HAL open science

Design with low complexity fine-grained Dual Core Lock-Step (DCLS) RISC-V processors

Pegdwende Romaric Nikiema, Angeliki Kritikakou, Marcello Traiola, Olivier Sentieys, Olivier Sentieys

To cite this version:

Pegdwende Romaric Nikiema, Angeliki Kritikakou, Marcello Traiola, Olivier Sentieys, Olivier Sentieys. Design with low complexity fine-grained Dual Core Lock-Step (DCLS) RISC-V processors. DSN 2023 - 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks, Jun 2023, Porto, Portugal. pp.224-229, 10.1109/DSN-S58398.2023.00062 . hal-04397673

HAL Id: hal-04397673

https://hal.science/hal-04397673v1

Submitted on 16 Jan 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Design with low complexity fine-grained Dual Core Lock-Step (DCLS) RISC-V processors

Pegdwende Romaric Nikiema, Angeliki Kritikakou, Marcello Traiola, Olivier Sentieys
Inria, Univ Rennes, CNRS, IRISA
pegdwende.nikiema@inria.fr, angeliki.kritikakou@irisa.fr, marcello.traiola@inria.fr, olivier.sentieys@inria.fr

***Abstract*—Embedded systems in critical domains require both hard real-time and reliable execution. Real-time execution requires bounds in the worst-case execution time, while reliable execution is under threat, as systems are becoming more and more sensitive to transient faults. Thus, systems should be enhanced with fault-tolerant mechanisms with bounded error detection and correction overhead. Such mechanisms are typically based on redundancy at different granularity levels. Coarse-grained granularity has low comparison overhead, but may jeopardize timing guarantees. Fine-grained granularity immediately detects and corrects the error, but its implementation has increased design complexity. To mitigate this design complexity, we leverage high-level specification languages to design intrusive fine-grained lockstep processors based on the use of shadow registers and rollback, with bounded error detection and correction time, being appropriate for critical systems.**

***Index Terms*—Transient faults, Reliability, Real-time, Fault-tolerance, RISC-V, High level specification**

I. Introduction

Embedded systems in critical domains, such as automotive, aviation, space domains, often require both hard real-time and reliable execution. Real-time execution is provided through timing guarantees that ensure that the worst-case execution time of the application can be bounded and it does not exceed a given latency requirement [1]. However, reliable execution is under threat due to the increased fault susceptibility of modern electronic systems. Due to the reduced transistor sizes and lower supply voltages of modern technologies [2], systems are becoming more and more sensitive to environmental sources [3], such as ionization, radiation, and high-energy electromagnetic interference, leading to temporary reliability violations, called transient faults. With the technology reduction, transient faults become dominant [3], occurring even under normal operation conditions, which was not the case with technology used a decade ago [4]. As systems become more and more prone to transient faults during execution [5], they should be enhanced with fault-tolerant mechanisms in order to provide reliable execution.

Fault-tolerant mechanisms are typically based on redundancy. To deal with faults occurring in the processors, the same set of operations with same inputs is executed on different processors. Such approach is promising, since the probability of having the same fault concurrently occurring on all processors is very small [6]. To detect/correct a fault, the outputs of redundant operations must be compared. This comparison can be performed at different granularity levels, from coarse-grained granularity, defined at the memory and I/O interface taking place at the end of application execution, to fine-grained granularity, occurring at the instruction level during the application execution. Although coarse-grained granularity has low overhead in terms of comparison, error detection/correction may occur long after the fault affects the system, thus jeopardizing timing guarantees in hard real-time systems [7]. On the contrary, fine-grained granularity provides immediate error detection/correction. Non-intrusive fine-grained approaches have typically high comparison overhead, since the outcome of all instructions (i.e., values and addresses) has to cross the shared communication network to reach memory and I/O in order to be compared [7]. Therefore, intrusive approaches are required, which, however, have increased design complexity due to large processor modifications required and the complex validation of the circuitry to perform the across-core comparisons every cycle [8].

To mitigate such complexity increase during the design process, we leverage high-level specification languages to design intrusive fine-grained lockstep processors, with bounded error detection and correction time, thus being appropriate for critical systems. By using high-level specification languages, such as C and C++, the processor design becomes less complex, as the processor model can easily be modified, expanded and verified, compared to HDL implementations [9]. With the help of High-Level Synthesis (HLS) tools, such high-level processor descriptions can be interpreted to create digital hardware which implements the same functionality. More precisely, we use an on open-source HLS implementation of a 32-bit RISC-V processor [9] as case study and we propose two intrusive fine-grained lockstep approaches, with different area and upper bound detection and correction time, , i.e., Partial Shadow Register with Rollback (PSRR) and Full Shadow Register (FSR). While dual Modular Redundancy (DMR) enables only fault detection, Triple Modular Redundancy (TMR) allows achieving error correction but also entails significant area and power overhead. Thus, we extend a DMR approach with fine-grained correction mechanisms – based on register shadowing and rollback mechanisms – in order to perform error correction, with reduced area overhead w.r.t. TMR.

II. Related work

Existing redundancy approaches applied for processors can be categorized as *non-intrusive* and *intrusive*.

Non-intrusive approaches do not modify the processor architecture, and are typically used when the internal architecture details are hidden or difficult to modify, e.g., Commercial Off-The-Shelf (COTS) processors. Depending on the type of cores, the approaches can be homogeneous or heterogeneous. For instance, a homogeneous approach uses two MicroBlaze soft cores to implement Dual Core LockStep (DCLS). When a core mismatch is detected, a roll-forward correction is applied that excludes temporary the faulty core, which is repaired through reconfiguration. Meanwhile, the correct processor continues execution and its state is copied as soon as faulty core is reconfigured. A heterogeneous DCLS approach uses ARM A9 as hard core and RISC-V as soft core [10]. Lockstep execution is achieved by inserting checkpoints in the application, where a synchronisation module is activated to check for mismatch between the status of the cores and apply roll-back. Heterogeneous approaches may have reduced performance, due to the low-speed processor that sets an upper bound to the DCLS performance. Note that, to perform lockstep with hard cores, processors should have specific architecture support. However, this functionality is not present on all processors [10]. Non-intrusive approaches are less flexible as they do not modify the internal processor architecture, leading to higher communication overhead.

Intrusive approaches modify internally the processor architecture. Hence, when rollback mechanisms are applied, they do not require to insert checkpoints at the application level. For instance, the Dynamic Adaptive Redundancy Architecture (DARA) is a homogeneous DCLS approach that was applied to RISC ISA SH-2 processors and is based on rollback to achieve error correction [11]. DARA adds additional hardware to check the consistency of all pipeline stage registers, between the lockstep cores. However, the proposed mechanism does not support faults occurring in branch instructions. Interleaved multithreated execution is used to implement a dual lockstep approach using two virtual RISC-V cores [12]. Other approaches extend the pipeline registers with error detection and correction codes, e.g., Duckcore extends a RISC-V core with Single Error Correction Double Error Detection (SECDED) in the pipeline stages [13]. However, such an approach can add significant overhead due to the encode and decode time. Other approaches apply triplicate components inside the RISC-V core to enhance its repliability. For instance, Control and Status Registers, Program Counter and the register file [14], FFs, LUTs, BRAMS, and DSPs [15], and the arithmetic and logic unit (ALU) are triplicated [16]. Furthermore, existing fine-grained approaches are based on HDL implementations, which are significantly more complex than high-level description, and they do not focus on providing upper bounds regarding the error detection and correction time of the applied fault-tolerant hardware mechanisms.

III. Proposed fine-grained intrusive DCLS

This section describes the two proposed DCLS approaches, i.e., *Partial Shadow Register with Rollback (PSRR)* and *Full Shadow Register (FSR)*, and reports the upper bound w.r.t. their detection and correction time. In this paper, we assume that register file, instruction memory, and data memory are protected by an Error Correction Code (ECC) mechanism.

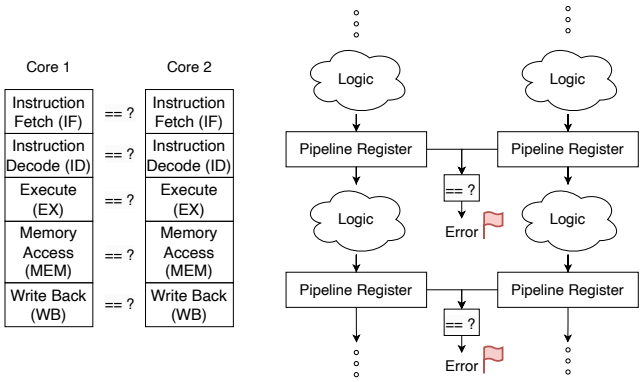

Fig. 1. Dual Core Lock-Step principle illustration

In both PSRR and FSR DCLS mechanisms, we use two identical cores executing the same instruction at each clock cycle. Each pipeline stage stores the result of its logic computation in a pipeline register. At each cycle, the proposed mechanism checks for execution consistency by comparing the pipeline registers of the two cores, as sketched in Figure 1. If no error is detected, the execution runs normally. Otherwise, the detected error indicates that a fault impacted the logic, which in turn generated a wrong result, or that a fault impacted the pipeline register itself, flipping a bit in the result. In this case, correction is applied, as described in the following sections.

A. Partial Shadow Register with Rollback (PSRR)

When an error is detected, the PSRR approach re-executes the instruction being processed in the faulty stage, i.e., it has to be re-fetched and go through the whole pipeline again. To be able to re-execute the instruction that was in the faulty pipeline stage, we modified the micro-architecture to store in each given pipeline stage the address of the current instruction being processed in that stage. In this way, when a fault is detected, we are able to retrieve the address of the instruction to re-execute. Furthermore, the result of the previous and current stages (that have already started processing the new instructions) have to be discarded, e.g. if an error was detected in the *EX* stage, the stages discarded are *IF*, *ID* and *EX*. As for the next stages, we let them continue as their execution is not impacted by the error, e.g., for an error detected in the *EX* stage, we let *MEM* and *WB* to continue. Finally, No Operation (NOP) instructions are introduced in the pipeline.

In the above described approach, the correctness of the address of the instruction, being processed in each stage, is key for the rollback mechanism to work correctly. A fault could modify such address, jeopardizing the rollback procedure. Therefore, we store a copy of such address for each pipeline stage – this is why we refer to the approach as *'Partial Shadow Register'*. This allows us to retrieve the correct address to rollback to, even when a fault impacts it in one core. To

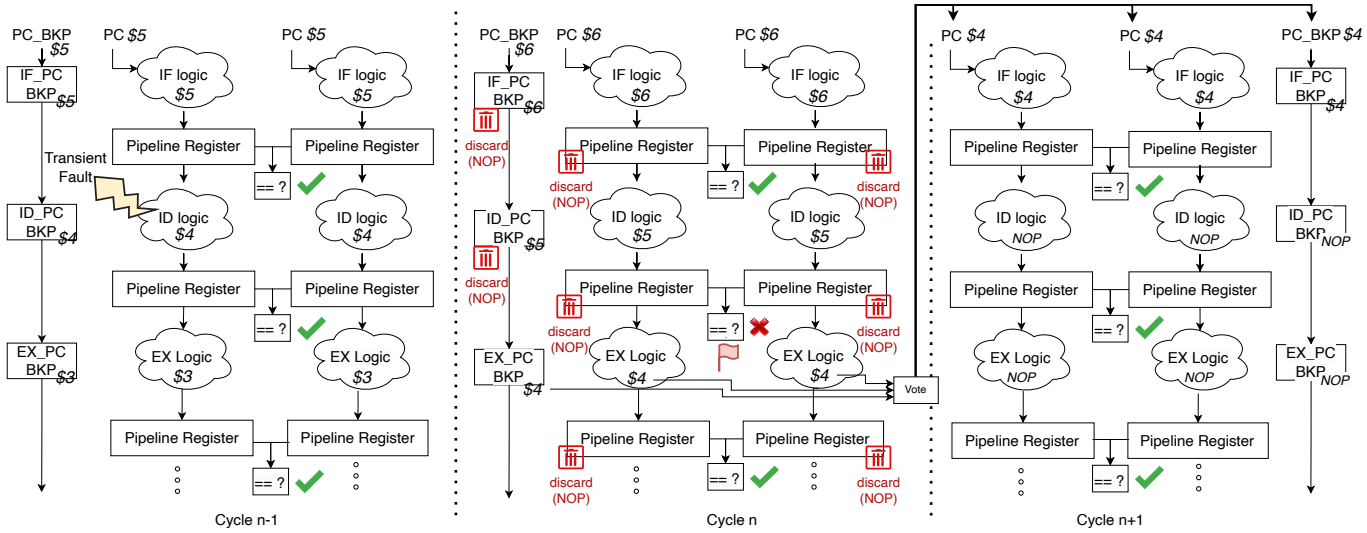

Fig. 2. PSRR error detection and rollback mechanism

achieve that, a vote among the addresses stored in the cores and the copy is made before rolling back. Figure 2 sketches the PSRR approach. To illustrate this mechanism, let us consider the assembly code snippet in the listing 1.

```
$1      addi    a5,a5,-16
$2      addi    a5,a5,-16
$3      add     a5,a4,a5
$4      sw      a5,-32(s0)
$5      mv      a4,a5
$6      jal     target
$7      mv      a4,a5
$8      mv      a5,a0
        ---
<target>
$9      sw        a5,-32(s0)
$10     mv    a4,a5
        ---
```

Listing 1. Illustration example program assembly code

TABLE I
PIPELINE STATUS FOR FAULT-FREE EXECUTION

	Execution cycle				
Pipeline stage	**n-1**	**n**	**n+1**	**n+2**	**n+3**
IF	5	6	7	8	9
ID	4	5	6	7	NOP
EX	3	4	5	6	NOP
MEM	2	3	4	5	6
WB	1	2	3	4	5

Table I shows a snapshot of the processor pipeline stages during a non-faulty execution of this program. NOP instructions at cycle $n+3$ in ID and EX stages are due to the jump instruction \$6. Let us suppose that, at some point during cycle $n-1$, an error impacts the result of ID stage, which is processing instruction \$4. As already mentioned, the result of ID stage can be impacted by a fault in the ID logic or directly in the pipeline register between ID and EX stages (IDtoEX). The mismatch is detected at cycle n by comparing the IDtoEX pipeline registers of the two cores in the DCLS. This makes invalid the instructions \$4, \$5, and \$6 in IF, ID, and EX stages. Hence, the rollback mechanism discards the execution results of these stages; the *MEM* and *WB* results are valid and instructions \$3 and \$2 will continue their execution, since they are not impacted by the fault effect. Finally, the Program Counter (PC) in cycle $n+1$ will be assigned to instruction \$4, after a vote among current instruction addresses stored in the cores and the copy (EX_PC_BKP in the figure). After the correction, we obtain the pipeline state shown in Table II. Finally, at cycle $n+2$ we obtain the same pipeline

TABLE II
PIPELINE STATUS WITH PSRR CORRECTION

	Execution cycle				
Pipeline stage	**n-1**	**n**	**n+1**	**n+2**	**n+3**
IF	5	*6*	*4*	*5*	*6*
ID	4	*5*	*NOP*	*4*	*5*
EX	3	*4*	*NOP*	*NOP*	*4*
MEM	2	3	*NOP*	*NOP*	*NOP*
WB	1	2	3	*NOP*	*NOP*

state that we had at cycle $n-1$ when the fault occurred (except for instructions \$1, \$2, and \$3 that were not impacted by the fault). Thus, in this example, the error detection and correction overhead is three cycles (from $n-1$ to $n+1$).

In general, the PSRR mechanism has a minimal error detection and correction overhead of two cycles: the cycle when the fault occurs and the cycle to detect the fault. Then, the deeper the impacted stage is in the pipeline, the more extra cycles are required. Thus, the upper bound in time for error detection and correction is given by the pipeline depth.

B. Full Shadow Register (FSR)

In the Full Shadow Register approach, we create a backup copy (BKP) of all the pipeline registers of the core, i.e.

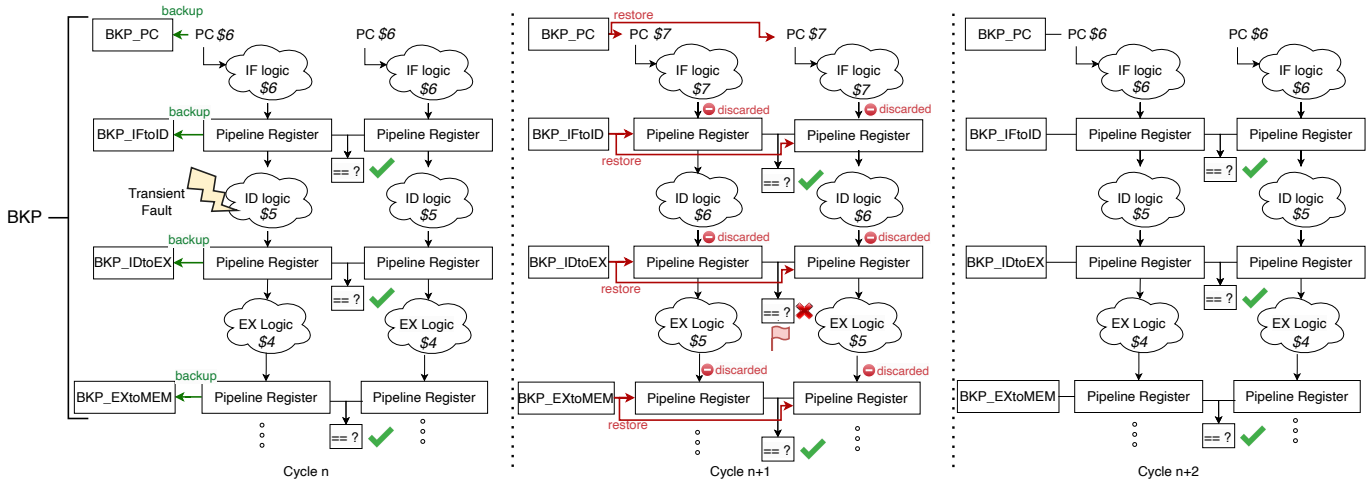

Fig. 3. FSR error detection and correction mechanism

for all the stages. The error detection and correction logic is the following: in each clock cycle n, we compare the pipeline registers of the two cores, containing the results of the computation of cycle $n-1$. If no error is detected, all the pipeline registers are copied to the backup copy BKP. When an error is detected in the comparison, a flag is raised, the results of the current computation are discarded and the pipeline registers of both cores are restored with the values in BKP. In this way, in the next cycle $n+1$, the pipeline will re-execute the cycle that was impacted by a fault. Figure 3 sketches the described FSR approach. Let us use again listing 1 as an example. As shown in Table III, let us suppose that,

TABLE III
PIPELINE STATUS WITH FSR CORRECTION

	Execution cycle				
Pipeline stage	**n-1**	**n**	**n+1**	**n+2**	**n+3**
IF	5	6	7	*6*	7
ID	4	5	6	*5*	6
EX	3	4	*5*	*4*	5
MEM	2	3	4	*3*	4
WB	1	2	3	*2*	3

at the end of cycle $n-1$, the computation had no errors. In cycle n, the pipeline registers are compared and, since there was no error, the content of the pipeline is copied to the BKP register. Let us now suppose that a fault impacts the ID stage in cycle n. In cycle $n+1$, the pipeline registers of the two cores are compared and an error is detected, due to the fault in cycle n. Thus, the results of the computations is discarded and the content of BKP is copied back. Finally – in cycle $n+2$ – the cycle impacted by the fault can be re-executed and the computations goes back to normal. The FSR approach entails a constant overhead of two clock cycles, namely the one where the fault occurred, and one to detect it and restore the BKP register. Therefore, compared to PSRR, FSR entails higher overhead to copy all the pipeline registers, but reduces the upper bound for the error detection and correction to two cycles.

C. Enhancement w.r.t. state of the art

Thanks to the high-level description of the proposed approaches and the RISC-V processor, we are able to perform fast iterations of both designing and evaluated through fault injection and to expose some drawbacks of the state-of-the-art approaches from which we derived our DCLS.

To give an example, the proposed PSRR approach has been inspired by DARA [11], where we were able to find and fix some shortcomings. To illustrate that, let us consider again the assembly code snippet in the listing 1 and the pipeline status in Table I. At cycle $n+2$ instruction \$6 – a `jump` – is in the EX stage. This will set the PC of the next cycle ($n+3$) to address \$9. Let us imagine that at cycle $n+1$ a fault impacted the results of the fetch stage, executing instruction \$7. The rollback approach used in DARA (and presented in section III-A) would detect the error in cycle $n+2$ and re-execute – in cycle $n+3$ – instruction \$7. However, this instruction was not meant to be executed, since instruction \$6 was an immediate jump to instruction \$9. We were able to spot this condition thanks to the proposed high-level DCLS implementation enabling fast design iterations and fault injections. Without any countermeasure, this condition would lead to wrong program execution, i.e., executing instructions that were not meant to be in the pipeline, ignoring the jump instruction. Thus, as a countermeasure, we added a further condition: if an error happens in a pipeline stage processing an instruction that will be discarded due to a jump being processed in the EX stage, we do not initiate the rollback procedure.

IV. EXPERIMENTAL RESULTS

A. Experimental setup

The open-source 32-bit RISC-V processor [9] consists of a standard 5-stage pipeline, i.e., Instruction Fetch (IF), Instruction Decode (ID), EXecute (EX), MEMory (MEM), Write-

Back (WB), including a forwarding mechanism, a hardware multiplier in its execution stage, and a Register File (RF) with 32 registers in the write-back stage.

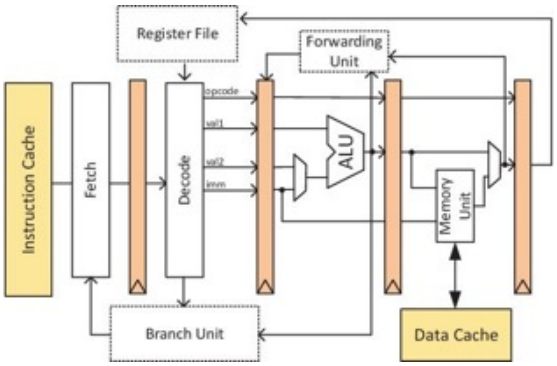

Fig. 4. RISC core with 5-stage pipeline, forward mechanisms, and data and instruction caches [9].

We perform Fault Injection (FI) experiments on four benchmarks, i.e., `Matmult`, `Qsort`, `Gaussian Filter`, `Moving Average`. We compare the FI results of three different configurations i) PSRR ii) FSR iii) the unprotected RISC-V. The FI campaign is achieved through a FI tool using a cycle-accurate bit-accurate (CABA) simulator for PSRR, FSR and the unprotected RISC-V processor versions. Prior to any fault injection, we execute the benchmark under study, without faults, in order to obtain a set of golden references: i) the application output, ii) the system state (processor registers), and iii) the number of cycles required for the execution of the application. Then, the core simulator executes the benchmark and injects faults to the registers, while the benchmark runs. The cycle to inject the faults is chosen randomly between the first cycle and the total number of cycles needed for the fault-free execution. The fault injection location is selected randomly following a uniform distribution. The injected fault is a bit-flip. After the fault injection and upon benchmark termination, the results are compared to the golden references, and the impact of faults is categorized as follows:

- *Execution Cycles Mismatch (ECM):* the execution cycles are different from those of the golden reference.
- *Hang (H):* the execution time of the application has exceeded a waiting threshold, and thus, it is assumed that it has entered an infinite loop. For these experiments, we set the threshold to eight times the golden number of cycles.
- *Crash (C):* the execution of the application has terminated unexpectedly and an exception has been thrown (out of bound memory access, misaligned PC, hardware trap, etc.)
- *Application Output Mismatch (AOM):* the application output is different from the golden reference.
- *Internal State Mismatch (ISM):* The system state (processor registers) are different from the golden reference.
- *Functionally Masked (FM):* The application has finished execution, with no difference w.r.t. the golden application.

In order to evaluate the proposed approaches, we follow the *statistical fault injection* approach [17] for each benchmark and processor version. The number of injected faults needed to obtain statistically meaningful results is given by $fault_injections = \frac{N}{1+e^2 \times \frac{N-1}{t^2 \times 0.25}}$, where N is the number of possible injection points (pipeline register bits $\times$ clock cycles), e is the desired error margin (1%), and t depends on the desired confidence level (t=3.1 for 99.8% confidence level). This leads to 25,000 injections per benchmark and processor version, giving a total of 300,000 injections.

B. Results

The Table IV shows the vulnerability metrics obtained by the FI campaign for the unprotected version and the two proposed DCLS mechanisms. The injected fault impacts significantly the unprotected processor, compared to the protected versions. When the fault is not masked, usually the application ends, but produces incorrect result (represented by the AOM). On the contrary, the DCLS RISC-V versions with the PSRR and FSM mechanisms are able to detect and correct the error.

TABLE IV
VULNERABILITY METRICS

Benchmark	AOM	ECM	ISM	Crash	Hang
Unprotected version					
`Matmult`	2,977	1,466	1,093	566	609
`Qsort`	930	1,687	1,451	737	629
`Gaussian Filter`	2,359	1,209	768	423	497
`Moving Average`	2,867	1,454	908	507	517
PSRR mechanism					
`Matmult`	0	18,165	0	0	0
`Qsort`	0	18,412	0	0	0
`Gaussian Filter`	0	16,819	0	0	0
`Moving Average`	0	17,568	0	0	0
FSR mechanism					
`Matmult`	0	22,141	0	0	0
`Qsort`	0	20,015	0	0	0
`Gaussian Filter`	0	18,034	0	0	0
`Moving Average`	0	18,948	0	0	0

A fault may significantly impact the timing behavior of the application, as highlighted by the number of ECM observed during the FI campaign in Table IV. For the unprotected version, the timing impact of the fault may be unpredictable, jeopardizing the timing bounds computed considering a fault-free system. Table VI depicted the maximum number of clock cycles when an ECM occured. For instance, for the `Gaussian Filter` the maximum observed cycles is $350,636$, whereas the fault-free execution is $59,084$ cycles, which is almost $6\times$ more. On the contrary, the proposed DCLS approaches bound the timing impact of the fault by detecting and correcting with a bounded overhead. From the experimental result, we have verified that the time to perform error detection and correction with FSR mechanism is constantly two execution cycles, whereas with the PSRR mechanism is bounded by 5 clock cycles. Note that, the ECM value of Table IV highlights how many times the proposed mechanisms have performed a rollback. The ECM difference between PSRR and FSR is the result of the selective protection of the PSRR, which performs

rollback from a mismatch, only if the value is going to be used.

TABLE V
ERROR DETECTION AND CORRECTION OVERHEAD

Approach	Max cycles observed	Additional bits
PSRR	5	160
FSR	2	716

Table V also shows the area overhead per protection mechanism. With the PSRR mechanism, we have achieved similar results compared to the FSR one, with lower area overhead.

TABLE VI
ECM VALUE FOR UNPROTECTED VERSION

Benchmark	Max cycles observed	Baseline cycle
`Matmult`	82,859	12,887
`Qsort`	33,125	4,429
`Gaussian Filter`	350,636	59,084
`Moving Average`	38,962	10,314

TABLE VII
PER PLACE IMPLICATION, NO PROTECTION

		Pipeline register (%)			
Benchmark	**Error Type**	**FtoDc**	**IDtoEX**	**ExtoMem**	**MemtoWb**
`Matmult`	AOM	18.61	34.53	14.58	10.14
	ISM	14.64	21.41	7.50	4.85
	ECM	23.47	36.43	13.23	8.66
	HANG	10.67	14.45	2.46	0.49
	CRASH	13.07	30.74	8.30	4.06
`Qsort`	AOM	20.22	33.98	13.87	6.13
	ISM	13.71	28.46	14.40	8.06
	ECM	21.64	37.05	16.54	7.94
	HANG	9.38	23.37	8.90	6.20
	CRASH	12.21	38.94	15.06	5.70
`Gaussian Filter`	AOM	19.92	39.38	12.80	4.92
	ISM	15.10	17.32	4.43	4.95
	ECM	24.32	41.85	11.00	4.05
	HANG	10.46	13.08	1.41	0.4
	CRASH	14.89	31.91	15.37	1.65
`Moving Average`	AOM	19.99	38.47	12.98	8.06
	ISM	15.20	18.94	7.38	6.06
	ECM	23.04	37.48	14.31	9.35
	HANG	10.25	8.90	1.55	1.16
	CRASH	12.62	38.86	11.64	2.76

Furthermore, the participation of each faulty pipeline register in each observed vulnerability class is shown in Table VII. For instance, for the `Gaussian Filter`, $1,209$ injections have produced ECM, and 41.85% of the $1,209$ (which corresponds to 506 injections) the injection occurred in the $IDtoEX$ pipeline register. We notice also the the $IDtoEX$ has a higher implication to the different error types.

V. CONCLUSION

This work focuses on critical systems that require high reliability and real-time execution, such as aerospace and defense systems, automotive, medical devices. Two fine-grained lockstep designs have been proposed for a RISC-V architecture using high-level specification language. Note that, the use of RISC-V architecture provides an open-source and customizable solution, which, along with HLS, allow the designers to quickly obtain reliable processors that can also be adapted to suit different reliability and timing needs, even for fine-grained intrusive approaches. The proposed mechanisms are based on shadow registers and rollback and ensure the correct execution of instructions by comparing the results of two identical processors at the end of each clock cycle. The mechanisms have been implemented and evaluated using a FI tool based on a CABA simulator and the results showed that they are able to quickly detect and correct errors with bounded performance overhead, whereas the consistency check is being done in parallel to the pipeline normal execution flow. Especially the PSRR approach has showed to be effective for dependable system while providing significant low area overhead compared to a TMR version.

Our future work directions include the implementation of the proposed DCLS designs on a FPGA and perform FI and extend the proposed approaches for Multiple Bit Upset (MBU).

ACKNOWLEDGMENT

This work has been funded by the French National Research Agency (ANR) through the FASY research project (ANR-21-CE25-0008).

REFERENCES

[1] S. Skalistis *et al.*, "Timely fine-grained interference-sensitive run-time adaptation of time-triggered schedules," in *IEEE RTSS*, 2019.
[2] A. Dixit *et al.*, "The impact of new technology on soft error rates," in *Int. Reliability Physics Symp.*, Apr. 2011, pp. 5B.4.1–5B.4.7.
[3] S. Rehman *et al.*, *Reliable Software for Unreliable Hardware: A Cross Layer Perspective*. Springer Publishing, 2016.
[4] F. Catthoor *et al.*, "Will chips of the future learn how to feel pain and cure themselves?" *IEEE Design & Test*, vol. 34, no. 5, pp. 80–87, 2017.
[5] S. Hamdioui *et al.*, "Reliability challenges of real-time systems in forthcoming technology nodes," in *IEEE/ACM DATE*, March 2013, pp. 129–134.
[6] M. Cui *et al.*, "Fault-tolerant mapping of real-time parallel applications under multiple dvfs schemes," in *IEEE RTAS*, 2021, pp. 387–399.
[7] C. Hernandez *et al.*, "Timely error detection for effective recovery in light-lockstep automotive systems," *IEEE TCAD*, vol. 34, no. 11, pp. 1718–1729, 2015.
[8] J. Abella *et al.*, "Security, reliability and test aspects of the risc-v ecosystem," in *IEEE ETS*, 2021, pp. 1–10.
[9] S. Rokicki *et al.*, "What you simulate is what you synthesize: Designing a processor core from c++ specifications," in *IEEE/ACM ICCAD*, 2019, pp. 1–8.
[10] Ã. B. de Oliveira *et al.*, "Lockstep dual-core arm a9: Implementation and resilience analysis under heavy ion-induced soft errors," *IEEE TNS*, vol. 65, no. 8, pp. 1783–1790, 2018.
[11] J. Yao *et al.*, "Dara: A low-cost reliable architecture based on unhardened devices and its case study of radiation stress test," *IEEE TNS*, vol. 59, no. 6, pp. 2852–2858, 2012.
[12] M. T. Sim *et al.*, "A dual lockstep processor system-on-a-chip for fast error recovery in safety-critical applications," in *IEEE IECON*, 2020, pp. 2231–2238.
[13] J. Li *et al.*, "Duckcore: A fault-tolerant processor core architecture based on the risc-v isa," *Electronics*, vol. 11, no. 1, 2022.
[14] L. Blasi *et al.*, "A RISC-V fault-tolerant microcontroller core architecture based on a hardware thread full/partial protection and a thread-controlled watch-dog timer," in *APPLEPIES*, 2019, pp. 505–511.
[15] A. E. Wilson *et al.*, "Neutron radiation testing of fault tolerant risc-v soft processor on xilinx sram-based fpgas," in *IEEE SCC*, 2019, pp. 25–32.
[16] D. A. Santos *et al.*, "A low-cost fault-tolerant risc-v processor for space systems," in *DTIS*, 2020, pp. 1–5.
[17] R. Leveugle *et al.*, "Statistical fault injection: Quantified error and confidence," in *IEEE/ACM DATE*, 2009, pp. 502–506.