Towards a Heterogeneous Fault-Tolerance Architecture based on Arm and RISC-V Processors

Cristiano Rodrigues, Ivo Marques, Sandro Pinto, Tiago Gomes, and Adriano Tavares
Centro ALGORITMI, University of Minho (PORTUGAL)
Corresponding author: mr.gomes@dei.uminho.pt

***Abstract*—Computer systems are permanently present in our daily basis in a wide range of applications. In systems with mixed-criticality requirements, e.g., autonomous driving or aerospace applications, devices are expected to continue operating properly even in the event of a failure. An approach to improve the robustness of the device's operation lies in enabling fault-tolerant mechanisms during the system's design. This article proposes Lock-V, a heterogeneous architecture that explores a Dual-Core Lockstep (DCLS) fault-tolerance technique in two different processing units: a hard-core Arm Cortex-A9 and a soft-core RISC-V-based processor. It resorts a System-on-Chip (SoC) solution with software programmability (available trough the hard-core Arm Cortex-A9) and field-programmable gate array (FPGA) technology, taking advantages from the latter to support the deployment of the RISC-V soft-core along with dedicated hardware accelerators towards the realization of the DCLS.**

***Index Terms*—Dual-core lockstep, fault tolerance, heterogeneous architectures, field programmable gate array, RISC-V, Arm.**

I. INTRODUCTION

Processors industry keeps moving fast towards reduced transistor's size, higher clock frequencies, and lower operating core voltages. However, many problems to digital systems have emerged due to such progress, like system failures caused by bit-flipping induced by many possible sources, e.g., radiation and voltage glitch. These problems can result in critical issues, not only in aerospace applications but also on daily basis systems [1]–[6]. This boosts research towards the necessity of developing and deploying fault tolerance systems in order to mitigate several failure situations, while keeping other important requirements such as system robustness, reliability, performance and security.

One way to deploy reliable devices in mixed-critical applications, is to provide them with fault tolerance techniques. Redundancy is one of the most used forms of fault tolerance mechanisms and several solutions can be already found in the literature. While some techniques replicate processing units in a technique called dual-core lockstep (DCLS) -implemented either loosely- or tightly-coupled to the processor- [4,7]–[11], others apply a triple modular redundancy (TMR) mechanism, where the processing units are triplicated and a voter module is added to the system [12]. Other techniques can be used in order to achieve fault tolerance systems, such as time redundancy applied to low-cost architectures [13], and virtualization-based systems [14], where several guests can virtually run over the same processing unit as if they were individually running each of them in one unique processor. This way, each guest operating system (OS) can replicate the execution of the same software application, while another guest acts as the voter module. These software-based systems can behave similarly to a hardware-based TMR without the need of replicating the hardware resources.

Fault tolerance techniques can be performed both in software and/or hardware, according to the available resources. With the ongoing technological trends, hybrid system-on-chip (SoC) solutions provide software programmability, available through hard-core processors, and field-programmable gate array (FPGA) technology that can be resorted for deploying soft-core processors or dedicated hardware accelerators in order to enhance the computation of several types of algorithms in terms of speed and energy consumption [15]–[17]. Despite several architectures and techniques for fault-tolerance being available in the literature, to the best of our knowledge, none of them targets heterogeneous architectures that resort hybrid SoC solutions to implement different processor architectures, either deployed in hard- or soft-core approaches.

This article presents the Lock-V, a heterogeneous architecture that explores a Dual-Core Lockstep (DCLS) fault tolerance technique in different processing units: a hard-core Arm Cortex-A9 and a soft-core RISC-V-based processors. The solution handles the system heterogeneity at different levels, such as at processors architecture (different instruction set architecture (ISA)), execution conditions, clock domains, etc. The available FPGA is used not only to deploy a soft-core processor, but also custom accelerators in a loosely-coupled fashion. These latter support the DCLS fault tolerance mechanism in order to synchronize and to verify the system's integrity during run-time execution.

The main contributions of this article are: (1) the Lock-V, a heterogeneous architecture that explores a DCLS fault tolerance technique; (2) the deployment of the DCLS on a hybrid SoC, providing support to multi-core heterogeneity (dual-core Arm Cortex-A9 processors, along with an untethered soft-core RISC-V-based processor, the lowRISC); (3) a loosely-coupled hardware accelerator, the xLockstep, used to support the DCLS architecture; and (4) the Lock-V framework that will allow the programmer to adapt the DCLS solution according to the application needs through the available application programming interface (API), and with future implementations of code-generation mechanisms.

978-1-7281-4878-6/19/$31.00 ©2019 IEEE

II. Background Overview

This section addresses three main topics to understand the development of the proposed fault tolerance system: (1) understanding the differences between a fault, an error and a failure; (2) the lockstep concept used as a fault tolerance technique; and (3) the RISC-V open-source ISA.

A. Fault, Error, and Failure

It is important to understand the concepts of fault, error and failures and how they can trigger a fault tolerance system to successfully recover from failure situations caused by errors. A fault tolerance system must continue to provide the specified service, even at the event of a fault, and should react to errors caused by faults, preventing the error propagation to a state of system failure. In [18] it is provided the main concepts and terminologies for the fault tolerance context:

- **Fault:** is defined as a logical manifestation caused by one or more physical defects, which change the normal operation of a component in a system;
- **Error:** is caused by one or more faults in a system when it transits into an internal state;
- **Failure:** occurs when some event deviates the delivered service from the specified service, a specified service is defined as a previously agreed description of the system behavior.

B. Redundancy and Dual-Core Lockstep (DCLS)

Fault tolerance characteristics can be added to a system by applying redundancy techniques, which can be both in hardware and/or software. Traditionally, hardware techniques use multiple instances of the same module, where each of them receives the same input and compares the generated output. However, such approaches can lead to a diagnostic decision for verifying if a fault had occurred when the outputs are different [19]. In the event of a fault, an error is usually generated, which is easily detected if the generated outputs are different. Depending on the implemented technique, the fault-tolerant system can adopt different approaches to perform a fast system recovery.

The most common redundancy hardware mechanisms are the duplication with comparison (DWC) and the TMR, which, respectively, duplicates and triplicates the execution of the main module. The main advantage of the TMR over the DWC is when fault occurs in just one execution module, the voter system can still determine which module produced the error and choose the valid output from the other two. However, this TMR-based solution is the most costly in terms of required hardware (and other related resources), since a third execution module and a voter must be added to the system [12]. The DCLS technique has proven to be a good option for fault tolerant systems, while requiring less resources than the TMR. It is a hybrid fault tolerance method based on the DWC, which uses dedicated hardware for error detection and core duplication. Likely the DWC, each core receives the same input and the extra hardware compares the output from each core, and when the outputs are different from each other, the mechanism detects an error. In case of error detection the system can: (1) continue its normal execution, giving priority to the output of one of the modules; (2) restore the system to a well-known integrity point (software execution checkpoint), which requires the system to create restore points; (3) completely restart or stop the system.

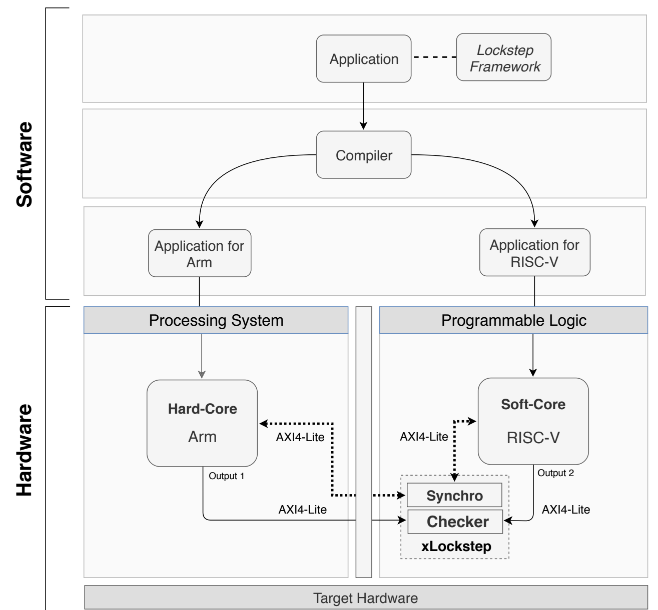

Fig. 1. Proposed DCLS heterogeneous architecture.

Despite all efforts in providing fault tolerance solutions, it is not possible to guarantee that a system is 100% error-free. In some techniques, a fault can affect redundant components all at the same time, which makes it hard to detect. This is known as a common-mode fault, and it can only be mitigated by introducing design diversity in the whole system. In a lockstep-based redundancy technique, this can be achieved by using different processors' architectures, as proposed by Lock-V.

C. RISC-V

RISC-V is an open-source ISA [20] based on a reduced instruction set computer (RISC). It was designed focusing embedded systems, Internet of Things (IoT), and other modern devices. RISC-V allows a new level of software and hardware freedom on architectures in an open extensible way. This ISA allows the implementation of RISC-V ISA-based cores and adapts them to fault tolerance techniques, in this case to DCLS. It is possible to create new processor instructions due to the architecture freedom, and target them to a specific purpose.

III. Lock-V

The Lock-V system, depicted in Fig. 1, can be split into two main components: the software block and the hardware block. Regarding the software, the Lockstep framework is responsible to generate the final machine binary code for a given application. Such binary, compiled for the two target

architectures (Arm and RISC-V), was generated and patched from the same source code application. The framework also provides a set of functionalities in order to allow users to insert and configure execution checkpoints in the source code. The checkpoints are predefined verification points, introduced prior the compilation time, in order to endow the system with lockstep functionalities. Such checkpoints are essential for the auxiliary mechanism of the DCLS architecture, the **Synchro** and **Checker** blocks. Their main tasks are, respectively, the synchronization of both cores and the verification of the processors' output in order to detect data integrity problems during code execution.

Regarding the hardware, the Lock-V is divided into two main areas, the processing system (PS) and the programmable logic (PL). The PS is mainly composed by a hard-core Arm Cortex-A9 processing unit and the associated software application. By its turn, the PL hosts a soft-core RISC-V processor (where the same software application also runs), and the hardware accelerators, which are responsible for deploying the lockstep functionalities, performed by the **Synchro** and **Checker** sub-modules. The PS and PL execute concurrently and are both connected through a standard advanced micro-controller bus architecture (AMBA) protocol, the advanced extensible interface (AXI), in order to exchange information among all hardware modules. The main hardware components of the Lock-V architecture are detailed as follows:

- **Arm Cortex-A9 processor:** a 32-bit processor that follows the ARMv7-A architecture and available in the PS as a hard-core processor. It runs the application machine binary code in parallel with the soft-core processor.
- **RISC-V processor:** a soft-core processor deployed in the FPGA fabric of the PL and it also runs the application code. This 64-bit processor is based on the lowRISC, an untethered implementation of the RISC-V ISA based on the Rocket Chip.
- **Lockstep accelerator (xLockstep):** a hardware accelerator deployed in the PL following a loosely-coupled approach, which was developed under the specification of the Chisel hardware construction language [21]. Such approach provides several advantages when compared with the tightly-coupled design, such as hardware customization, flexibility, and portability for using the xLockstep in other SoC and processor architectures. The xLockstep is responsible for the auxiliary lockstep mechanism and its main tasks are: (1) the synchronization of the code execution on both cores; (2) the comparison and verification of the outputs from each processor; (3) the control on the code execution when the compared outputs are validated and coherent; and (4) the ability to suspend the processors' execution when an error is found, until the error is processed and marked as solved.

A. *lowRISC*

Most of the freely available RISC-V soft-core implementations require host environment features, both for the booting process and for the processor to run and execute the application. Such implementations, e.g., Rocket Chip, are called tethered processors [22], as they require a host processor to start up and to interact with the environment. For this reason, we have selected the lowRISC core, which is an untethered processor built upon the Rocket Chip implementation of the RISC-V ISA that eliminates the need for a companion core, which is replaced with FPGA peripherals. The lowRISC processor has three important characteristics that fit with the soft-core requirements for the Lock-V: (1) it is an untethered soft-core processor, which is a key aspect for the implementation of the lockstep mechanism since each processor (Arm Cortex-A9 and RISC-V) have to execute their own binary machine code independently; (2) it is a 64-bit processor, different from the 32-bit Arm Cortex-A9; and (3) it is a customizable core, enabling the refactoring of the lowRISC processor to the project requirements, such as adding a master/slave Not A STandard Interface (NASTI) bus, which is similar to AXI, and tightly-coupled accelerators that can work as co-processors.

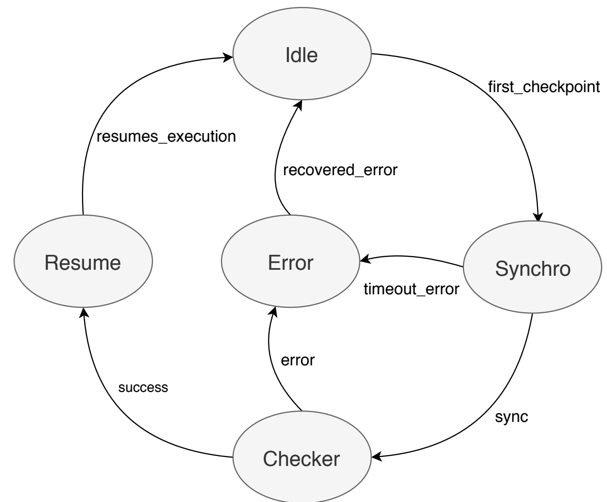

Fig. 2. Main finite state machine (FSM) of the xLockstep.

B. *xLockstep*

The xLockstep is a memory-mapped AXI-compliant peripheral deployed in the PL. It has two slave AXI-Lite interfaces, one for each processor. The accelerator has an exclusive bank of registers dedicated for each processor, being their access restricted by hardware. Therefore, each processor can only access their register bank. The xLockstep has more three sub-modules, two instances of the Synchro and one of the Checker modules. The module Synchro is responsible for ensuring that both processors are synchronized and the module Checker is responsible for comparing the output of both processors. Fig. 2 depicts the FSM of the xLockstep accelerator, which is composed of five states: Idle, Synchro, Checker, Resume, and Error. The FSM stays in the Idle state until the first checkpoint (from Arm or RISC-V processor) is reached. When this event occurs, the FSM changes the state to Synchro and waits for the second checkpoint to be reached, until a programmer-defined

timeout occurs. If that time is exceeded, an error by timeout in synchronizations is signalized and the FSM changes to the Error state. If the timeout is not exceeded, the FSM changes to Checker state. In the Checker state, a vector of outputs from both processors is compared and if they are different, the FSM changes its state to Error. Otherwise, if the outputs are the same, the FSM changes its state to Resume state in order to resume each processor execution. In the case of the FSM being in the Error state, the xLockstep stays in that state until both processors signalize that the error was corrected and the system is ready to recover from that state.

C. Synchro

Due to the difference in clock domains and architectures between the soft-core and hard-core processors, the program execution between them is asynchronous, demanding for the synchronization of both processors. For this purpose, it was created the Synchro module, which is used in two different scenarios. First, to synchronize the processors when a checkpoint is achieved, and second, to simultaneously return the code execution after the verification mechanisms of the lockstep have actuated. In both system operation scenarios, the xLockstep has to wait for both processors to indicate that they are ready to synchronize. This is achieved when: (1) the program reaches the checkpoint, and (2) both processors are ready to resume the execution. Therefore, to achieve those functionalities, the Synchro module implements a FSM with three states: Idle, Ready, and Sync. In the Ready state, the Synchro module expects both processors to enable the ***b_ready_to_sync*** bit, and afterwards, the Synchro gives feedback to both cores and enables the ***b_ready*** bit. Then, the state of the FSM changes to the state Sync. At this moment, the Synchro module is waiting for the synchronization's acknowledgment from each processor, which consists in disabling ***b_ready_to_sync*** bit. As a result, the processors' synchronization ends and both cores are synchronized.

D. Checker

For implementing the lockstep mechanism, both processors outputs must be compared. For that purpose, each core sends its output vector to the Checker module in order to perform their verification. The received outputs are stored in two different memory regions (one for each processor) by the Checker using a last in first out (LIFO) approach. Because both parts are involved in the data transfer process (processors and checker), both of them need to know the state of each other. For that, the Checker uses a control bit, ***b_Tx***, to coordinate the data transfer, which works in the following way: (1) when the Checker is available to receive and store an output, it puts its ***b_Tx*** bit to 0, signalizing the processor that it is available to perform the transaction. Next, it waits for the processor to signalize its availability to initialize a data transfer. After the data transfer, the Checker module clears the ***b_Tx*** bit and it is ready for another transaction. (2) after the data is received from both processors, at a given checkpoint, the Checker performs the comparison of the entire LIFO contents, checking for data

83C0001C	ARM_STATUS_REG	8000001C	RISCV_STATUS_REG
83C00018	UNUSED	80000018	UNUSED
83C00014	UNUSED	80000014	UNUSED
83C00010	UNUSED	80000010	UNUSED
83C0000C	UNUSED	8000000C	UNUSED
83C00008	ARM_TIMEOUT_REG	80000008	RISCV_TIMEOUT_REG
83C00004	ARM_CONTROL_REG	80000004	RISCV_CONTROL_REG
83C00000	ARM_DATA_REG	80000000	RISCV_DATA_REG

Fig. 3. The xLockstep peripheral memory address space.

integrity errors. There are two possible error cases that can be detected and signalized by the Checker to both processors. The first case occurs when an element from LIFO 1 is different from the respective element from LIFO 2. The second case results when the number of written outputs in both LIFO memories is different. The Checker LIFOs work as a circular buffer with limited size. Therefore, if one processors' output vector size cannot be accommodated by its respective LIFO, the Checker signalizes to the processor a busy state. This way, the Checker module is unaware of the data size and content, being the main concern only its storage and comparison. While the data is being processed, the processor waits for the Checker confirmation for the data processing in order to allow new data to be transferred (for the next checkpoint or for repeating the previous one).

E. xLockstep Framework

The xLockstep framework (currently under development) aims to be a tool that will help programmers to easily configure and use the Lock-V architecture, as well as to provide an API that is used to interact with the xLockstep accelerator. The framework will later support other features such as code inspection an code injection after the final application is done. The xLockstep API is composed by four functions: **`initXLockstep()`**, **`sync()`**, **`checker()`**, and **`errorFixed()`**, which are used to interact with the xLockstep. The **`initXLockstep()`** function is responsible to setup and initialize the xLockstep, as well as all the memory address space registers for each processor (Fig. 3). This function also sets the timeout value for the next checkpoint. The **`sync()`** function, is used for processors' synchronization. If the synchronization is not possible, an error is returned. The **`checker()`** function is responsible to handle the Checker functionalities, returning an error if both processor outputs, reported by the Checker module, are different. When an error occurs, the programmer should define the desired behavior, according to the application needs. After that, the **`errorFixed()`** function is called. This function signalizes the xLockstep accelerator that the error was processed and that the system is already in a normal state, reached after the error recovery.

TABLE I
CHECKER FUNCTIONAL TESTS.

V_A / V_B Size	$N_B = 0$	$N_B = 1$	$N_B = 2$	$N_B = 3$	$N_B = 4$	$N_B = 5$	V_{A_0}, V_{B_0}	V_{A_1}, V_{B_1}	V_{A_2}, V_{B_2}	V_{A_3}, V_{B_3}	V_{A_4}, V_{B_4}	State
$N_A = 0$	✗	✗	✗	✗	✗	✗	≠	=	=	=	=	✗
$N_A = 1$	✗	✓	✗	✗	✗	✗	=	≠	=	=	=	✗
$N_A = 2$	✗	✗	✓	✗	✗	✗	=	=	≠	=	=	✗
$N_A = 3$	✗	✗	✗	✓	✗	✗	=	=	=	≠	=	✗
$N_A = 4$	✗	✗	✗	✗	✓	✗	=	=	=	=	≠	✗
$N_A = 5$	✗	✗	✗	✗	✗	✓	=	=	=	=	=	✓

✗ Error; ✓Success; ≠ Element V_{A_n} different from V_{B_n}; = Element V_{A_n} equal to V_{B_n}.

IV. EVALUATION

The Lock-V architecture, and its main components, was deployed on a Zynq-7000 SoC, featuring a dual-core Arm Cortex-A9 and FPGA fabric used to host the RISC-V soft-core processor. In this implementation, the Arm Cortex-A9 is running at the frequency of 666 MHz and the lowRISC at the frequency of 25 MHz. These different clock domains will allow to check all the Synchro functionalities. In order to test the Lock-V and its main components, we have created three software modules: (1) a simple application to run on both processor architectures; (2) a test for the Synchro module, which prevents a checkpoint from being reached; and (3) a module for testing the Checker, that changes the output vectors for one of the cores. These software modules were inserted in the application with the purpose of testing the behavior of the xLockstep accelerator in the two fault detection situations:

- **A processor never achieves the checkpoint:** this can occur when a fault originates a bit-flip in the code memory, and the execution fails to match its original purpose. In such case, an error occurs due to the code execution never meeting the desired checkpoint or due a checkpoint timeout;
- **The processors output vectors are different:** the fault is detected when the V_{A_n} element is different from the V_{B_n} element, or when the number of expected elements in the vector V_A is different from the vector V_B.

A. Checker Module Functional Tests

Table I depicts a summary of the tests performed to the Checked module, where the size of the output vectors, as well as their content, are tested and compared. The possible input combinations for the Checker module depend on the output vector' size defined by the programmer. For the purpose of our tests, the vector size was set to five (one more element than the maximum storage size supported by each Checker's LIFO, which was set to four). The left side of Table I depicts the thirty-six possible combinations for testing the vector output's size. When V_A has a different size from V_B, the Checker module outputs an error. Besides the vector size, the Checker module also verifies the content of each vector's element. The right side of Table I depicts all the combinations that we have tested for testing the values of both vectors' elements, where two vectors of 5 elements were compared. The symbol ≠ is used when one element V_{A_n} does not match the content of respective element V_{B_n}. When the content of both vectors is different the Checker module outputs an error.

B. Synchro Module Functional Tests

Table II shows all combinations for the Synchro input signals that were tested, which can be summarized as follows:

- Both processors reach the checkpoint before the timeout;
- Only one of the checkpoints is reached before the timeout;
- None of the checkpoints is reached before the timeout.

Whenever the execution time between checkpoints is higher than the timeout, the Synchro outputs an error to the xLockstep. On the other hand, when both checkpoints are reached within the timeout value, the Synchro signalizes that both processes reached the checkpoint and the synchronization operation was executed with success.

C. PL Resources Utilization

Table III shows the hardware resources needed, after implementation, for the lowRISC soft-core and all the xLockstep modules. The results are expressed in terms of look-up tables (LUTs) and flip-flops (FFs). The lowRISC module is the most costly in terms of hardware needed, representing around 98% (34138 out of 34579) of LUTs and nearly 96% (16324 out of 16996) of FFs. This is due to the deployment of a soft-core RISC-V processor, rather than a hard-core implementation, which is one of the trade-offs of our solution. The solution provides flexibility and the possibility to customize the RISC-V architecture, but it comes with the cost of FPGA resources. Regarding the resources needed by the xLockstep accelerator (441 LUTs and 672 FFs), it is possible to conclude

TABLE II
SYNCHRO FUNCTIONAL TESTS.

Checkpoint Arm	Checkpoint RISC-V	State
*Y	**Y	✓
**Y	*Y	✓
Y	N	*timeout* ✗
N	Y	*timeout* ✗
N	N	✖

N Checkpoint Not Reached; **Y** Checkpoint Reached;
* Arrives First; ** Arrives in Second; ✖ Don't Care.

TABLE III
POST-IMPLEMENTATION RESULTS OBTAINED FROM VIVADO 2016.2

	HW module	*LUT*	*FF*
	lowRISC	34138	16324
xLockstep	*Axi_RISCV_Slave*	135	267
	Axi_ARM_Slave	122	269
	TopxLockstep	25	40
	Checker	148	90
	Synchro	6	3
	Synchro_to_Resume	5	3
	lowRISC + xLockstep	34579 (65%)	16996 (16%)

that the xLockstep has a lightweight implementation, and if both processors were available in the SoC in a hard-core implementation, the solution could resort an FPGA with less resources. Because the xLockstep follows a loosely-coupled approach, it is a good candidate to be used in other solutions, both in terms of hardware or processor architectures.

V. CONCLUSIONS AND FUTURE WORK

This paper presents a heterogeneous and fault-tolerance architecture, Lock-V, that explores a DCLS technique applied to different processor architectures. The proposed accelerator, the xLockstep, was deployed in a loosely-coupled fashion and connects a hard-core Arm Cortex-A9 and a soft-core RISC-V lowRISC, providing the lockstep capabilities to both processors at a very reduced hardware cost.

Hereafter, in order to keep improving the functionalities of the xLockstep accelerator, new features will be added: (1) a mechanism dedicated to perform fault-injection, which will be helpful in simulating the wrong device's operation due to bit-flipping; (2) the framework optimization in order to provide code injection capabilities. This feature will allow the code application to be automatically analyzed by the framework, which will choose the best places to deploy the lockstep checkpoints, and later create the data to configure the xLockstep accelerator; and (3) the exploration of the RISC-V open-source ISA, which will allow the creation of ISA instructions customized to the xLockstep peripheral. This will allow a complete deployment of the lockstep mechanism on the RISC-V architecture in a tightly-coupled fashion, which will help in understanding the advantages between both approaches, i.e., loosely-coupled and tightly-coupled.

VI. ACKNOWLEDGMENTS

This work has been supported by national funds through FCT -*Fundação para a Ciência e Tecnologia* within the Project Scope: UID/CEC/00319/2019.

REFERENCES

[1] R. C. Baumann, "Radiation-induced soft errors in advanced semiconductor technologies," *IEEE Transactions on Device and Materials Reliability*, vol. 5, no. 3, pp. 305–316, Sep. 2005.

[2] I. Hwang, S. Kim, Y. Kim, and C. E. Seah, "A Survey of Fault Detection, Isolation, and Reconfiguration Methods," *IEEE Transactions on Control Systems Technology*, vol. 18, no. 3, pp. 636–653, May 2010.

[3] F. Abate, L. Sterpone, C. A. Lisboa, L. Carro, and M. Violante, "New Techniques for Improving the Performance of the Lockstep Architecture for SEEs Mitigation in FPGA Embedded Processors," *IEEE Transactions on Nuclear Science*, vol. 56, no. 4, pp. 1992–2000, Aug. 2009.

[4] Á. B. de Oliveira, G. S. Rodrigues, and F. L. Kastensmidt, "Analyzing Lockstep Dual-Core ARM Cortex-A9 Soft Error Mitigation in freeRTOS Applications," in *Proceedings of the 30th Symposium on Integrated Circuits and Systems Design Chip on the Sands - SBCCI '17*. Fortaleza, Ceará, Brazil: ACM Press, 2017, pp. 84–89.

[5] E. Ozer, B. Venu, X. Iturbe, S. Das, S. Lyberis, J. Biggs, P. Harrod, and J. Penton, "Error Correlation Prediction in Lockstep Processors for Safety-Critical Systems," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Fukuoka: IEEE, Oct. 2018, pp. 737–748.

[6] J. Han, Y. Kwon, Y. C. P. Cho, and H.-J. 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 (A-SSCC)*. Seoul: IEEE, Nov. 2017, pp. 313–316.

[7] J. S. Klecka, W. F. Bruckert, and R. L. Jardine, "Error self-checking and recovery using lock-step processor pair architecture," May 21 2002, US Patent 6393582.

[8] A. B. de Oliveira, G. S. Rodrigues, F. L. Kastensmidt, N. Added, E. L. A. Macchione, V. A. P. Aguiar, N. H. Medina, and M. A. G. Silveira, "Lockstep Dual-Core ARM A9: Implementation and Resilience Analysis Under Heavy Ion-Induced Soft Errors," *IEEE Transactions on Nuclear Science*, vol. 65, no. 8, pp. 1783–1790, Aug. 2018.

[9] A. Hanafi, M. Karim, and A. E. Hammami, "Dual-Lockstep Microblaze-Based Embedded System for Error Detection and Recovery with Reconfiguration Technique," in *2015 Third World Conference on Complex Systems (WCCS)*. Marrakech: IEEE, Nov. 2015, pp. 1–6.

[10] H.-M. Pham, S. Pillement, and S. J. Piestrak, "Low-Overhead Fault-Tolerance Technique for a Dynamically Reconfigurable Softcore Processor," *IEEE Transactions on Computers*, vol. 62, no. 6, pp. 1179–1192, Jun. 2013.

[11] R. D. Kral, J. S. M. Chong, and A. L. Schreiber, "Implementation of a loosely-coupled lockstep approach in the xilinx zynq-7000 all programmable soc for high consequence applications." Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), Tech. Rep., 2017.

[12] P. Garcia, T. Gomes, F. Salgado, J. Cabral, P. Cardoso, M. Ekpanyapong, and A. Tavares, "A Fault Tolerant Design Methodology for a FPGA-Based Softcore Processor," *IFAC Proceedings Volumes*, vol. 45, no. 4, pp. 145–150, 2012.

[13] M. Pignol, "DMT and DT2: Two Fault-Tolerant Architectures developed by CNES for COTs-based Spacecraft Supercomputers," in *12th IEEE International On-Line Testing Symposium (IOLTS'06)*. Como, Italy: IEEE, 2006, pp. 203–212.

[14] S. Pinto, A. Tavares, and S. Montenegro, "Space and time partitioning with hardware support for space applications," *Data Systems In Aerospace, European Space Agency, ESA SP 736*, 2016.

[15] M. Berg and C. Michael, "FPGA Mitigation Strategies for Critical Applications, support of NASA/GSFC," Sep. 2018.

[16] T. Gomes, F. Salgado, A. Tavares, and J. Cabral, "CUTE Mote, A Customizable and Trustable End-Device for the Internet of Things," *IEEE Sensors Journal*, vol. 17, no. 20, pp. 6816–6824, Oct. 2017.

[17] F. Salgado, T. Gomes, J. Cabral, J. Monteiro, and A. Tavares, "DBTOR: A Dynamic Binary Translation Architecture for Modern Embedded Systems," in *2019 IEEE International Conference on Industrial Technology (ICIT)*, Feb 2019, pp. 1755–1760.

[18] J.-C. Laprie, "Dependable Computing and Fault Tolerance: Concepts and Terminology," in *Twenty-Fifth International Symposium on Fault-Tolerant Computing, 1995, ' Highlights from Twenty-Five Years'*. Pasadena, CA: IEEE, 1995, p. 2.

[19] Z. Gao, C. Cecati, and S. X. Ding, "A Survey of Fault Diagnosis and Fault-Tolerant Techniques—Part I: Fault Diagnosis With Model-Based and Signal-Based Approaches," *IEEE Transactions on Industrial Electronics*, vol. 62, no. 6, pp. 3757–3767, Jun. 2015.

[20] D. Patterson and A. Waterman, *The RISC-V Reader: An Open Architecture Atlas*, 1st ed. Strawberry Canyon, Nov. 2017.

[21] J. W. Jonathan Bachrach, Krste Asanović, "Chisel 3.0 Tutorial," EECS Department, UC Berkeley, Tech. Rep., 2017.

[22] M. Nöltner-Augustin, "RISC-V — Architecture and Interfaces The RocketChip," *COMPUTER ENGINEERING*, p. 6, 2016.