Hybrid Lockstep Technique for Soft Error Mitigation

M. Peña-Fernández[1], A. Serrano-Cases[2], A. Lindoso[3], S. Cuenca-Asensi[2], L. Entrena[3], Y. Morilla[4], P. Martín-Holgado[4], and A. Martínez-Álvarez[2]

***Abstract*—This work presents the evaluation of a new dualcore lockstep hybrid approach aimed to improve the fault tolerance in microprocessors. Our approach takes advantage of modern multicore processor resources to combine a software-based lockstep with a custom hardware observer. The first is used to duplicate data and instruction flows; meanwhile, the second is in charge of the control-flow monitoring. The proposal has been implemented in a dualcore ARM microprocessor and validated with low energy proton irradiation and emulated fault injection campaigns. The results show an improvement of one order of magnitude in the cross-section of the benchmarks tested, even considering the worst-case scenario.**

***Index Terms*—soft errors, proton irradiation, dualcore, lockstep, multithreading, fault tolerance.**

I. Introduction

Commercial processors are becoming a commodity for the implementation of critical electronic systems in a multitude of industrial domains: from traditional aerospace and military sectors to emerging markets like high-performance computing, autonomous vehicles or medical appliances. The superior flexibility and performance offered by their advanced multicore architectures make those devices a promising alternative to other specifically designed circuits. Unfortunately, the progressive miniaturization of the electronic components jointly with the high clock frequencies demanded by new applications are making the cores more vulnerable to radiation-induced faults. Therefore, providing some kind of fault tolerance to the microprocessors, i.e., the ability to continue the operation even in the presence of faults, is a key to enable these devices for safety-critical applications.

This work is focused on the enhancement of processors fault tolerance to soft errors, i.e., transient faults on memory cells that eventually can lead to system failures. Traditionally the protection of microprocessors is addressed by means of spatial and/or temporal redundancy to detect and/or to correct radiation induced faults. Depending on how they are implemented, the techniques are usually categorized as hardware or software based. Hardware techniques are those based on the replication of processing by means of redundant hardware blocks: registers, memories or even entire processing units. Dual and Triple redundant Core lockstep (DCLS/TCLS) [1], [2] replicate the whole processor and compare the system output every clock-cycle to detect any mismatch during the execution of the code. TCLS in addition, offers the ability to recover the system using the third core state.

Generally, only output data are checked for errors [2], [3]. However, control-flow errors may cause one of the processors to lose synchronization and eventually hang or get lost. Control-flow errors are not easy to detect as they may not have an immediate observable effect in the computed data. Moreover, it is common in dual cores that one of the processors acts as primary and the other as secondary. In such a case, the hang of the primary can lead to the crash of the entire system. Software techniques are aimed to protect the code execution on unreliable hardware, mostly commercial off-the-shelf (COTS) devices. Similarly to the hardware techniques, they introduce replication at different software levels: programs, functions, loops, instructions, etc... Although their implementations have a lower impact on development costs when compared with hardware techniques, its application presents relevant overheads, in terms of performance and memory footprint, that should be taken into consideration.

Unlike related works, this proposal tries to reduce the impact of the unavoidable unreliable software to achieve a reliable and efficient computation. In fact, no operating system nor external threading libraries are used at all. Our approach exploits the multithreading capability of modern microprocessors by means of multiple instances of the same program running in parallel on separate cores and without any communication between them, excluding a little piece of code for stall and synchronization purposes.

Usually, mitigation techniques are conceived assuming that memory chips are protected with some kind of EDAC (Error Detection and Correction) mechanism. In our case, no assumptions are made and the proposal includes procedures to mitigate softerrors even for non-protected memories. In addition, errors that affect the control flow in any core can be detected by a custom hardware observer (IP observer) connected to the trace subsystem. This IP core decodes on-the-fly the program traces and checks the obtained information.

Our hybrid and multithreaded DCLS lockstep based approach has shown improvements in error mitigation, even for

[1]M. Peña-Fernandez is with Arquimea ADS, 28918 Leganés, Spain (e-mail: mpena@arquimea.com).

[2]A. Serrano-Cases and S. Cuenca-Asensi and A. Martínez-Álvarez(✉) are with the Computer Technology Department, University of Alicante, Carretera San Vicente del Raspeig s/n, 03690 Alicante, Spain (e-mail: aserrano@dtic.ua.es; sergio@dtic.ua.es; Corresponding author: amartinez@dtic.ua.es).

[3]A. Lindoso and L. Entrena are with the Electronic Technology Department, Universidad Carlos III de Madrid, Leganes, 28911 SPAIN (e-mail: alindoso@ing.uc3m.es, entrena@ing.uc3m.es).

[4]Y. Morilla and Pedro Martin-Holgado are with the Centro Nacional de Aceleradores (CNA), Centro Nacional de Aceleradores, CSIC, JA, Universidad de Seville, E-41092 Seville, Spain, SPAIN (e-mail: ymorilla@us.es, pmartinholgado@us.es).

applications with high resilience to radiation. The technique has been validated with low energy proton irradiation and tested by means of emulated injection campaigns.

The paper is organized as follows. Section II introduces the software and hardware mechanism combined in our proposal. Section II makes a review of works related to multhreading and lockstep mitigation techniques. Section III describes the fault injection campaigns performed and the previous results analysis to estimate the contribution of the technique to the system reliability. Section IV reports the radiation experiments and their results. finally, Section V summarizes the conclusions of this work.

II. Related works

The emergence of multicore processors has enabled the execution of multiple copies of the same instruction flow on separated execution units. The technique known as Redundant Muti-Threading (RMT) along with the concept of Sphere of replication was proposed in [4] for detection and recovery of soft errors. Basically, the SoR defines the set of resources, hardware or software, which are replicated. This way, the values entering the SoR must be replicated, and the values leaving the SoR must be checked to assure their integrity. Initial RMT approaches included the processor pipeline and the register file in the SoR boundaries, relaying the correctness of the execution on helper structures such as Store buffers, Load and Branch queues, not present on real processors [5], [6]. Those proposals were tested on simulators with promising results but never evaluated on real devices.

Other approaches deal with soft errors considering their effect on the software running on the system. They address the problem from either the compiler, operating system or application level. Most of them make the assumption that memories are protected by an error detection mechanism. In [7] and [8] a custom compiler transforms the application code into two communicating threads: the leading thread performs all the load/store operations and sends the data to the trailing thread which replicates the ALU operations and compare the results. The SoR only includes computations; therefore they suffer from the vulnerable input replication and output comparison processes. Another approach [9] suggests to duplicate the memory read/write operation values to solve this problem, however it increases the synchronization and performance overheads up to 5x given the low granularity of the memory operations.

Other authors [10] propose specific Operating System services to support RMT execution providing error detection and recovery. In that work, the OS service replicates transparently the execution of applications at binary level by creating redundant threads in separate address spaces. Some studies propose the use of standard libraries such as OpenMP and Pthreads [11] to generate redundancy programmatically. Furthermore, authors in [12] and [13] propose the use of custom API (Application Programming Interfaces) and language directives to allow the programmer to define redundant threads and selectively decide the code regions and variables to be protected. Also, software tools like Trikaya [14] have been proposed to automatically transform the source and produce a DMR version with rollback re-execution. All those high-level approaches suffer from two main problems. The first is the performance overhead produced by the additional software layers. The second is the increment in the susceptibility to errors, due to the complexity introduced by the software stack and the Operating System itself. That is expressed by a higher number of both, silent data corruption events and Operating System crashes as pointed in [15].

There are a lower number of approaches based on hardware redundancy as they usually apply architectural modifications to real devices or include custom modules in programmable SoCs. In [16] authors add a hardware module to a standard implementation of lockstep mechanism over two PowerPC processors hardwired on a FPGA. The module reduces the checkpoint overhead by comparing only the modified addresses and values. A different approach is presented in [17] where two FPGA based boards are used to implement a rollback recovery method to protect periodic tasks running on two LEON processors. Softcore processors are also employed in [18]. In this case, Microblaze processors are configured in DMR to detect errors in the application outputs, meanwhile a TMR Picoblaze continuously reads the configuration memory looking for errors. Finally, the work [2] implements a roll-back/recovery mechanism using the programmable resources of an FPGA. The application is manually divided into several blocks delimited by verification points and it is executed simultaneously in both cores of an ARM cortex-A9 processor. Every time a verification point is reached, the context and data are saved in a dual-port private memory and compared to detect some mismatch by a custom hardware.

In our approach A Single Program Multiple Data (SPMD) scheme has been adopted for bare metal applications (without OS) which renders a reduced number of race conditions and lower control overhead compared to traditional solutions. This technique is combined with a custom IP that leverages the information provided by the on-chip debugging facilities to detect on-the-fly any control flow error. It results in an efficient implementation with a very low area usage.

Among all the reviewed works, only a few were tested in accelerated radiation experiments [2], [11], [14]. A more comprehensive surveys about Multithreading and Lockstep based mitigation techniques can be found in the respective surveys [19] and [20].

III. HYBRID LOCKSTEP APPROACH

Our approach combines hardware and software techniques that exploit common resources available in modern microprocessors. It is intended to be directly applied to them with minimum additions. Figure 1 shows the architecture resources that supports the proposal. Two redundant threads run simultaneously in two separate cores of a multicore processor sharing the on-chip memory to store a single copy of the code and private copies of the data. A software infrastructure was developed to endow the dual core system with the ability of running redundant threads on bare metal. It is composed of three elements. First, a modified Board Support Package

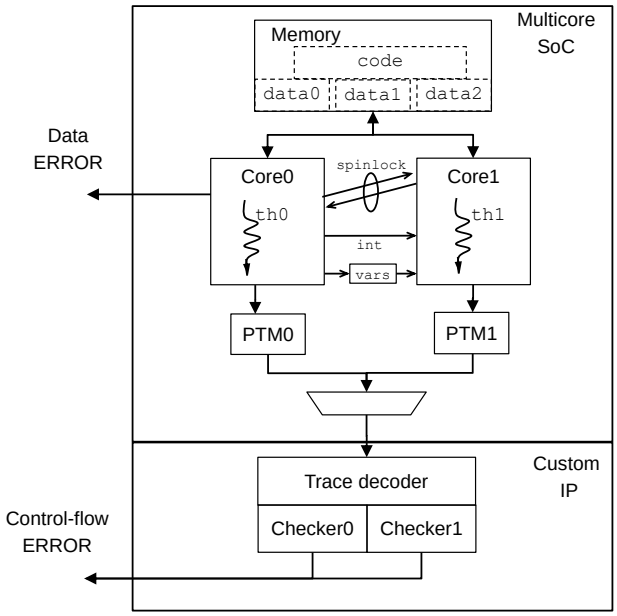

Fig. 1. Architecture overview.

(BSP) able to boot up the processor in SPMD mode. Second, a memory map and the associated linker scripts to build separate memory sections for each core. Finally, a thread support library that includes the synchronization and communication mechanisms, and the implementation of different macros and pre-processing directives to define the region of the code to be protected and the context that has to be restored in case of error.

The synchronization mechanism follows the spinlock method by means of exclusive load/store on shared variables (locks). Also, shared variables jointly with interrupts are used to implement a minimal communication to notify events between threads.

The on-chip trace modules, called PTM (Program Trace Macrocells), are used to extract execution information, one for each core. The trace information, containing program counter (PC) values related to the executed application of each core, is sent to a custom IP implemented in the programmable logic of the SoC, which is in charge of the decodification and monitorization of the cores activity.

On one hand, the software part of the technique is able to detect soft errors affecting the data during the computation. The detection of an error triggers a rollback recovery mechanism and can additionally be notified externally for further actions. The hardware IP, on the other hand, reports any anomalous behavior of the instruction flows. Using that information, specific recovery actions may be implemented.

A. Software implemented data protection

Taking advantage of the parallel computing that multiple cores can offer to improve data protection, the multithreaded mitigation technique called Duplication With Comparison and Re-Execution (DWC-R), first presented in [21], is also proposed and demonstrated under radiation here. This technique is based on the concept of Sphere-of-Replication (SoR) [22], which boundaries define the region of the code whose computation must be under protection. Within the SoR the instruction flow and the involved variables are replicated using lightweight threads. The verification of the correctness of data is only needed when an instruction sends the data outside the SoR. In any other case, the replicated threads progress in parallel working on their own data replica.

The implementation of the aforementioned technique requires some instrumentation of the source code (C or C++). This is undertaken by means of custom C++ macros and preprocessing directives, which are used to annotate what data variables belong to the SoR and thus, must be under protection; and the source code region (containing all read/write accesses to the SoR) where the mitigation will take place. The user just have to manually include the primitives in the original code and the compiler automatically produces the multithread version of the executable. The annotation of each data resource distinguishes the functional context of each variable, which means that those variables belonging to the .rodata (read-only data), .bss (uninitialized variables) and .data (initialized variables) data sections are indicated and processed conveniently.

Figure 2 shows an example of code instrumented with our technique. SoR boundaries are defined by the `SYNC` macro. It is also used to declare the context restoration and validation points by means of the variables that cross the SoR limits. All variables within the SoR (e.g. fooVar) are duplicated in its own core address space. Additional variables involved in the critical computation region (e.g. global variables) need to be explicitly replicated using the `XHARD` macro, which is overloaded depending on the memory section where the variables will be allocated. Each thread accesses to its copy using a pointer created and initialized with the `PTR` and `THREAD_REPLICA_VAR` primitives respectively.

Threads operation and synchronization can be seen in Fig. 3. In order to achieve detection and recovery capability, the SoR is triplicated by means of the replication of each data section. Two copies of each data section (.bss0, .bss1, .data0, .data1, .rodata0 and .rodata1) are automatically generated when spawning the two threads (primary and shadow). The corresponding addresses for each data section are automatically managed by a custom linker script. In addition, DWC-R allocates a third copy of data sections (.bss2, .data2 and .rodata2) to allow the restoration of the SoR variables in case of error.

When the program starts, two bare-metal threads (primary and shadow) are spawned in parallel using both independent shared memory processing units (Core0 and Core1) to replicate the full program computation. In case the program enters a protected region the threads are synchronized using a spinlock barrier, then the context is automatically updated by the primary thread (SoR-input checkpoint), which notifies to the shadow to start the computation of the critical region. Next a new synchronization is needed to guarantee that both threads have finished the computation before the SoR output checkpoint is reached. At this point, SoR computed variables are compared by the primary thread. In case of mismatch, both threads go back to the first checkpoint and performs

```
// Processing directives definition
#include "MultiThreadingHardening.h"

// XHARD: replaces the classical definition
// of variables by 3 copies allocated in
// the X section:
// -> BSS   : Not initialized data
// -> DATA  : Initialized data
// -> RODATA: Read only memory
//
// - char fooGlobalVar;
XHARD(char, fooGlobalVar)

// PTR: creates a pointer to each var copy
// The pointer has to be set by each core
#define fooGlobalVar PTR(fooGlobalVar)

void foo(int autoVar){
    int entryVar;
    int exitVar;
    ...
    // SYNC: defines the SoR entry point
    // and the context creation/restoration
    // point by means of the entry vars
    SYNC(entryVar, ...)

    // THREAD_REPLICA_VAR: sets the pointer
    // to fooGlobalVar copy. Each core
    // selects the copy using its coreID
    THREAD_REPLICA_VAR(char, fooGlobalVar)
        ...
        ///  Start of critical computation
        ...
        int fooVar = 4;
        fooGlobalVar = 3 << fooVar;
        exitVar = fooGlobalVar * entryVar;
        ...
        /// End of critical computation
    // SYNC: defines the SoR exit point
    // and the context validation by means
    // of exit vars
    SYNC(exitVar)

    // Non critical computation
    ...
}
```

Fig. 2. Snippet of C/C++ code instrumentation.

a context restoration (using the third copy) to re-execute the protected section. Conversely, the program continues the normal execution flow. It is worth mentioning that the minimal needed code instrumentation (green boxes) does not interfere with the original program flow.

B. Hardware implemented control-flow protection

To provide control-flow protection, we leverage the information available at the trace interface of the processor. The trace interface is a very common resource in modern microprocessors, enabling debug and profiling tasks during application development. However, the trace interface is usually left unused once the application is released, so it can be reused for other purposes. The trace interface is, by design, capable of providing relevant information about processor execution

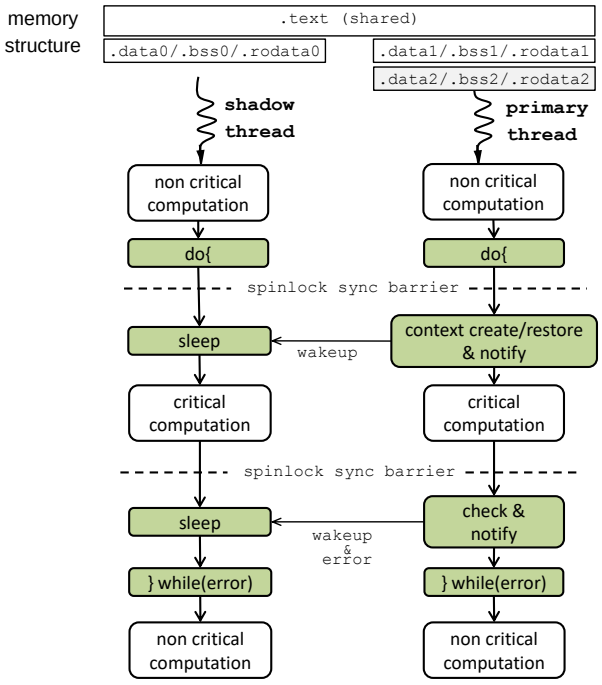

Fig. 3. Duplication With Comparison and Re-Execution (DWC-R) with 2 threads. Code program (.text memory section) is shared among both threads, while .data, .bss and .rodata memory sections are distributed.

without disturbing it, in a non-intrusive manner and with low latency. In a multicore system, the trace interface is typically shared between all cores, and it can be effectively used to gather execution information of all of them.

A custom observer IP core has been developed based on the trace interface specification and implemented in VHDL [23]. The IP can receive raw data packets from the trace interface, decode them and use that information to check the correctness of the execution flow of one or more processors in the system. The decoding and checking processes are performed online along with processor execution and the latency of the IP is less than 30 clock cycles to determine whether an error has occurred. The time that the trace takes to output the information about execution is also known to be low [24] so the overall detection latency can be as low as 500ns in a typical implementation.

The Program Trace Macrocells (PTM0 and PTM1) available at the trace subsystem of the SoC processing system have been configured to export trace information related to the program counter (PC) value of each core during execution. The amount of information exported by the PTM modules is not exhaustive, since it is compressed to optimize the bandwidth of the trace interface. However, it is enough to infer the execution flow of each core, as it includes information about all the branches taken by each processor. In the absence of branch information in the trace, sequential execution is assumed. The observer IP leverages trace information to gather the Program Counter (PC) value of both processors in the system and continuously check that value against a set of allowed ranges configured

by the user, where the application code is stored. If at any moment any of the processors present a PC value outside the allowed ranges, then the processor control flow is wrong and the application is about to fail, so the IP sets a signal to trigger a system reset with much lower latency than a common watchdog. In addition, the PC value is used to perform an additional check by the use of a PC-based watchdog. For the PC-based watchdog, the IP is continuously looking for a specific user-configurable PC value in the trace information. If the configured PC value does not appear in a configurable time because of a possible hang condition, the IP sets a signal indicating it. The PC-based watchdog has the advantage over a traditional watchdog that it is not relying at all in the application to reload it, but it will be reloaded using the trace information every time the application reaches a particular point in execution, which can be commonly set as the first instruction of the main application loop.

IV. SOFT ERROR MITIGATION ANALYSIS

To assess our hybrid approach two fault injection campaigns were carried out. The first campaign was designed to observe the impact of the software-based part of the technique (Multi-threading DWC-R) applied at different levels of granularity. The second was conceived to estimate the overall contribution of the complete hybrid technique to the applications reliability.

The matrix multiplication code of the project BEEBS [25], was selected as benchmark to operate on 20×20 32-bit integer matrices. The structure of the code, basically three nested loops, allowed to analyze the trade-offs between the granularity of the protection and the reliability obtained. The index of the first loop (the outer loop) runs through the rows of the first matrix. Defining here the boundaries of the SoR means a unique point of checking but a large amount of data to be verified (the whole matrix). The second index runs through the columns of the second matrix (inner loop). This boundary defines multiple checkpoints to verify the correctness of each resultant row. Finally, the innermost loop computes the multiplication of one row and one column to obtain the corresponding element of the result matrix. At this point every element must be checked increasing the number of synchronizations between threads but significantly reducing the amount of data to be verified. The output matrix is initialized to zero on each run, and a golden matrix is used to verify correctness. The output initialization ensures that the whole computation is performed and committed to memory, detecting possible masked errors due to intermediate cached calculations.

A. Simulated fault injection campaign

In the multithreading DWC-R analysis, we employed an instruction accurate simulator, WindRiver Simics [26], configured to inject bit-flips in the register files and memory sections of a ARM Cortex-A9 dual-core processor. Besides the unprotected version of the code (Original), four hardened versions were built to run on the dual-core: *StackH*, *DatH-Outer*, *DatH-Inner* and *DatH-Acc*. In the *StackH* version the matrix multiplication is protected using two threads, one per processor core, and triplicating the automatic variables. This benchmark performs the same computation as the code without protection, except for the inner-loop checks performed to the automatic variables. These checks aim to verify that the computation is being performed with the same data, and are achieved by checking the indexes that control the loops. Once each value from the output matrix has been calculated, the result is verified with the other thread output before committing the result to memory and exiting the SoR.

Finally, in the Dat-x versions, the SoR is extended from the calculation of each element of the resulting matrix (*DatH-Acc*) to the data residing on memory, resulting in the triplication of all involved matrices (*DatH-Outer*). In this case, the verification is carried out once each core has calculated the output matrix, resulting in a coarse grain check. At this point, only two copies of the output matrix are completed. Therefore, to complete the data triplication, the calculated matrices are compared and saved into the third copy if no errors are found. Otherwise, the matrices are restored to the initial state to restart the calculus from an error-free state.

The fault injector was configured to perform 1800 injections per core at the register file ($100 \cdot 18$ registers), 800 injections per memory section and core (200 injections per data replica at .rodata, .data, .bss and .stack sections) and 200 injection at .text section per core. Thus, 5200 injections of faults have been injected at the single-core original version and 10400 at multithreaded DWC-R versions. Faults were labeled as unACE (unnecessary for Architectural Correct Execution) when an injection is made, and they do not affect the result of the program's output, SDC (Silent Data Corruption) when the result is not correct but the program ends, and HANG if it does not end or exceeds a time limit. The programs have been evaluated having as a reference a faultless (ground truth) execution of themselves and adding a recovery time equal to the faultless execution duration. If this temporal restriction is exceeded, the program is considered that does not meet the valid requirements and the fault is labeled as HANG.

Raw event rates (SDC and HANG) from simulated fault injection campaign are shown in Figure 4. Results demonstrate that the unprotected version presents a high rate of SDC, above 30%, and a very low of percentage of HANG. These results are in accordance with the data intensive nature of the algorithm. This way, the successive muti-threading versions clearly decreases the SDC occurrence depending on the amount of data protected and the granularity of the checkpoints. The *StackH* version only protects the automatic variables (the indexes of the loops), so it gets a modest improvement of 2× but at the cost of increasing the HANG rate up to 8×. The *DatH-Acc* version protects every result individually, therefore it offers the best SDC rate (0.8%) improving the baseline rate by 38×. However, it involves a high number of checkpoints and threads synchronization which makes the code more prone to control flow errors. As a consequence the HANG rate is still increased by 5.7×. The *DatH-Outer* version presents the most balanced results, due to the low control overhead, with a very reduced SDC percentage (1%, i.e. improvement of 29.7×) and the lower HANG rate at the same time (1.8%). In terms of unACE faults it reaches up to 96.1%.

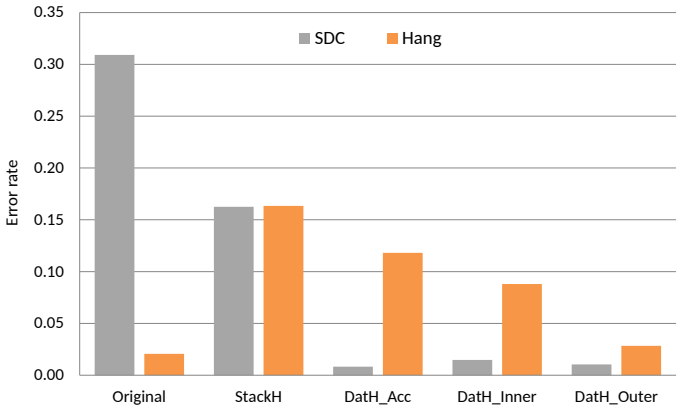

Fig. 4. Error rates for multithreading DCW-R technique.

B. Emulated fault injection campaign

The analysis of the proposed technique (data and control flow protection) was complemented with one additional fault injection campaign over those versions showing the most uneven behavior, i.e. *StackH* and *DatH-Outer*. The faults were emulated on a Zybo Board [27] connected to an external host (single-board computer) which was in charge of generating random seeds for injection and retrieving test results. In the event of any error during fault emulation, the external host power cycled the Zybo board. The campaigns were focused on the injection on the register file and are triggered randomly inside the processor itself by software emulated upsets [28] using timer interrupts. Upon each timer interrupt, the current state of the register file is saved on the processor stack before attending the interrupt service routine. Inside the routine, the software randomly introduces a bit-flip in a random bit of a random register of the saved stack state and returns. When the content of the stack is restored into the processor registers, the injected fault becomes effective. Only one fault is injected per benchmark execution, following a single error model, and in the case that re-execution is needed, no additional faults are injected.

The results obtained with this emulated fault injection campaign are shown in the Table I. In addition to the recovery capability exposed by the multithread part of the technique, the control-flow protection is able to identify conditions that would lead the dual-core system to lose synchronization and get stuck, such as the triggering of an non-managed exception. The observer IP can detect that events, which commonly cannot be corrected, with low latency to trigger a system reset, thus increasing overall availability of the system. A new category, named Potential Hang Detected (*PotHDetected*), is used to classify those events. In addition, the label Corrected was assigned to cases where data errors where detected and recovered. Columns of Table I show the number (*N*) and its relative percentage (*%*) of *TotalEvents* (total amount of faults that were able to be labeled), *HANGs*, *PotHDetected*, *SDC* and *Corrected* events for the *Original*, *StackH* and *DataH-Outer* versions of the software. In addition, a third column (*% TotalInj*) indicates the percentage of the labeled faults taking into account the total number of emulated faults.

The campaigns were run up to get a significant number of events (about one hundred). As expected, the Original benchmark presents the lower reliability in terms of SDC and, in this case no HANGs were produced. Similar to simulated campaigns *StackH* version improves the SDC and presents an important number of Corrected events even higher than *DatH-Outer* benchmark. Some specific register are exclusively used to access the stack (were the automatic variables are stored), and the majority of the faults are detected and corrected by the technique. On the contrary, *DatH-Outer* uses massively the memory to operate with the data which may explain the difference in terms of corrected errors, since the register file is the only target of the faults injected in this campaign. In summary, the hardware implemented control-flow protection provides an important reduction in the error rate associated to HANG events, without interfering the recovery capability of the data protection.

TABLE I
EMULATED INJECTION RESULTS

	Original			StackH			DatH-Outer		
	N	%	% TotInj	N	%	% TotInj	N	%	% TotInj
TotalEvents	101	100	2.70	817	100	0.94	122	100	0.45
HANG	0	0	0.00	0	-	-	0	-	-
PotHDetected	-	-	-	42	5.14	0.05	100	81.97	0.36
SDC	101	100	2.70	59	7.22	0.07	3	2.46	0.01
Corrected	-	-	-	716	87.64	0.82	19	15.57	0.07

TABLE II
TIME OVERHEADS AND RELATIVE IMPROVEMENT OF THE MWTF METRIC

	Original	StackH	DatH-Outer
Execution Time	613,526	1,808,117	1,544,784
Relative Execution Time	1×	2.9×	2.5×
Relative MWTF	1×	13.5×	97.7×

Finally, in order to estimate the overall reliability improvement of our hybrid approach, the Mean Work To Failure metric is provided. MWTF takes into account not only the fault coverage but also the period of time that the code is exposed to faults. Table II show the execution time of each benchmark and the relative MWTF, taking the original unprotected code as baseline. The protected codes present time overheads of 2.9× and 2.5× for *StackH* and *DatH-Outer* respectively. In the case of protecting the stack, the technique produces a large number of checking points and threads synchronization, which is the main cause of the shown overhead. On the contrary, the coarser granularity protection of *DatH-Outer* reduces notably the number of checkpoints, but introduces a large number of memory accesses to verify and restore the matrices which explains the excess in the execution time. Even though the performance penalty incurred by our proposal, the gain in MWTF is remarkable being of one order of magnitude when only the stack is hardened and reaching up to 97.7× when all data are triplicated.

V. RADIATION EXPERIMENTS

The Original and DatH-Outer benchmarks have been tested in the external beamline of the 18/9 Ion Beam Applications compact cyclotron located at *Centro Nacional de Aceleradores* (CNA) in Seville, Spain. The device under test (DUT) was a Xilinx Zynq-7010 SoC (System on a chip) [29] that integrates a hard-core dual-core ARM Cortex-A9 processor [30] along with programmable logic, interconnections and peripherals. The DUT was mounted on a commercial board (Zybo) [27] and irradiated in open air with 15.2 MeV protons. The energy of the protons in the active area of the silicon is about 10MeV, which is considered enough to produce single event effects on the 28nm technology device with no thinning [31].

The DUT configuration is stored in an SD card, which is inserted in the receptacle of the board. Upon power on, the DUT loads the code from the SD card to On-Chip-Memory (OCM) using a two-stage bootloader to initialize the programmable logic and boot the application. Because we use OCM for the benchmarks, we used a two-stage bootloader scheme. It is important to mention that all the benchmarks are executed using only OCM memory, which is inside the SoC, so all computing hardware including the memory is irradiated.

The external observer IP has been implemented in the programmable logic of the device and connected to the trace interface over the Extended Multiplexed Input Output (EMIO) interface available on Zynq device. The IP leverages the information produced by the Program Trace Macrocell (PTM) of each core, which provides relevant PC values during execution. The processors are running at nominal 650MHz clock frequency.

An external host, placed outside the beam and connected to the DUT through a serial communication interface, was used to control the experiment. The benchmarks provide a periodical message if no error is present and the code is instrumented to provide different codes depending on the observed error. In the case of any error, the host performs a power cycle to restart the DUT.

We distinguish the following error categories:

- Exception error. The processor execution flow has been abruptly interrupted by an unexpected exception, probably caused by a forbidden memory access. If not handled, this type of errors would become timeout errors.
- Timeout error. The processor has become unresponsive.
- Communication error. The serial communication with the processor has become corrupted, thus making impossible to identify further errors.
- Silent Data Corruption (SDC). The benchmark execution has finished with errors in the result matrix.

To test our technique under the worst-case scenario, all program memory sections (.rodata, .data, .bss and .text) were statically linked within the executable, loaded to the OCM, and therefore exposed to the beam. This way, the boundaries of the SoR were extended up to the on-chip-memory. To do this it was necessary to insert routines to verify and restore the three copies of the input data. Also included in the .rodata section were three copies of the golden results as well as routines to periodically refresh their values to avoid the accumulation of errors between successive runs of the benchmark.

The results obtained from the irradiation campaign are presented in Figure 5. It shows the cross-section of the aforementioned error classification, and the cross-section of all observed errors (Total Errors) for the benchmarks tested under radiation. Note that even the algorithm without protection (Original), which represents our starting point, presents high resilience to radiation (a low cross-section). However, our technique is able to improve the cross-section and, therefore, the vulnerability to soft errors.

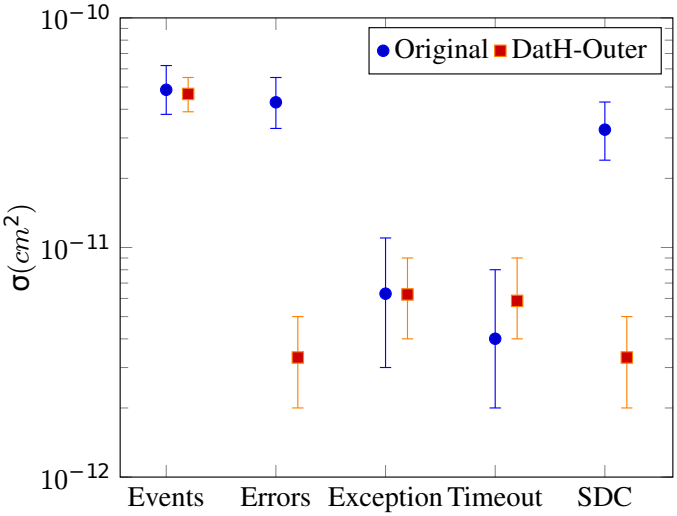

Fig. 5. Cross-section (cm^2) and 95% confidence intervals for DatH-Outer and unprotected Original benchmarks.

As expected, the cross-section of the Original benchmark throws the worst results in terms of Total Errors and SDC. Note that our technique is able to detect the Exception and Timeout events before they become errors, so they are not accounted in the Total Errors category for the DatH-Outer experiment. It results in 4.29e-11 cm^2 cross-section for the unprotected benchmark and 3.31e-12 cm^2 for the DatH-Outer, showing an improvement of one order of magnitude. It is worth noting that our proposal was tested under the worst-case scenario, thus presumably further improvements would be obtained using EDAC protected memories.

Remarkably, most errors are tagged as SDC due to the different matrices' corruption (input, calculated, and golden). It is remarkable that this protection can reduce the propagation of erroneous outcomes (SDC). More precisely, during the irradiation campaign, the technique was able to correct up to 144 errors, which is a good demonstration of the mitigation capabilities of the proposed technique.

Regarding the timeout errors, results show that the hardening approaches are more prone to hang the platform, what can be caused by one of the threads missing synchronization. In such a case, the system can enter an abnormal waiting state where both cores are waiting for each other to finish and cannot synchronize. Although undesirable, it can be considered less critical than SDC, since this situation may be detected with common processor mechanisms (e.g. watchdog).

VI. Conclusion

We have presented a new hybrid soft error mitigation technique for multicore processors based on multithreaded lockstep and a custom hardware IP that uses the trace port of the microprocessor to observe the control-flow. The technique has been validated with low energy proton irradiation and tested by means of emulated injection campaigns. Both campaigns show insights of reliability improvements. In one hand, fault injection campaigns have demonstrated the detection and recovery capabilities of the proposed approach. On the other hand, the irradiation campaign has validated the reliability improvements observed in the analysis of the fault injection results. Therefore, error mitigation is improved by using our hybrid multithreaded lockstep based approach for soft error mitigation.

Acknowledgement

This work has been supported in part by the Spanish Ministry of Science and Innovation under the projects PID2019-106455GB-C22, PID2019-106455GB-C21; and by the Community of Madrid under grant IND2017/TIC-7776.

References

[1] X. Iturbe, B. Venu, E. Ozer, and S. Das, "A triple core lock-step (TCLS) ARM® cortex®-r5 processor for safety-critical and ultra-reliable applications," in *2016 46th Annual IEEE/IFIP Int. Conference on Dependable Systems and Networks Workshop (DSN-W)*. IEEE, Jun. 2016, pp. 246–249.

[2] A. B. de Oliveira, G. S. Rodrigues, F. L. Kastensmidt, N. Added, E. L. A. Macchione, V. A. P. Aguiar, N. H. Medina, and M. A. G. Silveira, "Lockstep dual-core ARM a9: Implementation and resilience analysis under heavy ion-induced soft errors," *IEEE Transactions on Nuclear Science*, vol. 65, no. 8, pp. 1783–1790, Aug. 2018.

[3] F. Abate, L. Sterpone, and M. Violante, "A new mitigation approach for soft errors in embedded processors," *IEEE Transactions on Nuclear Science*, vol. 55, no. 4, pp. 2063–2069, Aug. 2008.

[4] S. K. Reinhardt and S. S. Mukherjee, "Transient fault detection via simultaneous multithreading," *SIGARCH Comput. Archit. News*, vol. 28, no. 2, p. 25–36, may 2000.

[5] J. C. Smolens, B. T. Gold, B. Falsafi, and J. C. Hoe, "Reunion: Complexity-effective multicore redundancy," in *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, 2006, pp. 223–234.

[6] J. F. Martinez, E. Ipek, C. LaFrieda, and R. Manohar, "Utilizing dynamically coupled cores to form a resilient chip multiprocessor," in *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*. Los Alamitos, CA, USA: IEEE Computer Society, jun 2007, pp. 317–326.

[7] K. Mitropoulou, V. Porpodas, and T. M. Jones, "Comet: Communication-optimised multi-threaded error-detection technique," in *2016 International Conference on Compliers, Architectures, and Sythesis of Embedded Systems (CASES)*, 2016, pp. 2.3.1–2.3.10.

[8] Y. Zhang, J. W. Lee, N. P. Johnson, and D. I. August, "Daft: Decoupled acyclic fault tolerance," in *2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2010, pp. 87–97.

[9] H. So, M. Didehban, Y. Ko, A. Shrivastava, and K. Lee, "Expert: Effective and flexible error protection by redundant multithreading," in *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2018, pp. 533–538.

[10] B. Döbel and H. Härtig, "Can we put concurrency back into redundant multithreading?" in *Proceedings of the 14th International Conference on Embedded Software*, ser. EMSOFT '14, Art. no. 19. Association for Computing Machinery, 2014.

[11] G. Rodrigues, F. Rosa, A. de Oliveira, F. L. Kastensmidt, L. Ost, and R. Reis, "Analyzing the impact of fault tolerance methods in ARM processors under soft errors running linux and parallelization APIs," *IEEE Transactions on Nuclear Science*, vol. 64, no. 8, pp. 2196–2203, 2017.

[12] D. P. e. a. Hukerikar S., Teranishi K., "Redthreads: An interface for application-level fault detection/correction through adaptive redundant multithreading," *International Journal of Parallel Programming*, vol. 46, p. 225–251, 2018.

[13] Y.-S. Chen and P.-S. Chen, "A software-based redundant execution programming model for transient fault detection and correction," in *2016 45th International Conference on Parallel Processing Workshops (ICPPW)*, 2016, pp. 66–71.

[14] H. Quinn, Z. Baker, T. Fairbanks, J. L. Tripp, and G. Duran, "Software resilience and the effectiveness of software mitigation in microcontrollers," *IEEE Transactions on Nuclear Science*, vol. 62, no. 6, pp. 2532–2538, Dec. 2015.

[15] J. S. Monson, M. Wirthlin, and B. Hutchings, "Fault injection results of linux operating on an FPGA embedded platform," in *2010 Int. Conference on Reconfigurable Computing and FPGAs*. IEEE, Dec. 2010, pp. 37–42.

[16] F. Abate, L. Sterpone, C. A. Lisboa, L. Carro, and M. Violante, "New techniques for improving the performance of the lockstep architecture for sees mitigation in fpga embedded processors," *IEEE Transactions on Nuclear Science*, vol. 56, no. 4, pp. 1992–2000, Aug. 2009.

[17] M. Violante, C. Meinhardt, R. Reis, and M. Sonza Reorda, "A low-cost solution for deploying processor cores in harsh environments," *IEEE Transactions on Industrial Electronics*, vol. 58, no. 7, pp. 2617–2626, 2011.

[18] H.-M. Pham, S. Pillement, and S. J. Piestrak, "Low-overhead fault-tolerance technique for a dynamically reconfigurable softcore processor," *IEEE Transactions on Computers*, vol. 62, no. 6, pp. 1179–1192, 2013.

[19] I. Oz and S. Arslan, "A survey on multithreading alternatives for soft error fault tolerance," *ACM Comput. Surv.*, vol. 52, no. 2, art. no. 47, pp. 1–38, mar 2019.

[20] E. W. Wächter, S. Kasap, X. Zhai, S. Ehsan, and K. McDonald-Maier, "Survey of lockstep based mitigation techniques for soft errors in embedded systems," in *2019 11th Computer Science and Electronic Engineering (CEEC)*, 2019, pp. 124–127.

[21] A. Serrano-Cases, F. Restrepo-Calle, S. Cuenca-Asensi, and A. Martínez-Álvarez, "Multi-threaded mitigation of radiation-induced soft errors in bare-metal embedded systems," *Journal of Electronic Testing*, vol. 36, pp. 47–57, Dec. 2019.

[22] G. Reis, J. Chang, N.Vachharajani, R. Rangan, and D. August, "Swift: Software implemented fault tolerance," in *Proc. Int. Symposium on Code Generation and Optimization*, 2005, pp. 243–254.

[23] M. Peña-Fernandez, A. Lindoso, L. Entrena, and 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.

[24] 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 Transactions on Nuclear Science*, vol. 66, no. 7, pp. 1457–1464, July 2019.

[25] J. Pallister, S. J. Hollis, and J. Bennett, "BEEBS: Open Benchmarks for Energy Measurements on Embedded Platforms," *CoRR*, vol. abs/1308.5174, 2013.

[26] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," *Computer*, vol. 35, no. 2, pp. 50–58, 2002.

[27] *Zybo Reference Manual*, Pullman, WA, USA, 2014.

[28] M. Peña-Fernandez, A. Lindoso, L. Entrena, M. García-Valderas, S. Philippe, Y. Morilla, and P. Martin-Holgado, "Ptm-based hybrid error-detection architecture for arm microprocessors," *Microelectronics Reliability*, vol. 88, pp. 925–930, 2018.

[29] *Zynq-7000 All Programmable SoC: Technical Reference Manual*, San Jose, CA, USA, 2016.

[30] *Cortex-A9 Technical Reference Manual r4p1*, Cambridge, U.K., 2012.

[31] 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.