Dependability in Embedded Systems: A Survey of Fault Tolerance Methods and Software-Based Mitigation Techniques

IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 24 October 2024, accepted 27 November 2024, date of publication 2 December 2024,
date of current version 10 December 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3509633

SURVEY

Dependability in Embedded Systems: A Survey of Fault Tolerance Methods and Software-Based Mitigation Techniques

MOHAMMADREZA AMEL SOLOUKI[1], (Member, IEEE), SHAAHIN ANGIZI[2], (Senior Member, IEEE), AND MASSIMO VIOLANTE[1], (Member, IEEE)
[1]Department of Control and Computer Engineering, Politecnico di Torino, 10129 Turin, Italy
[2]Department of Electrical and Computer Engineering, New Jersey Institute of Technology, Newark, NJ 07102, USA

Corresponding author: Mohammadreza Amel Solouki (mohammadreza.amelsolouki@polito.it)

ABSTRACT Fault tolerance is a critical aspect of modern computing systems, ensuring correct functionality in the presence of faults. This paper presents a comprehensive survey of fault tolerance methods and mitigation techniques in embedded systems, with a focus on both software and hardware faults. Emphasis is placed on real-time embedded systems, considering their resource constraints and the increasing interconnectivity of computing systems in commercial and industrial applications. The survey covers various fault tolerance methods, including hardware, software, and hybrid redundancy. Particular attention is given to software faults, acknowledging their significance as a leading cause of system failures, while also addressing hardware faults and their mitigation. Moreover, the paper explores the challenges posed by soft errors in modern computing systems. The survey concludes by emphasizing the need for continued research and development in fault tolerance methods, specifically in the context of real-time embedded systems, and highlights the potential for extending fault tolerance approaches to diverse computing environments.

INDEX TERMS Embedded systems, fault tolerance, reliability, analytical redundancy, dependability.

I. INTRODUCTION

A key facet of fault tolerance is ensuring the continued correct operation of modern computing systems despite internal faults. This paper focuses on fault tolerance in both software and hardware contexts, aiming to increase overall system dependability. In a fault-tolerant system, the goal is to facilitate seamless transitions to alternative modules and thereby sustain service provision in the face of faults, by either concealing faults or detecting errors. This comprehensive survey addresses methods for handling both software and hardware faults, highlighting their individual and combined impact on system reliability.

Failures arise when the behavior of a running system diverges from the system's expected behavior. Failures are caused by errors, while faults are the underlying cause of errors. Not all faults necessarily lead to errors, and a single fault can precipitate multiple errors. Similarly, a solitary error can culminate in multiple failures. Redundancy, in some form, is an essential component across all fault tolerance approaches to ensure the system's capacity to withstand faults. Redundant devices, networks, data, or applications are leveraged based on the fault class at hand.

The importance of fault tolerance extends to various domains, including automotive, aerospace, medical devices, and smart grids, each presenting unique challenges and requirements. For instance, in aerospace systems, fault tolerance is crucial to ensure the safety and reliability of aircraft operations, where failures can have catastrophic consequences. Medical devices also demand high reliability to prevent life-threatening malfunctions during critical health procedures. Similarly, smart grids require robust fault tolerance to maintain the stability and efficiency of power distribution networks, preventing widespread outages and ensuring consistent energy supply.

The primary objective of fault tolerance is to increase system dependability. To fulfill this aim, fault-tolerant systems must uphold specified service delivery, even amidst

The associate editor coordinating the review of this manuscript and approving it for publication was Zhaojun Steven Li.

component faults [1]. Novel technologies elevate various facets of our quality of life while concurrently bolstering societal productivity and efficiency. In the realm of automotive systems, the trajectory of emerging technological trends accentuates the introduction of novel features, expanding the array of onboard embedded systems and processors [2]. Within the automotive domain, these systems are engineered to optimize energy consumption, enrich user experiences through infotainment support, and institute autonomous and semi-autonomous control mechanisms encompassing methods like cruise control and autonomous piloting [3].

Both automotive and industrial production domains represent paradigmatic instances of safety-critical applications, wherein any functional malfunction of the supporting equipment, machinery, or devices could trigger dire repercussions, spanning critical injuries, fatalities, substantial property damage, or extensive environmental harm [4]. Consequently, the intricate electronic devices now integrated within these systems must rigorously adhere to safety, reliability, and security imperatives to ensure the seamless operation of the entire system.

Within the automotive domain, prominent corporations have invested substantial capital in new technologies to implement and broaden their applicability across various automotive functions. These applications encompass the development of diverse levels of vehicular autonomy driven by the benefits to user safety, security, traffic latency reduction, and energy efficiency. However, these technological advantages concurrently present various challenges yet to be definitively resolved. Established methodologies for designing and developing secure and safe devices could be repurposed for these novel applications. The automotive and autonomous machinery domains presently exploit innovative technologies, including Artificial Intelligence (AI) and computer vision, providing a distinct advantage in effecting more streamlined procedures. However, this trend introduces the ability for contemporary devices to integrate intricate algorithms, augmenting application complexity and imposing substantial constraints concerning real-time operation, available power resources, and performance thresholds [2], [3], [4].

In practice, the development of modern safety-critical applications hinges upon three core elements: *i)* robust high-performance operation and power efficiency, *ii)* cost-effectiveness, and *iii)* unwavering safety and reliability [5]. However, it is important to note that some safety-critical applications prioritize low-performance operation to maintain reliability. There are inherent trade-offs between these three core elements. For example, in certain medical devices, maintaining lower performance can reduce the risk of overheating and improve the overall reliability of the device. Similarly, in aerospace systems, the balance between performance and safety is critical, with some systems operating at lower performance levels to ensure maximum reliability and safety. These trade-offs must be carefully managed to meet the specific requirements of each application.

Various studies [6], [7], [8], [9], [10] have demonstrated that devices constructed using cutting-edge technologies are inherently susceptible to an array of faults manifesting during initial operational stages and, with greater frequency, throughout their active lifespan. These faults may arise from two primary sources: *(i)* inherent defects stemming from manufacturing processes or component fatigue, and *(ii)* environmental or external influences [11]. In the former case, device faults might emanate from manufacturing anomalies that evade detection during end-of-production testing, thereby precipitating unforeseen behaviors during operational life cycles. Furthermore, components within a device are predisposed to degradation (e.g., electro-migration or gate-oxide effects) following prolonged operation or even during periods of idleness (e.g., idle operational mode) [12], [13], [14]. Conversely, external influences also exert sway over device operation. Environmental factors temporarily or permanently alter electrical parameters, resulting in transient fault effects that impinge upon ongoing device applications. These fault effects propagate across the device as transient soft errors, which occur due to external factors like radiation or EMI, temporarily affecting hardware states during application execution. Exposure to high-energy particles (triggering radiation effects) or Electromagnetic Interference (EMI) increases device vulnerability to transient faults, disrupting the electronic charge of one or more storage components within the device and toggling the state of transistors employed for data storage. As this data courses through the circuitry, multiple errors can arise within the application. In the most extreme instances, external interventions can lead to permanent damage to the device. Soft errors, such as Single Event Upsets (SEUs) and Single Event Transients (SETs), occur when external radiation particles strike the microelectronic circuits, causing temporary changes in logic states without permanent damage to the hardware. These faults differ from permanent hardware failures and must be handled in real time to maintain system reliability.

Fault tolerance methods focus on detecting and recovering from faults, regardless of their types, to ensure the correct functioning of the system. To achieve a given reliability target, one commonly used fault tolerance technique is the utilization of redundancy, in terms of hardware, software, information, and time, exceeding what is normally required for system operation. Hardware redundancy techniques involve adding extra hardware components to detect or tolerate faults. For example, multiple cores or processors can be utilized instead of a single one, with each application being executed on a separate core/processor, enabling fault detection and even correction. Another technique, time redundancy, allocates extra time to perform system functions and detect faults, without violating the timing constraints of real-time systems. The re-execution technique is an example of time redundancy, where a faulty task is repetitively

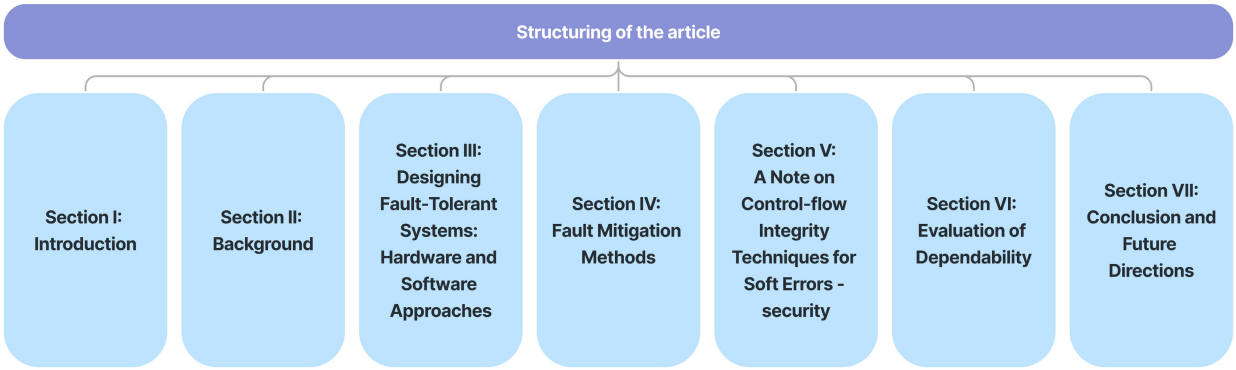

FIGURE 1. Structuring of the article.

executed on the same hardware until the correct output is obtained. Information redundancy techniques, such as error detection and correction coding, are commonly used in memory units, storage devices, and data communication to ensure reliability. Redundant Arrays of Independent Disks (RAIDs) are another example of information redundancy, where data is organized and stored in multiple configurations to enhance reliability. Additionally, software redundancy involves adding extra software to detect and tolerate faults. For example, N-version programming involves separate groups of programmers designing and coding a software module multiple times, reducing the likelihood of the same mistake occurring in all versions. Checkpointing stores the last fault-free state of a process in stable memory, allowing the system to roll back to that state and re-execute the application in case of a fault. By employing these fault tolerance methods, systems can ensure reliable functioning despite the occurrence of faults.

In this survey, we systematically explore fault tolerance methods and software-based mitigation techniques, focusing specifically on embedded systems with resource constraints such as limited memory and low-end computation environments. The fault tolerance techniques discussed were selected based on their demonstrated effectiveness in mitigating faults in embedded systems. The selection criteria emphasized techniques that address resource-constrained environments, ensuring that the methods are applicable in systems with limited power, memory, and processing capabilities. We evaluated each method for its fault detection coverage, implementation overhead, and ease of integration into real-world systems. Special attention was given to techniques proven to be adaptable across domains such as automotive, aerospace, and industrial applications. By taking a broad view of fault tolerance, we aim to provide insights into methods applicable across diverse fault types, with special attention to the resource constraints common to embedded environments. The surveyed methods are categorized based on their effectiveness in addressing hardware and software faults, as well as their adaptability to the unique constraints of embedded environments. Furthermore, a comparative analysis is conducted, evaluating key factors such as performance overhead, fault coverage, and complexity of implementation.

The organization of this paper, illustrated in Figure 1, is as follows: the background is reviewed in Section II. Section III explores various hardware and software approaches to designing fault-tolerant systems. Section IV explores various fault mitigation methods. Section V discusses control-flow integrity techniques for soft errors and security. Section VI explores the evaluation of dependability in fault-tolerant systems. Finally, the conclusion is provided in Section VII.

The key insights from each section are succinctly summarized in Table 1, providing a quick reference for readers to grasp the essential content discussed throughout this survey.

II. BACKGROUND

This section provides an overview of the different defects or upsets leading to permanent, intermittent, or transient faults. Specifically, the Single Event Effects (SEEs) are discussed with a focus on SEUs. Figure 2 visually represents the content discussed in this section.

Faults are logical-level abstractions of physical defects or upsets. In other words, faults describe the changes in device logic function caused by a defect or upset. Therefore, faults are defined here as any variation from the expected logical behavior of the underlying hardware. Faults can be further categorized as transient, intermittent, or permanent. Transient faults occur and then soon disappear. They manifest effects that can occur for a short period during the component's lifetime. Intermittent faults are characterized as a fault occurring, then vanishing, and then reoccurring, and so on. Examples of intermittent faults are signal interference, such as cross-talk between connections or communication lines. Permanent faults manifest and exist within the system until the defective component is repaired or replaced. These faults commonly occur due to manufacturing defects or physical damage to CMOS gates due to high charges. It is also possible that the device's electrical properties may mask some defects or upsets, causing no faults to appear.

Radiation exposure can result in both permanent defects and temporary upsets in electronic systems, affecting their behavior and reliability. This is particularly relevant for

TABLE 1. Summary of sections.

Section	Content and Key Points
Background	Register File, Integer Unit (IU) and Floating Point Unit (FPU), Bus Unit, Control Unit, Debug Unit, Instruction Cache, Data Cache; Impact of SEUs on different system components.
Designing Fault-Tolerant Systems	Hardware-Based (Prevention Strategies, Hardware Monitors, Hardware Redundancy), Software-Based (Design Diversity, Single-Design Approaches), Hybrid-Based; Description of fault tolerance techniques, including prevention strategies, monitors, redundancy methods, and integration of hardware and software approaches.
Fault Mitigation Methods	Fault Mitigation Techniques, Control Flow Checking (CFC) (Mechanisms, Automotive Applications), Sensitivity Analysis for Multi-bit Faults, AI Methods for Dependability, Repetition Execution, Lockstep; Overview of mitigation methods and detailed exploration of control flow checking and AI applications.
Control-flow Integrity Techniques	Control-flow Integrity (CFI) for Soft Errors and Security, Enhanced Security Measures, Data Integrity; Overview of CFI techniques to prevent security breaches due to soft errors, importance of distinguishing faults from attacks, additional security measures like IDS, redundancy, cryptography, access control, and audits.
Evaluating Dependability	Fault Injection, Simulation, Diagnosis Techniques, Models of Soft Errors; Exploration of techniques for evaluating dependability in fault-tolerant systems, summary of soft error models for fault injection.

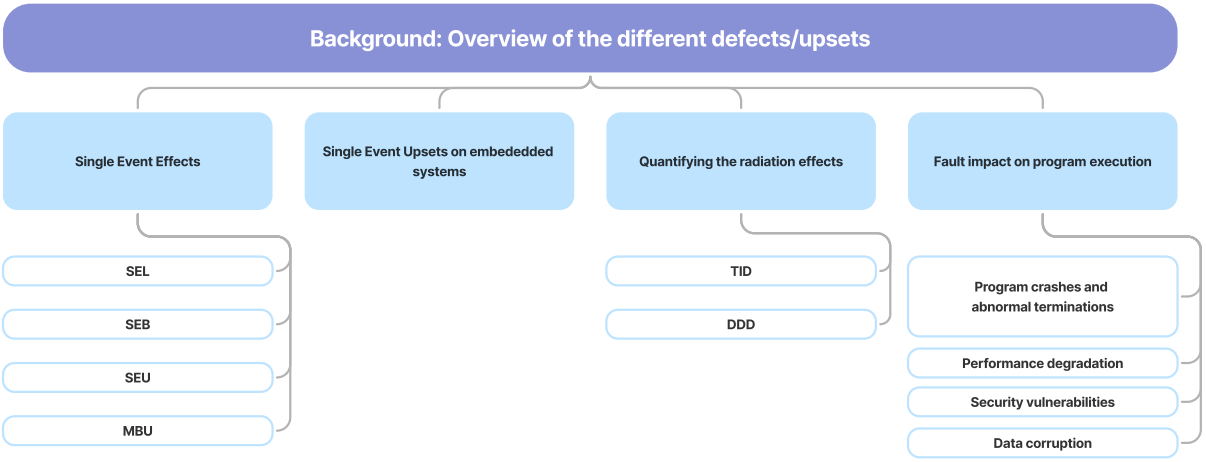

FIGURE 2. Overview of the different defects/upsets.

embedded systems used in aerospace, medical devices, and other high-radiation environments. By understanding and mitigating radiation effects, we can enhance the dependability of embedded systems in these critical applications.

A. INTRODUCTION TO SINGLE EVENT EFFECTS

One of the most important radiation effects is Single-Event Effects (SEEs), which are a result of the interaction between an energetic particle and a semiconductor device, leading to various manifestations. SEEs are typically caused by the deposition of charge by the particle or by the creation of a current pulse. The amount of charge or current required to cause a SEE depends on the device's type and the materials used in its construction [15].

When discussing SEEs, it is important to differentiate between permanent and transient effects [16]. Transient SEEs are temporary alterations in a device's state brought about by particle passage. These changes can stem from charge deposition or the creation of a current pulse. Transient effects refer to radiation-induced interference that ceases once the radiation dissipates. This temporary interference can involve variations in electrical signals, electronic device hardening, or system upsets that affect performance without causing permanent damage. Transient SEEs typically vanish within a few milliseconds and have no lasting impact on the device. One of the transient SEEs is the SET, which is a temporary change in the state of a device caused by the passage of a single energetic particle. SETs typically disappear after a few milliseconds, but they can cause errors in data processing.

On the other hand, permanent SEEs refer to changes in the device state that are caused by a particle's passage, resulting in irreversible damage that hampers proper functioning. Such effects encompass alterations in the semiconductor material's physical properties or degradation of circuitry, leading to long-lasting malfunctions or failures. Examples of permanent SEEs include Single-Event Latch-up (SELs), Single-Event Burnouts (SEBs), Single-Event Upsets (SEUs), and Multi-Bit Upsets (MBUs) [17].

SEL is a type of SEE where a high-current path is formed in the device, leading to permanent damage. SEL occurs when a parasitic thyristor structure within the device is inadvertently triggered, creating a low-impedance path and leading to excessive current flow that can cause permanent damage. SEB is a type of SEE caused by a single energetic particle passage, like a high-energy ion or neutron, resulting in irreversible damage. This phenomenon occurs when the particle deposits excessive energy, leading to localized

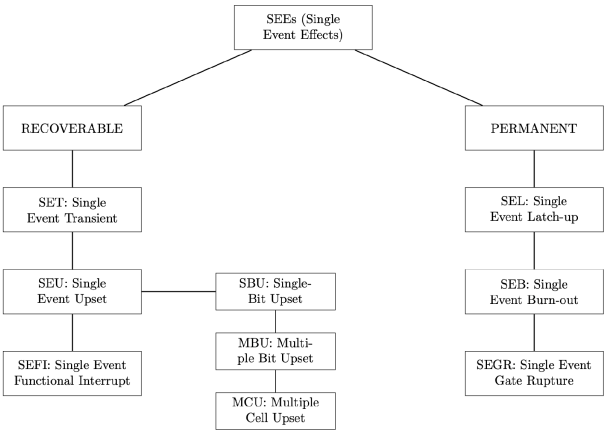

FIGURE 3. Classification of Single-Event Effects (SEEs).

heating and damage within the device's structure. This can result in a sudden increase in current or voltage, causing permanent damage or burnout. SEU is a permanent change in the state of a memory cell or register caused by the deposition of energy in the semiconductor material by a single energetic particle. It can cause a single bit to be flipped from a 0 to a 1 or vice versa [18], [19]. MBU refers to the SEU of multiple memory cells in close proximity caused by a single energetic particle. MBUs are less common than SEUs, but they can be more serious.

To conclude, SEEs are significant radiation-induced phenomena resulting from particle interactions with semiconductor devices. They can be categorized into transient effects, which are temporary disruptions with no lasting impact, and permanent effects, causing irreversible damage to devices. Examples of permanent SEEs include SELs, SEBs, SEUs, and MBUs, each with specific consequences for device functionality. Figure 3 shows the classification chart, providing a visual representation that enhances the comprehension of SEEs.

B. THE IMPACT OF SINGLE EVENT UPSETS ON EMBEDDED SYSTEMS

This subsection delves into the intricate details of SEUs, encompassing their diverse impacts on various components of embedded systems. In the realm of fault tolerance, it's crucial to distinguish between Silent Data Corruption (SDC) and Single Event Functional Interrupt (SEFI). SDC encompasses errors in memory and the final application output, while SEFI, a severe issue causing system hangs or crashes, directly impacts application execution and user experience. Most fault tolerance methods primarily target SDC, leaving SEFI, which can significantly disrupt system operation, relatively unaddressed [20], [21].

Figure 4 in the context of SEUs, it provides a visual representation of how these events affect different components.

However, it's imperative to elaborate on SDC and SEFI for clarity. SDC occurs when memory or final application output is corrupted, leading to inconsistencies in results. In contrast, SEFI, resulting from errors in control flow, leads to application crashes and processor hang. Only a few techniques can effectively detect both SDC and SEFI, such as those employing the lockstep principle based on redundancy to enhance processor dependability. SEUs can affect data flow or control flow in processors, influencing data flow errors or SEFIs [22].

Moving forward, this subsection primarily focuses on SEUs in detail. In the Register File, SEUs can corrupt data and cause errors in the application outputs, leading to inconsistencies in the results. If an SEU impacts a control register, it can result in errors in the execution flow of the program, and the system freezes. SEUs in the IU and FPU can lead to incorrect computations due to the pipelining in these arithmetic units. In the Bus Unit, bit flips in the embedded registers responsible for latching addresses and data can cause incorrect read or write operations. The Control Unit, which implements complex algorithms, may experience SEUs that trigger exception generation or disrupt the sequence. SEUs can also affect the Debug Unit, activating special execution modes and causing errors in the program's execution flow.

Moving on to the Instruction Cache, SEUs can result in corrupted outputs or processor freezes. The instruction caches typically consist of an SRAM array for storing fetched instructions and a tag array for validating or invalidating the fetched program. SEUs in the tag array can invalidate an instruction to be executed, leading to a cache miss and introducing a delay in program execution as the instruction needs to be fetched again. If an SEU validates an incorrect code, it can crash the program's flow. Additionally, an SEU can corrupt an instruction in the SRAM array. If the tag array validates this corrupted code, a wrong instruction will be executed, or an exception will be generated if the corrupted instruction is no longer part of the processor instruction set. However, if the tag array does not validate the corrupted instruction, the fault is masked, and no incorrect behavior is observed. The Instruction Cache section in Figure 4 captures these dynamics. Similar to instruction caches, Data Caches also consist of a tag array and a data array. Bit flips in the tag array can validate outdated data, resulting in incorrect outputs or invalidate data, causing delays (cache miss) in the application. If an SEU affects the data array, it can corrupt the output. However, if the data is outdated, the fault is masked, and no effects are observed [23].

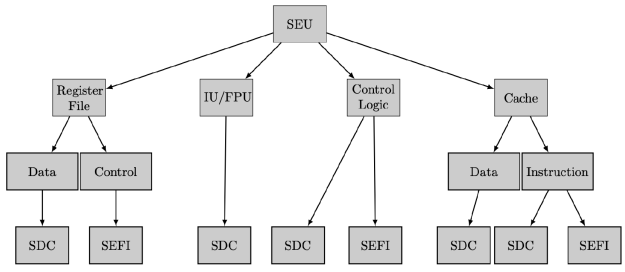

FIGURE 4. The impact of single error upsets on different parts of a processor [24].

C. QUANTIFYING THE RADIATION EFFECTS

To quantify the effects of radiation on electronic devices, various measurements have been developed. This article will discuss the most commonly used measures: Total Ionizing Dose (TID), and Displacement Damage Dose (DDD) [10], [23]. Total Ionizing Dose (TID) is the total amount of ionizing radiation absorbed by an electronic device over time. This radiation can lead to permanent damage, such as changes in the electrical properties of the semiconductor material, degradation of the circuitry, or an increase in leakage currents. TID is measured in units of rads (radiation absorbed dose) or grays (Gy). The amount of TID that a device can withstand before failing depends on the type of device and the materials used in its construction. TID can cause permanent damage to a device by creating defects in the semiconductor material. These defects can reduce the conductivity of the material or create hot spots that can lead to thermal breakdown. DDD is the amount of non-ionizing radiation that causes permanent damage to the crystal lattice of the semiconductor material. It can lead to changes in material properties, which can affect the performance of electronic devices. DDD is measured in units of Displacements Per Atom (DPA). The amount of DDD that a device can withstand before failing depends on the type of device and the materials used in its construction. DDD can cause permanent damage to a device by creating defects in the semiconductor material. These defects can reduce the conductivity of the material or create hot spots that can lead to thermal breakdown.

D. FAULT IMPACT ON PROGRAM EXECUTION

Understanding the impacts of faults on program execution is crucial for designing fault-tolerant embedded systems. Faults in such systems can lead to undesired outcomes, including program crashes, incorrect outputs, and compromised system functionality. In this survey, we focus on common faults affecting embedded systems, including those caused by power failures, communication errors, Electromagnetic Interference (EMI), and radiation-induced soft errors. Though these faults have different origins, they share common fault-tolerant methods in detection and recovery.

- Program Crashes and Abnormal Terminations: One of the primary consequences of faults in program execution is program crashes and abnormal terminations. Faults such as hardware failures, memory corruption, or unhandled exceptions can cause the program to terminate abruptly or enter an undefined state, resulting in system instability and potential data loss [1]. Researchers have proposed various techniques for detecting and recovering from program crashes, including CFC methods that verify the integrity of program execution path
- Incorrect Outputs and Results: Faults can lead to incorrect outputs and results, affecting the reliability and accuracy of embedded systems. Logic errors, data corruption, or faulty computations can result in incorrect data processing and decision-making, leading to undesirable consequences [25]. To mitigate the impact of such faults, researchers have explored techniques such as redundant computation, error-correcting codes, and diverse redundancy approaches.
- Performance Degradation: Faults in embedded systems can also cause performance degradation, resulting in decreased system efficiency and responsiveness. Resource management errors, such as memory leaks and inefficient scheduling, can lead to performance bottlenecks [26]. Researchers have investigated methods such as dynamic resource allocation and optimized scheduling algorithms.
- Security Vulnerabilities: Faults in embedded systems can introduce security vulnerabilities, jeopardizing the confidentiality, integrity, and availability of sensitive data. Faults such as input validation flaws, buffer overflows, or insecure communication protocols can be exploited by attackers to gain unauthorized access or perform malicious activities [27]. Researchers have proposed security-oriented fault mitigation methods, including secure coding practices, encryption algorithms, and intrusion detection systems.
- Data Corruption: Faults in embedded systems can lead to data corruption, compromising the reliability and integrity of stored data. Faults such as power failures, communication errors, or hardware malfunctions can result in data inconsistencies or loss [28]. To mitigate data corruption, researchers have explored techniques such as checksums, error detection and correction codes, and redundant storage mechanisms.

In summary, understanding the effects of faults on program execution is essential for designing fault-tolerant embedded systems. By considering the potential consequences of faults, researchers can develop effective fault mitigation methods.

III. DESIGNING FAULT-TOLERANT SYSTEMS: HARDWARE AND SOFTWARE APPROACHES

This section aims to investigate a range of hardware and software techniques for designing fault-tolerant systems, with a focus on their adaptation to embedded systems.

In the context of embedded systems, these methods must be adapted to account for the limited resources and specific operational requirements. For instance, hardware redundancy in embedded systems must balance fault tolerance with power and space constraints. Software redundancy techniques need to be lightweight to avoid overwhelming the limited processing capabilities of embedded devices. Hybrid approaches that integrate both hardware and software techniques provide an optimal solution for achieving system reliability within the resource constraints of embedded systems. Software Implemented Hardware Fault Tolerance (SIHFT) is an example of such hybrid methods; in SIHFT, software-based techniques are implemented to address hardware faults, reducing overhead while ensuring fault coverage [29].

The choice of fault tolerance approach depends on the specific application's requirements and constraints. For

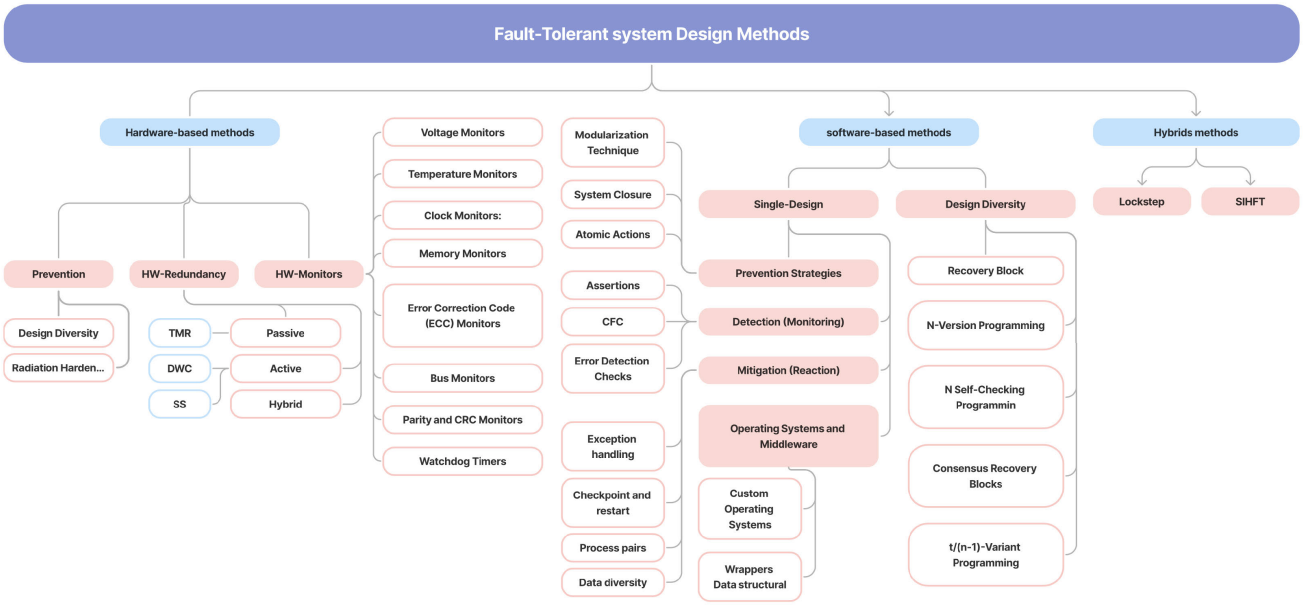

FIGURE 5. Fault-tolerant system design methods.

instance, a system that requires high availability, such as a telecommunications network, would typically opt for hardware-based fault tolerance methods. Conversely, a less critical system like a word processing application may utilize software-based fault tolerance methods [1], [30].

When selecting a fault tolerance method, several factors must be considered:

- *Application Criticality*: Critical applications such as aerospace, medical devices, and industrial control systems often require the highest levels of reliability and may therefore rely more heavily on hardware-based fault tolerance methods to ensure minimal downtime and maximum fault coverage.
- *Resource Availability*: Embedded systems often operate under stringent resource constraints, including limited processing power, memory, and energy. While hardware-based methods provide robust fault tolerance, they are often resource-intensive. In such cases, software-based methods like SIHFT may be more appropriate due to their adaptability to resource constraints [29].
- *Performance Overhead*: Fault tolerance mechanisms can introduce performance overhead. Hardware-based methods might have less performance impact compared to software-based methods, which could significantly slow down the system. Hybrid methods aim to strike a balance by combining the strengths of both approaches. For instance, combining SIHFT with a CFC module or integrating hardware Intellectual Property (IP) for CFC with SIHFT provides enhanced fault detection with minimal performance penalties, ensuring robust system integrity [29].
- *Cost Considerations*: The implementation cost of fault tolerance methods varies. Hardware redundancy can be expensive due to the need for additional components. Software-based methods typically incur lower costs as they primarily require additional development effort rather than additional hardware components. Hybrid methods may provide a cost-effective solution by combining elements of both hardware and software fault tolerance.
- *Scalability and Flexibility*: Software-based fault tolerance methods offer greater flexibility and scalability as they can be updated and scaled with less physical modification compared to hardware-based methods. This is particularly beneficial for systems expected to evolve over time or require frequent updates.

By considering these factors, system designers can make informed decisions on the most appropriate fault tolerance method for their specific application needs.

This section provides an overview of the advantages, limitations, and trade-offs of hardware, software, and hybrid fault-tolerant techniques. Table 2 summarizes these comparisons for embedded systems. Figure 5 illustrates the organization of this section, providing a visual representation of the fault-tolerant system design methods discussed. While fault tolerance, fault prevention, and fault forecasting address different aspects of system dependability, they are included in this taxonomy to provide a comprehensive view of the methods aimed at ensuring reliable system operation. Fault tolerance focuses on managing and recovering from faults after they occur, while fault prevention aims to avoid faults altogether, and fault forecasting predicts potential future faults to mitigate their impact. Although these approaches differ in their objectives and implementation stages, they complement each other in improving system reliability.

subsection III-A concentrates on Hardware-based fault tolerance methods. subsection III-B extensively discusses

Software-based Fault Tolerance methods, specifically the Single-Design Software Fault Tolerance methods. Lastly, subsection III-C presents Hybrid methods that incorporate a blend of hardware and software methods.

A. HARDWARE BASED FAULT TOLERANCE METHODS

Hardware-based techniques in fault detection and correction can be categorized into three main strategies: *prevention strategies* (aim to avoid faults before they occur by designing robust systems), *hardware monitoring as a detection strategy* (involve identifying faults as they occur to initiate appropriate corrective actions), and *hardware redundancy mitigation strategies* (focus on minimizing the impact of faults after they occur).

In embedded systems, these techniques must be designed to fit within the constraints of limited space, power, and computational capacity. The balance between fault tolerance and resource utilization is crucial to ensure the feasibility and effectiveness of these methods in embedded applications.

Hardware-based fault tolerance techniques involve incorporating redundancy at the hardware level to detect and correct faults. These methods often include fault detection circuits, error-correcting codes, and redundant hardware components.

Hardware monitors include methods like Built-In Self-Test (BIST) and Watchdog Timers. BIST allows the system to test its own components, while Watchdog Timers can detect and respond to system failures by resetting the system if it becomes unresponsive [31], [32].

Redundancy-based methods include techniques such as Triple Modular Redundancy (TMR), where three identical components work in parallel, and majority voting is used to decide the correct output. Other methods include N-Modular Redundancy (NMR) and redundancy at different levels such as logic gates, processors, and entire systems [33], [34].

These methods can be further enhanced through integration with software-based techniques like SIHFT. For example, combining hardware redundancy with SIHFT allows efficient fault detection while minimizing resource overhead. The SIHFT+Hardware IP for CFC approach strengthens the system's ability to detect and correct control-flow errors, enhancing both hardware fault detection and software-based error mitigation [29].

These methods are crucial for ensuring fault tolerance but must be optimized for embedded systems' specific requirements. For example, redundancy techniques must manage trade-offs between fault coverage and resource constraints, while monitoring methods must be designed to operate efficiently within the system's limited resources [1], [35].

The choice of hardware-based fault tolerance techniques depends on the specific application and constraints of the embedded system, balancing the need for fault detection and mitigation with the limitations of space, power, and computational resources.

1) PREVENTION

Prevention strategies aim to avoid faults before they occur by designing robust systems [36], [37]. Techniques include:

- *Design Diversity*: Utilizing multiple design versions to ensure that a single fault does not affect all versions. This can be applied in hardware through techniques such as diverse circuit designs to handle specific tasks [36].
- *Radiation Hardening*: Designing hardware components to be resistant to radiation-induced faults, which is especially important in aerospace and high-radiation environments [37].

2) HARDWARE MONITORS

Hardware monitors are specialized components or circuits integrated into a system to monitor the behavior of various components and signals continuously. These monitors are designed to detect anomalies, errors, or deviations from expected behavior, which could indicate the presence of faults or defects in the system [34]. By actively monitoring the system's operation in real-time, hardware monitors can provide early warnings and trigger appropriate actions to prevent or mitigate the effects of faults before they lead to system failures.

While hardware monitors offer quick detection, they can still result in increased power consumption and chip area, which may limit their use in resource-constrained systems.

There are several types of hardware monitors, each serving a specific purpose:

- *Voltage monitors*: These monitors supervise the supply voltage levels of critical components. If the voltage falls outside specified limits, it might indicate a fault or power-related issue [38].
- *Temperature monitors*: Monitoring the temperature of components is essential to prevent overheating and thermal damage. Temperature sensors and monitoring circuits can trigger alerts or take actions to cool down the system if temperatures become excessive [39], [40].
- *Clock monitors*: These monitors oversee clock signals to ensure proper timing and synchronization between components. Deviations in clock frequencies or signal integrity can lead to faults.
- *Memory monitors*: Monitoring memory operations helps identify errors in data storage or retrieval, which is crucial for maintaining data integrity [41].
- *Error Correction Code (ECC) Monitors*: ECC monitors detect and correct errors in memory or data storage systems using error-correcting codes, thereby enhancing data reliability [42].
- *Bus monitors*: These monitors supervise data and control buses for communication errors between components. Detecting bus errors can prevent data corruption or incorrect communication [40], [43].
- *Parity and CRC monitors*: These monitors use parity or Cyclic Redundancy Check (CRC) codes to detect data corruption during transmission or storage [44].

- *Watchdog timers*: Watchdog timers are hardware-based timers that must be periodically reset by the system's software. If the software fails to reset the timer within a specified time frame, the watchdog timer assumes a fault has occurred and initiates a system reset [45].

Hardware monitors work in conjunction with redundancy-based techniques to provide comprehensive fault detection and tolerance mechanisms. They contribute to creating fault-tolerant systems that can identify, isolate, and recover from faults, ultimately enhancing the overall reliability and availability of critical electronic systems.

3) HARDWARE REDUNDANCY

Redundancy-based techniques rely on hardware or time redundancy. These techniques involve the addition of extra hardware components to detect or tolerate faults. such as watchdog processors [46], checkers [47], or Infrastructure Intellectual Properties (I-IP) [48].

TMR and NMR are effective methods for fault detection but come with high costs and significant resource usage, as indicated in Table 2. For embedded systems, the challenge lies in balancing redundancy with space and power limitations.

Hardware redundancy can be implemented through passive, active, and hybrid methods.

Passive redundancy techniques use M-of-N systems where N components are present, and correct system operation is achieved when at least M components work correctly. For instance, Triple Modular Redundancy (TMR) is a 2-of-3 system, meaning it consists of three components performing the same action, and the result is voted on to determine the correct output [1], [35].

Active redundancy techniques include Duplication With Comparison (DWC), Standby-Sparing (SS), pair-and-a-spare, and watchdog timers. DWC involves parallel execution of two identical hardware components, with the output being compared to detect faults. However, DWC can only detect faults and not tolerate them. Standby-sparing utilizes one operational module and one or more spare modules. If a fault is detected in the main component, it is omitted from operation, and the spare component takes over. Pair-and-a-spare is a combination of DWC and SS techniques, where two modules are executed in parallel, and their results are compared to detect faults [1], [35].

Hybrid redundancy techniques integrate features from both active and passive hardware redundancies. Examples of hybrid redundancy include N modular redundancy with spare, sift-out modular redundancy, self-purging redundancy, and triple duplex architecture. Self-purging redundancy is based on NMR with spare techniques, where all modules actively participate in the system function. Sift-out modular redundancy utilizes special circuits, such as comparators, detectors, and collectors, to configure N identical modules in the system. Triple duplex architecture combines DWC with TMR to detect faulty modules and remove them from the system. These hardware-based techniques, although effective in fault detection and tolerance, come with a high cost, verification and testing time, area overhead, and increased power consumption [1], [35].

In summary, hardware-based fault detection and correction techniques fall into three main categories: prevention, hardware monitoring as a detection strategy, and hardware redundancy mitigation strategies.

Prevention strategies aim to avoid faults before they occur by designing robust systems. Techniques such as design diversity and radiation hardening are employed to ensure that faults do not occur or are minimized. Design diversity involves using multiple design versions to prevent a single fault from affecting all versions, while radiation hardening makes hardware components resistant to radiation-induced faults, which is especially critical in high-radiation environments like aerospace. Detection strategies involve continuously monitoring system components and signals to detect anomalies and provide early warnings, preventing faults from leading to system failures. Hardware monitors play a crucial role here, including voltage, temperature, clock, memory, ECC, bus, and parity/CRC monitors, as well as watchdog timers. These monitors ensure real-time detection of potential issues, allowing for prompt corrective actions. Mitigation strategies focus on minimizing the impact of faults after they occur. Redundancy techniques are key in this category, involving the addition of extra hardware components to detect and correct faults. These methods can be passive (such as M-of-N systems), active (including techniques like Duplication with Comparison (DWC) and Standby-Sparing (SS)), or hybrid (combining features of both passive and active methods). Examples include Triple Modular Redundancy (TMR), N-Modular Redundancy (NMR), sift-out modular redundancy, self-purging redundancy, and triple duplex architecture.

These methods are crucial for ensuring fault tolerance but must be optimized for embedded systems' specific requirements. Redundancy techniques must manage trade-offs between fault coverage and resource constraints, while monitoring methods must be designed to operate efficiently within the system's limited resources. Considerations such as verification, testing, area overhead, and power consumption are essential in the optimization process.

While effective, these hardware-based techniques come with costs such as increased verification and testing time, area overhead, and power consumption. Therefore, the choice of hardware-based fault tolerance techniques depends on the specific application and constraints of the embedded system, balancing the need for fault detection and mitigation with the limitations of space, power, and computational resources.

B. SOFTWARE BASED FAULT TOLERANCE METHODS

Software-based fault tolerance techniques are critical in maintaining system reliability, especially in embedded systems where resource constraints and dedicated functionalities necessitate efficient and effective solutions. These techniques must be optimized to function within the limited

computational and memory resources available in embedded environments.

Software-based methods offer greater flexibility and lower cost compared to hardware-based techniques but may introduce performance overhead, as shown in Table 2.

Software based Fault Tolerance methods can be divided into two categories: design diversity-based and single-design software fault tolerance. In design diversity-based methods, multiple diverse versions of a software module are created, often using different algorithms or programming languages. These versions run concurrently, and discrepancies are detected and resolved through voting mechanisms, enhancing reliability. On the other hand, single-design software fault tolerance focuses on enhancing the robustness of a single software design through techniques such as error detection, error handling, and recovery mechanisms.

1) DESIGN DIVERSITY BASED SOFTWARE FAULT TOLERANCES

Design diversity-based or multiple-version-based software fault tolerance involves using multiple versions or variants of software, either executed sequentially or in parallel. These versions are used as alternatives, with separate means of error detection, and can be implemented in pairs or larger groups for replication checks or masking through voting. The main idea is that components built differently should fail differently, so if one version fails on a specific input, at least one alternate version should be able to produce the correct output. This section explores various approaches to software reliability and safety through design diversity. However, ensuring the independence of failure among multiple versions and developing effective output selection algorithms are critical challenges in deploying multi-version software fault tolerance techniques.

Design diversity serves as a means of protection against uncertainty, specifically, design faults and their associated failure modes in software design. The objective of applying design diversity techniques to software design is to build program versions that fail independently and with a low probability of coincidental failures. Achieving this objective greatly reduces or eliminates the probability of encountering incorrect outputs during program execution. However, due to the complexity of software, the application of design diversity for software fault tolerance is currently more of an art than a science.

The concept of multiple-version software design was pioneered by Algirdas Avizienis and his team at UCLA in the 1970s, primarily focusing on software. Their research also explored the application of design diversity concepts to other system aspects such as the operating system, hardware, and user interfaces. Even with rigorous development and proper application of design diversity, there is still the issue of identical input profiles leading to common errors. Experiments have shown that error manifestations are not equally distributed across the input space, and the probability of coincident errors is influenced by the chosen inputs. Data diversity techniques can potentially mitigate this issue, but quantifying their effectiveness remains a challenge.

An important consideration in using multi-version software is the cost involved. Replicating the entire development effort, including testing, would be expensive. In some cases, where only certain parts of the functionality are safety-critical, applying design diversity only to those critical parts can reduce development and production costs. Reference [49] highlights the need to address the problem of identical input profiles as a common source of errors, highlighting that experiments have indicated unequal distribution of error manifestations across the input space. While data diversity techniques may reduce the impact of this error source, quantifying their effectiveness remains a challenge.

In summary, design diversity-based or multiple-version-based software fault tolerance offers a means of enhancing software reliability and safety by using multiple versions of software with independent failure properties. However, challenges exist in ensuring independence from failure and developing suitable output selection algorithms. The concept of design diversity has evolved as an art in software fault tolerance, with applications extending beyond software to other system aspects. The issue of identical input profiles leading to common errors requires attention, and while data diversity techniques may mitigate this, quantifying their effectiveness remains a challenge. The cost of using multi-version software is an important consideration, and selectively applying design diversity to critical parts can help reduce development and production costs.

In this study, we explore various fault tolerance approaches in software that incorporate design diversity, both with multiple versions and a single design. The approaches we focus on are as follows:

- *The Recovery Block Scheme*: The Recovery Block Scheme (RBS) combines the checkpoint and restart approach with multiple versions of a software component [50]. Before execution, checkpoints are created to allow for recovery after detecting errors. This ensures a valid operational starting point for the next version if an error is detected. Additionally, embedded checks are used to enhance error detection. The primary version executes more frequently compared to alternates, which are designed for degraded performance. Multiple versions can be executed sequentially or in parallel, depending on processing capability and desired performance. In the event that all alternates fail, the component must raise an exception to communicate its failure to the system.
- *The N-Version Programming Scheme*: The N-Version Programming Scheme (NVPS) is a multiple-version technique where all versions fulfill the same basic requirements, and the correctness of output decisions relies on comparing all outputs [51]. A voter selects the correct output, eliminating the need for an acceptance test based on the application. Developing NVPS requires considerable effort as all versions must adhere to the

same conditions, resulting in complexity comparable to creating a single version. Designing the voter can be challenging and may involve inexact voting. Different voters, such as the Formalized Majority Voter, Generalized Median Voter, Formalized Plurality Voter, and Weighted Averaging Techniques, can be used, with weights based on the application and individual versions' features.

- *The N Self-Checking Programming Scheme*: The N Self-Checking Programming Scheme (NSCPS) combines various structural variations of Recovery Blocks and N-Version Programming using multiple software versions [52]. Independent development of versions and acceptance tests based on shared requirements are used in this technique. NSCPS utilizes separate acceptance tests for each version, distinguishing it from the Recovery Blocks approach. The technique benefits from using an application-independent decision algorithm for selecting the correct output.
- *The Consensus Recovery Blocks Scheme*: The Consensus Recovery Blocks Scheme (CRBS) combines N-Version Programming and Recovery Blocks to achieve higher reliability compared to either approach individually [53]. The acceptance test in Recovery Blocks techniques lacks guidance and may have design faults, whereas voters in N-Version Programming can be unsuitable in certain cases. CRBS incorporates the first layer of decision-making using a similar algorithm to that of N-Version Programming. If the first layer declares a failure, the second layer, which utilizes acceptance tests similar to Recovery Blocks, is invoked. Although more complex than the individual techniques, CRBS has the potential to deliver a more reliable result.
- *The t/(n-1)-Variant Programming Scheme*: The t/(n-1)-Variant Programming Scheme (VPS) involves n variants and the t/(n-1) diagnosability measure to restrict faulty units to a subset of size at most (n-1), assuming a maximum of t faulty units. This approach differs from the previous methods in terms of the method used to isolate faulty units [54].

In summary, the utilization of design diversity-based or Multiple-Version-Based software fault tolerance techniques offers promising avenues to enhance software reliability and safety. These approaches leverage multiple versions of software, designed to fail independently, thereby reducing the likelihood of encountering erroneous outputs during program execution. However, the practical implementation of design diversity in software fault tolerance remains more of an art than a science due to the complexity of software and the challenges in ensuring independence from failure. Additionally, addressing the issue of identical input profiles leading to common errors and quantifying the effectiveness of data diversity techniques remain significant challenges. Furthermore, the cost implications of employing multi-version software must be carefully considered, and selective application of design diversity to critical components can help mitigate development and production expenses. The various fault tolerance approaches explored, such as the Recovery Block Scheme, N-Version Programming Scheme, N Self-Checking Programming Scheme, Consensus Recovery Blocks Scheme, and t/(n-1)-Variant Programming Scheme, provide diverse strategies to implement design diversity effectively in software fault tolerance.

2) SINGLE-DESIGN SOFTWARE FAULT TOLERANCE APPROACH

Single-design fault tolerance involves introducing redundancy to a single version of the software to detect and recover from faults. This subsection categorizes these methods into prevention, detection (monitoring), and mitigation (reaction) strategies to provide a clear and structured overview.

a: PREVENTION STRATEGIES

Prevention strategies aim to reduce the likelihood of faults through robust software design.

- *Modularization Techniques*: Utilizing modular decomposition with built-in protections to prevent abnormal behavior from propagating to other modules. Control hierarchy issues, such as visibility and connectivity, are considered to minimize the risk of uncontrolled system state corruption [55].
- *System Closure*: Ensuring that no action is allowed unless explicitly authorized, which is crucial for maintaining system integrity [55].
- *Atomic Actions*: Implementing activities where components interact exclusively with each other without interacting with the rest of the system. These actions provide error confinement and recovery capabilities [55].

b: DETECTION (MONITORING) STRATEGIES

Detection strategies involve identifying faults during execution to trigger corrective actions.

- *Assertions*: Logical statements inserted at different points in a program to reflect relationships between program variables [56].
- *Control Flow Checking (CFC)*: Partitioning the application program into Basic Blocks (BBs) and computing deterministic signatures for each block. Faults are detected by comparing the runtime signature with a precomputed one [55].
- Error Detection Checks: These checks can be located within modules or at their outputs and include [1], [55]:
  - *Replication Checks*: Comparing outputs of matching components, making them suitable for multi-version software fault tolerance.
  - *Timing Checks*: Detecting deviations from acceptable module behavior.
  - *Reversal Checks*: Computing inputs from outputs and comparing them to actual inputs.
  - *Coding Checks*: Checking relationships between actual and redundant information.

  - *Reasonableness Checks*: Detecting errors based on semantic properties of data.
  - *Data Structural Checks*: Inspecting properties of data structures, such as the number of elements, links, and pointers. Augmenting data structures with redundant structural data can enhance the effectiveness of structural checks.
  - *Runtime Checks*: Runtime Checks are standard error detection mechanisms in hardware systems and can be used as fault detection tools [54].

c: MITIGATION (REACTION) STRATEGIES

Mitigation strategies focus on recovering from faults to ensure continued operation [54].

- *Exception Handling*: Interrupting normal operations to handle abnormal responses. Exceptions are signaled by error detection mechanisms, and the design of exception handlers requires consideration of possible triggering events and appropriate recovery actions.
- *Checkpoint and Restart*: Commonly used recovery method where the system periodically saves its state (checkpointing) and can revert to this state (restart) upon encountering a fault.
  - *Static Restart*: Returns the module to a predetermined state.
  - *Dynamic Restart*: Uses dynamically created checkpoints.
- *Process Pairs*: Using two identical versions of software running on separate processors, with recovery managed through checkpoint and restart mechanisms. The primary processor actively processes input and creates output while generating checkpoint information for the backup processor. Upon error detection, the secondary processor loads the last checkpoint and takes over the primary processor's role [54].
- *Data Diversity*: Implementing ''input sequence workarounds'' and using different input re-expressions on each retry to enhance the success rate of checkpoint and restart procedures. Data diversity works hand in hand with the Process Pairs technique, allowing for different re-expressions of the input in the primary and secondary [57].

d: OPERATING SYSTEMS AND MIDDLEWARE

In the context of operating systems, software fault tolerance is crucial for ensuring proper application-level software functioning. Techniques include:

- *Custom Operating Systems*: Developed with structured design processes and advanced verification techniques for safety-critical applications [54].
- *Wrappers*: Middleware between the operating system and application software to monitor the flow of information and prevent undesirable values from propagating. Wrappers provide application-transparent fault tolerance functionality [58].

In conclusion, single-design software fault tolerance methods encompass prevention, detection, and mitigation strategies. Prevention strategies, such as modularization and system closure, aim to reduce the likelihood of faults [55]. Detection strategies use various checks and assertions to identify faults during execution [55], [56]. Mitigation strategies focus on recovering from faults using methods like checkpoint and restart, process pairs, and data diversity [54], [57]. Additionally, operating system techniques, such as custom operating systems and wrappers, enhance fault tolerance in mission-critical applications [54], [58]. While these methods offer advantages like the absence of additional auxiliary devices and good expansibility, they come with significant time and space overhead due to the inclusion of redundant instructions, impacting program performance.

C. HYBRID METHODS

Hybrid fault tolerance techniques combine both hardware and software approaches to enhance system reliability. For embedded systems, hybrid techniques provide a balanced solution by integrating the strengths of hardware and software methods while considering the system's resource limitations. One of the most prominent examples of a hybrid approach is SIHFT, which combines software-based fault detection with hardware redundancy to reduce both resource overhead and performance impact [29]. These techniques must be carefully designed to optimize fault detection and correction without imposing excessive overhead on the embedded system's limited resources.

Hybrid fault tolerance methods typically involve the integration of a SIHFT method with a hardware module designed to perform consistency checks within the processor. In a study by [59], SIHFT techniques are combined with a CFC module, which is responsible for monitoring the trace port of the processor. Another hybrid approach, proposed by [60], is known as Hybrid Error-detection Technique using Assertions (HETA). This method utilizes a watchdog module and assertions (or signatures) to address control-flow errors.

Lockstep is another hybrid fault tolerance technique that utilizes both software and hardware redundancy for error detection and correction [21], [61], [62], [63]. Lockstep involves executing the same application simultaneously and symmetrically in two identical processors. These processors are initialized to the same state and receive identical inputs during system start-up. During normal operation, the state of both processors should be identical at each clock cycle. By monitoring the processor's data, addressing, and controlling buses [64], a checker module periodically compares the outputs of the processors to check for inconsistencies. To enforce verification, specific points are inserted in the program to indicate when the application execution should be locked and the outputs compared. If any discrepancies are found, the lockstep system leverages a rollback method to restore the processors to a safe state. In the absence of errors, a checkpoint operation is performed, which stores the context

TABLE 2. Enhanced overview of techniques for addressing faults in embedded systems.

Technique	Fault Type Covered	Pros	Cons	Common Applications
Hardware	Hard and Soft Errors	- High fault detection - Fast fault detection - No software modification - High reliability	- High area and power overhead - High cost - Limited scalability	Aerospace, critical medical devices
Software	Mainly Soft Errors	- High flexibility - Low hardware overhead - Scalable - Some error correction	- High performance overhead - Focuses on single fault models - Increased development complexity	Embedded systems with limited resources
Hybrid	Hard and Soft Errors	- High detection efficiency - Can correct both soft and hard errors - Balance between performance and reliability	- Requires both software and hardware modifications - High design complexity - Can introduce performance and cost overhead	Mission-critical embedded systems, automotive

of the processor (including registers and main memory) in a secure memory location. Memories can be protected using ECC to prevent data corruption. ECC is capable of detecting and correcting single-bit errors and detecting double-bit errors. To recover from errors, the fault-free copy of the processor's context is retrieved from memory using the rollback method. The processor is then recovered to a state without errors and restarts the application execution from this point.

In summary, hybrid fault tolerance methods combine software and hardware approaches to enhance error detection and correction. Methods that integrate SIHFT with hardware techniques, such as watchdog timers or Hardware IP for CFC, provide an effective solution for enhancing fault detection and recovery while minimizing system overhead. One approach integrates SIHFT with CFC or Hybrid Error-detection Technique using Assertions (HETA) to monitor and address control-flow errors. For instance, combining SIHFT with watchdog timers allows for efficient system monitoring and fault detection, while the SIHFT+Hardware IP for CFC hybrid approach enhances control flow consistency and robustness [29]. Another hybrid method, known as Lockstep, executes applications in parallel on identical processors, comparing outputs and employing rollback and checkpoint mechanisms to ensure system reliability and error recovery. These hybrid approaches provide robust fault tolerance in critical systems.

Table 2 provides a comprehensive classification of techniques for addressing Random Hardware Failures (RHFs) in embedded systems. The techniques are categorized into three main types: hardware-based, software-based, and hybrid-based approaches. In the hardware category, these techniques offer advantages such as high fault detection rates, fast detection capabilities, and the absence of software modifications. However, they come with notable drawbacks, including the inability to correct detected errors, a predominant focus on a single fault model, substantial area and power overhead, implementation restricted to the physical level, and potentially high cost. Software-based techniques, on the other hand, boast high fault detection rates, flexibility, and minimal hardware modifications, with some capable of error correction. Nevertheless, they incur drawbacks such as high-performance overhead, a predominant focus on a single fault model, and concentration on either data or control flow, but not both. Hybrid techniques aim to combine the strengths of hardware and software approaches, achieving high fault detection efficiency, small area overhead, and the capability to detect and, in some cases, correct both SDC and SEFI. However, hybrid techniques also have their challenges, including the potential for high performance or area overhead and necessitating both software and hardware modifications. The insights provided by Table 2 pave the way for a nuanced understanding of the strengths and limitations associated with each category of techniques, facilitating informed decisions in selecting and implementing RHF mitigation strategies in embedded systems [24].

In summary, this section investigates a range of hardware and software techniques for designing fault-tolerant systems. The choice of fault tolerance approach depends on the specific application and can involve hardware-based, software-based, or hybrid methods. Each method addresses faults at different levels, ensuring comprehensive coverage of potential issues in embedded systems.

IV. FAULT MITIGATION METHODS

This section explores the most common fault mitigation methods applied to maintain system reliability during operation. While section III focused on the design-phase strategies for fault tolerance, this section addresses techniques implemented post-design to manage and mitigate faults as they occur in real-time. These methods are crucial for handling unexpected faults that escape the initial fault-tolerant design or arise due to unforeseen conditions.

In particular, transient faults such as soft errors, which are caused by external factors like cosmic radiation and electromagnetic interference, require specific mitigation techniques. ECC and radiation hardening are commonly used to address these soft errors, ensuring that systems can recover from temporary bit flips in hardware without lasting damage. These techniques are especially important for embedded systems exposed to harsh environments where radiation is a significant concern.

It is important to note that this section includes both primary fault mitigation methods and supporting techniques. The primary methods are those that directly detect, manage, or recover from faults during system operation, ensuring continued system operation and integrity. In addition to these primary methods, supporting techniques—such as Sensitivity Analysis—play a vital role in evaluating system vulnerabilities and guiding the improvement of fault tolerance mechanisms. While supporting techniques do not mitigate faults directly, they provide the foundation for designing and refining more effective fault mitigation strategies.

subsection IV-A discusses CFC, which ensures the correct execution sequence of instructions within a program, addressing both transient and permanent faults. Various CFC methods are examined, including AUTOSAR standards, techniques for handling permanent and transient faults, and CFI techniques for soft errors and security. subsection IV-C addresses Sensitivity Analysis for Multi-bit Faults at the Block Level, focusing on the challenges and methods for analyzing and mitigating the impact of multi-bit faults within basic blocks of embedded systems. This includes techniques such as fault injection, statistical analysis, and formal verification to enhance system reliability. subsection IV-D discusses AI Methods for Enhancing Dependability in Embedded Systems, highlighting how AI techniques, such as machine learning, deep learning, and reinforcement learning, are used to improve fault detection, diagnosis, prediction, and mitigation. These methods enable adaptive and self-healing capabilities, enhancing the overall robustness and reliability of embedded systems. subsection IV-E explains Repetition Execution, which involves retrying operations to handle transient faults. This technique is particularly useful for temporary errors that may be resolved upon re-execution. subsection IV-F describes Lockstep techniques, which synchronize operations across multiple processors or cores, comparing their outputs to detect and correct errors. This method is effective for ensuring consistency and reliability in critical applications.

The organization of this section reflects the balance between fault mitigation methods and the supporting techniques that enable their refinement and evaluation, as visually represented in Figure 6.

To provide a clearer comparison of the fault mitigation methods discussed, Table 3 summarizes the key advantages and limitations of each approach, highlighting critical trade-offs in terms of performance, resource requirements, and detection capabilities.

A. IN-DEPTH EXAMINATION OF CONTROL FLOW CHECKING METHODS

Various techniques have been proposed in the literature to address transient and permanent faults in different parts of a system, targeting both hardware and software components and relying on different forms of redundancy. Among these techniques, CFC stands out as it can cover faults affecting memory components containing the executable program, as well as the hardware components handling the program and its flow [65]. CFC has been suggested to handle reliability issues for both transient and permanent faults [66], [67], and more recently, it has been applied to address security issues caused by the injection of malicious faults [68], [69]. As suggested in [65], while CFC methods provide robust protection against control-flow errors, they are not without limitations. Specifically, [65] highlights that existing CFC techniques introduce additional vulnerabilities. For example, software-only CFC protection schemes increase system vulnerability by 18-21%, with performance overheads ranging from 17-38%. Hardware-based CFC methods, although slightly more effective, incur higher area and power costs, making them less viable in resource-constrained environments [65]. These findings suggest that while CFC techniques remain useful for detecting faults, their integration into modern systems should carefully balance the trade-offs between detection accuracy and system performance.

While security threats such as hacking, malware deployment, and unauthorized access can introduce malicious faults, these issues are distinct from the natural causes of faults discussed in this paper, such as radiation-induced soft errors and EMI. Although both types of faults can disrupt system operation, malicious faults are often intentional and require dedicated security-focused mitigation techniques, whereas this paper primarily addresses non-malicious faults stemming from environmental or operational factors. Malicious faults, within the context of fault tolerance, refer to deliberate and intentional actions taken by malicious actors to disrupt or compromise the normal functioning of a computer system, network, or software application. These actions are aimed at exploiting vulnerabilities in order to compromise the system's integrity, availability, or confidentiality [70]. Unlike transient and permanent faults, which often arise from natural hardware failures or environmental factors, malicious faults are caused by human intent and typically involve actions such as hacking, malware deployment, or unauthorized access.

In a cost-effective method proposed in [71], transient faults are detected through coarse-grain CFC, achieving efficiency by simplifying signature calculations within BBs and conducting checks at a coarse-grain level. To assess the effectiveness of this approach, a comprehensive fault injection campaign was conducted, using single bit-flips to model transient faults. Transient faults may not cause permanent damage to the hardware, but they can silently corrupt an application's correctness during runtime or even lead to system crashes. For instance, HP [72] reported frequent failures in their 2048-CPU system at the Los Alamos National Laboratory due to high-energy cosmic rays. A study [73] revealed that the BlueGene/L machine installed in Lawrence Livermore National Labs experienced soft errors approximately every four hours. Considering the estimated reliability drop per bit with each generation of processors [74], it becomes essential to provide transient fault protection schemes for both current and future systems. Transient fault detection techniques rely on different forms

TABLE 3. Comparison of fault mitigation methods.

Method	Fault Types Mitigated	Advantages	Limitations	Performance Overhead
Control Flow Checking (CFC)	Transient and Permanent Faults	High fault detection coverage, applicable to both hardware and software.	High code size and execution time overhead; detection limited to control-flow errors; struggles with intra-block faults	High
Repetition Execution	Transient Faults	No hardware modification required, effective for transient faults	Inefficient for permanent faults; requires re-execution, increasing time overhead	Medium
Lockstep Execution	Transient and Permanent Faults	High fault masking, ensures system consistency in critical applications	Requires significant hardware resources, high cost, and synchronization complexity	High
AI-Based Methods	Anomalies, Predictive Faults	Adaptive, capable of predicting and mitigating faults dynamically	Resource-intensive, requires training, and may introduce delays depending on complexity	Variable

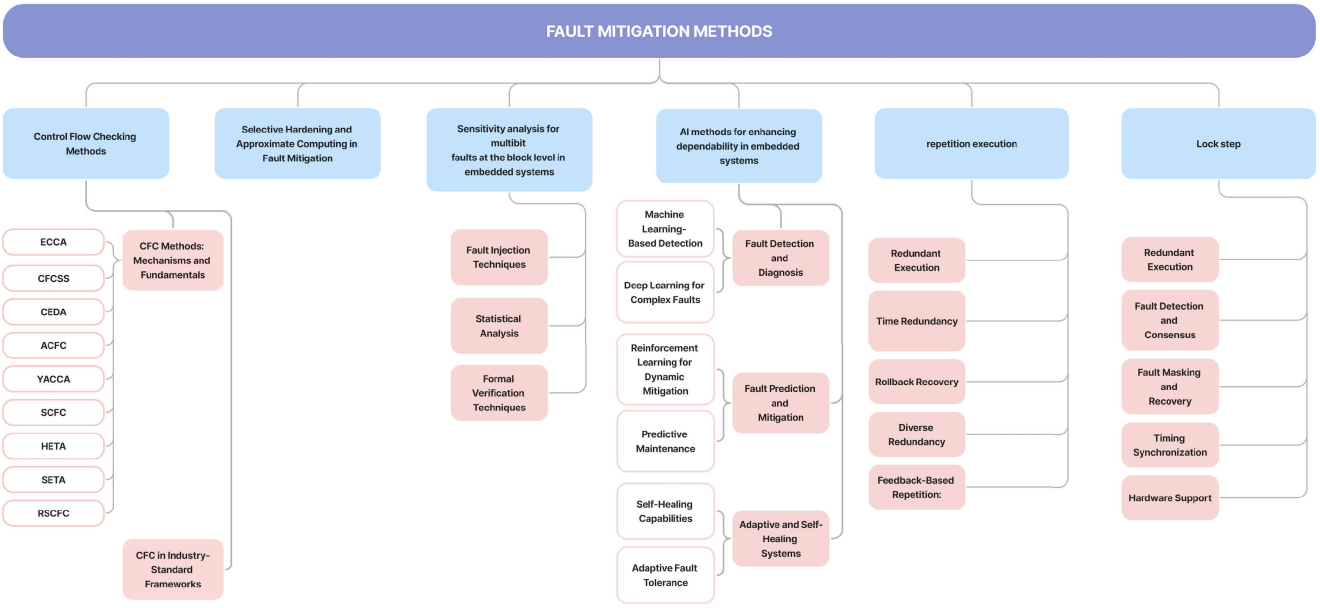

FIGURE 6. Fault mitigation methods.

of redundant checking, either in hardware or software. Hardware solutions like DMR, TMR, and watchdog processors [46] are employed in systems like IBM Z-Series servers [75], HP NonStop system [76], and Boeing 777 airplanes [77]. However, hardware-based solutions introduce unavoidable area and energy costs, making them unsuitable for commodity-embedded systems. Software-based redundant checking, on the other hand, is more appealing for transient fault detection due to its lower production costs and higher flexibility. Securing control flows is crucial for transient fault protection, as CFEs are more likely to cause programs to behave incorrectly. While traditional software methods [78], [79] provide high fault coverage, they inject a significant number of validating instructions into programs, resulting in moderate to large performance overhead. Recent studies [80], [81] attempt to reduce this validation overhead by injecting fewer instructions, but they may sacrifice fault coverage due to their heuristic approaches. Software-based transient fault detection techniques are categorized into data flow protection and control flow protection. Although data flow errors can be masked during program executions, CFEs are more challenging to hide. Researchers from industry and academia have been actively seeking solutions to counter the threat of transient faults in both hardware and software. Hardware-only solutions, with sufficient resources, are more efficient for a single, fixed reliability policy, while software-only solutions offer flexibility and lower costs. Software-only solutions can be deployed immediately on existing hardware by recompiling the application. However, devising correct software solutions for transient faults is a challenging task due to the numerous fault scenarios.

While CFC methods offer robust fault detection, they come with notable limitations. The primary drawback lies in their high code size and execution time overhead, caused by the insertion of additional instructions required to monitor control flow. Furthermore, CFC techniques are less effective in detecting intra-block faults unless complemented by other methods. Hardware-based CFC approaches, though effective,

can also be cost-prohibitive and may not be suitable for systems where resources, such as processing power and memory, are constrained [65].

In the automotive industry, CFC methods are particularly significant due to the stringent safety and reliability requirements. The following sections will delve into the fundamental mechanisms of CFC methods and their specific applications in automotive systems, exemplified by the AUTOSAR framework.

1) CFC METHODS: MECHANISMS AND FUNDAMENTALS

One common approach in CFC methods is signature monitoring, where redundant instructions are inserted into the software unit's source code. This method proves advantageous as it doesn't necessitate any special hardware or operating system requirements, making it adaptable to Commercial Off The Shelf (COTS) micro-controllers, even low-power units. Furthermore, CFC harmoniously complements hardware-based hardening techniques, such as watchdogs, and can be expedited through external hardware support for run-time signature execution and comparison.

In the context of CFC, the typical approach involves dividing the source code into BBs and meticulously inspecting the code within these blocks, along with the branches connecting them. To facilitate this process, a watchdog processor can be employed, enabling efficient and effective control flow verification. The errors analyzed in these methods fall into three general categories: a) illegal jumps within a BB, b) illegal jumps among BBs, and c) illegal jumps from a BB to the unused memory space. These illegal jumps result in Control Flow Errors (CFEs). The following shows six situations that jumps result in a CFE:

- an illegal jump from the end of one BB to the beginning of another BB.
- a legal but incorrect jump from the end of one BB to the beginning of another BB.
- a jump from the end of one BB to any point within another BB.
- a jump from any point within one BB to any point within another BB.
- a jump from any point within a BB to another point within the same block.
- an illegal jump from a BB to the unused space of memory, which refers to the space between BBs.

It is important to note that regardless of the approach used (software or hardware-based), in industrial applications, the method should be capable of handling the aforementioned errors while minimizing memory overhead and execution time increase.

CFC methods utilize Control Flow Graph (CFG) alongside signatures computed by redundant instructions to detect illegal jumps. The basic idea behind signature-monitoring techniques is to assign a static signature to each BB, along with a dynamic global signature. In all CFC detection methods, each BB is associated with a unique static signature. CFC methods employ precise detection approaches by

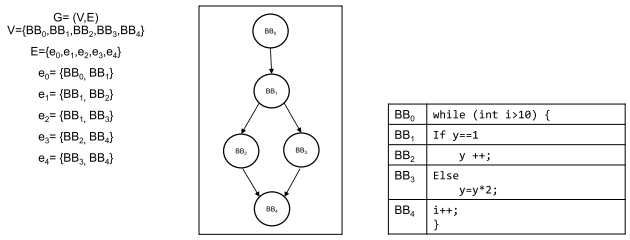

FIGURE 7. Example of program CFG and sample code. The execution path from BB1 to BB2 or from BB1 to BB3 is valid, while a jump from BB1 to BB4 is invalid and referred to as CFE. [85].

generating the CFG from high-level language source code, defining the BB signatures and their computation methods. During execution of the hardened software component, the signature values computed at run-time are compared with the predetermined signatures. In case of a mismatch, an error signal is activated to trigger the detection. The CFE detection methods can be divided into hardware-based methods [60], [82], mixed software-hardware methods [83], [84], and software-based methods. The hardware-based methods require additional hardware components to detect CFEs. Figure 7 provides a graphical representation of the CFG for a sample source code developed in the C language.

Some of the most commonly used CFC methods are based on comparing the run-time signature computed value with the expected values assigned to each block at the design or compile time. We will clarify them below to shed more light on the techniques.

In [86] authors proposed the Enhanced Control Flow Checking Using Assertions (ECCA) method. It is an enhanced version of Control-Flow Checking Using Assertions (CCA) [87] that is targeted for real-time distributed systems. ECCA overcomes the limitations of CCA by introducing a new assertion method that allows for the detection of control-flow errors that were previously undetectable by CCA. In ECCA, each BB in a program should be given a special numerical identification number. Specific assertions that use the identifiers of the involved BBs check the control flow when the processor executes a new BB. Extending the CCA technique, ECCA methods are able to identify all CFEs between various BBs. Still, ECCA methods are unable to identify errors within a single BB or faults that result in incorrect decisions being made on a conditional branch.

In Control Flow Checking by Software Signatures (CFCSS), which is covered in [78], all branches' destinations are evaluated before they jump, not their sources. A global variable named G is initialized with a program's first BB's signature while it is being executed. In order to determine the difference between the signatures of the source and target blocks, CFCSS uses the XOR function to calculate the target block signature from the source block's signature. By comparing the computed signature with the anticipated one, control flow will be examined. The technique outlined in [78] manually inserts control flow checking assertions. This will be accomplished by beginning each BB with a few instructions. First, setting the outgoing signature variable

after checking the incoming signature variable. This makes it possible to confirm the accuracy of the execution flow. It does not require specialized hardware, such as a CFC watchdog. It implies that CFCSS is applicable even in the absence of multitasking support by the operating system. If several BBs merge into a single BB at their ends, CFCSS cannot identify errors.

The authors of [79] proposed Control-flow error detection using assertions (CEDA) by assigning a signature verification at the start and end of each BB, and detecting the ''aliasing errors'' by maintaining unique signatures for each of the aliased blocks. Run-time signatures, which are inserted during compilation, are used by CEDA to identify errors in the control flow effectively. As a result, CEDA can identify all errors that violate the program flow graph, but it cannot identify illegal but correct jumps (according to the program flow graph). As a result, CEDA is unable to detect all the faults.

According to [88], Assertions for Control Flow Checking (ACFC) is a classification scheme for control flow faults and a CFC method that does not rely on predecessor-successor relationships between BBs. The method uses fewer instructions than earlier techniques. Consequently, the method has less memory overhead than the earlier techniques, but its detection performance suffers as a lot.

A CFC technique described in [89] is ''Yet Another Control-Flow Checking using Assertions'' (YACCA). In this technique, each BB entry and exit point receives a special signature. The benefit of this approach is that it allows for the detection of CFEs that occurred when the program flow changed from one BB's inside to that of one of its legitimate successors, even if the succeeding BB returns control to the BB that was subjected to the incorrect jump. This is possible because the signature is re-evaluated prior to each branch instruction to eliminate the CFE for the incorrect successor. In comparison to CFCSS, the YACCA has higher performance overhead and fewer undetected errors.

Reference [90] proposed Software-Based Control Flow Checking (SCFC). The method makes use of two run-time variables: one that holds the run-time values ID of the BBs and another that holds the run-time signature S. The compile-time signature is created using the same method as SEDSR [90]. In the BB, a CFE can be found in either the run-time ID or the run-time signature S that has the incorrect value. The compile-time value of the BB should be included in the ID, and the predecessor BB's signature should be included in the S. In the BB, ID and S are updated at various locations. After confirming it, the S is updated in the middle of the BB, and the ID is updated to the compile-time id of the succeeding block.

Another approach is Hybrid Error-detection Technique using Assertions (HETA) [60]. HETA can detect incorrect jumps during the program execution. HETA develops CEDA techniques and associates them with hardware resources, a watchdog, for achieving complete fault detection. Using HETA methods cannot detect 100% of the errors.

An alternative approach to detect CFEs in processors without hardware-implemented hardening techniques is the Software-only Error-detection Technique using Assertions (SETA) [91]. This method aims to reduce computation units' costs by utilizing two previously described techniques: Hardware-Enabled Timer-based Assertion (HETA) and Control-flow Error Detection Analysis (CEDA). Both techniques utilize run-time signatures to identify errors related to the control flow. Signatures are calculated in advance and compared with the signatures computed at run-time. To implement SETA, the application code is divided into BBs, and two types of BBs are defined: Type A and Type X. Type A BBs have multiple predecessors, at least one of which has multiple successors. BBs that do not meet these conditions are classified as Type X. The defined BBs are then grouped into networks, where BBs sharing a common predecessor belong to the same network. Each BB has two signatures: the Node Ingress Signature (NIS) and the Node Exit Signature (NES). The NIS is compared when entering the BB, while the NES is checked when exiting the BB. The NIS describes the current BB, while the NES is used to identify the successor network and its subsequent legal successor BBs.

Another technique proposed is the Relationship Signatures for Control Flow Checking (RSCFC) [92]. RSCFC encodes control flow relations between different BBs into specially formatted signatures and inserts CFC instructions at the head and end of every BB. This technique detects inter-block CFEs using three variables: a compile-time signature (si), the CFG locator (Li), and the cumulative signature (mi). RSCFC has a higher fault detection rate compared to CFCSS, but it incurs a higher performance overhead.

In summary, signature monitoring methods such as YACCA [89], CFCSS [78], CEDA [79], RASM [93], SEDSR [90], and ECCA [86] focus on monitoring run-time signatures with compile-time signatures at the BB level to address illegal inter-block jumps during application execution. These methods differ in how signatures are computed, and checks are performed. To enhance the existing methods that cover illegal intra-block jumps, instruction monitoring techniques have been developed. These include RSCFC [92], Software implemented error detection (SIED) [94], and Random Additive Control Flow Error Detection (RACFED) [95], which inspects the correct execution order of instructions. Additionally, a behavior-based software technique [96] and the Software Implemented Hardware Fault Tolerance (SIHFT) [97] approach have been presented for detecting CFEs in multi-core architectures and low-cost embedded systems used in safety-critical applications. SIHFT is especially suitable for applications where availability and execution speed are not major concerns.

Table 4 compares the detection coverage and overheads of different CFC methods. The measurements were made

by [93] and [95] on implementations at the assembly level. The authors used their Software-Implemented Fault Injection (SWIFI) tool to validate the comparisons between the techniques.

2) CFC IN INDUSTRY-STANDARD FRAMEWORKS

CFC methods have been adopted in various industry-standard frameworks to ensure safety and reliability in embedded systems. In this section, we briefly discuss the application of CFC within three prominent frameworks: AUTOSAR, ISO 26262, and IEC 61508. This comparative analysis highlights their similarities, differences, and contributions to fault tolerance mechanisms.

A good example of applying CFC methods in software architecture is the AUTomotive Open System ARchitecture (AUTOSAR) [98], [99]. AUTOSAR provides a modular software architecture designed to standardize interfaces and ensure system interoperability in automotive applications. The Watchdog Manager (WdM) is a key feature in AUTOSAR that monitors the execution flow of program instructions, helping to detect CFEs caused by transient or permanent faults. However, it is worth noting that AUTOSAR's specification does not include the concept of a full CFG, which limits its ability to comprehensively implement CFC techniques. AUTOSAR's strength lies in its ability to ensure real-time monitoring and fault detection in embedded systems, making it an industry standard in modern vehicles. Nevertheless, its reliance on standardized monitoring methods like WdM can be a limitation when compared to more comprehensive safety standards.

In contrast, ISO 26262 is an automotive-specific framework that emphasizes safety across the entire development lifecycle of automotive systems. While it does not focus specifically on CFC, it mandates the use of fault tolerance techniques to meet the desired Automotive Safety Integrity Levels (ASIL). These levels determine the required safety measures based on the system's risk assessment, including techniques like redundancy and control-flow verification. ISO 26262 also places a stronger emphasis on formal verification methods and rigorous safety analysis, such as Fault Tree Analysis (FTA) and Failure Mode and Effects Analysis (FMEA), to address system-level faults. In this context, CFC is a recommended method to enhance fault detection and handling in safety-critical software, but ISO 26262 is more general in its application across different fault mitigation techniques.

Similarly, IEC 61508 is a widely recognized safety standard that applies to industrial systems. It uses Safety Integrity Levels (SIL) to quantify the safety requirements for systems in sectors like manufacturing, nuclear power, and transportation. IEC 61508 allows for flexible implementation of fault tolerance mechanisms, including CFC, but its broader focus is on system-level safety. Unlike AUTOSAR, which is highly specific to the automotive industry, IEC 61508 can be adapted to a variety of embedded systems, and CFC may be implemented as one of several fault tolerance strategies. IEC 61508's focus on system-wide safety integration, rather than application-specific frameworks, allows it to address a wider range of faults, but it may lack the modularity and specific real-time features of AUTOSAR.

Comparison of Frameworks: While all three frameworks support fault tolerance techniques, their approach to CFC and system safety varies:

- *AUTOSAR*: Provides a modular, real-time architecture for automotive systems with specific monitoring tools like Watchdog Manager, but lacks comprehensive control-flow verification.
- *ISO 26262*: A general automotive safety standard that emphasizes a variety of fault tolerance techniques, including CFC, but does not specifically implement CFC methods.
- *IEC 61508*: Applies to a broader range of industries and offers flexibility in applying fault tolerance mechanisms, including CFC, with a focus on system-level safety and SIL levels.

In conclusion, while AUTOSAR plays a significant role in implementing real-time fault detection methods like Watchdog Manager in automotive applications, it should be viewed as a framework alongside other industry standards such as ISO 26262 and IEC 61508. Each of these frameworks offers unique contributions to fault tolerance, and their application depends on the specific requirements of the system being developed. A balanced consideration of these frameworks can provide deeper insights into how CFC and other fault tolerance mechanisms are implemented across different industries.

B. SELECTIVE HARDENING AND APPROXIMATE COMPUTING IN FAULT MITIGATION

Fault mitigation methods such as Selective Hardening and Approximate Computing have gained prominence due to their efficiency in resource-constrained environments and their ability to complement control flow techniques. These methods primarily focus on improving system dependability while minimizing performance and energy overheads, making them ideal for embedded systems and applications with stringent resource constraints.

1) SELECTIVE HARDENING

Selective Hardening is a technique that optimizes fault tolerance by applying hardening selectively to critical components within the system, rather than applying fault tolerance mechanisms across the entire system [100], [101]. This method reduces performance and energy overheads, as only the most vulnerable or critical parts of the system are protected against faults. Selective hardening has been particularly effective in systems where resources are limited, such as embedded systems and low-power devices. By focusing on critical components, selective hardening ensures system reliability while maintaining an optimal balance between performance and fault tolerance. This method works in conjunction

TABLE 4. Compare control flow control techniques [85].

CFC Method	Used Variables	Signatures	intra-block	detection performance [%]	Code size overhead [%]	Execution time overhead[%]
ECCA	4	prime-numbers	χ	73.5	36.0	244.8
CFCSS	2	randomized-bit	χ	75.8	15.2	76.6
YACCA	2	bit-field	χ	82.8	30.0	203.2
RSCFC	2	bit-field	✓	49.4	17.5	86.8
SEDSR	3	bit-field	✓	46.8	12.3	67.1
SCFC	3	bit-field	✓	60.4	22.9	115.7
SIED	2	random numbers	✓	52.4	14	115.7
RACFED	3	random numbers	✓	N.A.	N.A.	81.5

with control flow techniques like CFC, providing additional protection where it is most needed, such as in memory units or processing cores handling critical computations.

2) APPROXIMATE COMPUTING

Approximate Computing is another method that has gained traction in recent years, particularly in applications where energy efficiency and performance are critical. The technique allows for controlled approximation of computations, trading off error tolerance for improved performance and reduced energy consumption [102]. In many embedded systems, especially those used in multimedia processing, sensor fusion, or machine learning, not all computations require perfect accuracy [103]. Approximate computing leverages this fact by allowing certain non-critical computations to be performed with lower precision, thus saving power and computational resources. When used in conjunction with selective hardening or control flow techniques, approximate computing enables a more flexible and efficient approach to fault tolerance, where critical tasks maintain high accuracy and less critical tasks are approximated.

Both of these techniques complement traditional control flow methods such as CFC, offering a balanced approach to fault mitigation in modern embedded systems. By integrating data flow techniques like Selective Hardening and Approximate Computing, systems can achieve higher fault tolerance with minimal impact on performance, making these methods particularly suitable for low-power, resource-constrained environments.

C. SENSITIVITY ANALYSIS FOR MULTI-BIT FAULTS AT BLOCK LEVEL IN EMBEDDED SYSTEMS

Sensitivity analysis for multi-bit faults at the block level is an evaluation method used to understand the impact of faults on system reliability. While it is not a fault mitigation technique itself, it plays a crucial supporting role in fault tolerance. By analyzing how multi-bit faults propagate and affect system behavior, sensitivity analysis informs the design of more robust fault tolerance mechanisms. Multi-bit faults, which affect multiple bits within a basic block, pose greater challenges compared to single-bit faults due to their complex propagation patterns and higher impact on system reliability. As device scaling continues and operating environments become harsher, the likelihood of multi-bit faults increases, necessitating advanced analysis and mitigation techniques [104], [105].

While sensitivity analysis does not mitigate faults directly, it provides essential insights for designing more effective mitigation strategies. Multi-bit faults can have a more severe impact on system behavior compared to single-bit faults. A single flipped bit might not cause a functional error, but multiple flipped bits within a block can lead to unpredictable outcomes and potentially catastrophic system failures. Understanding the sensitivity of embedded systems to multi-bit faults is crucial for developing efficient fault tolerance techniques.

Researchers are actively exploring various techniques for sensitivity analysis of multi-bit faults at the block level. This includes:

- *Fault Injection Techniques*: Simulating multi-bit faults by injecting errors into specific bit combinations within a block to analyze their effect on system behavior [104].
- *Statistical Analysis*: Employing statistical methods to assess the probability of multi-bit faults occurring and their impact on system reliability [106].
- *Formal Verification Techniques*: Leveraging formal verification tools to analyze the behavior of hardware blocks under various multi-bit fault scenarios.

These techniques serve as diagnostic tools to assess system vulnerabilities, guiding the refinement of fault tolerance mechanisms.The findings from sensitivity analysis of multi-bit faults can guide the development of more robust fault tolerance mechanisms. This might involve redundancy techniques, stronger error correction codes, and architectural design considerations.

Challenges in analyzing multi-bit faults include complexity of fault propagation, increased computational overhead, and inadequacy of traditional detection mechanisms. Mitigation of multi-bit faults requires more sophisticated approaches than those used for single-bit faults, such as advanced coding schemes and dynamic reconfiguration methods.

Recent advances include probabilistic models, formal verification techniques, machine learning approaches, and enhanced fault injection tools, all contributing to more robust fault tolerance mechanisms.

Sensitivity analysis for multi-bit faults is a valuable technique for understanding fault impact, but its use is constrained by several factors. The computational overhead associated with analyzing multi-bit faults is significant, as these faults involve complex interactions and require detailed modeling. Furthermore, performing such analysis

often requires specialized tools, which can limit its practicality in real-time systems or environments where rapid fault detection is essential. As a result, its adoption may be restricted to high-criticality systems where detailed fault analysis outweighs performance concerns.

In conclusion, sensitivity analysis for multi-bit faults at the block level is critical for ensuring the dependability of embedded systems. Although it is not a fault mitigation method itself, it provides invaluable insights that contribute to the development of more effective fault tolerance techniques. By addressing the challenges and leveraging recent advancements, researchers can enhance system reliability in increasingly complex and interconnected computing environments.

D. AI METHODS FOR ENHANCING DEPENDABILITY IN EMBEDDED SYSTEMS

The integration of Artificial Intelligence (AI) methods in embedded systems has emerged as a promising approach to enhance system dependability. AI techniques can improve fault detection, diagnosis, and mitigation processes, offering a significant advantage over traditional methods. This section summarizes existing works on the application of AI methods to enhance the dependability of embedded systems.

AI techniques, such as machine learning, deep learning, and reinforcement learning, have been increasingly applied to various aspects of dependability in embedded systems. These methods provide advanced capabilities for handling complex fault scenarios and improving system robustness.

1) FAULT DETECTION AND DIAGNOSIS

- *Machine Learning-Based Detection*: Machine learning models, particularly supervised learning algorithms, have been employed to detect anomalies and faults in embedded systems. These models are trained on historical fault data to recognize patterns indicative of faults [107].
- *Deep Learning for Complex Faults*: Deep learning models, such as convolutional neural networks (CNNs) and recurrent neural networks (RNNs), have shown efficacy in detecting and diagnosing complex fault patterns that are difficult for traditional methods to identify [108], [109].

2) FAULT PREDICTION AND MITIGATION

- *Predictive Maintenance*: AI methods are used to predict potential faults before they occur, enabling proactive maintenance and reducing downtime. Predictive models analyze real-time data to forecast future faults and suggest maintenance actions [110].
- *Reinforcement Learning for Dynamic Mitigation*: Reinforcement learning algorithms have been explored for dynamic fault mitigation. These algorithms learn optimal mitigation strategies through interaction with the system environment, adapting to new fault scenarios in real-time [111].

3) ADAPTIVE AND SELF-HEALING SYSTEMS

- *Self-Healing Capabilities*: AI techniques enable the development of self-healing systems that can autonomously detect, diagnose, and repair faults. These systems use AI to monitor their own state and execute corrective actions without human intervention [112].
- *Adaptive Fault Tolerance*: AI methods facilitate adaptive fault tolerance mechanisms that adjust their fault tolerance strategies based on the current operating conditions and fault characteristics [112].

AI-based fault detection and mitigation methods provide dynamic, adaptive solutions but are not without limitations. These techniques are computationally intensive, requiring substantial processing power and memory, which may not be available in all embedded systems. Additionally, the training process for AI models can be resource-heavy, requiring large datasets and time to achieve accuracy. The potential for latency in decision-making, especially in real-time systems, is another challenge that must be addressed to ensure timely fault mitigation.

In conclusion, the application of AI methods in embedded systems significantly enhances their dependability by improving fault detection, diagnosis, prediction, and mitigation capabilities. Machine learning, deep learning, and reinforcement learning techniques provide advanced tools for handling complex fault scenarios, enabling the development of adaptive and self-healing systems. As research in this field continues to evolve, AI-driven approaches will play an increasingly vital role in ensuring the reliability and robustness of embedded systems in diverse application domains.

E. REPETITION EXECUTION

Repetition execution is a widely employed fault mitigation method in fault-tolerant embedded systems. By executing critical tasks multiple times and comparing the results, repetition execution aims to detect and tolerate faults that may occur during program execution. This subsection provides an overview of repetition execution techniques and their effectiveness in mitigating faults.

- *Redundant Execution*: One approach to repetition execution involves redundant execution, where critical tasks are executed multiple times in parallel. The results obtained from each execution are compared, and a majority voting or a consensus-based decision method is used to determine the correct result [113]. Redundant execution techniques can mitigate both permanent and transient faults, ensuring the system's resilience to unexpected failures.
- *Time Redundancy*: In time-redundant execution, critical tasks are executed at different time instances, providing redundancy in the temporal domain. By repeating the execution of tasks at periodic intervals, fault detection and recovery mechanisms can be incorporated [114].

Time redundancy is particularly effective in mitigating transient faults that may occur intermittently.

- *Rollback Recovery*: Rollback recovery is a technique that combines repetition execution with checkpointing. Periodically, checkpoints are taken to capture the system's state. In the event of a fault, the system can roll back to a previous checkpoint and re-execute the tasks from that point to ensure correctness and consistency [115]. Rollback recovery provides fault tolerance and can handle permanent faults that affect the system's state.
- *Diverse Redundancy*: Diverse redundancy is a technique that combines repetition execution with diversity in the implementation or design of critical tasks. Multiple versions of the same task are executed in parallel, each using a different algorithm, implementation, or platform. By incorporating diversity, the system can tolerate faults that affect only a subset of the redundant tasks, ensuring higher reliability and fault tolerance.
- *Feedback-Based Repetition*: Feedback-based repetition execution involves continuously monitoring the system's behavior and adapting the repetition method accordingly. Fault detection mechanisms analyze the system's output and dynamically adjust the repetition execution parameters, such as the number of repetitions or the timing of task execution, to optimize fault tolerance [116]. This approach improves the system's resilience by adapting to changing fault conditions.

An example of leveraging repetition execution fault mitigation methods is in the automotive domain, where ISO26262 acknowledges recovery through repetition as an accepted error-handling method. This approach involves resetting the specific hardware components involved in a faulty execution and re-executing the affected software components, as described in the AUTOSAR standard for automotive software design. Furthermore, ISO26262-6:2011 clause 10.4.3 states that when generating test cases for software resource usage testing, it is essential to determine the maximum execution time of the program under analysis to demonstrate the schedulability of the integrated system [117].

Taking a repetition execution approach can help enhance the CFC methods, in accurately and efficiently solving detection problems within BBs and across procedures, as well as addressing the issues of control flow error detection hysteresis and reducing time overhead. Existing control flow error detection methods based on signature analysis have limitations due to their reliance on a single type of signature, struggling to balance program residual failure rate and time overhead. To overcome these limitations, a proposed technique called basic block repetition is introduced [118], which involves executing a program multiple times while monitoring the behavior of its BBs to identify anomalies or deviations in control flow, signaling the presence of errors. The process includes instrumentation, execution, monitoring, and analysis of BBs. BB repetition can be detected using various techniques, such as static analysis with hash tables or control flow graphs and dynamic analysis using tracing tools to track executed BBs. This approach offers valuable insights into the control flow dynamics of software systems, aiding in the identification and debugging of CFEs ultimately improving software reliability, security, and performance.

Repetition execution proves effective for transient faults but presents limitations in certain scenarios. This method struggles to address permanent faults, as merely repeating execution may not resolve hardware-level issues. Additionally, the need for multiple re-executions significantly increases the time overhead, making this technique less suitable for real-time systems where timing constraints are critical. Such drawbacks limit its applicability in environments where performance and time efficiency are paramount.

In summary, repetition execution is a prominent fault mitigation method in fault-tolerant embedded systems. By executing critical tasks multiple times and comparing the results, repetition execution techniques can effectively detect and tolerate faults.

F. LOCK STEP

Lockstep is a widely used fault mitigation method in fault-tolerant embedded systems. It involves redundant execution of critical tasks in parallel. The redundant executions are kept synchronized to ensure consistency and fault tolerance. This subsection provides an overview of the lockstep approach for fault mitigation [21], [61], [62], [63].

- *Redundant Execution*: In the lockstep approach, critical tasks are redundantly executed in parallel with multiple identical copies or processors. These copies execute the same instructions simultaneously, and the outputs are compared to detect discrepancies caused by faults [119]. Lockstep execution is particularly effective in mitigating permanent faults that affect the consistent behavior of a system.
- *Fault Detection and Consensus*: Lockstep execution relies on fault detection mechanisms to identify inconsistencies among redundant copies. By comparing the outputs of redundant tasks, fault detection algorithms can detect faults and initiate recovery actions. Consensus-based techniques, such as majority voting or Byzantine fault tolerance algorithms, are commonly used to determine the correct output when discrepancies arise [120].
- *Fault Masking and Recovery*: The redundant nature of lockstep execution provides fault masking capabilities. If a fault occurs in one copy, the correct output can still be obtained by comparing the results of the other copies. This fault-masking property enhances the fault tolerance of embedded systems. In case of a fault, recovery mechanisms can be triggered to restore the system to a consistent state [121].
- *Timing Synchronization*: Synchronization of the redundant copies is crucial in lockstep execution to maintain consistency. Precise timing synchronization is required to ensure that the copies execute instructions at the same

rate and in the same order. Time-triggered protocols, clock synchronization techniques, or global time references are employed to achieve timing synchronization among the redundant copies [122].

- *Hardware Support*: Hardware-level support plays a vital role in implementing the lockstep approach efficiently. Specialized hardware architectures and components, such as redundant processors, comparators, and error detection circuits, are designed to facilitate lockstep execution. These hardware features ensure synchronized execution, fault detection, and fault recovery in a timely and efficient manner [123], [124].

An evolution of this technique, known as Macro Lockstep, extends the traditional lockstep approach to modern multi-core and distributed systems, where complete synchronization may not always be necessary. Macro Lockstep allows loosely coupled cores or processors to operate semi-independently, synchronizing only at critical checkpoints rather than after every instruction. This method significantly reduces the overhead associated with traditional lockstep, making it more suitable for resource-constrained environments and systems where power efficiency is critical.

Macro Lockstep is particularly effective in distributed embedded systems, where full synchronization across processors can be inefficient. By synchronizing at higher-level checkpoints, the system can detect and recover from faults while maintaining better performance and scalability. This makes Macro Lockstep a valuable solution in modern fault-tolerant designs, balancing fault detection, performance, and power efficiency.

While lockstep execution is a powerful fault tolerance mechanism, it comes at the cost of increased system complexity and resource demands. Implementing traditional lockstep requires substantial hardware resources, leading to higher system costs and greater power consumption. Moreover, ensuring synchronization between redundant processors introduces challenges, as timing mismatches can lead to errors. However, by using Macro Lockstep, these resource demands can be mitigated, allowing for better scalability and efficiency in modern embedded systems. Lockstep ensures fault detection, fault tolerance, and system recovery through the redundant execution of critical tasks in parallel, and Macro Lockstep offers an extension of these capabilities with reduced synchronization overhead.

Lockstep execution is a powerful fault tolerance mechanism but comes at the cost of increased system complexity and resource demands. Implementing this method requires substantial hardware resources, leading to higher system costs and greater power consumption. Moreover, ensuring synchronization between redundant processors introduces challenges, as timing mismatches can lead to errors. These factors make lockstep execution more suitable for systems with significant hardware availability, but less so for cost-sensitive or resource-limited applications.

All in all, lockstep execution is a powerful fault mitigation method in fault-tolerant embedded systems. By redundantly executing critical tasks in parallel and comparing the results, lockstep ensures fault detection, fault tolerance, and system recovery.

V. A NOTE ON CONTROL-FLOW INTEGRITY TECHNIQUES FOR SOFT ERRORS - SECURITY

CFI techniques are employed to ensure that a program functions as intended without being affected by soft errors, which can arise from external factors like radiation, power surges, or electromagnetic disturbances. These errors have the potential to cause unintended consequences, such as data loss, diminished system reliability, and even security breaches [125].

Soft errors, including SEUs and SETs, can disrupt the control flow of an embedded system, potentially compromising both system security and reliability. When soft errors affect control paths, they can lead to unauthorized execution paths, creating vulnerabilities in the system's control flow that attackers could exploit.

While fault tolerance traditionally focuses on managing non-malicious faults like soft errors, it is important to address security concerns because intentional attacks, such as hacking or malware, can exploit vulnerabilities created by such faults. In this context, security and fault tolerance are closely related, as both aim to ensure the system's dependable operation.

The primary purpose of CFI techniques is to mitigate the risk of security breaches resulting from soft error-induced deviations by implementing a set of rules on the program's Control Flow Graph (CFG). This graph represents the program's control flow and the relationships between its various components. These rules dictate the permissible execution paths and prevent any unauthorized or malicious alterations to the control flow. One commonly used CFI technique is ''Strict Control Flow Integrity'' (SCFI), which enforces rules to maintain the integrity of the program's control flow graph during execution. Any attempt to deviate from this graph is detected and prevented, thus safeguarding the program's integrity. Security becomes particularly relevant in embedded systems because attackers may exploit soft errors or other system vulnerabilities to launch attacks that compromise system integrity, availability, or confidentiality. Thus, combining CFI techniques for fault tolerance and security ensures that systems remain resilient in the face of both accidental and intentional disruptions. Additional CFI techniques include shadow-stack-based CFI, implicit CFI, and hybrid CFI. Soft errors can affect the direct as well as indirect branches, and hence CFI, as is, is not directly applicable for soft errors. Though direct branches can also be protected in a manner similar to dynamic branches, but the already high overhead (20%-60% for dynamic branches only) would become prohibitive [126].

While CFI techniques are crucial for addressing control flow deviations, it's important to distinguish between non-malicious faults and malicious attacks. A fault is typically non-malicious and can be evaluated using probability measures such as failure rate or mean time to failure.

In contrast, an attack is deliberate and targets the system's weakest points, often exploiting vulnerabilities to compromise the system's integrity, availability, or confidentiality.

In summary, CFI techniques serve as a set of measures to protect software systems from security breaches caused by soft errors. By enforcing strict rules on the program's control-flow graph, these techniques can identify and prevent any unauthorized or malicious changes to the program's execution, thereby bolstering its security and reliability.

A. ENHANCED SECURITY MEASURES BEYOND CFI

Given the deliberate nature of attacks, additional security measures must be considered to complement CFI techniques and provide a robust defense against both faults and attacks:

- *Intrusion Detection Systems (IDS)*: IDS can monitor network traffic and system activities for suspicious behavior indicative of an attack. Implementing IDS alongside CFI can enhance detection capabilities, particularly for sophisticated attacks that manipulate control flows.
- *Redundancy and Diversity*: Employing multiple, diverse implementations of critical functions can make it harder for an attacker to compromise the system. Techniques such as N-version programming and redundant execution can provide additional layers of security.
- *Cryptographic Techniques*: Encrypting data and communications ensures that even if an attacker gains access to the system, they cannot easily read or tamper with the data. Cryptographic techniques also support secure boot processes and authenticated updates, enhancing overall system security.
- *Access Control Mechanisms*: Implementing strict access controls ensures that only authorized entities can access or modify system components and data. Role-based access control (RBAC) and mandatory access control (MAC) are examples of such mechanisms.
- *Regular Security Audits and Updates*: Conducting regular security audits helps identify and mitigate vulnerabilities before they can be exploited. Keeping the system and its components up-to-date with the latest security patches is crucial for maintaining security.

B. DATA INTEGRITY

In addition to control flow integrity, data integrity is another critical aspect of system security, especially in the context of transient faults or soft errors. Ensuring data remains accurate, consistent, and reliable throughout its lifecycle is paramount for safeguarding against potential vulnerabilities and risks.

In the context of transient faults and soft errors, the integrity of data can be compromised by events such as radiation-induced SEUs or EMI. Soft errors are a type of transient fault but specifically refer to hardware memory bit flips due to external environmental influences like cosmic rays. These errors can be caused by various factors, such as cosmic radiation, electrical noise, or electromagnetic interference, and can adversely impact the integrity of stored data. To mitigate such risks, data integrity measures involve implementing error detection and correction techniques, such as checksums and parity bits, to detect and correct any errors that may occur.

Maintaining data integrity is relatively straightforward in a standalone system with a single database. This is achieved through the use of database constraints and transactions, typically managed by a Data Base Management System (DBMS). Transactions should adhere to the ACID principles (atomicity, consistency, isolation, and durability) to ensure data integrity. Most databases support ACID transactions, which aids in preserving data integrity. However, data integrity in cloud-based systems refers to the preservation of data accuracy. It is crucial to ensure that data remains unchanged and is not lost due to unauthorized user actions. Data integrity forms the foundation for cloud computing services like Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS) [127]. In addition to storing large volumes of data, cloud environments typically offer data processing services. methods such as RAID-like methods and digital signatures can be employed to maintain data integrity in cloud systems.

Remote verification of data integrity in the cloud is a prerequisite for deploying applications. Bowers et al. introduced the ''Proofs of Retrievability'' theoretical framework, which combines error correction codes and spot-checking to facilitate remote data integrity checks [128]. The High-Availability and Integrity Layer (HAIL) system utilizes the Proofs of Retrievability (POR) method to verify data storage across different clouds, ensuring redundancy of copies and enabling availability and integrity checks [129]. Schiffman et al. proposed the use of Trusted Platform Modules (TPM) for remote data integrity checks [130].

Due to numerous entities and access points in a cloud environment, authorization plays a vital role in ensuring that only authorized entities interact with data. By preventing unauthorized access, organizations can have greater confidence in data integrity. Monitoring mechanisms provide increased visibility, enabling the identification of any alterations made to data or system information that may affect its integrity. While cloud computing providers are entrusted with maintaining data integrity and accuracy, it is important to establish a third-party supervision method alongside users and cloud service providers.

In summary, data integrity is paramount for safeguarding data accuracy and consistency, particularly in the context of transient faults or soft errors. Techniques like checksums and parity bits are used to detect and correct errors. While standalone systems can ensure data integrity through database constraints and ACID transactions, cloud-based systems require remote verification methods, such as Proofs of Retrievability and Trusted Platform Modules, to maintain data accuracy across various access points. Authorization and monitoring mechanisms also play crucial roles in preserving data integrity in the cloud, requiring collaboration among

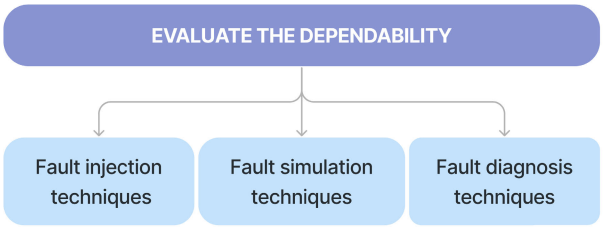

FIGURE 8. Evaluate the dependability.

users, providers, and third-party oversight for effective security.

VI. EVALUATE THE DEPENDABILITY

The evaluation of dependability in fault-tolerant systems is explored in this section. Figure 8 illustrates the organization of this section. It encompasses fault injection techniques, fault simulation techniques, and fault diagnosis techniques. It has provided essential insights into the dependability of numerous systems and has sparked extensive research in various areas. Several mature fault injection tools have been developed, some of which have been successfully implemented in industrial settings [131], [132]. However, there is a prevailing perception in other communities that fault injection is a solved research problem, with the remaining challenges being primarily engineering-related. Nevertheless, fault injection has been a subject of research for many years [133], [134].

Over time, numerous techniques have been proposed to assess different types of fault effects and analyze faulty behavior. Nonetheless, as technology has advanced, the demands for fault injection have become increasingly rigorous. More complex devices necessitate higher performance to conduct larger fault injection campaigns within an acceptable time frame [133].

Fault injection, the deliberate introduction of faults into a Circuit Under Test (CUT), has proven to be an effective method for evaluating susceptibility to soft errors. This approach enables researchers to introduce faults, thus reducing experiment duration artificially. The goals of fault injection include verifying circuit fault tolerance, predicting circuit behavior in the presence of faults, identifying critical components that require mitigation, and validating mitigation approaches.

Various fault injection methods have been proposed, with physical fault injection utilizing particle accelerators being the most widely accepted. However, this method is costly and suitable only for certified circuits. Other physical techniques involve the use of lasers or electromagnetic interference. Alternatively, logical fault injection can be employed by altering register or memory contents within the CUT and observing the effects. This method is simpler and more cost-effective to implement, but it has limitations as it may not provide access to all circuit components. Additional techniques are needed for effective guidance and validation of mitigation techniques during the design phase. These techniques involve evaluating mitigation needs and effectiveness using logical fault injection on a design model. For this purpose, simulation, emulation, or a combination of both, along with appropriate fault models and design tools, can be utilized. The bit-flip model is commonly utilized for logical emulation of radiation-induced faults, allowing for the injection of single and multiple faults. Table 5 provides a summary of how the bit-flip model can be used to model single and multiple effects.

Fault injection necessitates a suitable CUT model, and the level of detail depends on the type of fault being considered. Performance is a critical factor in fault injection, as a sufficient number of faults must be injected to achieve statistical significance. Recent developments have focused on enhancing the performance of fault injection to enable larger campaigns and support more comprehensive fault analysis. [135]

- *Fault injection techniques*
  Fault injection is a testing and debugging technique where faults are intentionally introduced into hardware, firmware, or software to observe system behavior. Fault injection can be performed at various stages (pre-silicon, post-silicon, in-field) and abstraction levels (hardware, software, firmware), allowing for versatile testing of system fault tolerance. While physical techniques such as particle accelerators offer high accuracy, they are costly and limited to certified systems. Logical fault injection, altering registers or memory contents, provides a cost-effective alternative but may not cover all components comprehensively. Fault injection is most effective in verifying fault mitigation strategies and identifying vulnerable system components, especially in post-silicon stages [131].
- *Fault simulation techniques:* Fault simulation, unlike fault injection, uses software models to emulate faults in a system without physically altering the hardware. This technique can be applied at any design stage, making it particularly valuable in the early phases of development. Fault simulation is versatile, allowing for functional, structural, or behavioral fault analysis across different levels of complexity. Its primary advantage is flexibility and cost-efficiency, but simulation results may lack the real-world accuracy of fault injection. Thus, fault simulation is often used to refine designs before validating them with fault injection [134].
  Fault simulation, unlike fault injection, uses software models to emulate faults in a system without physically altering the hardware. This technique can be applied at any design stage, making it particularly valuable in the early phases of development. Fault simulation is versatile, allowing for functional, structural, or behavioral fault analysis across different levels of complexity. Its primary advantage is flexibility and cost-efficiency, but simulation results may lack the real-world accuracy of fault injection. Thus, fault simulation is often used

TABLE 5. Models of soft errors for fault injection. Single-Event Upset (SEU), Multiple-Cell Upset (MCU), Single-Event Multiple Transient (SEMT), Single-Event Transient (SET) [137].

Feature	SEU/MCU	SET/SEMT
Effect	Single/multiple bit-flip	Single/multiple bit-flip
Where?	Any flip-flop	Any gate
When?	Any clock cycle	Any time
For how long?	1 clock cycle (typically)	Variable pulse width

to refine designs before validating them with fault injection.

- *Fault diagnosis techniques:* Fault diagnosis is a debugging technique that utilizes testing and analysis methods to identify and locate faulty components or regions in memory and register circuits. The level at which fault diagnosis is performed, such as gate, transistor, or layout, depends on the type and complexity of the circuit. The stage of fault diagnosis, whether pre-silicon, post-silicon, or in-field, is determined by the availability and accessibility of the circuit. The techniques for fault diagnosis vary based on the fault type, model, level, and stage. Common fault diagnosis techniques for memory and register circuits include signature analysis, parity check, checksum, syndrome decoding, ECC, Error Detection And Correction (EDAC), or fault localization [136].

In summary, while fault injection techniques have been extensively discussed and emphasized, it is important to recognize the complementary roles of fault simulation and fault diagnosis techniques. Fault simulation provides a non-invasive way to study the effects of faults using software models, making it suitable for various stages of design and testing. Fault diagnosis, on the other hand, is essential for pinpointing and rectifying faults post-deployment. Together, these techniques offer a comprehensive toolkit for evaluating and enhancing the dependability of fault-tolerant systems.

VII. CONCLUSION AND FUTURE DIRECTIONS

In this paper, we explored and surveyed various fault tolerance methods designed to mitigate random hardware failures in embedded systems, focusing on real-time embedded systems. The dual focus on software and hardware highlights the comprehensive approach necessary for improving system dependability and extends the potential for fault tolerance strategies to diverse computing environments. The increasing use of embedded systems in safety or mission-critical applications necessitates advanced and reliable fault tolerance methods to ensure seamless automation and operational efficiency in commercial and industrial contexts. The significance of fault tolerance in modern computing systems was discussed, and insights were given on how fault tolerance techniques enhance system dependability by masking faults and detecting errors, allowing for uninterrupted service provision in the presence of internal faults. The paper has highlighted the differences in using hardware or software redundancy to achieve fault tolerance goals, ensuring reliable system functioning.

Special attention has been given to software fault tolerance, as software faults are a leading cause of system failures. While software engineering endeavors to remove most deterministic design faults, it is practically impossible to guarantee that complex software designs are entirely free of such faults. Hence, software fault tolerance techniques are employed as an additional layer of protection to ensure continued service at an acceptable level of performance and safety. Moreover, the increasing complexity and optimization of computer systems for price and performance have introduced the challenge of soft errors or transient bit-errors. This challenge emphasizes the critical role of fault tolerance in modern computing systems, as these errors can potentially lead to system malfunctioning.

The survey has covered various fault tolerance techniques, including hardware, software, and hybrid redundancy, providing valuable insights into their benefits and applicability in different contexts. Additionally, we have discussed fault tolerance approaches tailored specifically for resource-constrained embedded systems, acknowledging the importance of considering limited memory and low-end computation environments in such systems. Nevertheless, more research and development in fault tolerance methods is needed, particularly in the realm of real-time embedded systems, to ensure the reliable and resilient operation of interconnected computing systems.

Despite the comprehensive nature of the techniques reviewed, several limitations must be acknowledged. First, many of the fault tolerance methods, particularly those involving redundancy, introduce significant performance overhead, which can hinder their use in real-time systems with strict resource constraints. Moreover, the scalability of these techniques, especially in complex and evolving embedded systems, remains a significant challenge.

The applicability of certain fault tolerance methods also varies across different domains. Techniques that are highly effective in environments such as aerospace or medical devices, where reliability is paramount, may not be suitable for cost-sensitive applications like consumer electronics or automotive systems. Furthermore, with the increasing integration of artificial intelligence (AI) and machine learning (ML) into embedded systems, new fault models and challenges are likely to emerge, requiring future innovations in fault tolerance.

These limitations suggest several directions for future research. There is a need for more adaptive and scalable fault tolerance mechanisms that can dynamically adjust to changing system requirements without incurring excessive performance overhead. Hybrid approaches, combining hardware and software solutions, could also provide a balance between reliability, performance, and resource efficiency, making them applicable to a wider range of embedded system applications.

Overall, the unique characteristics of embedded systems, such as resource constraints, real-time requirements, and dedicated functionalities, must be carefully considered to develop effective fault tolerance strategies. By addressing these special features, researchers can enhance the reliability and stability of embedded systems in various critical applications. This survey provides valuable insights into fault mitigation techniques and emphasizes the significance of fault tolerance in ensuring the dependability and functionality of modern computing systems. The presented methods, such as control-flow checking (CFC), redundancy approaches, optimized resource management, and security-oriented measures, pave the way for further advancements in the field of fault tolerance and its application in critical computing systems.

Comparison with Existing Surveys: While our survey provides a broad and detailed exploration of fault-tolerance methods, it complements and expands on previous surveys in the field. For instance, [34] presents a detailed analysis of fault-tolerance techniques with a particular emphasis on power, energy, and thermal issues. This focus is highly relevant for energy-constrained embedded systems but does not cover the broader spectrum of fault-tolerance techniques comprehensively. Reference [138] offers a comprehensive survey on system dependability for real-time embedded software, emphasizing fault avoidance, fault detection and correction, and fault tolerance. Our work builds on this by integrating more recent developments and providing a detailed examination of both hardware and software approaches. Finally, [139] discusses the design and architectures for dependable embedded systems, focusing on a co-design approach that spans various levels of abstraction in the design process. While this approach is essential for ensuring dependability, our survey provides a more granular analysis of specific fault-tolerance techniques and their applications in embedded systems.

REFERENCES

[1] E. Dubrova, *Fault-tolerant Design*. Cham, Switzerland: Springer, 2013.
[2] B. Fleming, "Microcontroller units in automobiles [automotive electronics]," *IEEE Veh. Technol. Mag.*, vol. 6, no. 3, pp. 4–8, Sep. 2011.
[3] J. P. Trovao, "Trends in automotive electronics [automotive electronics]," *IEEE Veh. Technol. Mag.*, vol. 14, no. 4, pp. 100–109, Dec. 2019.
[4] J. C. Knight, "Safety critical systems: Challenges and directions," in *Proc. 24th Int. Conf. Softw. Eng.*, May 2002, pp. 547–550.
[5] H. Ardebili, J. Zhang, and M. G. Pecht, "10 - trends and challenges," in *Encapsulation Technologies for Electronic Applications* (Materials and processes for electronic applications), 2nd ed., H. Ardebili, J. Zhang, and M. G. Pecht, Eds., Norwich, NY, USA: William Andrew Publishing, 2019, pp. 431–479. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780128119785000109
[6] S. Khan, S. Hamdioui, H. Kukner, P. Raghavan, and F. Catthoor, "BTI impact on logical gates in nano-scale CMOS technology," in *Proc. IEEE 15th Int. Symp. Design Diag. Electron. Circuits Syst. (DDECS)*, Apr. 2012, pp. 348–353.
[7] S. Hamdioui, D. Gizopoulos, G. Guido, M. Nicolaidis, A. Grasset, and P. Bonnot, "Reliability challenges of real-time systems in forthcoming technology nodes," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2013, pp. 129–134.
[8] I. Agbo, M. Taouil, S. Hamdioui, P. Weckx, S. Cosemans, F. Catthoor, and W. Dehaene, "Read path degradation analysis in SRAM," in *Proc. 21th IEEE Eur. Test Symp. (ETS)*, May 2016, pp. 1–2.
[9] S. Pae, J. Maiz, C. Prasad, and B. Woolery, "Effect of BTI degradation on transistor variability in advanced semiconductor technologies," *IEEE Trans. Device Mater. Rel.*, vol. 8, no. 3, pp. 519–525, Sep. 2008.
[10] K. Iniewski, *Radiation Effects in Semiconductors*. Boca Raton, FL, USA: CRC Press, 2018.
[11] D. Oliveira, S. Blanchard, N. Debardeleben, F. F. Dos Santos, G. P. Dávila, P. Navaux, C. Cazzaniga, C. Frost, R. C. Baumann, and P. Rech, "Thermal neutrons: A possible threat for supercomputers and safety critical applications," in *Proc. IEEE Eur. Test Symp. (ETS)*, May 2020, pp. 1–6.
[12] D. Gil-Tomás, J. Gracia-Morán, J.-C. Baraza-Calvo, L.-J. Saiz-Adalid, and P.-J. Gil-Vicente, "Studying the effects of intermittent faults on a microcontroller," *Microelectron. Rel.*, vol. 52, no. 11, pp. 2837–2846, Nov. 2012. https://www.sciencedirect.com/science/article/pii/S0026271412001886
[13] M. Radetzki, C. Feng, X. Zhao, and A. Jantsch, "Methods for fault tolerance in networks-on-chip," *ACM Comput. Surveys*, vol. 46, no. 1, pp. 1–38, Jul. 2013, doi: 10.1145/2522968.2522976.
[14] S. Borkar, N. P. Jouppi, and P. Stenstrom, "Microprocessors in the era of terascale integration," in *Proc. Design, Autom. Test Eur. Conf. Exhib.*, Apr. 2007, pp. 1–6.
[15] E. Petersen, *Single Event Effects in Aerospace*. Hoboken, NJ, USA: Wiley, 2011.
[16] R. Gaillard, "Single event effects: Mechanisms and classification," in *Soft Errors in Modern Electronic Systems*. Cham, Switzerland: Springer, 2011, pp. 27–54.
[17] F. L. Kastensmidt, L. Carro, and R. A. da Luz Reis, *Fault-Tolerance Techniques for SRAM-based FPGAs*, vol. 1. Cham, Switzerland: Springer, 2006.
[18] R. C. Baumann, "Radiation-induced soft errors in advanced semiconductor technologies," *IEEE Trans. Device Mater. Rel.*, vol. 5, no. 3, pp. 305–316, Sep. 2005.
[19] F. Wang and V. D. Agrawal, "Single event upset: An embedded tutorial," in *Proc. 21st Int. Conf. VLSI Design (VLSID)*, Jan. 2008, pp. 429–434.
[20] R. Koga, S. H. Penzin, K. B. Crawford, and W. R. Crain, "Single event functional interrupt (SEFI) sensitivity in microcircuits," in *Proc. RADECS 97. 4th Eur. Conf. Radiat. Effects Compon. Syst.*, Sep. 1997, pp. 311–318.
[21] M. Violante, C. Meinhardt, R. Reis, and M. Sonza Reorda, "A low-cost solution for deploying processor cores in harsh environments," *IEEE Trans. Ind. Electron.*, vol. 58, no. 7, pp. 2617–2626, Jul. 2011.
[22] R. Velazco, P. Fouillat, and R. Reis, *Radiation Effects on Embedded Systems*. Heidelberg, Germany: Springer, 2007.
[23] F. Kastensmidt and P. Rech, "Radiation effects and fault tolerance techniques for FPGAs and GPUs," in *FPGAs and Parallel Architectures for Aerospace Applications: Soft Errors and Fault-Tolerant Design*. Cham, Switzerland: Springer, 2016, pp. 3–17.
[24] A. B. D. Oliveira. (2017). *Applying Dual-core Lockstep in Embedded Processors To Mitigate Radiation-induced Soft Errors*. [Online]. Available: https://www.lume.ufrgs.br/bitstream/handle/10183/173785/001061371.pdf?sequence=1
[25] V. P. Nelson, "Fault-tolerant computing: Fundamental concepts," *Computer*, vol. 23, no. 7, pp. 19–25, Jul. 1990.
[26] Z. Gao, C. Cecati, and S. X. Ding, "A survey of fault diagnosis and fault-tolerant techniques—Part I: Fault diagnosis with model-based and signal-based approaches," *IEEE Trans. Ind. Electron.*, vol. 62, no. 6, pp. 3757–3767, Jun. 2015.
[27] S. Ravi, A. Raghunathan, P. Kocher, and S. Hattangady, "Security in embedded systems: Design challenges," *ACM Trans. Embedded Comput. Syst.*, vol. 3, no. 3, pp. 461–491, Aug. 2004.
[28] P. Koopman, *Better Embedded System Software*. Pittsburgh, PA, USA: Drumnadrochit Education, 2010.
[29] O. Goloubeva, M. Rebaudengo, M. S. Reorda, and M. Violante, *Software-Implemented Hardware Fault Tolerance*. Springer, 2006.
[30] I. Koren and C. M. Krishna, *Fault-Tolerant Systems*. San Mateo, CA, USA: Morgan Kaufmann, 2020.
[31] M. A. Kochte and H.-J. Wunderlich, "Self-test and diagnosis for self-aware systems," *IEEE Des. Test. IEEE Des. Test*, vol. 35, no. 5, pp. 7–18, Oct. 2018.

[32] I. Yarza, I. Agirre, I. Mugarza, and J. P. Cerrolaza, "Safety and security collaborative analysis framework for high-performance embedded computing devices," *Microprocessors Microsyst.*, vol. 93, Sep. 2022, Art. no. 104572. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0141933122001247

[33] B. Kada and H. Kalla, "An efficient fault-tolerant scheduling approach with energy minimization for hard real-time embedded systems," in *Proc. Int. Workshop Distrib. Comput. for Emerg. Smart Netw.* Cham, Switzerland: Springer, 2019, pp. 102–117.

[34] S. Safari, M. Ansari, H. Khdr, P. Gohari-Nazari, S. Yari-Karin, A. Yeganeh-Khaksar, S. Hessabi, A. Ejlali, and J. Henkel, "A survey of fault-tolerance techniques for embedded systems from the perspective of power, energy, and thermal issues," *IEEE Access*, vol. 10, pp. 12229–12251, 2022.

[35] I. Koren and C. M. Krishna, *Fault-tolerant Systems*. San Mateo, CA, USA: Morgan Kaufmann, 2020.

[36] S. Deepanjali, A. Shaik, S. Noor Mahammad, and S. Beautlin, "Evolvable hardware for fault mitigation in control circuits," in *Proc. 37th Int. Conf. VLSI Design 23rd Int. Conf. Embedded Syst. (VLSID)*, Jan. 2024, pp. 672–677.

[37] C. Gauer, B. J. LaMeres, and D. Racek, "Spatial avoidance of hardware faults using FPGA partial reconfiguration of tile-based soft processors," in *Proc. IEEE Aerosp. Conf.*, Mar. 2010, pp. 1–11.

[38] M. Abdullah Soyturk, K. Parasyris, B. Salami, O. Unsal, G. Yalcin, and L. Bautista Gomez, "Hardware versus software fault injection of modern undervolted SRAMs," 2019, *arXiv:1912.00154*.

[39] K. Skadron, M. R. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan, "Temperature-aware microarchitecture," *ACM SIGARCH Comput. Archit. News*, vol. 31, no. 2, pp. 2–13, May 2003.

[40] S. Velusamy, W. Huang, J. Lach, M. Stan, and K. Skadron, "Monitoring temperature in FPGA based SoCs," in *Proc. Int. Conf. Comput. Design*, Oct. 2005, pp. 634–637.

[41] K.-S. Min, H.-D. Choi, H.-Y. Choi, H. Kawaguchi, and T. Sakurai, "Leakage-suppressed clock-gating circuit with zigzag super cut-off CMOS (ZSCCMOS) for leakage-dominant sub-70-nm and sub-1-V-V_{DD} LSIs," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 14, no. 4, pp. 430–435, Apr. 2006.

[42] J. Kim, M. Sullivan, S.-L. Gong, and M. Erez, "Frugal ECC: Efficient and versatile memory error protection through fine-grained compression," in *SC, Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, Nov. 2015, pp. 1–12.

[43] S.-C. Chen, "Research on electrical automation monitoring system model of power plant based on CAN bus," *J. Electr. Comput. Eng.*, vol. 2022, pp. 1–11, Apr. 2022.

[44] F. Cheng, A. Liu, J. Ren, and K. Feng, "CRC-aided parity-check polar coding," *IEEE Access*, vol. 7, pp. 155574–155583, 2019.

[45] M. Pohronská and T. Krajčovič, "FPGA implementation of multiple hardware watchdog timers for enhancing real-time systems security," in *Proc. IEEE EUROCON - Int. Conf. Comput. Tool*, Apr. 2011, pp. 1–4.

[46] A. Mahmood and E. J. McCluskey, "Concurrent error detection using watchdog processors–A survey," *IEEE Trans. Comput.*, vol.TC 37, no. 2, pp. 160–174, Feb. 1988.

[47] T. M. Austin, "DIVA: A reliable substrate for deep submicron microarchitecture design," in *MICRO-32. Proc. 32nd Annu. ACM/IEEE Int. Symp. Microarchit.*, Nov. 1999, pp. 196–207.

[48] C. A. Lisboa, M. I. Erigson, and L. Carro, "System level approaches for mitigation of long duration transient faults in future technologies," in *Proc. 12th IEEE Eur. Test Symp. (ETS)*, May 2007, pp. 165–172.

[49] F. Saglietti, "Strategies for the achievement and assessment of software fault-tolerance," *IFAC Proc. Volumes*, vol. 23, no. 8, pp. 303–308, Aug. 1990.

[50] B. Randell and J. Xu, "The evolution of the recovery block concept," *Softw. Fault Tolerance*, vol. 3, pp. 1–22, Mar. 1995.

[51] A. Avizienis, "The methodology of N-version programming," in *Software Fault Tolerance*, M. R. Lyu, Ed. Chichester, U.K.: Wiley, 1995, pp. 23–46.

[52] J.-C. Laprie, J. Arlat, C. Béounes, and K. Kanoun, "Definition and analysis of hardware-and-software fault-tolerant architectures," in *Predictably Dependable Computing Systems*. Springer, 1995, pp. 103–122.

[53] R. K. Scott, J. W. Gault, and D. F. McAllister, "Fault-tolerant software reliability modeling," *IEEE Trans. Softw. Eng.*, vol. SE-13, no. 5, pp. 582–592, May 1987.

[54] D. K. Pradhan, *Fault-Tolerant Computer System Design*, vol. 132. Englewood Cliffs, NJ, USA: Prentice-Hall, 1996.

[55] W. Torres-Pomales, "Software fault tolerance: A tutorial," NASA Langley Res. Center, Nat. Aeronaut. Space Admin. (NASA), Hampton, VA, USA, Tech. Rep. NASA/TM-2000-210616, 2000.

[56] M. Z. Rela, H. Madeira, and J. G. Silva, "Experimental evaluation of the fail-silent behaviour in programs with consistency checks," in *Proc. Annu. Symp. Fault Tolerant Comput.*, Jun. 1996, pp. 394–403.

[57] V. F. Nicola, "Checkpointing and the modeling of program execution time," Dept. of Computer Science and Dept., Univ. Twente, Enschede, The Netherlands, Tech. Rep. TR-CTIT-94-23, 1994.

[58] M. Russinovich and Z. Segall, "Fault-tolerance for off-the-shelf applications and hardware," in *Proc. 25th Int. Symp. Fault-Tolerant Comput. Dig. Papers*, Jun. 1995, pp. 67–71.

[59] E. Chielle, B. Du, F. L. Kastensmidt, S. Cuenca-Asensi, L. Sterpone, and M. S. Reorda, "Hybrid soft error mitigation techniques for COTS processor-based systems," in *Proc. 17th Latin-Amer. Test Symp. (LATS)*, Apr. 2016, pp. 99–104.

[60] J. R. Azambuja, M. Altieri, J. Becker, and F. L. Kastensmidt, "HETA: Hybrid error-detection technique using assertions," *IEEE Trans. Nucl. Sci.*, vol. 60, no. 4, pp. 2805–2812, Aug. 2013.

[61] F. Abate, L. Sterpone, and M. Violante, "A new mitigation approach for soft errors in embedded processors," in *Proc. 9th Eur. Conf. Radiat. Its Effects Compon. Syst.*, Sep. 2007, pp. 1–6.

[62] J. Gomez-Cornejo, A. Zuloaga, U. Kretzschmar, U. Bidarte, and J. Jimenez, "Fast context reloading lockstep approach for SEUs mitigation in a FPGA soft core processor," in *Proc. IECON - 39th Annu. Conf. IEEE Ind. Electron. Soc.*, Nov. 2013, pp. 2261–2266.

[63] H.-M. Pham, S. Pillement, and S. J. Piestrak, "Low-overhead fault-tolerance technique for a dynamically reconfigurable softcore processor," *IEEE Trans. Comput.*, vol. 62, no. 6, pp. 1179–1192, Jun. 2013.

[64] 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.

[65] A. Rhisheekesan, R. Jeyapaul, and A. Shrivastava, "Control flow checking or not? (for soft errors)," *ACM Trans. Embedded Comput. Syst. (TECS)*, vol. 18, no. 1, pp. 1–25, Feb. 2019.

[66] A. Shrivastava, A. Rhisheekesan, R. Jeyapaul, and C.-J. Wu, "Quantitative analysis of control flow checking mechanisms for soft errors," in *Proc. 51st ACM/EDAC/IEEE Design Autom. Conf. (DAC)*, Jun. 2014, pp. 1–6.

[67] A. Benso, S. Di Carlo, G. Di Natale, and P. Prinetto, "A watchdog processor to detect data and control flow errors," in *Proc. 9th IEEE On-Line Test. Symp., (IOLTS)*, Jul. 2003, pp. 144–148.

[68] A. Chaudhari, J. Park, and J. Abraham, "A framework for low overhead hardware based runtime control flow error detection and recovery," in *Proc. IEEE 31st VLSI Test Symp. (VTS)*, Apr. 2013, pp. 1–6.

[69] J. A. Abraham and R. Vemu, "Control flow deviation detection for software security," U.S. Patent 12 484 839, Dec. 31, 2009.

[70] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Trans. Dependable Secure Comput.*, vol. 1, no. 1, pp. 11–33, Jan. 2004.

[71] Z. Zhang, S. Park, and S. Mahlke, "Path sensitive signatures for control flow error detection," in *Proc. 21st ACM SIGPLAN/SIGBED Conf. Lang., Compil., Tools Embedded Syst.*, Jun. 2020, pp. 62–73.

[72] S. Mukherjee, *Architecture Design for Soft Errors*. San Mateo, CA, USA: Morgan Kaufmann, 2011.

[73] G. Bronevetsky, B. de Supinski, and M. Schulz, "A foundation for the accurate prediction of the soft error vulnerability of scientific applications," Dept. Lawrence Livermore National Lab. (LLNL), Livermore, CA, USA, Tech. Rep. 2009.

[74] S. Borkar, "Microarchitecture and design challenges for gigascale integration," in *Proc. 37th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Portland, OR, USA, Dec. 2004, p. 3.

[75] W. Bartlett and L. Spainhower, "Commercial fault tolerance: A tale of two systems," *IEEE Trans. Dependable Secure Comput.*, vol. 1, no. 1, pp. 87–96, Jan. 2004.

[76] D. Bernick, B. Bruckert, P. D. Vigna, D. Garcia, R. Jardine, J. Klecka, and J. Smullen, "Nonstop/spl reg/advanced architecture," in *Proc. Int. Conf. Dependable Syst. Netw. (DSN)*, 2005, pp. 12–21.

[77] Y. C. Yeh, "Triple-triple redundant 777 primary flight computer," in *IEEE Aerosp. Appl. Conference. Proc.*, vol. 1, 1996, pp. 293–307.

[78] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Control-flow checking by software signatures," *IEEE Trans. Rel.*, vol. 51, no. 1, pp. 111–122, Mar. 2002.

[79] R. Vemu and J. Abraham, "CEDA: Control-flow error detection using assertions," *IEEE Trans. Comput.*, vol. 60, no. 9, pp. 1233–1245, Sep. 2011.
[80] D. S. Khudia and S. Mahlke, "Low cost control flow protection using abstract control signatures," in *Proc. 14th ACM SIGPLAN/SIGBED Conf. Lang., Compil. Tools Embedded Syst.*, Jun. 2013, pp. 3–12.
[81] Z. Zhu, J. Callenes-Sloan, and B. Carrion Schafer, "Control flow checking optimization based on regular patterns analysis," in *Proc. IEEE 23rd Pacific Rim Int. Symp. Dependable Comput. (PRDC)*, Dec. 2018, pp. 203–212.
[82] L. Parra, A. Lindoso, M. Portela, L. Entrena, F. Restrepo-Calle, S. Cuenca-Asensi, and A. Martínez-Alvarez, "Efficient mitigation of data and control flow errors in microprocessors," *IEEE Trans. Nucl. Sci.*, vol. 61, no. 4, pp. 1590–1596, Aug. 2014.
[83] R. Ahmed, M. El Sayed, S. A. Gadsden, J. Tjong, and S. Habibi, "Automotive internal-combustion-engine fault detection and classification using artificial neural network techniques," *IEEE Trans. Veh. Technol.*, vol. 64, no. 1, pp. 21–33, Jan. 2015.
[84] B. Zheng, Y. Gao, Q. Zhu, and S. Gupta, "Analysis and optimization of soft error tolerance strategies for real-time systems," in *Proc. Int. Conf. Hardw./Softw. Codesign Syst. Synth. (CODES+ISSS)*, Oct. 2015, pp. 55–64.
[85] M. A. Solouki, J. Sini, and M. Violante, "An experimental evaluation of control flow checking for automotive embedded applications compliant with ISO 26262," *IEEE Access*, vol. 11, pp. 51185–51198, 2023.
[86] Z. Alkhalifa, V. S. S. Nair, N. Krishnamurthy, and J. A. Abraham, "Design and evaluation of system-level checks for on-line control flow error detection," *IEEE Trans. Parallel Distrib. Syst.*, vol. 10, no. 6, pp. 627–641, Jun. 1999.
[87] L. D. McFearin and V. S. Nair, "Control flow checking using assertions," *Dependable Comput. Fault Tolerant Syst.*, vol. 10, pp. 183–200, 1998.
[88] R. Venkatasubramanian, J. P. Hayes, and B. T. Murray, "Low-cost on-line fault detection using control flow assertions," in *Proc. 9th IEEE On-Line Test. Symp., IOLTS.*, 2003, pp. 137–143.
[89] O. Goloubeva, M. Rebaudengo, M. S. Reorda, and M. Violante, "Improved software-based processor control-flow errors detection technique," in *Proc. Annu. Rel. Maintainability Symp.*, 2005, pp. 583–589.
[90] S. A. Asghari, H. Taheri, H. Pedram, and O. Kaynak, "Software-based control flow checking against transient faults in industrial environments," *IEEE Trans. Ind. Informat.*, vol. 10, no. 1, pp. 481–490, Feb. 2014.
[91] E. Chielle, G. S. Rodrigues, F. L. Kastensmidt, S. Cuenca-Asensi, L. A. Tambara, P. Rech, and H. Quinn, "S-SETA: Selective software-only error-detection technique using assertions," *IEEE Trans. Nucl. Sci.*, vol. 62, no. 6, pp. 3088–3095, Dec. 2015.
[92] A. Li and B. Hong, "Software implemented transient fault detection in space computer," *Aerosp. Sci. Technol.*, vol. 11, nos. 2–3, pp. 245–252, Mar. 2007.
[93] J. Vankeirsbilck, N. Penneman, H. Hallez, and J. Boydens, "Random additive signature monitoring for control flow error detection," *IEEE Trans. Rel.*, vol. 66, no. 4, pp. 1178–1192, Dec. 2017.
[94] B. Nicolescu, Y. Savaria, and R. Velazco, "SIED: Software implemented error detection," in *Proc. Proceedings. 16th IEEE Symp. Comput. Arithmetic*, 2003, pp. 589–596.
[95] J. Vankeirsbilck, N. Penneman, H. Hallez, and J. Boydens, "Random additive control flow error detection," in *Proc. Int. Conf. Comput. Saf., Rel., Secur.* Springer, 2018, pp. 220–234.
[96] M. Maghsoudloo, H. R. Zarandi, and N. Khoshavi, "An efficient adaptive software-implemented technique to detect control-flow errors in multi-core architectures," *Microelectron. Rel.*, vol. 52, no. 11, pp. 2812–2828, Nov. 2012.
[97] O. Goloubeva, M. Rebaudengo, M. Sonza Reorda, and M. Violante, "Soft-error detection using control flow assertions," in *Proc. 18th IEEE Symp. Defect Fault Tolerance VLSI Syst.* Boston, MA, USA: IEEE, 2003, pp. 581–588.
[98] (2022). *Autosar: Specification of Watchdog Manager (2011).* [Online]. Available: https://www.autosar.org/fileadmin/standards/R21-11/CP/AUTOSAR_SWS_WatchdogManager.pdf
[99] (2022). *Autosar: Specification of Operating System (2011).* [Online]. Available: https://www.autosar.org/fileadmin/standards/R20-11/CP/AUTOSAR_SWS_OS.pdf
[100] I. Thomm, M. Stilkerich, R. Kapitza, D. Lohmann, and W. Schröder-Preikschat, "Automated application of fault tolerance mechanisms in a component-based system," in *Proc. 9th Int. Workshop Java Technol. Real-Time Embedded Syst.*, Sep. 2011, pp. 87–95.
[101] L. Cassano, A. Miele, F. Mione, N. Tonellotto, and C. Vallati, "Design of fault-tolerant distributed cyber–physical systems for smart environments," *IEEE Embedded Syst. Lett.*, vol. 14, no. 2, pp. 79–82, Jun. 2022.
[102] G. S. Rodrigues, F. L. Kastensmidt, and A. Bosio, *Approximate Computing and Its Impact on Accuracy, Reliability and Fault-Tolerance*. Springer, 2022.
[103] V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Analysis and characterization of inherent application resilience for approximate computing," in *Proc. 50th Annu. Design Autom. Conf.*, May 2013, pp. 1–9.
[104] A. Chatzidimitriou, G. Papadimitriou, C. Gavanas, G. Katsoridas, and D. Gizopoulos, "Multi-bit upsets vulnerability analysis of modern microprocessors," in *Proc. IEEE Int. Symp. Workload Characterization (IISWC)*. Orlando, FL, USA: IEEE, 2019, pp. 119–130.
[105] J. Ren, X. Chen, D. Liu, M. Duan, R. Liu, and C. Wang, "Forseti: An efficient basic-block-level sensitivity analysis framework towards multi-bit faults," in *Proc. Design, Autom. Test Eur. Conf. Exhibit. (DATE)*, Feb. 2021, pp. 94–97.
[106] H. Cho and K.-W. Kwon, "Modeling application-level soft error effects for single-event multi-bit upsets," *IEEE Access*, vol. 7, pp. 133485–133495, 2019.
[107] M. Fernandes, J. M. Corchado, and G. Marreiros, "Machine learning techniques applied to mechanical fault diagnosis and fault prognosis in the context of real industrial manufacturing use-cases: A systematic literature review," *Int. J. Speech Technol.*, vol. 52, no. 12, pp. 14246–14280, Sep. 2022.
[108] S. Zhang, S. Zhang, B. Wang, and T. G. Habetler, "Deep learning algorithms for bearing fault diagnostics—A comprehensive review," *IEEE Access*, vol. 8, pp. 29857–29881, 2020.
[109] A. Ruospo, E. Sanchez, L. M. Luza, L. Dilillo, M. Traiola, and A. Bosio, "A survey on deep learning resilience assessment methodologies," *Computer*, vol. 56, no. 2, pp. 57–66, Feb. 2023.
[110] Q. Hu, P. Sun, S. Yan, Y. Wen, and T. Zhang, "Characterization and prediction of deep learning workloads in large-scale GPU datacenters," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, Nov. 2021, pp. 1–15.
[111] Z. Xiang, Y. Bao, Z. Tang, and H. Li, "Deep reinforcement learning-based sampling method for structural reliability assessment," *Rel. Eng. Syst. Saf.*, vol. 199, Jul. 2020, Art. no. 106901.
[112] K. Khalil, O. Eldash, A. Kumar, and M. Bayoumi, "Intelligent fault-prediction assisted self-healing for embryonic hardware," *IEEE Trans. Biomed. Circuits Syst.*, vol. 14, no. 4, pp. 852–866, Aug. 2020.
[113] K. P. Gostelow, "The design of a fault-tolerant, real-time, multi-core computer system," in *Proc. Aerosp. Conf.*, Mar. 2011, pp. 1–8.
[114] V. A. Bogatyrev and A. V. Bogatyrev, "Functional reliability of a real-time redundant computational process in cluster architecture systems," *Autom. Control Comput. Sci.*, vol. 49, no. 1, pp. 46–56, Jan. 2015.
[115] P. Pop, V. Izosimov, P. Eles, and Z. Peng, "Design optimization of time- and cost-constrained fault-tolerant embedded systems with checkpointing and replication," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 17, no. 3, pp. 389–402, Mar. 2009.
[116] A. Sharif Ahmadian, M. Hosseingholi, and A. Ejlali, "A control-theoretic energy management for fault-tolerant hard real-time systems," in *Proc. IEEE Int. Conf. Comput. Design*, Oct. 2010, pp. 173–178.
[117] *Road Vehicles—Functional Safety*, document ISO 26262:2018, 2018.
[118] M. Obata, J. Shirako, H. Kaminaga, K. Ishizaka, and H. Kasahara, "Hierarchical parallelism control for multigrain parallel processing," in *Proc. 15th Int. Workshop Lang. Compil. Parallel Comput.*, College Park, MD, USA. Springer, Jul. 2005, pp. 31–44.
[119] F. Haas, "Fault-tolerant execution of parallel applications on x86 multi-core processors with hardware transactional memory," Ph.D. dissertation, Dept. Comput. Sci., Friedrich-Alexander-Universität Erlangen-NÃ¼rnberg, Erlangen, Germany, 2019.
[120] G. S. Veronese, M. Correia, A. N. Bessani, L. C. Lung, and P. Verissimo, "Efficient byzantine fault-tolerance," *IEEE Trans. Comput.*, vol. 62, no. 1, pp. 16–30, Jan. 2013.
[121] M. Baleani, A. Ferrari, L. Mangeruca, A. Sangiovanni-Vincentelli, M. Peri, and S. Pezzini, "Fault-tolerant platforms for automotive safety-critical applications," in *Proc. Int. Conf. Compil., Archit. Synth. Embedded Syst.*, Oct. 2003, pp. 170–177.
[122] S. K. Reinhardt and S. S. Mukherjee, "Transient fault detection via simultaneous multithreading," in *Proc. 27th Annu. Int. Symp. Comput. Archit.*, 2000, pp. 25–36.

[123] C. M. Jeffery and R. J. O. Figueiredo, "A flexible approach to improving system reliability with virtual lockstep," *IEEE Trans. Dependable Secure Comput.*, vol. 9, no. 1, pp. 2–15, Jan. 2012.
[124] M. Peña-Fernández, A. Serrano-Cases, A. Lindoso, S. Cuenca-Asensi, L. Entrena, Y. Morilla, P. Martín-Holgado, and A. Martínez-Álvarez, "Hybrid lockstep technique for soft error mitigation," *IEEE Trans. Nucl. Sci.*, vol. 69, no. 7, pp. 1574–1581, Jul. 2022.
[125] G. Dar, G. D. Natale, and O. Keren, "Nonlinear code-based low-overhead fine-grained control flow checking," *IEEE Trans. Comput.*, vol. 71, no. 3, pp. 658–669, Mar. 2022.
[126] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Trans. Inf. Syst. Secur.*, vol. 13, no. 1, pp. 1–40, Oct. 2009.
[127] K. Chandrasekaran, *Essentials of Cloud Computing*. Boca Raton, FL, USA: CRC Press, 2014.
[128] K. D. Bowers, A. Juels, and A. Oprea, "Proofs of retrievability: Theory and implementation," in *Proc. ACM Workshop Cloud Comput. Secur.*, Nov. 2009, pp. 43–54.
[129] K. D. Bowers, A. Juels, and A. Oprea, "HAIL: A high-availability and integrity layer for cloud storage," in *Proc. 16th ACM Conf. Comput. Commun. Secur.*, New York, NY, USA, Nov. 2009, pp. 187–198, doi: 10.1145/1653662.1653686.
[130] J. Schiffman, T. Moyer, H. Vijayakumar, T. Jaeger, and P. McDaniel, "Seeding clouds with trust anchors," in *Proc. ACM Workshop Cloud Comput. Secur. Workshop*, Oct. 2010, pp. 43–46.
[131] A. Benso and P. Prinetto, *Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation*, vol. 23. Springer, 2003.
[132] J. M. Voas and G. McGraw, *Software Fault Injection: Inoculating Programs Against Errors*. Hoboken, NJ, USA: Wiley, 1997.
[133] H. Ziade, R. A. Ayoubi, and R. Velazco, "A survey on fault injection techniques," *Int. Arab J. Inf. Technol.*, vol. 1, no. 2, pp. 171–186, 2004.
[134] H. M. Quinn, D. A. Black, W. H. Robinson, and S. P. Buchner, "Fault simulation and emulation tools to augment radiation-hardness assurance testing," *IEEE Trans. Nucl. Sci.*, vol. 60, no. 3, pp. 2119–2142, Jun. 2013.
[135] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, "Fault injection techniques and tools," *Computer*, vol. 30, no. 4, pp. 75–82, Apr. 1997.
[136] S. X. Ding, *Model-based Fault Diagnosis Techniques: Design Schemes, Algorithms, and Tools*. Springer, 2008.
[137] L. Entrena, M. García-Valderas, A. Lindoso, M. Portela-Garcia, and E. San Millán, "Fault injection methodologies," in *Radiation Effects on Integrated Circuits and Systems for Space Applications*. Cham, Switzerland: Springer, 2019, pp. 127–144, doi: 10.1007/978-3-030-04660-6_6.
[138] K. Almakadmeh and G. Al Qahmouss, "A comprehensive survey of system dependability for real time embedded software," in *Proc. Int. Conf. Intell. Inf. Process., Secur. Adv. Commun.* Batna, Algeria: ACM, 2015, pp. 1–7.
[139] J. Henkel et al., "Design and architectures for dependable embedded systems," in *Proc. 7th IEEE/ACM/IFIP Int. Conf. Hardw./Softw. Codesign Syst. Synth.*, 2011, pp. 69–78.

MOHAMMADREZA AMEL SOLOUKI (Member, IEEE) received the B.Sc. degree in computer engineering from the Islamic Azad University of Mashhad, Mashhad, Iran, in 2013, and the M.Sc. degree in computer architecture from the Department of Computer Engineering, Qazvin Islamic Azad University, Qazvin, Iran, in 2017, and the Ph.D. degree in control and computer engineering from the Department of Control and Computer Engineering (DAUIN), Politecnico di Torino, Turin, Italy.

His research focuses on design for test (DFT), fault tolerance, and reliability in embedded systems. He has been developing software solutions to enhance system reliability and functional safety (ISO 26262), particularly for automotive and industrial applications.

SHAAHIN ANGIZI (Senior Member, IEEE) received the Ph.D. degree in electrical engineering from the School of Electrical, Computer and Energy Engineering, Arizona State University (ASU), Tempe, AZ, USA, in 2021. He is currently an Assistant Professor with the Department of Electrical and Computer Engineering, New Jersey Institute of Technology (NJIT), Newark, NJ, USA; and the Director of the Advanced Circuit-to-Architecture Design (ACAD) Laboratory. He has authored or co-authored more than 130 research papers in top-ranked journals and EDA conferences. His primary research interests include ultra-low-power in-memory computing based on volatile and non-volatile memories, in-sensor computing for IoT, brain-inspired (neuromorphic) computing, and accelerator design for deep neural networks and bioinformatics. He was a recipient of the Best Ph.D. Research Award (first-place) of Ph.D. Forum at IEEE/ACM DAC, in 2018; two Best Paper Awards of IEEE ISVLSI, in 2017 and 2018; and the Best Paper Awards of ACM GLSVLSI, in 2019 and 2023.

MASSIMO VIOLANTE (Member, IEEE) received the M.S. and Ph.D. degrees from the Politecnico di Torino, Italy. He is currently an Associate Professor with the Politecnico di Torino. From 2008 to 2009, he was a Research Assistant with the Institute of Physics, Academia Sinica, Tapei, Taiwan. He is very active in technological transfer activities with industries in the automotive and industrial sectors on topics, such as functional safety, model-based design, and embedded system design and validation. He is scientific responsible of several research projects in the area of embedded systems for space, avionic, and automotive applications in collaboration with the European Space Agency, Thales Alenia Space, ITT, CNH Industrial, ELDOR, and Magneti Marelli. He has published more than 150 articles in the area of testing and designing reliable embedded systems, and co-authored two books. His main research interests include the design and validation of embedded system for safety- and mission-critical applications, with an emphasis on the use of commercial off-the-shelf components, such as multi core processors and field programmable gate arrays in automotive, avionic, and space applications. His research interests include the development of surface processing and biological/medical treatment techniques using nonthermal atmospheric pressure plasmas, fundamental study of plasma sources, and fabrication of micro- or nanostructured surfaces. He served as the Program Co-Chair and the General Co-Chair for the IEEE Defect and Fault Tolerance in VLSI and Nanotechnology Systems, in 2011 and 2012; and the Program Chair for the IEEE European Test Symposium, in 2012.

Open Access funding provided by 'Politecnico di Torino' within the CRUI CARE Agreement