Received 12 June 2024, accepted 2 July 2024, date of publication 9 July 2024, date of current version 19 July 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3425579

RESEARCH ARTICLE

Dynamic Triple Modular Redundancy in Interleaved Hardware Threads: An Alternative Solution to Lockstep Multi-Cores for Fault-Tolerant Systems

MARCELLO BARBIROTTA, FRANCESCO MENICHELLI, ABDALLAH CHEIKH, ANTONIO MASTRANDREA, MARCO ANGIOLI, (Graduate Student Member, IEEE), AND MAURO OLIVIERI, (Senior Member, IEEE)

Department of Information Engineering, Electronics and Telecommunications (DIET), Sapienza University of Rome, 00184 Rome, Italy

Corresponding author: Marcello Barbirotta (marcello.barbirotta@uniroma1.it)

ABSTRACT Over the years, significant work has been done on high-integrity systems, such as those found in cars, satellites and aircrafts, to minimize the risk that a logic fault causes a system failure, thus having functional safety as a key requirement. In this study, we employ an innovative approach to harness the benefits of both Dual Modular Redundancy and Triple Modular Redundancy techniques within an Interleaved-Multi-Threading microprocessor core, by means of a microarchitecture design capable of dynamically switching from Dual Modular Redundancy to Triple Modular Redundancy in case of faults. We explain the quantitative results obtained from an extensive fault injection simulation campaign on the fault tolerant core compared with its previous version regarding fault tolerant capabilities. The results show that in several application cases the fault resilience improvement and the hardware and timing overhead are better compared to the lockstep-based dual core approach. The proposed technique achieves 98,6% fault mitigation at the expense of only 4 clock cycles for roll-back overhead, with no checkpointing redundancy.

INDEX TERMS Circuit faults, digital integrated circuits, fault detection, fault tolerant computing, field programmable gate arrays, microprocessors, multithreading, radiation hardening (electronics), redundancy, robustness.

I. INTRODUCTION

The recent evolving technological era is scaling transistor feature sizes resulting in very densely integrated chips with elevated clock frequencies and low operating voltages [1], [2]. Due to such technological evolution, along with more and more complex computing system architectures, today's systems exhibit an increased susceptibility to transient errors triggered by radiation phenomena, leading to issues like bit-flip occurrences and the subsequent risk of failures [3], [4]. Transient errors are non-permanent faults caused by various phenomena, including electrical noise, electromagnetic interference, temperature fluctuations, or cosmic radiation. While they do not typically cause permanent damage, they still impact the correctness and stability of a system.

Mitigating transient errors involves implementing error-detection or error-correction mechanisms, as well as strategies for recovering or re-executing the affected parts of the application. Research efforts aim to develop techniques enhancing the overall resilience while paying attention to the overheads in cost and performance.

Ad-hoc techniques dedicated to fault tolerant control algorithms exist, based on fault detection and diagnosis.

The associate editor coordinating the review of this manuscript and approving it for publication was Yun Lin.

After detecting a fault, the control algorithm reconfigures to compensate for the fault effect and stabilize the system. The method of reduced-order observers [5] estimates system states and faults without being influenced by the estimation error. Dynamic Output Feedback [6] incorporates past and present output measurements to determine the control action. Our work, as well as the compared lockstep architectures, differs from algorithm-specific solutions as it relies on a hardware-based method for error detection and recovery within microprocessor cores executing generic application programs.

Triple Modular Redundancy (TMR), regardless of the abstraction level at which it is applied (e.g., registers, units, or sub-systems), offers the advantage of nearly immediate error correction but comes at the cost of tripling hardware resources or execution time. On the other hand, Double Modular Redundancy (DMR) imposes a lower overhead but typically requires a relatively expensive procedure to recover the correct system state. In fact, since DMR just allows error detection, a higher-level handler (e.g. at the software level) is invoked to restore a previously saved safe state (system *roll-back*). In addition to the performance penalty of the roll-back, the periodic saving of a safe state (system *checkpoint*), inherently impacts the performance. Consequently, traditional DMR implementations incur a cost globally exceeding twice that of a non-redundant system [7].

In the technical literature and industrial practice, lockstep dual-core techniques represent the most used approach for introducing DMR redundancy to enhance the reliability and resilience of microprocessor systems. Lockstep architectures consist of two identical processor cores executing the same instruction flow, with a shift of some cycles to introduce time diversity resilience, periodically comparing their results, along with checkpoint and rollback recovery procedures. While it can be considered an established solution, adopting a lockstep dual-core configuration presents its challenges, including the need for core synchronization, extra delay components and data storage for comparisons and checkpoints [8].

In [9], starting from a RISC-V Interleaved-Multi-Threading (IMT) core equipped with a temporal-spatial TMR scheme, we developed a special form of DMR scheme called *Dynamic TMR*, based on thread duplication along with a third thread that is dynamically activated only in case of a recovery procedure. From the software point of view, the architecture operates like a lockstep dual core while being implemented by a dual IMT execution to reduce the hardware impact.

The innovation of the proposed approach is to combine the lightness of DMR with the safety of TMR without compromising performance, and also drastically mitigating the cited gaps and limitations of DMR-based lockstep architectures, such as checkpointing and rollback overhead.

This work significantly extends the results presented in [9] through a deeper design implementation discussion and a novel detailed comparison with other fault-tolerant (FT) architectures presented in literature, addressing power consumption, achievable clock frequency, hardware resource utilization and checkpoint/recovery timing degradation. The results are obtained by an extensive fault injection campaign with Single Event Upset (SEU) faults targeting all register bits in the microarchitecture while executing widespread benchmark application kernels.

The main contribution of this work to the state of the art is composed of the following:

- The work provides a comparison between *IMT* and multi-core approaches to fault tolerance. The demonstrated advantage of the proposed method is that *Dynamic TMR* behaves as a *DMR* lockstep dual core yet without the overhead of duplicating the whole hardware and with drastically reduced impact of checkpoint saving and rollback;
- It illustrates the detailed hardware modifications required in an IMT core to implement the *Dynamic TMR*;
- It details the advantage of the proposed approach in terms of in-depth quantitative data on the Architecture Vulnerability Factor (AVF) and Mean Work Between Failure (MWBF), derived from an extensive fault simulation campaign;
- It details all the comparisons of results with the most important studies on Dual Core and Triple Core lock-step techniques in the literature. The reported comparison data cover mitigation rate, checkpoint time, recovery time, and hardware overhead.

The rest of the article is organized as follows: Section II discusses the related works about DMR and lockstep architectures, providing the basis for performance comparison; Section III discusses the microarchitectural details of the proposed design; Section IV reports the performance evaluation methodology along with the results; Section V analyzes the hardware overhead; Section VI discusses the results compared to other FT architectures; Section VII summarizes the conclusions.

II. RELATED WORKS

DMR techniques intrinsically had significant advantages over TMR concerning area overhead and power consumption reduction, with less efficiency and reliability. Several research works have been presented aiming to enhance DMR approaches. In [10], three different checkpointing schemes for DMR in multi-core architectures, namely Store-Compare-Checkpointing (CSCP), Store-Checkpointing (SCP) and Compare-Checkpointing (CCP), are compared to determine the optimal checkpointing frequency for minimizing the Mean Execution Time of a single task. In [11], the authors employ DMR with design diversity between two modules to generate distinct error patterns and easily detect mismatches, still relying on checkpointing and restoration procedures. More recently, in the study by Popov et al. [12], the authors introduce a DMR scheme with global/partial reservation, performing a comparative analysis of its reliability with respect to TMR. The DMR approach with global reservation involves a duplicated DMR structure comprising four identical modules. When one of the modules in the first pair encounters a failure, it is dynamically replaced by the other pair. Similarly, the DMR with partial reservation employs four identical units, and when one of the two units fails, only the failed unit is substituted with a reserved one. The study indicates that TMR is more reliable than global reservation DMR but less reliable than partial reservation DMR, which is more expensive regarding hardware resources. A novel technique called Complementary Double Modular Redundancy (CDMR), inspired by the Markov Random Field (MRF) theory, is introduced in the work by Li et al. [13]. The technique involves optimizing two voting stages with MRF, leading to improved performance, reduced overhead and enhanced fault resilience in the voting logic.

It is worth noting that redundancy is not always mandatory for the whole duration of an application execution. DMR may be activated only when needed, based on runtime Architectural Vulnerability Factor (AVF) estimation [14]. Nomura et al. [15] present a method for implementing sampling-DMR, which enables using DMR for just a small fraction of time, such as 1% of a 5 million-cycle time slot. This approach yields benefits in terms of power consumption and design complexity. Furthermore, Matsuo et al. [16] introduce an FPGA-CPU architecture featuring a self-monitoring scheme with health indicators. These indicators, which track parameters like temperature and system usage, are used to determine the health state of both systems, enabling selecting the most reliable system and facilitating software-based voting when results become available. Authors in [17] created a multi-core processor with four cores (single-cycle RISCV RV32IM instruction set) working in DMR pairs that can be reconfigured to activate a normal execution mode or a fault tolerant execution mode, so that the number of executed instructions per time unit is reduced only in the second case. When the processor is in fault tolerant mode, the two pairs of cores execute two copies of the same instructions. A mismatch in the final comparison of the two results can lead to keeping the same program counter and re-executing the incorrect instruction.

The most used DMR technique in industrial applications is the dual-core lockstep (DCLS) architecture. DCLS was implemented in FPGA by [18] using a Virtex II-Pro platform. Authors in [19] present a DMR system which combines a lockstep approach for error detection and checkpointing for error recovery. They employ a checker logic block to identify errors and periodically generate an interrupt request for checkpoint creation through a DMA transfer. Similarly, in [20] a DMR system is implemented using a dual-core lockstep architecture involving two cores within an ARM Cortex-A9 processor. This setup runs the FreeRTOS operating system and employs interrupts to manage checkpoint operations and rollback procedures. When a CPU triggers an interrupt, the active thread is halted, and the processor registers are saved in the stack. Authors in [21] built the SafeLS, a DCLS architecture made by two NOEL-V RISC-V cores, testing two different Sphere of Replication (SoR) called FullCore and CachesExcluded. In the first one, the SoR includes caches and a memory management unit (MMU); in the second one, these are excluded. Similarly, they also worked on SafeDE [22], which contains a monitoring module to enable lowly intrusive diverse redundancy with a flexible scheme that can be turned on or off conveniently in a lightweight lockstep environment. Nikiema et al. [23] proposed a fine-grained DCLS using HLS with two different approaches and different upper bound detection and correction levels: Partial Shadow Register with Rollback (PSRR) and Full Shadow Register (FSR). The two approaches use two identical cores executing the same instruction at each clock cycle, and each pipeline stage stores the result of its logic computation in a pipeline register, checking the execution consistency every clock cycle. In the first approach, when a fault is detected, the current instruction (stored in a shadow register) is re-fetched again inside the pipeline. In contrast, in the other approach, all the shadow registers are substituted inside the pipeline. Another approach is detailed in [24], where a DCLS configuration is built with a 666 MHz Arm A9 hard-core and a 25 MHz LowRISC soft-core on the Zynq-7000 platform. This system allows for the suspension, resumption, restoration from checkpoints and seamless continuation of execution. In [25], Silva et al. introduce CEVERO, a RISC-V System-on-Chip deployed on a PULP platform. CEVERO comprises two Ibex cores operating in a lockstep configuration, which are continuously monitored by a hardware unit to check for the occurrence of errors in each executed instruction. The traditional DCLS configuration is also found in systems based on commercial ARM Cortex-M7 processors [26] and Cortex-R processors. The recovery procedure in these systems involves resetting the cores or returning to a previously save checkpoint. In Iturbe et al. [27], a Triple Core Lockstep (TCLS) architecture is introduced, comprising three identical Cortex-R5 lockstep cores, managed by a central Assist Unit responsible for voting on the generated outputs. Specific assembly routines are triggered when discrepancies arise to save the state within a dedicated ECC-protected Stack (113 registers). These registers are restored after a global reset, requiring around 2351 clock cycles (including the save and restore time). Authors in [28] propose two FPGA-based methodologies (Lock-VA & Lock-VM) in which a heterogeneous architecture uses the DCLS technique at the ISA level. Each core receives the same input, and the system can pause, resume, or revert to a checkpoint. In another approach [29], a heterogeneous architecture compares two ARM cores to a MicroBlaze, operating in a TCLS fashion. The design incorporates verification points in the code to store the execution state within the FPGA BRAM memory and compare the registers. As soon as the first ARM core reaches one of these VPs, the status of the core is saved inside the BRAM memory, and the execution is locked until the other ARM core reaches the same point and the produced results are compared. If the comparison gives no error, the VP is saved as a checkpoint, and the execution continues. In contrast, in case of errors, the state of the core is compared with that of the MicroBlaze core, finding and recovering the faulty ARM core. Finally, Rogenmoser et al. [30] introduced a pioneering approach called Hybrid Modular Redundancy (HMR) to reduce the additional costs associated with conventional methods of radiation hardening and modular redundancy, involving a cluster of RISC-V processors in a versatile DCLS and TCLS arrangement. The arrangement can be adjusted in real-time with two recovery methods, one software-based and the other hardware-based, trading-off between performance and space utilization.

Regarding lockstep techniques, it is important to distinguish between two possible approaches, namely non-intrusive and intrusive [23]. Non-intrusive ones (actually, the majority of those described above) do not modify the core internal architecture (allowing COTS or third-party IP usage). The lockstep execution is achieved by a synchronization module wrapper, which checks for a mismatch between the status of the cores at the interface level generating interrupts in case of faults. Software procedures periodically create checkpoints and are responsible for rolling-back in case of fault revelation. Conversely, intrusive approaches modify the internal hardware structure of the core and do not generally require creating checkpoints at the software level since faults are detected as they occur (single bits in the case of SEU), requiring execution of just the last instruction. Examples of such techniques are [23] and the approach presented in this work.

III. DYNAMIC TMR: A NEW APPROACH

The DMR architecture presented in this work evolved from the *Buffered TMR* technique described in [31] and [32]. The Buffered TMR exploits the intrinsic capability of an Interleave-Multi-Threading (IMT) core of running three identical threads, each having its own Register File (RF), Program Counter (PC) and Control/Status Registers (CSRs), yet sharing the pipeline logic. Each instruction reaching the execution stage stores its result in thread-dedicated write-back buffers, and voting is applied when all results are available, thus performing an intrinsic TMR protection and correct result retention. In an IMT pipeline, executing the same instructions in identical threads on the same hardware units in different clock cycles protects the architecture from both Single Event Upset (SEU) faults in sequential logic and Single Event Transient faults (SET) in combinational logic [9].

To mitigate the performance loss associated to Buffered TMR [9], we conceived an architecture that we named *Dynamic TMR*, in which two identical threads are executed in a DMR Interleave-Multi-Threading pipeline, while the third thread is only activated in case of fault detection. The Buffered TMR issue is that it implies a fixed execution time triplication, which is efficiently mitigated by the proposed Dynamic DMR technique. The inactive thread will not insert any instruction in the pipeline, ideally saving 1/3 of the execution time with respect to the Buffered TMR, thus resulting in higher perceived performance for the active threads.

Our baseline processor allows for three hardware threads: Thread 2, Thread 1 and Thread 0. In the Dynamic TMR scheme, only Treads 2 and 1 are normally active, while Thread 0, which we call the *auxiliary thread*, is activated just in case of fault detection to implement instruction-level roll-back and voting.

IV. KLESSYDRA-dfT03: THE MICROARCHITECTURE

In conventional DMR systems like dual-core lockstep (DCLS), periodic checkpoint procedures introduce a time overhead and hardware overhead to store checkpoint data. Furthermore, depending on the fault rate the system is subject to, re-executing many instructions that were correctly completed uselessly adds time overhead to the computing system.

In the proposed microarchitecture, the hardware IMT mechanism is exploited to save and retrieve the correct state with instruction granularity, minimizing the performance impact of checkpointing and restoring operations. The system is able to restore the state corresponding to the instruction preceding the clock cycle in which the fault was detected, and only the instruction that experienced the fault is repeated. The checkpoint saving mechanism requires introducing a single register that saves the address of the last correct instruction and restores it in the PCs of the three threads in case of fault. This is possible since the result of an instruction is written to its destination (either the register file or the data memory) only after voting among the three results produced by the same instruction in the three threads, so the processor state is never corrupted and does not need restoring.

Figure 1 depicts the essential elements of the proposed design. The microarchitecture resembles a classical 4-stage single-issue in-order pipeline, with the IMT-specific feature of having multiple, thread-dedicated PCs, RFs, and CSRs. In the execution stage, a Load-Store (LS) unit manages memory accesses while all other operations are managed by the Execution unit. In the Figure, blocks indicated with "ECC" represent Error Correcting Code logic applied to values coming from the data and program memories, blocks indicated with "V" represent voting logic among three values, while blocks indicated with "D" represent fault detection logic by comparison of two values.

Overall, the architecture may operate in three *modes*:

- **Normal, or Detection, mode:** Identical Threads 2 and 1 execute the same instructions in an interleaved fashion to provide spatial and temporal redundancy while the PCs, RFs, Write-Back (WB) buffers and LS buffers implement a buffered comparison to possibly detect a fault occurrency. The signal flows in this mode are represented by blue arrows in Figure 1).

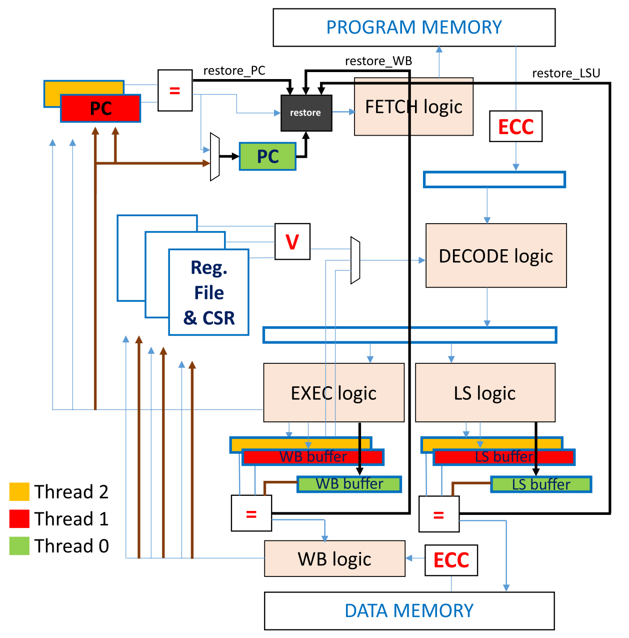

FIGURE 1. **Klessydra-dfT03 microarchitecture. Blue arrows: Normal mode; black arrows: Restore mode; brown arrows: End Restore Phase.**

- **Restore mode:** If the detection logic identifies a fault, specific control signals labelled as "restore_" in Figure 1 are activated, causing the core to enter the Restore mode. Notably, faults are detected before writing any result in the RF or the memory. The signal flows in this mode are represented by black arrows in Figure 1). The "restore_" signals trigger the restore block, waking up the inactive auxiliary thread. As the new thread enters the IMT pipeline, it retrieves the address of the last successfully executed instruction, indicated by the Checkpoint-PC register (further discussed in the next section), while the other threads hold a stall state. The duration of Restore mode is never longer than 8 clock cycles, and it is 2 cycles in the vast majority of cases.
- **End-of-Restore mode:** Once the recovered instruction is executed, the resulting output is compared with the non-matching results previously generated by Threads 2 and 1, implementing a TMR majority voting system. The signal flows in this mode are represented by brown arrows in Figure 1). The voted value is then written back into the RFs or the memory, and the recovery procedure ends with the suspension of the auxiliary Thread 0 and the loading of the subsequent instruction address into the PCs of Threads 2 and 1, continuing the execution in Normal mode. The duration of the End-of-Restore mode is 2 cycles.

The following description outlines the modifications to the three main architectural units required to implement the *Dynamic TMR* (dTMR) technique: the PC unit, the LS unit, and the RF Write-Back unit.

A. PROGRAM COUNTER UPDATE AND RESTORE UNIT

A fault in a Program Counter (PC) can result in an invalid instruction fetch or a spurious jump. To mitigate this risk, if a fault affects a PC value, the dedicated detection will detect it and trigger the Restore operating mode.

In any multi-threading processor structure, each Thread operates with its own PC. In the Klessydra-dfT03 microarchitecture, the PC of the auxiliary Thread 0 is used as a checkpoint register (Checkpoint-PC), saving the address of the last correct instruction for potential recovery if a fault is detected. During Normal operating mode, the Checkpoint-PC is updated every two cycles (corresponding to the interleaved time slots of Thread 2 and Thread 1, respectively). The updated value is produced by the agreement of the PCs of Thread 1 and Thread 2, obtained by the dedicated detection logic. Figure 2 depicts the temporal instruction flow, representing threads by different colours within the instruction pipeline. When the Restore procedure starts as a consequence of a PC fault detection, the PC of Thread 0 already contains the address of the last uncomitted instruction (green colour). As a result, the instruction is re-fetched, re-inserted in the pipeline and executed again. Under the reasonable assumption that no further faults occur during Restore mode (a statistically realistic scenario, considering typical fault rates [33]), the execution of this last correct instruction yields the valid address of the subsequent instruction, now stored in the PC of Thread 0. This address is then loaded in the PCs of Thread 2 and Thread 1, enabling fetching the next correct instruction and resume Normal operating mode.

The Program Counter Update/Restore unit structure is illustrated in Figure 3. It contains the standard logic for updating the PC in case of jumps, exceptions, or regular operations. Additionally, it includes a block designed to handle the "Thread Sleeping" function, which is responsible for waking up and suspending threads. The "PC voting & restoring block", along with the "PC control block", manages the detection process based on signals coming from the executing threads. Furthermore, it supervise the Restore operating mode, based on the restore_ signals received from the LS unit and the RFs.

B. LOAD STORE UNIT

The Load Store (LS) unit has been deeply redesigned for fault tolerance with respect to the original unprotected core due to the specific features of memory access operations.

Barely replicating memory operations would lead to repeating the effects of multiple memory accesses to the same location. While this may be harmless when loading/storing data from/to the memory, replicated read or write accesses at special addresses, such as memory-mapped peripherals, could produce undesired side effects. Moreover, replicating store operations to the data memory would result in storing the sole value written by the last thread accessing the memory without further checks. Therefore,

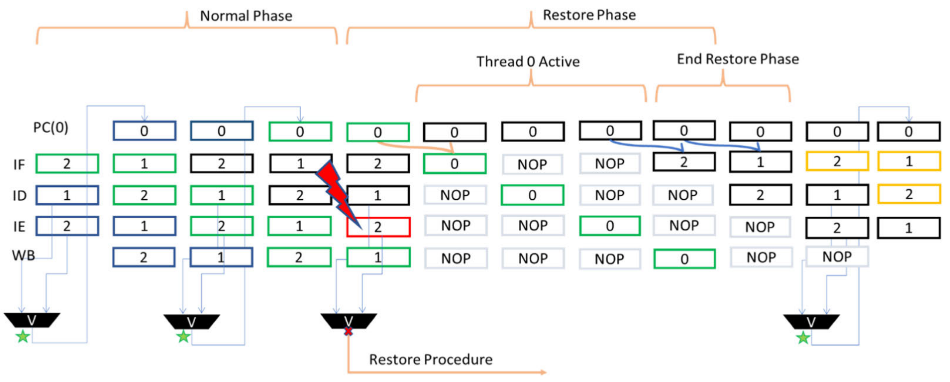

FIGURE 2. Restore phase for program counter unit. The green stars represent successful comparison, while the red x represents a detected error.

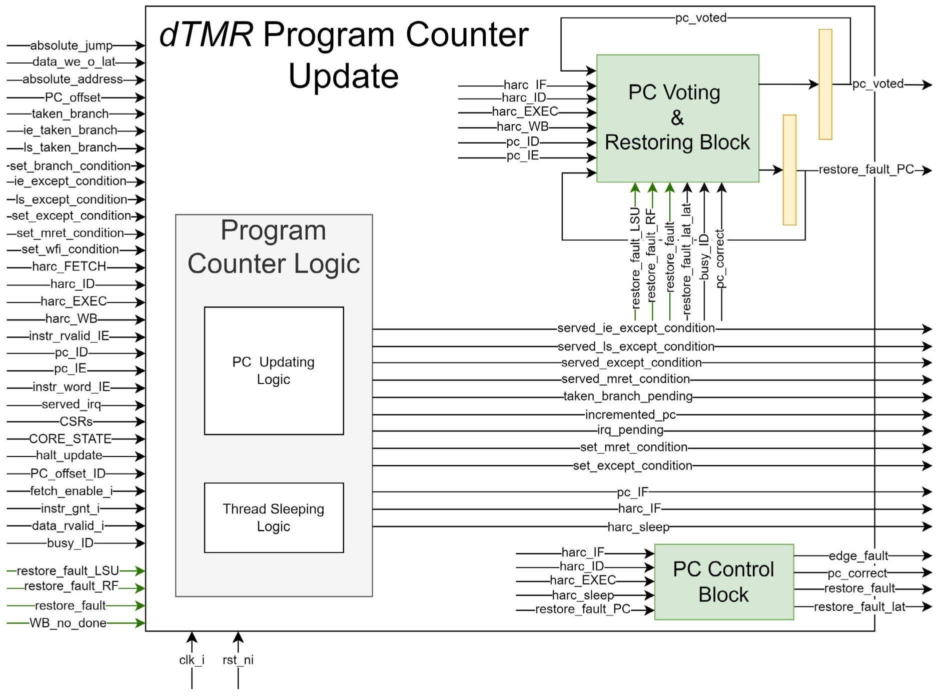

FIGURE 3. Internal structure of the program counter update unit.

similarly to the solution adopted in [32], we designed an LS unit capable of buffering all the signals produced by load/store operations in Thread 2, then comparing with the signals produced by Thread 1 in the next clock cycle and executing a single load/store memory access after error checking.

Notably, since buffering logic and comparison logic resulted in worsening the delay of the post-synthesis critical path in the LS unit, we decided to augment the internal state machine from two states to three states, one of which dedicated to the comparison operation. The latency increase affects only the load/store operation executed by Thread 1, so that the total latency passes from 4 cycles to 5 cycles for both threads. The resulting reduction in the critical path delay is largely dominant over the increased latency of load/store instruction execution.

The LS unit internal structure is shown in Figure 4. The ‘‘dTMR Logic Control Block’’ is responsible for buffering the signals coming from the threads and is required for the comparison operation. Inside this block, further units manage the handshaking signals, such as ‘‘load_valid’’ and ‘‘store_valid’’, which notify when a thread has finished buffering, and the load/store execution status signals, such as ‘‘LS_is_running’’, which notifies if there is a flying memory access.

The ‘‘dTMR Logic Control Block’’ operation is controlled by a Finite State Machine whose specification is sketched in Algorithm 1 as a pseudo-code state diagram. The buffered

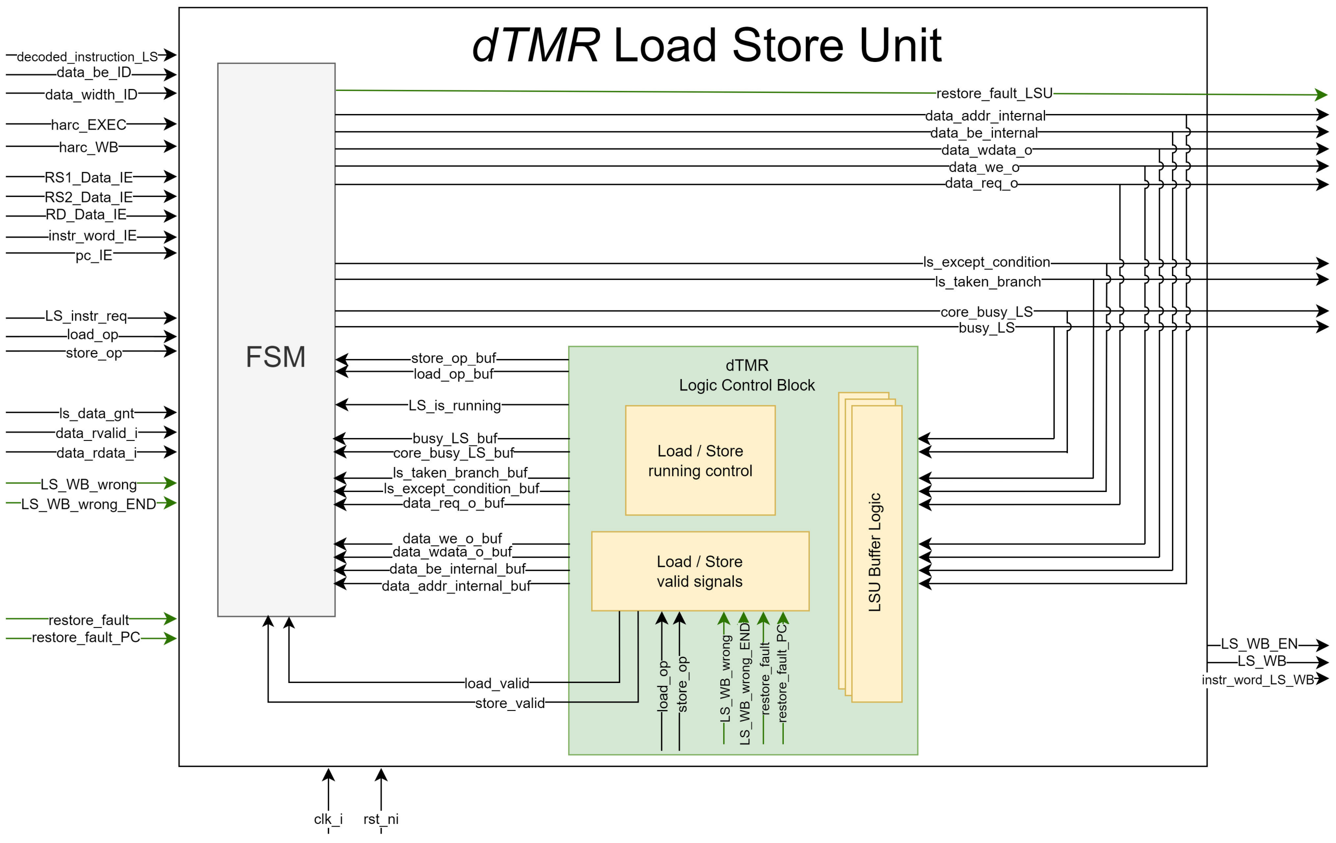

FIGURE 4. Internal structure of the Dynamic TMR Load Store Unit (LS unit).

values address, and data are used during the comparison step. In case of successful comparison the load/store access is executed to memory/peripherals. In case of mismatch (fault detected) the “restore_LSU” signal is activated, directly connected to the “restore block” (as visible in the microarchitecture pipeline in Figure 1).

A cycle-accurate waveform diagram of the signals involved is shown in Figure 5. It is possible to observe the buffering of some of the signals responsible for the load/store operations and then the comparison cycle during the execution of Thread 1.

Finally, the LS unit also receives signals from other fault detection units (green arrows in Figure 4), such as the “restore_fault_PC” signal, which notifies a fault detection occurrence in other units, to be followed by a pipeline flush and start of Restore operating mode.

C. REGISTER FILE AND WRITE BACK UNIT

The key component of the dynamic TMR microarchitecture is the Write Back (WB) management unit connected to the Register Files (RFs). Similarly to the *Buffered TMR* scheme [32], the results coming from the EXEC unit are temporarily saved in WB buffer registers associated with the two running threads and immediately compared for fault detection. In the proposed scheme, a critical aspect is represented by the presence of only two threads interleaved in the pipeline. Delaying the actual writing of the results until after both are ready to be compared causes the need for data bypass logic (usually avoided in pure IMT processor cores [34]).

Figure 6 shows the WB unit internal structure including the RFs. It is possible to observe how the bypass block generates the two bypass signals (for the two source operands) “rs1_bypass” and “rs2_bypass”, based on the destination register used during the previous instruction and the information from the decoding unit.

When in Restore mode, the result of the faulty instruction re-executed by the auxiliary thread is also saved in a third WB buffer, and majority voting is performed to solve the faulty result. When each thread produces the result of the instruction, the “Write Back Buffer control” block inserts it into the respective buffer. The value from the voting process, saved in the “WB_buf_voted” register, is written to the RF when the “WB_EN Buffer Logic” asserts the WB enable signals “WB_EN_buf”.

When in Normal mode, the “Fault control block” notifies discrepancies while comparing the WB buffer values. These can be due to faults in data and/or control signals during the instruction processing in the pipeline.

The most complex case is when a fault is detected in a long latency operation (e.g. integer division), and the previous instruction was a data processing operation that has already

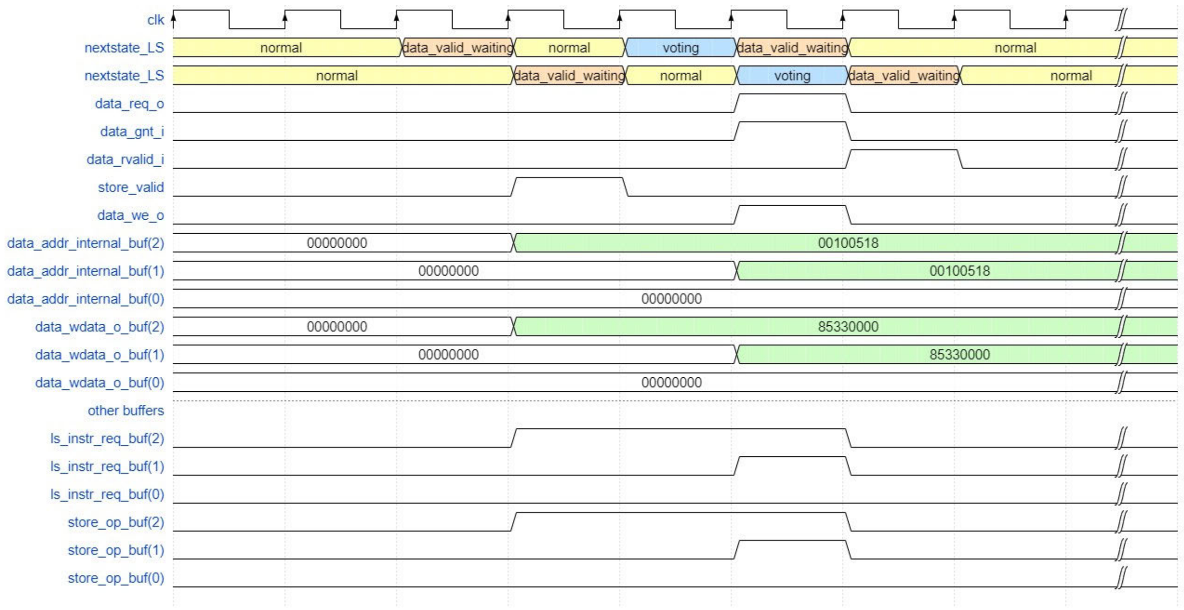

FIGURE 5. Load store unit: waveform implementation.

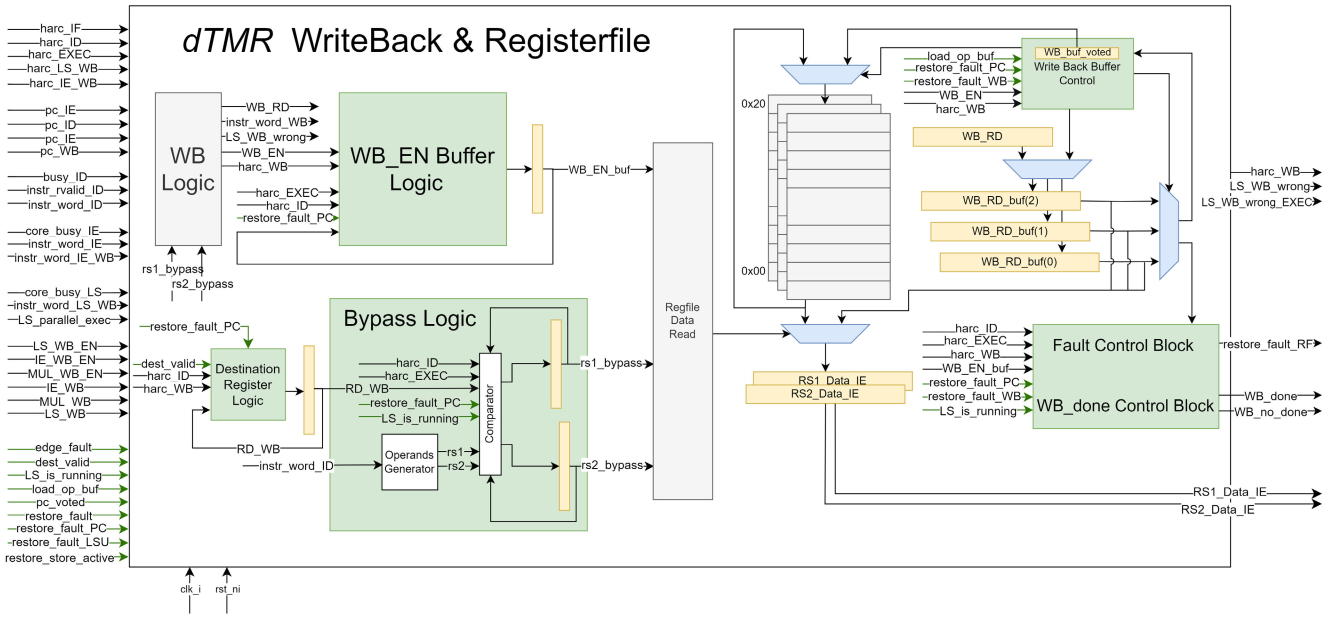

FIGURE 6. Internal structure of the write-back and register file management unit.

completed the WB phase. The recovery procedure starts with a pipeline flush, and the preceding data processing instruction is executed again. All the units also take the *harc* (hardware thread counter) signal into account, which notifies which thread the instruction belongs to, as well as fault detection signals (in green color) coming from the other fault detection units.

V. FAULT TOLERANCE ANALYSIS

The applied Fault Injection (FI) analysis is based on the Time Frame Spanning method described in [35], which deterministically targets each of the synchronous register bits within the microarchitecture, while simulating the execution of a set of software test programs at the Register Transfer Level.

At first, the FI procedure was implemented on the baseline microarchitecture lacking any fault tolerance mechanism (Klessydra-T03 [36]. This preliminary analysis identifies those bits that lead to program failures when subjected to faults. Such bits are called Architecturally Correct Execution (ACE) bits [37] and are inherently associated with the executed software program. We used this information to

Algorithm 1 For Finite State Machine LSU

```
1:  when $state = normal$
2:    if $load\_oporstore\_op$:
3:      [output assignment for buffered load/store op]
4:      $nextstate\_LS \leftarrow voting$ when ($harc\_EXEC = 1$
5:        or $harc\_EXEC = 0$) else $data\_valid\_waiting$
6:    endif
7:  when $state = voting$
8:    if $load\_op\_buf$ or $store\_op\_buf$ or $LS\_is\_running$:
9:      if $harc\_EXEC = 0$:
10:       [output assignment for real load/store from
    Thread 0 buf.]
11:     elsif $restore\_fault\_PC = 0$:
12:       if $voting = 1$: (fault)
13:         [ output assignment data retention ]
14:         $restore\_fault\_LSU\_wire \leftarrow 1$
15:         $nextstate\_LS \leftarrow data\_valid\_waiting$
16:       elsif $voting = 0 and restore\_fault\_PC = 0$:
17:         [ output assignment buffered load/store ]
18:         $LS\_is\_running\_wire \leftarrow 1$
19:         $nextstate\_LS \leftarrow data\_valid\_waiting$
20:       elsif $voting = 0 and restore\_fault\_PC = 1$: (restore)
21:         [output assignment data restoring]
22:         $nextstate\_LS \leftarrow normal$
23:       endif
24:     endif
25:   endif
26: when $state = data\_valid\_waiting$
27:   if $data\_rvalid\_i$ or $load\_valid$ or $store\_valid$:
28:     if $store\_op$ or $store\_op\_buf\_wire$ :
29:       [ output assignment receiving data]
30:     elsif $load\_op$ or $load\_op\_buf\_wire$ :
31:       [ output assignment sending data]
32:     endif
33:     $nextstate\_LS \leftarrow normal$
34:   else
35:     $nextstate\_LS \leftarrow data\_valid\_waiting$
36:   endif
```

reduce the fault analysis simulation time in the fault-tolerant microarchitecture by limiting the analysis to ACE bits since non-ACE bits do not cause failures anyway.

The analysis assumes that no additional fault occurs during the few clock cycles in Restore mode consequent to a fault (a statistically quite realistic assumption in the real-world scenarios of interest). To speed up the simulation, fault injection (FI) is active during Normal operating mode. The analysis of the FI results focuses on Single Event Upset (SEU) effects and assumes that physical SEU faults manifest uniformly in time and in space. It is important to note that concepts such as *error margin* and *confidence level*, as used in the classical Monte Carlo random FI, can not be applied in this approach because it does not test a statistical sample of the population of bits. Instead, it entails testing the entire ACE bit population in the microarchitecture to assess the achieved FT coverage.

A. TIME-FRAME-SPAN METHODOLOGY

Different from a statistical Monte-Carlo approach, where faults are randomly distributed during program execution, in the time-frame-span methodology, the faults are injected deterministically on each target bit of the microarchitecture within specific time intervals during the whole program execution. The approach leads to assessing the time percentage of the program execution during which a target bit of the microarchitecture is Architecturally Correct Execution (ACE), and ultimately the average number of ACE bits in a generic clock cycle during program execution.

The program execution time is preemptively divided into m intervals, referred to as time frames. For each target bit j in the architecture, the RTL simulation of the program execution is run m times, one for each selected time frame, while injecting faults on bit j only within the selected time frame. If a fault injected on bit j produces a program failure, the selected time frame is marked as failing. Supposing we collect m_F failing time frames out of the total m and assuming that SEUs in the physical system have a uniform time distribution, we can estimate the probability that bit j is an ACE state for the architecture:

$$P_F(j) \leq \frac{m_F(j)}{m} \qquad (1)$$

As expressed by (1), the analysis establishes an upper bound to the failure probability [9].

For the scope of the present work, we set the time frame length as 1/10 of the benchmark execution time. In the view of a characterization of the risk according to standards as ISO 26262, which is beyond the scope of this research, the accuracy of the probability estimation may be improved by refining the time resolution of the FI time frames.

The obtained failure probabilities lead to an upper limit for the average number of ACE bits N_{ACE}, calculated as the sum of the estimated failure probabilities of all the N register bits in the microarchitecture during the program execution [37], [38], [39], as expressed in (2), while N_{ACE} can be utilized to derive the Architectural Vulnerability Factor (AVF) as stated in (3).

$$N_{ACE} = \sum_{j=1}^{N} P_F(j) \qquad (2)$$

$$\text{AVF} = \frac{N_{ACE}}{N} \qquad (3)$$

The obtained AVF expresses the average fraction of bits in the architecture that are vulnerable to faults (i.e. they cause a failure if faulted) in the given application execution. The AVF value can also be used to calculate the Mean Work to Failure (MWTF) value, which expresses the average amount of work (i.e. application runs) that a system can perform until reaching a failure [40], [41] when subject to a certain

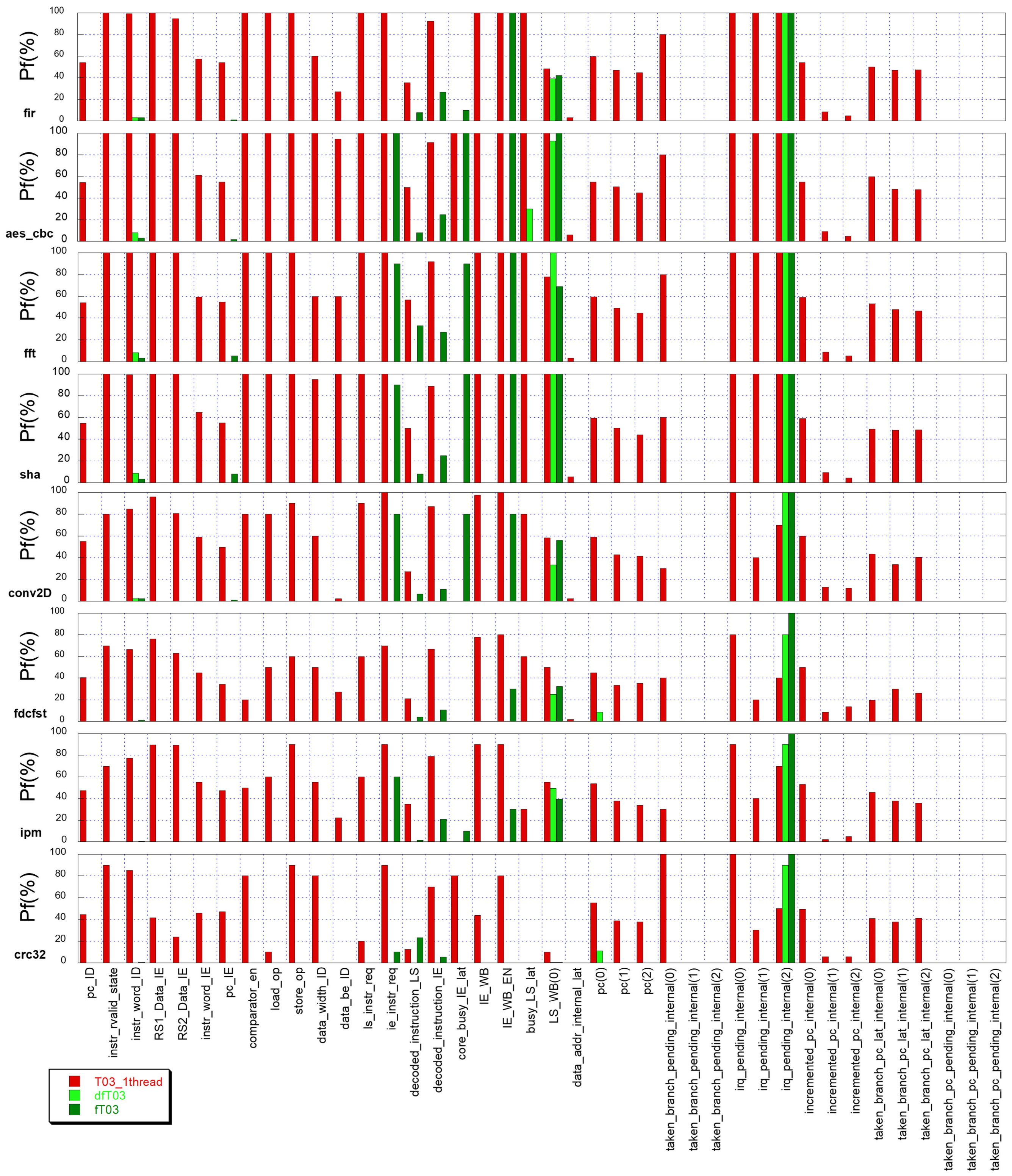

FIGURE 7. Fault injection results with a time frame spanning approach for the T03, fT03, and dfT03 cores. Target registers on the horizontal axis and failure probability (%) on the vertical axis.

fault rate. The MWTF metric is a widely used reference to assess and compare the reliability of computing systems. It can be calculated as the reciprocal of the average number of application failures, which in turn is given by the product

TABLE 1. Benchmark setup for fT03, T03 and dfT03 architectures. The total execution time is reported in [ms] with a 10-frame division and a fault every 30-40 clock cycles under specific faults per frame. During the test, each potentially ACE bit is fault-injected 10 times the number of Faults per frame.

		fT03		T03		dfT03	
Bench-mark	**Description**	**Execution Time [ms]**	**Faults per frame**	**Execution Time [ms]**	**Faults per frame**	**Execution Time [ms]**	**Faults per frame**
FIR	Finite Impulse Response filter	0.725	194	0.646	173	0.529	141
AES	Advanced Encryption Standard decryption algorithm	1.385	370	1.244	332	0.969	259
FFT	Fast Fourier Transform calculation	1.595	425	1.342	358	1.146	306
SHA	Secure Hash Algorithm calculation	1.792	478	1.589	424	1.268	338
CONV2	2D image convolution	0.257	69	0.216	58	0.197	53
FDCTFST	Fast discrete cosine transform calculation	0.098	26	0.090	24	0.076	21
IPM	Inflexion point method calculation	0.204	55	0.178	48	0.150	40
CRC32	32-bit cyclic redundancy check decoding	0.200	53	0.185	50	0.158	42

of the assumed raw fault rate Rf, the application program execution time T_exec and the architecture vulnerability factor AVF:

$$\mathrm{MWTF} = \frac{1}{Rf \cdot \mathrm{AVF} \cdot T\_exec} \qquad (4)$$

B. RESULTS ON TEST PROGRAMS

We analyzed the performance of the dynamic TMR Klessydra-dfT03 core while running the benchmarks used for the Klessydra-fT03 core verification setup and summarized in Table 1, representing typical computational kernels of embedded applications. As stated before, to evaluate the advantages offered by the proposed technique, we compared the Klessydra-dfT03 core with the unprotected three-thread Klessydra-T03 core. To get a consistent comparison with respect to fault resilience, the program benchmarks are executed in the T03 core on a single thread, keeping the other two threads busy executing NOP instructions (unaffected by faults). This should be considered when reading data on execution time on T03 in Table 1.

Additionally, we also produced a comparison with the fault tolerant Klessydra-fT03 core, which exploits triple temporal redundancy through the Buffered TMR technique [31], [32]. From the total clock cycle count reported in Table 1, it is evident that completing the same program on the Klessydra-dfT03 core requires approximately two-thirds of the total cycles compared to the Klessydra-fT03 core. This efficiency derives from the fact that the dfT03 core operates as a temporal DMR core in the absence of faults. Regarding the T03 core, keeping two threads in a busy waiting state eliminates the occurrence of pipeline stalls, allowing for an execution time slightly faster with respect to the fT03 core.

The results, as depicted in Figure 7, illustrate the probability of failure (Pf) concerning the unprotected T03 core (indicated by red bars) and the protected fT03 and dfT03 cores (represented by dark and light green bars) for all the target registers, each of which contains ACE bits. For registers wider than one bit, we averaged the probability of failure among all the bits.

In the unprotected architecture, the failure rates approach nearly 100% for most registers. When examining the dfT03 bars, it becomes evident that failures have a significant reduction across nearly all registers for all the benchmarks. Notably, failures are completely removed for many registers, even for registers with relatively large bit widths (e.g. 32 bits).

TABLE 2. AVF estimation from (3) for all the analyzed benchmarks, it is possible to see the Architectural Vulnerability Factor improved by an order of magnitude with respect to T03, reflecting the high resilience and low vulnerability of dfT03.

Benchmarks	T03	dfT03	fT03
FIR	0, 5005	0, 0193	0, 0436
AES	0, 5409	0, 0444	0, 0695
FFT	0, 5257	0, 0470	0, 0627
SHA	0, 5372	0, 0473	0, 0719
CONV2	0, 4712	0, 0165	0, 0394
FDCTFST	0, 3640	0, 0157	0, 0251
IPM	0, 4354	0, 0221	0, 0354
CRC32	0, 3396	0, 0061	0, 0102

A limitation of the proposed design is evidenced in the non-zero failure probabilities resulting for some architecture bits. One specific scenario concerns the highest failing ACE register in the protected dfT03 architecture, the *LS_WB* register, responsible for storing the value read from the load-store unit during load operations. Unfortunately, this register cannot be *buffered* to apply the DMR check, as the load operation is performed only once and control signals already buffer the data from the load-store unit. Even though the memory bus can be ECC-protected, ensuring the correctness of the loaded value into *LS_WB*, a fault occurring in the register is actually not corrected by the proposed technique since the bus read operation is assumed to be a non-repeatable operation. Thus, a classic spatial TMR mechanism should be the preferred choice to protect that specific register.

The other registers with non-zero failure probabilities in the dfT03 core are limited to three, including i) the Thread 2 pending interrupt signal, which would lead to unexpected jumps on interrupt routines, ii) the decoded instruction from the ID stage, iii) the Thread 0 PC. Comparing the two redundant architectures, we observe that the dfT03 core allowed additional protection of some internal registers, resulting in reduced fault sensitivity with respect to fT03.

To better quantify the degree of protection, we use the metrics defined in Section V-A: the architectural vulnerability factor (AVF) and the normalized Mean Work to Failure (MWTF). Table 2 presents the AVF results calculated using the data produced by the FI simulations on the three cores. Lower AVF values indicate greater resilience. For all the benchmarks, the resilience of the dfT03 core improves by an order of magnitude compared to the T03 core and several times compared to the fT03 core. Longer benchmarks exhibit

TABLE 3. The trade-off between reliability and performance can be obtained by the Mean Work To Failure estimation (4), considering the analyzed benchmarks under a specific radiation rate. Using the AVF values from Table 2 and Execution time values from Table 1, MWTF still confirms high resilience for the dfT03 core.

Benchmarks	T03	dfT03	fT03
FIR	$3,69E+02$	$3,26E+05$	$4,45E+04$
AES	$1,64E+02$	$3,23E+04$	$9,11E+03$
FFT	$1,61E+02$	$2,41E+04$	$9,66E+03$
SHA	$1,30E+02$	$2,16E+04$	$6,52E+03$
CONV2	$1,24E+03$	$1,23E+06$	$1,54E+05$
FDCTFST	$5,00E+03$	$3,49E+06$	$1,02E+06$
IPM	$1,77E+03$	$8,58E+05$	$2,43E+05$
CRC32	$2,79E+03$	$1,29E+07$	$3,49E+06$

a higher AVF, which can be attributed to the likelihood that a longer benchmark utilizes more hardware registers (thereby exposing them to failure) than a short benchmark. A similar trend is observed in the MWTF metric defined in (4), having as raw fault rate the weighted average of the values visible in Figure 7. Table 3 shows the normalized MWTF with respect to the fault rate for the three cores under consideration.

The advantages of the method with respect to the unprotected T03 core and the triple-redundant fT03 cores are evidenced by two different metrics, AVF and MWTF, with the proposed dfT03 core exhibiting an order of magnitude advantage over T03 and an advantage over FT03 as well.

VI. IMPACT ON HARDWARE RESOURCE UTILIZATION

Table 4 shows the hardware resource utilization of all the architectures synthesized using Xilinx Vivado 2019 on a Genesys2 board based on the Xilinx Kintex-7 FPGA. Observing the reported data, the proposed design exhibits a slight increase in LUTs and FFs compared to the fT03 version due to the thread recovery logic. When comparing Klessydra-dfT03 to the original Klessydra-T03 version, the increased hardware resources are justified by the introduction of fault tolerance features. Regarding dynamic energy consumption, the final value is similar for all the architectures. The Klessydra-dfT03 core exhibits a longer critical path, having to compare the results from different hardware units inside the Restoring Block located in the Program Counter Unit. The proposed dynamic TMR (dTMR) core can reach 185 MHz clock frequency, compared to 200 MHz and 220 MHz for fT03 and T03 cores, respectively.

VII. RESULT COMPARISON WITH THE STATE OF THE ART

We conducted a comparative analysis between the results obtained for the proposed dTMR core and the results reported in the literature for multi-core lockstep architectures, considering both non-intrusive (checkpointing) and intrusive architectures. Table 5 presents a comparison chart listing RISC-V and non-RISC-V architectures, previously introduced in section II. We divide the discussion addressing full mitigation approaches and partial mitigation approaches.

A. FULL MITIGATION APPROACHES

A FT technique that reportedly detects and corrects all injected faults is identified with the attribute *full mitigation*.

In [25], the reported the recovery time overhead, which is 40 cycles, with a 2X area overhead in LUTs and FFs. The authors do not provide information about the quantity of injected faults, yet the tests are limited to the protected units, with FI affecting only the instruction register. Although the work reports full mitigation capabilities, identifying all the injected errors, it does not present numerical data regarding the general fault tolerance performance, nor the checkpoint overhead.

In [17], the execution on the cores is shifted in time so that two out of four cores execute the PC+4 instruction while the other two cores execute the previous PC instruction. With this mechanism, instruction checkpointing may occur with zero overhead, paying in terms of core quadruplication, while the roll-back overhead is not declared. The occupancy consists of 12866 LUTs and 8367 FFs on the FPGA, including the four cores and the checker modules.

Based on shadow registers, the approaches by Nikiema et al. [23] can detect all the injected faults and replace the failed instruction in one clock cycle. No other details are given on the hardware overhead and the impact on the clock frequency, except for reporting that the Partial Shadow Register with Rollback (PSRR) approach requires 160 additional register bits, while the Full Shadow Register (FSR) approach requires 716 additional register bits.

The Hybrid Modular Redundancy (HMR) approaches by Rogenmoser et al. [30] reach full mitigation performance by implementing triple-core lockstep (TCLS) and dual-core lockstep (DCLS) schemes employing multiple cores to execute the same task. The software-based recovery requires 363 clock cycles in the triple core version and implies a 1.3% area overhead for error checking logic over the original 12-core RISC-V cluster. The hardware-based method in the triple core version provides failure recovery in just 24 clock cycles, with a 9.4% area overhead for error checking logic over the original 12 core cluster.

B. PARTIAL MITIGATION

The remaining part of the compared works presents partial mitigation results often linked to specific architectural aspects or benchmark setups.

In [24], which features a heterogeneous dual-core architecture, the authors do not provide information about the achieved fault mitigation extent nor the quantity of injected faults. The approach presents a small overhead in the LowRISC softcore architecture, required for the communication with the ARM core. No data is reported on the checkpoint and rollback time overhead.

In [19] and [20], based on software checkpoint and restore routines, the execution time overhead depends on the chosen checkpointing interval. The authors present a configuration with a checkpoint every 300 write cycles, resulting in a time

TABLE 4. Synthesis results obtained from Xilinx Vivado 2019 on a Genesys2 board (Kintex-7 FPGA).

Core	LUTs	FFs	Energy [pJ/cycle]	Max Freq [MHz]
T03 (non-hardened)	5524	4489	380	220
fT03 (hardened)	6429	4905	390	200
dfT03 (hardened)	6923	5019	390	185

TABLE 5. Performance comparison for relevant cores cited in Section II. Notes: (1) Authors do not report failure rate of unprotected architecture; (2) FI affects only specific units (3) FI affects only instruction register.

Work	FT scheme	FI method	Benchmarks	Hardware overhead	Checkpoint overhead	Rollback overhead	Reported failure mitigation
[25]	DCLS	FI hardware block	Fibonacci	2X FFs, 2X LUTs	Not declared	40 cycles	100% (1)(3)
[24]	DCLS	FI on err. detection module	Ad-hoc benchmark	1.04X FF 1.01X LUT, + 1 ARM core	Not declared	SW dependent (not reported)	Not available
[19]	DCLS	FI hardware block	FIR, Ellipf, Kalman, MatMul	2X BRAMs, 2.10X LUTs	17% to 54%	10M cycles	84%
[20]	TCLS	FI hardware block	MatMul	3.44X FFs, 3.75X LUTs	59% to 81%	Not declared	50% to 70%
[17]	TCLS	FI on FPGA config mem	Dijkstra, Quicksort, Correlation, MatMul	4.01X FFs, 4.54X LUTs	0 cycles	Not declared	100%
[22]	DCLS	FI simulation	TACLeBench	2.02X FFs, 2.01X LUTs	Not declared	Not declared	90% to 99% (2)
[23]	DCLS	FI simulation	Matmult, Qsort, Gaussian Filter, Moving Average	logic not declared + 1 core	0 cycles	2-5 cycles	100%
[28]	DCLS	FI hardware block	Fib	2.19X FFs, 1.23X LUTs + 1 core	3128 cycles	2852 cycles	96,68%
[28]	DCLS	FI hardware block	Fib	1.11X FFs, 1.07X LUTs + 1 core	335 cycles	248 cycles	95,70%
[29]	TCLS	FI hardware block	MatMul	15.81X FFs, 19.48X LUTs + 1 ARM core	12.1% to 62.6%	Not declared	98%
[30]	TCLS	FI simulation	MatMul	1.013X to 1.093X logic area + 2 cores	0 cycles	363 - 24 cycles	100%
[27]	TCLS	FI simulation	canrdr01, ttsprk01, rspeed01, puwmod01, tblook01, matrix01, aifirf01	1.08X to 1.33X logic area + 2 ARM cores	1171 cycles	1180 cycles	85.28% to 94.4%
This work	dTMR	FI simulation	FFT, Fir, Sha, Aes-Cbc, Crc32, Conv2D, Ipm, Fdctfst	1.11X FFs, 1.25X LUTs	0 cycles	4 cycles	96,8% to 98,6%

overhead that varies from 17% for a FIR benchmark to 54% for a Kalman filter benchmark. The reported mitigation is 84% of the failures induced by the injected faults.

Similarly, in [20], the authors compared a matrix multiplication benchmark executed on a bare-metal environment and on a FreeRTOS environment, demonstrating a checkpointing overhead ranging from 59% for the bare-metal case to 81% for the FreeRTOS case. Regarding the achieved fault mitigation, the authors observed a range from 60% to 70% for the bare-metal environment and a range from 50% to 60% for the FreeRTOS environment.

In the DCLS SafeDE approach [22], the reported FPGA utilization reaches 76K LUTs and 34K FFs, despite the SafeDE block itself being relatively small (261 LUTs and 417 FFs). The solution inserts a maximum delay of 20 cycles between the main and follower cores. The authors do not declare the temporal overheads due to checkpointing and roll-back. The FI analysis was carried out on one benchmark (factorial and accumulation) using stuck-at and bit-flip faults for the duration of 1 cycle and 10 cycles, with a failure mitigation level that varies from 90.25% to 99%.

Authors in [28] implement two heterogeneous DCLS architectures (64-bit and 32-bit, respectively), obtaining 96.68% and 95.70% fault mitigation, respectively. They declare that checkpoint saving takes 3128 clock cycles in the Lock-VA architecture and 335 cycles in the Lock-VM architecture. The restore procedure via rollback takes 2852 and 248 cycles in Lock-VA and Lock-VM, respectively. From the hardware cost point of view, considering only the softcore part, the overhead is 1.23X LUTs and 2.19X FFs for the Lock-VA, being 1.07X LUT and 1.11X FF for the Lock-VM.

The work in [29] presents a TCLS system reported to achieve 98% fault mitigation. Timing efficiency in the design is achieved when the execution time of the useful computation (e.g. matrix multiplication) consists of a large fraction over the total block execution time considering the other operations (i.e. consistency check and checkpoint) inside a verification point [29], moving from 12.1% for 60 $\times$ 60 MatMul and 62.6% for 10 $\times$ 10 MatMul. From the hardware perspective, the whole core occupies around 23.7K LUTs and 18.4K FFs, counting the checker logic and the TMR MicroBlaze module.

In the triple-core lockstep processor by Iturbe et al. [27], the instruction output is compared every clock cycle. The context-saving routine that generates the checkpoint takes 1171 clock cycles, while the restore routine takes 1180 cycles. Considering just the three cores, the logic area overhead varies from 1.27x to 1.33x, going down to 1.06x - 1.08x when including the shared memory system. On the total injected faults per benchmark we report the soft error results, although the authors have also tried other types of fault injection. The injected soft errors produce up to 5.6% of failures in the TCLS output, with up to 14.72% latent errors, ranging from 85.28% to 94.4% fault mitigation.

C. FINAL COMPARISON

When considering fault tolerance, directly comparing different FT cores may not bring meaningful insights unless they undergo the same fault injection tests. Nonetheless, it is possible to assess the effectiveness of a protection technique by evaluating the percentage of mitigated faults with respect to the total injected faults.

In the present work, by averaging the results in Figure 7, the error mitigation rate ranges between 96.8% and 98.6% for the ACE bits in the architecture on the tested benchmarks. The values are higher or comparable to most of the works in Table 5. Moreover, these values are reported (as well as the FI results in 7 without considering extra FT measures (e.g. TMR on specific registers) suggested by the analysis of the results.

In the presence of faults, the restore routine additionally affects the program execution time with dependence on fault rate, making direct comparisons more difficult [9]. The common recovery time for the dfT03 architecture is four clock cycles, as the pipeline is flushed and restarted (memory instructions normally lead to higher latency but are less frequent). The recovery time is then smaller than any other technique using software rollback routines since many described approaches require the restoration of the entire context, and only some of them, like the works in [23], designed with fine-grained intrusive approach, introduce less or comparable overhead.

From the checkpoint overhead, our work does not introduce any particular penalty like [19], [20], [27], [28], and [29], having the checkpoint automatically saved every two clock cycles by the dynamic TMR approach, with the last correct program counter saved in a dedicated register for each executed instruction. Leaving aside the works [22] and [25], in which FI analyses are performed only on specific architectural units, our work reaches 98.6% in fault mitigation. It is worth noting that full mitigation techniques as [17], [23], and [30] reach 100% mitigation at the cost of hardware complexity. In particular, the works in [17] and [30] use 4 cores and 3-12 cores, respectively. The work in [23] uses a dual-core lockstep technique and intrusive hardware customization, which allows the detection of faults directly in the pipeline and not only at the output interface, thus ensuring greater coverage.

Generally speaking, our approach shows better or comparable results in terms of SEU fault tolerance to many works in the literature, guaranteeing better hardware occupations with just 1.11X in FFs and 1.25X in LUTs, limiting to a minimum temporal overhead due to the checkpoint and restore processes, even compared to intrusive designs, which implement these procedures purely in hardware.

The advantages of the method with respect to existing dual- and triple-core lockstep methods are evidenced by reduced hardware cost and/or drastically reduced time overhead for Checkpointing and Rollback, along with superior or equal failure mitigation rate.

The shortcomings of the approach are those of any time-redundancy method that partially trades off performance for safety, thus making it suitable in systems where safety and hardware cost are the primary concerns.

Looking at computational complexity, the intrinsic double redundancy of the proposed method implies doubling the computational load, like in dual-core lockstep architectures. In dual-core solutions, the two cores execute two identical threads, as in the proposed method the IMT core executes two identical threads. When considering triple-core lockstep solutions, the computational load is triplicated, and so is the hardware cost. It is true that dual- and triple-core solutions translate computational complexity in increased hardware cost by implementing spatial redundancy, while the proposed solution translates computational complexity in increased execution time by implementing temporal redundancy. Yet, the additional computational complexity due to Checkpoint overhead and Rollback overhead is reduced with respect to conventional dual-core lockstep, as resulting from Table 5, so that the overall average computational complexity of the proposed solution is lower than in the compared previous methods.

VIII. CONCLUSION

The proposed work extensively investigated the design, implementation and fault tolerance evaluation of the dynamic TMR (dTMR) approach in an interleaved multi-threading RISC-V compliant core, in comparison of more conventional lockstep multi-core schemes. The FI campaign, based on a deterministic time-frame-span approach that covered all the register bits in the microarchitecture, produced a detailed quantitative evaluation of the achieved failure mitigation rate on the execution of real-world benchmark routines.

The investigated dTMR core design exhibited an average failure mitigation rate of up to 98.6%, with negligible performance penalty in case of fault detection. Compared to an extensive list of lockstep multi-core architectures, both DCLS and TCLS, the dTMR implementation trades off speed (because of time redundancy) for fault tolerance, while existing lockstep schemes trade-off hardware area for fault tolerance. The proposed solution has less than 25% hardware overhead over the corresponding non-protected architecture and 2X execution time penalty because of time redundancy introduced by thread replication. Lockstep solutions always imply at least a 2X hardware overhead, and in most cases they pay a time penalty for checkpointing and for rollback in case of fault. Overall, the dTMR scheme in multi-threading cores results in representing a preferable choice whenever time redundancy is more convenient than space redundancy, with the advantage of practically zero penalty in case of fault-detection.

REFERENCES

[1] Z. Abbas, M. Olivieri, U. Khalid, A. Ripp, and M. Pronath, "Optimal NBTI degradation and PVT variation resistant device sizing in a full adder cell," in *Proc. 4th Int. Conf. Rel., INFOCOM Technol. Optim.*, Sep. 2015, pp. 1–6.

[2] A. Buzzin, A. Rossi, E. Giovine, G. de Cesare, and N. P. Belfiore, "Downsizing effects on micro and nano comb drives," *Actuators*, vol. 11, no. 3, p. 71, Feb. 2022.

[3] L. Blasi, F. Vigli, A. Cheikh, A. Mastrandrea, F. Menichelli, and M. Olivieri, "A RISC-V fault-tolerant microcontroller core architecture based on a hardware thread full/partial protection and a thread-controlled watch-dog timer," in *Proc. Int. Conf. Appl. Electron. Pervading Ind., Environ. Soc.*, 2019, pp. 505–511.

[4] M. Barbirotta, A. Cheikh, A. Mastrandrea, F. Menichelli, and M. Olivieri, "Analysis of a fault tolerant edge-computing microarchitecture exploiting vector acceleration," in *Proc. 17th Conf. Ph.D. Res. Microelectron. Electron. (PRIME)*, Jun. 2022, pp. 237–240.

[5] J. Han, X. Liu, X. Wei, and X. Zhu, "Reduced-order observer-based finite time fault estimation for switched systems with lager and fast time varying fault," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 71, no. 1, pp. 350–354, Jan. 2024.

[6] J. Han, X. Liu, X. Xie, and X. Wei, "Dynamic output feedback fault tolerant control for switched fuzzy systems with fast time varying and unbounded faults," *IEEE Trans. Fuzzy Syst.*, vol. 31, no. 9, pp. 3185–3196, 2023, doi: 10.1109/TFUZZ.2023.3246061.

[7] J. Dongarra, T. Herault, and Y. Robert, *Fault Tolerance Techniques for High-Performance Computing*. Cham, Switzerland: Springer, 2015.

[8] E. W. Wächter, S. Kasap, X. Zhai, S. Ehsan, and K. McDonald-Maier, "Survey of lockstep based mitigation techniques for soft errors in embedded systems," in *Proc. 11th Comput. Sci. Electron. Eng. (CEEC)*, Sep. 2019, pp. 124–127.

[9] M. Barbirotta, A. Cheikh, A. Mastrandrea, F. Menichelli, M. Ottavi, and M. Olivieri, "Evaluation of dynamic triple modular redundancy in an interleaved-multi-threading RISC-V core," *J. Low Power Electron. Appl.*, vol. 13, no. 1, p. 2, Dec. 2022.

[10] S. Nakagawa, S. Fukumoto, and N. Ishii, "Optimal checkpointing intervals of three error detection schemes by a double modular redundancy," *Math. Comput. Model.*, vol. 38, nos. 11–13, pp. 1357–1363, Dec. 2003.

[11] P. Reviriego, C. J. Bleakley, and J. A. Maestro, "Diverse double modular redundancy: A new direction for soft-error detection and correction," *IEEE Des. Test. IEEE Des. Test. Comput.*, vol. 30, no. 2, pp. 87–95, Apr. 2013.

[12] G. Popov, M. Nenova, and K. Raynova, "Reliability investigation of TMR and DMR systems with global and partial reservation," in *Proc. 7th Balkan Conf. Lighting*, Sep. 2018, pp. 1–4.

[13] Y. Li, Y. Li, H. Jie, J. Hu, F. Yang, X. Zeng, B. Cockburn, and J. Chen, "Feedback-based low-power soft-error-tolerant design for dual-modular redundancy," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 26, no. 8, pp. 1585–1589, Aug. 2018.

[14] R. Vadlamani, J. Zhao, W. Burleson, and R. Tessier, "Multicore soft error rate stabilization using adaptive dual modular redundancy," in *Proc. Design, Autom. Test Eur. Conf. Exhibition*, Mar. 2010, pp. 27–32.

[15] S. Nomura, M. D. Sinclair, C.-H. Ho, V. Govindaraju, M. De Kruijf, and K. Sankaralingam, "Sampling+ DMR: Practical and low-overhead permanent fault detection," *ACM SIGARCH Comput. Archit. News*, vol. 39, no. 3, pp. 201–212, 2011.

[16] I. B. M. Matsuo, L. Zhao, and W.-J. Lee, "A dual modular redundancy scheme for CPU–FPGA platform-based systems," *IEEE Trans. Ind. Appl.*, vol. 54, no. 6, pp. 5621–5629, Nov. 2018.

[17] S. Shukla and K. C. Ray, "A low-overhead reconfigurable RISC-V quad-core processor architecture for fault-tolerant applications," *IEEE Access*, vol. 10, pp. 44136–44146, 2022.

[18] F. Abate, L. Sterpone, and M. Violante, "A new mitigation approach for soft errors in embedded processors," *IEEE Trans. Nucl. Sci.*, vol. 55, no. 4, pp. 2063–2069, Aug. 2008.

[19] M. Violante, C. Meinhardt, R. Reis, and M. Sonza Reorda, "A low-cost solution for deploying processor cores in harsh environments," *IEEE Trans. Ind. Electron.*, vol. 58, no. 7, pp. 2617–2626, Jul. 2011.

[20] 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 *Proc. 30th Symp. Integr. Circuits Syst. Design (SBCCI)*, Aug. 2017, pp. 84–89.

[21] M. Sarraseca, S. Alcaide, F. Fuentes, J. C. Rodriguez, F. Chang, I. Lasfar, R. Canal, F. J. Cazorla, and J. Abella, "SafeLS: An open source implementation of a lockstep NOEL-V RISC-V core," in *Proc. IEEE 29th Int. Symp. On-Line Test. Robust Syst. Design (IOLTS)*, Jul. 2023, pp. 1–7.

[22] F. Bas, S. Alcaide, G. Cabo, P. Benedicte, and J. Abella, "SafeDE: A low-cost hardware solution to enforce diverse redundancy in multicores," *IEEE Trans. Device Mater. Rel.*, vol. 22, no. 2, pp. 111–119, Jun. 2022.

[23] P. R. Nikiema, A. Kritikakou, M. Traiola, and O. Sentieys, "Design with low complexity fine-grained dual core lock-step (DCLS) RISC-V processors," in *Proc. 53rd Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. Supplemental Volume (DSN-S)*, Jun. 2023, pp. 224–229.

[24] C. Rodrigues, I. Marques, S. Pinto, T. Gomes, and A. Tavares, "Towards a heterogeneous fault-tolerance architecture based on arm and RISC-V processors," in *Proc. 45th Annu. Conf. IEEE Ind. Electron. Soc.*, vol. 1, Oct. 2019, pp. 3112–3117.

[25] I. Silva, O. do Espírito Santo, D. do Nascimento, and S. Xavier-de-Souza, "CEVERO: A soft-error hardened SoC for aerospace applications," in *Anais Estendidos Simpósio Brasileiro De Engenharia De Sistemas Computacionais*. Porto Alegre, Brasil: SBC, 2020, pp. 121–126, doi: 10.5753/sbesc_estendido.2020.13100. [Online]. Available: https://sol.sbc.org.br/index.php/sbesc_estendido/article/view/13100

[26] J. Yiu, "Design of SOC for high reliability systems with embedded processors," in *Proc. Embedded World Conf.*, 2015, pp. 1–12.

[27] X. Iturbe, B. Venu, E. Ozer, J.-L. Poupat, G. Gimenez, and H.-U. Zurek, "The arm triple core lock-step (TCLS) processor," *ACM Trans. Comput. Syst.*, vol. 36, no. 3, pp. 1–30, Aug. 2018.

[28] I. Marques, C. Rodrigues, A. Tavares, S. Pinto, and T. Gomes, "Lock-V: A heterogeneous fault tolerance architecture based on arm and RISC-V," *Microelectron. Rel.*, vol. 120, May 2021, Art. no. 114120.

[29] S. Kasap, E. W. Wächter, X. Zhai, S. Ehsan, and K. D. McDonald-Maier, "Novel lockstep-based fault mitigation approach for SoCs with roll-back and roll-forward recovery," *Microelectron. Rel.*, vol. 124, Sep. 2021, Art. no. 114297.

[30] M. Rogenmoser, Y. Tortorella, D. Rossi, F. Conti, and L. Benini, "Hybrid modular redundancy: Exploring modular redundancy approaches in RISC-V multi-core computing clusters for reliable processing in space," 2023, *arXiv:2303.08706*.

[31] M. Barbirotta, A. Cheikh, A. Mastrandrea, F. Menichelli, F. Vigli, and M. Olivieri, "A fault tolerant soft-core obtained from an interleaved-multi- threading RISC-V microprocessor design," in *Proc. IEEE Int. Symp. Defect Fault Tolerance VLSI Nanotechnol. Syst. (DFT)*, Oct. 2021, pp. 1–4.

[32] M. Barbirotta, A. Cheikh, A. Mastrandrea, F. Menichelli, and M. Olivieri, "Design and evaluation of buffered triple modular redundancy in interleaved-multi-threading processors," *IEEE Access*, vol. 10, pp. 126074–126088, 2022.

[33] C. Carmichael, E. Fuller, J. Fabula, and F. Lima, "Proton testing of SEU mitigation methods for the virtex FPGA," in *Proc. Mil. Aerosp. Appl. Program. Log. Devices MAPLD*, 2001, pp. 1–12.

[34] A. Cheikh, S. Sordillo, A. Mastrandrea, F. Menichelli, and M. Olivieri, "Efficient mathematic accelerator design coupled with an imt RISC-V microprocessor," in *Applepies*, vol. 627. Springer, 2019, pp. 505–511.

[35] M. Barbirotta, A. Mastrandrea, F. Menichelli, F. Vigli, L. Blasi, A. Cheikh, S. Sordillo, F. Di Gennaro, and M. Olivieri, "Fault resilience analysis of a RISC-V microprocessor design through a dedicated UVM environment," in *Proc. IEEE Int. Symp. Defect Fault Tolerance VLSI Nanotechnol. Syst. (DFT)*, Oct. 2020, pp. 1–6.

[36] A. Cheikh, G. Cerutti, A. Mastrandrea, F. Menichelli, and M. Olivieri, "The microarchitecture of a multi-threaded RISC-V compliant processing core family for IoT end-nodes," in *Applications in Electronics Pervading Industry, Environment and Society*. Cham, Switzerland: Springer, 2019, pp. 89–97.

[37] N. J. George, C. R. Elks, B. W. Johnson, and J. Lach, "Transient fault models and AVF estimation revisited," in *Proc. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Jun. 2010, pp. 477–486.
[38] L. Osinski, T. Langer, and J. Mottok, "A survey of fault tolerance approaches on different architecture levels," in *Proc. 30th Int. Conf. Archit. Comput. Syst.*, Apr. 2017, pp. 1–9.
[39] I. Oz and S. Arslan, "A survey on multithreading alternatives for soft error fault tolerance," *ACM Comput. Surveys*, vol. 52, no. 2, pp. 1–38, Mar. 2020.
[40] G. Abich, J. Gava, R. Garibotti, R. Reis, and L. Ost, "Applying lightweight soft error mitigation techniques to embedded mixed precision deep neural networks," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 68, no. 11, pp. 4772–4782, Nov. 2021.
[41] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, D. I. August, and S. S. Mukherjee, "Design and evaluation of hybrid fault-detection systems," in *Proc. 32nd Int. Symp. Comput. Archit.*, 2005, pp. 148–159.

MARCELLO BARBIROTTA received the M.S. degree (cum laude) in electronics engineering from Sapienza University of Rome, Rome, Italy, in 2020, where he is currently pursuing the Ph.D. degree with the Department of Information Engineering, Electronics and Telecommunications. His current research interests include analysis techniques and models for fault resilience within digital microprocessor architectures, targeting RISC-V cores, and microcontrollers.

FRANCESCO MENICHELLI received the M.S. degree (cum laude) in electronic engineering, in 2001, and the Ph.D. degree in electronic engineering from the University of Rome "La Sapienza," in 2005. He is currently an Assistant Professor with Sapienza University of Rome, Italy. He has co-authored more than 40 publications in international journals and conference proceedings. His scientific interests focus on low-power digital design and, in particular, system-level and architectural-level techniques for low-power consumption, power modeling, and simulation of digital system platforms.

ABDALLAH CHEIKH received the B.S. and M.S. degrees in electrical engineering from Rafik Hariri University, Lebanon, in 2014 and 2016, respectively, and the Ph.D. degree majoring in computer architecture from Sapienza University of Rome, Italy, in 2020, with a focus on computer organization and design. He is currently a Postdoctoral Researcher with Sapienza University of Rome. His research interests include designing, implementing, and verifying a wide range of microprocessor architectures, vector accelerators, and morphing processors.

ANTONIO MASTRANDREA received the M.S. (Laurea) degree (cum laude) in electronics engineering and the Ph.D. degree from Sapienza University of Rome, in 2010 and 2014, respectively. He is currently a Research Assistant with the Department of Information Engineering, Electronics and Telecommunications, Sapienza University of Rome, Italy. His current research interests include digital system-on-chip architectures and nano-CMOS circuits oriented toward high-speed computation.

MARCO ANGIOLI (Graduate Student Member, IEEE) received the M.S. degree (cum laude) in electronics engineering, in 2022. He is currently pursuing the Ph.D. degree with the Department of Information Engineering, Electronics and Telecommunications, Sapienza University of Rome, Rome, Italy. His current research interests include the digital design of hardware accelerators for the implementation and execution of artificial intelligence algorithms on low-power embedded systems with stringent real-time requirements.

MAURO OLIVIERI (Senior Member, IEEE) received the M.S. (Laurea) degree (cum laude) in 1991, and the Ph.D. degree in electronics and computer engineering from the University of Genoa, Italy, in 1994. He was an Assistant Professor with the University of Genoa, from 1995 to 1998. He joined Sapienza University of Rome, Italy, as an Associate Professor, in 1998. He is currently a Visiting Researcher with Barcelona Supercomputing Center in the European Processor Initiative Project. He authored more than 100 articles and a textbook in three volumes on digital VLSI design. His research interests include microprocessor core designs and digital nanoscale circuits. He has been a member of TPC of IEEE Date and was the General Co-Chair of IEEE/ACM ISLPED'15. He is an Evaluator of European Commission in the ECSEL Joint Undertaking.

Open Access funding provided by 'Università degli Studi di Roma "La Sapienza" 2' within the CRUI CARE Agreement