Peña-Fernández, M., Serrano-Cases, A., Lindoso, A., García-Valderas, M., Entrena, L., Martínez-Álvarez, A. & Cuenca-Asensi, S. (2019). Dual-Core Lockstep enhanced with redundant multithread support and control-flow error detection. *Microelectronics Reliability*, vol. 100-101, 113447.

DOI: 10.1016/j.microrel.2019.113447

Dual-Core Lockstep Enhanced with Redundant MultiThread Support and Control-Flow Error Detection

M. Peña-Fernández[a], A. Serrano-Cases[b], A. Lindoso[c*], M. García-Valderas[c], L. Entrena[c], A. Martínez-Álvarez[b], S. Cuenca-Asensi[b]

[a] Arquimea *Ingenieria SLU., Leganes, Madrid, Spain*
[b] Departament of Computer Technology, *University of Alicante, Spain*
[c] *Electronic Technology Department, University Carlos III of Madrid, Spain*

Abstract

This work presents a new Dual-Core LockStep approach to enhance fault tolerance in microprocessors. The proposed technique is based on the combination of software-based data checking and trace-based control-flow checking through an external hardware module. The hardware module is connected to the trace interface and is able to observe the execution of all the processors in the architecture. The proposed approach has been implemented for a dual core commercial processor. Experimental results demonstrate that the proposed technique has a high error detection capability with up to 99.63% error coverage.

1. Introduction

Microprocessors are the backbone of digital electronic systems. The progress of manufacturing technologies and the reduction of the transistor feature size has made microprocessors cheap and suitable for a huge variety of applications. At the same time, the susceptibility to soft errors, mainly caused by ionizing particles, has grown to become a concern [1] in an increasing number of cases. For high reliability applications, microprocessors are required to be fault-tolerant, i.e., to be able to continue operation in the event of failure. In the past, fault-tolerant microprocessors were required for systems working in harsh environments, such as aerospace, but today they are increasingly demanded even at ground level.

Techniques to protect microprocessors against soft errors can be classified into software and hardware techniques. Software techniques are very flexible, but they are inherently limited [2]. Hardware techniques that require modifying the microprocessor are often not feasible because the design and manufacturing of a microprocessor is a costly process that can only be afforded for high volume production. In contrast, Dual Modular Redundancy (DMR) is an attractive solution with high error detection capabilities. As microprocessors are today rather cheap, duplication is not expensive. In fact, multicore devices have become very common even for low-end devices, and state-of-the art MCU are starting to introduce safety features, which are becoming more relevant to automatic control, such as in the autonomous automotive industry and aerospace[3].

Dual-Core Lockstep (DCLS) [4-7] is a DMR fault-tolerant technique that can exploit the availability of multicore devices. It consists in two processors simultaneously running the same set of operations, syncing their output each cycle and triggering a recovery routine in case of discrepancy. This architecture is described in the white paper ISO 26262, where the DCLS processors are also referred as "ASIL-D MCUs". Despite the safety features introduced, ASIL-D MCUs do not eliminate the need to implement other safety measures at software and system level. Several MCU's and processors have successfully implemented this feature, for instance, Freescale MPC5643L [8], PPC405 Lockstep System on ML310 and the ARM Cortex-M33, Cortex-R4, Cortex-R5 and Cortex-R7 [3, 9]. The ARM Cortex-R5, has been integrated in several platforms, such as TI Hercules TMS570 microcontrollers and in the Xilinx UltraScale MPSoCs. However, it has been reported that the recovery process presents high

---

[*] Corresponding author. alindoso@ing.uc3m.es
Tel: +34 (091) 6245964 ; Fax: +34 (091) 6249430

overheads, around x1000 compared to a Triple Core Lock-Step [9].

Software DLCS divides the processing into steps, ranging from individual instructions to a set of functions. After each step, the results of the computations produced by each processor are compared. If they do not match, a rollback mechanism is triggered to restore the system back to a consistent state. The comparison of the computation results provided by the two processors is the key aspect of DCLS. Generally, only output data are checked for errors [5-6]. However, control-flow errors may cause one of the processors to lose synchronization and eventually hang or get lost. Control-flow errors are not easy to detect as they may not have an immediate observable effect in the computed data. Moreover, it is common in dual cores that one of the processors acts as a master and the other as a slave. In such a case, the hang of the master can lead to the crash of the entire system. A possible solution to this problem is to use timeout watchdog monitors to detect unusually long computation times [6]. However, this approach is weak and results in a high error detection latency. Moreover, control-flow errors may produce latent effects that may remain in the system after it is restored even though the output data are correct.

In this work we propose an enhanced DCLS approach that uses two complementary mechanisms: observation of information provided by the trace subsystem to monitor the execution control flow and a multithread software-based scheme to detect and recover from data inconsistencies.

Most microprocessors today provide a trace subsystem for debugging purposes which is able to report the microprocessor control flow in a seamless and non-intrusive manner without affecting the execution. Under normal operation, the trace subsystem can be reused to monitor the control flow of the processor [10-11]. Errors that affect the control flow in any of the processors can be detected by on-line decoding the corresponding program traces and checking the obtained information [12].

Modern processor architectures and Operating Systems (OS) commonly support the parallel execution of different threads and processes. Those capabilities have been exploited in different approaches by executing several replicas of the code on the same processor (SMT-Simultaneous Multithread) [13], on separate cores (CMP-Chip Level Multiprocesor) [14] or using a mixture of them [15]. All those approaches rely on complex software stacks that include, in addition to the OS, different support libraries in order to reduce the development time and ease the management of the replicated threads/processes [16-18]. However, every software layer added introduces new vulnerabilities that degrade the overall reliability of the applications.

In our approach, a CMP scheme has been adopted for bare metal applications (without OS) which renders a reduced number of race conditions and lower control overhead compared to traditional solutions. The redundant threads execute on different cores and, eventually, check their outputs. In case of discrepancies, threads are forced to re-execute the critical regions as recovery mechanism. The proposed approach can be considered a relaxed lockstep execution where protection may be applied with different granularity, from the whole application to just some critical regions of the code. Therefore, a suitable trade off can be established between the number checkpoints and the time overhead produced in case of recovery.

The proposed approach has been implemented and evaluated on a dual-core ARM Cortex-A9 [19]. The microprocessor is a hard core in a Zynq FPGA [20]. The proposed trace monitor has been implemented in the programmable logic. The proposed technique has been tested with an injection campaign of 871837 faults that resulted in 43769 errors. Experimental results show that the proposed approach shows excellent error detection capabilities with a percentage of detected errors of up to 99.63%.

This paper is organized as follows: section 2 describes the proposed lockstep approach, section 3 presents the experimental results and finally section 4 summarize the conclusion of this work.

2. Proposed Lockstep approach

2.1. Architecture

Contrarily to other approaches that propose new hardware structures to extend the architecture of CMP processors, our solution is intended to be directly applied to modern multicore processors. The architecture uses redundant multithread support for data error detection combined with trace monitoring for control-flow error detection.

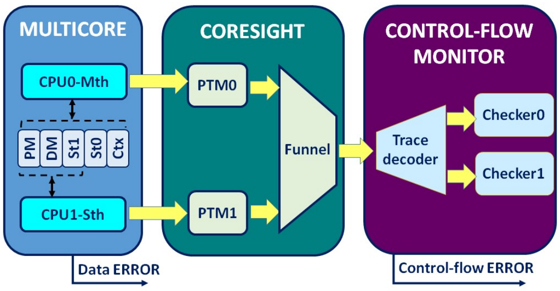

Fig. 1. Dual-Core LockStep Architecture

Fig. 1 shows the architecture of the proposed DCLS approach. It is divided in three main blocks: multicore microprocessor (Multicore), ARM Trace subsystem (Coresight) and Control Flow Monitor. The Control-Flow Monitor is a small piece of hardware that can be embedded in a FPGA. Data error detection is implemented in the Multicore block and is described in subsection 2.2. The remaining blocks (Coresight and Control-flow monitor) are described in subsection 2.3.

2.2. Data error detection

A pure software mechanism has been developed for data error detecting and recovering. It is based on modular redundancy strategy and relies on the parallel execution of redundant threads on separate cores. This way it could be easily adapted to Dual or Triple Modular Redundancy depending on the number of available cores [21].

Similar to traditional threads supported by OS like Linux (POSIX Threads API) or specialized libraries (OpenMP API), in our model the two threads, named master thread (MTh) and shadow thread (STh) respectively, share instructions (see PM on Fig1) and data memories (see DM on Fig 1). However, as a key difference, each thread has its own stack (i.e. the memory region for storing temporary and automatic variables and function output variables) which allows to hold the replicas of the data. In addition, a third stack memory region is reserved to store the Context (Fig 1: Ctx). Since MTh is in charge of checking the Context and output variables of every region under consideration, this thread has access to the three stacks.

The application is divided into several Critical Regions of code. Each one is characterized by the input or Context variables (i.e. global and local variables) and output variables. For the sake of completeness, in this work the Context contains all the information needed to define the status of the execution, i.e. the input variables to the region. As a limitation of the current implementation of our approach, we are not able to save the status of the processor's cache, and therefore we assume it is disabled. As a side effect of this choice, the application should be written in such a way that input variables are not modified during the execution of the code within a region, so that in case of recovery, the integrity of those variables is preserved.

Critical regions are delimited by annotation primitives in C or C++ and follow the concept of Sphere of Replication (SoR) established in [22]. To this end, the code section is instrumented to check the correctness of the variables and to synchronize the execution using barriers and mutex (mutual exclusion). The SoR defines the code regions where thread replication and parallel execution will take place. When the instruction flow reaches the SoR (critical region), a Context Check is performed by the MTh. If there are no discrepancies, the Context variables are saved to the Context Stack by the MTh. Every time the instruction flow goes out the SoR boundaries, consistency checks are automatically carried out by the MTh on the data stored on both stacks. A recovery procedure, i.e. the re-execution of the critical region, is executed if any discrepancy is found.

Depending on the boundaries and the situation of the critical regions within the code, the protection can be applied at different levels of granularity to get the best trade-off between performance overhead and latency to recover from a fault.

Figure 2 shows an example of annotated pseudocode for the Matrix Multiplication algorithm. As can be seen, `SYNC` and `CHECK` annotations enclose the inner-most loop and define a region. Also, this mechanism provides synchronization for each thread and automatic check over Context variables `i,j` and output variable `acc` respectively. The localization of the region defines a lockstep execution with NxN checkpoints (steps) and the recovery latency is equivalent to the execution time of the inner loop (lines 9-13) plus an additional assignment (line 9). A finer granularity could be obtained by including just the inner loop body in the critical region. As a result, the recovery latency decreases to just the execution of a line of code, at the cost of increasing the number of checkpoints up to NxNxN. Note that by removing the 8th and 12th lines (which are responsible for the automatic code instrumentation) we obtain the original unprotected code. It is remarkable that both `SYNC` and `CHECK` functions can be compiled along with the software application as they are written in C. Thus, to protect an application, the user is just required to manually introduce both functions enclosing the critical sections in the source code. In the future, a tool to insert automatically the calls for these functions is planned to be developed to enhance the scalability of this solution.

Algorithm 1:

```
NT= 2        //Number of threads
A = Matrix[N][N]
B = Matrix[N][N]
C = Matrix[N][N]
procedure MxM
  for i = 0 to N-1 do
    for j = 0 to N-1 do
      //Start of region
      SYNC(i,j;NT)
      acc= 0
      for k=0 to N-1 do
        acc += A[i][k] · B[k][j]
      CHECK(acc;NT)
      //End of region
      C[i][j]=acc
```

Fig. 2. Redundant Threaded Matrix Multiplication

In addition to the code annotation, other software tweaks were implemented to endow the dual core system with the ability of running redundant threads on bare metal. In first place, it was necessary to modify the Board Support Package (BSP) in order to initialize the platform and start up all cores presented in the architecture. It is common that, in bare metal environments, the BSP provides the minimal files to boot up the platform. However, BSPs only cover a little subset of the most common ways to boot a system. The default boot sequence is controlled by CPU0 while the other CPU gets into an infinite busy-waiting loop. This default initialization code was changed to allow a boot in SPMD mode (Single Program Multiple Data). In second place, the memory map and the associated linker scripts were modified to support separate stack sections for each core. Finally, a spin-lock mechanism was added to allow the synchronization of the cores.

2.3. Control-flow error detection

The control flow of both cores is observed through an external hardware IP (Control-flow Monitor), which is connected to the trace interface. The proposed approach is a multicore extension of the technique presented in [12].

The dual-core ARM Cortex-A9 presented in the selected device contains a trace subsystem based in CoreSight modules [23]. Of the available CoreSight modules in the selected device, our system uses only the PTM (Program Trace Macrocell) [24]. PTM is a trace source CoreSight subtype. A PTM cell is associated to a single core of the architecture. Thus, for this work two PTM instances, PTM0 and PTM1, are used, which are linked respectively to core 0 and core 1 (Fig 1). Both trace sources are multiplexed and sent through the trace interface. The external IP decodes the trace information and checks the correctness of the execution-flow by controlling the PC addresses of the executed instructions in both cores.

Two techniques are used to detect incorrect execution: confidence range checking and address watchdog checking. The former consists in configuring the external IP to treat some instruction-memory address ranges as valid. While the instruction addresses of a core are within its own confidence ranges, no error is assumed. The latter is also related with the core instruction address, but in this case only one specific address is configured to be checked periodically, namely the first instruction of each step. It the required instruction address is not received within a configurable time, a timeout is asserted. In the case an unexpected instruction or timeout is detected in any of the two cores, an error signal is triggered. The IP works on-line with very small latency. No additional information is required or stored before execution takes place, and the required configuration register values can be determined at compilation time. The external IP can be configured from software as an AXI peripheral.

To ease the use of the external IP, all user-defined application functions have been targeted to a specific region of memory defined by the programmer in the linker script, so most of the code is inside the same confidence interval. Some native or library functions that cannot be targeted to this region have also been protected using three more confidence intervals. The first instruction of the main loop of the code has been selected as the instruction address to be checked by the watchdog. As the application is hardened with DCLS, the external IP has been configured to check both CPUs with the very same parameters.

3. Experimental results

A fault injection campaign has been performed to test the proposed technique. Faults were injected only in one of the two cores in the selected architecture. The Mth core have been selected for injection as it is the most critical considering that it performs context and data checking.

Faults were injected in the register file adapting the technique presented in [12]. This technique generates bit-flips randomly in the register file. An external controller has been used in order to determine, classify and collect the observed and detected errors. In radiation environments, errors can also affect memories which are not covered by the utilized technique. Memories that are exposed to radiation are usually protected with redundancy techniques such as EDAC. In previous radiation campaigns [12], we have validated the fault injection approach with quite accurate error detection match between radiation and injection results even though we only injected faults in the register file.

The control-flow monitor is located in the FPGA and radiation can affect its behaviour. Xilinx SEM IP [25] can be used to protect the FPGA and the circuit it contains.

The experiments have been carried out on commercial ZYBO boards featuring a Xilinx Z7010 Zynq [20] as the device under test (DUT). Both ARM Cortex-A9 cores in the DUT are clocked at 650MHz frequency. At the beginning of the application, the external controller generates a random seed which is used to generate the injection parameters (time instant, register number and bit index) and the initialization values of program data. When the DUT has received the seed, the execution starts, and the injector as well. A fault is injected every five iterations of the application main loop. In the case no error appears in these five iterations, a silent error is assumed, and a new injection is produced. In the event of an error, the external controller registers the results and power cycles the DUT to start a new injection.

The results of the fault injection campaign are summarized in Table 1. Two experiments were accomplished with two versions of a matrix multiplication benchmark: unoptimized (-O0: column 2 of Table 1) and optimized with maximum effort (-O3, column 3 of Table 1). Both benchmarks use matrices of 32x32 32-bit integer elements.

For -O0 benchmark 591821 faults were injected resulting in 20011 (3.38%) errors. For -O3 benchmark 280016 faults were injected resulting in 23758 (8.48%) errors.

The error categories reported in Table 1 are:

- Det. Hang: The fault has produced a functional interrupt in the system, which has been detected by the external IP.
- Hang: The fault has produced a functional interrupt in the system, which has not been detected.
- Det. Only IP: The external IP has reported a control-flow error while no functional interrupt has been produced.
- Det. Data: The fault has produced a data error which has been detected by the software.
- SDC: Silent Data Corruption, the fault has produced a data error which has not been detected.
- Comm: Communications malfunction between the DUT and the external controller that makes impossible to classify the error.
- Total: Total number of errors

Table 1
Injection campaign results

Error type	-O0		-O3	
	# errors	% errors	# errors	% errors
Det. Hang	15,462	77.27%	15,879	66.84%
Hang	23	0.11%	64	0.27%
Det. Only. IP	75	0.37%	135	0.57%
Det. Data	4,338	21.68%	6,725	28.31%
SDC	50	0.25%	940	3.96%
Comm	63	0.31%	15	0.06%
Total	20,011	100.00%	23,758	100.00%

Table 1 shows the high error detection capability of the proposed approach, with 99.63% errors detected for the unoptimized version and 95.77% for the optimized one. For this metric we have not considered Comm errors, as it is not possible to categorize them.

Regarding control-flow errors, it is noticeable that most of them are related with system functional interrupts, which are mainly caused by exceptions or loss of lockstep synchronization. However, few of them (Det. Only IP) have not produced this effect. These can be associated with control-flow errors that do not produce a hang on the system, for example, an error that causes a branch to a wrong, but valid, code region or an error in a loop index causing an unexpectedly bigger execution time. Although these errors can be false positives, it is highly recommendable to consider them as real errors for preventive reasons. Code optimization produces a small reduction on control-flow error detection rates.

With respect to data errors, there is a small portion of SDCs. These errors are caused by faults that are injected after the software check. Note that fault injection is non-stop, so data may be corrupted at any time and therefore an error may appear at the final check of the test used to categorize the results. Nevertheless, such errors could be detected if the final check is also made in lockstep. Anyhow, SDC errors represent a very small portion, particularly when executing unoptimized code. This effect increases when optimization is introduced because the compiler changes the order in which some operations are made to achieve higher throughput. Also, as the optimized code gets shorter, the probability to inject an error after the software check gets higher. It is remarkable that optimization has been introduced on the very same code that produced the unoptimized version, meaning that no further effort has been done to enhance error detection for the optimized version, so there is room for improvement. Even so, the data detection penalty is restrained and could be affordable for some applications.

In relation to error rates, optimized code has higher susceptibility to errors: 8.48% of injected faults produced errors while in the unoptimized version only 3.38% of injected faults resulted in error. Furthermore, data has demonstrated to be more prone to errors in the optimized version as 32.27% of total errors were data errors contrasting with the 21.93% of the unoptimized code. These two effects are related to a much higher use of registers in the case of the optimized version. Considering these results, it is interesting to go deeper on how registers are related to errors, extracting the injector information when the error occurs. Fig. 3 presents a comparison of the register sensitivity distribution in both code versions.

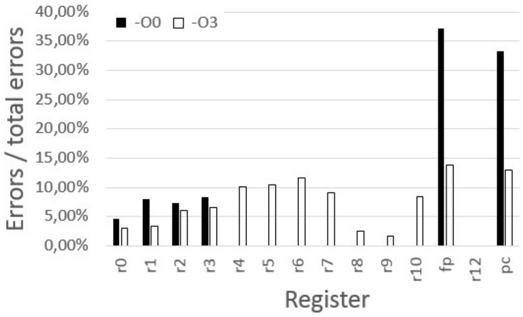

Fig. 3. Register sensitivity to errors

Results in Fig. 3 demonstrate the more extensive usage of registers in the case of optimized (-O3) code, as all core registers except r12 produce errors. In the case of unoptimized code, only four general purpose registers, r0 to r3, are used so faults injected on r4 to r10 have no impact on errors. Frame pointer (fp) and program counter (pc) are strongly related to the execution control-flow and have the highest error rates in both cases.

4. Conclusions

This work presents a Dual-Core Lockstep approach enhanced with redundant multithread support and control-flow error detection. Data error detection and recovering is based on the parallel execution of redundant threads on separate cores and a pure software technique that checks the correctness of the data and synchronizes the execution checks. Control-flow protection is accomplished by an external hardware IP that monitors the execution trace of the two cores in a non-intrusive way with small latency.

Experimental results demonstrate that control-flow errors are very likely, so that both data and control-flow checking are needed for effective error detection. The proposed approach achieves a high error detection rate (up to 99.63% error coverage) with low latency.

Acknowledgements

This work was supported in part by the Spanish Ministry of Economy and Competitiveness under projects ESP2015-68245-C4-1-P, ESP2015-68245-C4-3-P and by the Community of Madrid under grant IND2017/TIC-7776.

References

[1] R. C. Baumann, Radiation-induced soft errors in advanced semiconductor technologies, IEEE Trans. on Device and Materials Rel., vol. 5, no. pp. 305-316, 2005.

[2] J. R. Azambuja, S. Pagliarini, L. Rosa, and F. L. Kastensmidt, Exploring the limitations of software-only techniques in SEE detection coverage, Journal of Electronic Testing, no. 27, (2011), pp. 541–550.

[3] X. Iturbe, B. Venu and E. Ozer, "Soft error vulnerability assessment of the real-time safety-related ARM Cortex-R5 CPU," 2016 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), Storrs, CT, 2016, pp. 91-96.

[4] N. S. Bowen and D. K. Pradham, Processor and memory based checkpoint and rollback recovery, Computer, vol. 26, no. 2, pp. 22–31, Feb. 1993.

[5] A. B. de Oliveira et al., 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.

[6] F. Abate, L. Sterpone, and M. Violante, A new

mitigation approach for soft errors in embedded processors, IEEE Transactions on Nuclear Science, vol. 55, no. 4, pp. 2063–2069, Aug. 2008.

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

[8] BERNON-ENJALBERT, Valerie, et al. Safety Integrated Hardware Solutions to Support ASIL D Applications. 2013.

[9] X. Iturbe, B. Venu, E. Ozer and S. Das, "A Triple Core Lock-Step (TCLS) ARM® Cortex®-R5 Processor for Safety-Critical and Ultra-Reliable Applications," 2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop (DSN-W), Toulouse, 2016, pp. 246-249.

[10] M. Portela-García et al.. On the use of embedded debug features for permanent and transient fault resilience in microprocessors. Microprocessors and Microsystems, 36(5), pp. 334-343, 2012.

[11] L. Entrena, A. Lindoso, M. Portela-García, L. Parra, B. Du, M. Sonza Reorda, L. Sterpone, Fault-tolerance techniques for soft-core processors using the Trace Interface, In "FPGAs and Parallel Architectures for Aerospace Applications. Soft Errors and Fault-Tolerant Design", Springer, 2015.

[12] M. Peña-Fernandez, A. Lindoso, L. Entrena, M. Garcia-Valderas, S. Philippe, Y. Morilla, P. Martin-Holgado. PTM-based hybrid error-detection architecture for ARM microprocessors. Microelectronics Reliability, 88, pp. 925-930, 2018.

[13] T. N. Vijaykumar, I. Pomeranz, and K. Cheng, "Transient-fault recovery using simultaneous multithreading," in Proc. 29th Annu. Int. Symp. Comput. Archit., Anchorage, AK, USA, 2002, pp. 38–87.

[14] M. Gomaa, C. Scarbrough, T. N. Vijaykumar and I. Pomeranz, "Transient-fault recovery for chip multiprocessors," 30th Annual International Symposium on Computer Architecture, 2003. Proceedings., San Diego, CA, USA, 2003, pp. 98-109.

[15] K. Chen, G. v. der Bruggen, and J. Chen, "Reliability optimization on multi-core systems with multi-tasking and redundant multi-threading," IEEE Transactions on Computers, vol. 67, no. 4, pp. 484–497, April 2018.

[16] A. Shye, J. Blomstedt, T. Moseley, V. J. Reddi, and D. A. Connors, "Plr: A software approach to transient fault tolerance for multicore architectures," IEEE Transactions on Dependable and Secure Computing, vol. 6, no. 2, pp. 135–148, April 2009.

[17] H. Mushtaq, Z. Al-Ars, and K. Bertels, "Efficient software-based fault tolerance approach on multicore platforms," in Proc. Design, Autom. Test Eur. Conf. Exhibit. (DATE), Grenoble, France, 2013, pp. 921–926.

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

[19] ARM, Cortex-A9 MPCore Technical Reference Manual, 2011.

[20] "Zynq-7000 All Programmable SoC: Technical Reference Manual",Xilinx Inc., Technical Ref. Manual UG585, Sept. 2016.

[21] A. Serrano-Cases, F. Restrepo-Calle, S. Cuenca-Asensi1 and A. Martínez-Álvarez, "Softerror mitigation for multi-core processors based on thread replication" Proceedings of 20th IEEE Latin American Test Symposium, Chile, March 2019

[22] S. K. Reinhardt and S. S. Mukherjee, "Transient fault detection via simultaneous multithreading," Proceedings of 27th International Symposium on Computer Architecture (IEEE Cat. No.RS00201), Vancouver, BC, Canada, 2000, pp. 25-36.

[23] "CoreSight Components. Technical Reference Manual", ARM Ltd., DDI0314H, 2009.

[24] "CoreSight Program Flow Trace. Architecture Specification", ARM Ltd., IHI 0035B, 2011.

[25] "Soft Error Mitigation Controller v4.1", Product Guide, Xilinx Inc., PG036, Nov. 2014