Microelectronics Reliability 124 (2021) 114297

Contents lists available at ScienceDirect

Microelectronics Reliability

journal homepage: www.elsevier.com/locate/microrel

Novel lockstep-based fault mitigation approach for SoCs with roll-back and roll-forward recovery

Server Kasap [a,*], Eduardo Weber Wächter [b], Xiaojun Zhai [c], Shoaib Ehsan [c], Klaus D. McDonald-Maier [c]

[a] *School of Computing, Electronics and Mathematics, Coventry University, Coventry, UK*
[b] *School of Engineering, Warwick University, Coventry, UK*
[c] *School of Computer Science and Electronic Engineering, University of Essex, Colchester, UK*

A R T I C L E I N F O

Keywords:
Lockstep
Reliability
Fault tolerance
Soft error mitigation
Zynq APSoC
ARM cortex-a processor
MicroBlaze processor

A B S T R A C T

All-Programmable System-on-Chips (APSoCs) constitute a compelling option for employing applications in radiation environments thanks to their high-performance computing and power efficiency merits. Despite these advantages, APSoCs are sensitive to radiation like any other electronic device. Processors embedded in APSoCs, therefore, have to be adequately hardened against ionizing-radiation to make them a viable choice of design for harsh environments. This paper proposes a novel lockstep-based approach to harden the dual-core ARM Cortex-A9 processor in the Xilinx Zynq-7000 APSoC against radiation-induced soft errors by coupling it with a MicroBlaze TMR subsystem in the programmable logic (PL) layer of the Zynq. The proposed technique uses the concepts of checkpointing along with roll-back and roll-forward mechanisms at the software level, i.e. software redundancy, as well as processor replication and checker circuits at the hardware level (i.e. hardware redundancy). Results of fault injection experiments show that the proposed approach achieves high levels of protection against soft errors by mitigating around 98% of bit-flips injected into the register files of both ARM cores while keeping timing performance overhead as low as 25% if block and application sizes are adjusted appropriately. Furthermore, the incorporation of the roll-forward recovery operation in addition to the roll-back operation improves the Mean Workload between Failures (MWBF) of the system by up to ≈19% depending on the nature of the running application, since the application can proceed faster, in a scenario where a fault occurs, when treated with the roll-forward operation rather than roll-back operation. Thus, relatively more data can be processed before the next error occurs in the system.

1. Introduction

Cleaning up the legacy of nuclear waste is one of Europe's most critical and complicated environmental remediation projects, which is expected to cost as much as £220bn over the next 120 years [1]. Because of the extreme adverse effects of the ionizing radiation on biological tissues, it is perilous for humans to clean up radioactive waste inside a nuclear power plant, which is abundant in these areas especially after a nuclear accident. As a result, deploying robots are highly encouraged and desired in radiation environments such as nuclear power plants and radioactive waste disposal sites.

Although human beings are spared from entering harsh environments when employing robots, this is not an easy task, since the electronic circuits in these robots are prone to radiation effects as well. Following a series of meltdowns at the Fukushima Daiichi nuclear power plant in Japan as a consequence of a tsunami strike [2], this has become even clearer as the robots dispatched to track radiation levels and expedite the clean-up process broke down very quickly, because of the radiation-induced damage to their circuits. Therefore, adverse effects of radiation on electronic circuits have to be substantially mitigated if robots are to be employed in nuclear environments; microprocessors should be particularly treated against radiation as they are responsible for the control and coordination of operations within the system.

High-energy particles (e.g. alpha particles, heavy ions, neutrons) or electromagnetic waves (e.g. X-rays, gamma rays) striking the semiconductor substrate provoke faults which might lead to soft errors in

* Corresponding author.
E-mail addresses: server.kasap@coventry.ac.uk (S. Kasap), eduardo.weber-wachter@warwick.ac.uk (E.W. Wächter), xzhai@essex.ac.uk (X. Zhai), sehsan@essex.ac.uk (S. Ehsan), kdm@essex.ac.uk (K.D. McDonald-Maier).

https://doi.org/10.1016/j.microrel.2021.114297
Received 5 August 2020; Received in revised form 6 July 2021; Accepted 18 July 2021
Available online 5 August 2021

electronic circuits under radiation [3]. While these errors have a transient behaviour that does not permanently harm digital/analogue circuits, they have a significant impact on system reliability and reliability, particularly more as feature size of transistors scales down resulting in very densely integrated microchips [4]. Many mission-critical applications could have been implemented in All-Programmable Systems-on-Chips (APSoCs) which combine programmable logic (PL) layer (i.e. SRAM-based FPGA layer) with embedded processors in the processor subsystem (PS) layer. Such APSoCs enjoy the merits of higher performance, lower energy consumption, and favourable time-to-market and cost [5]. Unfortunately, these highly-integrated circuits, which involve a set of homogeneous or heterogeneous processor cores, are very susceptible to transient faults that might even lead to total system failures.

Soft errors affect processors by corrupting values stored in memory elements such as registers, cache, data and instruction memories, which may cause the processor to execute an application inaccurately, thus resulting in silent data corruptions (SDCs) or functional interrupts (FIs), such as hangs and crashes, in the system. In the PL side, soft errors can manipulate the SRAM memory storing the configuration bitstream along with user memories such as Flip-Flops (FFs) and Block RAMs (BRAMs), all of which might induce shifts in the device functionality and performance. Therefore, the adoption of techniques to mitigate radiation-induced transient faults is the only viable way to leverage the benefits of APSoCs in radiation environments. In this regard, several fault-mitigation methods have been proposed in the literature. However, most of them protect the device from either SDCs or FIs; few methods are successful against both.

To improve the reliability and availability of the dual-core ARM Cortex-A9 processor embedded in the Xilinx Zynq-7000 APSoC, we have adapted a fault mitigation technique, the triple-core lockstep technique (TCLS). The TCLS approach proposed in this work couples these two ARM cores in the PS with one MicroBlaze core implemented in the PL in order to replicate the execution of the same application in a lockstep manner, along with a checker module monitoring and checking the outputs of the ARM cores for any inconsistencies. The MicroBlaze core in the FPGA part is protected against soft errors using a Triple Modular Redundancy scheme. Furthermore, the proposed technique combines both checkpointing and roll-back/roll-forward operations at the software level to provide dependability. Fault-free copies of processor core states are stored in safe memories during checkpoints, whereas roll-back and roll-forward operations constitute fault recovery mechanisms, which respectively restore a processor core to a previous safe state or to the current safe state of the other core which happens to be healthy [6]. The innovation in our research lies in the fact that, in a lockstep-based methodology, this is the first time a MicroBlaze core is coupled with hard-core ARM processors in order to support roll-forward recovery along with roll-back recovery. With the introduction of the roll-forward recovery, system performance has been improved in terms of Mean Workload between Failures (MWBF), as will be discussed in the results section of the paper. Furthermore, the approach can be conveniently extended to Xilinx Zynq UltraScale+ MPSoCs.

Fault injection experiments, which emulate bit-flips in ARM register files in a non-intrusive manner, were performed to analyze the efficiency, effectiveness and fault coverage of our proposed TCLS technique. Experiments indicate that the TCLS approach applied to the dual-core ARM Cortex-A9 processor embedded in Xilinx Zynq-7000 APSoC is capable of mitigating around 98% of the bit-flips injected while keeping the timing performance overhead as low as 25%, when certain conditions are satisfied, under fault-free conditions.

The paper is set out as follows. Section 2 presents the impact of radiation on electronics, particularly on processors, as well as a summary of current fault mitigation strategies with a particular focus on the lockstep technique. Section 3 elaborates on the architecture and implementation methodology of the proposed TCLS approach, while Section 4 describes the fault injection mechanism employed during validation experiments for the approach. Implementation and experimental results are subsequently evaluated in Section 5. Finally, conclusions are drawn in Section 6 along with plans for future.

2. Background

This section motivates the need for protection against soft errors by addressing the effects of radiation on electronics in general and then discusses the effects of soft errors in processors. Furthermore, several significant fault-tolerance techniques are introduced as reported in the literature. The lockstep technique is then particularly discussed as it is one of the most promising techniques.

2.1. *Radiation effects on electronics*

Ionizing radiation can potentially harm electronics in three major ways. The first-way radiation exposure affects electronics is called Total Ionizing Dose (TID) [7], which relates to an electronic device's accumulated, irreversible damage (i.e. hard error) that results in the degradation of the device over the time. It happens when charge carriers are inserted, as radiation strikes, into the insulators of the device, where they are trapped and thereby change the electrical characteristics [8]. The second major group of harmful effects caused by radiation is commonly called Single-Event Effects (SEEs) [7]. These faults are most frequently temporary (i.e. soft error) and do not cause permanent harm like TID; however, they can provoke undesired changes in behaviour. Excess charge carriers, produced as silicon atoms get ionized under radiation, are responsible for these transient effects. If a large amount of these charges accumulates in a particular region, an event called Single-Event Transient (SET) might arise where logical values of lines in that region are briefly disrupted till the dissipation of the excess charges. Nevertheless, if a storage unit latches the new value of a line, a longer-lasting effect on the system output, i.e. Single-Event Upset (SEU), will occur which can generally be resolved by restoring all values through a system reset. However, Single-Event Functional Interrupts (SEFIs), a type of SEU, cannot be dealt with a simple reset [9]. Displacement Damage (DD) [7] is the third and final major irradiation effect occurring when a high-speed particle hits and displaces silicon atoms from their locations. These movements cause defects in the silicon substrate; thus, the electrical features of the device are altered.

2.2. *Effects of soft errors in processors*

SEUs can readily affect the data-flow and control-flow of a processor, which is a significant concern for safety-critical applications. Upsets in the values stored in memory elements can lead to data-flow errors caused by the execution of incorrect operations or data manipulation. The execution of an incorrect operation occurs when a bit-flip corrupts the program code, leading to an incorrect instruction. On the other hand, if the bit-flip affects data used as an input by an operation, outputs of that operation would most probably be incorrect. In both types of data-flow errors, the final outputs of the application would be inaccurate, which is classified as SDC.

When an SEU affects the control-flow, the processor may execute the program incorrectly, thus causing either an application crash or a processor hang, which is classified as FIs. Upsets in the control-flow may lead to branch errors, such as erroneous creation or deletion of a branch and incorrect branch decision. The erroneous creation is due to a bit-flip that sets a non-branch instruction into a branch, which leads the program flow to an incorrect address, whereas the erroneous deletion occurs when the branch instruction is transformed into another instruction; thus, a due branch may not be taken. A bit-flip in a conditional branch may also result in an incorrect decision regarding taking the branch or not. Furthermore, an SEU can modify the register holding the target address of a branch instruction, thus assigning an incorrect address to the program execution. Moreover, if the program counter (PC) is affected by a soft error, the next instruction to be executed changes, leading the program flow to an incorrect address.

2.3. Fault-tolerance techniques

As the sophistication of embedded systems grows, their vulnerability to errors is adversely affected due to an increase in critical points of failure. The adoption of fault mitigation or fault tolerance techniques is therefore vital if APSoCs are to be used in radiation environments. Fault tolerance techniques that enhance embedded processor reliability can be categorised as hardware-, software- and hybrid-based techniques [10].

The hardware-based techniques, which mainly rely on spatial redundancy, provide two or more instances of a hardware component, such as processors, memories, buses or power supplies, for protection against soft errors. This class of techniques include Triple Modular Redundancy (TMR) [11], Duplication with Comparison (DWC) [7] and hardware monitors [12] which incorporate watchdog or checker modules to monitor the system and detect errors by verifying the control-flow related memory accesses of the target processor. These techniques can protect the system from errors in the computation outputs, i.e. SDCs, as exemplified in [13].

Software-implemented hardware fault tolerance (SIHFT) approaches handle hardware malfunctions by merely shielding the software without any hardware alteration. These techniques rely on adding redundant software code for comparison to detect errors. However, they exhibit a high-performance overhead, which may not be viable for some real-time systems. This kind of techniques, such as ABFT [14], HETA [15] and S-SETA [16], detect control-flow faults leading to FIs, which manifest themselves as hangs or crashes, and then place the system into a fail-safe state. In general, both hardware- and software-based techniques are not capable of correcting errors, but rather detecting them, to avoid a failure which would have adverse effects on the entire mission. Furthermore, they protect the system either from SDCs or FIs, but not both. However, there are some exceptional SHIFT techniques, such as SWIFT-R [17] and S-SWIFT-R [18], which provide data-flow level protection supporting both the detection and recovery of SDCs as well as FIs; some faults in the data-flow can produce FIs and consequently can be avoided through data protection mechanisms like SWIFT-R.

The hybrid techniques are the ones that use a SIHFT method combined with a hardware intellectual property (IP), which performs consistency checks in the processor, making them effective against both SDCs and FIs. For instance, the lockstep technique is a hybrid fault-tolerance technique based on software and hardware redundancy. It employs the concepts of checkpointing and recovery mechanisms (e.g. roll-back recovery, roll-forward recovery) at the software level, and processor replication and checker circuits at the hardware level, as explained in the next sections. Therefore, it is capable of both error detection and correction.

2.4. Lockstep technique

A standard lockstep technique works by simultaneously and symmetrically running the same programme on identical processors with identical code and data inputs. The status of the processors is, therefore, the same from clock to clock during regular operation. Consequently, they are assumed to carry out the same operations, in an error-free execution, thus allowing the monitoring of the processor's data, address and control buses.

As mentioned before, within a lockstep based system, there is a checker module that tracks the processors and systematically examines their state checking for any discrepancies; verification points are inserted into the application programme where the execution of the application is locked to facilitate this process of state evaluation. If no discrepancy is identified among processor states, processors are deemed to be faultless, and a checkpoint operation is carried out; otherwise, the lockstep system recovers the processors to a healthy state through either a roll-back or a roll-forward mechanism.

Checkpointing is an operation where processors' error-free contexts are stored in secure memories which are protected against soft errors by using an Error-Correcting Code (ECC), a TMR scheme or any other appropriate method. Note that ECC can catch and fix single-bit errors and merely catch double-bit errors [19]. The context of a processor is identified as the data resources, e.g. data values in registers, caches and main memory, processed during its execution of a programme which are required for the system repair in case of a failure. Furthermore, roll-back is a recovery method which restores a fault-free copy of a processor's previous context from the secure memory for all the relevant processors. In contrast, the current context of a processor that happens to be healthy is loaded into a faulty processor, within the roll-forward recovery operation, from the associated secure memory. Processors are recovered to a safe state without errors as a result of the roll-back operation, and then restart the execution of their programme from that past stage, thus potentially wasting valuable run-time. As for the roll-forward recovery, again all processors are set to a safe, consistent state, but not to a previous state; therefore, there is no need for a re-execution, thus facilitating a lower performance overhead.

The lockstep technique's most significant merit is its ability to detect and correct both SDCs and FIs, in contrast to many other fault tolerance techniques. Several researchers have developed and implemented their lockstep technique version, such as those in [20–25], to make a range of processors resistant to radiation-induced soft errors, extensively analyzed and compared in [26]. Two of these significant works are discussed next.

Oliveira et al. introduced dual-core lockstep (DCLS) technique to counteract radiation-caused faults in ARM-A9 processors embedded in Zynq-7000 APSoC using roll-back recovery [24]. Their DCLS system was composed of a dual-core ARM processor, two BRAM memories, an external SDRAM memory along with a checker module. The proposed DCLS executed the same application program in both ARM cores concurrently, where the programme was partitioned into blocks with a verification point (VP) placed between each. When a VP is reached, DCLS will pause the execution, and the hardware module will compare the outputs of two ARM cores. In the case where no discrepancy is observed, the system will be deemed healthy, and the execution of the next block will be initiated. Otherwise, both cores will be recovered using the roll-back mechanism via processor interruption.

The work in [25] presents a similar dual-core lockstep approach where software-based data error detection and recovery through redundant parallel threads has been combined with hardware-based control-flow error detection through an external module tracing both cores. The proposed technique provides a very high error coverage where only data-flow errors are corrected via a roll-back based recovery scheme.

Our work has been developed to expand on the above works through the addition of a third processor core, thus evolving it into a triple-core lockstep approach with support for the roll-forward operation which proves to be very beneficial, as explained in the remainder of the paper.

3. Proposed triple-core lockstep technique

The proposed TCLS technique is implemented on the dual-core ARM processor along with a MicroBlaze processor in the Xilinx Zynq-7000 APSoC [27] which incorporates a 28 nm programmable logic (PL) layer along with an embedded ARM processor on its processing subsystem (PS) layer. In this paper, we employ a TUL PYNQ-Z2 design and development board [28], featuring a wire-bonded Zynq XC7Z020-1CLG400C chip, i.e. the device under test (DUT), for implementation purposes.

The PL part of the DUT is based on an Artix-7 SRAM-based FPGA incorporating 630 KB fast Block RAM (BRAM), 13.3 K logic slices and 220 digital signal processing (DSP) slices; this is the part hosting the MicroBlaze core. Furthermore, the PS is composed of a 32-bit dual-core

ARM Cortex-A9 processor, a 256 KB dual-port on-chip SRAM memory (OCM), dynamic memory controller with 8 DMA channels and four high-performance AXI3 slave ports along with several types of input/output (I/O) units. Each ARM core has independent 32 KB L1 data and instruction caches, while a 512 KB unified cache is shared between them. Moreover, the PYNQ-Z2 board incorporates an external 512 MB Double Date Rate 3 (DDR3) SDRAM with a 16-bit bus along with a 16 MB Quad-SPI flash memory.

3.1. Architecture

The proposed TCLS system architecture is composed of two ARM cores (*CPU0* and *CPU1*), a MicroBlaze core (*CPU2*), a *Checker-Injector* module, three dual-port BRAM memory blocks, an external DDR SDRAM memory and other miscellaneous blocks (see Fig. 1). The MicroBlaze core has been implemented in the PL side of the Zynq APSoC and is triplicated at the module level using the TMR scheme, where each input/output port has been connected to a voter which chooses an output bit based on the majority of input bits for each individual bit in the port. This scheme has been adopted to protect the MicroBlaze core from soft errors which may occasionally occur in the configuration memory associated with the core. User memories internal to the core are protected against bit-flips by this scheme, as well. Note that TMR-protected MicroBlaze core is acting as one single core. Furthermore, all MicroBlaze cores have been configured in such a way that they support all arithmetical operations including floating-point operations. The ARM cores on the other hand are application-level Cortex processors. Therefore, there are no limitations in our system pertaining to arithmetical computations. Note that all cache levels available for both ARM and MicroBlaze processors have been disabled within software to enhance the system reliability; it has been shown that caches increase the radiation sensitivity of processor-based systems [29].

As illustrated in Fig. 1, each ARM and MicroBlaze core is attached to its respective dual-port BRAM memory, which is used to store the application data and processor context of the corresponding core. However, processor cores are sharing the external SDRAM memory, which stores program instructions for each core at distinct locations. These three BRAM memory blocks, whose size are adjusted based on the given application, are located in the PL part of the Zynq APSoC and are accessed through an individual AXI BRAM controller via an Advanced eXtensible Interface (AXI) Interconnect block by both ARM cores. However, the MicroBlaze core can only access its own allocated BRAM memory, i.e. *BRAM Memory2*, through a private AXI BRAM controller (not shown in Fig. 1). Furthermore, the MicroBlaze core is connected to the SDRAM memory via an AXI SmartConnect block. *BRAM Memory0* and *BRAM Memory1* were protected against soft errors using the TMR scheme, whereas ECC circuitry has been added to both BRAM controllers associated with the *BRAM Memory2* to achieve the same goal.

As shown in Fig. 1, *Checker-Injector* module is connected to the second ports of *BRAM Memory0* and *BRAM Memory1*, and is assigned with two tasks: 1) to control the lockstep execution and verify the consistency of *CPU0* and *CPU1* at each step; 2) to inject faults into the system for testing purposes as will be further detailed in Section 4. This module is a custom IP designed in VHDL and implemented in the PL side of the Zynq APSoC; it is protected against soft errors using the TMR scheme as applied to the MicroBlaze core. It is noteworthy to mention that while the intermediate output data of both ARM cores are stored in their respective BRAM memories, so that *Checker-Injector* module can perform comparison operations across them, final data outputs of ARM cores are transferred to the SDRAM memory; thus, they can be examined to check whether they match up to the expected, golden, data outputs. Furthermore, the MicroBlaze core holds its intermediate output data in its corresponding BRAM memory as well, and its final outputs are also transferred to the SDRAM memory to be used as golden data outputs. Finally, all components in the PL side of our system are clocked at ≈91 MHz, while the dual-core ARM processor in the PS is running at 650 MHz.

3.2. Methodology

The given lockstep approach works by simultaneously running the same application software in all three cores, where the programme is partitioned into code execute blocks, i.e. portions of the original application code coupled with redundant code required for realizing a verification point (VP) which incorporates consistency check, checkpointing, and recovery routines. As such, a VP is present between each code block and at the beginning of the programme code. It is worth noting that, depending on the application specifications, the number of code blocks in which the initial programme is partitioned can be customized.

The functional block diagram for the proposed TCLS technique is given in Fig. 2. Furthermore, the execution flowchart for the processor cores applying the given lockstep technique is illustrated in Fig. 3. As soon as the program execution reaches a VP on an ARM core, the status of that particular core, which is a signature representing the actual CPU state, is written on its own BRAM memory, and then the execution on the core is locked as can be observed in Fig. 3. When both ARM cores are waiting locked at the same VP, the *Checker-Injector* module, also referred

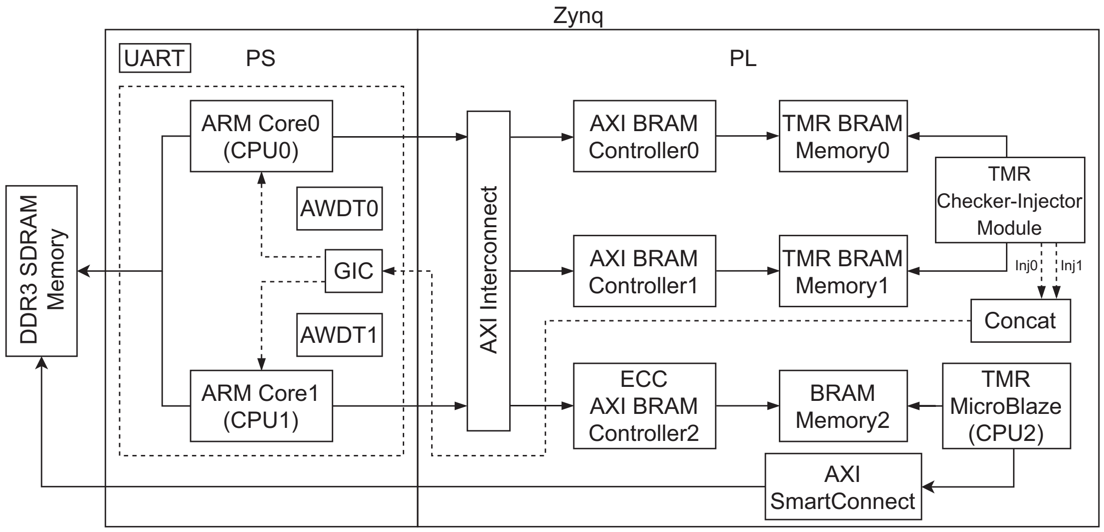


Fig. 1. Block diagram of the proposed triple-core lockstep technique (TCLS).

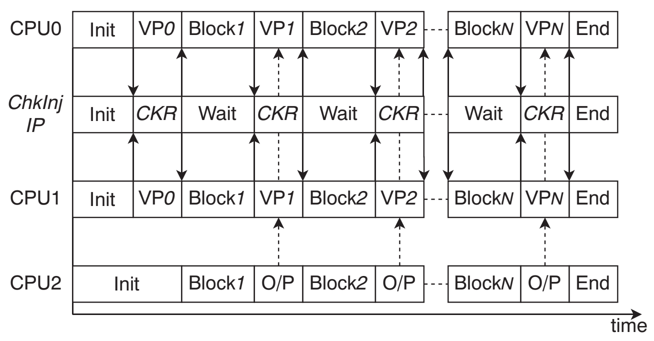


Fig. 2. Functional block diagram for the proposed TCLS technique (*Init* = Initialization, *CKR* = Checker), *O/P* = Outputs).

to as *ChkInj IP*, generates an interrupt for each core, i.e. *CPU0* and *CPU1*, to facilitate access to the registers of both ARM cores. Subsequently, first output results and then register files of *CPU0* and *CPU1* are checked and compared by the *Checker-Injector* module as denoted by the sign *CKR* in Fig. 2. If no discrepancy is observed between the outputs and register values of *CPU0* and *CPU1*, the system is deemed to be in a safe state, and then a new interrupt is individually generated by the *Checker-Injector* module for *CPU0* and *CPU1* to launch a checkpoint operation and save the ARM cores' context, which is further explained in the subsection 3.4.

However, in such a case where discrepancies between the output results of ARM cores are detected, the outputs generated by the MicroBlaze core, i.e. *CPU2*, for the current VP are fetched from its corresponding BRAM memory to be compared against the respective outputs of the two ARM cores. If the outputs of *CPU2* match with the outputs of one of the ARM cores, i.e. *CPU0* or *CPU1*, then the core with matching results would be deemed to be healthy and the other one to be faulty. In such a case, an interrupt will be generated by the *Checker-Injector* module to recover the faulty ARM core using the roll-forward mechanism explained in the subsection 3.5. However, if output results of neither *CPU0* nor *CPU1* matches with these of *CPU2*, both ARM cores would be recovered using the roll-back mechanism (see subsection 3.5) following an interrupt generation by the *Checker-Injector* module. Note that in the case where output results of *CPU0* and *CPU1* match, but there is a discrepancy between their registers files which indicates that an output error is imminent, the roll-back operation is carried out, as well, for both cores to preclude any future errors. In such a scenario, triggering roll-forward mechanism is not viable because the ARM cores and the MicroBlaze core have distinct programmer's models, that is, they have different register file structures making it unfeasible to perform register file comparisons to detect which ARM core is faulty. Therefore, fault mitigation with the roll-forward operation is not applicable in this case.

At the end of the checkpoint or roll-back/roll-forward operations mentioned above, the *Checker-Injector* module writes a flag on the two BRAM memories associated with the ARM cores, i.e. *CPU0* and *CPU1*, to unlock them, thus enabling their execution of the application till the next VP, where the same cycle will repeat as shown in Fig. 3. Note that although the MicroBlaze core, i.e. *CPU2*, operates synchronously with *CPU0* and *CPU1*, no checkpoint or recovery operation is particularly performed for it, as it is assumed to be immune to soft errors thanks to its TMR protection, for the sake of improving overall system performance. Furthermore, the MicroBlaze cores do not actually constitute a significant performance bottleneck in our system although it runs much slower than ARM cores. The reason is that under normal conditions when there are no faults detected, the pair of ARM Cortex-A9 cores and MicroBlaze core operate independently with no direct interference between them; thus, the MicroBlaze core do not slow down the ARM cores during the fault-free operation. It is only when a fault is detected in the system is the MicroBlaze core waited upon to synchronize with the ARM cores. Only in this case, a timing performance loss is incurred due to the MicroBlaze

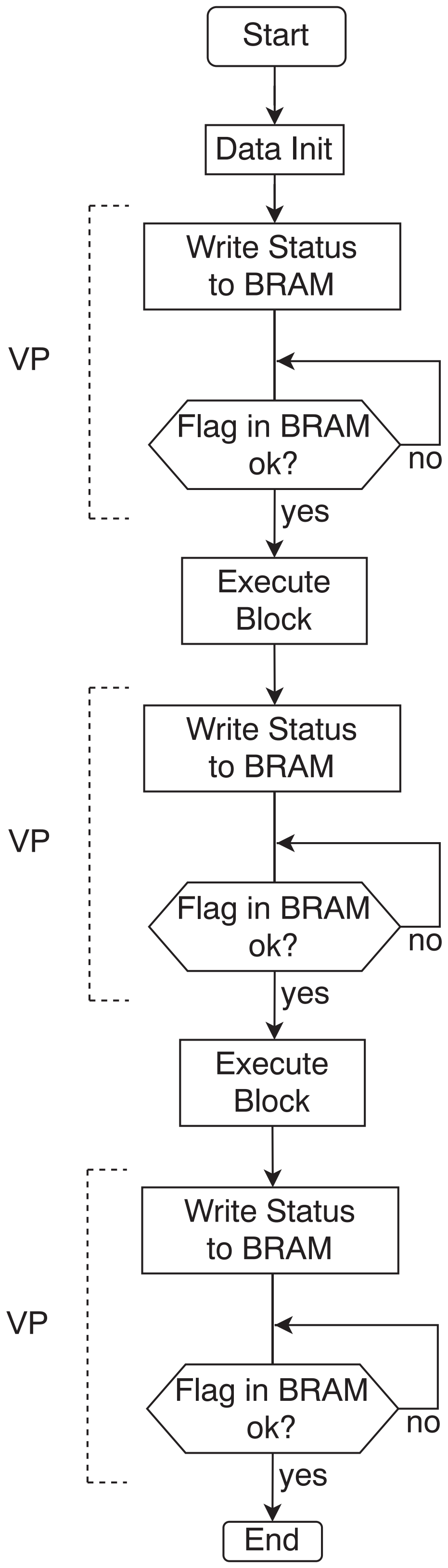


Fig. 3. Flowchart for the processor cores applying the TCLS technique.

core for the benefit of saving the system from an SDC or a crash using one of the provided recovery methods. Nevertheless, our experiments show that the time overhead due to the MicroBlaze core is relatively small (less than 10%) compared to the execution time of a block within the process; this is mainly because when computationally useful operations are halted in ARM cores within the verification points, the MicroBlaze core keeps executing its instructions which helps it to keep in pace with the ARM cores operating with a larger clock frequency.

3.3. *Interrupt implementation*

As mentioned above, interrupt mechanism is frequently employed in the TCLS technique through the General Interrupt Controller (*GIC*) in Zynq SoC (see Fig. 1) to perform many operations, i.e. consistency check, checkpoint, roll-back and roll-forward operations. Before we delve into these operations, the way interrupts work in a processor system will be briefly explained next in this subsection.

When a processor core processes an interrupt, the following steps are carried out in the sequence provided: I) the actual thread under execution is paused; II) the register file of the processor core, i.e. the context, is saved into the corresponding stack memory; III) the dedicated interrupt routine is executed to serve the given interrupt; IV) the saved context is restored by the processor core from the stack at the end of the interrupt routine mentioned in step III; V) the previous thread continues its execution on the processor core from the point it left off. It is noteworthy that during the execution of interrupt routines, both ARM cores are switched from the *IRQ* operation mode to the privileged *System* mode, which uses the same registers as the *User* mode. Since ARM adopts the scheme of *Banked Registers* [30], this is a vital step to facilitate access, within interrupt routines, to the same register contents as in the normal program execution.

3.4. *Consistency check and checkpoint implementations*

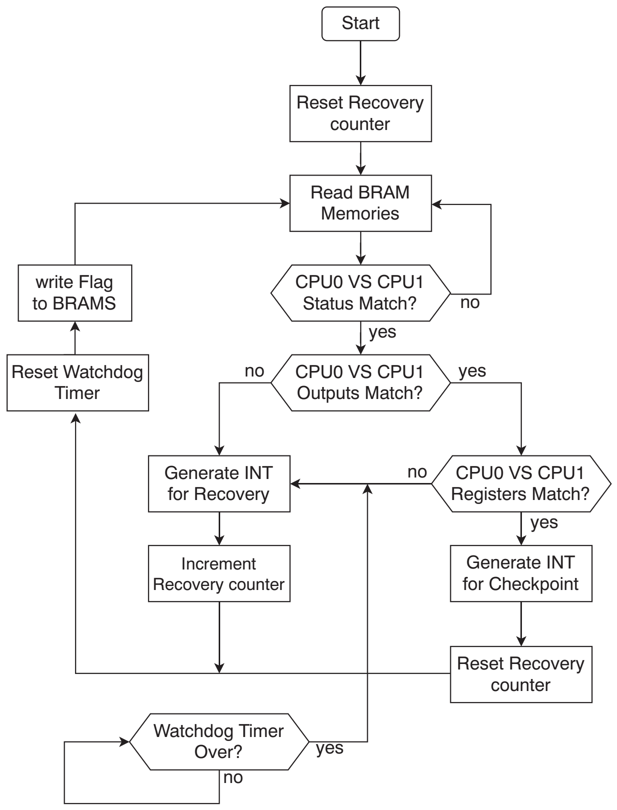
The *Checker-Injector* module is a special-purpose IP which is designed to snoop the operations of the two ARM cores by interrupting them and accessing their own BRAM memories. This module exhibits two modes of operation: In the first mode, it checks and compares the execution of *CPU0* and *CPU1*, and takes one of the remedial actions if any inconsistency is detected between the cores, as illustrated in Fig. 4, depending on the current value of the *Recovery Counter* (see Table 1); the second mode of operation is pertaining to the fault injection which will be discussed in Section 4. Note that it can be configured to operate in the first mode alone or in both modes at the same time.

3.4.1. *Consistency check operation*

As mentioned before, the data verification is required when processor cores reach a VP to ensure that they are in a correct state. By this token, the *Checker-Injector* module compares the ARM cores' registers in its first mode of operation after verifying the consistency of their output results. If no mismatch is detected in either registers or outputs, the module initiates a checkpoint operation as discussed later in this subsection. Note that at the beginning of programme execution, the general-purpose registers at *CPU0* and *CPU1* are set to null values to start with a consistent state.

The ARM cores are individually interrupted, to facilitate the consistency check operation for the registers, during which the register files of both ARM cores are saved into their respective stack memories (see step II in subsection 3.3). Following this step, the interrupt routine customized for the checkpoint operation is individually launched on *CPU0* and *CPU1*, at step III, which accesses the stack memory of the processor core it is executing on and duplicates the register values stored on the stack into a particular location within the BRAM memory associated with the core. The *Checker-Injector* module then accesses these locations on BRAM memories assigned for *CPU0* and *CPU1* to make comparisons and detect any inconsistencies. On the other hand, comparisons for the output results of the processor cores are readily applicable since results produced by an ARM core are always stored on its corresponding BRAM memory at known locations.

The *Checker-Injector* module incorporates a watchdog timer to ensure



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

Fig. 4. Process flowchart for the *Checker-Injector* module.

Table 1
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

that the application execution will not hang in one of the code blocks due to a fault occurring in either ARM cores. At the beginning of every code block, this timer is configured with a time amply suitable for the given code block. If both of the ARM cores, i.e. *CPU0* and *CPU1*, do not reach to the same VP before the allocated time elapses, the *Checker-Injector* module would interpret this as a system inconsistency; therefore, it will initiate one of the available recovery mechanisms (see Fig. 4).

3.4.2. *Checkpoint operation*

When the consistency between *CPU0* and *CPU1* is confirmed, a checkpoint operation is initiated by the *Checker-Injector* module to save consistent states (or contexts) of the ARM processor cores through the utilization of the interrupt mechanism. In the following discussion, an ARM core's context would be assumed to contain general-purpose registers (i.e. R0-R12), a stack pointer (i.e. SP or R13), a link register (i.e. LR or R14) and a program counter (i.e. PC or R15). All these registers are located within the register file of the processor core. Note that application data stored in BRAMs are not included in the checkpointing process as part of the processor context since these BRAM blocks have already been secured against soft errors through TMR or ECC techniques. Furthermore, cache memories have been disabled for all cores.

As mentioned above, the ARM cores are individually interrupted to facilitate the checkpoint operation in which register files of these cores are saved into the stack memories. Subsequently, the checkpoint-related interrupt routines triggered on *CPU0* and *CPU1* individually start accessing stack memories of the processor cores in order to copy the register values on stack memories into specific locations within respective BRAM memories where they are stored until the next checkpoint. The checkpoint operation is completed by the time processor core returns from the interrupt routine at step V.

It is worth noting that context is redundantly written into a second location within the relevant BRAM memory during the first checkpoint operation, which is preserved until the end of the application execution; the goal of this approach is to protect the system against soft errors occurring during the context storage. By this way, it is possible to recover the system to the beginning of the first code block. Furthermore, a partial checkpoint is performed by default at the very start of the application to facilitate a software reset, i.e. a return to the beginning of the application, when necessary. The next subsection will explain those two approaches.

3.5. Roll-back, roll-forward and soft reset implementations

If a mismatch is detected in either registers or outputs by the *Checker-Injector* module during the consistency check operation, one of the available recovery options will be triggered depending on the current value of the recovery counter incorporated in the *Checker-Injector* module, as shown in Table 1. The recovery counter is incremented after each time a recovery operation is triggered and is reset to zero after each time a checkpoint operation is launched, as illustrated in Fig. 4. The next method of recovery from Table 1 will be thus selected and applied in the given order if an applied recovery method does not help to reach a safe and consistent state between ARM cores (at which point a new checkpointing takes place). As a note from the table, either a roll-forward operation or a roll-back operation is performed when the recovery counter has the zero value, depending on the nature of the mismatch detected during the consistency check. For other values of the recovery counter, there are different methods, which are based on the roll-back recovery, to be deployed in the increasing order of severity for persistent errors (see Table 1). These recovery methods will be explained in the following subsections.

3.5.1. Roll-forward operation

If a mismatch is detected between the outputs of *CPU0* and *CPU1*, but the outputs of *CPU2* match the outputs of either ARM core, i.e. *CPU0* or *CPU1*, the core with matching results is considered to be healthy, while the other one is deemed to be faulty. In this case, the roll-forward mechanism is initiated by the *Checker-Injector* module using the interrupt mechanism to recover the faulty ARM core.

Within the roll-forward operation, the ARM cores' contexts are individually accessed by utilizing the interrupt mechanism, as is the case for the checkpoint operation. During the dedicated interrupt routine for the roll-forward operation, stack memory locations of the faulty ARM core, i.e. either *CPU0* or *CPU1*, storing the relevant register file are overwritten with the corresponding register values, i.e. context, of the healthy ARM core via a transfer from the specific locations of the BRAM memory associated with the healthy core. However, some modifications are required on these register values before loading them into the faulty core, since the ARM cores are operating on different program and data memory locations, although they are executing the same application program with the same input data. Therefore, differences between register values of the ARM cores should be pre-evaluated at run-time, especially for the special purpose registers (i.e. SP, LR, PC), for each individual application before the actual execution.

These evaluated differences should be taken into account while transferring context from one core to another in order to assure a healthy roll-forward operation. When the faulty ARM core restores the transferred context from its stack memory into its registers at step IV of the interrupt mechanism (see subsection 3.3), it would be recovered to the same, safe state as the healthy processor. Thus, there would be no need to return to a previous point and re-execute any past code block, which is illustrated in Fig. 5 where the faulty *CPU1* core is recovered to the state of the healthy *CPU0* core after a mismatch detection in *VP7*. Consequently, in Fig. 5, both cores proceed with their execution with *Block8* with no re-execution of past code execute blocks. Note that roll-forward operation is only applied on the faulty core, while the healthy core simply waits.

3.5.2. Roll-back operation

If the aforementioned roll-forward operation is not applicable, because output results of neither *CPU0* nor *CPU1* matches with those of *CPU2* or because their output results match, but there is a discrepancy between their register values, both ARM cores would be recovered using the roll-back operation launched by the *Checker-Injector* module through individual interruptions of both cores. A roll-back operation is deployed to recover the system to a previous safe state (or context) saved into the relevant BRAM memory during one of the checkpoints.

As in the case of the roll-forward operation, the interrupt mechanism is employed to access ARM cores' contexts individually. While the interrupt routine for the roll-back operation is executed, specific stack memory locations of *CPU0* and *CPU1* allocated to store register files are respectively overwritten with the corresponding register values, i.e. context, saved at certain locations within relevant BRAM memories. When an ARM core restores its context from the relevant stack memory at step IV of the interrupt mechanism, it would be recovered to a safe and healthy state. In this case, the application execution in both ARM cores returns to a previous verification point, and the relevant code execute block is re-executed, as illustrated in Fig. 6, where a mismatch is detected during *VP7* that causes a roll-back operation, therefore *Block7* is repeated in both ARM cores subsequently.

Under normal circumstances, the roll-back operation will recover the system to the immediately preceding checkpoint, e.g. *VP6* in Fig. 6. However, for some reason, that checkpoint may not constitute a safe state; therefore, the recovery would be unsuccessful. In such a case, the recovery will be made to the first checkpoint as discussed towards the end of subsection 3.4. This operation is referred to as roll-back first recovery, and is illustrated in Fig. 7 where the execution re-starts at *Block1* after *VP7*. If this does not work either, the checkpoint performed at the very beginning of the application will be the next destination for the recovery operation; this process amounts to an application reset where the application will be executed all the way from the start, which is clearly depicted in Fig. 8.

3.5.3. Soft system reset operation

If neither roll-forward nor roll-back operations have been successful in recovering the system due to a hang or crash in either ARM cores, the only remaining solution is to apply a system-wide soft reset via the configuration of the individual watchdog timers of ARM cores (i.e. *AWDT0* and *AWDT1*) [27], which is located within the processing subsystem (see Fig. 1), to fire after a short time. This configuration is expected to be carried out by the surviving ARM core. When either *AWDT0* or *AWDT1* fires at the set time, the ARM cores would be re-booted with the same application program and the programmable logic, i.e. FPGA,

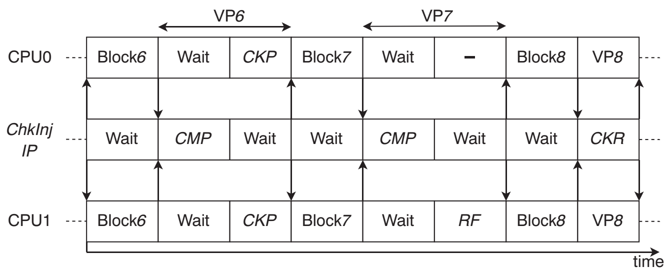


Fig. 5. Execution flow for the roll-forward operation (*CMP* = Compare, *CKP* = Checkpoint, *CKR* = Checker).

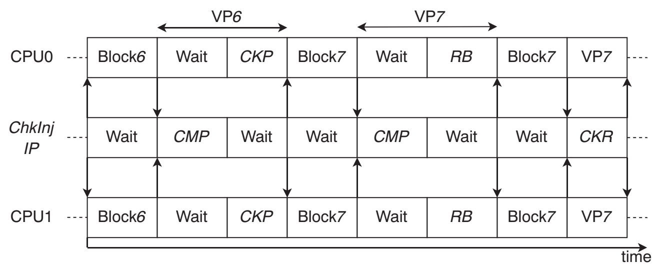


Fig. 6. Execution flow for the roll-back operation (*CMP* = Compare, *CKP* = Checkpoint, *RB* = Roll-back, *CKR* = Checker).

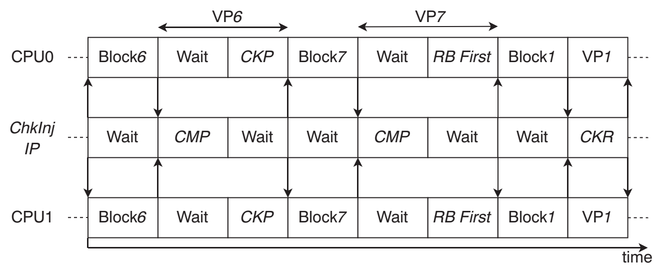


Fig. 7. Execution flow for roll-back to the first checkpoint (*CMP* = Compare, *CKP* = Checkpoint, *RB First* = Roll-back First, *CKR* = Checker).

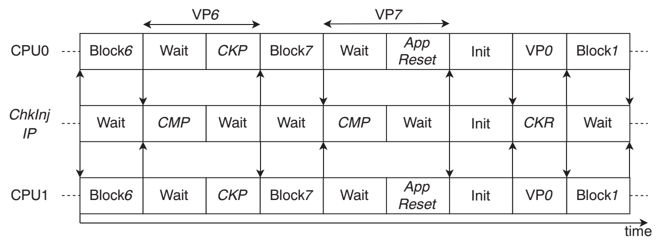


Fig. 8. Execution flow for the application reset operation (*CMP* = Compare, *CKP* = Checkpoint, *App Reset* = Application Reset, *CKR* = Checker).

part of the system would be reconfigured with the same bitstream; thus, the entire system will have a fresh start. Note that if both of the ARM cores are suffering a severe issue, then soft reset would not work as expected; the only option, in this case, would be power-cycling the entire system, i.e. a hard reset.

4. Fault injection technique

In order to evaluate the efficiency of the soft error mitigation provided by our TCLS approach, we have adopted a fault injection technique which emulates hardware faults by injecting bit-flips at the registers of the ARM cores within the processing subsystem of the Zynq APSoC. As the ARM cores under consideration constitute a hard-core Cortex-A9 processor, only a set of registers within it can be accessed for carry out fault injections. These target registers are the general-purpose (i.e. R0-R12) and special-purpose registers (i.e. SP, LR, PC) within the register file of an ARM core. Since these registers are widely used by any application, bit-flip injections to these register would likely cause an output error. The fault injection strategy adopted in our work is the same one as in [31] and employs the interrupt mechanism, which is detailed in subsection 3.3, in order to be as less intrusive as possible [32], [33]. The system architecture and methodology of this fault injection approach will be elaborated on next in this section.

4.1. Architecture

Fault injection experiments are taking place in the same environment as the one presented in Fig. 1 with an addition of a host computer connected through the UART peripheral core within the PS of the Zynq APSoC. The setup used to perform and analyze fault injection experiments is composed of the following two units:

- The *system logger* module: is a Python script running a host computer whose duty is to receive and store the outcomes of the fault injection experiments as transmitted from the Zynq APSoC through a UART-based serial communication.
- The *Checker-Injector* module: is the same module discussed in subsection 3.4, but this time it is operated in its second mode of operation as well as the first mode in order to perform the fault injection procedure following the methodology discussed in the next subsection.

4.2. Methodology

The methodology for the fault injection procedure mentioned above is composed of two main steps as described next. At the beginning of the application execution, 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 into which the fault injection will take place along with the number (from 0 to 31) of the specific bit to be flipped within that register. It is noteworthy to mention that the randomly evaluated injection time is relative to the execution time of the randomly selected code block; thus, a bit-flip can be injected at any time during that code block. Furthermore, the ability to select the code block results in a better-controlled fault injection process.

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 it hits the specified injection time. When the time is up, the *Checker-Injector* module individually interrupts both ARM cores. However, at the chosen ARM core only, the interrupt routine customized for the fault injection applies an XOR mask to the target register, thus flipping the specified bit in the register.

During our experiments, we have classified errors occurring due to the injected faults based on a scheme depicted in Table 2. In this scheme, the injected fault is labeled as *UNACE* (unnecessary for architecturally correct execution) when it does not affect either the register values or the output values of the ARM processor system. However, if the final output results generated by at least one of the ARM cores mismatch the golden results when compared at the end of the application, it is assumed that *SDC* (silent data corruption) has occurred during application execution. Furthermore, the case where an injected bit-flip causes a hang or a crash in the system is classified as *Hang*.

On the other hand, *Mitigated Faults w/ RF* and *Mitigated Faults w/ RB* occur when a mismatch in output results or register files of the ARM cores is detected and corrected by a roll-forward operation or a roll-back operation, respectively. Furthermore, *Mitigated Faults w/ RBF* represent faults which have been corrected by a roll-back to the first checkpoint,

Table 2
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

including the cases where an application reset have been applied. Moreover, *Mitigated Hangs* represent cases where a system hang or crash is detected and recovered by a soft system reset within the TCLS technique. Note that no bit-flip injection experiments were performed in the PL configuration memory of the Zynq chip since the focus of our work has been to verify the behaviour of the hard-core ARM Cortex-A9 processor as faults are injected into its registers.

5. Implementation and experimental results

This section presents the analysis of the implementation results, and describes the outcomes of the timing and fault injection performance experiments performed on TCLS-based design, referred to as *TCLS design*, and on other design setups. We have selected two benchmark applications to evaluate the timing and fault-injection performances of the proposed TCLS approach. The first benchmark application performs matrix multiplications, which are compute intensive operations widely used in real-life applications [34], while the second benchmark application encrypts electronic data using 256-bit Advanced Encryption Standard (AES) Algorithm [35] which is a memory-bounded algorithm that takes plain data in groups of 256 bits and converts them into ciphered data using keys of 256 bits.

Within each matrix-multiplication benchmark application, several matrix multiplication operations are performed on different input matrices made up of 32-bit signed data, where each full matrix multiplication operation corresponds to one code block, as shown in Fig. 3, surrounded by VPs. Within our experiments, we have considered benchmark applications operating with different matrix sizes (i.e. 20×20, 30×30, 40×40, 50×50, 60×60) and a different number of full matrix multiplication operations (i.e. 3, 6 and 12) to analyze how the block size and the number of block partitions affects the timing and soft-error recovery capacity of the proposed approach. On the other hand, the 256-bit AES benchmark application encrypts 32 integers in each block partition for a total of 3200 integer data over 10 blocks. Note that all benchmark applications are running in a bare-metal environment.

To compare and validate the efficiency of the TCLS approach, three other design versions have been set up in addition to *TCLS design*, namely *Unhardened design*, *Unprotected design* and Dual-Core Lockstep (DCLS) design (*DCLS design*). The *Unhardened design* version executes its applications only on *CPU0* where *BRAM Memory0* is used to store relevant application data. Therefore, it has no protection against soft errors other than TMR protection enabled on its BRAM memory. Furthermore, the *Unprotected design* version is equivalent to *TCLS design* with all protection mechanisms disabled, whereas the *DCLS design* version is on par with the design presented in [24], which does not support the feature of roll-forward operation as the MicroBlaze core has been removed. Note that *Unprotected design* is employed during fault injection experiments rather than *Unhardened design* because it still has reporting features enabled for the errors detected during verification points.

5.1. Resource consumption analysis

Tables 3 and 4 present resource consumption in terms of the number of LUTs, registers and slices, along with DSP block, Block RAM (BRAM) and ARM core utilization counts, for *Unhardened design* and *TCLS design*, respectively. Note that utilization counts of BRAM and controller pairs are listed under the major module (i.e. TMR *ChkInj IP* or TMR MicroBlaze) they are associated with.

Table 3
Resource consumption for unhardened design implementation

Resource Type	Total	Utilization Rate
ARM Cores	1	50%
Slice LUTs	1218	2.3%
Slice Registers	1165	1.1%
Slices	564	4.3%
Block RAMs	48	34.3%
DSP Blocks	0	0.0%

Table 4
Resource consumption for TCLS design implementation (Application Processor Configuration).

Resource Type	TMR ChkInj IP	TMR MicroBlaze	Total	Utilization Rate
ARM Cores	–	–	2	100%
Slice LUTs	4491	22,132	28,683	53.9%
Slice Registers	2648	18,341	23,286	21.9%
Slices	1615	6950	8775	66.0%
Block RAMs	96	23	119	85.0%
DSP Blocks	0	18	18	8.2%

As shown in the given tables, *TCLS design* requires considerably more resources compared to *Unhardened design* mainly because of the triplicated MicroBlaze core involved that is configured to be an application processor with a Memory Management Unit (MMU) where all types of exceptions and arithmetic operations are supported. The benefit, however, is that it supports running applications with high run-time performance requirements under any bare-metal or operating system environment. Nonetheless, if a typical microcontroller configuration is chosen for the MicroBlaze cores which does not include a MMU and supports minimal types of exceptions, significantly less resource usage can be achieved (see Table 5) while still preserving the same performance as its arithmetical unit is not compromised. The only caveat with the typical MicroBlaze configuration is that it does not support any traditional operating system.

Under the application processor settings, the TMR MicroBlaze module consumes over 22 K and 18 K slice LUTs and registers, respectively, along with 23 BRAMs and 18 DSP blocks. However, when MicroBlaze cores are typically configured as mentioned, LUT and register consumption of the TMR MicroBlaze module considerably drops to 17 K and 13 K, respectively, with a slight reduction in the count of BRAMs and no change in the consumption of DSP blocks. For both cases, TMR *ChkInj IP* (i.e. TMR *Checker-Injector* module) uses much fewer slice resources than TMR MicroBlaze; however, it is the module which consumes the highest count of BRAMs because BRAM memories employed by TMR *ChkInj IP* are protected against soft errors through the application of the TMR technique, as well. Finally, the resource overhead of *TCLS design* is 100% in terms of the ARM core utilization in any case. Note that the resource utilization of some auxiliary modules is not individually listed in Tables 4 and 5, but rather included in the total count column.

5.2. Timing performance analysis for matrix multiplication benchmarks

Table 7 reports timing figures in milliseconds (ms) as required by *Unhardened design* and *TCLS design* to execute matrix multiplication benchmarks mentioned above, under a fault-free scenario, for five different matrix sizes and three different application sizes in terms of the number of block partitions. Table 7 also presents the percentage performance overhead of *TCLS design* for each case with respect to *Unhardened design*. Note that for *TCLS design*, compiler optimization level *O3*

Table 5
Resource consumption for TCLS design implementation (Typical Processor Configuration).

Resource Type	TMR ChkInj IP	TMR MicroBlaze	Total	Utilization Rate
ARM Cores	–	–	2	100%
Slice LUTs	4500	17,160	23,730	44.9%
Slice Registers	2649	13,474	18,420	17.3%
Slices	1584	5504	7516	56.5%
Block RAMs	96	20	116	82.9%
DSP Blocks	0	18	18	8.2%

has been employed for the subroutines of the benchmark programs evaluating matrix multiplication operations in order to boost the performance; level *O0* has been used on the other hand for the remaining parts of the benchmarks to disable any optimization in the source code which has potential to corrupt the lockstep-based execution of *TCLS design*. However, benchmark programs were entirely compiled with level *O3* for *Unhardened design* to facilitate a realistic comparison.

Clearly, timing performance overheads are considerably higher when the matrix size (i.e. block size) is very small, for instance, overheads reach up to 96.4%, 122.9% and 155.9% for the applications sizes of 12, 6 and 3 blocks, respectively, when the matrix size of 20×20 is chosen. However, as the block size is increased, time overheads tend to fall significantly, as low as to 25.7% at the matrix size of 60×60 for the case of 12 block partitions. This point leads us to the conclusion that timing efficiency in *TCLS design* is achieved when the execution time of the useful computation (e.g. matrix multiplication) has a higher fraction over the total block execution time relative to redundant operations (i.e. consistency check and checkpoint operations) inside a verification point. Table 6 supports this conclusion by presenting execution times of a single block & VP case, for varying matrix sizes, along with percentage ratios of VP over block execution times. It is clear that as the matrix size (i.e. block size) increases, the VP ratio drops to 12.1% as opposed to 62.6% for the smallest block size.

Another conclusion can be drawn from Table 7; as more code execute blocks are employed within an application, that is to say, as the application size grows, timing performance overheads of *TCLS design* become more favourable for the given matrix size. Finally, although time overheads associated with *TCLS design* may not suit some hard real-time systems, these overheads would be tolerable for many systems requiring high reliability and dependability under harsh environments once block and application sizes are appropriately adjusted through trial and error based on the nature of the given application program.

5.3. Fault-injection performance analysis for matrix-multiplication benchmarks

An intensive fault injection run was carried out in Xilinx Zynq-7000 APSoC mounted on the PYNQ-Z2 board for three design setups in order to evaluate and validate the soft error resiliency of the proposed TCLS approach via the matrix-multiplication benchmark which is a compute-intensive application. Tables 8, 9 and 10 present the fault injection results for *Unprotected design*, *DCLS design* and *TCLS design*, respectively, with error margins (EM) and confidence intervals (CI) for the 95th percentile, where over 3000 runs of 50×50 matrix-multiplication benchmarks were performed for each design with the application size of 12 blocks. Note that one bit-flip was injected per application run employing the mechanism explained in Section 4.

The given tables, along with the bar chart provided in Fig. 9 to aid visualization, demonstrate that rates for *SDC*s and *Hang*s are quite high, i.e. 10.3% and 30.9%, respectively, for *Unprotected design*, while *DCLS design* and *TCLS design* have been able to significantly lower the rates of *SDC*s and *Hang*s down to as low as 0.1% and 0.9%, respectively, for a compute-intensive application, due to their possession of several error protection mechanisms (see Section 3). Although *DCLS design* and *TCLS design* mitigate hangs & crashes almost at the same rate by a soft system reset, the total application rate of roll-back and roll-back first operations is not same for these two designs, which is 52.0% for *DCLS design* and 44.7% for *TCLS design*. This drop of 7.3% is due to the provision of roll-forward feature in *TCLS design*, which is very beneficial for considerably reducing the overall execution time of the application under exposure to fault injections. The rationale behind this reduction is that the re-execution of even one block partition, which is mandated by roll-back based recovery operations, can waste many milliseconds depending on the block size as proven in Table 6. This drop in the execution time under faulty conditions will result in a higher MWBF which is defined to be the amount of data precisely processed before a failure happens [36]; this is because more data can be processed before a fatal error when a

Table 6
Single block and VP computation times for TCLS design.

Matrix Size	Block Time (ms)	VP Time (ms)	VP Ratio
60×60	2.87	0.35	12.1%
40×40	1.34	0.21	15.6%
20×20	0.48	0.19	39.3%
10×10	0.19	0.12	62.6%

Table 7
Timing Performance comparison of the TCLS design with respect to Unhardened design.

Application Size (# Blocks)	Matrix Size	Unhardened Design Execution Time (ms)	TCLS Design Execution Time (ms)	TCLS Design Overhead
12	60×60	32.62	41.01	25.7%
	50×50	22.64	30.21	33.4%
	40×40	14.54	19.70	35.5%
	30×30	8.20	12.91	57.5%
	20×20	3.70	7.27	96.4%
6	60×60	18.54	23.81	28.4%
	50×50	12.83	17.59	37.1%
	40×40	8.27	11.57	39.8%
	30×30	4.66	7.75	66.3%
	20×20	2.10	4.69	122.9%
3	60×60	11.32	15.30	35.2%
	50×50	7.85	11.28	43.7%
	40×40	5.05	7.65	51.4%
	30×30	2.85	5.17	81.4%
	20×20	1.27	3.25	155.9%

Table 8
Fault injection results for 50×50 matrix multiplication with no protection enabled (Unprotected design).

	Count	Rate	EM	95% CI
UNACE	2183	58.8%	1.58%	57.2% - 60.4%
SDC	384	10.3%	0.98%	9.4% - 11.3%
Hang	1145	30.9%	1.49%	29.4% - 32.3%
Total	3712			

Table 9
Fault injection results for 50×50 matrix multiplication with no roll-forward correction enabled (DCLS design).

	Count	Rate	EM	95% CI
UNACE	816	26.1%	1.54%	24.5% - 27.6%
SDC	3	0.1%	0.10%	0.0% - 0.2%
Hang	27	0.9%	0.32%	0.5% - 1.2%
Roll-Back	1613	51.5%	1.75%	49.8% - 53.3%
Roll-Back First	16	0.5%	0.25%	0.3% - 0.8%
Soft System Reset	657	21.0%	1.43%	19.6% - 22.4%
Total	3132			

Table 10
Fault injection results for 50×50 matrix multiplication with full protection enabled (TCLS design).

	Count	Rate	EM	95% CI
UNACE	811	26.7%	1.57%	25.2% - 28.3%
SDC	3	0.1%	0.10%	0.0% - 0.2%
Hang	28	0.9%	0.34%	0.6% - 1.3%
Roll-Forward	194	6.4%	0.87%	5.5% - 7.3%
Roll-Back	1303	43.0%	1.76%	41.2% - 44.7%
Roll-Back First	50	1.7%	0.45%	1.2% - 2.1%
Soft System Reset	645	21.3%	1.46%	20.0% - 22.7%
Total	3034			

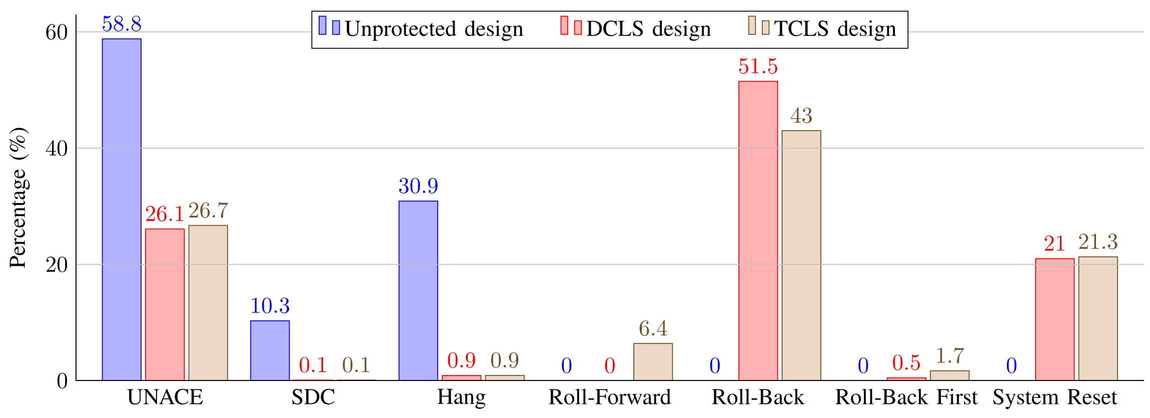


Fig. 9. Comparison of fault injection experiment results for designs with different protection schemes – 50×50 matrix multiplication application.

recoverable error is treated faster with the roll-forward operation rather than an operation based on roll-back recovery.

The *TCLS design* version exhibits almost the same fault injection performance when executing matrix multiplication applications for different matrix sizes as proven by stacked bar charts in Fig. 10. Notice from the given tables and charts that *SDC*s occur very seldom, i.e. 0.1% of the time, for *TCLS design*, but they do still happen because faults injected into the special-purpose registers falsely directs the application execution to the end. In such a misdirection case, since all protection mechanisms are bypassed, and output results are not entirely computed, *SDC*s are most possibly detected when comparisons with the golden output data are performed at the end. Furthermore, some hangs and crashes cannot be recovered by *TCLS design*, albeit infrequently. Such *Hang*s occur when a bit-flip injected into critical bits of the special-purpose registers cause severe data or prefetch aborts which cannot be recovered even when a soft system reset is applied.

The effect of bit-flips on individual registers has also been analyzed in this work, and reported in the form of stacked bar charts shown in Figs. 11 and 12 for *Unprotected design* and *TCLS design*, respectively, where same benchmark application as in Fig. 9 was harnessed for the sake of consistency. All general-purpose and special-purpose registers (i.e. SP, LR and PC) have been included in the analysis where it has been considered that R11 is used as Frame Pointer (FP) during execution to control stack access mechanism along with SP; thus, R11 is treated separately from other general-purpose registers which are encoded as

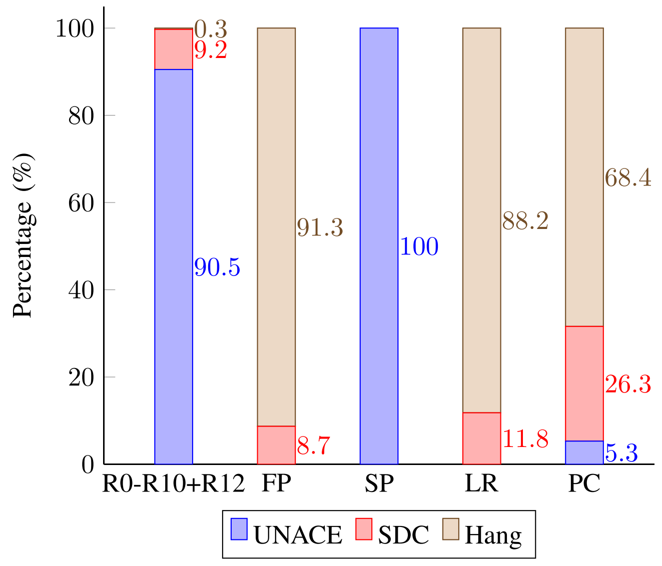


Fig. 11. Fault injection experiment results for different registers – design with no protection (Unprotected design), 50×50 matrix multiplication.

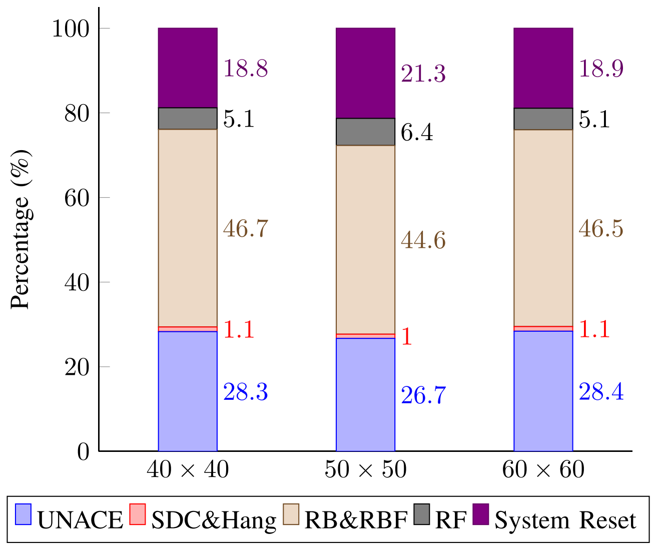


Fig. 10. Comparison of fault injection experiment results for different size matrix multiplication applications – fully protected design (TCLS design).

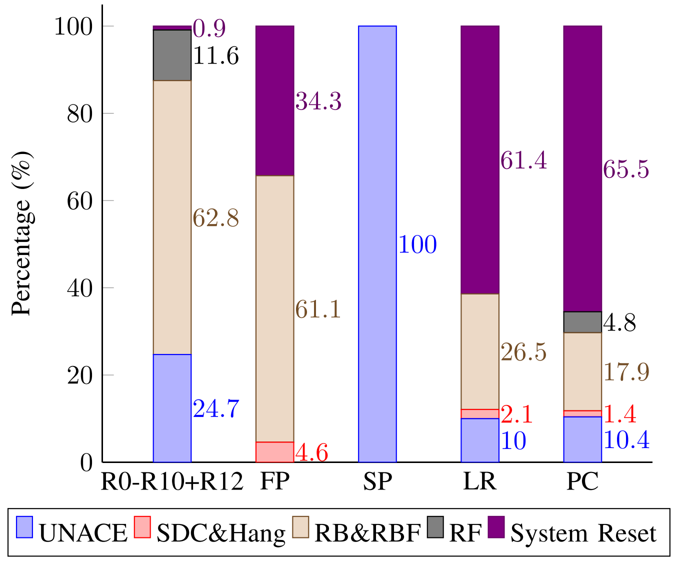


Fig. 12. Fault injection experiment results for different registers – fully protected design (TCLS design), 50×50 matrix multiplication.

$R0$-$R10+R12$ in the figures. As observed, 9.2% and 0.3% of bit-flips result in *SDC*s and *Hang*s, respectively, for general-purpose registers in *Unprotected design*, whereas both of these rates are effectively annulled in the *TCLS design* primarily via roll-back and roll-forward operations. Furthermore, fault injections into SP do not cause errors for both designs because SP is updated by FP on return from a subroutine. Therefore, bit-flips in FP have serious effects in *Unprotected design*, that is to say, *Hangs* occur over 91% of the time when FP is affected by fault injections. However, these hangs or crashes are significantly reduced in *TCLS design* with the application of soft system resets and roll-back operations. Finally, bit-flips in LR and PC cause high rates of *SDC* (up to 26.3%) and *Hang* (up to 88.2%) in *Unprotected design* as anticipated. The total rate of these errors are lowered to as small as 1.4% in *TCLS design*, which is achieved overwhelmingly by soft system resets, i.e. over 60% of the time.

If we compare the proposed approach with a similar work in [24], where two hard-core ARM Cortex-A9 processors are used without a TMRed MicroBlaze processor to support the roll-forward recovery operation, a decrease in the process disruptions are observed; for a similar matrix-multiplication application, hangs or SDCs occur at the rate of 2.59% in [24] whereas these incidents are observed at a fraction of that rate, i.e. at 1% of time (0.9% due to hangs and 0.1% due to SDCs), within our protection scheme. Furthermore, we provide a premise for a roll-forward recovery at the rate of 6.4% for the same matrix-multiplication experiments, which reduces the overall execution time compared to an approach based merely on roll-back recovery as in [24] for the reasons explained above.

5.4. Fault-injection performance analysis for 256-bit AES encryption benchmarks

Another intensive fault injection run was carried out in Xilinx Zynq-7000 APSoC for two design setups in order to evaluate and validate the soft error resiliency of the proposed TCLS approach, this time, via a memory-bounded application, i.e. the aforementioned 256-bit AES encryption benchmark. Tables 11 and 12 present the fault injection results for *Unprotected design* and *TCLS design*, respectively, with error margins (EM) and confidence intervals (CI) for the 95th percentile, where over 3000 runs of 256-bit AES encryption of plain data were performed for each design with the application size of 10 blocks. Again, one bit-flip was injected per application run.

A bar chart is also provided in Fig. 13 based on the given tables which proves that *TCLS design* has been successful in reducing the rates of *SDC*s and *Hangs* down to 0.7% and 1.2%, respectively, for a memory-bounded application, from the corresponding rates of 20.4% and 31.1% for *Unprotected design*. However, compared to the fault-injection experiments conducted for the compute-intensive application in Section 5.3, the rate at which a roll-forward recovery operation are called to mitigate soft errors detected in the system has risen from 6.4% to 18.9% for *TCLS design*. This increase in the roll-forward recovery rate directly translates into a system performance increase at a similar rate for the given application in terms of MWBF, compared to the case where no roll-forward recovery is supported.

Actually, all memory-bounded applications will enjoy a better MWBF thanks to the proposed approach because as data memories are frequently accessed by the application under faulty conditions, the *Checker-Injector* module would more often detect mismatches across the intermediate output data of ARM cores, i.e. *CPU0* and *CPU1*, stored within *BRAM Memory0* and *BRAM Memory1* memories (see Fig. 1). Consequently, more roll-forward recovery operations are triggered than roll-back recovery operations when faults are occurring due to radiation, which significantly improves the system MWBF for the reasons explained in Section 5.3.

As a last note, an alternative approach to employing ARM Cortex cores in radiation environments is to utilize TMR-based protection schemes in the FPGA fabric which seems to present better reliability than our approach to a certain extent. The work in [37] presents good estimations of functional failures when different versions of TMR are used to implement Cortex-M0 soft-core processors on Xilinx 7-series FPGA; it is shown that TMR can mask upsets on the FPGA configuration memory while incurring a hardware footprint 6.7 times larger than the unhardened version. However, the accumulation of bit-flips in the configuration memory can still lead to faults on multiple modules and overcome the TMR masking capability over the time. Even a distributed fine-grain TMR implementation reinforced with configuration memory scrubbing provides 100% of reliability till the accumulation of merely five faults; further accumulation of bit-flips in the configuration memory will overcome the TMR masking capability, thus the reliability will drop considerably. These results indicate that TMRed versions of ARM Cortex soft-cores in the FPGA fabric cannot sustain 100% reliability too long while incurring an area overhead of 6.7× even for the simplest and tiniest ARM Cortex core. Clearly, application-grade ARM Cortex processors which are widely used in radiation environments would not even fit into the FPGA fabric when its hardware is triplicated in a course-grain fashion under a TMR-based protection scheme.

Thus, we can assert that our approach which combines hard-core ARM Cortex-A9 processor with TMRed MicroBlaze processor in the FPGA fabric presents an optimal solution against radiation-induced soft errors where hard-core ARM processor is not affected by the bit-flips occurring in the FPGA configuration memory, while TMRed MicroBlaze processor enables the roll-forward based recovery which improves the mean workload between two failures. Note that even when the TMRed MicroBlaze processor fails due to the excessive bit-flips in the configuration memory, our approach can still protect the process on the ARM cores by degrading itself to the dual-core lockstep (DCLS) technique and merely supporting roll-back based recovery schemes.

Table 11
Fault injection results for 256-bit AES encryption with no protection enabled (Unprotected design).

	Count	Rate	EM	95% CI
UNACE	1660	48.5%	1.68%	46.9% - 50.2%
SDC	697	20.4%	1.35%	19.0% - 21.7%
Hang	1063	31.1%	1.55%	29.5% - 32.6%
Total	3420			

Table 12
Fault injection results for 256-bit AES encryption with full protection enabled (TCLS design).

	Count	Rate	EM	95% CI
UNACE	650	21.3%	1.45%	19.8% - 22.7%
SDC	20	0.7%	0.29%	0.4% - 1.0%
Hang	35	1.2%	0.38%	0.8% - 1.5%
Roll-Forward	578	18.9%	1.39%	17.5% - 20.3%
Roll-Back	1412	46.3%	1.77%	44.5% - 48.0%
Roll-Back First	15	0.5%	0.25%	0.2% - 0.7%
Soft System Reset	343	11.2%	1.12%	10.1% - 12.4%
Total	3053			

6. Conclusion

All-Programmable System-on-Chips (APSoCs) are a desirable implementation choice for systems utilised in nuclear environments due to their high-performance computing and power efficiency merits. Despite the advantages APSoCs possess, they are sensitive to radiation like any other electronic device. Processors incorporated in APSoCs, therefore, should be well hardened against radiation to become a viable option for unfavourable environments. This paper introduces a novel triple-core lockstep (TCLS) approach to accomplish fault tolerance for the dual-core ARM Cortex-A9 processor in the Xilinx Zynq-7000 APSoC against soft errors; this is achieved by coupling the ARM processor with a

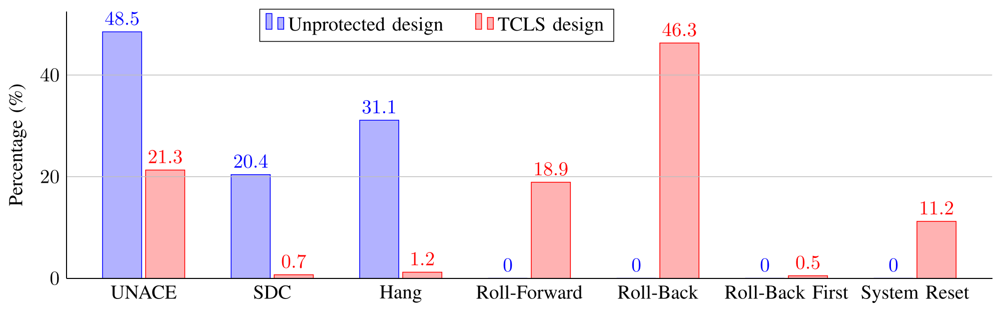


Fig. 13. Comparison of fault injection experiment results for unprotected and fully protected designs – 256-bit AES encryption application.

MicroBlaze TMR subsystem implemented in the FPGA logic. The proposed technique uses the concepts of checkpointing along with roll-back and roll-forward mechanisms at the software level (i.e. software redundancy), and the processor replication and checker circuits at the hardware level (i.e. hardware redundancy).

Fault injection experiments have been performed to evaluate the proposed TCLS approach. The results confirm that the given approach has enhanced the reliability and availability of the hard-core ARM processor with a high rate (i.e. around 98%) of corrected and recovered faults. Furthermore, timing performance overhead is as low as 25% under fault-free conditions when block and application sizes are adjusted appropriately. Moreover, integrating the roll-forward process into the system results in up to ≈19% higher MWBF which is because, when handled with the roll-forward operation rather than the roll-back operation, the programme will progress quicker in the cases of fault occurrence. Thus, more data can be computed before the next error occurs. Note that memory-bounded applications tend to benefit more from this approach in terms of better MWBF because, in this case, the *Checker-Injector* module would more often detect mismatches across the intermediate output data of ARM cores stored within their allocated BRAM memories, thus triggering a roll-forward recovery operation rather than a roll-back operation when a soft error occurs.

As future works, we plan to submit our system to neutron radiation experiments in order to validate the approach in real harsh environments. Furthermore, operating benchmark applications on a chosen embedded Linux OS or RTOS distribution is envisaged to evaluate shifts in the timing and soft-error correction performances of the proposed TCLS approach under operating system environments as compared to bare-metal. Finally, we are working upon adapting our approach for Zynq UltraScale+ MPSoCs where we can take advantage of the available heterogeneous processing subsystem incorporating both a quad-core ARM Cortex-A53 processor and a dual-core ARM Cortex-R5 processor.

CRediT authorship contribution statement

Server Kasap: Conceptualization, Data curation, Investigation, Methodology, Software, Validation, Writing – original draft, Writing – review & editing. **Eduardo Weber Wächter:** Investigation, Methodology, Validation, Writing – review & editing. **Xiaojun Zhai:** Supervision, Resources, Writing – review & editing. **Shoaib Ehsan:** Funding acquisition, Writing – review & editing. **Klaus D. McDonald-Maier:** Funding acquisition, Project administration, Supervision, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

This work is supported by the UK Engineering and Physical Sciences Research Council through grants EP/P017487/1, EP/R02572X/1 and EP/V000462/1.

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] Fukushima Daiichi nuclear power plant accident. https://www.scmp.com/news/asia/east-asia/article/2077394/dying-robots-and-failing-hope-fukushima-clean-falters-six-years.
[3] R. Baumann, Soft errors in advanced computer systems, IEEE Des. Test Comput. 22 (3) (May 2005) 258–266.
[4] T. Li, J.A. Ambrose, R. Ragel, S. Parameswaran, Processor design for soft errors: challenges and state of the art, ACM Comput. Surv. 49 (3) (Nov. 2016), https://doi.org/10.1145/2996357 [Online]. Available:.
[5] S. Hauck, A. DeHon, Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation. 1em plus 0.5em minus 0.4em, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.
[6] D.K. Pradhan, N.H. Vaidya, Roll-forward and rollback recovery: performance-reliability trade-off, IEEE Trans. Comput. 46 (3) (March 1997) 372–378.
[7] D.K. Pradhan (Ed.), Fault-tolerant Computer System Design. 1em plus 0.5em minus 0.4em, Prentice-Hall Inc., Upper Saddle River, NJ, USA, 1996.
[8] T. Nidhin, A. Bhattacharyya, R. Behera, T. Jayanthi, K. Velusamy, Understanding radiation effects in SRAM-based field programmable gate arrays for implementing instrumentation and control systems of nuclear power plants, Nucl. Eng. Technol. 49 (8) (2017) 1589–1599.
[9] M. Wirthlin, High-reliability FPGA-based systems: space, high-energy physics, and beyond, Proc. IEEE 103 (3) (March 2015) 379–389.
[10] S. Kasap, E.Weber Wächter, X. Zhai, S. Ehsan, K. Mcdonald-Maier, Survey of soft error mitigation techniques applied to LEON3 soft processors on SRAM-based FPGAs, IEEE Access 8 (2020), pp. 28 646–28 658.
[11] C. Carmichael, Triple Module Redundancy Design Techniques for Virtex FPGAs, Xilinx Inc., San Jose, CA, USA, 2006. XAPP197 Application Note.
[12] L. Parra, A. Lindoso, M. Portela-Garcia, L. Entrena, B. Du, M.S. Reorda, L. Sterpone, A new hybrid nonintrusive error-detection technique using dual control-flow monitoring, IEEE Trans. Nucl. Sci. 61 (6) (Dec 2014) 3236–3243.
[13] H. Quinn, Z. Baker, T. Fairbanks, J.L. Tripp, G. Duran, Robust duplication with comparison methods in microcontrollers, IEEE Trans. Nucl. Sci. 64 (1) (Jan 2017) 338–345.
[14] Kuang-Hua Huang, J.A. Abraham, Algorithm-based fault tolerance for matrix operations, IEEE Trans. Comput. C-33 (6) (June 1984) 518–528.
[15] J.R. Azambuja, M. Altieri, J. Becker, F.L. Kastensmidt, HETA: hybrid error-detection technique using assertions, IEEE Trans. Nucl. Sci. 60 (4) (Aug 2013) 2805–2812.
[16] E. Chielle, G.S. Rodrigues, F.L. Kastensmidt, S. Cuenca-Asensi, L.A. Tambara, P. Rech, H. Quinn, S-SETA: selective software-only error-detection technique using assertions, IEEE Trans. Nucl. Sci. 62 (6) (Dec 2015) 3088–3095.
[17] G.A. Reis, J. Chang, D.I. August, Automatic instruction-level software-only recovery, IEEE Micro 27 (1) (2007) 36–47.
[18] F. Restrepo-Calle, A. Martínez-Álvarez, S. Cuenca-Asensi, A. Jimeno-Morenilla, Selective SWIFT-R, J. Electron. Test. 29 (2013) 825–838.
[19] G.C. Clark, J.B. Cain, Error-correction Coding for Digital Communications, 1st ed., Springer Publishing Company Inc., New York, NY, USA, 1981.
[20] H.H. Ng, PPC405 Lockstep System on ML310, Xilinx Inc., San Jose, CA, USA, 2007. XAPP564 Application Note.
[21] F. Abate, L. Sterpone, C.A. Lisboa, L. Carro, M. Violante, New techniques for improving the performance of the lockstep architecture for SEEs mitigation in FPGA embedded processors, IEEE Trans. Nucl. Sci. 56 (4) (Aug 2009) 1992–2000.
[22] M. Violante, C. Meinhardt, R. Reis, M.Sonza Reorda, A low-cost solution for deploying processor cores in harsh environments, IEEE Trans. Ind. Electron. 58 (7) (July 2011) 2617–2626.

[23] H. Pham, S. Pillement, S.J. Piestrak, Low-overhead fault-tolerance technique for a dynamically reconfigurable softcore processor, IEEE Trans. Comput. 62 (6) (June 2013) 1179–1192.
[24] A.B. de Oliveira, G.S. Rodrigues, F.L. Kastensmidt, N. Added, E.L.A. Macchione, V. A.P. Aguiar, N.H. Medina, M.A.G. Silveira, Lockstep dual-Core ARM A9: implementation and resilience analysis under heavy ion-induced soft errors, IEEE Trans. Nucl. Sci. 65 (8) (Aug 2018) 1783–1790.
[25] 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, Microelectron. Reliab. 100–101 (2019), 113447.
[26] E.W. Wächter, S. Kasap, X. Zhai, S. Ehsan, 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.
[27] Zynq-7000 SoC, Xilinx Inc., San Jose, CA, USA, 2018. UG585 Technical Reference Manual.
[28] TUL PYNQ-Z2 board. http://www.tul.com.tw/ProductsPYNQ-Z2.html.
[29] L.A. Tambara, P. Rech, E. Chielle, J. Tonfat, F.L. Kastensmidt, Analyzing the impact of radiation-induced failures in programmable SoCs, IEEE Trans. Nucl. Sci. 63 (4) (Aug 2016) 2217–2224.
[30] ARM Cortex-A Series Programmer's Guide v4.0, ARM Inc., Cambridge, UK, 2013.
[31] Á.B. de Oliveira, L.A. Tambara, 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.
[32] S. Rezgui, R. Velazco, R. Ecoffet, S. Rodriguez, J.R. Mingo, Estimating error rates in processor-based architectures, IEEE Trans. Nucl. Sci. 48 (5) (2001) 1680–1687.
[33] R. Velazco, S. Rezgui, R. Ecoffet, Predicting error rate for microprocessor-based digital architectures through C.E.U. (Code emulating Upsets) injection, IEEE Trans. Nucl. Sci. 47 (6) (2000) 2405–2411.
[34] 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, M. Wirthlin, Using benchmarks for radiation testing of microprocessors and FPGAs, IEEE Trans. Nucl. Sci. 62 (6) (2015) 2547–2554.
[35] M. Dworkin, E. Barker, J. Nechvatal, J. Foti, L. Bassham, E. Roback, J. Dray, Advanced Encryption Standard (AES), 2001, 2001-11-26.
[36] J. Lienig, H. Bruemmer, in: Fundamentals of Electronic Systems Design, Springer International Publishing, Cham, 2017, pp. 45–73, ch. Reliability Analysis.
[37] L.A.C. Benites, F. Benevenuti, A.B. De Oliveira, F.L. Kastensmidt, N. Added, V.A. P. Aguiar, N.H. Medina, M.A. Guazzelli, Reliability calculation with respect to functional failures induced by radiation in TMR arm cortex-M0 soft-Core embedded into SRAM-based FPGA, IEEE Trans. Nucl. Sci. 66 (7) (2019) 1433–1440.