 64x64 matrix size is because in this case the original data fits in the cache while the hardened data exceeds the cache size. On the contrary, for the conventional SWIFT implementation the overhead is generally much larger. In addition to the memory access overhead (only 3 times in this case) there is the overhead due to replicated instructions. The performance benefits will increase for applications with a higher ratio of arithmetic operations to memory accesses.

TABLE I: PERFORMANCE OVERHEAD

	Cache Disabled		Cache Enabled	
Size	SWIFT	NEON-SWIFT	SWIFT	NEON-SWIFT
8x8	9.43	3.36	3.21	3.75
16x16	10.83	3.85	3.68	4.10
32x32	7.93	3.82	5.13	4.31
64x64	7.93	4.00	8.36	5.42
128x128	7.74	4.11	6.90	4.48

The relatively large performance overhead of the SWIFT cases with cache disabled is due to two main factors: the amount of redundant data and operations, and the optimization performed by the compiler. The second factor is very relevant in the case of complex processors, such as ARM Cortex A9, which include out-of-order execution, branch prediction, etc. Thus, the structure of the code has a strong impact on the performance. In our case, we used a default coding and compilation for maximum optimization (-O3). The original code of the matrix multiplication is very regular and the compiler makes a good job at optimizing it. However, the SWIFT code uses three times the data and requires and additional inner loop to repeat operations. We have also repeated the analysis using optimization level 2 (-O2), although the results are not shown here for the sake of brevity. Interestingly, the performance overhead figures are similar for the SWIFT case and much better for the SWIFT-NEON case (between 1.71 and 1.91). This result shows that SWIFT code is hard to optimize. On the other hand, as the general optimization level is reduced, the performance benefits of NEON are more relevant.

It must be noted that the proposed approach is intended for data intensive applications. For control intensive applications, i.e., with many conditional changes in the control flow, this approach may produce poor results if used in a straightforward manner. This is because the SIMD coprocessor produces performance benefits once data is loaded into the SIMD register file and has limited support for conditional instructions. In general, taking full advantage of SIMD co-processor performance generally requires specific algorithm restructuring. This is true for redundant computations as well. However, the use of ad-hoc coding techniques to optimize the use of the SIMD co-processor is beyond the scope of this paper.

B. Neutron radiation results

The neutron radiation experiments were performed with cache-enabled versions of the matrix multiplication benchmark for 32x32 and 128x128 matrix sizes. For each matrix size, we tested three code versions: original version with no software-implemented fault tolerance (Unhardened); software-implemented fault tolerance (SWIFT); and the proposed approach using NEON (NEON-SWIFT). The total fluence per benchmark was 2.15×10^{11} n/cm^2 collected in approximately 2 days of effective exposure time. Trace error detection using the PDTC was enabled in all cases, as it does not affect the execution.

Table II summarizes the results of the experiments. The errors were classified into the following categories:

- Det. Hang: A control-flow error detected by the PDTC. These errors typically result in an unexpected exception that makes the processor hang or crash.
- Corr. Data: Detected and corrected data error in triplicated data.
- Det. Control: Detected error in control data.
- Det. Trace: Error detected by the PDTC that could not be confirmed by any other means.
- SDC (Silent Data Corruption): unmasked data error.
- Hang: undetected timeout or exception.

The last two categories (SDC and Hang) are the errors that are not detected nor corrected. The last two rows in Table II show the cross sections and their 95% confidence intervals for each case, considering all errors and considering only the undetected errors (SDC and Hang), respectively. For convenience, a bar graph of the cross sections is also shown in Fig. 6.

Most of the errors were in the category of Det. Hang, except for the case of 32x32 matrix with unhardened code. Recent work with the same device [25] has reported that Hang errors dominate when L2 cache is used. The particular case of 32x32 matrix with unhardened code is because in this case all matrices fit in the L1 data cache, so that no L2 cache access is produced during the execution.

TABLE II: NEUTRON RADIATION RESULTS

	32 x 32 (cache enabled)			128 x 128 (cache enabled)		
Error category	Unhardened	SWIFT	NEON-SWIFT	Unhardened	SWIFT	NEON-SWIFT
Det. Hang	22 (7.46%)	1,508 (85.7%)	1,459 (84.7%)	4,699 (98.7%)	5,056 (88.9%)	4,687 (86.0%)
Corr. Data	0 (0.00%)	152 (8.64%)	135 (7.84%)	0 (0.00%)	597 (10.50%)	510 (9.36%)
Det. Control	0 (0.00%)	0 (0.00%)	56 (3.25%)	0 (0.00%)	0 (0.00%)	201 (3.69%)
Det. Trace	23 (7.80%)	97 (5.51%)	72 (4.18%)	17 (0.36%)	25 (0.44%)	50 (0.92%)
SDC	250 (84.75%)	3 (0.17%)	1 (0.06%)	41 (0.86%)	7 (0.12%)	0 (0.00%)
Hang	0 (0.00%)	0 (0.00%)	0 (0.00%)	1 (0.02%)	1 (0.02%)	0 (0.00%)
Total	295 (100%)	1760 (100%)	1,723 (100%)	4,758 (100%)	5,686 (100%)	5,448 (100%)
Cross section (all errors) cm^2	1.37×10^{-9} $(1.22 \times 10^{-9}$, $1.54 \times 10^{-9})$	8.17×10^{-9} $(7.80 \times 10^{-9}$, $8.56 \times 10^{-9})$	8.00×10^{-9} $(7.63 \times 10^{-9}$, $8.39 \times 10^{-9})$	2.21×10^{-8} $(2.15 \times 10^{-8}$, $2.27 \times 10^{-8})$	2.64×10^{-8} $(2.57 \times 10^{-8}$, $2.71 \times 10^{-8})$	2.53×10^{-8} $(2.46 \times 10^{-8}$, $2.60 \times 10^{-8})$
Cross section (undet. errors) cm^2	1.16×10^{-9} $(1.02 \times 10^{-9}$, $1.31 \times 10^{-9})$	1.39×10^{-11} $(0.29 \times 10^{-11}$, $4.07 \times 10^{-11})$	4.64×10^{-12} $(0.12 \times 10^{-12}$, $2.59 \times 10^{-11})$	1.95×10^{-10} $(1.41 \times 10^{-10}$, $2.64 \times 10^{-10})$	3.71×10^{-11} $(1.60 \times 10^{-11}$, $7.32 \times 10^{-11})$	0 (0, $1.71 \times 10^{-11})$

The results in Table II show that a bigger matrix suffers more errors. Both Data and Hang errors increase. The particular effect in the case of Hang errors is due to the much larger amount of memory accesses required to multiply bigger matrices. In the case of Data errors, the bigger the matrix, the higher the number of operations that must be performed to obtain a result, so the probability to get an error on the result increases. As a consequence, the cross section, considering all errors, increases with matrix size. However, for a given matrix size, the cross sections considering all errors of SWIFT and NEON-SWIFT approaches are similar. The unhardened cross section is also similar, except in the case of 32x32 matrix size for the reasons explained in the previous paragraph.

Both the SWIFT and the NEON-SWIFT versions are able to correct a significant amount of data errors. It is noticeable the contribution of the control data to the complete error detection figure in the case of NEON-SWIFT approach, which shows the lowest SDC error count for both matrix sizes. Moreover, the NEON-SWIFT hardened benchmarks present less undetected errors (SDC or Hang) than the conventional SWIFT approach.

The small number of undetected Hang errors in all benchmarks demonstrates the great contribution of trace monitoring to the effectiveness of the proposed approach. Finally, referring to errors that are only detected by the PDTC, it is not possible to confirm if they were actual errors or errors that only affected the detection logic. Anyhow, they are a low proportion and it is generally advisable to consider them as real errors in a conservative approach.

These results also demonstrate the high error detection and mitigation capabilities of the proposed approach. Fig. 6 shows the cross sections in a bar graph. For the 32x32 matrix multiplication benchmark, 99.94% error coverage is achieved and the cross section is reduced from 8.00×10^{-9} cm^2 to 4.64×10^{-12} cm^2. In the case of 128x128 matrix size, 100.00% error coverage is obtained, reducing the cross-section from 2.53×10^{-8} cm^2 to less than 1.71×10^{-11} cm^2. In comparison, the conventional SWIFT implementation also produces good error coverage when combined with trace monitoring, but it always produces more undetected errors and lower performance.

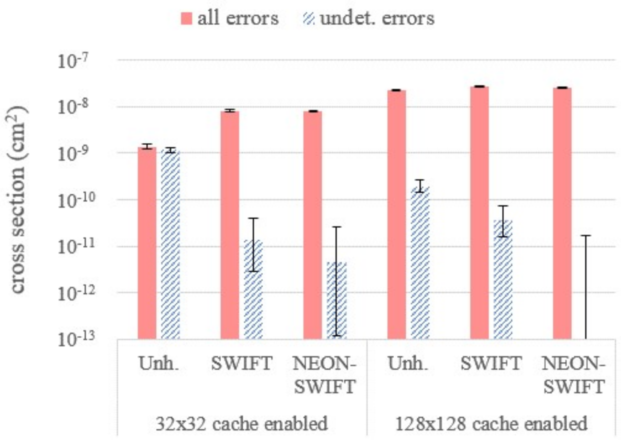


Fig. 6. Cross section bar graph

The data shown in Table II also demonstrates that both data errors and control-flow errors need to be addressed to achieve a high error coverage.

V. Conclusions

This work presents an error detection and correction approach for microprocessors that combines the use of the SIMD coprocessor and the trace interface. Data error correction is achieved by using software-implemented redundancy and the SIMD coprocessor is used to accelerate computation over redundant data by parallel processing. To increase the error coverage, a control data lane is added to each replicated data set and checked at synchronization points. Control data work as control signatures of the sequence of instructions executed on the data. This technique is complemented by Trace Monitoring to detect control-flow errors that are difficult to handle from a pure software approach, such as memory exceptions. To this purpose, we use an external IP connected to the trace interface that observes the program execution.

Experimental results with neutron irradiation demonstrate the high coverage of the proposed method. The neutron cross section of errors that were not corrected nor detected was reduced by more than three orders of magnitude. Such a reduction was possible because both data errors and control-flow errors were efficiently covered. Furthermore, the use of the SIMD coprocessor increases performance and error coverage with respect to a conventional SWIFT implementation.

The proposed approach is intended for data-intensive applications. For control-intensive applications, with many conditional changes in the control flow, a straightforward implementation may not produce significant performance benefits. In this case, code optimization and restructuring approaches need to be investigated to take full advantage of the SIMD co-processor.

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] S. Mukherjee, “Architecture Design for Soft Errors”, Elsevier, 2008.

[2] M. Rebaudengo, M. Sonza Reorda, M. Violante, “Software-level soft-error mitigation techniques,” in Soft errors in modern electronic systems, M. Nicolaidis (Ed.), Springer, pp. 253-285, 2011.

[3] N. Oh, P. P. Shirvani, and E. J. McCluskey, “Control-flow checking by software signatures,” IEEE Transactions on Reliability, vol. 51, no. 2, pp. 111–122, Mar. 2002.

[4] “Cortex A9 Technical Reference Manual” (r4p1), ARM Ltd., Cambridge, U.K., 2012.

[5] 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.

[6] M. Peña-Fernandez, A. Lindoso, L. Entrena, M. Garcia-Valderas, Y. Morilla, P. Martín-Holgado, “Online Error detection through trace infrastructure in ARM microprocessors”, IEEE Transactions on Nuclear Science, vol. 66, no:7, pp. 1457-1464, Jul. 2019.

[7] R. Vemu and J. A. Abraham, “CEDA: Control-flow error detection through assertions,” Proc. 12th IEEE Intl. On-line Testing Symp. (IOLTS), pp. 151–158, 2006.

[8] L. Parra et al., "Efficient Mitigation of Data and Control Flow Errors in Microprocessors," in IEEE Transactions on Nuclear Science, vol. 61, no. 4, pp. 1590-1596, Aug. 2014.

[9] N. Oh, P. P. Shirvani, and E. J. McCluskey, “Error Detection by Duplicated Instructions in Super-Scalar Processors,” IEEE Transactions on Reliability, vol. 51, vo. 1, pp. 63–75, Mar. 2002.

[10] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, “SWIFT: Software Implemented Fault Tolerance,” Proc. 3rd Intl Symp. on Code Generation and Optimization (CGO), pp. 243–254, Mar. 2005.

[11] G. A. Reis, J. Chang, and D. I. August, “Automatic Instruction-Level Software-Only Recovery,” IEEE Micro, vol. 27, no. 1, pp. 36–47, Jan. 2007.

[12] M. Peña-Fernández, A. Serrano-Cases, A. Lindoso, M. García-Valderas, L. Entrena, A. Martínez-Álvarez, S. Cuenca-Asensi, “Dual-Core Lockstep enhanced with redundant multithread support and control-flow error detection”, Microelectronics Reliability, vol. 100–101, article no. 113447, Sept. 2019.

[13] L. Parra et al., “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.

[14] 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, pp. 293-306, 2016.

[15] A. Lindoso, L. Entrena, M. García-Valderas and 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.

[16] A. Lindoso, M. Garcia-Valderas, L. Entrena, “Analysis of neutron sensitivity and data-flow error detection in ARM microprocessors using NEON SIMD extensions”, Microelectronics Reliability, vol. 100–101, article no. 113346, Sept. 2019.

[17] “Cortex™-A9 NEON™ Media Processing Engine, Technical Reference Manual”, (r3p0), ARM Ltd., Cambridge, U.K., 2011.

[18] “NEON Programmer’s Guide”, ARM Ltd., Cambridge, U.K., 2013.

[19] “CoreSight Architecture Specification”, ARM Ltd., Cambridge, U.K., IHI0029D, 2013.

[20] “CoreSight Program Flow Trace. Architecture Specification”, ARM Ltd., Cambridge, U.K., IHI0035B, 2011.

[21] “CoreSight Components. Technical Reference Manual”, ARM Ltd., Cambridge, U.K., DDI0314H, 2009.

[22] M. Peña-Fernandez, A. Lindoso, L. Entrena, M. Garcia-Valderas, Y. Morilla, P. Martín-Holgado, “PTM-based hybrid error-detection architecture for ARM microprocessors”, Microelectronics Reliability, vol. 88–90, pp. 925–930, 2018.

[23] “Zynq-7000 All Programmable SoC: Technical Reference Manual”, Xilinx Inc, document UG585, 2016.

[24] “Zybo Reference Manual”, Digilent Inc, Pullman, DC, USA, 2014.

[25] M. Peña-Fernandez, A. Lindoso, L. Entrena, M. Garcia-Valderas. "The Use of Microprocessor Trace Infrastructures for Radiation-Induced Fault Diagnosis". IEEE Transactions on Nuclear Science, vol. 67, no. 1, pp. 126-134, Jan. 2020.

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