Hybrid Modular Redundancy: Exploring Modular Redundancy Approaches in RISC-V Multi-core Computing Clusters for Reliable Processing in Space

MICHAEL ROGENMOSER, ETH Zürich, Switzerland
YVAN TORTORELLA, DAVIDE ROSSI, and FRANCESCO CONTI, Università di Bologna, Italy
LUCA BENINI, ETH Zürich, Switzerland and Università di Bologna, Italy

Space Cyber-Physical Systems such as spacecraft and satellites strongly rely on the reliability of onboard computers to guarantee the success of their missions. Relying solely on radiation-hardened technologies is extremely expensive, and developing inflexible architectural and microarchitectural modifications to introduce modular redundancy within a system leads to significant area increase and performance degradation. To mitigate the overheads of traditional radiation hardening and modular redundancy approaches, we present a novel Hybrid Modular Redundancy approach, a redundancy scheme that features a cluster of RISC-V processors with a flexible on-demand dual-core and triple-core lockstep grouping of computing cores with runtime split-lock capabilities. Further, we propose two recovery approaches, software-based and hardware-based, trading off performance and area overhead. Running at 430 MHz, our fault-tolerant cluster achieves up to 1,160 MOPS on a matrix multiplication benchmark when configured in non-redundant mode and 617 and 414 MOPS in dual and triple mode, respectively. A software-based recovery in triple mode requires 363 clock cycles and occupies 0.612 mm^2, representing a 1.3% area overhead over a non-redundant 12-core RISC-V cluster. As a high-performance alternative, a new hardware-based method provides rapid fault recovery in just 24 clock cycles and occupies 0.660 mm^2, namely, ~9.4% area overhead over the baseline non-redundant RISC-V cluster. The cluster is also enhanced with split-lock capabilities to enter one of the available redundant modes with minimum performance loss, allowing execution of a mission-critical portion of code when in independent mode, or a performance section when in a reliability mode, with <400 clock cycles overhead for entry and exit. The proposed system is the first to integrate these functionalities on an open-source RISC-V-based compute device, enabling finely tunable reliability versus performance trade-offs.

M. Rogenmoser and Y. Tortorella contributed equally to this research.

The presented work was supported by Thales Alenia Space.
We acknowledge support by the EU H2020 Fractal project funded by ECSEL-JU Grant Agreement No. 877056. We acknowledge support by TRISTAN, which has received funding from the Key Digital Technologies Joint Undertaking (KDT JU) under Grant Agreement No. 101095947. The KDT JU receives support from the European Union's Horizon Europe's research and innovation programmes.
The authors thank Luca Rufer for his valuable contribution to the research project.

Authors' addresses: M. Rogenmoser, Integrated Systems Laboratory, ETH Zürich, Gloriastrasse 35, 8092 Zürich, Switzerland; e-mail: michaero@iis.ee.ethz.ch; Y. Tortorella, D. Rossi, and F. Conti, Electrical, Electronic, and Information Engineering Department, University of Bologna, Viale Risorgimento 2, 40126 Bologna, Italy; e-mails: {yvan.tortorella, davide.rossi, f.conti}@unibo.it; L. Benini, Integrated Systems Laboratory, ETH Zürich, Gloriastrasse 35, 8092 Zürich, Switzerland and Electrical, Electronic, and Information Engineering Department, University of Bologna, Viale Risorgimento 2, 40126 Bologna, Italy; e-mail: lbenini@iis.ee.ethz.ch.

2378-962X/2025/01-ART8 $15.00
https://doi.org/10.1145/3635161

CCS Concepts: • **Hardware → Fault tolerance**; • **Computer systems organization → Reliability**; **Multicore architectures;**

Additional Key Words and Phrases: RISC-V, adaptive fault tolerance, space vehicle computer, reliable computing

ACM Reference format:
Michael Rogenmoser, Yvan Tortorella, Davide Rossi, Francesco Conti, and Luca Benini. 2025. Hybrid Modular Redundancy: Exploring Modular Redundancy Approaches in RISC-V Multi-core Computing Clusters for Reliable Processing in Space. *ACM Trans. Cyber-Phys. Syst.* 9, 1, Article 8 (January 2025), 29 pages.
https://doi.org/10.1145/3635161

1 INTRODUCTION

Cyber-Physical Systems (CPSs) integrate real-time computing and communication capabilities with monitoring and control actions over components in the physical world [47]. To face the harshness of the space environment, modern space systems such as satellites and spacecraft require tight coupling between onboard processing, communication (cyber), sensing, and actuation (physical) elements [35]. The orbit determination and control subsystems on a small spacecraft or in satellites' constellations provide a clear link between onboard processing and sensing elements of the spacecraft's physical environment [16], becoming increasingly critical as small spacecraft become ever more capable [35]. Radiation-induced soft errors such as **Single Event Transients (SETs)** and **Single Event Upsets (SEUs)** can occur more frequently in space than at ground level, creating the need for additional hardware to mitigate detrimental effects on the system [52]. In this scenario, digital computing systems representing the decisional part of a **Space Cyber-Physical System (S-CPS)** must be designed to be reliable and tolerate faults induced by cosmic radiation.

Various solutions exist to protect electronics from the adverse effects of radiation [52]. S-CPSs typically rely on **Radiation Hardening by Design (RHBD)** techniques to add reliability at the technology level, yet scaling of these approaches lags behind the scaling of their commercial counterparts, significantly impacting **performance, power, and area (PPA)** of these designs. Insulating techniques [2], and polymer shielding [46] can also help mitigate soft errors. Furthermore, it is also possible to enhance the fault tolerance capabilities of digital systems by introducing redundancy at different levels in their design flow. Temporal redundancy techniques rely on repeated executions of the same work to determine the correct result [19]. Spatial or modular redundancy techniques rely on multiple hardware blocks executing the same task and comparing the results [23]. These approaches rely on rigid schemes for repetition in space and time of redundant blocks or tasks. Hence, they can severely impact the PPA of computing platforms. This leads to a significant gap between **System on Chips (SoCs)** for space and their commercial counterparts, which make use of technological advances unencumbered by radiation-induced faults.

Therefore, the increasing demand for strong processing capabilities in space [57] is pushing researchers toward lower-overhead solutions and use of more advanced technologies to close this gap in PPA. In recent years, the advent of RISC-V and open-source hardware has encouraged the development of high-performance SoCs for various domains without licensing or other restrictions. This includes the space domain, where custom modifications to improve properties such as reliability [17] and fault tolerance are often required. Among proposed architectures, heterogeneous systems with multi-core computing clusters have gained traction in the space industry [24] due to increased performance and flexibility for computation and **Digital Signal Processing (DSP)** workloads [36]. While multiple processors offer increased performance for parallelizable tasks, they also provide a unique opportunity for reliability enhancements: multiple cores can execute identical tasks, comparing their results to detect and react to faults.

In this article, we introduce a multi-core RISC-V-based computing system for space featuring an innovative **Hybrid Modular Redundancy (HMR)** approach. We leverage the independent cores available in a multi-core RISC-V cluster built on a commercial technology for redundant execution in a dynamically runtime configurable manner, supporting on-demand reconfiguration in **Dual-Core Lockstep (DCLS)** and **Triple-Core Lockstep (TCLS)** modes. This extends the **On-Demand Redundancy Grouping (ODRG)** presented in Reference [44], which presents a TCLS configurable cluster with software-based recovery and configuration prior to startup. Our design extends this by allowing each application to configure its reliability setting according to its requirements, possibly decided at runtime. Furthermore, we implement two recovery alternatives, software and hardware-assisted, comparing their impact on the hardware resources and performance in case of a fault. The checking, voting, and switching hardware in the implemented design does not affect the internals of the processor core, allowing for the use of verified RISC-V processor cores without requiring any internal (potentially erroneous) modifications to rapidly build a reliable system. What we propose is, to the best of our knowledge, the most compact, flexible, and configurable computing cluster offering the best trade-off between performance and reliability.

Besides extending our previous work [44], we introduce the following novel contributions:

— A re-configurable computing cluster for Dual-Core Lockstep and Triple-Core Lockstep execution capable of tackling compute-intensive and safety-critical applications at the same time. The proposed cluster can be configured so that the computing cores can operate independently during high-performance execution or in Dual/Triple-Core Lockstep mode in case of safety-critical tasks.
— Robust hardware extension for fast fault recovery. We introduced dedicated Error-Correcting Codes-protected status registers to restore the state of the computing cores to the closest reliable state in time. This feature allows the cores to perform cycle-by-cycle backups of their internal state in the protected registers, reducing by 15× the required time to recover from a fault over the software-based approach.
— A novel runtime-programmable split-lock mechanism, allowing for fast switching and reconfiguration between the available redundant modes. With these features, it is possible to explicitly define portions of code within a *mission-critical section*, configuring the cores for safe lockstep execution, or within a *performance section*, disabling the lockstep execution temporarily, with minimum configuration switching overhead.

To validate our proposed approach, we implemented the RISC-V cluster in Global Foundries 22 nm technology, achieving up to 430 MHz operating frequency and 1,160 MOPS when configured in independent mode and 617 and 414 MOPS in **Double Modular Redundancy (DMR)** and **Triple Modular Redundancy (TMR)** mode, respectively. With only software-based recovery features, the proposed cluster occupies 0.612 mm^2 with just 1.3% area overhead over the non-redundant configuration, featuring 363 clock cycles time-to-recovery in triple mode. When enhanced with hardware-based recovery features, it provides rapid fault recovery in just 24 clock cycles occupying 0.660 mm^2, ~9.4% area overhead over the baseline RISC-V cluster. The proposed split-lock mechanism allows for entering and exiting a redundancy mode in <400 clock cycles for mission-critical code execution, or <200 exiting and re-entering the redundancy mode for temporarily more performance. To foster future research in computer architecture for space, we release the proposed architecture as the first fully open-source[1] RISC-V-based multi-core cluster with a finely tunable trade-off between reliability and performance.

[1] https://github.com/pulp-platform/redundancy_cells

2 RELATED WORK

2.1 Design Trends for CPS in Space

In recent years, S-CPS and onboard processors in space have been striving for more performance, fueled by more advanced mission requirements and higher expectations for onboard electronics resulting from the advancement of commercial technology [21, 43]. However, reliability and dependability [11, 57] remain key concerns in the space domain, because, with high levels of cosmic rays, errors are more frequent than at ground level and occur too often for a system to perform its intended mission reliably.

To achieve higher levels of performance, the industry at large has steered towards multi-core processing architectures, leveraging multiple parallel processing cores to increase performance. Players in space computing have adopted this approach: systems such as the BAE RAD5500 architecture [10], the Gaisler GR740 architecture [29], and the DAHLIA NG-ultra architecture [14] leverage quad-core architectures built on existing, radiation-hardened processes. Other designs, like Ramon Chip's RC64 [24], leverage large processor arrays for DSP calculations to improve performance. However, these systems, built on PowerPC, SPARC, ARM, and a custom **Instruction Set Architecture (ISA)**, respectively, suffer from high costs due to specialized and limited production and lack of commercial technology and software support, further increasing their cost [17]. Otherwise, the reliability and dependability of these designs built on **Radiation-Hardened (Rad-Hard)** technology increase the chances of mission success, making them the go-to choice for complex, high-cost space missions.

However, not all missions in the space domain require these high tolerance levels. New Space missions with smaller, more cost-effective satellites tend to rely more on **Commercial Off-the-Shelf (COTS)** electronics to design the spacecraft [50]. These components offer significantly more performance at a lower cost, sacrificing reliability, which can be tolerated for non-critical applications, such as in CubeSats [50] or for non-critical machine learning workloads [21]. However, even for these satellite missions, certain aspects, such as communication and control, still require some radiation tolerance in COTS solutions, often guaranteed with watchdog timers.

However, large-scale infrastructure may have satellites as one part of the pipeline but can contain many different components, from IoT sensor nodes to high-performance server infrastructure. In such cases, using many different components with different programming paradigms can lead to untenable cost overheads, where component reuse can greatly benefit the designers. In such a case, reliability may be required by the satellite but is counter-productive for the power-constrained sensor node requiring high performance and energy efficiency. Furthermore, reusing a single component both for commercial and reliability environments can greatly reduce component cost, leveraging scale to combat the cost issue plaguing the space industry.

One of the recent trends in custom SoC design is the shift to RISC-V, leveraging a modern and open-source ISA to develop custom hardware. RISC-V is also quickly impacting the space domain [17, 20], which relies on modified custom designs to ensure proper reliability and fault tolerance. In using this new ISA, the space industry can benefit not only from designing custom, reliable hardware for their purposes but also from the commercial environment. Companies developing dedicated hardware for space have also adopted this approach: Gaisler is developing the NOEL-V architecture [3], Microchip is leveraging SiFive's processor architectures for NASA's next High Performance Space Computing project [37], and dedicated projects like De-RISC [55] are pushing this development. In this work, we continue on this trend by exploiting an open-source industrially verified RISC-V core [22, 27] as a building block for our system.

Table 1. Qualitative Comparison of Related Work with Corresponding Features and Configuration

	System	ISA	Cores	Reliability Method	Configurable	Development
Rad-Hard	Gaisler GR740 [29]	SPARC	4	Rad-Hard & Error-Correcting Codes (ECC)	✗	Commercial
	BAE RAD5500 [10]	PowerPC	4	Rad-Hard	✗	Commercial
	DAHLIA [14]	ARM	4	Rad-Hard	✗	Commercial
	Ramon Chips RC64 [24]	CEVA X DSP	64	Rad-Hard	✗	Commercial
Arch.	Duck-Core [38]	RISC-V	1	ECC	✗	Research
	STRV [53]	RISC-V	1	gate-level TMR	✗	Research
	Gkiokas and Schoeberl [25]	RISC	1(3)	block-level TMR	✗	Research
	SHAKTI-F [28]	RISC-V	1	block-level DMR & ECC	✗	Research
System-level	ARM DCLS [5, 31]	ARM	2	DCLS	~	Commercial
	ARM TCLS [32]	ARM	3	TCLS	~	Research
	AURIX TriCore [30]	TriCore	3	TCLS	~	Commercial
	CEVERO [49]	RISC-V	2	DCLS	✗	Research
	Shukla and Ray [48]	RISC-V	4	DCLS	✓	Research
	Kempf et al. [34]	SPARC	2	DCLS & ECC	✓	Research
	SafeDE [8]	RISC-V	2	diverse redundancy	✓	Research
	SafeDM [9]	RISC-V	2	diverse redundancy	✓	Research
	Marcinek and Pleskacz [39]	RISC-V	2	DCLS	✗	Research
	This work	**RISC-V**	**12**	**DCLS & TCLS & ECC**	✓	**Research**

2.2 Fault Tolerance Approaches

2.2.1 Radiation-induced Faults in Space. Since the beginning of spaceflight, the radiation environment has been an object of investigation [51], showing to be significantly more severe than on earth, with a flux over 10^7 particles/cm^2/day [12]. The effect on electronics, also essential for spaceflight's development, became apparent shortly after the first space missions [59], where high energy ions from cosmic rays cause charge separation within semiconductors, leading to voltage spikes within transistors. These voltage spikes can lead to inverted bits within memory cells or transient spikes at the output of logic gates. These are called "soft errors," as they can be corrected simply by re-writing the logic value and do not cause permanent faults within the circuit. A further subdivision of soft errors is between **Single Event Transients (SETs)** for transient events in combinational logic and **Single Event Upsets (SEUs)** for events affecting state-keeping logic and memories. Both SETs sampled by a register and SEUs (directly) can cause incorrect system behavior. While early integrated circuits seemed tolerant to these adverse effects, the hazardous space environment severely affects the more deeply integrated nodes used onboard newer satellites [56], leading to a requirement for reliable electronics. For example, on a 65 nm technology node, systems experience on the order of 10^{-7} errors/bit/day; this error rate is increasing for smaller technology nodes and is more and more due to SETs as clock frequencies are increasing [17]. The error rate is also estimated to vary with the flying orbit of the system [18], leading to a higher probability of a SET-induced erroneous value being sampled.

Fault mitigation approaches for ASICs' protection vary from using Radiation-Hardened technologies to architectural and system-level modifications intended to be adopted independently from the chosen technology to software-based approaches for protecting COTS devices. The state-of-the-art approaches are qualitatively discussed in the following and summarized in Table 1.

2.2.2 Radiation-Hardened by Design. **Radiation-Hardened (Rad-Hard)** technologies rely on silicon-level techniques, like transistor resizing and low-level design modifications, to increase the robustness of the technology cells towards particles striking the silicon. Industry groups like Gaisler [4, 29], BAE [10], DAHLIA [14], and Ramon Chips [24] mostly rely on Rad-Hard technologies to increase the fault-resilience of space-grade multi-core processors. This increases the designs' fault tolerance capabilities by reducing the engineering effort in implementing architectural solutions for application-specific SoCs. However, one of the negative aspects of Rad-Hard cells is that they are not readily available, being significantly more expensive than standard

technological processes. Moreover, Rad-Hard technologies, while significantly more resilient towards soft errors than commercial technologies, are not completely immune [17]. To integrate the tolerance features, Rad-Hard cells are significantly larger than those used in standard technologies due to their resilience requirements. Radiation hardening requires major investments when moving to a new technology node, both in time and money. Therefore, Rad-Hard libraries and design kits are generally only available for older technology nodes, significantly lagging behind commercial counterparts. This leads to increased costs due to custom, low-volume designs for low-performance and low-efficiency designs. While this is acceptable for small, mission-critical systems, it is problematic if applications require higher computing performance on a spacecraft.

2.2.3 Architectural Modifications. Rather than relying on technology-level hardening, a processor's architecture can also be used to enhance a system's reliability. This can mitigate one of the key performance and efficiency concerns, allowing modern technology nodes to be used for the design, albeit requiring more advanced architectural considerations and design. The two common approaches are (i) **ECC** for information redundancy of the stored bits (for example, in registers or memories) and (ii) modular redundancy, namely, **Double Modular Redundancy (DMR)** and **Triple Modular Redundancy (TMR)**, at various levels in the design, ensuring that multiple redundant copies of a critical module process the same information and produce the same outcome.

ECC is one of the most common and efficient ways to protect static data in SoCs' memories and registers. The encoding and decoding logic allows for individual data words' protection with limited additional bits, resulting in far less overhead than modular redundancy approaches. It is often used to further bolster designs protected by Rad-Hard technologies, improving their reliability. Other examples include the Duck-Core [38], which implements ECC in the pipeline registers, rolling back to allow re-execution of the last instruction in case a fault occurs.

When considering modular redundancy, the size of the replicated module considerably impacts the design effort, functionality, and performance. STRV [53] implements TMR at a very fine granularity within a RISC-V core, replicating the circuitry and voting after each register to ensure correct processing. Similarly, Gkiokas and Schoeberl [25] propose a TMR approach inside a five-stage dual issue time predictable core triplicating the fetch, decode, and execute stage of the pipeline. The results provided by the three execution stages are propagated to a voter in charge of deciding which should continue in the pipeline to the memory and register write-back stages.

The two approaches, ECC and modular redundancy, can also be combined well. SHAKTI-F [28] uses a hybrid approach, using ECC for registers and memory while implementing DMR for the ALU within the processor's execution stage, ensuring the calculation is executed correctly. While these modifications can significantly increase reliability, they require severe architectural modification of the underlying component and unrecoverable area overhead. Furthermore, Duck-Core and SHAKTI-F focus only on data protection within the cores, leaving the problem of control-flow faults uncovered.

The final architectural modifications to consider are full replications of the component blocks, using modular redundancy at a far coarser granularity. Prime examples are **Dual-Core Lockstep (DCLS)** and **Triple-Core Lockstep (TCLS)**, replicating the entire processing core and adding checkers and voters at the boundary. These system-level solutions are closer to the approach proposed in this work and are detailed further below in Section 2.3.

2.2.4 Software Approaches. Reliability can also be tackled in software, leveraging existing hardware COTS and adding fault tolerance in a later step. One option is to leverage multiple threads, duplicating [1] or triplicating [7] an application on separate threads executing concurrently or sequentially. These multiple executions are matched with a final checking step, either in hardware or software, to ensure correct execution. While these approaches require little to no modification

of the underlying hardware, they incur a heavy penalty in performance, either through additional time for execution or the use of multiple available threads, limiting parallelism. Performance is further penalized by the checking step, requiring either time or dedicated hardware resources to check for correctness. While this approach can properly handle faults in the data path, some faults in the processor's internal control logic may lead to an unrecoverable fault, possibly stalling a thread, thereafter requiring a watchdog timer to restart all processing properly with an extreme penalty.

Another software mitigation strategy is re-execution, which heavily uses COTS hardware and temporal reliability. Shoestring [19] duplicates instructions to protect only those segments of code that can result in user-visible faults when subjected to a software error without first showing faulty behavior. Similarly, Inherent Time Redundancy [42] consists in re-executing a program instruction with varying inputs. The assumption behind this approach is that certain microarchitectural events in a processor execution depend only on the instructions and not on the provided inputs. Jagtap et al. [33] use system reset with a watchdog timer to re-execute software that experienced a fault. Contrarily to hardware redundancy, it is possible to implement software-based approaches without introducing area overhead, as the replication of hardware components that have to be checked or voted on is unnecessary. Checking or voting can be performed in software but remains vulnerable to errors. Furthermore, re-execution and software checking introduces non-negligible processing overhead, impacting the performance of the protected application.

2.3 System-level Reliability Methods

System-level redundancy approaches make it possible to increase the reliability of digital systems without affecting the internal architecture of vulnerable components. ARM Cortex-R processors [5] and other ARM offerings feature a **Dual-Core Lockstep (DCLS)** option [31], allowing the SoC to group two cores within the design. Furthermore, this implementation allows for reconfiguration of the architecture while the cores are in reset, switching from a locked operation to a split one. The architecture does not specify the behavior in case an error occurs, relying on the surrounding SoC to determine fallback behavior and to handle and repair any remaining faults, generally with a reset or relying on check-pointing. However, critical applications on CPS with tight timing constraints may require faster repair mechanisms or even continuous operation, not tolerating any downtime due to an error that occurred. Onboard computers in spacecraft are responsible for executing critical real-time operations, which might tolerate minimum execution time variation to avoid the mission's failure [58]. Thus, ensuring that the system can satisfy strict time requirements during the recovery from incurring faults is essential.

Error recovery procedures for a DCLS implementation can be quite complex, as it is not possible to determine the correct output in case of mismatch. Therefore, Iturbe et al. [32] implemented a **Triple-Core Lockstep (TCLS)** design with the Cortex-R5 processor. The approach relies on an *assist unit* that wraps the cores to group them in triplets so that all the grouped cores can operate in lockstep with identical inputs and outputs decided with majority voting. The *assist unit* relies on Rad-Hard technology, while the remaining components use commercial technologies and are reinforced with modular redundancy and ECC. Any fault in a single core can be repaired during operation as the two remaining cores continue processing. A re-synchronization routine of the core ensures that any errors remaining in its state bits can be corrected. The cores store all the required internal state information in system memory, are reset by the *assist unit*, and read their stored state again from memory. While not discussed in detail in Reference [32], we assume the reconfiguration behavior of ARM TCLS to be similar to the DCLS implementation, requiring a reset to switch between a split and a locked state. Another common processing core for reliable systems is the AURIX TriCore [30], built on a custom ISA for embedded applications. This processor also

relies on a TCLS implementation to detect and correct faults, requiring a reset to recover from latent errors.

CEVERO [49] is an example of DCLS implementation using RISC-V-based Ibex [15] cores. Unlike ARM, it relies on a hardware-based recovery mode that consists in restoring their internal state, namely, the **Program Counter (PC)** and **Register File (RF)**, to a known safe one. Thus, CEVERO implements a *safety island* containing a backup copy of the cores' PC and RF, replicated every time the cores have matching outputs. In case of a mismatch, the two cores roll back to the last saved status. This is done by resetting the two cores, copying the backup copy of the PC and RF back into the cores, and restarting their operation. This solution introduces additional area overhead due to the backup registers. However, it requires only 40 clock cycles for a rollback to complete, significantly increasing the performance of the rollback procedure compared to other SW-based approaches. In this design, the cores in lockstep cannot be decoupled, creating a strong trade-off between area overhead and performance. In addition, the PC and RF alone do not represent the complete state to which a core should recover, leading to incomplete recovery. Moreover, no protection mechanism is applied to ensure the reliability of the backup registers.

Shukla and Ray [48] implement a re-configurable DCLS approach within a quad-core processor based on the RISC-V ISA. The simple, custom-designed core is paired with a mode control unit and a fault detection and correction unit, which ensure proper execution and correction depending on a dedicated mode selection signal. If a fault is detected, then the PC is held to repair the fault. While this may correct any error during execution, an error in the RF might remain uncorrected.

Kempf et al. [34] propose a dual-core adaptive runtime-selectable lockstep based on a LEON processor. While normally operating in independent mode, a master core will request the slave core's assistance with a critical task requiring reliability. The slave core halts its execution and assists the master core with the critical task, requiring a pipeline flush to start proper lockstep execution. The register file is not affected, as this is assumed to be ECC protected, and only the master core's register file is used. The LEON core is extended with a commit stage between the execute and memory stages to compare the results of the lockstepped cores' execution. If an error occurs, then the checker does not commit the instruction and flushes the pipeline to re-execute the failed instruction. This implementation offers an adaptive lockstep mechanism with fast hardware-based entry and exit switching, but it does not fully protect the core, focusing only on the data flow. Furthermore, deep modifications to the LEON core's architecture are still needed, as direct access to the registers of the main core and the ALU of the slave core is required, and an additional pipeline stage is added.

SafeDE [8], SafeDM [9], and Marcinek and Pleskacz [39] propose system-level approaches that enforce the concept of diversity in the execution of redundant threads. In particular, SafeDE enforces diversity by running redundant threads on different cores but introduces limitations in terms of applicability. It cannot be used to ensure the safe execution of parallelizable applications or for applications that require IO accesses. Otherwise, SafeDM allows a wider applicability range. However, the two proposed solutions only mitigate the effect of common cause failures. Marcinek and Pleskacz propose a variable-delayed dual-core lockstep approach to avoid common mode failure, but with no possibility to decouple the two cores for independent execution.

The work presented in this article belongs to the hardware redundant approaches. It proposes the implementation of system-level hardware enhancements such as independent/DCLS/TCLS split-lock, and ECC-protected backup status registers for fast re-configuration, re-synchronization, and recovery of a multi-core computing cluster. Furthermore, as detailed in Section 3, our approach relies on using unmodified industrially verified cores, therefore guaranteeing fault protection without requiring re-verification of the protected cores from a functional viewpoint.

3 ARCHITECTURE

The hardware template we rely on is the multi-core **Parallel Ultra-Low Power (PULP)** cluster [45], which can be extended with an external host subsystem featuring a controller core, a larger memory, and interfaces for external peripherals. The host subsystem is not area and performance-critical, and we assume it can be made fault-tolerant by means of high-overhead techniques.

3.1 Background: The PULP Cluster

The PULP cluster [45] features a parametric number of 32-bit CV32E40P cores [22], enhanced for fast DSP calculations and industrially verified by the OpenHW Group [27]. Each core features an instruction interface connected to a hierarchical instruction cache [13] consisting of one private bank per core, each configured to store 512 B. Each private bank fetches instructions from a larger, shared cache of 4 KiB, improving the performance of applications using the Single Program Multiple Data paradigm.

In addition, each core features a data interface that connects it to the rest of the system through dedicated demultiplexers for direct access to the system's **Tightly-Coupled Data Memory (TCDM)** and all other memory-mapped peripherals. The TCDM comprises a parametric number of 32 bit word-interleaved memory banks featuring single-cycle access latency. A banking factor of 2 (i.e., the number of memory banks is twice the number of cores) is typically used to minimize the memory banks contention probability, guaranteeing data sharing overhead smaller than 5%, even for highly memory-intensive workloads. If a collision occurs, then round-robin arbitration guarantees fairness and avoids starvation.

Along with the connection to the TCDM, the cores have access to both memory-mapped devices within the cluster and the host domain through a peripheral interconnect. One of the core-local peripherals is the **Direct Memory Access (DMA)** unit, capable of up to 64 bit/cycle data transfers in either direction, full duplex between the external larger memories in the host subsystem and the cluster's local TCDM.

Finally, the cores directly connect to an event unit [26] responsible for synchronization barriers within the cluster. Each core attempts to read from the corresponding register within the event unit and only receives a response once all cores request the same barrier address. Furthermore, the event unit manages interrupts within the cluster, masking and forwarding incoming interrupt signals to the responsible core.

To ensure the cluster can easily access the host system, the cluster has both an input and an output AXI port connected to an AXI interconnect. Through this interconnect, a host can access the cluster's internal memory and peripherals for configuration. To ensure proper operation, the instruction cache, the DMA, and the cores through the peripherals interconnect have access to the host system's memory.

The PULP ecosystem offers open-source software for parallel code execution on the multi-core cluster [41]. For various host systems, the software is provided to configure and boot the accelerating cluster and basic low-level drivers that allow for parallel software execution. The software ecosystem also provides a variety of parallel applications and benchmarks, enabling fast development of custom workloads.

3.2 On-Demand Redundancy Grouping

The baseline of the work described in this article is the **On-Demand Redundancy Grouping (ODRG)** [44], which consists in a wrapper that allows for TCLS grouping of the cluster cores. We have extended the ODRG by also allowing DCLS grouping and implementing hardware extensions for fast fault recovery and re-synchronization of the redundant cores. Furthermore, we extended

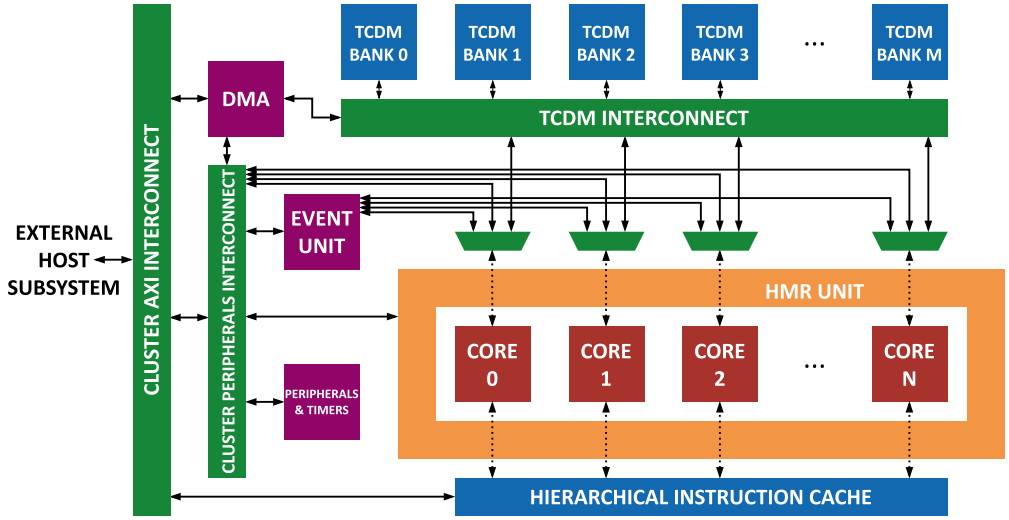


Fig. 1. Integration of the HMR unit within the PULP cluster.

the previous work with a faster split-lock mechanism that allows for rapid runtime-selectable reconfiguration of the cluster. This allows the execution of dedicated portions of code that can be defined as critical in one of the available redundant modes while operating in individual mode, or vice versa, a non-critical portion of code while operating in a reliable mode.

3.3 System-level Integration

We focus on implementing modular redundancy within the PULP cluster to safeguard the individual processor cores' correct operation. For this purpose, our design approach and experimental evaluations assume that the modules outside the cores, particularly the TCDM and the instruction memories, are reliable and protected by dedicated safety-critical techniques, such as ECC encodings and memory scrubbers. In the HMR cluster, the cores are grouped into lockstep, either in DCLS or TCLS. All inputs and outputs to the cores pass through a HMR unit surrounding the cluster cores, as shown in Figure 1, where the HMR unit is integrated in the PULP cluster described in Section 3.1. In a PULP cluster with 12 cores, the HMR unit can easily divide all cores into groups of 2 and 3 for DCLS and TCLS, respectively, without remainder. Should a different number of cores be chosen, the respective mode will be unavailable for the remaining cores.

To allow access to its configuration registers, the HMR unit exposes a peripherals memory port connected to the peripherals interconnect in the PULP cluster. Through this interconnect, any core within the cluster and any controller outside the cluster with access to the host memory port can configure the HMR unit. Furthermore, this memory port also allows the readout of all other memory-mapped registers within the unit, such as the current reliability state and error statistics measured within the unit.

3.4 Hybrid Modular Redundancy Block

Modular redundancy requires multiple hardware instances performing the same operation to ensure correct execution, with the result being compared in a DMR case or voted in a TMR case. For reliable calculations, all inputs of two or three cores are directly connected with each other, ensuring that the cores receive identical signals for subsequent processing. The outputs of the cores are then connected to dedicated checkers or majority voters to ensure matching calculations or detect

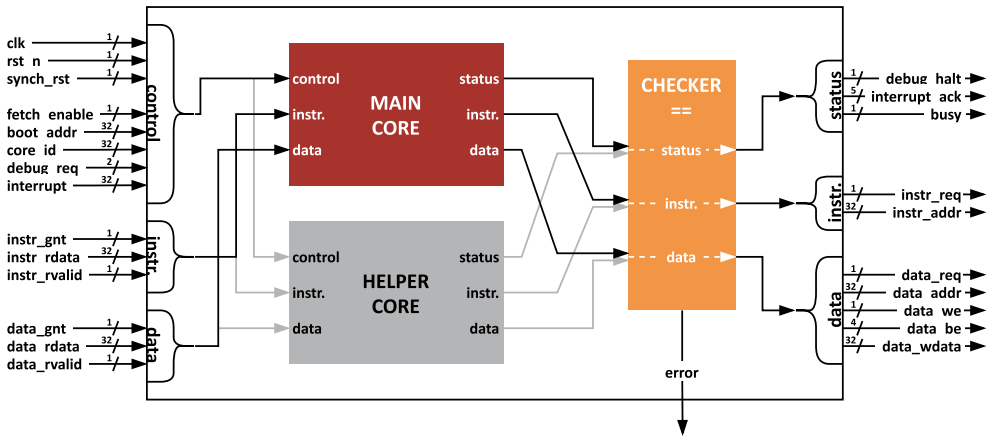


Fig. 2. **Dual-Core Lockstep Mode (DCLS-mode)** grouped cores.

any faults that might have occurred and then apply a recovery. When grouped, the locked cores behave within the system as a single virtual core.

While the cores are protected in case a fault occurs, the checkers and majority voters may also be vulnerable to soft errors. To mitigate this, we assume the cores implement additional protections for the bus, for example using ECC, being checked on a protocol level within the protected core region. Thus, any error occurring within the checkers and majority voters will result in a bus error, which can then be corrected.

3.4.1 Dual-Core Lockstep. When configured in **Dual-Core Lockstep Mode (DCLS-mode)**, the cores in the PULP cluster are paired, identifying one main core and one helper core, as shown in Figure 2. From a system perspective, one core is disabled to no longer act on the system's interconnect, while the other continues processing based on both cores' execution.

The cores of each group are configured to receive the same input information, so they are expected to generate the same output results. The outputs of each core in a group are then propagated to a bitwise checker that detects whether the results produced by the two cores are different. If the results of the two cores match, then the checker selects only the result of the main core as an effective output of the pair, propagating it to the rest of the system. However, in case of a mismatch, the checker raises an error signal, indicating that a fault affected the status of one of the two cores in the group. Since it is impossible to know which core produced the correct output in DCLS-mode, the checkers gate their outputs toward the system to avoid fault propagation. Furthermore, the error is directly signaled to the system to start a recovery procedure.

3.4.2 Triple-Core Lockstep. In the **Triple-Core Lockstep Mode (TCLS-mode)**, three cores are grouped, identifying one main core and two helper cores, as shown in Figure 3. As in DCLS-mode, the inputs from the system are shared among the cores in the group, ensuring that they operate on identical data and control signals. However, unlike in DCLS-mode, the outputs of the cores are connected back to the system through bitwise majority voters. Logic to disable the connection is unnecessary, as the voter properly selects the correct output. Each majority voter compares the outputs from the grouped cores and raises an error signal only if it detects a mismatch between them. Unlike DCLS-mode, since it is possible to vote over the results of three cores in TCLS-mode, the state of the faulty core can be restored using the state of the other two non-faulty cores. While not advisable with high error rates, letting the two non-faulty cores continue their operation in

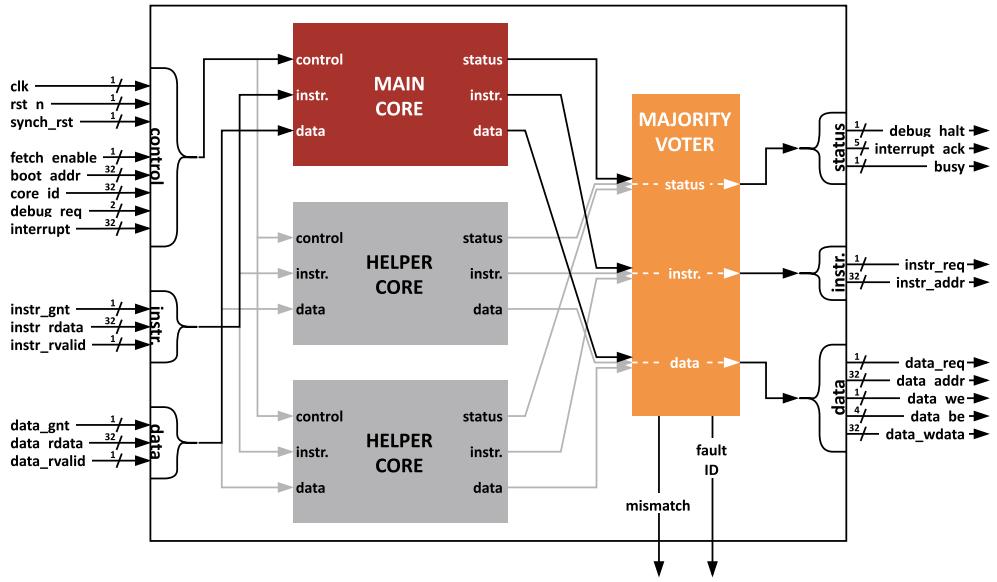


Fig. 3. **Triple-Core Lockstep Mode (TCLS-mode)** grouped cores.

lockstep without correction can be enabled within the HMR unit, delaying the faulty core's re-synchronization. However, if a second mismatch is detected, then the grouped cores must enter a re-synchronization routine immediately.

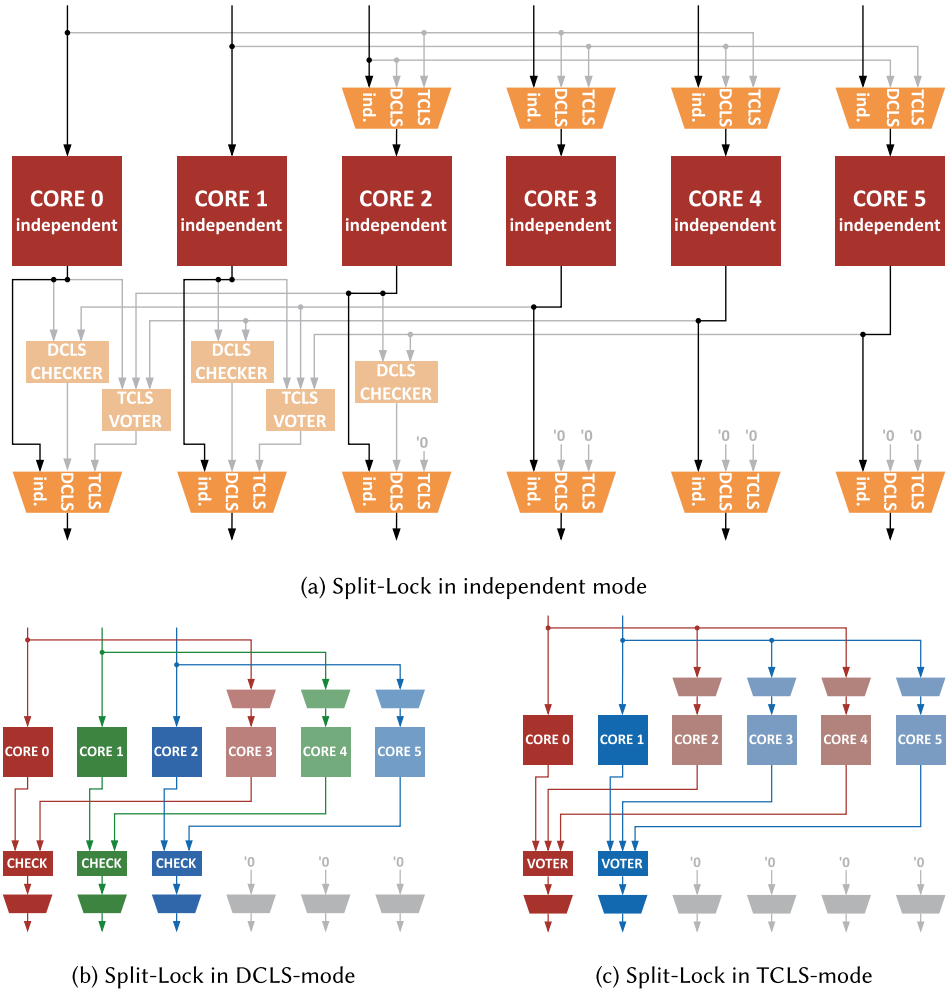
3.4.3 Parametrization. While the HMR unit offers both DCLS-mode and TCLS-mode configurations, the hardware can be parametrized to disable these different modes or enforce them permanently. This means that the HMR unit can offer either only DCLS-mode, only TCLS-mode, or both, supporting configuration switching with the split-lock explained below. If a desired instantiation does not require split-lock functionality, then DCLS-mode or TCLS-mode can be enforced permanently. However, these configurations were not investigated in detail.

The HMR unit is designed to be agnostic to the cores, requiring proper configuration to adjust to the core's interface. We tested it with the CV32E40P [22] core, the default for the PULP cluster and the configuration used here, and the Ibex [15] core.

3.5 Split-Lock Mechanism

Locking cores together into redundant configurations permanently is limiting for some application domains that might tolerate the occurrence of faults and require higher performance. For example, a satellite must satisfy tighter resilience constraints during orbital maneuvers than during image processing for the satellite's payload. Thus, the proposed PULP cluster can be configured before startup to work in the independent mode, DCLS-mode, or TCLS-mode, following the grouping procedure shown in Figure 4. This allows any application running on the host subsystem to configure the cluster for its purposes and reliability requirements after a cluster reset and before its boot sequence, locking the execution for the accelerated application into the specified reliability mode.

The main differentiator of the cores in the cluster is their respective identifier (ID). It starts at zero and is incremented for each additional core, allowing for easy identification of work sections in a parallelized task. To keep this convenience when the reliability modes are enabled, the DCLS-mode and TCLS-mode grouping of the cores is performed in an interleaved fashion, as shown in



(a) Split-Lock in independent mode

(b) Split-Lock in DCLS-mode

(c) Split-Lock in TCLS-mode

Fig. 4. Split-Lock configurations using available multiplexing in a six-core example.

the six-core example depicted in Figure 4. Figure 4(b) (DCLS-mode) shows that core 0 is grouped with core 3, while in Figure 4(c) (TCLS-mode), it is grouped with core 2 and core 4. This results in preserving the lowest core IDs when enabling all cores in the DCLS-mode or TCLS-mode reliability configurations, keeping a simple parallelization software scheme. Furthermore, each reliability group can be configured independently, allowing some cores to execute a reliable application without impeding the others from operating in independent mode.

Changing the reliability mode is also possible at runtime, allowing the cores to switch configurations based on the demands of the software application executed thereon. When starting in the individual configuration, the application can explicitly declare a *mission-critical section* with a portion of code that must be executed reliably. Alternatively, when starting in the reliable configuration, the application can declare a *performance section* with a portion of code that does not require stringent reliability guarantees. This allows a system to make use of the tradeoffs between reliability and performance on a single SoC with minimal configuration overheads.

3.5.1 Mission-Critical Section. The *mission-critical section* is a temporary reliable state of execution where two or three cores operate in lockstep, while otherwise remaining in the independent configuration. It is designed such that all cores assisting the main core for reliable execution only pause their own thread, returning to it after the *mission-critical section* completes. As the cores' internal registers (**Program Counter (PC)**, **Register File (RF)**, and **Control and Status Registers (CSRs)**) are used during the reliable execution, the helper cores' state is temporarily saved to the stack in the TCDM, allowing them to retrieve it later.

To enter the *mission-critical section*, the main core executes a custom software routine, shown in Listing 1, that writes the desired state (DCLS-mode or TCLS-mode) into the configuration register within the HMR unit. This triggers an interrupt in the cores that have to participate in the group, indicating they must start the execution of a *mission-critical section*. To ensure the cores are awake and available to serve it, the interrupt is routed through the cluster's event unit waking up the cores even if they are asleep waiting for a barrier. The interrupt service routine outlined in Listing 2 stores the internal state of the cores onto the stack in the TCDM. The internal state is represented by 30 modifiable registers of the **Register File (RF)**, excluding the **Stack Pointer (SP)**, and by specific CSRs, such as the *MEPC*, which stores the last **Program Counter (PC)** executed before that interruption.

Once the entire state is stored onto the stack, the SP is stored in a memory-mapped register within the HMR unit, indicating that this *unload* phase is complete, and the *reload* phase can start. After storing the state, the cores enter a synchronization barrier and are locked together, ensuring they all continue as one. Here, it is also possible to trigger a synchronous clear towards the locked cores to bring the internal flip-flops of the cores to their default value without polluting the reset network. As the SP storage register is linked to the core's ID, the grouped cores that share the same ID after grouping read the SP register and load the state back from the stack in parallel. After completion, the interrupt service routine is exited into the *mission-critical section* outlined by the software.

To exit the *mission-critical section*, the locked cores write the new desired state (i.e., independent mode) to the HMR unit configuration registers. This write operation frees all the cores from their locked state, so the main core can continue its execution, as its internal state is still consistent. However, the helper cores must return to the state before the *mission-critical section* entry. Therefore, the HMR module synchronously clears the helper cores bringing them back to the boot sequence, where they use their memory-mapped SP register to reload their state back from the TCDM.

3.5.2 Performance Section. The *performance section* allows an application typically requiring reliable execution to forego the stringent requirements and temporarily leverage the higher parallel performance possible with independent cores. It is designed to allow the main thread to continue processing, with the assisting cores branching from the main thread temporarily, after which the temporary threads are resolved at the end of the performance section.

```
int mission_critical_sec(int (*fn_handle)()) {
    enable_lockstep();
    // Interrupt locks cores together
    int ret = fn_handle();
    disable_lockstep();
    return ret;
}
```

Listing 1. Mission-Critical Section Wrapping Function.

```
void synchronization_irq(void) {
    store_state_to_stack();
    store_pc_to_reg(core_id());
    barrier();
    // Cores are locked together
    // Synchronous Clear Possible
    reload_pc_and_state(core_id());
    asm volatile ("mret");
}
```

Listing 2. Mission-Critical Section Entry Interrupt Service Routine.

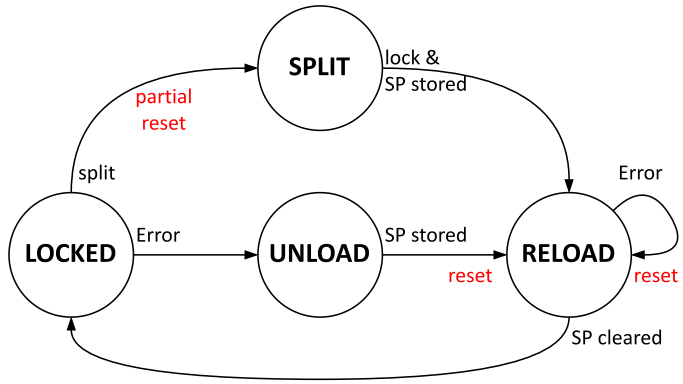


Fig. 5. **Finite State Machine (FSM)** of the TCLS-mode controller.

To enter the *performance section*, the cores split apart by a single write to a configuration register in the HMR unit. While the main core can continue the main thread, the newly separated cores simply update their SP to an independent region and also continue the application. Differentiated based on the core's ID, the separated cores operate on their own stack to avoid interfering with the main thread's stack.

Once the *performance section* is complete, the main core stores its internal state, and all cores enter a synchronization barrier to be re-grouped. The state of the separated cores is abandoned, as the section is only intended to be temporary to provide additional performance. Once locked together after the synchronization barrier, the cores load the main core's state in lockstep, similar to the entry of the *mission-critical section* described above, and continue the application.

3.6 Fault Recovery

In the DCLS-mode and TCLS-mode outlined in Section 3.4, a fault would not immediately impact the system as the outputs of the cores are either gated in DCLS-mode or voted in TCLS-mode. However, further execution may not be possible, as the outputs remain gated in the DCLS-mode, and any additional errors in another core of the group in TCLS-mode may cause complete failure. To avoid this scenario, the possibly corrupted state within the cores must be corrected, and the locked cores must be re-synchronized. It is also essential to guarantee that the system recovery can be performed in time to avoid critical tasks from failing. On-board computers execute critical tasks that must be guaranteed to complete in very limited time intervals [58]. Consequently, if incoming faults corrupt the system's status, then ensuring a minimal time to recovery to meet real-time operations' constraints is fundamental.

3.6.1 TCLS-Mode Software Re-synchronization Routine. When an error occurs in TCLS-mode affecting a single core, the cluster can continue operating, as the voters overrule the invalid outputs of the erroneous core. Furthermore, in TCLS-mode, the three cores provide redundant internal state information, allowing the system to recover to a fully functional and correct state without additional hardware. This implies storing the internal state of one of the non-faulty cores into the stack placed in TCDM during the *unload* stage and then reloading the state back to all three cores during the *reload* stage, similar to the *mission-critical section*.

To do this, as shown in Figure 5, the TCLS-mode FSM of the HMR unit enters the *unload* state and sends an interrupt to the cores through the event unit, triggering an interrupt service routine. This routine stores the internal state of the non-faulty cores (i.e., PC, RF, and CSRs) into the stack.

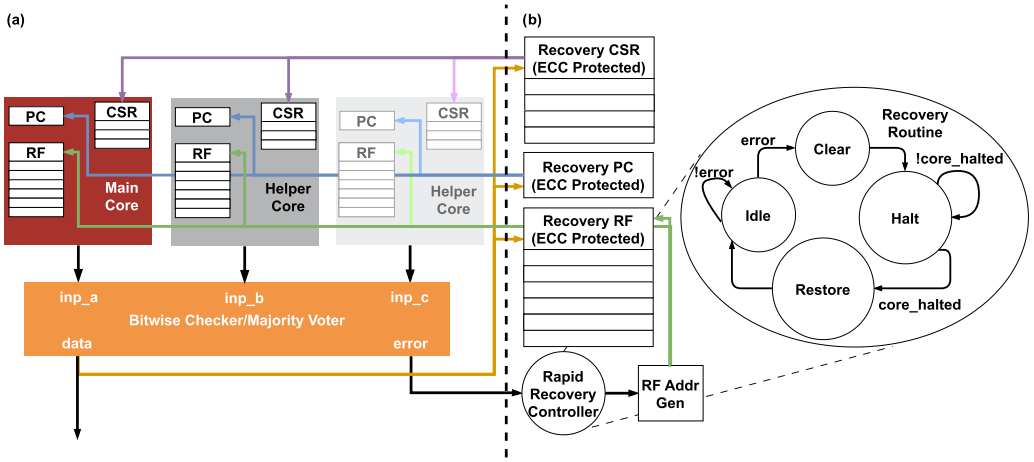


Fig. 6. Implementation of the *rapid recovery* hardware extension, highlighting (a) the grouped cores, and (b) the *rapid recovery* region containing the ECC-protected status registers and the finite state machine of the *rapid recovery* routine.

During the execution of the *unload* routine by all three cores, the bitwise voter scans the contents of these cores and stores only the state of the non-faulty cores in memory. Then, the SP is stored in the same memory-mapped register mentioned in the *mission-critical section*, which is located within the HMR unit. This indicates to the TCLS-mode FSM that the *unload* stage is complete, allowing it to enter the *reload* state. On this transition, the TCLS-mode FSM can also trigger a synchronous clear toward the locked cores if configured.

To refill the cores with the correct state, the cores use the memory-mapped SP register to reload the main core's state stored in memory back into the locked cores. Furthermore, the cores can be cleared again if another fault happens during the *reload* phase, avoiding the execution of another *unload* operation, since a safe state was already stored in memory. To properly execute this routine even after a clear, the boot sequence of the cores directly checks the SP register to see if the stored value is not zero. If its content is not zero, then that value is used by the cores as a reload SP. However, if the register content is zero, then the cores continue with a normal boot sequence. Therefore, once the reload is complete, the software stores back a zero value to the SP register in the HMR unit, switching the TCLS-mode FSM back into a nominal *run* state. The cores then execute an `mret` instruction, returning to the application running before the fault recovery interrupt was triggered.

3.6.2 Hardware-based Re-synchronization and Rapid Recovery Extension. The software-based *unload* and *reload* procedures execution can be further improved by integrating dedicated hardware for *rapid recovery* with minor modifications to the processor core's architecture. As in the software-based recovery mechanism, *rapid recovery* relies on the fact that the state of each core is defined by its PC, RF, and CSRs. Instead of copying the state of the cores to memory and exploiting a software-based interrupt service routine, we extended the HMR unit by introducing a hardwired recovery engine for each group of cores. Figure 6(a) shows the cores grouped in DCLS-mode or TCLS-mode, connected to a bitwise checker or a majority voter, as described in Section 3.4. The *rapid recovery* region, depicted in Figure 6(b), comprises four main modules: recovery PC, recovery RF, recovery CSRs, and a *rapid recovery* controller.

During normal processing without any errors, a backup of the main core's PC, RF, and CSRs content is copied into the registers of the *rapid recovery* region. This operation is done each cycle, as the core's interface was modified to expose the write ports of PC, RF, and CSRs to propagate

them to the backup registers. Hence, the backup write operation is accomplished in a single cycle. Furthermore, the recovery PC, recovery RF, and recovery CSRs are protected with internal ECC encoding/decoding, configurable through parameter selection at design time.

If the checkers or voters detect a mismatch, then they raise an error signal that blocks write operations to the registers in the *rapid recovery* region, ensuring their content is not corrupted by a faulty state. The same error signal triggers the *rapid recovery* controller that starts the recovery routine, depicted in Figure 6(b). The error flag from the checker or voter sends the FSM of the *rapid recovery* routine from the *Idle* state to the *Clear* state. This transition raises a synchronous clear signal to the faulty group, which brings the internal flip-flops of the cores to their default value without polluting the reset network. Then, the recovery routine jumps into the *Halt* state, where a debug request signal is raised towards the faulty cores, forcing them to enter the debug mode. Shortly after they receive the debug request, the cores raise the halt response signal toward the *rapid recovery* controller. In debug mode, the cores are halted and allow the recovery hardware to access their internal registers without interference.

Once the faulty cores have raised the halt response signal, the recovery routine jumps into the *Restore* state. In this state, the PC, RF, and CSRs content of the faulty cores is reloaded from the recovery PC, recovery RF, and recovery CSRs. This process is executed sequentially, restoring the content of all 31 modifiable registers of the RF, exploiting its two write ports in parallel. The PC and the CSRs are reloaded in parallel to the RF.

The same hardware introduced to enable the *rapid recovery* can be used to speed up the process of entering a *mission-critical section* and exiting the *performance section*. The recovery PC, recovery RF, and recovery CSRs are used to back up the state of the main core continually so that this state can be used to enter the routine safely. As the helper cores software is designed to be executed in order in the *mission-critical section*, they still require their state to be saved to the stack in the TCDM. Therefore, the main core immediately enters the barrier once the interrupt occurs, while the helper cores store their state to the stack and enter the barrier afterward. After all the cores have entered the barrier and are locked together, the *rapid recovery* hardware executes its recovery routine, synchronizing the state of the locked cores to continue into the *mission-critical section* or return to the reliable mode after the *performance section*.

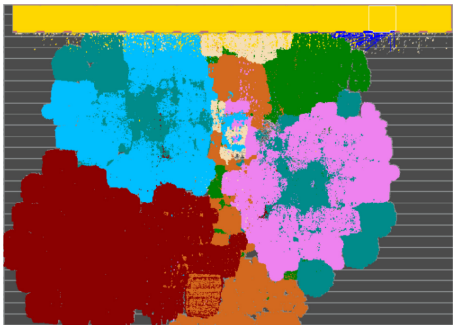
4 EVALUATION

4.1 Experimental Setup

Our evaluation setup consists of a PULP cluster featuring 12 CV32E40P cores explained in Section 3.1, with an integrated HMR unit as explained in Section 3.3. To evaluate the cluster's performance in different modes, the cluster was locked into the respective configuration before boot, avoiding unnecessary mode-switching overhead within the measurement. All performance evaluations were conducted in RTL simulation using QuestaSim, gathering performance data by using performance counters within the RTL or using simulation traces. We evaluated the split-lock and fault recovery performance by entering a *mission-critical section* or *performance section* twice to warm up the instruction caches and then collected the results corresponding to the hot cache measurement. For implementation purposes, we target GlobalFoundries 22 nm technology using Synopsys Design Compiler for synthesis (slow corner at $f_{\text{targ}} = 430$ MHz, $V_{DD} = 0.72$ V, $T = -40\,^{\circ}\text{C}, 125\,^{\circ}\text{C}$) and Cadence Innovus for full-cluster Place & Route in the same operating point.

4.2 Physical Implementation

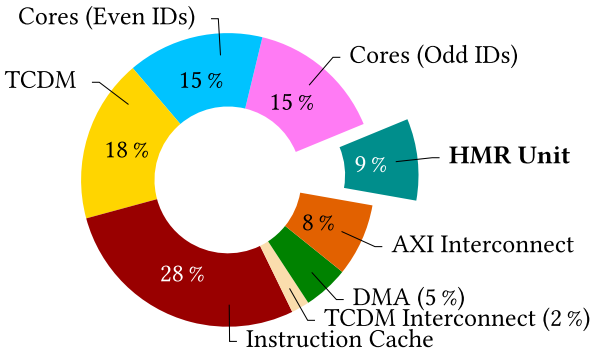
Figure 7(a) shows the post-layout implementation of our redundant PULP cluster with all redundancy modes and *rapid recovery* extension enabled, where we highlighted the cores with even and odd identifiers separately. Figure 7(b) shows the area occupation and the related overheads of



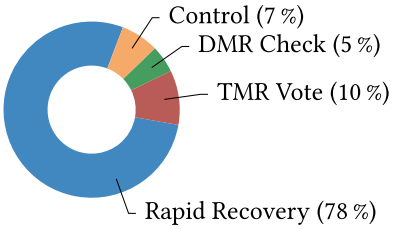
(a) Layout of the physical implementation of the fault-tolerant PULP cluster featuring full HMR and *rapid recovery*.

(b) PULP cluster area comparison in all available configurations.

PULP Cluster Area [mm^2]		Overhead
Baseline	0.604	-
DMR	0.605	0.3%
TMR	0.608	0.7%
HMR	0.612	1.3%
With Rapid Recovery		
DMR	0.654	8.4%
TMR	0.657	8.8%
HMR	0.660	9.4%



(c) Area breakdown of the fault-tolerant PULP cluster featuring full HMR and *rapid recovery*, with the same color as in the physical implementation layout.



(d) HMR unit area breakdown.

Fig. 7. Physical implementation of the fault-tolerant PULP cluster.

all the available PULP cluster configurations over the standard implementation, while Figure 7(c) shows the cluster area breakdown. In the chosen corner, the PULP cluster occupies 0.660 mm^2, where almost 27% is the RISC-V cores, and around 29% is the instruction cache. The HMR unit, the wrapper in which we encapsulate all redundancy features, accounts for 9% of the entire cluster area.

Figure 7(d) shows the area breakdown of the HMR unit, highlighting that the *rapid recovery* extension accounts for 79% of it. The TMR voters and DMR checkers account for 10% and 5% of the area occupation, respectively, while the shared control logic accounts for a 7% area overhead. The HMR unit is designed in a parametrized fashion, so it is possible to instantiate the unit without the hardware enhancement features for *rapid recovery*, allowing a software-based recovery routine only. As shown in Figure 7(b), the area overhead of the HMR unit with only software recovery features is limited to almost 1% of the PULP cluster area, at a high cost in fault recovery performance. The fault-tolerant cluster could be synthesized at up to 430 MHz operating frequency in the selected corner with no timing impact over the baseline implementation.

4.3 Performance Evaluation

4.3.1 Compute Performance. To evaluate the processing performance of the proposed fault-tolerant cluster, we initially target a highly parallel matrix-matrix multiplication benchmark. For

Table 2. Summary of the Performance Provided by the Proposed PULP Cluster in All the Redundant Available Configurations

			Base	DMR	TMR	DMR-R	TMR-R
						Rapid Recovery	
MatMul Performance		[MOPS @ 430 MHz]	1,165	617	414	617	414
SW-based MatMul Performance		*[MOPS @ 430 MHz]*	*1,165*	*576*	*351*	—	—
CFFT Performance		[MOPS @ 430 MHz]	989	531	385	531	385
Recovery Latency		[cycles]	—	—	363	24	24
Mission—Critical Performance	entry	[cycles]	—	534	410	397	310
	exit main		—	22	23	22	23
	exit help		—	147	165	184	182
	entry		—	134	82	125	82
	exit		—	373	311	183	94

evaluation, a size of 24×24×24 was chosen, meaning $2 \times (24 \times 24 \times 24) = 27{,}648$ Ops, for optimal parallelization in all configurations. Furthermore, a highly parallel, quantized **Complex Fast Fourier Transform (CFFT)** was evaluated, representing a typical workload for a S-CPS, e.g., used in radar processing. A length of 2,048 was chosen, which amounts to $10 \times \frac{n \log_2(n)}{2} = 112{,}640$ Ops. Table 2 summarizes the performance achieved by the fault-tolerant PULP cluster, showing that at 430 MHz our fault-tolerant cluster delivers a matrix-matrix multiplication compute performance of 414 MOPS in TCLS-mode, which increases to 617 MOPS in DCLS-mode and 1,165 MOPS in independent mode with all 12 cores available. Similarly, for the CFFT, the TCLS-mode achieves a compute performance of 385 MOPS, increasing to 531 MOPS in DCLS-mode and 989 in independent mode. This performance boost of 1.8–1.9× for DCLS-mode over the independent mode, and 2.5–2.8× for TCLS-mode over independent mode justifies the choice to enable quick switching between different modes to balance the trade-off between performance and fault resilience. Furthermore, the hardware enhancements introduced for the *rapid recovery* do not affect the compute performance of the PULP cluster.

To compare our hardware-based solution, we implemented a software-based redundant execution of the same matrix-matrix multiplication kernel utilizing the parallel cores in the PULP cluster for reliability. In the software-only approach, cores are similarly grouped (e.g., core 0 and core 6 in a DMR approach), operating on the same chunk of the kernel data. At the end of the computation, the two cores check each other's results, raising an error in case of inconsistency. If the check results in no error, then only one of the two cores (core 0, for example) validates the result storing the chunk in the memory. The use of software redundancy introduces significant overhead in designing the software, as it needs to be accounted for in each application manually. Further, it introduces a significant performance overhead due to memory contention (multiple cores accessing the same resources) and due to the mutual checking mechanism, resulting in 7% performance overhead during DMR execution and 11% overhead in the TMR case over the hardware lockstep approach. In addition, if the execution is corrupted in the DMR software, then the only way to recover from the fault is to repeat the computation, introducing additional non-negligible overhead that varies with the size of the executed kernel. Moreover, the overhead due to mutual checking in the software-only redundant execution depends on the size of the result.

4.3.2 Recovery Performance. The software-based recovery routine, described in Section 3.6.1, required 363 cycles to execute a recovery in TCLS-mode, as shown in Table 2. Figure 8 highlights that this is split between the *unload* section, requiring 247 cycles, after which it executes a

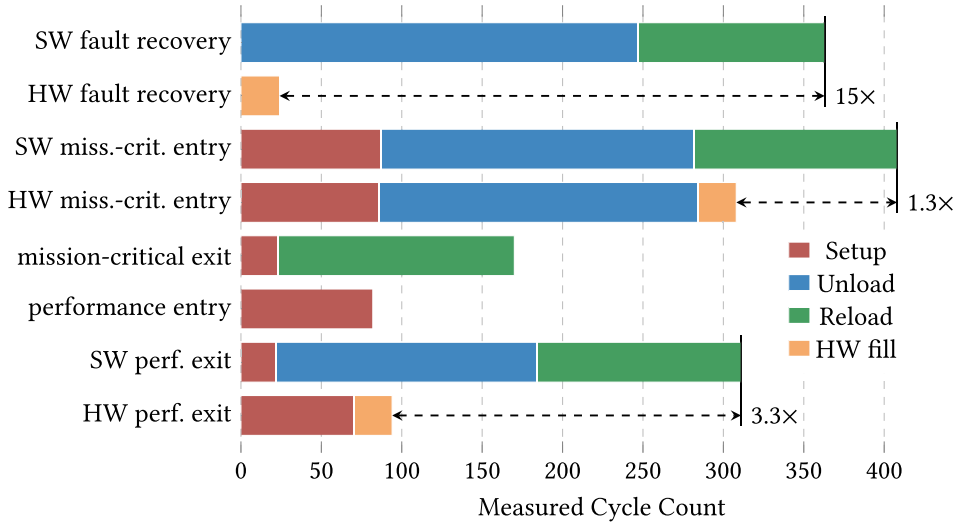


Fig. 8. Cycle count breakdown of the fault recovery and split-lock overheads.

synchronous clear of the core. Another 116 cycles are required for the *reload* phase, reloading the cores' state from memory. The *rapid recovery* hardware contributes positively by introducing a 15× speed-up during the fault recovery, thus reducing it to just 24 cycles. This recovery time initially requires 4 cycles to set up the *rapid recovery* controller, start the recovery routine, synchronously clear the cores, and send them the debug request. Additional 4 cycles are needed to receive the debug response from the cores, and 16 cycles to restore the PC, RF, and CSRs in parallel, after which they will continue processing. Furthermore, the *rapid recovery* hardware also allows for fast fault recovery in DCLS-mode, requiring the same fixed 24 cycles. The cycle-by-cycle backups of the cores' status make it possible to roll back the cores' execution to the closest safe state in time, restarting the execution from the last PC that entered the execution stage.

4.3.3 Split-lock Performance. We evaluate the performance of the proposed split-lock mechanism by measuring the cycles required to enter or exit a *mission-critical section* for the main and helper cores as well as a *performance section*, as shown in Table 2. The entry measurement starts from the point the main core initiates the entry into the relevant section to the point the internal code is being executed.

For the *mission-critical section*, Figure 8 shows that the main core first sets up the registers for the intended mode within 87 cycles, during which time the helper core(s) can continue processing. At this point, the interrupt is issued to all cores, launching them into the *unload* stage, where the cores save their state to memory and enter a barrier, requiring 198 cycles. Finally, during the *reload* phase, the cores reload the main core's state from memory in the locked configuration, requiring 126 cycles, after which the cores execute the *mission-critical section* code. When exiting the *mission-critical section*, the main core goes directly into the subsequent software, requiring only 23 cycles to set the configuration and return. However, the helper cores need to execute the *reload* procedure, requiring 147 cycles to continue the previously interrupted thread.

For the *performance section*, Figure 8 shows lower entry and exit cycle requirements than the *mission-critical section*. Notably, entry into the *performance section* only requires 82 cycles of setup. This is due to the cores only splitting apart and, if needed, updating a SP, not fully reloading a known state as they can keep the current execution state. Furthermore, the *performance section*

exit drops the state of the assisting cores, relying fully on the main core's state, thus requiring significantly fewer cycles than the *mission-critical section*.

Introducing the *rapid recovery* hardware helps speed up entering a *mission-critical section* as the software *reload* stage can be replaced with a 24-cycle hardware fill from the continuously backed-up main core, meaning ∼ 25% less than the software-based approach for a 1.3× speedup. Similarly, exiting a *performance section* also makes use of the continually replicated main core state, leading to a 3.3× speedup. Exiting a *mission-critical section* or entering a *performance section* behaves the same way as in the software-based approach.

4.4 Reliability Assessment

With the HMR implementation outlined in this work, any error within a single core propagating to its outputs will be detected when mismatching with the paired cores, and corrective action will be taken. Furthermore, with the *rapid recovery* extension enabled, core-internal signals backing up the RF are also checked, reducing the likelihood of latent errors within the cores' states. While it aims to detect and correct all possible SETs happening at the cores' interface or within the cores themselves, the proposed redundancy approach does not tackle SEUs and single hard errors that might corrupt the data and instruction memory hierarchy, which we thus assume to be reliable.

To test the behavior when a fault occurs, a single bit of the cores' interface signals was inverted using a `force` command during the execution of a test within an RTL simulation. With the cores in the locked state, this confirms that errors were detected and that the designed corrective action was initiated. Furthermore, select state bits within the cores' RF were flipped to confirm the proper behavior of the recovery implementations. All faults injected into the system were detected and corrected or overwritten before use, thus always leading to the correct termination of the running application.

The design was further subjected to a more extensive fault injection campaign, where all registers within a core were randomly subjected to faults. Running 7,717 individual RTL simulations of the matrix multiplication performance benchmark in the TCLS-mode configuration with software recovery, a single register was picked and flipped for each simulation run. All simulations terminated successfully, with 12% of injected errors leading to a recovery, meaning the other faults were masked.

Other research has conducted extensive fault investigation of the CV32E40P core used in our system [6], showing that many injected faults do not affect the program execution, with the most critical module being the controller module. As only 12% of injected errors lead to corrective action, this confirms the results of Asciolla et al. [6], where 44%–100% of injected errors lead to no correction, depending on the component.

4.5 Recovery Use-case Analysis: Satellite Onboard Image Processing

One of the most common use cases of multi-core S-CPS is satellite onboard image processing. Satellites' onboard sensors generate a large amount of data that heavily loads the data link and delays processing. When images captured onboard need to be processed through complex deep learning algorithms, the satellite first transmits images to the cloud computing data center of the ground station. Then, the data center operates on the received data using deep learning models and distributes the processing result to the user [54]. To reduce the overhead given by raw data transmission, the data processing can be directly moved onboard a spacecraft, only sending valuable information over the communication link. This requires the onboard processing systems to offer enough performance to process data while also offering reliability with quick recovery from incurring faults.

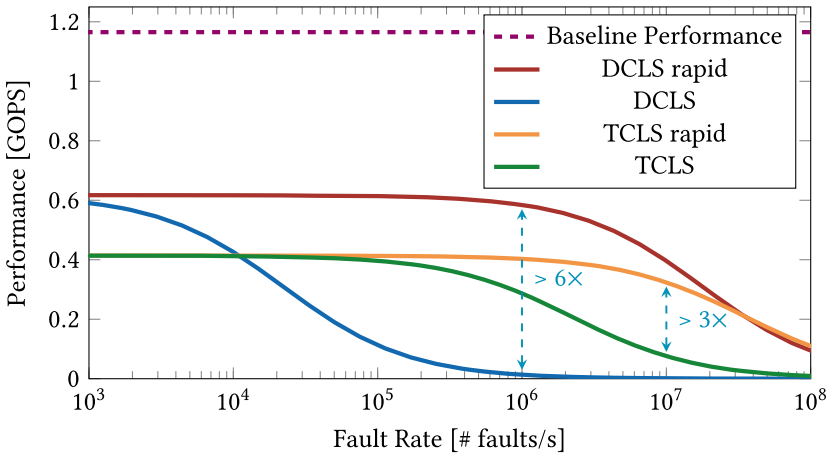


Fig. 9. PULP cluster performance degradation at increasing fault rate.

Figure 9 shows the performance degradation of our radiation-tolerant cluster when affected by multiple faults in all the available redundancy configurations. We considered a condition where the incurring faults happen during the execution time of a given application, with an increasing fault rate. In TCLS-mode, if a fault affects one of the three grouped cores, the other two continue their operation. After some time, if another fault happens and affects another of the cores of the same group, then it forces the entire group to perform a recovery. However, in DCLS-mode without *rapid recovery*, the cluster operation is restarted at each fault occurrence.

The analysis conducted shows that the worst-case scenarios are the software-based recovery procedures. In the DCLS-mode case, a performance drop of 50% happens with $\sim 2 \times 10^4$ faults/s injected. In the TCLS-mode, the same performance drop happens with around 2×10^6 faults/s. When using the *rapid recovery* extension, the increased fault rate reduces the performance of the computing cluster much slower, as shown by the DCLS rapid and TCLS rapid curves in Figure 9. The *rapid recovery* feature allows for more than 6× performance speedup during fault recovery over the software-based procedure at 10^6 faults/s, with the performance of the DCLS rapid case remaining almost constant. Similarly, TCLS rapid delivers 3× speedup over the software-based recovery with 10^7 faults/s injected. Interestingly, the DCLS rapid performs better than TCLS rapid until the fault rate reaches around 3×10^7 faults/s. As the fault rate increases further, the performance in TCLS rapid is the best, because, with TCLS, we do not necessarily need to recover immediately from a fault. If one of the three grouped cores is faulty, then the computation can continue with the other two cores, and a re-synchronization is only needed if another fault affects the same group of cores. In the DCLS rapid case, a re-synchronization is needed as soon as a fault happens. With *rapid recovery* features, we could observe a 50% performance degradation with a fault rate of around 2×10^7 faults/s and 4×10^7 faults/s for DCLS rapid and TCLS rapid, respectively. Furthermore, the conducted analysis shows that the proposed fault-tolerant PULP cluster allows for higher computing performance depending on the criticality of the application.

In independent mode, we only have two ways to determine that a fault happened. The first case is instructions faults, meaning incoming faults compromise the executed instructions sending the victim core in an unknown state. In this case, the faulty core stalls, and an external watchdog restarts the cluster operation. The second scenario is that of data faults. Suppose a fault affects the calculation of the cluster cores. In that case, we can only identify that an error happens if the host core knows the computation results a priori and can verify the results produced by the cluster

computation. Both these two scenarios imply a significant overhead, similar to or worse than the DCLS software-based recovery.

The analysis conducted so far helps understand the behavior of the system when exposed to varying fault rates. The fault rate corresponding to the worst-case estimation for a low-earth orbit device exposure to cosmic rays is in the order of $1.1—1.7 \times 10^{-3}$ faults/s (meaning 100/150 faults/day) and can vary with the flying orbit of the system and with the technology node [17, 18]. Furthermore, the probability that faults happening in sequence affect the same group of cores is low. In this scenario, the probability that the chip is corrupted by only one single fault during the entire execution of an application is high, even if the application runs for several seconds. If the fault hits one of three cores configured in TCLS with or without *rapid recovery*, then this fault may not affect the execution time of the application. Here, the two remaining cores can continue their operation without interruption until another fault hits the same group.

However, deeper considerations are needed if the system is configured in DCLS. With *rapid recovery*, a single fault recovery has no impact on an application that runs for more than thousands of clock cycles (meaning at least 2 μs at the considered frequency). The motivation is that the *rapid recovery* extension guarantees 24 clock cycles (approximately 56 ns at 430 MHz) to complete the entire recovery procedure. Thus, if an application lasts several seconds, then the DCLS rapid configuration does not introduce any significant degradation in the execution time.

The same does not hold in the case of DCLS. In this configuration, the only way to recover from a fault hitting a group of cores is to restart the application execution. The consequence is that the time to recovery varies depending on two factors: the moment the fault occurs during execution and the duration of the application. The moment the fault corrupts the executed application (if at the beginning, in the middle, or at the end) impacts the time to recovery significantly. In the worst case, the fault happens close to the end of the execution. In this situation, the time to recovery equals the time of the application itself, because it must be repeated almost entirely. The second factor, the execution time of the application (hundreds of microseconds or tens of seconds), further affects the time to recovery. A critical operation may have a strict deadline that is only slightly longer than the required computation time. In this case, the unpredictability of the recovery time can severely impact the real-time capabilities of a system, potentially leading to a failure of the mission. The ISO 26262 standard provides a definition for the **Fault Tolerant Time Interval (FTTI)**, which is the duration required to identify a fault within the system, initiate a response to the fault, and return the system to a safe state prior to the occurrence of a hazardous event. FTTI is composed of three components: the time needed for fault detection (Diagnostic Test Interval), the time taken to respond to the fault (Fault Reaction Time), and the time required to re-establish a safe operational state (Safe Tolerance Time) [40]. The precise definition of these parameters is contingent on the specific application executed by the system and should be minimized to ensure the system operates safely across various scenarios.

In Figure 10, we show how the recovery cost from incurring errors affects the execution of an application when the error rate ranges between 10^{-3} faults/s and 1 faults/s. We show the recovery cost through a runtime overhead expressed in seconds, highlighted on the color bar on the right, and such overhead is computed taking into account a minimum of 10^{-3} faults happening per application execution, on average.

For the DCLS configuration, we have considered that the incurring faults happen in the middle of the execution, which is the fault occurrence expected value. The consequence is that for every fault affecting the system, the overhead introduced for the recovery is half the execution time. Under these assumptions, the plots show that the PULP cluster configured in DCLS without the *rapid recovery* has a minimum runtime overhead of 10 μs. In particular, at a 1×10^{-3} faults/s rate, if the executed application requires 1–100 s, then the runtime overhead equals the execution

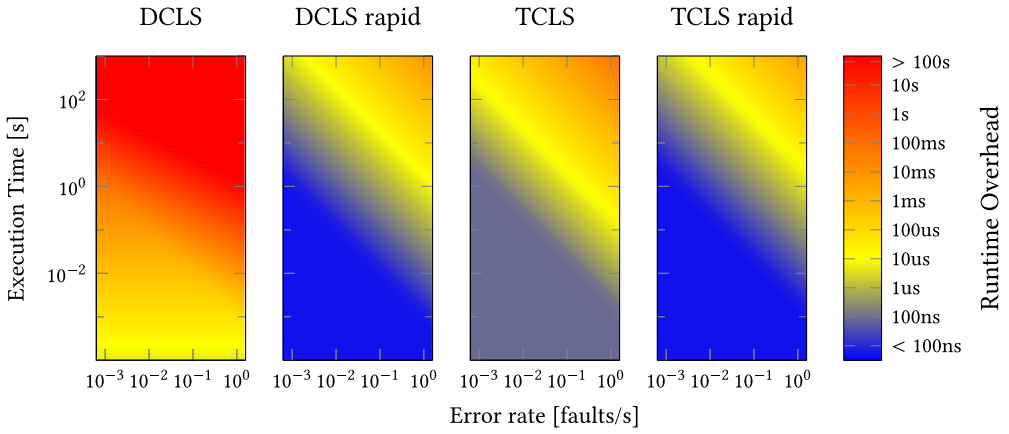


Fig. 10. PULP cluster recovery cost from incurring faults in the available redundant modes.

time of the given application, hence being not suitable for real-time executions. Conversely, when configured in DCLS rapid, TCLS, and TCLS rapid, there is a wide range where the recovery cost is negligible. The result is that the runtime overhead due to the recovery is always lower than 10 ms, also at higher fault rates. It is crucial to highlight that the rapid recovery feature enables the DCLS configuration to align with real-time constraints. This allows for a balance between computational performance and the capacity for real-time fault recovery, facilitating the reduction of the FTTI. It is important to note that the *rapid recovery* feature makes it possible for the DCLS configuration to meet real-time constraints, allowing for a trade-off between computing performance and real-time fault recovery capabilities.

5 STATE-OF-THE-ART COMPARISON

Table 3 shows the details of our implementation alongside the State of the Art. Compared with ARM, their TCLS [32] implementation features a split-lock mechanism that is based on resetting the system to configure it in independent or TCLS-mode. Furthermore, the recovery routine they propose takes 2351 clock periods to conclude, meaning 6.5× slower than our software-based TCLS-mode. ARM's DCLS [31] also features split-lock functionalities decided during system reset. Our split-lock allows for higher flexibility and performance, making the cluster capable of runtime switching between independent and DCLS-mode or TCLS-mode in 681 and 597 clock cycles, respectively, with further 1.17× and 1.08× reduction with the *rapid recovery* hardware support. Moreover, ARM's TCLS implementation introduces 27% area overhead over the single core implementation, while the HMR unit with *rapid recovery* features proposed in this work introduces just 9% area overhead over the baseline implementation.

Kempf et al. [34] propose a dual-core adaptive runtime-selectable lockstep processor based on a LEON core with internal modifications for instruction comparison. The fault recovery is software-based, with a time-to-recovery that is application dependent. Our *rapid recovery* hardware extension, however, allows for fixed 24 cycles to recover from occurring faults. Also, the area overhead of the solution proposed by Kempf et al. accounts for 21.3% in terms of CLBs over the regular LEON implementation, while our full HMR unit accounts for just 9% of area overhead over a standard 12-cores PULP cluster, with a similar dual-core configuration needing only 8.4%.

CEVERO [49] proposes a DCLS system with no split-lock capabilities and with a similar *rapid recovery* hardware extension to ours, with a recovery procedure that takes 1.67× longer than ours. Furthermore, CEVERO disregards copying the CSRs, which are fundamental in defining the

Table 3. State-of-the-Art Comparison Table for Qualitative Comparison on the Top Side and Quantitative Comparison on the Bottom Side

	ISA	Reliability Method	Fault Recovery	Split-Lock	Area Increase	Recovery Cycles	Frequency [MHz]	Area [mm^2]	Open-Source
This Work	**RISC-V**	**DCLS-mode**	**SW**	✓	**1.4%**	**–**	**430**	**0.608**	✓
			HW		**7.8%**	**24**		**0.657**	
		TCLS-mode	**SW**		**1.8%**	**363**		**0.605**	
			HW		**8.3%**	**24**		**0.654**	
ARM DCLS [31]	ARM	DCLS	SW	✓[1]	—	App. dep.	—	—	✗
ARM TCLS [32]	ARM	TCLS	SW	✓[1]	27%	2351	314	3.640	✗
Shukla and Ray [48]	RISC-V	DCLS	HW	✓[1]	17.9%	1	41	0.223	✗
Kempf et al. [34]	SPARC	DCLS	SW	✓	21.3%[2]	—	100	—	✗
CEVERO [49]	RISC-V	DCLS	HW	✗	—	40	—	—	✓
SHAKTI-F [28]	RISC-V	block-level DMR	HW	✗	20.5%	3	330	0.325	✗
Gkiokas and Schoeberl [25]	RISC	block-level TMR	HW	✗	15%[2]	0	50.56	—	✗
Duck Core [38]	RISC-V	ECC	HW	✗	0.7%[2]	3	50	—	✗
SafeDE [8]	RISC-V	diverse redundancy	HW	✗	0.79%[2]	—	—	—	✓
SafeDM [9]	RISC-V	diverse redundancy	HW	✗	3.4%[2]	—	—	—	✓

[1] Mode switching only possible during reset. [2] Overhead in CLBs or LUTs.

complete recovery state of the core, and the backup copies of PC and RF are not protected by error correction, meaning there is no guarantee that the backup state is reliable.

Shukla and Ray [48] present a quad-core RISC-V-based processor re-configurable for DCLS operation, introducing up to 17.9% area overhead over the base implementation, while our HMR unit offers more flexibility by introducing just 9% area overhead over a standard 12-cores PULP cluster. In addition, Shukla and Ray rely on saving just the last executed PC for the recovery, which is insufficient to determine the entire state of the grouped cores.

In Table 3, we also compare our design with other works, such as SHATKI-F [28], Duck Core [38], and Gkiokas and Schoeberl [25]. The first two works propose modifications to RISC-V cores' internal microarchitecture for pipeline rollback. However, the third proposes extending a RISC architecture with triple repetition of all the pipeline stages until the execution stage. The results produced by the three execution stages are then voted and propagated to the memory and write-back stages of the pipeline. The latter two pipeline stages do not enforce any redundancy technique to guarantee consistency, thus leaving the memory and write-back stages unprotected. The proposed solutions show a valuable approach that leads to just 3-to-0 clock cycles to perform a fault recovery, allowing for a single core to be reliable without the need for redundant grouping, thus saving resources. In contrast, the extensive modifications required by the internal architecture of the core can significantly change the behaviour, compromising its formal verification.

We also compared our work with SafeDE [8] and SafeDM [9], proposing hardware monitors to enforce diverse redundancy in a multicore RISC-V system, tackling the coverage of common cause failures in redundant threads. In SafeDE, the only way to understand if the diversity exists in the executed thread is to compute the distance, in terms of instructions count, between the head and the trailer core, with the trailer core being stalled for the required clock cycles to respect the diversity constraint. However, there is no description of how to mitigate the case in which one of the two executions is corrupted by a fault, so we assume that the two cores are reset and forced to re-execute the code. Similar considerations apply to SafeDM, where it is unclear if a fault between the two redundant executions can even be detected.

Finally, Table 3 also shows qualitative comparisons to the industry-standard approach of Rad-Hard technology augmented with ECC for memories. This approach, adopted by Gaisler [29], BAE [10], Ramon Chips [24], and the DAHLIA project with the NG-ultra [14], significantly simplifies integration of reliability. However, these devices generally feature lower performance and lower efficiency than counterparts built on commercial technologies and often come at a significant

price premium due to the use of Rad-Hard technology. Furthermore, in contrast to our solution, they offer no way to disable these reliability methods in case they are not needed, sacrificing performance in case they are still used in these cases. If a large-scale device deployment requires certain nodes with reliability and certain nodes without, then either different devices need to be used, requiring additional development overhead, or certain nodes pay in performance, efficiency, and cost. Finally, architectural solutions can more easily benefit from advances in technology, as these can more easily be ported to the most modern commercial technologies, not requiring additional development of new radiation-hardened libraries and design kits.

6 CONCLUSION

In this work, we enhanced a PULP cluster with fault-tolerant capabilities introducing a novel Hybrid Modular Redundancy approach. Our solution allows for flexible on-demand dual-core and triple-core lockstep grouping with dynamic runtime split-lock capabilities and hardware/software-based recovery approaches, being the first system to integrate similar functionalities on an open-source RISC-V-based compute device for finely tunable reliability versus performance trade-offs. The hardware-based recovery provides rapid fault recovery in just 24 clock cycles with just $\sim$ 9% area overhead over the baseline implementation, while the software-based recovery takes 363 clock cycles introducing just 1.3% area overhead. The proposed runtime split-lock mechanism allows for entering and exiting one of the available redundant modes with minimal performance loss, allowing execution of mission-critical portions of code or sections requiring performance while otherwise in reliable mode with <300 and <200 cycles of configuration overhead, respectively.

REFERENCES

[1] Sergi Alcaide, Leonidas Kosmidis, Carles Hernandez, and Jaume Abella. 2020. Software-only based diverse redundancy for ASIL-D automotive applications on embedded HPC platforms. In *Proceedings of the IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT'20)*. 1–4. https://doi.org/10.1109/DFT50435.2020.9250750 ISSN: 2377-7966.

[2] M. L. Alles, R. D. Schrimpf, R. A. Reed, L. W. Massengill, R. A. Weller, M. H. Mendenhall, D. R. Ball, K. M. Warren, T. D. Loveless, J. S. Kauppila, and B. D. Sierawski. 2011. Radiation hardness of FDSOI and FinFET technologies. In *Proceedings of the IEEE International SOI Conference*. 1–2. https://doi.org/10.1109/SOI.2011.6081714 ISSN: 1078-621X.

[3] Jan Andersson. 2020. Development of a NOEL-V RISC-V SoC targeting space applications. In *Proceedings of the 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W'20)*. IEEE, 66–67. https://doi.org/10.1109/DSN-W50199.2020.00020 ISSN: 2325-6664.

[4] Jan Andersson, Magnus Hjorth, Fredrik Johansson, and Sandi Habinc. 2017. LEON processor devices for space missions: First 20 years of LEON in space. In *Proceedings of the 6th International Conference on Space Mission Challenges for Information Technology (SMC-IT'17)*. IEEE, 136–141. https://doi.org/10.1109/SMC-IT.2017.31

[5] ARM. 2011. Cortex-R5 Technical Reference Manual. Retrieved from https://developer.arm.com/documentation/ddi0460/d

[6] Dario Asciolla, Luigi Dilillo, Douglas Santos, Douglas Melo, Alessandra Menicucci, and Marco Ottavi. 2020. Characterization of a RISC-V microcontroller through fault injection. In *Applications in Electronics Pervading Industry, Environment and Society*, Sergio Saponara and Alessandro De Gloria (Eds.). Springer International Publishing, Cham, 91–101. https://doi.org/10.1007/978-3-030-37277-4_11

[7] Marcello Barbirotta, Abdallah Cheikh, Antonio Mastrandrea, Francesco Menichelli, and Mauro Olivieri. 2022. Design and evaluation of buffered triple modular redundancy in interleaved-multi-threading processors. *IEEE Access* 10 (2022), 126074–126088. https://doi.org/10.1109/ACCESS.2022.3225975

[8] Francisco Bas, Sergi Alcaide, Ruben Lorenzo, Guillem Cabo, Guillermo Gil, Oriol Sala, Fabio Mazzocchetti, David Trilla, and Jaume Abella. 2021. SafeDE: A flexible diversity enforcement hardware module for light-lockstepping. In *Proceedings of the IEEE 27th International Symposium on On-Line Testing and Robust System Design (IOLTS'21)*. 1–7. https://doi.org/10.1109/IOLTS52814.2021.9486715 ISSN: 1942-9401.

[9] Francisco Bas, Pedro Benedicte, Sergi Alcaide, Guillem Cabo, Fabio Mazzocchetti, and Jaume Abella. 2022. SafeDM: A hardware diversity monitor for redundant execution on non-lockstepped cores. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'22)*. 358–363. https://doi.org/10.23919/DATE54114.2022.9774540 ISSN: 1558-1101.

[10] Richard Berger, Steve Chadwick, Ernesto Chan, Richard Ferguson, Patrick Fleming, Jane Gilliam, Michael Graziano, Mary Hanley, Andrew Kelly, Marla Lassa, Bin Li, Robert Lapihuska, Joe Marshall, Hugh Miller, Dave Moser, Dan Pirkl, Dale Rickard, Jason Ross, Brian Saari, Dan Stanley, and Joe Stevenson. 2015. Quad-core radiation-hardened system-on-chip power architecture processor. In *Proceedings of the IEEE Aerospace Conference*. IEEE, 1–12. https://doi.org/10.1109/AERO.2015.7119114 ISSN: 1095-323X.

[11] Md Zakirul Alam Bhuiyan, Sy-yen Kuo, Damian Lyons, and Zili Shao. 2018. Dependability in cyber-physical systems and applications. *ACM Trans. Cyber-Phys. Syst.* 3, 1 (Sept. 2018), 1:1–1:4. https://doi.org/10.1145/3271432

[12] S. Bourdarie and M. Xapsos. 2008. The near-earth space radiation environment. *IEEE Trans. Nuclear Sci.* 55, 4 (2008), 1810–1832. https://doi.org/10.1109/TNS.2008.2001409

[13] Jie Chen, Igor Loi, Eric Flamand, Giuseppe Tagliavini, Luca Benini, and Davide Rossi. 2023. Scalable hierarchical instruction cache for ultralow-power processors clusters. In *Proceedings of the Conference on IEEE Transactions on Very Large Scale Integration Systems (VLSI'23)*. 1–14. https://doi.org/10.1109/TVLSI.2022.3228336

[14] Estelle Danard and Benoit Leroy. 2022. NG-Ultra: A system-on-chip suiting the upcoming space missions. Retrieved from https://dahlia-h2020.eu/wp-content/uploads/2022/07/DASIA_2022_NG-Ultra.pdf

[15] Pasquale Davide Schiavone, Francesco Conti, Davide Rossi, Michael Gautschi, Antonio Pullini, Eric Flamand, and Luca Benini. 2017. Slow and steady wins the race? A comparison of ultra-low-power RISC-V cores for Internet-of-Things applications. In *Proceedings of the 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS'17)*. IEEE, 1–8. https://doi.org/10.1109/PATMOS.2017.8106976

[16] Stefano Di Mascio, Alessandra Menicucci, Eberhard Gill, Gianluca Furano, and Claudio Monteleone. 2021. On-board decision making in space with deep neural networks and RISC-V vector processors. *J. Aerospace Info. Syst.* (June 2021), 1–17. https://doi.org/10.2514/1.I010916

[17] Stefano Di Mascio, Alessandra Menicucci, Eberhard Gill, Gianluca Furano, and Claudio Monteleone. 2021. Open-source IP cores for space: A processor-level perspective on soft errors in the RISC-V era. *Comput. Sci. Rev.* 39 (Feb. 2021), 100349. https://doi.org/10.1016/j.cosrev.2020.100349

[18] Joshua Engel, Keith S. Morgan, Michael J. Wirthlin, and Paul S. Graham. 2006. *Predicting On-Orbit Static Single Event Upset Rates in Xilinx Virtex FPGAs*. Faculty Publications 1307. Los Alamos National Laboratory. Retrieved from http://hdl.lib.byu.edu/1877/431

[19] Shuguang Feng, Shantanu Gupta, Amin Ansari, and Scott Mahlke. 2010. Shoestring: Probabilistic soft error reliability on the cheap. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'10)*. ACM, New York, NY, 385–396. https://doi.org/10.1145/1736020.1736063

[20] Gianluca Furano, Stefano Di Mascio, Alessandra Menicucci, and Claudio Monteleone. 2022. A European roadmap to leverage RISC-V in space applications. In *Proceedings of the IEEE Aerospace Conference (AERO'22)*. IEEE, 1–7. https://doi.org/10.1109/AERO53065.2022.9843361 ISSN: 1095-323X.

[21] Gianluca Furano, Antonis Tavoularis, and Marco Rovatti. 2020. AI in space: Applications examples and challenges. In *Proceedings of the IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT'20)*. IEEE, Frascati, Italy, 1–6. https://doi.org/10.1109/DFT50435.2020.9250908

[22] Michael Gautschi, Pasquale Davide Schiavone, Andreas Traber, Igor Loi, Antonio Pullini, Davide Rossi, Eric Flamand, Frank K. Gürkaynak, and Luca Benini. 2017. Near-threshold RISC-V core With DSP extensions for scalable IoT endpoint devices. *IEEE Trans. Very Large Scale Integr. Syst.* 25, 10 (Oct. 2017), 2700–2713. https://doi.org/10.1109/TVLSI.2017.2654506

[23] Ran Ginosar. 2012. Survey of Processors for Space. In *Proceedings of DASIA 2012 DAta Systems in Aerospace*, L. Ouwehand (Ed.). https://ui.adsabs.harvard.edu/abs/2012ESASP.701E..10G/abstract

[24] Ran Ginosar, Peleg Aviely, Tsvika Israeli, and Henri Meirov. 2016. RC64: High performance rad-hard manycore. In *Proceedings of the IEEE Aerospace Conference*. IEEE, 1–9. https://doi.org/10.1109/AERO.2016.7500697

[25] Christos Gkiokas and Martin Schoeberl. 2019. A fault-tolerant time-predictable processor. In *Proceedings of the IEEE Nordic Circuits and Systems Conference (NORCAS): NORCHIP and International Symposium of System-on-Chip (SoC'19)*. 1–6. https://doi.org/10.1109/NORCHIP.2019.8906947

[26] Florian Glaser, Giuseppe Tagliavini, Davide Rossi, Germain Haugou, Qiuting Huang, and Luca Benini. 2021. Energy-efficient hardware-accelerated synchronization for shared-L1-memory multiprocessor clusters. *IEEE Trans. Parallel Distrib. Syst.* 32, 3 (Mar. 2021), 633–648. https://doi.org/10.1109/TPDS.2020.3028691 Conference Name: IEEE Transactions on Parallel and Distributed Systems.

[27] OpenHW Group. 2023. OpenHW Group | OpenHW Group. Retrieved from https://www.openhwgroup.org/

[28] Sukrat Gupta, Neel Gala, G. S. Madhusudan, and V. Kamakoti. 2015. SHAKTI-F: A fault tolerant microprocessor architecture. In *Proceedings of the IEEE 24th Asian Test Symposium (ATS'15)*. 163–168. https://doi.org/10.1109/ATS.2015.35 ISSN: 2377-5386.

[29] Magnus Hijorth, Martin Aberg, Nils-Johan Wessman, Jan Andersson, Remy Chevallier, Russel Forsyth, Rolad Weigand, and Luca Fossati. 2015. GR740: Rad-hard quad-core LEON4FT system-on-chip. In *Proceedings of the DASIA Conference*, Vol. 732. ESA, 7. Retrieved from https://ui.adsabs.harvard.edu/abs/2015ESASP.732E...7H

[30] Infineon. 2016. AURIX—TriCore Datasheet. Highly Integrated and Performance Optimized 32-Bit Microcontrollers for Automotive and Industrial Applications. Retrieved from https://www.infineon.com/dgdl/TriCore_Family_BR-2016_web.pdf?fileId=5546d46152e4636f0152e59a1581001d

[31] Xabier Iturbe, Balaji Venu, and Emre Ozer. 2016. Soft error vulnerability assessment of the real-time safety-related ARM cortex-R5 CPU. In *Proceedings of the IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT'16)*. IEEE, 91–96. https://doi.org/10.1109/DFT.2016.7684076 ISSN: 2377-7966.

[32] Xabier Iturbe, Balaji Venu, Emre Ozer, Jean-Luc Poupat, Gregoire Gimenez, and Hans-Ulrich Zurek. 2019. The arm triple core lock-step (TCLS) processor. *ACM Trans. Comput. Syst.* 36, 3 (Aug. 2019), 1–30. https://doi.org/10.1145/3323917

[33] Pushpak Jagtap, Fardin Abdi, Matthias Rungger, Majid Zamani, and Marco Caccamo. 2020. Software fault tolerance for cyber-physical systems via full system restart. *ACM Trans. Cyber-Phys. Syst.* 4, 4 (Aug. 2020), 47:1–47:20. https://doi.org/10.1145/3407183

[34] Fabian Kempf, Thomas Hartmann, Steffen Baehr, and Juergen Becker. 2021. An adaptive lockstep architecture for mixed-criticality systems. In *Proceedings of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI'21)*. IEEE, 7–12. https://doi.org/10.1109/ISVLSI51109.2021.00013 ISSN: 2159-3477.

[35] Andrew T. Klesh, James W. Cutler, and Ella M. Atkins. 2012. Cyber-physical challenges for space systems. In *Proceedings of the IEEE/ACM 3rd International Conference on Cyber-Physical Systems*. 45–52. https://doi.org/10.1109/ICCPS.2012.13

[36] Andreas Kurth, Alessandro Capotondi, Pirmin Vogel, Luca Benini, and Andrea Marongiu. 2018. HERO: An open-source research platform for HW/SW exploration of heterogeneous manycore systems. In *Proceedings of the 2nd Workshop on AutotuniNg and aDaptivity AppRoaches for Energy efficient HPC Systems*. ACM, 1–6. https://doi.org/10.1145/3295816.3295821

[37] Steven Leibson. 2023. NASA Recruits microchip, SiFive, and RISC-V to develop 12-core processor SoC for autonomous space missions. *EEJournal* (Jan. 2023). Retrieved from https://www.eejournal.com/article/nasa-recruits-microchip-sifive-and-risc-v-to-develop-12-core-processor-soc-for-autonomous-space-missions/

[38] Jiemin Li, Shancong Zhang, and Chong Bao. 2021. DuckCore: A fault-tolerant processor core architecture based on the RISC-V ISA. *Electronics* 11, 1 (Dec. 2021), 122. https://doi.org/10.3390/electronics11010122

[39] Krzysztof Marcinek and Witold A. Pleskacz. 2023. Variable delayed dual-core lockstep (VDCLS) processor for safety and security applications. *Electronics* 12, 2 (Jan. 2023), 464. https://doi.org/10.3390/electronics12020464

[40] Helmut Martin, Kurt Tschabuschnig, Olof Bridal, and Daniel Watzenig. 2017. Functional safety of automated driving systems: Does ISO 26262 meet the challenges? In *Automated Driving*, Daniel Watzenig and Martin Horn (Eds.). Springer International Publishing, Cham, 387–416. https://doi.org/10.1007/978-3-319-31895-0_16

[41] Fabio Montagna, Giuseppe Tagliavini, Davide Rossi, Angelo Garofalo, and Luca Benini. 2021. Streamlining the OpenMP programming model on ultra-low-power multi-core MCUs. In *Architecture of Computing Systems (Lecture Notes in Computer Science)*, Christian Hochberger, Lars Bauer, and Thilo Pionteck (Eds.). Springer International Publishing, Cham, 167–182. https://doi.org/10.1007/978-3-030-81682-7_11

[42] Vimal Reddy and Eric Rotenberg. 2007. Inherent time redundancy (ITR): Using program repetition for low-overhead fault tolerance. In *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*. IEEE, 307–316. https://doi.org/10.1109/DSN.2007.59

[43] D. Reinhardt, U. Dannebaum, M. Scheffer, and M. Traub. 2019. High performance processor architecture for automotive large scaled integrated systems within the european processor initiative research project. SAE Technical Paper 2019-01-0118, 2019. https://doi.org/10.4271/2019-01-0118

[44] Michael Rogenmoser, Nils Wistoff, Pirmin Vogel, Frank Gürkaynak, and Luca Benini. 2022. On-demand redundancy grouping: Selectable soft-error tolerance for a multicore cluster. In *Proceedings of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI'22)*. IEEE, Nicosia, Cyprus, 398–401. https://doi.org/10.1109/ISVLSI54635.2022.00089 ISSN: 2159-3477.

[45] Davide Rossi, Francesco Conti, Andrea Marongiu, Antonio Pullini, Igor Loi, Michael Gautschi, Giuseppe Tagliavini, Alessandro Capotondi, Philippe Flatresse, and Luca Benini. 2015. PULP: A parallel ultra low power platform for next generation IoT applications. In *Proceedings of the IEEE Hot Chips 27 Symposium (HCS'15)*. IEEE, 1–39. https://doi.org/10.1109/HOTCHIPS.2015.7477325

[46] Kashif Shahzad, Ayesha Kausar, Saima Manzoor, Sobia A. Rakha, Ambreen Uzair, Muhammad Sajid, Afsheen Arif, Abdul Faheem Khan, Abdoulaye Diallo, and Ishaq Ahmad. 2022. Views on radiation shielding efficiency of polymeric composites/nanocomposites and multi-layered materials: Current state and advancements. *Radiation* 3, 1 (Dec. 2022), 1–20. https://doi.org/10.3390/radiation3010001

[47] Jianhua Shi, Jiafu Wan, Hehua Yan, and Hui Suo. 2011. A survey of cyber-physical systems. In *Proceedings of the International Conference on Wireless Communications and Signal Processing (WCSP'11)*. 1–6. https://doi.org/10.1109/WCSP.2011.6096958

[48] Satyam Shukla and Kailash Chandra Ray. 2022. A low-overhead reconfigurable RISC-V quad-core processor architecture for fault-tolerant applications. *IEEE Access* 10 (2022), 44136–44146. https://doi.org/10.1109/ACCESS.2022.3169495

[49] Igor Silva, Otávio do Espírito Santo, Diego do Nascimento, and Samuel Xavier-de Souza. 2020. CEVERO: A soft-error-hardened SoC for aerospace applications. In *Proceedings of the Anais Estendidos do Simpósio Brasileiro de Engenharia de Sistemas Computacionais (SBESC'20)*. 121–126. https://doi.org/10.5753/sbesc_estendido.2020.13100

[50] Armen Toorian, Ken Diaz, and Simon Lee. 2008. The CubeSat approach to space access. In *Proceedings of the IEEE Aerospace Conference*. 1–14. https://doi.org/10.1109/AERO.2008.4526293

[51] J. A. Van Allen, G. H. Ludwig, E. C. Ray, and C. E. McILWAIN. 1958. Observation of high intensity radiation by satellites 1958 alpha and gamma. *J. Jet Propuls.* 28, 9 (Sept. 1958), 588–592. https://doi.org/10.2514/8.7396

[52] Eduardo Weber Wachter, Server Kasap, Xiaojun Zhai, Shoaib Ehsan, and Klaus McDonald-Maier. 2019. Survey of lockstep based mitigation techniques for soft errors in embedded systems. In *Proceedings of the 11th Computer Science and Electronic Engineering (CEEC'19)*. IEEE, 124–127. https://doi.org/10.1109/CEEC47804.2019.8974333

[53] A. Walsemann, M. Karagounis, A. Stanitzki, and D. Tutsch. 2023. STRV—A radiation hard RISC-V microprocessor for high-energy physics applications. *J. Instrument.* 18, 02 (Feb. 2023), C02032. https://doi.org/10.1088/1748-0221/18/02/C02032

[54] Junyong Wei and Suzhi Cao. 2019. Application of edge intelligent computing in satellite internet of things. In *Proceedings of the IEEE International Conference on Smart Internet of Things (SmartIoT'19)*. IEEE, 85–91. https://doi.org/10.1109/SmartIoT.2019.00022

[55] Nils-Johan Wessman, Fabio Malatesta, Stefano Ribes, Jan Andersson, Antonio García-Vilanova, Miguel Masmano, Vicente Nicolau, Paco Gomez, Jimmy Le Rhun, Sergi Alcaide, Guillem Cabo, Francisco Bas, Pedro Benedicte, Fabio Mazzocchetti, and Jaume Abella. 2022. De-RISC: A complete RISC-V based space-grade platform. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'22)*. IEEE, 802–807. https://doi.org/10.23919/DATE54114.2022.9774557 ISSN: 1558-1101.

[56] D. C. Wilkinson, S. C. Daughtridge, J. L. Stone, H. H. Sauer, and P. Darling. 1991. TDRS-1 single event upsets and the effect of the space environment. *IEEE Trans. Nuclear Sci.* 38, 6 (Dec. 1991), 1708–1712. https://doi.org/10.1109/23.124166

[57] Guoqi Xie, Gang Zeng, Jiyao An, Renfa Li, and Keqin Li. 2018. Resource-cost-aware fault-tolerant design methodology for end-to-end functional safety computation on automotive cyber-physical systems. *ACM Trans. Cyber-Phys. Syst.* 3, 1 (Sept. 2018), 4:1–4:27. https://doi.org/10.1145/3162052

[58] Ying Zhang and Krishnendu Chakrabarty. 2003. Fault recovery based on checkpointing for hard real-time embedded systems. In *Proceedings of the 18th IEEE Symposium on Defect and Fault Tolerance in VLSI Systems*. 320–327. https://doi.org/10.1109/DFTVS.2003.1250127 ISSN: 1550-5774.

[59] J. F. Ziegler and W. A. Lanford. 1979. Effect of cosmic rays on computer memories. *Science* 206, 4420 (1979), 776–788. https://doi.org/10.1126/science.206.4420.776

Received 8 March 2023; revised 14 November 2023; accepted 16 November 2023