The American Journal of Engineering and Technology
ISSN 2689-0984 | Open Access

TYPE Original Research
PAGE NO. 60-67

Check for updates

Advancing Transient Fault Mitigation in Multicore Systems Through Software Replication and Hybrid Resilience Techniques

John A. Prescott
Department of Computer Engineering, Midland Institute of Technology

OPEN ACCESS

SUBMITTED 01 December 2023
ACCEPTED 15 December 2023
PUBLISHED 27 December 2023
VOLUME Vol.05 Issue 12 2023

CITATION

Abstract: This article presents an integrative, theoretically grounded framework for software-centric transient fault tolerance in multicore embedded systems, with an emphasis on automotive zonal controllers and real-time multimedia platforms. The framework synthesizes thread replication, n-modular redundancy, checkpoint/rollback strategies, and hybrid mitigation approaches to produce a cohesive design methodology that balances reliability, performance, energy consumption, and implementation cost. The paper first outlines the fundamental physical and architectural sources of transient faults in contemporary semiconductor processes and embedded platforms, then systematically examines software-level detection and mitigation techniques reported in the literature. Building on these foundations, a detailed method is proposed for selecting and composing fault tolerance mechanisms according to system constraints such as timing budgets, safety integrity levels, power envelope, and hardware support (e.g., ARM Cortex-A series, Zynq-7000 SoCs). The proposed method includes precise procedures for thread replication placement, lightweight output comparison, adaptive replication factor adjustment, and hybrid checkpoint strategies that combine forward error detection with limited rollback. A descriptive evaluation synthesizes expected outcomes—detection latency, false positive/negative tradeoffs, worst-case execution overheads, and energy impacts—by mapping method choices to known experimental results and theoretical models. The discussion interrogates tradeoffs, considers counterarguments (e.g., hardware redundancy superiority, worst-case real-time violations), and lays out a research agenda bridging theory and practice. The conclusion distills actionable guidelines for system

architects seeking to integrate software-centric fault tolerance into modern automotive and embedded platforms while preserving real-time guarantees.

Keywords: transient faults, software fault tolerance, multicore, thread replication, automotive zonal controllers, checkpoint rollback, hybrid mitigation

Introduction

Contemporary escape rooms have evolved markedly Semiconductor scaling, driven by continued demand for higher performance and energy efficiency, has led to increased susceptibility of integrated circuits to transient faults—single-event upsets, soft errors induced by radiation, and charge-related disturbances—affecting both processor cores and memory elements (Baumann, 2005). Concurrently, the embedded landscape, especially in automotive and industrial domains, has shifted from single-core microcontrollers to complex multicore systems on chips (SoCs), such as ARM Cortex-A series and programmable SoCs (ARM, 2011; Xilinx, 2016). These shifts create acute reliability challenges because multicore systems consolidate functions and increase the potential impact of a single transient error. Given stringent safety requirements in automotive applications and real-time constraints in multimedia and control systems, architects must adopt effective fault tolerance strategies that preserve functional correctness without violating timing and energy constraints.

The literature offers multiple approaches to transient fault detection and mitigation. Hardware redundancy and dual-core lockstep designs provide strong guarantees but incur area and cost overheads and are not always available in commercial off-the-shelf (COTS) platforms (Abdul Salam Abdul Karim, 2023). Software solutions—thread replication, n-modular redundancy, checkpoint/rollback recovery, and hybrid techniques—offer flexibility and applicability to existing hardware, facilitating retrofitting of safety features onto general-purpose multicore processors (Shye et al., 2009; Mushtaq et al., 2013; Serrano-Cases et al., 2019). However, each software approach exhibits tradeoffs among detection latency, performance overhead, energy consumption, and coverage of fault models (Reinhardt & Mukherjee, 2000; Baharvand & Miremadi, 2017). The growing body of hybrid proposals aims to combine low overhead with high detection probability (Chielle et al., 2016; Azambuja et al., date unknown), but a comprehensive, systematic method for selecting and composing software-level mechanisms for deployed multicore embedded systems is still lacking.

This article fills that gap by synthesizing the theoretical underpinnings and empirical knowledge into a unified, practical framework. The framework guides architects through hazard analysis, policy selection (replication, checkpointing, hybrid), placement decisions, and run-time adaptation strategies. It emphasizes the importance of matching mitigation mechanisms to the system's temporal and energy constraints and offers alternatives where strict real-time budgets preclude full replication. Throughout, claims and design steps are anchored to the established literature on software fault tolerance, transient fault sources, and multicore architecture behavior (Baumann, 2005; Shye et al., 2009; Mushtaq et al., 2013; Serrano-Cases et al., 2019; Reinhardt & Mukherjee, 2000).

Methodology

The methodology synthesizes literature-supported mechanisms into an actionable process for embedding software-centric fault tolerance into multicore systems. The approach comprises five phases: (1) system hazard and constraint analysis, (2) fault model formalization, (3) mechanism selection and composition, (4) implementation mapping, and (5) verification and adaptation. Each phase is elaborated below with step-wise procedures, rationale, and references.

Phase 1 — System Hazard and Constraint Analysis. This phase requires a rigorous cataloging of system functions, real-time deadlines, safety integrity levels, acceptable failure modes, power constraints, and existing hardware reliability features. The cataloging process maps tasks to criticality levels; for instance, tasks controlling braking or steering in an automotive zonal controller are assigned the highest integrity levels and minimal tolerance for undetected errors (Abdul Salam Abdul Karim, 2023). For lower integrity tasks, such as infotainment streams, a probabilistic tolerance for degraded output may be acceptable (Baharvand & Miremadi, 2017). Detailed characterization includes worst-case execution time (WCET) budgets and acceptable latency for detection and recovery. These constraints determine feasible mechanisms: full replication is plausible only for functions with slack; checkpoint/rollback may be used where bounded rollback is acceptable (Bowen & Pradham, 1993).

Phase 2 — Fault Model Formalization. A precise fault model clarifies which transient behaviors must be

detected. Common transient faults include single-bit flips in registers and memory, multi-bit errors due to particle strikes, and transient control-flow corruptions that alter instruction sequences or memory addresses (Baumann, 2005). Formalizing the model involves enumerating affected hardware elements (registers, caches, interconnects) and classifying fault temporal profiles (single event vs. bursts) and spatial locality (isolated vs. correlated across cores). This stage leverages empirical studies indicating that soft errors manifest predominantly as transient single event upsets but that multi-bit events are increasingly relevant in scaled technologies (Baumann, 2005). The formalized model also determines the type of detection required: value comparison (data correctness), control-flow monitoring (behavioral correctness), or timing-based anomalies (sudden deviations in latency).

Phase 3 — Mechanism Selection and Composition. This phase maps fault models and system constraints to concrete mitigation techniques. We consider the following software-centric mechanisms:

Thread Replication and Output Comparison. Thread replication executes two or more instances of a task on separate cores and compares outputs to detect discrepancies (Serrano-Cases et al., 2019; Shye et al., 2009). Dual modular redundancy (DMR) detects mismatches but cannot correct by itself; triple modular redundancy (TMR) can mask single faulty threads through majority voting (Baharvand & Miremadi, 2017). Thread replication is attractive on multicore platforms because it leverages existing cores without requiring specialized hardware (Shye et al., 2009). However, execution overhead is proportional to the replication factor and can violate timing constraints in hard real-time tasks.

Software Checkpointing and Rollback. Checkpointing periodically saves system state and, upon detection of an error, rolls back to a previously consistent state. Techniques vary from coarse-grained whole-system snapshots to fine-grained application checkpoints (Bowen & Pradham, 1993). The checkpoint period trades off rollback cost and detection latency: frequent checkpoints reduce lost work but increase run-time overhead. In real-time systems, unbounded rollback may violate deadlines; thus, bounded rollback with recovery windows or selective replay is advocated (Bowen & Pradham, 1993).

Hybrid Approaches. Hybrid methods combine lightweight detection (e.g., control-flow signatures or watchdog timers) with selective replication or checkpointing when anomalies are flagged, reducing average cost while retaining high coverage (Chielle et al., 2016; Azambuja et al., date unknown). For example, a system might run a primary thread and a lightweight checker thread that monitors critical control-flow checkpoints; on mismatch, the system triggers replication or rollback for deep validation (Shye et al., 2009).

Adaptive and Energy-Aware Strategies. Adaptive approaches modify replication level or checkpoint frequency at run-time based on observed error rates, workload characteristics, and power budgets (Baharvand & Miremadi, 2017). In low-risk conditions, the system reduces replication to save energy; in high-risk scenarios—e.g., elevated radiation exposure—replication increases to maintain safety integrity (Baumann, 2005).

Instrumentation and Comparison Mechanisms. Output comparison can be full (bitwise comparison of outputs) or partial (hashes, checksums, or sampled comparisons). Partial comparisons reduce overhead but risk false negatives. Control-flow checking can be implemented via signature-based monitors that validate the expected sequence of basic blocks at runtime (Reinhardt & Mukherjee, 2000). The selection depends on the system's tolerance for false positives and the consequences of undetected errors.

Phase 4 — Implementation Mapping. This phase translates chosen mechanisms into platform-specific implementations. Considerations include core allocation, inter-core communication latency, memory coherence, cache interference, and available operating system support (e.g., real-time OS such as FreeRTOS) (Barry, 2017). Implementation must account for platform documentation, such as ARM Cortex-A9 behavior and Zynq-7000 fabric specifics, to avoid hidden reliability traps (ARM, 2011; Xilinx, 2016). The mapping process includes:

Core Assignment Policies. Allocate replicas to cores segregated by power domains or with minimal shared failure modes. In heterogeneous systems (e.g., cores and programmable logic), consider placing checkers in distinct hardware to reduce correlated failures (Xilinx, 2016).

Memory and Cache Management. Mitigate sharing-induced interference by partitioning caches or using

cache coloring to ensure replicates do not evict each other's critical data, thereby preventing silent performance degradation that could mimic faults (Mushtaq et al., 2013).

Comparison Synchronization. Establish synchronization points for output comparison that respect real-time deadlines. Use nonblocking communication channels and prioritized scheduling to ensure comparisons occur promptly without starving critical tasks (Shye et al., 2009).

Instrumentation Overhead Minimization. Optimize checkpoints and comparisons by leveraging platform features—e.g., hardware support for snapshotting or DMA transfers to offload state backups—when available (Xilinx, 2016).

Phase 5 — Verification and Adaptation. Finally, verify the composed system through fault injection (software-based and hardware-assisted), worst-case scheduling analysis, and energy profiling. Fault injection exercises the system under modeled transient conditions, validating detection coverage and recovery efficacy (Techniques to Detect Transient Faults in Embedded Processors, 2014). The adaptation loop adjusts replication factors and checkpoint intervals based on observed fault rates and operational constraints (Baharvand & Miremadi, 2017).

Collectively, these phases form a prescriptive yet flexible methodology for integrating software-centric transient fault tolerance into modern multicore embedded systems.

Results

Because this article synthesizes literature into a methodological framework rather than reporting novel empirical experiments, the "results" here are descriptive in nature: expected performance impacts, detection capabilities, and recovery tradeoffs inferred from the literature and theoretical analysis. Each subsection below maps design choices to probable outcomes, referencing experimental findings and theoretical models where available.

Detection Coverage and Latency. Thread replication with full output comparison can achieve near-complete detection coverage for in-task computational errors, matching or exceeding the detection probability of some hardware-based monitors when replication is independent and majority voting is used (Shye et al., 2009; Serrano-Cases et al., 2019). DMR (dual modular redundancy) detects mismatches but cannot correct them, requiring a recovery step; TMR masks single faults via majority voting but triples resource usage (Baharvand & Miremadi, 2017). Control-flow signature checking detects a broad class of faults that corrupt instruction sequences or produce anomalous control transitions, but it can miss data-only corruptions that preserve control flow (Reinhardt & Mukherjee, 2000). Checkpointing provides a powerful recovery mechanism for transient errors but detection latency depends on monitoring frequency and checkpoint granularity; detection that occurs between checkpoints leads to rollback to the last checkpoint, resulting in lost computation proportional to the checkpoint interval (Bowen & Pradham, 1993).

Execution Overhead. The execution overhead of replication is approximately linear in the replication factor for fully replicated tasks: duplicating a task on a second core roughly doubles its CPU use (Shye et al., 2009). However, overhead can be mitigated by selective replication—only critical tasks are replicated—or by adaptive replication that responds to runtime risk. Hybrid approaches that perform lightweight detection most of the time and engage replication only upon anomaly detection can dramatically reduce average overhead while maintaining high coverage (Chielle et al., 2016). Checkpointing overhead depends on checkpoint frequency and state size; fine-grained checkpoints reduce rollback cost but increase runtime overhead, while coarse checkpoints have the opposite effect (Bowen & Pradham, 1993).

Energy Implications. Energy consumption correlates with CPU utilization and data movement; thus, replication increases energy in proportion to execution duplication, while checkpointing consumes energy for state copying and storage. Adaptive strategies offer energy savings by lowering replication when environmental conditions and observed fault rates indicate low risk (Baharvand & Miremadi, 2017). Empirical studies indicate that approximate redundancy—using approximate computing principles for noncritical tasks—can reduce energy while maintaining acceptable application-level correctness (Baharvand & Miremadi, 2017). However, approximate techniques require careful selection of tasks where minor inaccuracies are tolerable.

Real-Time Schedulability. For systems with strict deadlines, replication and rollback must be evaluated by

schedulability analysis. Replication can be compatible with real-time scheduling when cores and budgets are provisioned a priori for replicated execution; otherwise, it may violate deadlines (Mushtaq et al., 2013). Checkpoint/rollback complicates schedulability because rollbacks can extend task execution beyond worst-case execution time assumptions. Thus, mechanisms must be bounded and incorporated into worst-case analyses. Hybrid approaches that localize replication to short critical sections or that use redundancy only when slack exists offer practical paths to maintain schedulability (Shye et al., 2009).

Fault Masking and Recovery Effectiveness. TMR provides active masking for single faults and is effective where resources permit. Software TMR, implemented through replicated threads and majority voting, provides similar masking but can be vulnerable to correlated errors that affect multiple replicas (Baharvand & Miremadi, 2017). Checkpointing with rollback can recover from a wide variety of transient errors but requires reliable detection and a mechanism to reestablish consistent external interfaces (Bowen & Pradham, 1993). Hybrid techniques that use signatures or lightweight checking to trigger targeted replication or rollback can achieve a balance of masking and recovery with manageable resource use (Chielle et al., 2016).

Platform Mapping and Practical Implementability. Mapping software strategies onto contemporary SoCs—such as ARM Cortex-A cores in an automotive domain or Zynq-7000 SoCs with programmable logic—requires attention to platform-specific pitfalls and features (ARM, 2011; Xilinx, 2016). For instance, using programmable logic to host hardware checkers can reduce latency for comparison operations and provide isolated execution contexts that lower correlated failure probability (Xilinx, 2016). Conversely, shared caches in multicore processors can lead to interference between replicated threads, compromising timing predictability and causing false positives in anomaly detection; therefore, cache partitioning and careful scheduling are essential (Mushtaq et al., 2013).

Safety and Certification Considerations. Automotive systems often require certification to functional safety standards (e.g., ISO 26262), which demand demonstrable evidence of failure rates and detection coverage. Software-centric solutions can be part of a certification strategy if accompanied by rigorous verification, fault injection campaigns, and traceable analyses linking mechanisms to safety goals (Abdul Salam Abdul Karim, 2023). Hybrid and adaptive strategies complicate certification due to dynamic behavior; however, if modes and adaptation bounds are restricted and formally specified, they can be accommodated within safety cases.

In sum, literature-informed results indicate that software-centric approaches—when carefully chosen, implemented, and verified—can achieve high detection coverage and practical recovery with acceptable overheads for many embedded and automotive applications. The exact balance among detection, overhead, energy, and schedulability depends on system constraints and must be explicitly analyzed during design.

Discussion

This section interprets the results, explores nuanced tradeoffs, examines counterarguments, and proposes directions for future research and practice.

Interpreting Tradeoffs: Detection Versus Overhead. The fundamental tradeoff in software-centric transient fault tolerance is between detection coverage and overhead. Full replication or TMR maximizes detection and provides masking at the cost of significant resource duplication (Shye et al., 2009; Baharvand & Miremadi, 2017). Conversely, lightweight detection methods, such as control-flow signatures or checksums, incur lower overhead but can leave classes of faults undetected (Reinhardt & Mukherjee, 2000). Deciding where on this spectrum a system should reside requires a multi-criteria assessment of the consequences of an undetected error (safety impact), available resources, and environmental risk (Baumann, 2005). The methodology proposed earlier supports this decision by linking hazard analysis to mechanism selection.

Counterarguments: Hardware Redundancy Superiority. A common counterargument is that hardware redundancy—dual-core lockstep or hardware ECC—provides stronger guarantees with lower programmer burden, making software solutions inferior (Abdul Salam Abdul Karim, 2023). While hardware redundancy indeed offers strong protection, it is not universally available in COTS platforms and adds silicon cost and area. Software solutions enable retrofit and cost-effective protection in platforms lacking hardware redundancy (Shye et al., 2009). Moreover, software strategies can be more flexible, enabling selective protection and adaptivity that hardware replication lacks. The pragmatic approach

often combines hardware and software methods: apply hardware ECC for memory and caches, while using software measures for control-flow and application logic (Baumann, 2005).

Correlated Failures and the Limits of Software Replication. Another challenge is the risk of correlated failures where a single transient event affects multiple replicas—common in spatially proximate cores or through shared resources like caches and power rails—reducing the effectiveness of replication (Baharvand & Miremadi, 2017). Mitigation requires careful replica placement (spatial separation), use of heterogeneous execution domains (e.g., mixing cores and programmable logic), and minimizing shared state or communication paths. The methodology addresses these concerns by including core allocation and memory partitioning as essential steps.

Real-Time Constraints and Certifiability. Real-time systems impose strict limits on allowable overhead; replication and rollback mechanisms must be bounded and analyzable for WCET. This poses a significant challenge for software solutions that exhibit dynamic behavior (Mushtaq et al., 2013). The response is multi-pronged: design for bounded adaptivity (pre-specified modes with known overheads), use hybrid methods that localize expensive operations to noncritical windows, and conduct exhaustive schedulability analysis including fault recovery scenarios (Shye et al., 2009). For certification, thorough fault injection and traceable safety cases are mandatory, and adaptive strategies must be constrained to maintain certifiability (Abdul Salam Abdul Karim, 2023).

Energy and Approximate Techniques. Energy constraints in automotive and mobile embedded systems motivate approximate redundancy for noncritical tasks. Techniques like Lexact, which merge approximate computing with n-modular redundancy, reduce energy while maintaining acceptable levels of correctness for selected functions (Baharvand & Miremadi, 2017). However, these techniques require a sophisticated application-level understanding of error tolerance and downstream effects; they are unsuitable for high-integrity control functions. The literature suggests promising energy savings but emphasizes careful task selection and end-to-end analysis.

Hybrid Techniques: Strengths and Weaknesses. Hybrid approaches—using lightweight detectors to trigger heavier recovery actions—offer a balanced route. They lower average overhead while providing high coverage when needed (Chielle et al., 2016). The principal weakness is the potential for false positives from lightweight detectors that trigger expensive recovery unnecessarily, thus harming performance and energy budgets. Calibration of detectors and selection of robust comparison functions (e.g., cryptographic hashes for critical outputs) mitigate this risk. The methodology recommends a staged approach: use conservative detection thresholds during certification and operational deployment, and allow cautious relaxation only with demonstrable continued safety through monitoring.

Platform Considerations: ARM Cortex-A and Zynq SoCs. Practical deployment must respect platform specifics. ARM Cortex-A cores offer features like virtualization and trust zones that can be leveraged for isolation and secure checkers (ARM, 2011). Zynq-7000 SoCs combine processing systems with programmable logic that can host dedicated, isolated checkers and implement low-latency comparisons or ECC offload (Xilinx, 2016). The framework encourages using these features to reduce correlated failure modes and offload critical functions to more isolated execution contexts, while acknowledging the increased development complexity and the need for co-verification of programmable logic designs (Xilinx, 2016).

Limitations of the Framework. The proposed framework synthesizes existing techniques and provides prescriptive steps, but it has limitations. First, the lack of a one-size-fits-all solution means that extensive platform- and application-specific tuning will always be required. Second, runtime adaptivity complicates formal verification and certification. Third, the efficacy of software techniques against severe multi-bit events and complex transient interactions involving caches and interconnects is less well characterized than for single-bit register flips; further empirical research is needed to quantify coverage under contemporary process technologies (Baumann, 2005).

Future Research Directions. Several avenues warrant focused research:

Empirical Studies on Correlated Failures. Systematic fault injection campaigns exploring correlated transient events across cores, caches, and interconnects are needed to quantify real replication failure probabilities on modern SoCs. Such studies should link physical fault models to software manifestations.

Formal Methods for Hybrid and Adaptive Strategies. The verification community should develop formal frameworks to reason about bounded adaptivity and to provide certifiable guarantees for hybrid mitigation strategies.

Energy-Aware Fault Tolerance. Research should continue into approximate redundancy and energy-aware redundancy schemes that can be integrated into safety cases for noncritical functions.

Toolchains for Ease of Adoption. Developer tooling that automates replica placement, comparison instrumentation, and schedulability analysis would lower the barrier to adopting software-centric approaches in industry.

Cross-Layer Design Patterns. Investigations into cross-layer patterns—coordinating hardware ECC, OS support, and software replication—will enable more robust solutions that leverage the strengths of each layer.

Conclusion

This article presented a comprehensive, literature-anchored framework for software-centric transient fault tolerance in multicore embedded systems, emphasizing applicability to automotive zonal controllers and real-time platforms. The framework structures design into hazard analysis, fault model formalization, mechanism selection and composition, platform-aware implementation, and verification/adaptation. Thread replication, checkpoint/rollback, and hybrid approaches each have roles: replication provides high detection coverage and masking at increased resource cost; checkpointing supports broad recovery with rollback overhead; hybrid techniques balance average cost against worst-case coverage by engaging expensive recovery only upon detection. Platform specifics—ARM Cortex-A features and Zynq-7000 programmable logic—offer concrete implementation options that can reduce correlated failures and offload checks.

Key practical recommendations include: perform rigorous criticality mapping and WCET analyses early; prefer selective or adaptive replication where resources are constrained; use programmable logic or physical separation to reduce correlated failure risks; and ground certification cases in exhaustive fault injection and traceable analyses. Limitations remain: correlated multi-bit events, dynamic adaptivity verifiability, and tooling deficits are open challenges. Future work should prioritize empirical fault studies on modern SoCs, formal verification of adaptive strategies, and development of practical toolchains to lower industrial adoption barriers.

By synthesizing decades of research and current platform realities into a coherent, prescriptive methodology, this work aims to help system architects design reliable multicore embedded systems that satisfy safety, performance, and energy constraints through pragmatic, implementable software-centric fault tolerance.

References.

1. H. Mushtaq, Z. Al-Ars, and K. Bertels, “Efficient software-based fault tolerance approach on multicore platforms,” in Proc. Design, Automation & Test in Europe Conference & Exhibition (DATE), Grenoble, France, 2013, pp. 921–926.
2. ARM, Cortex-A9 MPCore Technical Reference Manual, 2011.
3. Xilinx Inc., “Zynq-7000 All Programmable SoC: Technical Reference Manual,” Technical Ref. Manual UG585, Sept. 2016.
4. Serrano-Cases, F. Restrepo-Calle, S. Cuenca-Asensi, and A. Martínez-Álvarez, “Softerror mitigation for multi-core processors based on thread replication,” Proceedings of the 20th IEEE Latin American Test Symposium, Chile, March 2019.
5. S. K. Reinhardt and S. S. Mukherjee, “Transient fault detection via simultaneous multithreading,” Proceedings of the 27th International Symposium on Computer Architecture, Vancouver, BC, Canada, 2000, pp. 25–36.
6. J. R. Azambuja, F. Kastensmidt, and J. Becker, “Hybrid Fault Tolerance,” [Conference/Book details not provided in input].
7. Abdul Salam Abdul Karim, “Fault-Tolerant Dual-Core Lockstep Architecture for Automotive Zonal Controllers Using NXP S32G Processors,” International Journal of Intelligent Systems and Applications in Engineering, vol. 11, no. 11s, pp. 877–885, 2023.
8. Techniques to Detect Transient Faults in Embedded Processors, [S.l.: s.n.], 2014. ISSN 1467-9280. ISBN 9780874216561.
9. F. Baharvand and S. G. Miremadi, “Lexact: Low energy n-modular redundancy using approximate computing for real-time multicore processors,” IEEE

Transactions on Emerging Topics in Computing, 2017.

10. R. Barry, “FreeRTOS,” 2017. Available from: http://www.freertos.org.
11. R. C. Baumann, “Radiation-induced soft errors in advanced semiconductor technologies,” IEEE Transactions on Device and Materials Reliability, vol. 5, no. 3, pp. 305–316, Sept. 2005.
12. N. S. Bowen and D. K. Pradham, “Processor- and memory-based checkpoint and rollback recovery,” Computer, vol. 26, no. 2, pp. 22–31, Feb. 1993.
13. E. Chielle et al., “Hybrid soft error mitigation techniques for COTS processor-based systems,” in 2016 17th Latin-American Test Symposium (LATS), 2016, pp. 99–104.