Novel Lockstep-based Approach with Roll-back and Roll-forward Recovery to Mitigate Radiation-Induced Soft Errors

Server Kasap, Eduardo Weber Wächter, Xiaojun Zhai, Shoaib Ehsan and Klaus D. McDonald-Maier

School of Computer Science and Electronic Engineering
University of Essex
Colchester, UK
{server.kasap, eduardo.wachter, xzhai, sehsan, kdm}@essex.ac.uk

***Abstract*—An attractive option for realizing applications in radiation environments is to employ All-Programmable System-on-Chips (APSoCs) thanks to their high-performance computing and power efficiency merits. Despite APSoC's advantages, like any other electronic device, they are prone to radiation effects. Processors found in APSoCs must, therefore, be adequately hardened against ionizing-radiation to become a viable alternative for harsh environments. This paper proposes a novel triple-core lockstep (TCLS) approach to secure the Xilinx Zynq-7000 APSoC dual-core ARM Cortex-A9 processor against radiation-induced soft errors by coupling it with a MicroBlaze TMR subsystem in Zynq's programmable logic (PL) layer. The proposed strategy uses software-level checkpointing principles along with roll-back and roll-forward mechanisms (i.e. software redundancy), and hardware-level processor replication as well as checker circuits (i.e. hardware redundancy). Results of fault injection experiments show that the proposed solution achieved high soft error security by mitigating about 99% of bit-flips injected into both ARM cores' register data.**

***Index Terms*—Lockstep, Reliability, Fault Tolerance, Soft Error Mitigation, Zynq APSoC, ARM Cortex-A Processor, MicroBlaze Processor**

I. Introduction

Cleaning up the toxic radioactive waste is one of Europe's most critical and complicated environmental remediation projects, which is estimated to cost £220bn over the next 120 years [1]. Owing to the extreme adverse effects of ionizing radiation on biological tissues, cleaning up radioactive waste inside a nuclear power plant is very risky for humans. Consequently, employing robots in radiation environments such as nuclear power plants and nuclear waste disposal sites is highly motivated and desirable. Electronic circuits in these robots, however, are still vulnerable to radiation effects. Henceforth, if robots are to be deployed in these situations, the radiation effects on electronic circuits must be significantly mitigated, particularly for operations control processors.

Energetic particles (e.g. alpha particles) or electromagnetic waves (e.g. gamma rays) striking transistor semiconductor substrates in radiation conditions cause transient error (soft errors) errors in electronic circuits [2]. Several mission-critical applications have recently been introduced in All-Programmable Systems-on-Chips (APSoCs), which combine a programmable logic (PL) layer with embedded processors in the processing subsystem (PS) layer. Unfortunately, these highly-integrated circuits, involving a variety of processor cores, are susceptible to transient errors. Soft errors impact processors by corrupting values stored in their memory elements, such as registers, cache, data and instruction memories, that may cause the processor to run an operation inaccurately, resulting in silent data corruptions (SDCs) or functional interruptions (FIs), i.e. hangs and crashes, in the device. Therefore, implementing strategies to minimize transient faults caused by radiation is essential to the implementation of APSoCs in radiation environments.

In this study, we have adapted a fault mitigation technique, i.e. triple-core lockstep technique (TCLS), to improve the reliability and availability of the dual-core ARM Cortex-A9 processor embedded in the Xilinx Zynq-7000 APSoC. The proposed TCLS method combines the two ARM cores in the PS with one MicroBlaze core implemented in the PL to replicate the same application execution. The technique incorporates software-level checkpoint and roll-back / roll-forward operations to provide reliability. During checkpoints, fault-free copies of processor core states are stored in safe memories, while roll-back and roll-forward operations are fault recovery mechanisms that restore a processor core to a previous safe state or to the current safe state of the other core that happens to be healthy [3]. In a lockstep-based technique, this is the first time a MicroBlaze core is coupled with hard-core ARM processors to support roll-forward recovery along with roll-back recovery, which improves system efficiency in terms of Mean Workload Between Failures (MWBF).

Furthermore, fault injection experiments that non-intrusively simulate bit-flips in ARM register files were conducted to evaluate the fault mitigation effectiveness and performance of our proposed TCLS technique. Experiments indicate that the TCLS approach applied to the dual-core ARM Cortex-A9 processor will mitigate about 99% of the bit-flips injected, while maintaining a timing overhead of as low as 25% under fault-free conditions.

Section II summarizes the effects of radiation on processors in particular, and briefs on an existing fault-mitigation technique, i.e. the lockstep technique. Section III elaborates on the proposed TCLS approach, while Section IV describes the fault injection mechanism employed during validation experiments. Experimental results are subsequently evaluated in section V, while conclusions and future plans are drawn in section VI.

II. Background

This section motivates the need for soft error protection by addressing soft error effects in processors. The lockstep technique is then introduced and clarified as a highly effective strategy for fault tolerance.

A. *Effects of Soft Errors in Processors*

Single-Event Upsets (SEUs) [4] can easily influence a processor's data-flow and control-flow, which is a major concern for security-critical applications. Upsets in values stored in memory elements can result in data-flow errors caused by incorrect operations or data manipulation. The execution of an incorrect operation occurs when the program code is corrupted by a bit-flip, leading to an incorrect instruction. Furthermore, if the bit-flip affects data used as a process input, outputs of that process will most likely be incorrect. For both types of data flow faults, the application's final outputs will be unreliable, known as SDC.

If the SEU affects the control-flow, the processor may execute the program incorrectly, causing either an application crash or a processor to hang that is classified as FIs. Control-flow upheavals can lead to branch errors, such as incorrect branch creation or deletion and incorrect branch decisions. The wrong creation is due to a bit-flip that sets a non-branch instruction to a branch, which leads the program flow to a wrong address, whereas the wrong deletion occurs when the branch instruction is converted to another instruction, so the proper branch can not be taken. A bit-flip in a conditional branch may also result in an incorrect decision on whether or not to take a branch. Besides, if the program counter (PC) is affected by a soft error, the program flow will be disrupted.

B. *Lockstep Technique*

Hybrid fault-tolerance techniques are those that use the software-implemented hardware fault tolerance (SIHFT) method combined with a hardware Intellectual Property (IP) that performs consistency checks in the processor, making them effective against both SDCs and FIs. For example, lockstep is a hybrid fault-tolerance technique based on software and hardware redundancy. It uses software-level concepts of checkpointing and recovery mechanisms (e.g. roll-back recovery, roll-forward recovery) and hardware-level replication and checker circuits.

Typically, lockstep technique works by running the same application simultaneously and symmetrically in identical processors that are initialized to the same state, with identical code and data inputs, during system start-up. As mentioned above, there is an IP, i.e. a checker module, which monitors the processors and compares their status periodically to check for inconsistencies within the lockstep system. To facilitate the comparison process, the verification points are inserted into the application program to indicate when the execution of the application must be locked and the state must be compared. If there is no discrepancy between processor states, processors are assumed to be fault-free and a checkpoint operation is performed; otherwise, the lockstep system restores the processors to a healthy state through a recovery mechanism, e.g. roll-back operation. At the end of the roll-back operation, processors would be recovered to a safe, error-free state and then restart the execution of their application from that point, potentially wasting valuable run-time. The most significant advantage of the lockstep technique is its ability to detect and correct both SDCs and FIs, as opposed to many other fault-tolerance techniques.

Several researchers have designed and implemented their version of the lockstep technique, such as those in [5]–[9], as reported in [10]. As a remarkable piece of work, Oliveira et al. proposed a dual-core lockstep (DCLS) fault-tolerance technique to mitigate radiation-induced faults in ARM-A9 processors embedded in Zynq-7000 APSoC using roll-back recovery [9]. The DCLS system consisted of a dual-core ARM processor, two BRAM memory, an external SDRAM memory, and a checker module. The proposed DCLS carried out the same application in both ARM cores at the same time as the application was divided into blocks with a verification point (VP) added between each. Our work has been developed to expand on the work referred to above by adding a third processor core, thus transforming it into a triple-core lockstep approach with support for the roll-forward operation.

III. Proposed Triple-Core Lockstep Technique

The proposed TCLS technique is applied to the dual-core ARM processor along with the MicroBlaze processor in the Xilinx Zynq-7000 APSoC [11]. This device combines a 28 nm programmable logic (PL) layer with an integrated ARM processor on its programmable subsystem (PS) layer. This paper uses a TUL PYNQ-Z2 design and development board [12] featuring a wired Zynq XC7Z020-1CLG400C chip, i.e. the device under test (DUT), for implementation purposes. Furthermore, the PYNQ-Z2 board incorporates an external 512 MB Double Date Rate 3 (DDR3) SDRAM.

A. *Architecture*

The proposed system architecture of TCLS consists of two ARM cores (*CPU0* and *CPU1*), a MicroBlaze core (*CPU2*), a module *Checker-Injector*, three dual-port BRAM memory blocks, an additional DDR SDRAM memory and other miscellaneous blocks (see Fig. 1). The MicroBlaze core was implemented in the PL side and is tripled at module level using the TMR scheme where each input/output port was linked to a majority vote. This scheme was introduced to protect the MicroBlaze core from soft errors that may occur in the configuration memory. This scheme also protects internal core memories from bit-flips. Besides, all cache levels available

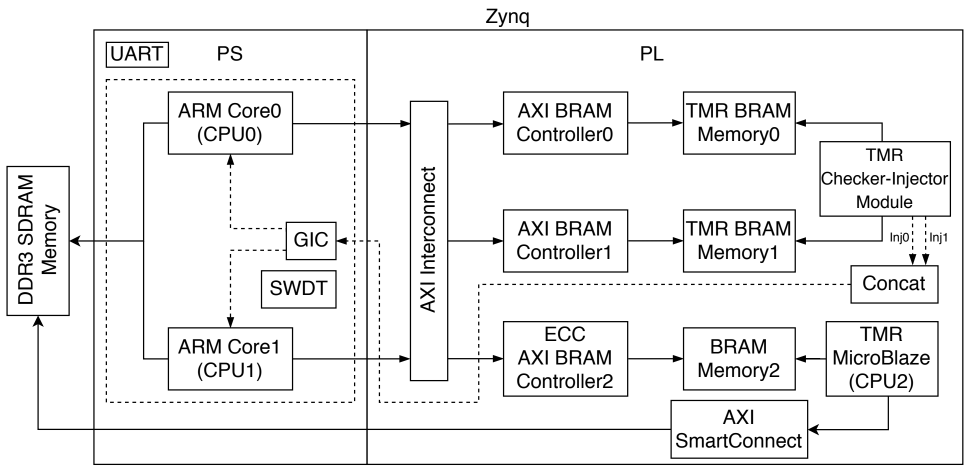

Fig. 1. Block diagram of the proposed triple-core lockstep technique (TCLS)

for both ARM and MicroBlaze processor were disabled to improve device reliability, as it is proven that cache utilization adversely affects the radiation sensitivity of an embedded system [13].

As illustrated in Fig. 1, each ARM and MicroBlaze core is linked to its own private dual-port BRAM memory used to store the corresponding core application data and processor context. Processor cores, however, share external SDRAM memory that stores program instructions for each core at different locations. These three BRAM memory blocks are located in the PL portion of Zynq APSoC and are accessed by an individual AXI BRAM controller through an Advanced eXtensible Interface (AXI) interconnect block by both ARM cores. However, the MicroBlaze core can only access its own allocated BRAM memory through a private AXI BRAM controller (not shown in Fig. 1). Notice that *BRAM Memory0* and *BRAM Memory1* were secured against soft errors using TMR, while Error-Correcting Code (ECC) circuitry [14] was applied to both BRAM controllers associated with *BRAM Memory2* for the same reason.

As shown in Fig. 1, the *Checker-Injector* module is attached to the second ports of *BRAM Memory0* and *BRAM Memory1* and has to objectives: firstly, to monitor the execution of the lockstep and verify the accuracy of *CPU0* and *CPU1* at each step; secondly, to inject faults into the system for testing purposes. This module is a custom-built IP which is implemented in Zynq APSoC's PL side; it is also protected against soft errors using TMR. Finally, though all of our system's PL components are clocked at $\approx$ 91 MHz, the PS dual-core ARM processor operates at 650 MHz.

B. Methodology

The proposed lockstep technique operates by running the same application concurrently in all three cores, where the application is partitioned into code execute blocks, i.e. portions of the original program code combined with redundant software code, i.e. consistency check, checkpoint and recovery routines. There is a verification point (VP) located between each code block. Also, a VP is positioned at the beginning of the program code such that the number of VPs equals the number of execution blocks plus one.

Fig. 2 displays functional block diagram for the proposed TCLS technique operating under fault-free conditions. When the program execution hits a VP on an ARM core, the status of that specific core, which is a signature reflecting the actual CPU state, is written on its corresponding BRAM memory and then the execution is locked on the core. During this lock, the *Checker-Injector* module, also called *ChkInj IP*, creates an interrupt for each core, i.e. *CPU0* and *CPU1*, through the General Interrupt Controller (*GIC*) in Fig. 1, to allow access to both ARM core registers. Subsequently, first output results and then the register files of *CPU0* and *CPU1* are checked and compared by the *Checker-Injector* module as indicated in Fig. 2. If no difference is detected between the outputs and the register values of *CPU0* and *CPU1*, the device is deemed to be in a safe state, and a new interrupt is created individually by the *Checker-Injector* module for *CPU0* and *CPU1* to start a checkpoint operation and save the context of the ARM cores, which is further explained in the subsection III-C.

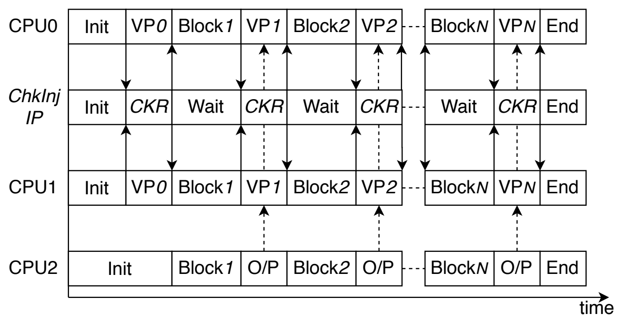

Fig. 2. Functional block diagram for the proposed TCLS technique (*Init* = Initialization, *CKR* = Checker)

However, if a mismatch is found in the output results of *CPU0* and *CPU1*, the MicroBlaze core outputs, i.e. *CPU2*, are accessed for the current VP in their respective BRAM memory for comparison with the corresponding outputs of the two ARM cores. If *CPU2* outputs match the output of one of the ARM cores, then the core with matching outputs will be considered safe and the other faulty. In that case, the *Checker-Injector* module must produce an interrupt to retrieve the defective ARM core using the roll-forward mechanism discussed in subsection III-D. However, if the output results of neither *CPU0* nor *CPU1* match those of *CPU2*, both ARM cores will be recovered using the roll-back method (see subsection III-D) following an interruption by the *Checker-Injector* module. The ARM cores and the MicroBlaze core have different programmer's model; thus, it is not feasible to run comparisons over their register values to identify which ARM core is troubled. Consequently, if the output results of ARM cores match, but there is a mismatch between their registers files, the roll-back operation has to be performed for both cores to prevent any potential output errors. Note that while the MicroBlaze core operates synchronously with *CPU0* and *CPU1* in a lock-step fashion, no checkpoint or recovery operation is performed particularly for it.

When a processor core is interrupted, the following steps are performed in the specified sequence: I) the actual thread being executed is paused; II) the processor core register file, i.e. context, is saved in the corresponding stack memory; III) the assigned interrupt routine is executed to handle the interrupt; IV) the saved context is restored by the stack at the end of the interrupt routine; V) the previous thread begins its execution from the stage it left off. When executing interrupt routines, both ARM cores are switched from *IRQ* mode to the privileged *System* mode that uses the same registers as *User* mode [15]. This is a critical step in enabling access to the same register data used during normal program execution.

C. *Consistency Check and Checkpoint Implementations*

The *Checker-Injector* module is a special-purpose IP designed to snoop the two ARM cores by interrupting them and accessing their respective BRAM memories. This module exhibits two operating modes: In the first mode, it checks and compares the execution of *CPU0* and *CPU1*, and takes one of the remedial measures if any discrepancy is found between the cores, as shown in Fig. 3, depending on the current value of *Recovery Counter* (see Table I); the second mode of operation relates to the fault injection described in section IV. Note that it can be configured to operate in the first mode alone, or both modes simultaneously.

TABLE I
RECOVERY METHOD OPTIONS

Recovery Counter Value	Recovery Method
0	Roll-Forward or Roll-Back
1	Roll-Back First
2	Application Reset
3	Soft System Reset

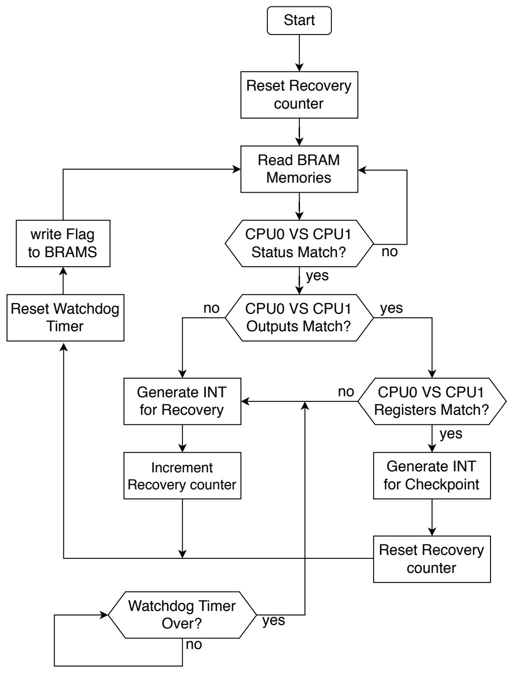
Start
Reset Recovery counter
Read BRAM Memories
CPU0 VS CPU1 Status Match?
no
yes
write Flag to BRAMS
Reset Watchdog Timer
CPU0 VS CPU1 Outputs Match?
no
yes
Generate INT for Recovery
CPU0 VS CPU1 Registers Match?
no
yes
Increment Recovery counter
Generate INT for Checkpoint
Reset Recovery counter
Watchdog Timer Over?
yes
no

Fig. 3. Process flowchart for the *Checker-Injector* module

a) Consistency check operation: As stated earlier, consistency checking is required when processor cores enter a VP to ensure they are in the right state. To facilitate the register consistency test process, ARM cores are independently interrupted during which both ARM cores' register files are stored in their respective stack memories (see step II in subsection III-B). Following this step, the interrupt routine specialized for checkpoint operation is performed independently on *CPU0* and *CPU1*, at stage III, which accesses the processor core's stack memory and duplicates the values stored on the stack to a specific location within the core-associated BRAM memory. Then, the *Checker-Injector* module accesses these positions on BRAM memories allocated to *CPU0* and *CPU1* to compare and detect any inconsistencies. On the other hand, comparisons for processor core outputs are readily applicable, as results provided by an ARM core are always stored at known locations on its corresponding BRAM memory.

The module *Checker-Injector* includes a watchdog timer to ensure that the program execution does not hang in either of the code blocks due to a fault occurring in either ARM cores. At the beginning of each code block, this timer is configured with a suitable time. If both the ARM cores, i.e. *CPU0* and *CPU1*, do not enter the same VP before the allocated time expires, the *Checker-Injector* module will perceive this as a system inconsistency, thus triggering one of the recovery mechanisms available (see Fig. 3).

b) Checkpoint operation: When the consistency between *CPU0* and *CPU1* is verified, the *Checker-Injector* module initiates a checkpoint operation to preserve compatible states (or contexts) of ARM processor cores using the interrupt mechanism. In the following section, it is presumed that the context of an ARM core contains general-purpose registers (i.e. R0-R12), the stack pointer (i.e. SP or R13), the link register (i.e. LR or R14) and the program counter (i.e. PC or R15) within the processor core register file. Note that application data stored in BRAMs are not used as part of the processor context in the checkpointing phase, as these BRAM blocks have already been protected against soft errors.

As mentioned above, the ARM cores are interrupted individually to facilitate the checkpoint process where these cores' register files are stored in stack memories. Subsequently, checkpoint-related interrupt routines triggered on *CPU0* and *CPU1* individually start accessing processor cores' stack memories to copy the register values to certain positions within the respective BRAM memories where they are stored until the next checkpoint.

D. Roll-back and Roll-forward Implementations

If the *Checker-Injector* module detects a mismatch during the consistency check process, one of the available recovery options will be activated depending on the current value of the recovery counter in the *Checker-Injector* module, as shown in Table I. The recovery counter is increased each time a recovery operation is triggered (see Fig. 3); therefore, if an applied recovery method does not help to achieve a secure and consistent state among ARM cores (at which point a new checkpointing occurs), the next recovery method will be selected and applied in the given order. When the recovery counter has the value of zero, either a roll-forward operation or a roll-back operation is performed depending on the essence of the mismatch.

a) Roll-forward operation: If a discrepancy is found in the output results of *CPU0* and *CPU1*, but the outputs of *CPU2* match the outputs of either ARM core, i.e. *CPU0* or *CPU1*, then the core with matching outputs is considered safe, whilst the other core is deemed defective. In this case, the *Checker-Injector* module initiates the roll-forward mechanism to recover the defective ARM core. Inside roll-forward operation, the contexts of ARM cores are accessed individually using the interrupt function, as is the case for checkpoint operation. During the dedicated roll-forward interrupt routine, stack memory locations of the faulty ARM core storing the relevant register file is overwritten with the corresponding register values, i.e. context, of the healthy ARM core after some modifications required by the fact that ARM cores work on different program and data memory locations. When the defective ARM core restores the transferred context from its stack memory to its registers at interrupt mechanism stage IV, it will be restored to the same safe state as the healthy processor. Thus, there is no need to return to a previous point and re-execute any previous code block.

b) Roll-back operation: If the roll-forward operation is not feasible, because the output results of neither *CPU0* nor *CPU1* match those of *CPU2* or because their output results match, but there is a mismatch between their register files, both ARM cores will be recovered using the roll-back operation initiated by the *Checker-Injector* module. A roll-back operation is deployed to restore the device to a previous safe state (or context) saved in the appropriate BRAM memory during one of the checkpoints, where the interrupt function is used to individually access the contexts of ARM cores. As the roll-back interrupt routine is executed, specific stack memory locations of *CPU0* and *CPU1* allocated to store register files are overwritten with the corresponding register values, i.e. context, stored in aforementioned BRAM memory locations. Once an ARM core restores its context from the appropriate stack memory at the interrupt mechanism stage IV, it will be restored to a safe, healthy state. In this case, the program execution in both ARM cores returns to a previous verification point and subsequently re-executes the appropriate code block.

Under normal conditions, roll-back will return the device to the immediately preceding checkpoint. Nonetheless, this checkpoint might not be a safe state for some reason, so recovery will be ineffective. In such a scenario, recovery is made at the first checkpoint. This operation is called first-recovery roll-back. When this does not work either, the checkpoint at the very start of the program will be the next destination for the recovery process; this procedure is called an application reset. If due to a hang or crash in one of the ARM cores, neither roll-forward nor roll-back operations have succeeded in restoring the system, the only remaining option is to apply a system-wide soft reset through the system watchdog timer (*SWDT*) [11].

IV. Fault Injection Technique

To evaluate the efficiency of the soft error mitigation provided by our TCLS approach, we have adopted a fault injection technique that emulates hardware faults by injecting bit-flips into the ARM core registers. These target registers are general-purpose (i.e. R0-R12) and special-purpose registers (i.e. SP, LR, PC) located in the register files of the ARM cores. The fault injection strategy adopted in our work is the same as in [16], i.e. the interrupt mechanism is used to be minimally intrusive. Fault injection experiments are carried out in the same environment as those presented in Fig. 1 with the addition of a host computer connected to the PS of the Zynq APSoC via the UART peripheral core.

At the beginning of the execution of the application, the ARM core *CPU0* configures the *Checker-Injector* module with a random injection time, a random code block number, and a random target location containing the number (0 or 1) of the ARM core under consideration and the number (from 0 to 15) of the register in which the error is injected along with the number (from 0 to 31). Note that the randomly evaluated injection time is relative to the run time of the randomly selected code block. A bit-flip can, therefore, be injected at any time during the code block.

When the *Checker-Injector* module is launched after its configuration, it waits until the selected code block is reached, and then starts counting the clock cycles with a timer until the specified injection time is reached. When the time is up, the *Checker-Injector* module interrupts both ARM cores individually. However, the interrupt routine customized for the fault injection only applies the XOR mask to the target register at the selected ARM core, thus flipping the specified bit in the register. During our experiments, we classified errors occurring based on the scheme provided in Table II.

TABLE II
ERROR CLASSIFICATION FOR FAULT INJECTION EXPERIMENTS

Classification	Description
UNACE	Ineffective faults
SDC	Output result errors
Hang	System hangs/crashes
Mitigated Faults w/ RF	Correction by roll-forward operation
Mitigated Faults w/ RB	Correction by roll-back operation
Mitigated Faults w/ RBF	Correction by roll-back first operation
Mitigated Hangs/Crashes	Recovery by soft system reset

V. Implementation and Experimental Results

This section presents an analysis of the implementation results and describes the results of timing and fault injection performance experiments performed on TCLS-based design, *TCLS design*. We selected benchmark applications performing matrix multiplications, which are widely used in real-life applications [17], to evaluate the timing and fault-injection performance of the proposed TCLS approach. Within each benchmark application, several matrix multiplication operations are performed, in a bare-metal environment, on different input matrices consisting of 32-bit signed data, where each full matrix multiplication operation corresponds to one execution block surrounded by VPs.

In addition to the *TCLS design*, three other design versions have been set up, namely *Unhardened design*, *Unprotected design* and Dual-Core Lockstep (DCLS) design (*DCLS design*). The *Unhardened Design* version will run its applications on *CPU0* only. It has therefore no protection against soft errors. In addition, the *Unprotected design* version is equivalent to *TCLS design* with all protection mechanisms disabled, whereas the *DCLS design* version is comparable to the design in [9].

A. Timing Performance Analysis

Fig. 4 presents timing figures in milliseconds (ms) as required by *Unhardened Design* and *TCLS Design* to perform the above benchmarks, in a fault-free scenario, for five different matrix sizes and three different application sizes in terms of the number of block partitions, by providing one corresponding plot for each application size for both designs. Note that for *TCLS design*, the compiler optimization level *O3* was used for the subroutines of the benchmark programs evaluating matrix multiplication operations to boost performance; on the other hand, the level *O0* was used for the remaining parts of the benchmarks to disable any optimization in the source code that might corrupt the lockstep-based execution. However, the benchmark programs were compiled entirely with *O3* for *Unhardened Design* to facilitate a realistic comparison.

Timing performance overheads with *TCLS design* are considerably higher when the matrix size (i.e. block size) is very small, e.g. overheads are 96.4%, 122.9% and 155.9% for applications with 12, 6 and 3 blocks, respectively when the matrix size is 20×20. However, as the size of the block increases, time overheads tend to fall significantly, as low as 25.7% for a matrix size of 60×60. This point leads us to the conclusion that the timing efficiency in *TCLS design* is achieved when the execution time of the useful computation (e.g. matrix multiplication) is a high proportion within the overall block execution time.

B. Fault-Injection Performance Analysis

An intensive fault injection campaign was carried out in Xilinx Zynq-7000 APSoC mounted on the PYNQ-Z2 board for three design configurations to evaluate the soft error resilience of the proposed TCLS approach. Tables III, IV and V present the results for *unprotected design*, *DCLS design* and *TCLS design*, respectively. Over 3000 runs of 50×50 matrix multiplication benchmarks with an application size of 12 blocks were performed per design. The tables in question show that the rates for *SDC*s and *Hang*s are quite high, i.e. 10.34% and 30.85%, respectively, for *Unprotected design*, while *DCLS design* and *TCLS design* were able to significantly lower *SDC*s and *Hang*s to as low as 0.10% and 0.86%, respectively, due to their possession of protection mechanisms.

TABLE III
FAULT INJECTION RESULTS FOR 50×50 MATRIX MULTIPLICATION WITH NO PROTECTION ENABLED (UNPROTECTED DESIGN)

	Count	Rate
UNACE	2183	58.81%
SDC	384	10.34%
Hang	1145	30.85%
Total	3712	

TABLE IV
FAULT INJECTION RESULTS FOR 50×50 MATRIX MULTIPLICATION WITH NO ROLL-FORWARD CORRECTION ENABLED (DCLS DESIGN)

	Count	Rate
UNACE	816	26.05%
SDC	3	0.10%
Hang	27	0.86%
Roll-Back	1613	51.50%
Roll-Back First	16	0.51%
Soft System Reset	657	20.98%
Total	3132	

Note that the total application rate for roll-back and roll-back first operations is not the same for these two designs, which is 52.0% for *DCLS design* and 44.6% for *TCLS design*. This drop of 7.4% is due to the provision of a roll-forward feature in *TCLS design*, which is very beneficial in reducing

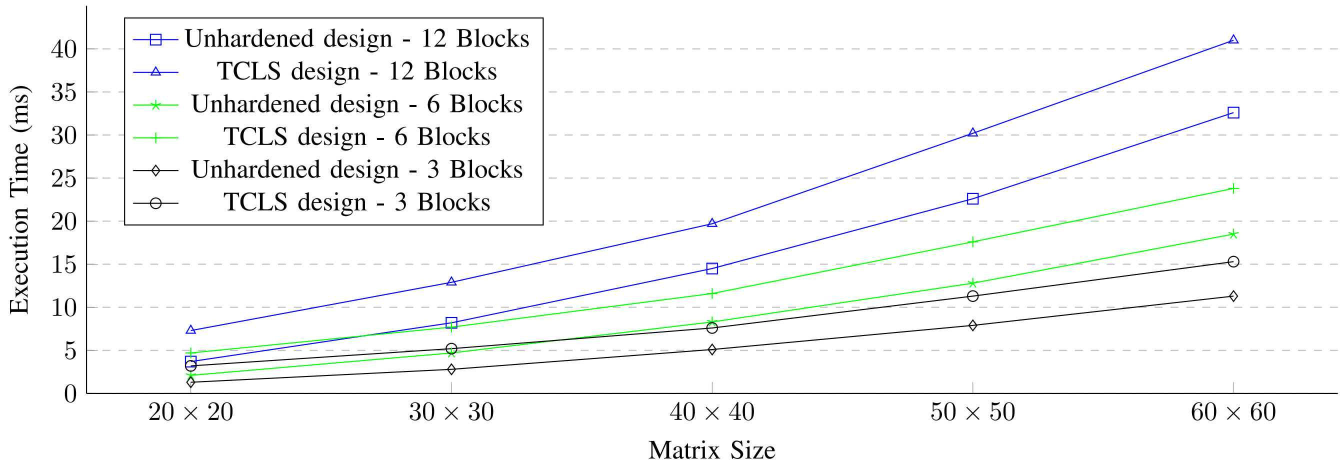

Fig. 4. Matrix multiplication execution times for different application and matrix sizes

TABLE V
FAULT INJECTION RESULTS FOR 50×50 MATRIX MULTIPLICATION WITH FULL PROTECTION ENABLED (TCLS DESIGN)

	Count	**Rate**
UNACE	811	26.73%
SDC	3	0.10%
Hang	28	0.92%
Roll-Forward	194	6.39%
Roll-Back	1303	42.95%
Roll-Back First	50	1.65%
Soft System Reset	645	21.26%
Total	3034	

the overall run-up time of applications exposed to radiation-induced faults. This reduction in execution time will result in a higher MWBF [18]; more data can be processed before a fatal error, such as a hang or crash, occurs when a recoverable error is handled more quickly by a roll-forward operation than by a roll-back operation.

VI. Conclusion

Processors found in APSoCs must be adequately hardened against ionizing radiation to become a viable alternative for harsh environments. This paper proposes a novel triple-core lockstep (TCLS) approach to secure the Zynq's dual-core ARM Cortex-A9 processor in the Xilinx Zynq-7000 APSoC from radiation-induced soft errors by coupling it with the Zynq's programmable logic (PL) to enable roll-forward recovery along with roll-back recovery.

Fault injection tests indicate that the given method has improved the reliability and efficiency of the hard-core ARM processor with a high rate (about 99%) of corrected and recovered faults, while the timing overhead is as low as 25% under fault-free conditions. Moreover, implementing the roll-forward operation enables a 7% higher MWBF. As future research, we plan to further test our approach's effectiveness with other benchmark applications.

Acknowledgment

This work is supported by the UK Engineering and Physical Sciences Research Council through grants EP/R02572X/1 and EP/P017487/1.

References

[1] "NDA," https://www.gov.uk/government/publications/nuclear-provision-explaining-the-cost-of-cleaning-up-britains-nuclear-legacy/nuclear-provision-explaining-the-cost-of-cleaning-up-britains-nuclear-legacy.

[2] R. Baumann, "Soft Errors in Advanced Computer Systems," *IEEE Design Test of Computers*, vol. 22, no. 3, pp. 258–266, May 2005.

[3] J. Arm, Z. Bradac, and R. Štohl, "Increasing Safety and Reliability of Roll-back and Roll-forward Lockstep Technique for Use in Real-Time Systems," *IFAC-PapersOnLine*, vol. 49, pp. 413–418, 12 2016.

[4] S. Kasap, E. Weber Wächter, X. Zhai, S. Ehsan, and K. Mcdonald-Maier, "Survey of Soft Error Mitigation Techniques Applied to LEON3 Soft Processors on SRAM-Based FPGAs," *IEEE Access*, vol. 8, pp. 28 646–28 658, 2020.

[5] H. H. Ng, "PPC405 Lockstep System on ML310," Xilinx Inc., San Jose, CA, USA, XAPP564 Application Note, 2007.

[6] 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.

[7] 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, July 2011.

[8] H. 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, June 2013.

[9] 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.

[10] 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 *Computer Science and Electronic Engineering Conference (CEEC 2019)*, 2019, pp. 124–127.

[11] "Zynq-7000 SoC," Xilinx Inc., San Jose, CA, USA, UG585 Technical Reference Manual, 2018.

[12] "TUL PYNQ-Z2 board," http://www.tul.com.tw/ProductsPYNQ-Z2.html.

[13] L. A. Tambara, P. Rech, E. Chielle, J. Tonfat, and F. L. Kastensmidt, "Analyzing the Impact of Radiation-Induced Failures in Programmable SoCs," *IEEE Transactions on Nuclear Science*, vol. 63, no. 4, pp. 2217–2224, Aug 2016.

[14] G. C. Clark and J. B. Cain, *Error-Correction Coding for Digital Communications*, 1st ed. New York, NY, USA: Springer Publishing Company Inc., 1981.

[15] "ARM Cortex-A Series Programmer's Guide v4.0," ARM Inc., Cambridge, UK, 2013.

[16] Á. B. de Oliveira, L. A. Tambara, and F. L. Kastensmidt, "Exploring Performance Overhead Versus Soft Error Detection in Lockstep Dual-Core ARM Cortex-A9 Processor Embedded into Xilinx Zynq APSoC," in *International Symposium on Applied Reconfigurable Computing (ARC 2017)*, April 2017, pp. 189–201.

[17] H. Quinn, W. H. Robinson, P. Rech, M. Aguirre, A. Barnard, M. Desogus, L. Entrena, M. Garcia-Valderas, S. M. Guertin, D. Kaeli, F. L. Kastensmidt, B. T. Kiddie, A. Sanchez-Clemente, M. S. Reorda, L. Sterpone, and M. Wirthlin, "Using Benchmarks for Radiation Testing of Microprocessors and FPGAs," *IEEE Transactions on Nuclear Science*, vol. 62, no. 6, pp. 2547–2554, 2015.

[18] J. Lienig and H. Bruemmer, *Fundamentals of Electronic Systems Design*. Cham: Springer International Publishing, 2017, ch. Reliability Analysis, pp. 45–73.