Microelectronics Reliability 120 (2021) 114120

Contents lists available at ScienceDirect

Microelectronics Reliability

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

Lock-V: A heterogeneous fault tolerance architecture based on Arm and RISC-V☆

Ivo Marques, Cristiano Rodrigues, Adriano Tavares, Sandro Pinto, Tiago Gomes *

Centro ALGORITMI, University of Minho, Portugal

ARTICLE INFO

Keywords:
Dual-core lockstep
Design diversity
Fault tolerance
Field-programmable gate array
RISC-V
Arm

ABSTRACT

This article presents Lock-V, a heterogeneous fault tolerance architecture that explores a dual-core lockstep (DCLS) technique to mitigate single event upset (SEU) and common-mode failure (CMF) problems. The Lock-V was deployed in two versions, Lock-VA and Lock-VM by applying design diversity in two processor architectures at the instruction set architecture (ISA)-level. Lock-VA features an Arm Cortex-A9 with a RISC-V RV64GC, while Lock-VM includes an Arm Cortex-M3 along with a RISC-V RV32IMA processor. The solution explores field-programmable gate array (FPGA) technology to deploy softcore versions of the RISC-V processors, and dedicated accelerators for performing error detection and triggering the software rollback system used for error recovery. To test Lock-V in both versions, a fault-injection mechanism was implemented to cause bit-flips in the processor registers, a common problem usually present in heavy radiation environments.

1. Introduction

Since the genesis of computing science, electrical systems are continuously subject to reliability problems. While in the beginning, these were due to components' nature, such as vacuum tubes and relays [1,2], nowadays, and despite the ever-growing semiconductor technology, such problems are mainly associated with the highly increased systems' hardware and software complexity. This new silicon era keeps providing reduced transistor's size, higher clock frequencies, lower operating core voltages, and hardware components with lower power consumption and higher performance ratios. However, systems are now more sensitive to single event upset (SEU) errors, which can be induced by radiation phenomena, causing bit-flip problems [3]. Dependability, one of the main properties of a computing system, consists of the ability of a system to be trustworthy and reliable by avoiding failures that can last longer than acceptable or even be harmful to people and the environment. Achieving dependability through security can be easily attained with virtualization-based techniques, specially through static partitioning schemes used to enhance real-time systems, where different levels of criticality can be assigned to virtual partitions over the same hardware resources [4–9]. However, these approaches usually lack in exploring design diversity to protect against common-mode failure (CMF), which consists of a fault that simultaneously affects all components and propagates without being detected. Dependability can be achieved using other mitigation approaches, such as fault prevention, fault tolerance, fault removal, and fault forecasting [10], which are more suitable to prevent SEU errors and CMF.

Regarding fault tolerance systems, their main goal is to continue operating properly in the event of active faults, preventing them from propagating to failures. This can be achieved by deploying error detection and recovery mechanisms [10,11]. Error detection can be performed by using concurrent or preemptive strategies. While in the first strategy the detection mechanism executes in parallel with the normal system delivering, the preemptive detection suspends the system's delivery and checks for errors or dormant faults. Achieving error detection requires the comparison of the output from redundant components, deployed both in hardware and software [12]. In some strategies, this is performed by comparing the output from lockstep operations, detecting faults when there are two running systems, known as dual-core lockstep (DCLS), or detecting faults and automatically correcting errors via a voting mechanism that analyzes the output from three systems, known as triplecore lockstep (TCLS). Concerning the error recovery, DCLS approaches can restore the system integrity to a previous state, e.g., using rollback techniques.

☆ This work has been supported by FCT - *Fundação para a Ciência e Tecnologia* within the R&D Units Project Scope: UIDB/00319/2020.

* Corresponding author.
E-mail addresses: ivo.marques@algoritmi.uminho.pt (I. Marques), cristiano.rodrigues@algoritmi.uminho.pt (C. Rodrigues), atavares@dei.uminho.pt (A. Tavares), sandro.pinto@dei.uminho.pt (S. Pinto), mr.gomes@dei.uminho.pt (T. Gomes).

https://doi.org/10.1016/j.microrel.2021.114120
Received 16 November 2020; Received in revised form 28 February 2021; Accepted 1 April 2021
Available online 16 April 2021

Current strategies to develop fault tolerance systems include the combination of different error detection and system recovery approaches [10,11,13]. However, dealing with CMF errors requires design diversity in the redundant modules performing the same functionality [13]. CMF errors can be caused by any source that creates dependencies among the redundant components, making them vulnerable to the same faults, e.g., power sources or shared hardware resources [14]. The design diversity is usually achieved by: (1) applying time diversity, which introduces execution cycle delays between both processors [15]; (2) using micro-architectural diversity [16]; and (3) deploying instruction set architecture (ISA) diversity [17,18]. The combination of these solutions can endow safety/mixed-critical systems with higher protection levels against SEU, mitigating CMF by using fault tolerance along with design diversity.

This article presents Lock-V, a fault tolerance solution highly robust against SEU and CMF deployable on both low- and high-end devices. Lock-V contributes to the state-of-the-art with: (1) a loosely-coupled DCLS system with design diversity at ISA-level that combines two different class of processors, an Arm Cortex-A9 with a RISC-V RV64GC (Lock-VA), and an Arm Cortex-M3 along with a RISC-V RV32IMA processor (Lock-VM); (2) a fault injection mechanism to evaluate the proposed architecture outside a real case scenario; and (3) a detailed evaluation and comparison of both implementations in terms of performance, error detection, and system recovery capabilities. However, Lock-V does not address hardened by design at component- or circuit-level, e.g., leveraged by Self Restoring Logic flip-flops and latch-up resistant SRAM. Also, no configuration scrubbing is deployed, the lockstep accelerator is not TMR-protected, and other dependability issues like safety and security were not mitigated. If demanded, safety and security issues could be mitigated by deploying Lock-V under Bao hypervisor [9]. Bao is a minimal, standalone and clean-slate implementation of the static partitioning architecture for Armv8 and RISC-V platforms.

2. Related work

Bit-flip problems, early present in aerospace environments, have been thoroughly addressed over the years [19–22]. Regarding lockstep systems, many fault tolerance solutions already exist in the literature [15,21,23–25], following a tightly- or loosely-coupled approach. In a tightly-coupled strategy, the comparison is performed in each system clock, while with a loosely-coupled the comparison can be only done periodically through a checkpoint system [23]. Some processor architectures already provide hardcore lockstep capabilities to assist fault tolerance systems, while other solutions resort FPGA to deploy and customize softcore processors along with extra hardware features to support the lockstep system [16,26,27].

Lock-V provides a fault tolerance architecture deployed into two processor architectures, using redundancy and design diversity at the ISA-level (Arm and RISC-V). Targeting low- and high-end devices, Lock-V improves the methodologies and techniques initially proposed in [18], and, for the best of our knowledge, there are no similar implementations beyond the contributions provided by this work. Table 1 shows fault tolerance systems that are closely related to Lock-V, summarizing them in terms of architecture, lockstep approach, redundancy strategy, and design diversity support.

2.1. Tightly-coupled approaches

The solutions provided by [15,16,29] use a DCLS system following a time-diversity approach. For instance, Yiu [15] implements a delay of 2 clock cycles between two Arm Cortex-M7 processors. Kottke et al. [16] propose a DCLS solution deployed in FPGA with two softcore processors that implement a delay of 1.5 clock cycles between cores. By using time diversity, these solutions perform faster error detection, mainly against to CMF. The architecture in [27] includes a DCLS with two MicroBlaze softcore processors deployed in the FPGA. The architecture also includes a comparator circuit to detect errors, and a multiplexer to connect the processors' outputs. Furthermore, this system uses TCLS with three softcore PicoBlaze units in the configuration engine for running without any errors.

Table 1
Gap analysis between existing lockstep solutions and Lock-V.

	Architecture		Lockstep		D. diversity
	Core	FPGA	Type[a]	Redundancy	Yes/no
Abate [21]	Hardcore	Yes	L	DCLS	No
Hanafi [28]	Hardcore	Yes	L	DCLS	No
Yiu [15]	Hardcore	No	T	DCLS	No
Kral [23]	Hardcore	Yes	L	DCLS	No
Oliveira [24]	Hardcore	Yes	L	DCLS	No
Han [29]	Hardcore	No	T	DCLS	No
Kottke [16]	Softcore	Yes	T	DCLS	No
Cornejo [26]	Softcore	Yes	T	DCLS	No
Pham [27]	Softcore	Yes	T	DCLS/TCLS	No
Iturbe [25]	Hardcore	No	T	TCLS	No
Ainsworth [30]	–	–	L	MMR	Yes[b]
Lock-V	**Both**	**Yes**	**L**	**DCLS**	**Yes**

[a] T - tightly-coupled; L - loosely-coupled.
[b] Design diversity at microarchitectural level.

In [25], it is used a tightly-coupled TCLS system with three Arm Cortex-R5. Due to the safety-critical nature of the Arm Cortex-R5, each core includes hardware mechanisms to deal with errors, as well as a DCLS system. In contrast to DCLS solutions, this solution has a recovery system that executes without any software intervention.

2.2. Loosely-coupled approaches

Some loosely-coupled lockstep implementations use a DCLS system with a hardcore processor beside an FPGA, which is used to implement custom modules to support the synchronization and comparisons between cores [21,23,24]. In [30], the lockstep system proposes design diversity at microarchitectural level. The proposed architecture is composed of a main high-performance core that executes in parallel with small multiple checker units. The mechanism to detect errors consists in verifying each checker application fragment, while the main core executes the entire application. The design diversity is applied with partial replication of the main core between the checker units. Despite the solution proposing a high-performance technique for fault tolerance by using a new way of parallelism in the state-of-the-art, it is not yet implemented and the main core is currently unavailable.

3. Lock-V

Lock-V is a fault tolerance DCLS system that follows a checkpoint and recovery strategy while exploring design diversity at ISA-level. Due to their architectural differences, processor's registers cannot be directly compared, and the error detection cannot be performed at instruction level (in a tightly-coupled way). Therefore, it must use checkpoints along the application execution flow to output specific data/register values for the comparison. The Lock-V solution, as depicted in Fig. 1, is divided into the ***Code Generation*** and the ***Hardware Architecture*** modules.

The ***Hardware Architecture*** module includes the Arm and RISC-V processor units, the *xLockstep* accelerator with DCLS capabilities which is not itself resilient to reliability failures, and memory mapped Advanced Microcontroller Bus Architecture (AMBA) interfaces to connect the processors with the *xLockstep* accelerator. Moreover, each processor unit includes an independent cache and memory system. Despite the existing solutions to provide the I/O interface to redundant systems [12,23], protecting such interface is out of the scope of this work. The ***Code Generation*** module consists of the application's code

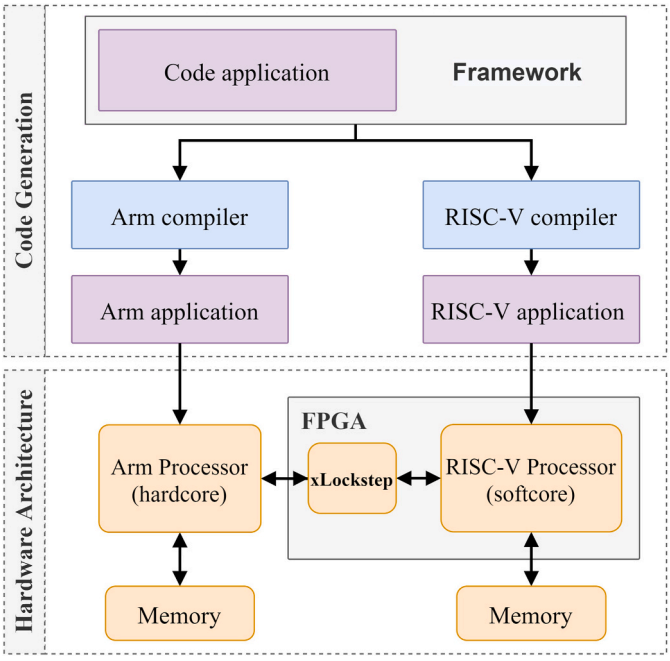

Fig. 1. Lock-V DCLS fault tolerance solution.

and the Framework. The Framework is responsible for adapting applications for the Lock-V architecture, i.e., it provides services for the error recovering capabilities, before its compilation to the target architecture.

3.1. Hardware architecture: xLockstep accelerator

The DCLS capabilities are provided by the *xLockstep* accelerator (Fig. 2), which is composed of the following modules: (1) ***synchro***, (2) ***checker***, and (3) ***finite state machine (FSM)***. The *xLockstep* accelerator has also a set of data registers to assist the error comparison process, and two AMBA interfaces, one for each core, which is used to communicate with both processors. Each interface includes a set of memory-mapped registers for control, status, and data operations.

3.1.1. Synchro

The *Synchro* module includes a counter that is responsible for synchronizing both cores, which is activated when one of the processors reaches a checkpoint. If the counter reaches the defined timeout before the second processor reaching its own checkpoint, the *Synchro* module flags a timeout error, also known as a hang error, which occurs when a fault originates a system crash or reaches an infinite loop. Otherwise, there are no errors and the system can follow its normal execution. The timeout value corresponds to a 32-bit register that is loaded when the first core reaches the checkpoint and it can be configured by the Framework. The timeout may vary according to the final application needs and it must be defined accordingly. A good start point is to evaluate the worst execution time between each checkpoint and set the timeout with twice this value. An incorrect timeout value may cause hang errors and halt the application execution.

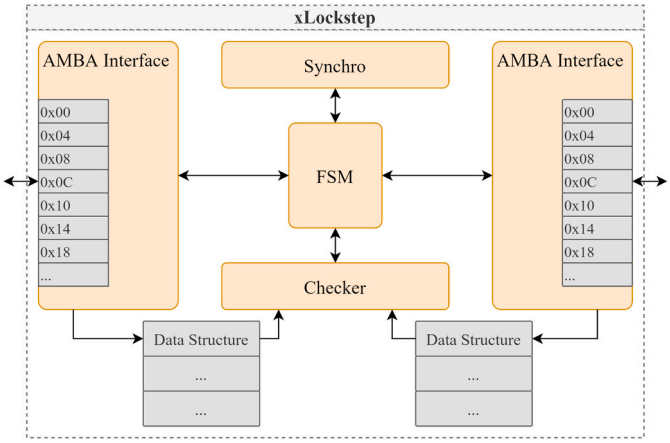

Fig. 2. xLockstep architecture overview.

3.1.2. Checker

The *Checker* module performs the error detection functionalities by comparing a set of data registers (32-bit width) from both processors. If a silent data corruption (SDC) occurs, which corresponds to a fault that caused an error in the comparing outputs, the module flags an error state. This module presents some differences between the Lock-VA and the Lock-VM implementations. Lock-VA uses a LIFO-based data structure to compare simultaneously multiple 32-bit word registers, while the Lock-VM implementation can only compare four 32-bit registers in each clock-cycle.

3.1.3. Finite state machine

The FSM module, responsible for the management and control of the *xLockstep* accelerator, is composed by five main states: (1) *Idle*, (2) *Synchro*, (3) *Checker*, (4) *Resume*, and (5) *Error*. The *xLockstep* commutes from *Idle* to *Synchro* state when the first processor reaches a checkpoint. In this state, the *xLockstep* starts the synchronization process, where one of the following situations can occur to the second processor: (1) the checkpoint is reached within the timeout, and the *xLockstep* changes to the *Checker* state; or (2) the timeout value is reached, meaning that the checkpoint was not reached in time by the second processor. In this case, the *xLockstep* changes to the *Error* state. During the *Checker* state the *xLockstep* compares the output from both processors. If the outputs mismatch, the *Checker* module flags an error and changes to *Error* state, otherwise, the FSM follows to the *Resume* state. In the *Error* state, the *xLockstep* waits for the system to be restored to a previous state of integrity. Once the error is fixed, the synchronization can be reached, and the *xLockstep* can switch again to *Resume* state. In the *Resume* state and after the synchronization is completed by both processors, the *xLockstep* resumes its execution and changes to *Idle* state.

3.2. Code generation: Lock-V framework

While the error detection capabilities are handled in hardware by the *xLockstep* accelerator, the recovery capabilities are performed via software by the Lock-V Framework. The error detection and the system recovery functionalities are available through the following set of services:

initLockV(): initializes the *xLockstep* accelerator and triggers the synchronization process;
checkpoint(): notifies the *xLockstep* when a checkpoint is reached, sends its output for the comparison task, and waits for the synchronization state. The synchronization is achieved when both processors reach the checkpoint and the data to be compared are consistent in both cores. On a successful synchronization, a new saveContext() is performed. Otherwise, a rollback() must be executed.
saveContext(): creates a restore point by saving the processor's context according to its ISA implementation;
rollback(): restores the system to its last state of integrity, previously saved by the *saveContext()* service;
errorFix(): notifies the *xLockstep* accelerator when an error is fixed.

Since Lock-V follows a loosely-coupled approach, mainly due to the exploration of design diversity at ISA-level, the software requires checkpoints throughout the application code to allow the integrity

verification on both processors. This enables the system restoring to a previous integrity state through a rollback operation. Fig. 3 shows the execution flow of an application running in the Lock-V DCLS architecture using the services provided by the Framework. The system starts with the synchronization between both processors. If the synchronization properly succeeds, the processor's contexts are saved and the application starts running. When a checkpoint is reached and if there are no errors in the processors' output, a new processor's context is saved and the application runs until the next checkpoint. On a checkpoint state, if it is detected an active fault that affects the application and corrupts the data in one of the processors, the rollback system is triggered and the execution is taken from the previous checkpoint where a valid context can be recovered. The checkpoints should be carefully chosen by the user according to the application needs and must be manually patched at the chosen critical code points. Checkpoint insertions are recommended around message passing between software tasks, or when the application interacts with the external world.

3.3. System recovery

Achieving full system recovery demands all the application data to be saved. However, this may require a considerable amount of data to be reliably stored in memory, causing great overheads in the saving and restoring processes. To avoid saving and restoring the whole system data, some hardware platforms already provide memory with safety and data protection mechanisms such as Error Correcting Code (ECC) and TMR-based memories, among others [19,31]. Therefore, and assuming that the Lock-V can resort to such memory systems, the great source of errors is likely to be from the processors' register file, one of the most critical parts of the processor [32–36]. These errors usually occur due to SEU that cause bit-flips problems. Thus, context saving and rollback mechanisms implemented by the Lock-V target only the processor's register file.

Although the system's memory can be protected from external faults, e.g., by using ECC memories, stored data can still be affected by the propagation of faults from the register file during the save and restore processes [36]. Such problems can be magnified when processors with load/store architectures are used, since all instructions and register operations are only performed through memory accesses. To avoid error propagation from registers to the memory, Lock-V restricts its utilization to register-related operations, which means that only local variables can be used, stored either in registers or in the stack. Thus, and besides protecting the register file, the system recovery also needs to protect the stack. Saving the stack and register file represents the minimum memory required to ensure the proper operation of a lightweight recovery system, preventing register file faults from propagating to the memory.

3.4. Context saving

Storing processors' context is performed by saving the register file and stack. Although the logic behind this process being the same for both cores, the utilization of different processor architectures dictates different implementations of the context saving and rollback mechanisms. Fig. 4 depicts the saving context operation on an Arm's processor architecture. When the **saveContext** service is invoked, the main Frame Pointer (FP) and the Link Register (LR) are saved in the function stack. Afterward, a copy of the register file is stored, which also includes the LR and FP registers. Next, the main FP and Stack Pointer (SP) are used to store the stack. In order to copy all the stack data, a pointer is assigned with the value of the base of the stack (hold by the FP). After that, another pointer is assigned with the top of the stack (hold by the SP). Afterward, a third pointer is assigned with the base address of the saved stack. The stack is then saved, word by word, until the base pointer (r1) matches the top pointer (r0). From now, a copy of the register file and the **main()** stack is safely stored to be used in a context restore (rollback) operation. Regarding the memory required by the final application in the context saving process, the register file and stack backups must be stored in protected ECC or TMR memories. However, remaining data must be saved somewhere according to their values.

3.5. Context restoring (rollback)

In order to perform the processors' rollback, depicted by Fig. 5, the stack and register file need to be restored by the same order used in the context saving process, i.e., the stack is handled before the register file. If the register file is restored first, some registers can be corrupted. The rollback operation can be triggered in two situations: (1) when the *xLockstep* detects an error in the processors' output; and (2) when the processor detects illegal operations, undefined instructions, load or store operations to illegal memory addresses, etc. When these faults occur, the processor enters into an exception handler (Arm) or a trap (RISC-V). When an error is detected by the *xLockstep*, a rollback needs to be performed in both cores. This is required since in DCLS systems it is not possible to detect which core is the error source. However, when an exception or trap occurs, the rollback just needs to be performed in the

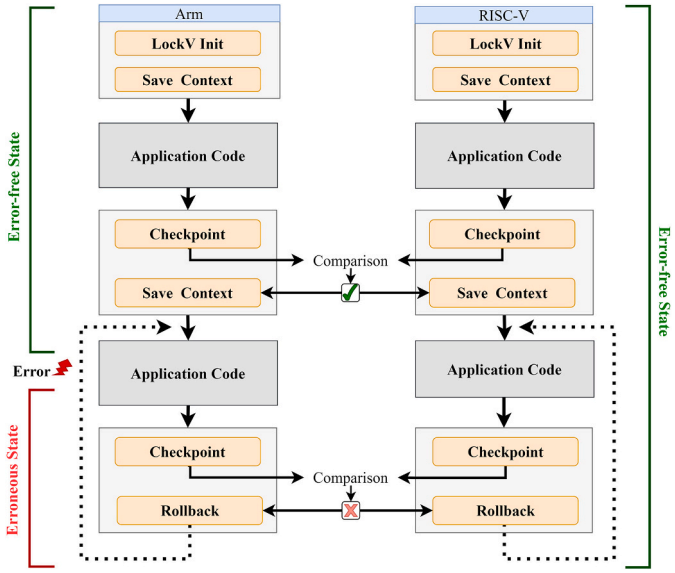

Fig. 3. DCLS execution flow with a rollback situation.

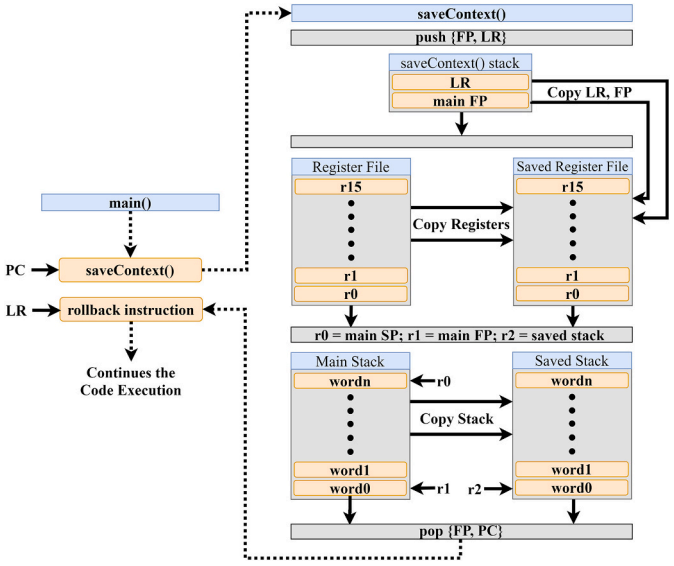

Fig. 4. Context saving on a Arm's processor.

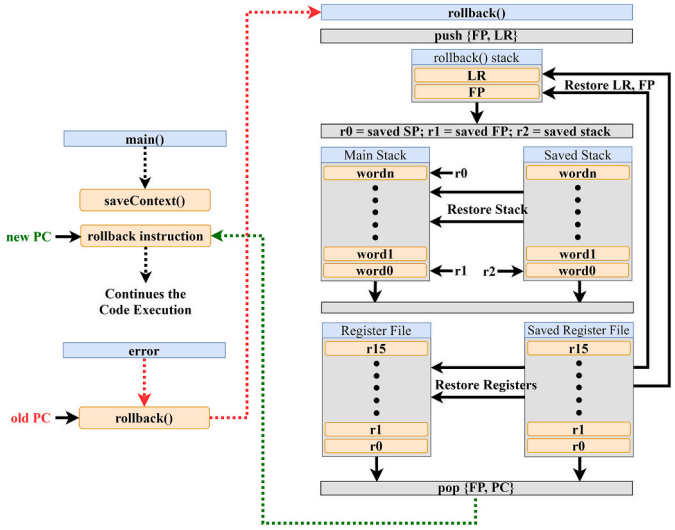

Fig. 5. Context restoring on a Arm's processor.

processor that originated the fault. Despite the existence of the rollback operation, some errors must be studied and handled accordingly, e.g., non-random errors systematically caused by the same source.

4. Evaluation

To test the functionality and adaptability of our solution to different classes of target devices while keeping the fault-tolerance capabilities, Lock-V was deployed and tested in two versions, targeting two different classes of processors: (1) Lock-VA was deployed on a Zynq-7000 programmable system-on-chip (SoC) by Xilinx, and (2) Lock-VM on a Smartfusion2 SoC from Microsemi. Due to implementation and platform constraints, Lock-VA also uses temporal diversity in the processors' clock, i.e., the Arm Cortex-A9 processor runs at 666 MHz, while the RISC-V runs at 25 MHz. In the Lock-VM, both processors, Arm Cortex-M3 and RISC-V, run at the same clock speed of 25 MHz. The performed experiments allowed to evaluate the functionality and characterize the Lock-V architecture within the following metrics: (1) FPGA resources required for deploying the accelerator and the RISC-V processor; (2) the generated memory footprint with and without Lock-V; and (3) the execution footprint, with and without the Lock-V architecture. Despite being important, at the current stage of this work, power consumption tests were left aside from this evaluation. Furthermore, and to properly evaluate the fault tolerance system, the Lock-V architecture was tested under a fault injection mechanism to simulate a real case scenario, avoiding performing complex heavy-ions irradiation tests.

4.1. FPGA resources utilization

Table 2 shows, for each implementation, the hardware resources required by the *xLockstep*, along with its sub-modules, and respective RISC-V processor. The results are expressed in terms of Look-Up Tables (LUT) and Flip-Flops (FF) in the Lock-VA implementation, while in the Lock-VM the same results are expressed in 4-Inputs Look-Up Table (4-LUT) and D-Type Flip-Flops (DFF), according to the platforms' nomenclatures. Lock-VA requires a total of 42,124 LUT and 35,850 FF, which corresponds to 79.20% and 33.70% of the available resources of the Zedboard Zynq-7000 SoC. Lock-VM requires a total of 13,494 4-LUT and 7516 DFF, which is 15.66% and 8.72% of the available 4-LUT and DFF of the SmartFusion2 M2S090TS SoC.

In both implementations, the softcore version of the RISC-V processor represents the component with higher resource utilization. For the Lock-VA, the RISC-V module (lowRISC) requires around 81% (34,138 out of 42,124) of LUT and nearly 46% (16,324 out of 35,850) of FF,

Table 2
Lock-VA and Lock-VM FPGA resources utilization.

Module	Sub-module	Lock-VA		Lock-VM	
		LUT	FF	4-LUT	DFF
xLockstep	AMBA interface 0[a]	122	269	158	224
	AMBA interface 1[a]	135	267	158	224
	Checker	148	90	129	12
	Synchro	11	6	81	35
	Extra	25	40	62	49
	Sub-total	**441**	**672**	**588**	**544**
	Percentage (100%)	0.83%	0.63%	0.68%	0.63%
Softcore RISC-V	LowRISC	34,138	16,324	–	–
	Mi-V RV32IMA	–	–	12,537	6774
Total		**42,124**	**35,850**	**13,494**	**7516**
Percentage (100%)		79.20%	33.70%	15.66%	8.72%

[a] Lock-VA uses AXI-Lite and Lock-VM uses APB3.

while in the Lock-VM, the RISC-C core (MI-V) represents nearly 90.13% and 92.91% of the used 4-LUT and DFF, respectively. The *xLockstep* accelerator is the hardware module that consumes less hardware resources. In the Lock-VA implementation it only requires 441 LUT and 672 FF, which for the entire implementation, is around 1% of the LUT, and nearly 2% of the FF. In Lock-VM, the *xLockstep* uses 588 4-LUT and 544 DFF, which corresponds to 4.36% 4-LUT and 7.24% DFF over the entire Lock-VM implementation.

4.2. Memory footprint

Table 3 presents the memory footprint (in bytes) for both implementations, with and without the Lock-V architecture. In the Lock-VA version, adding fault tolerance capabilities to the application, the memory footprint increases, on average, nearly 6% in Arm deployment, and 8.3% in the RISC-V. For the Lock-VM implementation, including fault tolerance in the application causes a slight increase of approximately 3600 bytes. More specifically, there is an increase of 3584 bytes in the Arm Cortex-M3 side, and an increase of 3596 bytes for the RISC-V implementation. This represents a memory increase of 5.6% and 29.6%, respectively.

Despite Arm and RISC-V deployments of the Lock-V framework practically generate the same memory footprint, there are still small differences between them, mainly due to the difference in processors' ISAs and application binary interface. Therefore, the binary machine code produced by the compiler is different. In the Lock-VA version, there are additional factors that increase both RISC-V and Lock-V memory overheads. While the RISC-V follows a 64-bit architecture (words of 8-byte width) with stack alignment of 16 bytes, the Arm complies with a

Table 3
Lock-VA and Lock-VM memory footprint.

	.text	.data	.bss	Total
Arm Cortex-A9				
Application without Lock-VA	19,552	1152	22,580	43,284
Application with Lock-VA	22,096	1216	22,580	45,892
Lock-VA overhead	2544	64	0	2608
Arm Cortex-M3				
Application without Lock-VM	4656	16	59,072	63,744
Application with Lock-VM	6384	80	60,864	67,328
Lock-VM overhead	1728	64	1792	3584
RISC-V LowRISC				
Application without Lock-VA	45,864	97	647	46,608
Application with Lock-VA	49,212	616	660	50,488
Lock-VA overhead	3348	519	13	3880
RISC-V Mi-V RV32IMA				
Application without Lock-VM	3824	128	8196	12,148
Application with Lock-VM	6624	256	8864	15,744
Lock-VM overhead	2800	128	668	3596

32-bit architecture (words of 4-byte width), with a stack alignment of 4 bytes. Furthermore, in the Lock-VM implementation, the Arm Cortex-M3 uses a stack alignment of 4 bytes, while the RISC-V follows a stack alignment of 8 bytes.

4.3. Execution footprint

Table 4 presents the execution footprint, in terms of clock cycles, required by the Lock-V Framework services. In Lock-VA, the measurements were performed in the RISC-V processor, since it represents the bottleneck on the implementation, while in the Lock-VM, the measurements were made in the Arm processor.

4.3.1. Context saving and rollback

In the Lock-VA, saving the processor context and stack has a cost of 3128 clock cycles, while in the Lock-VM a total of 335 clock cycles is required. Restoring the system through rollback uses 2852 and 248 clock cycles in Lock-VA and Lock-VM implementations, respectively. The differences in Lock-VA and Lock-VM are due to the amount of data that each implementation needs to save and restore. The Lock-VM solution requires a smaller stack, thus, less data has to be saved.

4.3.2. Checkpoint

Regarding the checkpoint mechanism, the impact of comparing different vector sizes (varying from 1 to 100 elements) in the processors' output data was evaluated. In the Lock-VA implementation, without errors in data to compare, the checkpoint task requires 10,420 clock cycles for one good element, but by each additional ten elements, the latency increases around 24,675 clock cycles. When the comparison mismatches, the checkpoint overhead increases, on average, around 1226 clock cycles. In the Lock-VM solution, the checkpoint uses 1459 clock cycles to compare one valid element, and by each added ten good elements, the system increases the required clock cycles by 1767. When the checkpoint compares data with errors, this service performs better and requires, on average, less than 22 clock cycles.

4.3.3. Calculating the Fibonacci sequence

Taking a practical example, the execution footprint was evaluated, with and without errors, by calculating inside a function the first 10, 15, and 20 elements of the well-known *Fibonacci* sequence. The scalability of the *Fibonacci* function allows understanding the impact of using the Lock-V with one or more checkpoints during its execution. The evaluation was performed by running the execution without errors in the sequence, and with an error in the sequence's first or last comparison element. Moreover, for each condition, the Lock-V was tested with one or *N* checkpoints, where *N* corresponds to the number of elements calculated by the function. For instance, when the *Fibonacci* function calculates the first ten elements and only one checkpoint is in use, it will only be reached after the calculation of the ten elements, and the checkpoint sends ten elements for comparison. If the system includes 10 checkpoints, each one will be reached after calculating each element, and thus, only one element is sent to the compare operation.

Table 4
Lock-V execution footprint.

		Lock-VA		Lock-VM	
		Clock cycles	@25 MHz (μs)	Clock cycles	@25 MHz (μs)
saveContext		3128	125.12	335	13.4
rollback		2852	114.08	248	9.92
Checkpoint					
	No. data				
No errors	1	10,420	416.8	1459	58.36
	10	32,851	1314.04	2154	86.16
	100	254,921	10,196.84	18,059	722.36
	Avg. increm.[b]	24,675	987	1767	70.68
With error[a]	1	11,698	467.92	644	25.76
	10	34,118	1364.72	2134	85.36
	100	256,079	10,243.16	18,041	721.64
	Avg. increm.[b]	24,675	987	1767	70.68
Overhead	1	1278	51.12	−815	−32.6
	10	1267	50.68	−20	−0.8
	100	1158	46.32	−18	−0.72

[a] Error in the last element to compare.
[b] Average increment in 10 elements.

Table 5 shows the obtained results for the execution overhead relative to the baseline test, i.e., the system without Lock-V functionalities. For instance, calculating the first 10 elements with no errors in the processor's output, causes an overhead increase of 14.11% in the Lock-VA and 4.25% in the Lock-VM implementation. For the worst-case scenario (N checkpoints) and the system with no errors, the overhead in the Lock-VA is 112,66% and 1.84% when calculating the Fib(10) and Fib(20) elements of the sequence. When the system has an error in the first element and uses N checkpoints, the overhead from Fib(10) to Fib (20) reduces from 148.40% to 2.09%. Furthermore, if the error is in the last element, the overhead decreases from 324,77% to 103.59%. This represents the worst- and best-case scenarios for the error detection, where the error can be in the last, or in the first element of the comparison values. The main reason for the higher values in calculating Fib (10) in comparison with the other calculations, lies in the fact that, in the Lock-VA, for small interactions and outputs, the system still needs to use all Framework functionalities, delivering all outputs to the *xLockstep* accelerator. This overhead is slowly dissipated when the number of iterations (checkpoints) increases.

Under the same conditions, the Lock-VM solution presents a smaller overhead than the Lock-VA implementation. This is directly related to architectural features of the Arm Cortex-M processors, exhibited in Lock-VM implementation, which requires less amount of data to be saved during the context saving operation. When there are no data errors and the system uses N checkpoints, the overhead decreases from 32.15% to 0.46%. And when there is an error in the first element and N checkpoints are used, the overhead for Fib(10) decreases to 41.00% and for Fib(20) it decreases around 0.54%. The worst scenario occurs when the error is in the last element and N checkpoints are used, where the overhead decreases from 165.67% to 100.94%. In both cases, with the increase of the application execution, the overhead of the Lock-V framework reduces drastically.

Curiously, when the error is in the first element, the overhead for using the Lock-V with N checkpoints is lower than when using just one. This is due the reduced execution granularity of the error detection, causing the system to detect and correct errors faster. When just one checkpoint is used, the verification can only be performed at the end of the program. Therefore, if an error occurs, the system can only perform a rollback operation when the program finishes its execution, resulting always in an overhead greater than 100%. In contrast, when N checkpoints are used, the error can be detected faster and the system can be recovered earlier.

4.4. Fault injection

In order to test and evaluate the Lock-V, a fault injection mechanism, based on [37], and a system monitor were added in both Lock-VA and Lock-VM implementations. This fault injection aims at emulating bit-flips that may occur in harsh environments due to SEU. Faults were only injected on the Arm processor, but it could have been done on the RISC-V side, or both. The goal is to force comparisons mismatch to cause a rollback, thus it is not relevant which processor originates corrupted data. Faults are randomly injected at any time, by using a timer interruption, causing random bit-flips in a random register of the Arm bank registers currently in use. On each round, a timer triggers the fault injection procedure which randomly chooses one bit of one of the 16 general purpose Arm registers to be flipped. This happens in the

Table 5
Execution overhead with *Fibonacci* function.

N. of checkpoints		Lock-VA			Lock-VM		
		Fib(10)	Fib(15)	Fib(20)	Fib(10)	Fib(15)	Fib(20)
No error	1	14.11%	1.02%	0.19%	4.25%	0.44%	0.04%
	N	112.66%	14.39%	1.84%	32.15%	4.07%	0.46%
Error (first elem.)	1	129.21%	102.38%	100.14%	109.36%	100.96%	100.09%
	N	148.40%	17.95%	2.09%	41.00%	4.86%	0.54%
Error (last elem.)	1	129.54%	102.38%	100.14%	109.27%	100.96%	100.09%
	N	324.77%	128.91%	103.59%	165.67%	108.24%	100.94%

following way: (1) the register file is copied; (2) the fault is injected in the replicated register file through a XOR operation with the register and the random bit to flip; and next, (3) the register file is restored with the fault injected.

In the Lock-VA implementation, the sparing Arm Cortex-A9 is used as a system monitor, while in the Lock-VM, a dedicated monitoring hardware accelerated with a UART interface was deployed in the FPGA. During the test run, both systems execute an application in parallel with the fault injection mechanism, while the monitor system tracks the number of faults and errors that occurred. Table 6 summarizes the results obtained from both systems. The main goal of this test is to force both SDC and hang errors. SDC errors occur when the data output from both cores is different, while hang errors occur when one of the cores does not reach the checkpoint. In the Lock-VA implementation, a total of 45,543 faults was injected, which have originated 933 errors, 137 by hang, and 796 SDC errors. When using the Lock-V architecture, the total number of errors reduces drastically, where only 31 errors (out of 933) were not corrected by the Lock-V mechanism (28 hang errors and 3 SDC errors). This result shows an error correction rate of nearly 97%.

For the Lock-VM tests, a total of 98,957 faults was injected, forcing 93 errors (85 SDC, and 8 hang errors). In the case of SDC errors, the system was able to correct 83 (out of 85) and for the hang errors, only 6 (out of 8) were corrected. For the total injected faults, 98,957, the Lock-VM can achieve an error correction rate of 95.7%. However, the fault injection process described above must be further extended towards more detailed analysis of soft errors in components of Xilinx Zynq-7000, as done in [38,39].

5. Conclusions and future work

This work proposes Lock-V, a loosely-coupled fault tolerance system that uses a DCLS technique with design diversity at the ISA-level, providing effective protection against SEU and CMF. Whereas typical lockstep approaches provide a strong fault tolerance technique, they lack in presenting protection against CMF. The proposed architecture was deployed and evaluated in two versions, Lock-VA that is composed of an Arm Cortex-A9 combined with a RISC-V RV64GC, and Lock-VM, which features an Arm Cortex-M3 along with a RISC-V RV32IMA processor. The fault injection system, applied to both approaches, revealed to be highly efficient in the error correction task, and lightweight in terms of execution overhead. Moreover, its small memory and execution footprints leverages the reliability of the lockstep system.

Due to its modular implementation, both in terms of hardware and software, Lock-V can be easily ported to other FPSoC systems and processor architectures with only minor architectural changes. Hereafter, the Lock-V architecture will be enhanced with the following features: (1) improve data transfers by resorting the direct memory access (DMA) mechanism; (2) provide support for more processor architectures; (3) improve the Framework functionality to support profiling features that can be used for a better checkpoint insertion, independent from the final application.

Table 6
Fault injection results with and without Lock-V.

Error type	Injected faults	Errors		Error correction
		Without Lock-V	With Lock-V	
Lock-VA				
Hang	–	137	28	79.56%
SDC	–	796	3	99.62%
Total	45,543	933	31	96.68%
Lock-VM				
Hang	–	8	2	75.00%
SDC	–	85	2	97.65%
Total	98,957	93	4	95.70%

CRediT authorship contribution statement

Ivo Marques: Methodology, Software, Investigation, Validation, Writing - Original draft, Writing - Reviewing and editing.
Cristiano Rodrigues: Methodology, Software, Investigation, Validation, Writing - Original draft, Writing - Reviewing and editing.
Adriano Tavares: Supervision, Resources, Conceptualization, Investigation, Writing - Reviewing and editing.
Sandro Pinto: Resources, Supervision, Investigation.
Tiago Gomes: Supervision, Resources, Writing - Original draft, Writing - Reviewing and editing, Investigation.

Declaration of competing interest

The authors have no affiliation with any organization with a direct or indirect financial interest in the subject matter discussed in the manuscript.

References

[1] A. Aviziens, Fault-tolerant systems, IEEE Trans. Comput. C-25 (12) (1976) 1304–1312.
[2] W.H. Pierce, Failure-tolerant Computer Design 247, Academic Press, 1965.
[3] E. Normand, Single event upset at ground level, IEEE Trans. Nucl. Sci. 43 (6) (1996) 2742–2750.
[4] G. Heiser, The role of virtualization in embedded systems, in: Proceedings of the 1st Workshop on Isolation and Integration in Embedded Systems, IIES '08, Association for Computing Machinery, 2008, p. 11–16.
[5] M. Masmano, I. Ripoll, A. Crespo, J. Metge, Xtratum: a hypervisor for safety critical embedded systems, in: Proceedings of the 11th Real-time Linux Workshop, 2009, pp. 263–272.
[6] R. West, Y. Li, E. Missimer, M. Danish, A virtualized separation kernel for mixed-criticality systems, ACM Trans. Comput. Syst. 34 (3) (Jun. 2016).
[7] R. Ramsauer, J. Kiszka, D. Lohmann, W. Mauerer, Look mum, no VM exits! (almost), in: Workshop on Operating Systems Platforms for Embedded Real-time Applications (OSPERT), 2017.
[8] S. Pinto, H. Araujo, D. Oliveira, J. Martins, A. Tavares, Virtualization on TrustZone-enabled microcontrollers? Voilà!, in: 2019 IEEE Real-time and Embedded Technology and Applications Symposium (RTAS), 2019, pp. 293–304.
[9] J. Martins, A. Tavares, M. Solieri, M. Bertogna, S. Pinto, Bao: a lightweight static partitioning hypervisor for modern multi-core embedded systems, in: M. Bertogna, F. Terraneo (Eds.), Workshop on Next Generation Real-time Embedded Systems (NG-RES 2020), Vol. 77 of OpenAccess Series in Informatics (OASIcs), Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2020, pp. 3:1–3:14.

[10] A. Avizienis, J. Laprie, B. Randell, C. Landwehr, Basic concepts and taxonomy of dependable and secure computing, IEEE Trans. Dependable Secure Comput. 1 (1) (2004) 11–33.
[11] M. Al-Kuwaiti, N. Kyriakopoulos, S. Hussein, Network dependability, fault-tolerance, reliability, security, survivability: a framework for comparative analysis, in: 2006 International Conference on Computer Engineering and Systems, 2006, pp. 282–287.
[12] E. Ozer, B. Venu, X. Iturbe, S. Das, S. Lyberis, J. Biggs, P. Harrod, J. Penton, Error correlation prediction in lockstep processors for safety-critical systems, in: 2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2018, pp. 737–748.
[13] E. Dubrova, Fault-tolerant Design, Springer-Verlag New York, 2013.
[14] L. M. Kaufman, S. Bhide, B. W. Johnson, Modeling of common-mode failures in digital embedded systems, in: Annual Reliability and Maintainability Symposium. 2000 Proceedings. International Symposium on Product Quality and Integrity (Cat. No.00CH37055), 2000, pp. 350–357.
[15] J. Yiu, Design of SoC for high reliability systems with embedded processors, in: Embedded World Conference, 2015.
[16] T. Kottke, A. Steininger, A reconfigurable generic dual-core architecture, in: International Conference on Dependable Systems and Networks (DSN'06), 2006, pp. 45–54.
[17] S. Mitra, N.R. Saxena, E.J. McCluskey, Common-mode failures in redundant VLSI systems: a survey, IEEE Trans. Reliab. 49 (3) (2000) 285–295.
[18] C. Rodrigues, I. Marques, S. Pinto, T. Gomes, A. Tavares, Towards a heterogeneous fault-tolerance architecture based on arm and RISC-V processors, in: IECON 2019 - 45th Annual Conference of the IEEE Industrial Electronics Society, Vol. 1 (2019) 3112–3117.
[19] R.C. Baumann, Radiation-induced soft errors in advanced semiconductor technologies, IEEE Trans. Device Mater. Reliab. 5 (3) (2005) 305–316.
[20] I. Hwang, S. Kim, Y. Kim, C.E. Seah, A survey of fault detection, isolation, and reconfiguration methods, IEEE Trans. Control Syst. Technol. 18 (3) (2010) 636–653.
[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).
[22] P. Garcia, T. Gomes, F. Salgado, J. Cabral, P. Cardoso, M. Ekpanyapong, A. Tavares, A fault tolerant design methodology for a FPGA-based softcore processor, IFAC Proc. Vol. 45 (2012) 145–150.
[23] R.D. Kral, J.S.M. Chong, A.L. Schreiber, Implementation of a loosely-coupled lockstep approach in the Xilinx Zynq-7000 all programmable SoCTM for high consequence applications, in: 42nd Annual GOMACTech Conference, Reno, NV, 2017.
[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).
[25] X. Iturbe, B. Venu, E. Ozer, S. Das, A Triple Core Lock-Step (TCLS) ARM® Cortex®-R5 processor for safety-critical and ultra-reliable applications, in: 2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop (DSN-W), 2016, pp. 246–249.
[26] J. Gomez-Cornejo, A. Zuloaga, U. Kretzschmar, U. Bidarte, J. Jimenez, Fast context reloading lockstep approach for SEUs mitigation in a FPGA soft core processor, in: IECON 2013 - 39th Annual Conference of the IEEE Industrial Electronics Society, 2013, pp. 2261–2266.
[27] H. Pham, S. Pillement, S.J. Piestrak, Low-overhead fault-tolerance technique for a dynamically reconfigurable softcore processor, IEEE Trans. Comput. 62 (6) (2013) 1179–1192.
[28] A. Hanafi, M. Karim, A. E. Hammami, Dual-lockstep microblaze-based embedded system for error detection and recovery with reconfiguration technique, in: Third World Conference on Complex Systems (WCCS), 2015, pp. 1–6.
[29] J. Han, Y. Kwon, Y. C. P. Cho, H. Yoo, A 1GHz fault tolerant processor with dynamic lockstep and self-recovering cache for ADAS SoC complying with ISO26262 in automotive electronics, in: 2017 IEEE Asian Solid-State Circuits Conference, 2017, pp. 313–316.
[30] S. Ainsworth, T. M. Jones, Parallel error detection using heterogeneous cores, in: 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2018, pp. 338–349.
[31] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, L. Alvisi, Modeling the effect of technology trends on the soft error rate of combinational logic, in: Proceedings International Conference on Dependable Systems and Networks, 2002, pp. 389–398.
[32] F.M. Lins, L.A. Tambara, F.L. Kastensmidt, P. Rech, Register file criticality and compiler optimization effects on embedded microprocessor reliability, IEEE Trans. Nucl. Sci. 64 (8) (2017) 2179–2187.
[33] M. A. Abazari, M. Fazeli, A. Patooghy, S. G. Miremadi, An efficient technique to tolerate MBU faults in register file of embedded processors, in: The 16th CSI International Symposium on Computer Architecture and Digital Systems, 2012, pp. 115–120.
[34] A. Ramos, A. Ullah, P. Reviriego, J.A. Maestro, Efficient protection of the register file in soft-processors implemented on Xilinx FPGAs, IEEE Trans. Comput. 67 (2) (2018) 299–304.
[35] G.P. Saggese, N.J. Wang, Z.T. Kalbarczyk, S.J. Patel, R.K. Iyer, An experimental study of soft errors in microprocessors, IEEE Micro 25 (6) (2005) 30–39.
[36] G. Memik, M. T. Kandemir, O. Ozturk, Increasing register file immunity to transient errors, in: Design, Automation and Test in Europe, 2005, pp. 586–591 Vol. 1.
[37] R. Velazco, S. Rezgui, R. Ecoffet, Predicting error rate for microprocessor-based digital architectures through C.E.U. (Code Emulating Upsets) injection, IEEE Transactions on Nuclear Science 47 (6) (2000) 2405–2411.
[38] L. A. Tambara, F. L. Kastensmidt, N. H. Medina, N. Added, V. A. P. Aguiar, F. Aguirre, E. L. A. Macchione, M. A. G. Silveira, Heavy ions induced single event upsets testing of the 28 nm Xilinx Zynq-7000 all programmable SoC, in: 2015 IEEE Radiation Effects Data Workshop (REDW), 2015, pp. 1–6.
[39] V. Vlagkoulis, A. Sari, J. Vrachnis, G. Antonopoulos, N. Segkos, M. Psarakis, A. Tavoularis, G. Furano, C. Boatella Polo, C. Poivey, V. Ferlet-Cavrois, M. Kastriotou, P. Fernandez Martinez, R.G. Alia, K.O. Voss, C. Schuy, Single event effects characterization of the programmable logic of Xilinx Zynq-7000 FPGA using very/ultra high-energy heavy ions, IEEE Trans. Nucl. Sci. 68 (1) (2021) 36–45.