# Software Fault Tolerance in Real-Time Systems: Identifying the Future Research Questions

FEDERICO REGHENZANI, DEIB, Politecnico di Milano, ITA and ESTEC, European Space Agency, NLD
ZHISHAN GUO, Department of Computer Science, North Carolina State University, USA
WILLIAM FORNACIARI, DEIB, Politecnico di Milano, ITA

Tolerating hardware faults in modern architectures is becoming a prominent problem due to the miniaturization of the hardware components, their increasing complexity, and the necessity to reduce costs. Software-Implemented Hardware Fault Tolerance approaches have been developed to improve system dependability regarding hardware faults without resorting to custom hardware solutions. However, these come at the expense of making the satisfaction of the timing constraints of the applications/activities harder from a scheduling standpoint. This article surveys the current state-of-the-art of fault tolerance approaches when used in the context of real-time systems, identifying the main challenges and the cross-links between these two topics. We propose a joint scheduling-failure analysis model that highlights the formal interactions among software fault tolerance mechanisms and timing properties. This model allows us to present and discuss many open research questions with the final aim to spur future research activities.

CCS Concepts: • **Computer systems organization** → **Real-time systems**; **Dependable and fault-tolerant systems and networks**; • **Software and its engineering** → **Software fault tolerance**; *Real-time systems software;*

Additional Key Words and Phrases: Real-time, fault-tolerance, mixed-criticality

**ACM Reference format:**
Federico Reghenzani, Zhishan Guo, and William Fornaciari. 2023. Software Fault Tolerance in Real-Time Systems: Identifying the Future Research Questions. *ACM Comput. Surv.* 55, 14s, Article 306 (July 2023), 30 pages.
https://doi.org/10.1145/3589950

## 1 INTRODUCTION AND MOTIVATION

Hard real-time systems must satisfy both temporal and logical requirements. At the design phase, system correctness is usually proved, for the former, by a scheduling analysis and, for the latter, by formal verification or extensive testing. However, even assuming flawless design of the

This work has received funding from the European Space Agency (OSIP grant no. 4000133770/21/NL/MH/hm), ICSC National Research Center in High-Performance Computing, Big Data and Quantum Computing, and NSF (grant no. CCF-2028481).
Authors' addresses: F. Reghenzani, DEIB, Politecnico di Milano, via Ponzio 34/5, Milano, ITA, 20133, ESTEC, European Space Agency, Keplerlaan 1, Noordjwick, NLD, 2201; email: federico.reghenzani@polimi.it; Z. Guo, Department of Computer Science, North Carolina State University, 2262 EB II, 890 Oval Dr, Raleigh, North Carolina, USA, 27606; email: zguo32@ncsu.edu; W. Fornaciari, DEIB, Politecnico di Milano, via Ponzio 34/5, Milano, ITA, 20133; email: william.fornaciari@polimi.it.

0360-0300/2023/07-ART306 $15.00
https://doi.org/10.1145/3589950

system, hardware can still be the cause of violation of these constraints due to its natural susceptibility to faults. In critical systems, it is essential to guarantee a certain degree of resilience to hardware faults. No system can tolerate an unlimited amount of hardware faults: safety-critical standards dictate precise fault rates that functions performed by the system must achieve to be considered compliant. This necessary dependability can be achieved by employing hardware solutions that mitigate or tolerate the faults, for instance, by redundancy. Alternatively, software techniques can be employed with the crucial advantage of not requiring any custom hardware components. These software techniques, when applied to hardware faults, are usually under the umbrella term *Software-Implemented Hardware Fault Tolerance* (abbreviated with different acronyms, SIFT [104], SWIFT [87], or SIHFT [37]). SIFT solutions have a positive impact on hardware cost and production development. This convenience becomes a vital requirement for Commercial Off-The-Shelf (COTS) hardware in which strict fault tolerance is not a common design goal. The availability of COTS hardware led to significant trends in both the automotive and aerospace sectors over the past few years in order to reduce design costs and time to market. Even space agencies have active working groups studying how to integrate COTS hardware in critical systems [70, 80].

The computational power demand of emerging applications is increasing, and hardware manufacturers are striving to increase the computational capability of the computing platforms. However, this process is obstructed by the single-core performance barrier, mainly due to thermal and power constraints, which make increasing clock frequency difficult. Therefore, to achieve the desired performance, the hardware components are becoming increasingly smaller and implement complex features (pipelines, multicore, multilevel caches, etc.). These undaunted trends have a double effect: increasing the fault rate and making the real-time problem of formally guaranteeing the temporal requirements harder. In turn, they create a challenging environment [21]: on one hand, the increased fault rate pushes for SIFT approaches; on the other hand, the timing problem makes their use difficult. The major difficulty in using modern hardware in critical systems is computation of the Worst-Case Execution Time (WCET) of the tasks, an essential metric to perform a correct scheduling analysis and, therefore, prove the temporal correctness. A partial solution to the WCET problem is the mixed-criticality model for real-time tasks, as described later in Section 2.6.

**Motivation**. Software fault tolerance algorithms have a clear impact on real-time performance: the necessity to reserve computational resources and time for recovery routines makes scheduling more challenging. Several state-of-the-art fault-tolerant solutions exist, and many real-time scheduling algorithms have been developed in the last 40 years. However, analyzing the real-time characteristics of fault tolerance algorithms and improving scheduling in this direction received less attention from the research community. As we present later in this article, the opposite problem – how scheduling decisions impact fault tolerance – has been analyzed by very few works. There is a great potential for the research community to address the many open challenges on fault-tolerant real-time systems. Special focus can be given to mixed-criticality systems, an attractive future direction as also stated in the Burns and Davis survey [18]: "Although there is clear link between Fault-Tolerant and Mixed-Criticality, there has not yet been much work published that directly addresses fault-tolerant mixed criticality systems."

**Contributions**. This article summarizes the relevant literature, identifies current open challenges, and proposes possible future directions for research on real-time fault-tolerant systems, including mixed-criticality systems. The focus of the article is on SIFT approaches, that is, software techniques to tolerate hardware faults. SIFT and, consequently, this article do not deal with software

faults, either functional (e.g., bugs) or timing (e.g., incorrect estimation of the WCET). Rather, the focus is on resilience to hardware faults. In this survey, we:

- Introduce the basic concepts of fault tolerance and related topics, in addition to providing a background of real-time modeling (Section 2);
- Review the current literature on (hard) real-time scheduling analyses and techniques when fault tolerance — in particular, SIFT — is considered (Section 3);
- Propose a joint model for scheduling and failure analyses for SIFT, highlighting the interaction between the two analyses (Section 4);
- Thanks to the joint model, outline the current challenges and future possible research directions for fault-tolerant real-time systems (Section 5).

The goal of this article is not to survey the fault-tolerance approaches, nor the real-time scheduling algorithms, for which many literature reviews are available.[1] Instead, we focus on the intersection of these two topics, particularly on the identification of the open problems.

## 2 BACKGROUND

This section provides the knowledge on reliability engineering theory needed for understanding the issues and clearly defines all of the related terminology.

### 2.1 Terminology

*Fault*, *error*, and *failure* are terms sometimes used interchangeably, but there is a clear distinction between them. While different definitions exist, in this article, we use the ones from the IEEE Standard 610.12 [48]:

- *A **fault** is a defect in a hardware device or component.* This definition includes any variation of the actual behavior from the designed behavior, for example, a bit flip or a broken transistor. There are many models for faults. The most common one for transient faults is the *Single Event Upset (SEU)*, when a bit is accidentally flipped in the memory: we expect a 1 but it is actually 0 or vice versa. Many other models exist, such as *Multiple Event Upset (MEU)* or *Single Hard Error (SHE)*.
- *An **error** is the difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition.* The error is the effect on the system caused by the realization of a fault. An example is an SEU in the memory location containing the value read from a distance sensor, which changes the computed distance from the real "10 meters" to the erroneous "26 meters." In contrast to a fault, an error is a deviation in the software functionality.
- *A **failure** is the inability of a system or component to perform its required functions within specified performance requirements.* A failure occurs when the system does not perform according to its requirements due to the presence of one or more errors. Considering the previous example, if the system is required to carry out the distance value with an accuracy of 1 meter, the "26 meters" output is a violation of the system requirements and, thus, a failure.

From the previous definitions, it is possible to notice that a fault can exist yet it does not produce an error — for example, an SEU in an unused memory region —and an error can exist while it does not produce a failure — for example, when the erroneous value in a variable does not impact the system output or the output is still compliant with the accuracy requirements.

[1] We omitted citations here due to the large number of survey articles available in the literature for these two vast fields. A quick search with keywords will return numerous results by major publishers.

Table 1. Possible Criteria for Fault Classification Proposed by Avižienis et al. [5] and Classes Featured in This Article (Third Column)

| Criterion | Possibility | Selected |
|---|---|---|
| Phase | development/operational | **operational** |
| Location | internal/external | **internal** |
| Phenomenological cause | natural/human-made | **natural** |
| Dimension | software/hardware | **hardware**[*] |
| Temporal persistence | permanent/transient | *all* |
| Objective | malicious/non-malicious | **non-malicious**[†] |
| Intent | deliberate/non-deliberate | *irrelevant* |
| Capacity | accidental/incompetence | *irrelevant* |

[*] Hardware refers to the cause of the faults, but the presence of a hardware fault may impact the software behavior.
[†] Although malicious faults are not the main subject of this work, a brief discussion is available in Section 5.7.

### 2.2 Faults Taxonomy

Different ways to classify the characteristics of faults exist. For instance, Avižienis et al. [5] propose eight different criteria to classify faults as delineated in Table 1. These criteria lead to a combination of 256 different fault classes. In order to clearly define the boundaries of the discussion in this article, we restrict our analysis according to the third column of Table 1. The choices have been driven by the necessity to focus on the faults that are (1) not directly linked to human activities and, therefore, subject to nondeterministically quantifiable uncertainties; (2) possible to be modeled with a joint failure-timing model; (3) related to the operational phase of the system, thus, not considering any design mistake. This selection also allows us to focus on the SIFT techniques because we consider only the hardware as a fault source.

The *temporal persistence* criterion plays an important role in the subsequent discussion of modeling faults in real-time systems. Avižienis et al. [5] identified two classes for this criterion: transient and permanent. In the following, we characterize these two classes and add another class: intermittent faults.

**Transient faults**. Transient faults are defined as the faults that produce a temporary effect on the system, and their causes disappear (almost) immediately. The SEU model is the most commonly used representation of these faults. They can originate from three main causes [44]:

(1) Alpha radiations emitted from impurities of the materials used for the chip package that contain spurious radioactive isotopes.
(2) High-energy cosmic rays that, when hitting the hardware components, emit alpha and gamma radiations.
(3) Neutron-induced fission, caused by boron atoms used in the reflow soldering process or p-doped state of the transistors, emitting alpha and gamma radiations.

The probability of events caused by (1) and (3) can be reduced (but not eliminated) by improving manufacturing processes, whereas the events caused by (2) can be only partially reduced by shielding the hardware. However, large and heavy shields are often unfeasible from weight and size standpoints, especially for aerospace applications. In addition to these causes, poorly designed hardware may introduce transient faults caused by, for example, signal cross-talks, electromagnetic effects of memory cells [56], or improperly filtered power fluctuations [36]. The continuous technological advancements in decreasing hardware size and the increasing number of transistors

make computing systems more susceptible to transient faults. It has been observed [63] that on a standard server DRAM without Error-Correcting Code (ECC), the fault rate is in the order of $10^{-5}$ per hour. Approximately 30% of these faults caused a failure in software functionalities. However, in the space environment or with field-programmable gate array (FPGA) devices, this probability value is significantly higher, reaching $10^{-2}$ per hour [71, 96]. Due to the increasing susceptibility to transient faults of modern architectures, some recent works do not limit the analysis to a single event. Rather, they also consider MEU faults [3, 7]. The use of error-correcting devices (such as ECC memories) makes a system resilient to SEUs transparently with respect to the software, and some approaches can guarantee partial resilience to MEUs. Their implementation, however, requires a specialized hardware implementation that clashes with the goal of using COTS hardware for critical applications.

**Intermittent faults**. Intermittent faults are similar to transient faults regarding the persistence property, but they tend to occur in *fault bursts*, that is, a continuous and rapid sequence of transient faults. They are mainly caused by external environmental conditions, such as temperature and voltage variations, hardware issues (both manufacturing defects and wear out), or High-Intensity Radiated Fields (HIRF) [39, 67]. The latter case is a sensitive problem for avionics applications [4], especially in proximity to airports due to the presence of a large number of RF devices. While temperature-, voltage-, or RF-induced faults tend to occur in random locations, hardware-related intermittent faults repeatedly affect the same location. This particular case of hardware-related faults has been studied in the works of Constantinescu [24, 25] and of Gracia-Moran et al. [39].

**Permanent faults**. Permanent faults have been extensively studied in the literature, and they are characterized by the fact that a permanent fault continues to exist until the component is repaired or replaced. Hardware is always subject to faults caused by *infant mortality* (early-stage faults), random uncontrollable faults, and component wear out, which comprise the well-known *bathtub curve* hazard function [105]. In addition, environmental effects may be sufficiently powerful to break down one transistor or other components. A resource that suffers from a permanent fault is usually considered full-failed and removed from the pool of available resources. In 2018, Gunawi et al. [41] proposed[2] a further classification of permanent faults that may allow the resource to remain available in degraded modes:

- *Fail-Stop*: The resource/component is completely not usable. For example, one of the cores of the processor stops working.
- *Fail-Partial*: One part of the resource is offline, whereas another part is still usable. An example is a stuck bit in a memory location: that specific cell is considered failed, but we can continue to use the remaining memory by remapping the task memory spaces.
- *Fail-Slow*: The resource components still work but at reduced performance. Two examples: (1) a fault in the cooling system that forces the operating system to reduce the frequency of processors to keep the temperature in an acceptable range; and (2) the failure of one out of two requires the shutdown of some computing components to remain in the power budget.

Clearly, the difference between Fail-Stop and Fail-Partial depends on the granularity that we select for the resource. For example, if a single processor is considered to be one single resource, it means that the failure of one core is a Fail-Partial fault, whereas if the core is considered to be a single resource, its failure is considered to be Fail-Stop.

[2]The work focused on large-scale systems, but the concepts can also be applied to embedded systems.

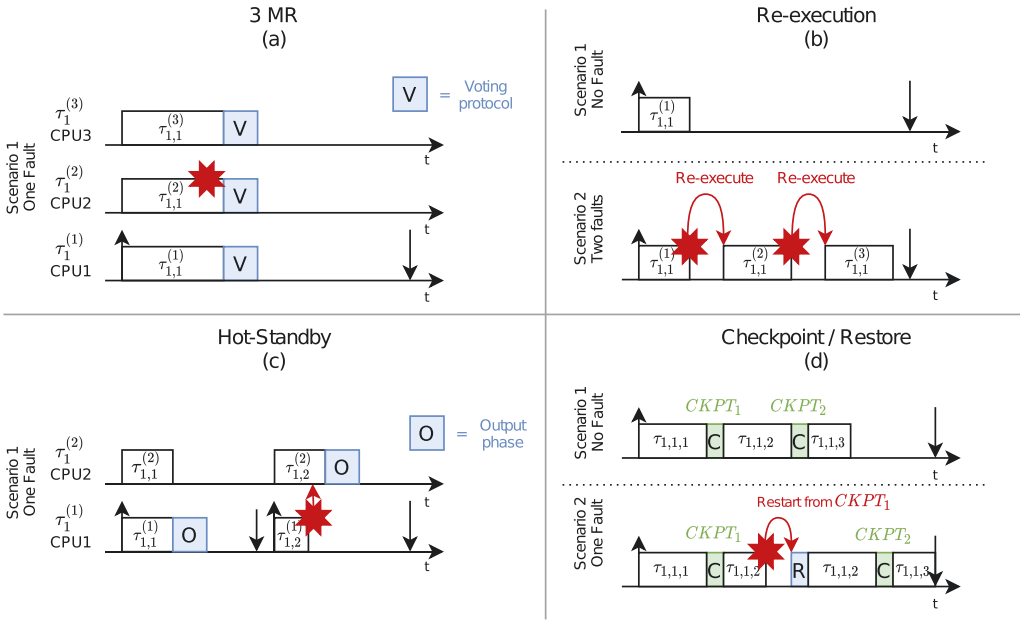


Fig. 1. The four main software fault recovery approaches. In (a), (b), and (c), the symbol $\tau_{i,j}^{(k)}$ represents the $j$-th job of the $i$-th task, and $k$ identifies the redundancy or re-execution job. Instead, in (d), $\tau_{i,j,k}$ is the $k$-th part of the job $\tau_{i,j}$. In the depicted example, the job $\tau_{1,1}$ is composed of three parts.

It should be noted that some classes of faults are categorized as permanent or transient depending on the context. For example, certain Single Event Latchup (SEL) faults can be restored by cycling the power of the device. However, in the context of hard real-time systems, such faults are usually considered permanent because, in most applications, the reboot delay potentially causes a deadline miss or the loss of data.

### 2.3 Fault Recovery

Software fault recovery techniques are categorized in *space redundancy* and *time redundancy*. The former resembles the traditional hardware fault tolerance techniques by replicating the workload over different resources. The latter consists of replicating the workload on different time periods, possibly on the same resource. The four main approaches are depicted in Figure 1.

**Space redundancy techniques**. The *N-modular redundancy (NMR)* (also called *active redundancy* [48]) technique is the simplest approach to software fault tolerance, similar to traditional hardware redundancy. Each critical task is replicated N times (e.g., 2MR, 3MR). The main task and the replica tasks perform exactly the same operations, and a voting mechanism decides which output is the correct one. The voter can be implemented in hardware or software. In this article, we focus on software voters, whose taxonomy is described in the survey-style article by Latif-Shabgahi et al. [60]. The redundancy tasks usually run on different memory spaces to be resilient with SEUs, and in the case of availability on multicores/processors, they run on different computational units. The number of replicas is usually immutable. For this reason, N-MR is also categorized in static-redundancy techniques. Instead, its dynamic counterpart is *reconfigurable duplication* (also called *standby redundancy* [48]). This technique consists of still having $(N - 1)$ replicas that, however, remain on standby and perform full computation only if a fault is detected. We can further classify the standby redundancy techniques in the following.

- *Hot-Standby*: Both the main task and replicas compute the output, but only the main task actually carries out the result, whereas the replicas' outputs are suppressed in normal operation. If a fault is detected, the main task output is suppressed and one of the replicas becomes the main task.
- *Cold-Standby*: The task replicas do not perform any calculation; they just keep the internal state updated. If a fault occurs, one cold-standby replica is elected as the new main task and starts to perform full operations. The delay in carrying out the result when a fault occurs is higher in this case compared with hot-standby, but it reduces the overhead in normal operation.

These two definitions may change depending on the context. Other terms are often used interchangeably, for example, *Warm-Standby* is sometimes used as a synonym for *Hot-Standby*.

**Time redundancy techniques**. Many different approaches have been proposed as time redundancy fault tolerance techniques. The most common ones are *re-execution*, *checkpoint/restart*, *recovery blocks*, and *forward error recovery*. The underlying concept of all of them is to execute a recovery routine when a fault is detected. In particular:

- The *re-execution* (or *retry*) technique restarts a job (potentially multiple times) if a fault is detected. The extra computation is identical to the original job. If the state of the job or the environment needs to be restored after a fault, an additional procedure, called *Backward Error Recovery*, is performed. In some previous works (e.g., [43, 78]), due to scheduling reasons, the currently preempted jobs are also re-executed even if not affected by the fault. This last technique is called *multiple recovery*.
- The *checkpoint/restart* (or *checkpoint/restore*) is an evolution of the re-execution technique, which periodically saves the state of the tasks (checkpoint). Fault detection triggers the restart from the last available checkpoint (restart). In contrast to the re-execution technique, the computation does not have to restart from scratch, reducing the response time in the case of a fault. However, it introduces an additional overhead for the checkpoint, even in the absence of faults. This strategy is typical of High-Performance Computing (HPC), in which tasks take hours or days to complete and restarting from scratch is not affordable.
- *Recovery blocks* can be considered as another variation of re-execution. In the case of a fault, a different version of the code is run to perform the same function (possibly with degraded performance). This technique also allows solving common-mode errors, for example, unexpected software bugs.
- *Forward error recovery* consists of executing a special routine that fixes the error without re-executing the original task. It is usually implemented with an exception-handling mechanism: an exception is raised/thrown, breaking the execution flow, and dedicated handling code manages the erroneous condition. This is typical of storage and network algorithms that use error-correcting code to fix the error. However, the use of this technique is usually limited to a few application scenarios.

### 2.4 Fault Detection and Isolation

The purposes of fault detection and fault isolation are to determine the presence of a fault (or an error) in the system and to identify the location and characteristics of the fault, respectively. Together with the previously mentioned fault recovery techniques, they are bundled under the umbrella term *Fault Detection, Isolation, and Recovery*. Many surveys on state-of-the-art hardware/software strategies are available [47, 92, 99].

A robust fault detection algorithm is essential for certain fault recovery techniques such as re-execution. In fact, while NMR is intrinsically able to detect and isolate the faults using the voting

mechanism, time redundancy techniques need proper methods that do not require re-executing the whole task to detect the faulty condition. Such methods often detect errors instead of faults. Hardware solutions for fault detection include (1) *fail-signal processors* [77] that can detect (usually, with a co-processor) abnormal functional or timing behaviors; (2) error-detecting code, mainly used in memories, bus, and communication devices; (3) watchdogs, which detect timing violations of the software; and (4) hardware acceptance tests. For software techniques, dedicated fault detection routines could run in parallel, at predefined intervals, or at the end of each job. They usually look for signature verification (e.g., control-flow graph signatures [62, 76]), run software watchdogs, or perform acceptance tests (also called *sanity checks* [90]). The acceptance tests can be performed in many different ways that strongly depend on the application itself. Good examples come from control theory, in which several different acceptance tests have been developed to verify the output of controllers (see Hwang et al. [47] for a comprehensive survey).

Hardware fault-detection techniques usually have no impact on the execution time — being executed in parallel or transparently to the software — or included in the architectural timing specification. Instead, for software fault detection, algorithm overhead should be taken into account in the design phase. The most common case is to execute the software acceptance test at the end of each job. From a modeling standpoint, the overhead of the fault-detection algorithm is usually included in the WCET of each task [16, 81, 101]. Instead, in the case of algorithms running in parallel, for example, in a multithreading setup [102], we need to consider the overhead separately. However, to the best of our knowledge, no works tried to analyze the timeliness of such parallel software fault-detection approaches.

### 2.5 Fault Injection

In both industrial and academic fields, testing the system against faults is essential to verify its resilience to possible errors. To effectively test detection and recovery capabilities, we cannot wait until a sufficient number of transient faults occurs; thus, artificial faults have to be injected into the system. Fault injection techniques can be categorized [45, 115] as follows.

- *Hardware-Based*: Using hardware devices to inject faults, including with contact (e.g., connect a probe to cause voltage spikes) or without contact (e.g., RF disturbances).
- *Software-Based*: Software that runs "in the background" and writes on memory locations according to a random or predefined pattern.
- *Simulation-Based*: The hardware platform is simulated, and the fault injection is introduced at Register-Transfer Level (e.g., in VHDL). The faults follow specific patterns and distributions determined at design time and cannot change at runtime.
- *Emulation-Based*: The hardware platform is written on an FPGA in addition to some extra blocks to allow a host computer to inject fault in a transparent way with respect to the processor under analysis.

All of these options have both advantages and disadvantages. Hardware-based fault injections are perfect representatives of the system and the faults. However, such mechanisms require complex devices, especially if one is trying to simulate transient faults. For example, nuclear devices are used to emit ion radiations and test the hardware resilience to faults [55], but this approach is very expensive and not practical for software development purposes. Simulating a fault in a particular area of the system (e.g., one particular bit of the memory) is also not easy with hardware techniques. Conversely, software-based solutions are very easy to be implemented, but from a real-time perspective, they break the timing property of the system, making an analysis of their overhead necessary. They may also not be able to stimulate certain system parts that are not directly accessible to software. Consequently, achieving representativeness is challenging [73]. Simulation-based

solutions are not intrusive and have maximum fault controllability. However, they are tricky to develop, and a considerable simulation time is needed to gather enough data. Emulation-based approaches are usually relatively easy to implement, but they have the same issues as software-based techniques, including the timing properties breaking.

In conclusion, there exist many different techniques to inject faults, each with its own advantages and disadvantages. Which technique to choose depends on the context, the necessary test accuracy, and the availability of test equipment.

### 2.6 Mixed-Criticality Systems

In critical systems, not all functions require the same level of guarantees in terms of timing or functional correctness. *Criticality* is a concept present in many safety-critical standards (e.g., DO-178B [89] and IEC 61508 [49]) and is usually assigned to each function provided by the system. It represents the mapping of the failure of a component (either software or hardware) with the impact on the whole system and environment safety. For instance, the aviation standard DO-178B has five criticality levels, named Design Assurance Levels (DALs). The highest DAL, DAL A, is assigned to the functions whose failure effects can cause catastrophic damage (loss of the airplane and human lives), while the lowest, DAL E, is the criticality level assigned to functions with no impact on flight safety in the case of malfunction.

The necessity to reduce design and production costs of systems, also in terms of non-functional properties (such as energy, power, weight, etc.), spurred the trend to consolidate multiple applications into the same system, possibly with different criticality levels. Such systems are called *mixed-criticality systems*. Regarding the software and timing domains, a criticality level is assigned to each software task. In the traditional Vestal's model for mixed-criticality [100], a set of different WCETs is assigned to the tasks, each one computed with a given confidence level depending on the criticality. The survey of Burns and Davis [17] recaps the recent developments in mixed-criticality by the real-time community.

The software failure model of a task is usually considered in the logical sense – the task produces an incorrect output – or in the timing sense – the task does not produce a correct output by the prescribed time. In dual-criticality systems, the `HI`-criticality tasks are allowed to "fail" in the timing sense by overrunning their `LO`-criticality WCET due to an underestimation of it. How to apply fault tolerance and graceful degradation, making the system robust and resilient, has been the subject of previous works (e.g., see [19]). This concept of tolerating temporal faults is orthogonal to the topics in this article. Instead, we focus on hardware faults and SIFT. In Section 4.2, we describe in detail the failure model for software tasks considered in this work.

## 3 LITERATURE REVIEW

This section presents the state-of-the-art of the scientific literature dealing with fault tolerance mechanisms applied to real-time systems. It is not the goal of this section to provide an extensive review of the fault-tolerance techniques, which is already present in other surveys available in the literature, for example, [40, 91]. A survey published in 2018 [66] reviewed the state-of-the-art works on joint fault and timing analyses, focusing on multicores, WCET estimation in the presence of hardware faults, and on fault injection mechanisms. This survey by Löfwenmark and Nadjm-Tehrani [66] can be considered orthogonal to this section, which instead focuses on the timing properties of fault-tolerance techniques.

The classification of the papers according to the used fault tolerance mechanism(s) is summarized in the Venn diagram of Figure 2. In the next subsections, we describe and analyze these papers.

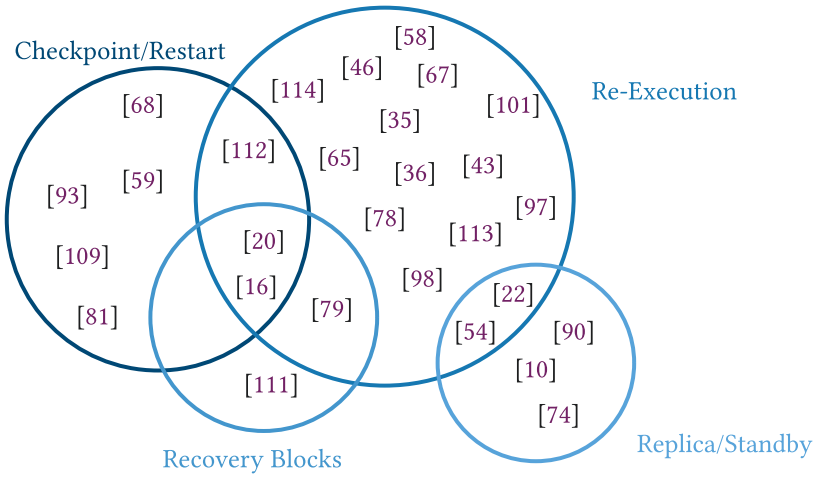


Fig. 2. The state-of-the-art papers classified based on the used fault tolerance technique.

### 3.1 Re-execution

The re-execution strategy is the most studied in the literature as the software-level fault tolerance mechanism for real-time systems. To the best of our knowledge, Pandya and Malek in 1994 [78][3] is the first work on real-time schedulability analysis that considers fault tolerance aspects, specifically the re-execution of the tasks in the case of transient faults. The authors computed the utilization upper bound that guarantees the schedulability of tasks under the *Rate Monotonic (RM)* scheduler ($U < 0.5$), assuming that one fault can occur and tasks restarted. In particular, the fault must occur with a frequency not greater than the inverse of the maximum task period, that is, $T_F \geq \max_{\tau_i} T_i$, where $T_F$ is the minimum interarrival time for the faults and $T_i$ the period of the $\tau_i$ task. Their approach is *multiple recovery*; it restarts all of the partially completed tasks when the fault occurs, including the preempted ones. This has been done in order to guarantee that the RM priority remains fixed. Ghosh et al. in 1995 [35] extended this work in 1998 [36]. They proposed the utilization bounds for the re-execution strategy, improving the previous bounds of Pandya and Malek [78]. The main change is to restart just the failed task and not the whole set of active tasks. They also briefly studied the case of multiple failures of the same task or of different tasks.

The concept of interarrival time for the faults was also considered by Burns and Davis [16] in 1996, in which the authors proposed a schedulability analysis based on the response time analysis for fixed-priority schedulers as a function of $T_F$. The authors computed the schedulability conditions for re-execution, forward error recovery, recovery blocks, and checkpoint/restart techniques. In 1999, this work was extended [20] by providing probabilistic bounds and the exact formulation for the minimum interarrival time of faults when they are distributed according to the Homogeneous Poisson Process assumption (described later in Section 4.3).

In 2009, Zhu and Aydin [114] studied the effect of dynamic voltage and frequency scaling (DVFS) on reliability. While reducing frequency and voltage has beneficial effects regarding permanent faults, it also increases the probability of transient faults, which has been analyzed by the authors. Their scheduling algorithm also reclaims the slack space to run recovery tasks. A 2022 survey on

[3]The paper had been published in the IEEE journal in 1998, but the original technical report has been available since 1994. This is also verifiable in the paper of Burns and Davis [16] published in 1996.

energy-aware scheduling algorithms [110, Section 5] describes some works that considered both the scheduling and reliability problems when transient faults are caused by DVFS settings.

The re-execution technique was also studied by Many and Doose [67] in the case of intermittent fault bursts, focusing more on disturbance duration rather than fault frequency. Their approach works for RM or *Deadline Monotonic (DM)* scheduling. They provided the schedulability analysis as a function of burst duration and max processor utilization. Similarly, in 2014, Haque et al. [43] proposed the utilization bound per Earliest Deadline First (EDF) with re-execution as a function of the burst duration. Thekkilakattil et al. [98] suggest a fault-tolerant scheme such that the schedulability is guaranteed if no more than one fault burst per hyperperiod occurs. If more than one fault burst occurs, the processor frequency is increased to provide the best possible computational power to schedule the re-execution and the remaining jobs.

In 2022, Kritikakou [58] studied the timing impact of re-execution in heterogeneous computing when applied to GPUs subject to SEUs.

### 3.2 Mixed-Criticality

The first two works on mixed criticality and fault tolerance were proposed in 2014 by Pathan [79] and Huang et al. [46]. Both converted the fault tolerance problem to a dual-criticality problem. The former exploits the re-execution mechanism and the recovery routines for HI-criticality tasks, whereas the LO-criticality tasks are dropped. The latter work used the re-execution mechanism and proposed a degradation of the performance of LO-criticality tasks instead of task dropping. The authors also derived a simple failure analysis to estimate the probability of a LO-criticality task to be dropped or degraded, linking it with the requirement of the probability of failure per hour. As a follow-up of the previous work, Lin et al. [65] proposed an EDF scheduling scheme based on the re-execution of faulty HI-criticality tasks and the maximization of the LO-criticality task executions. Then, an online slack reclaiming algorithm to accommodate recovery tasks for LO-criticality tasks is discussed. Re-execution in a mixed-criticality context was also considered by von der Brüggen et al. [101], which modeled a system with *normal* and *abnormal* modes. In normal mode, all of the tasks meet the deadlines. When a fault occurs, the abnormal mode is activated, and the deadlines are guaranteed for HI-criticality tasks only, whereas LO-criticality tasks have bounded worst-case tardiness. The authors also considered intermittent faults.

In 2017, Zhou et al. [113] proposed another mixed-criticality approach in which the HI-criticality tasks have redundancy similar to a replica but with a time redundancy. To tolerate 1 fault, they run the HI-criticality task 3 times, and a voting mechanism decides the output. To tolerate 2 faults, 5 identical jobs are run, and so on. If needed, the LO-criticality jobs are dropped. Thekkilakattil et al. [97] proposed in 2015 an extension of their previous work on re-execution tasks [98] which guarantees the fault tolerance requirements for the HI-criticality task only, whereas for LO-criticality the fault tolerance execution is not guaranteed after the occurrence of a fault. A different approach based on the application of re-execution and/or replica to task graph-based application has been proposed by Kang et al. [54] in 2014. They analyzed the static schedulability problem in mixed-criticality and multiprocessor context. More recently, in 2019, Safari et al. [90] studied the scheduling of replica tasks in a dual-criticality and multicore setup by also considering energy optimization. In 2022, Chen et al. [23] studied the link between resource sharing and transient faults in a mixed-criticality context. The authors developed a fault-tolerant scheme in which the normal sections of the tasks are protected by re-execution, whereas the critical sections are protected by replicas executed on a multicore processor.

Two mixed-criticality approaches were presented in 2011 and 2013 by Axer et al. [6] and Bolchini Miele [14]. Both papers considered a simplified mixed-criticality model in which the importance of each task affects only the fault tolerance and not the task timing properties, thus focusing more

on design-space exploration at the system level rather than on mixed-criticality schedulability analysis.

Finally, it is worth citing Gonzalez [38], who, in 1997, years before the seminal Vestal [100] paper on mixed-criticality, proposed an adaptive fault tolerance technique that degrades the execution of *noncritical tasks* when faults occur.

### 3.3 Checkpoint/Restart

The Checkpoint/Restart (C/R) strategy is a common approach for HPC systems, but it has also been studied for real-time systems. The first works on timing properties for C/R can be found in the 1970s [34, 108] and 1980s [95]. However, the first works considering hard real-time requirements are the papers on the response time analysis by Burns et al. [16, 20] in 1996 and 1999. The subsequent works focused on studying mainly the checkpoint rate, including the follow-up of the works by Burns et al. by Punnekkat et al. [81]. The same year, in 2001, Kwak et al. [59] studied the optimality of checkpoint periods using a Markov chain to model the Poisson distributed faults. The work uses a control theory–like approach considering the multiple tasks case only with timeline scheduling. In 2014, Zhengyong et al. [112] studied the optimality of the checkpoint rate when considering intermittent burst faults and compared C/R with the re-execution strategy.

The first paper on real-time and fault tolerance with the goal of reducing the energy consumption of a C/R approach was published in 2004 by Melhem et al. [68]. They assumed one single transient fault per job. They followed two approaches: (1) periodic checkpoints; and (2) checkpoint intervals have different sizes, increasing the frequency when approaching the deadline. They find the optimal period length for the first case; they computed the reduction in the processor's speed for the second case. The reference scheduler is EDF. Two years later, in 2006, Zhang and Chakrabarty [109] proposed another approach for power and energy management based on the C/R mechanism. Their work focused on hyperperiod-oriented feasibility tests, that is, guaranteeing that $k$ faults occurring in one hyperperiod are tolerable for a fixed-priority scheduler.

In 2016, Salehi et al. [93] proposed a tuning of the checkpoints frequency and processor frequency based on both offline and online decisions. The nonuniformity of checkpoint intervals (that are postponed as much as possible while guaranteeing $k$-fault tolerance) allows the energy to be reduced. The authors focused more on the energy and fault aspects than timing properties.

In HPC, C/R has also been used to implement predictive reliability. A health monitor constantly checks the system status, estimating its reliability to proactively report an imminent fault. This strategy usually applies to permanent faults and allows the resource manager to migrate a task to a different core or even a different system [85, 103].

### 3.4 Other Approaches

In 1997, Mosse et al. [72] studied standby-sparing systems on single- and multiprocessor scenarios, providing a theoretical model for certain classes of applications and standby-sparing strategies.

The extensive work in 2013 of Zhao et al. [111] studied the case of frame-based tasks with precedence constraints, expressed as a Direct Acyclic Graph (DAG) running on a system with energy management strategies (DVFS) and a novel shared recovery scheme. This strategy is based on recovery blocks that share the same time slots, turning to active in the case of fault. In this way, it is not necessary to allocate time to each recovery block task (considering a single-fault scenario). A DAG was also used by Chen et al. [22] to model a mixed fault tolerance approach that includes both NMR and re-execution. The authors provided design strategies for real-time systems to select the best fault-tolerance approach according to scheduling requirements in multicore systems.

In 2018, a short paper by Niu et al. [74] proposed the development of a scheduler for standby-sparing systems (2NR hardware redundancy) with the aim of minimizing energy consumption.

In 2016, a mixed hardware/software approach was proposed by the authors of [2]. This paper exploited the Simplex architecture that provides a hardware *rescue* computing unit that monitors the *main* unit. When a fault is detected, the rescue unit reboots the main unit and provides simple control of the system outputs to guarantee a minimal set of functional requirements while the main unit is rebooting. This is an example of re-execution fault tolerance applied to the whole system instead of at the task level.

Many other approaches to fault tolerance have been proposed in the context of distributed real-time embedded systems. We decided to not include them in this literature review section because they are out of scope with respect to timing analysis. Some examples include the work of Feng and Lee [32] in event-based systems with the C/R mechanism and the work of Emberson and Bate [27] based on hot backup replicas.

### 3.5 SIFT and Industry Standards

Depending on the application domain, different standards apply for safety-critical systems. Some standards allow or suggest the use of software fault tolerance techniques, for example, the EN 50128 [31] for critical software in the railway industry explicitly cites all of the aforementioned fault recovery techniques. The ECSS-Q-HB-60-02A handbook [30] by the European Space Agency (ESA) describes various software replica approaches as possible SIFT mechanisms for COTS hardware. However, the standards do not generally allow use of the failure rate at the software level. ISO 26262 (Functional Safety in Road Vehicles) [50], DO-178B/C (Software Considerations in Airborne Systems and Equipment Certification) [88], and EN 50128 explicitly disallow the software to have a defined probability of failure. This limitation comes from the impossibility of estimating the probability of a software fault – intended as a bug or defect – to occur. However, if we implement SIFT by restricting the fault model according to Table 1, it is possible to compute exactly the fault and failure rate after SIFT because the probability of the original event (the hardware fault) is known. Reconciling the software and hardware fault/failure probability, as described in the following sections, allows better exploitation of the computing resources and formally guarantees the satisfaction of the failure rate requirements.

## 4 CROSS-MODELING A FAULT-TOLERANT REAL-TIME SYSTEM

Most of the state-of-the-art works presented in the previous section used a traditional real-time model. In this section, we propose a model extension based on resource assignment functions, which allows a more accurate representation of the fault probabilities for each task. Then, we present how to compute the probability that a job is affected by a fault and the consequent impact on the failure requirement. As shown in Section 4.3, the use of this joint time-fault model can improve the satisfaction of failure and schedulability requirements. Section 5 discusses possible future work that may exploit this model.

### 4.1 Task and System Model

The task model is defined like a traditional real-time task model: the task set is identified by $\bar{\tau} = \{\tau_1, \tau_2, \ldots, \tau_n\}$, and each task $\tau_i = (C_i, T_i, D_i, p_i^{\text{req}})$ is modeled with the WCET $C_i$, the period or minimum interarrival time $T_i$ of the jobs, and the deadline $D_i$ (assumed constrained: $D_i \leq T_i$). It is possible to consider further timing properties, but they are not necessary for the goals of this work, since we are not focusing on a particular scheduling algorithm. The value $p_i^{\text{req}}$ represents the safety requirement, that is, the maximum probability of a task failure to occur that we must guarantee. All time values are expressed in workload units, that is, as the number of clock cycles when the task is run on a unit-speed processor. The $p_i^{\text{req}}$ value is instead expressed as frequency

over time unit, as detailed in Section 4.3. Each activation of a task is called a *job* and is represented by $\tau_{i,j}$. Each job has an activation or release time $a_{i,j}$, a start time $s_{i,j}$, and a finishing time $f_{i,j}$.

The resources of the systems $\bar{R} = \{R_1, R_2, \ldots, R_n\}$ represent all of the hardware components that can be assigned to each task, for example, processors, memories, and I/O devices. Each task is assigned to at least one resource. The assignment is performed statically offline or dynamically online depending on the scheduling and resource management strategy employed. We express the association of a task to its resource with the *resource assignment function*:

$$a(\tau, R) = \begin{cases} 1 & \text{if } \tau \text{ assigned to } R \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

With the resource assignment function, it is already possible to express the failure requirement by considering the probability that a fault occurs in a job $\tau_{i,j}$ as the merged probability of a fault to occur in the resources assigned to the task, that is, $\forall R : a(\tau_i, R) = 1$. To improve the granularity of the fault computation, we create more detailed representations of the resource usage by a task:

- The *active space-share assignment function* $0 \leq a_S^A(\tau, R) \leq 1$ and the *inactive space-share assignment function* $0 \leq a_S^I(\tau, R) \leq 1$. They are a generalization of the previous definition of resource assignment function representing the amount (in percentage) of the resource $R$ that is used by the task $\tau$ with respect to the fault probability. The active space-share assignment refers to when one job of the task started its execution but not yet finished (interval $[s_{i,j}, f_{i,j}]$). In this case, any fault occurring during the job execution can directly cause an error. The inactive refers to the time after the finishing time and before the next start time $[f_{i,j}, s_{i,j+1}]$. In this second case, any fault occurring before the next job execution may cause an error during the next job execution. This distinction is typical of memory resources and allows a better fine-grained estimation of the fault probability, reducing total job fault probability by excluding unused memory regions from the calculus. For example, let us consider a memory modelled with a uniform fault rate among the memory cells. Thus, if the task $\tau_1$ is assigned to memory $R_1$ of size 1024 MB, then $a_S^A(\tau_1, R_1) = 0.25$ and $a_S^I(\tau_1, R_1) = 0.125$ mean that the task is using at most 256 MB of $R_1$ when active and 128 MB of $R_1$ when inactive. For CPUs, this function typically assumes the Boolean values 0/1, that is, a task can be assigned or not to the CPU, but not half-assigned.
- The *exposure time share* $0 < a_T(\tau, R) \leq 1$, which is the maximum percentage of the task to be active in the resource $R$ over its period. If the resource active period corresponds to the task active period, then $a_T(\tau_i, R_j) = (\max_k(f_{i,k} - s_{i,k}))/T_i$. Note that $a_T(\tau_i, R_j) \leq 1$; otherwise, the scheduling would not be feasible. The difference $T_i - a_T(\tau_i, R_j)$ is the inactive time according to the definition of active and inactive space-share assignment.

*Example 1.* Let us consider a single-core single-memory system composed of two tasks $\tau_1$ and $\tau_2$ with $D_1 = T_1 = 50$, $D_2 = T_2 = 100$, and $C_1 = 15, C_2 = 45$ (the choice of harmonic periods and implicit deadlines has been made to simplify the example, but they are not limitations of the model). By using a preemptive RM scheduler,[4] we obtain the schedule depicted in Figure 3, which is identically repeated in each hyperperiod. Let us also assume that the memory size is 128 MB and both tasks use 1 MB of the memory when inactive and 16 MB when active. With these assumptions, the tasks will suffer from an error in their next job execution if a fault occurs during their inactive

[4]The task set is schedulable because the utilization is $U = \frac{15}{50} + \frac{45}{100} = 0.75$, which is compliant with the 2-task RM schedulability condition: $0.75 \leq 0.8284$.

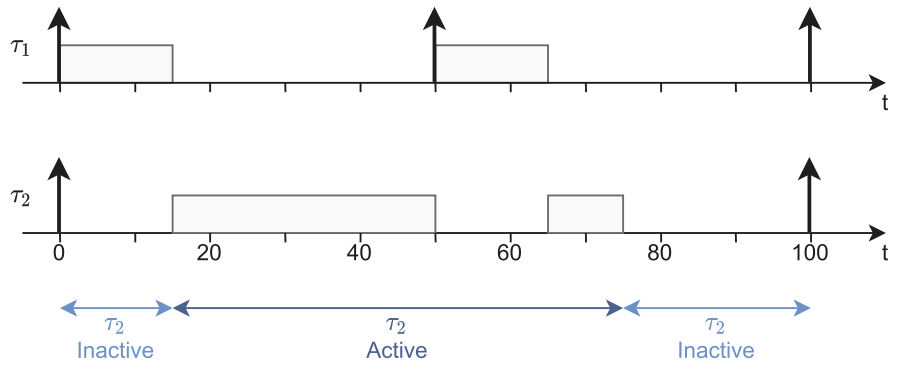


Fig. 3. An example of two tasks scheduled with preemptive RM. The active and inactive periods of $\tau_2$ are annotated.

period inside the 1 MB of memory or during the active period inside the 16 MB of memory. We can then compute the following values for the assignment functions:

- $a_S^A(\tau_1, MEM) = a_S^A(\tau_2, MEM) = 0.125$
- $a_S^I(\tau_1, MEM) = a_S^I(\tau_2, MEM) \approx 0.007$
- $a_S^A(\tau_1, CPU) = a_S^A(\tau_2, CPU) = 1$
- $a_S^I(\tau_1, CPU) = a_S^I(\tau_2, CPU) = 0$
- $a_T(\tau_1, CPU) = a_T(\tau_1, MEM) = 0.3$
- $a_T(\tau_2, CPU) = a_T(\tau_2, MEM) = 0.6$

### 4.2 Failure Models

In the this subsection, we propose a failure model and analysis to show how the previous task model – particularly the assignment functions – is helpful to improve the ability to comply with the requirements. As with the previous task model, the failure model will allow us in Section 5 to propose novel research directions. We limit the discussion of this section to transient and permanent hardware faults by assuming them independent and identically distributed. This assumption is realistic and widely adopted in industrial contexts [82].

**Hardware fault rate.** We split the hardware fault rate into transient fault rate $\lambda_T(R_i)$ and permanent fault rate $\lambda_P(R_i, t)$. In the latter case, the fault probability of a resource $R_i$ depends on the time, while the former is independent. The two fault rates are independent among them; thus, the total fault rate can be expressed as $\lambda(R_i, t) = \lambda_P(R_i, t) + \lambda_T(R_i)$. The traditional model for permanent faults is the bathtub curve [1]: hardware components have, in general, higher failure rates when they are young or old because of infant mortality and aging effects, respectively. Conversely, they exhibit a nearly-constant failure rate in the middle of their life due to random failures. Many approaches in safety-critical systems considered only this part of the device lifetime by assuming extensive initial testing and the setting of an expiration date for the device [57]. Conversely, the failure rate of components related to transient faults does not change with time, and it is constant if the system parameters are kept constant (e.g., same voltage and frequency, controlled temperature, etc.). As stated at the beginning of this section, all of the faults are assumed to be identically distributed. If this assumption is not valid, that is, a resource having different sub-components with different fault rates, this model can still be used by considering the sub-components as separated resources.

**The dangerous failure model.** To build a failure model for our tasks, we first need to describe how fault probabilities are modeled in reliability engineering. The basic metric for requirements

is called *failure rate* $\lambda$, which is expressed as the number of failures per unit of time. When considering dangerous failure requirements, the system is considered non-repairable because its failure can cause immediate damage. In this case, the *Mean-Time To Failure (MTTF)* and the *Mean-Time Between Failure (MTBF)* are equal and defined as $\frac{1}{\lambda}$. In 2015, the standard ISO 13849-1[75] defined the $\text{MTTF}_\text{D}$ as *Mean-Time To Dangerous Failure* to better identify the probability of a failure that has the potential to put the system in a hazardous state. In the traditional definition of hard real-time systems, a task can never miss its deadlines; otherwise, the system is considered failed. This matches with the concept of dangerous failures. In what follows, we assume that a hard real-time system performs a critical safety function that can potentially create a dangerous situation if the output is incorrect or not provided in time. This assumption has a crucial impact on what we have to compute for the failure analysis: we are not interested in how much frequent subsequent job failures (logical or timing) occur, but we need to compute the first failure probability, i.e., the $\text{MTTF}_\text{D}$. For example, let us consider these two scenarios:

- The job $\tau_{i,j}$ of a task overruns the deadline or outputs a wrong result, and all subsequent jobs $\tau_{i,j+1}, \tau_{i,j+2}, \ldots$ are correct.
- The job $\tau_{i,j}$ of a task overruns the deadline or outputs a wrong result, and all subsequent jobs $\tau_{i,j+1}, \tau_{i,j+2}, \ldots$ are dropped and never executed.

These two situations are equivalent when analyzing dangerous failures because even a single failure of a job of the task may cause unacceptable consequences in our safety-critical system.

**The (k,n)-failure model.** In contrast to dangerous failures, the (k,n)-failure model allows a limited number of jobs to fail in a given interval, that is, the system is considered to be working if at least k-out-of-n jobs correctly execute. This failure definition, from a real-time standpoint, is related to the umbrella term *weakly hard real-time systems* [9]. The next subsection focuses on the dangerous failure model and not on the (k,n)-failure model. However, (k,n) still presents interesting research challenges; we will resume the discussion of this model in Section 5.7.

### 4.3 Verifying the Task Failure Requirements

The statistical process representing the occurrence of faults is the *Bernoulli process*, which models a sequence of Boolean *random variables* $E_1, E_2, \ldots,$. These random variables assume the following values: $E_k = 1$ if a fault occurs at time[5] $t = k$, $E_k = 0$ otherwise. We also define the random variable $E(t) = \sum_i^t E_i$ as the number of faults occurring in a time interval $[0; t]$. In the case of dangerous faults, we are looking to the probability $P(E(t) > 0)$. Since $E(t)$ can only be a positive or null value, it can be written as

$$P(E(t) > 0) = 1 - P(E(t) = 0) = 1 - \prod_t (1 - p) = 1 - (1 - p)^n, \tag{2}$$

where $p$ is the probability of observing a fault in a single time unit. This formula is valid because the fault probabilities are independent for transient and permanent faults as discussed in Section 4.2. The following shows how to derive the probability of a job failure from the hardware failure rate $\lambda(R_i)$ in order to verify whether the task complies or not with the failure requirements. For the sake of completeness, it should be noted that for large $t$ and small $p$, this probability converges to the Poisson distribution with $k = 0$, that is, $P(E_i > 0) \to 1 - e^{-nt}$. This is frequently used in safety analyses [82] and is called the *Homogeneous Poisson Process (HPP)* hypothesis.

**Probability of a fault to occur in a given job.** The hardware fault rate is often expressed as the probability of observing a fault in a given time interval, usually *per hour*. For this reason, we write

[5]We consider the time as a discrete variable because, in computing, at the finest scale it corresponds to clock cycles, which is a discrete quantity.

$\mathfrak{F}$ to represent how many time units are in the time interval used as reference for the hardware fault rate $\lambda(R_i)$. For example, if $\lambda(R_i)$ is expressed *per hour* and the unit-speed processor has a frequency of 1 MHz, then $\mathfrak{F} = 3.6 \cdot 10^9$. Having this number, we compute the fault probability at any time unit by inverting Equation (2):

$$\lambda'(R_i) = 1 - [1 - \lambda(R_i)]^{\frac{1}{\mathfrak{F}}}. \tag{3}$$

We then derive the probability that a fault affects the job execution of a given task. This probability depends on which resources it uses, the space-share value and time-share value, and the execution time. The probability that a fault affects a task $\tau_i$ during the active period is

$$p_i^A = 1 - \left[ \prod_{\forall k : R_k \in \mathcal{R}} \left(1 - a_S^A(\tau_i, R_k)\lambda'_k\right) \right]^{a_T(\tau,R)T_i} \tag{4}$$

and during the inactive period is

$$p_i^I = 1 - \left[ \prod_{\forall k : R_k \in \mathcal{R}} \left(1 - a_S^I(\tau_i, R_k)\lambda'_k\right) \right]^{[1-a_T(\tau,R)]T_i}. \tag{5}$$

Finally, we derive the probability of task failure as $p_i^{A+I} = 1 - (1 - p_i^A)(1 - p_i^I)$ and use it to verify the requirement $p_i^{\mathsf{req}}$ by changing the time scale from job to the measurement unit that is used by $p_i^{\mathsf{req}}$, in expressed, in a similar way we performed in Equation (3).

*Example 2.* Consider the tasks of Example 1 and their assignment functions. The probability of the CPU to experience a fault is $\lambda(\text{CPU}) = 10^{-6}/h$ and the probability of the memory to suffer from an SEU in a memory cell is $\lambda(\text{MEM}) = 10^{-4}/h$. We then compute the respective probability at the single time unit, in this case by considering the unit-speed processor frequency at 1 MHz. Then, $\lambda'(\text{CPU}) = 3.3 \cdot 10^{-16}$ and $\lambda'(\text{MEM}) = 2.7 \cdot 10^{-14}$, respectively. Having computed both the assignment functions and the probability of hardware fault, we can now compute the probability that a job fails in the absence of fault-tolerance mechanisms:

- $p_1^A = 5.7 \cdot 10^{-14}$ and $p_1^I = 7.7 \cdot 10^{-15}$
- $p_2^A = 2.3 \cdot 10^{-13}$ and $p_2^I = 8.8 \cdot 10^{-15}$

from which we derive $p_1^{A+I} = 6.4 \cdot 10^{-14}$ and $p_2^{A+I} = 2.4 \cdot 10^{-13}$. Since there are, respectively, 72 000 000 and 36 000 000 jobs each hour, the probability that at least one job fails per hour is $p_1^{\mathsf{hour}} = 4.6 \cdot 10^{-6}/h$ and $p_2^{\mathsf{hour}} = 8.5 \cdot 10^{-6}/h$. As a term of comparison, without using the space assignment function, this value would be $\approx 10^{-4}$ for both tasks, and by considering the space assignment function but not the time assignment function, the value would be $\approx 1.3 \cdot 10^{-5}/h$. □

This example showed how, in the proposed model, a tight estimation of the failure probability reduces the time and space utilization of the resources and, in turn, improves satisfaction of the scheduling and failure requirements.

## 5 OPEN CHALLENGES AND FUTURE RESEARCH DIRECTIONS

In this penultimate section before conclusions, we enumerate the key challenges that arise from the literature review and proposed model.

### 5.1 How Scheduling Decisions Impact on Fault Tolerance

As a trivial example to show why scheduling decisions can impact failure requirements, consider the same example of Figure 3. According to Example 1, the two tasks are scheduled with a preemptive RM scheduler. However, if we consider a non-preemptive version for RM, task $\tau_2$ is not

preempted at $t = 50$ and runs until $t = 60$. The active time for task $\tau_2$ is then considerably reduced, while the $\tau_1$ active period remains equivalent and it can finish its execution by the deadline. By recomputing the probability following the same procedure of Example 2, we obtain $6.5 \cdot 10^{-6}/h$, which is $\approx 24\%$ lower than the previous failure rate. An RM scheduler gives high priority to shorter tasks (and, consequently, the lowest exposed period), but they are not necessarily the most critical tasks, that is, with the lowest value of $p_i^{\text{req}}$. In general, we can state that reducing the preemptability of the task set reduces the active time of the tasks (and, thus, their failure rate), but it also reduces the schedulability of the task set.

The initial part of the inactive time of a job is the time interval between the activation and the start of the task. In the example of Figure 3, this value is 20 for the task $\tau_1$. Depending on the application, the data input of the task can be read by the task itself or they may be already stored somewhere in memory. In this case, the input data are also subject to faults. If the data size is sufficiently large, it may be convenient to further split the inactive period into separate periods (i.e., $[a_{i,j}; s_{i,j}]$ and $[f_{i,j}; a_{i,j+1}]$) and analyze them separately from the failure analysis: The inactive period after the job execution does not require taking into account the faults occurring in the input memory area. Novel scheduling algorithms may be developed to reduce the $s_{i,j} - a_{i,j}$ time for the jobs with large input data quantities. To the best of our knowledge, no previous works address the problem of faults occurring in the input data memory region.

A challenge related to the input problem is data dependency among tasks. Many recent works in the real-time community use the DAG task model [11], in which each task (or sub-task) is functional dependent on the input/output of other tasks (or sub-tasks), in a so-called parent–child relation. The presence of such a model creates new challenges in failure analysis because a fault may cause a cascade fault in subsequent tasks. The data shared among the tasks are also affected during the time interval between the end of the parent and the start of the child. Similar to the earlier discussion, a scheduling algorithm that minimizes this time may improve compliance with the failure requirements.

*Research Question 1.* How can we build reliability-aware schedulers that minimize the active time and, consequently, the exposure time $a_T(\tau, R)$, thus reducing the task failure rate $p_i^{A+I}$ while maintaining good schedulability levels?

*Research Question 2.* If a task requires the reading of a large amount of input data, how do we implement a scheduler aware of this condition and that minimizes the delay between the activation time and the start time of such task(s)?

*Research Question 3.* The DAG task model presents several challenges for real-time schedulers, and the data dependency among the tasks creates a shared region of memory that can be affected by faults. How can we perform proper scheduling to minimize the exposure time of these memory areas?

### 5.2 Scheduling Analysis of Fault Tolerance Approaches

Fault tolerance techniques are usually analyzed separately in the state-of-the-art works, sometimes comparing one against the other. An evolution of this approach is to combine multiple techniques, for example, NMR and re-execution. The optimization of the integration of multiple techniques can be advantageous from the scheduling standpoint to improve schedulability and, at the same time, the compliance of failure requirements. The optimization of the checkpoint rate in the context of heterogeneous fault tolerance mechanisms is another possible research direction.

Sporadic tasks are, in general, more difficult to handle in real-time systems unless we consider them as periodic, allocating the same amount of time a periodic task would need. This problem

becomes even more critical when multiple fault tolerance mechanisms have to be employed for such tasks, because allocating the required time in each period may lead to waste a consistent amount of resources that would be underutilized when the sporadic task is not released. However, in the presence of multiple sporadic tasks, we can say, in general, that it is improbable that multiple tasks are both activated and affected by a fault at the same time. If the probability of activation is known or estimable, the time required for the fault tolerance routines can be shared among the sporadic tasks and the failure analysis correctly computed. Further investigations on how to deal with sporadic tasks, allocate their fault tolerance routines and evaluate different fault tolerance strategies are needed to better comprehend the effects on scheduling and failure analyses.

Conversely, the class of intermittent faults (described in Section 2.2) requires proper handling of multiple faults in a short time period (fault bursts). This is challenging for both fault tolerance and scheduling. A few works recently dealt with this class of faults that also lack well-assessed fault models. Ferreira et al. [33] showed that intermittent faults are the cause of 90% of the total faults of a CAN network when used in an automotive context. Linked to the fault burst problem is the handling of MEU faults occurring at the same time.

Permanent faults are not necessarily full-system failures: as described in Section 2.2, they can be categorized as fail-stop, fail-partial, and fail-slow. These failure modes present different challenges from the real-time standpoint: the total or partial unavailability of the resource or the resource slowdown reduces the computational capabilities in different ways, which should be managed to allows the critical tasks to run, possibly without the fault tolerance routines or by dropping non-critical tasks. This scheduling involves several of the following challenges. For instance, the challenges in using DVFS are the same as those of the fail-slow case (see Section 5.4) or the mixed-criticality approach to select the task to drop (see Section 5.3).

At the end of Section 5.1, we discussed how scheduling decisions impact the failure analysis of DAG tasks. When the computed failure probability of the tasks does not adhere to the requirements, the implementation of fault tolerance strategies becomes necessary. However, time redundancy fault tolerance algorithms, such as re-execution, make real-time scheduling more difficult because in the DAG task model, each task can delay all of the child tasks if a fault occurs and, for example, a re-execution is needed. On the other hand, the main advantage of DAG task models – that is, the possibility of parallelizing the workload – would be reduced if space-redundancy techniques are employed, such as NMR, which occupies the processor to run the fault-tolerance algorithms instead of running the parallel workload. However, the presence of a DAG task model can also be beneficial. It allows exploitation of the idle processors when subsequent tasks are blocked by dependencies to dynamically run preventive fault-tolerant executions and reduce the number of time-redundancy algorithms required. The DAG task model presents many challenges and opportunities for future works.

*Research Question 4.* Integrating different fault tolerance techniques may improve the satisfaction of failure requirements while providing better schedulability conditions. How do we allocate resources and perform scheduling in this heterogeneous context?

*Research Question 5.* Sporadic tasks are difficult to handle in real-time systems, and more challenges are present when fault-tolerance routines need to be scheduled. How do we efficiently schedule tasks without wasting resources when sporadic tasks are not activated? What are the consequences for failure analysis and schedulability?

*Research Question 6.* Which fault models, fault-tolerance strategies, and scheduling algorithms can be developed to react to intermittent faults and MEUs effectively?

*Research Question 7.* How do we reconfigure real-time scheduling to react to fail-stop, fail-partial, and fail-slow permanent faults with the goal of maximizing system utility?

*Research Question 8.* How do we efficiently schedule fault tolerance algorithms when DAG task models are employed? Is it possible to exploit the parallelism information available in the DAG to better allocate preventive fault tolerance jobs?

### 5.3 Mixed-Criticality and Fault Tolerance

As briefly introduced in Section 2.6, Vestal proposed the mixed-criticality model for real-time systems in his seminal 2007 paper [100], in which a criticality level is assigned to each task that conceptually corresponds to the criticality specified for the safety analysis. From the timing analysis perspective, a task is characterized by one or more WCET values estimated at different levels of assurance. In the event that a task overruns one of its WCET estimations, a *system mode switch* occurs. In this case, the system reconfigures the scheduling strategy to still guarantee the timing constraints of the high-criticality tasks. The traditional method consists of dropping the lower-criticality tasks. For example, let us assume to have two criticality levels (LO and HI). To each LO-criticality task, we assign a WCET $C_i^{\mathtt{LO}}$. For HI-criticality tasks, we assign a WCET $C_i^{\mathtt{LO}}$ and a WCET $C_i^{\mathtt{HI}}$, with $C_i^{\mathtt{LO}} < C_i^{\mathtt{HI}}$. The scheduler begins scheduling the tasks by considering the $C_i^{\mathtt{LO}}$ of all tasks. If an overrun occurs, the LO-criticality tasks are dropped to allow the HI-criticality tasks to be scheduled according to $C_i^{\mathtt{HI}}$. However, several discussions recently emerged on the task-dropping strategy [13, 29], which is considered far from the industry and standards viewpoint.

The reason why task-dropping does not fit with the safety standards is the common misconception of what *criticality* is [29]. If a task is LO-criticality, it does not mean that is also a non-important task that can be dropped when needed: a LO-criticality task is still a critical task, performing a critical function subject to certification requirements. Dropping tasks during mode switch creates a "dependency" between the tasks: the functional correctness of a task does not depend only on the task itself; rather, it is influenced by the behavior of other tasks. This is a violation of the independence property, which is instead required by safety-critical standards, such as IEC61508-3 [49]: *"It shall be demonstrated either (1) that independence is achieved by both in the spatial and temporal domains, or (2) that any violation of independence is controlled."* To solve this problem, we propose to "control" this violation by joining the scheduling and failure analyses in order to obtain quantitative data on the probability that task dropping occurs, which can be, in turn, used in the safety analysis. Recent works tried to overcome this issue by applying different strategies than job dropping: for example, precise scheduling the task by using DVFS to sacrifice energy in HI-criticality mode [12, 94, 106].

Instead of triggering the mode switch due to an optimistic estimation of the WCET (an event for which we do not know its probability), we propose to trigger it when faults occur and fault-tolerance mechanisms need to be scheduled. In this way, we obtain two benefits:

(1) The probability of the mode switch can be computed and used in the failure analysis to quantify the "violation of independence."
(2) The MC scheduling algorithms improve the schedulability ratio compared with traditional non-MC scheduling.

To better describe this concept, we consider the following motivational example:

*Example 3.* Let us consider the task set depicted in Table 2. Tasks $\tau_1$ and $\tau_2$ belong to the highest level, DAL A, $\tau_3$ to DAL B, and $\tau_4$ to DAL D. The maximum probabilities of failure associated to the tasks are derived from the DO-254 standard [89]. According to the failure probability, we can

Table 2. Example of a Task Set, where $T_i$ is the Period, $D_i$ is the Deadline, $C_i$ is the WCET, and $p_i^{\text{req}}$ is the Required Probability of Failure

| Task | $T_i = D_i$ | $C_i$ | Criticality | $p_i^{\text{req}}$ |
|---|---|---|---|---|
| $\tau_1$ | 50 | 10 | DAL A | $10^{-9}$ |
| $\tau_2$ | 1000 | 75 | DAL A | $10^{-9}$ |
| $\tau_3$ | 250 | 50 | DAL B | $10^{-7}$ |
| $\tau_4$ | 100 | 25 | DAL D | $10^{-3}$ |

consider three criticality levels: $\tau_1$ and $\tau_2$ are HI-criticality tasks, $\tau_3$ is an MI-criticality task, and $\tau_4$ is a LO-criticality task.

We could have estimated the WCET at different levels of assurance (not shown in Table 2), as we did in a traditional mixed-criticality setup. However, for instance, it is not possible to drop the MI-criticality task $\tau_3$ because of the overrun of the MI-criticality WCET of the HI-criticality task. The task $\tau_3$ still provides an essential and inalienable feature with a well-defined probability of failure. By allowing other tasks to drop its execution, we are violating the previously mentioned task-independence requirement. Our proposal instead maps the mode switch to a fault event. Let us consider that the job fault probability is $p = 10^{-4}/h$ for all of the tasks and that we use re-execution to improve the resilience to faults. In this case, task $\tau_4$ does not require re-execution because the job fault probability already complies with the failure requirements. Instead, task $\tau_3$ must be restarted one time in the case of a fault, whereas $\tau_1$ and $\tau_2$ requires tolerating two faults, that is, two re-executions, to reach the requirement $p = 10^{-9}/h$. Having these values, we can map the WCET as follows:

- $C_1^D = 10$, $C_1^B = 20$ ($1^{\text{st}}$ re-execution), $C_1^A = 30$ ($2^{\text{nd}}$ re-execution)
- $C_2^D = 75$, $C_2^B = 150$ ($1^{\text{st}}$ re-execution), $C_2^A = 225$ ($2^{\text{nd}}$ re-execution)
- $C_3^D = 50$, $C_3^B = 100$ ($1^{\text{st}}$ re-execution)
- $C_4^D = 25$

If a fault occurs in a job of $\tau_3$, the system mode switches, the job of $\tau_3$ is re-executed, and $\tau_4$ is dropped. However, in contrast to the usual mixed-criticality case, we know the exact probability that this occurs ($p = 10^{-4}/h$). Thus, we can control the effect of $\tau_3$ on the failure requirements of $\tau_4$. The same applies for faults occurring in $\tau_1$ and $\tau_2$.

The use of this MC model modified for re-execution adds new challenges for real-time scheduling but also provides the opportunity to exploit the traditional MC theory in a compliant way. Reghenzani et al. [83] showed, in 2022, that the use of MC scheduling algorithms with task sets having re-execution jobs can improve the schedulability of such task sets. At the same time, we can obtain an MC system compliant with safety-critical standards.

A 2021 work by Burns [15] outlined a similar model for fault tolerance in which the WCET is categorized as $C(NotRobust)$ and $C(Robust)$. Existent works on mixed-criticality can be re-adapted for such similar cases. For example, the work by Guo et al. [42] studied the dynamic priority scheduling of tasks with permitted failure probability. The task failure probability relates to the execution time distribution but can be remapped to the fault probability.

*Research Question 9.* How do we build a convincing set of arguments to introduce into industry standards the use of an exactly computed software failure probability in the context of SIFT?

*Research Question 10.* What are the challenges in using current academic approaches on mixed-criticality setups in industrial applications subject to safety-critical standards?

*Research Question 11.* When the mixed-criticality strategy of Section 5.3 is employed, we know the exact probability of a mode switch to occur. Can this information be exploited to perform safe probabilistic scheduling?

*Research Question 12.* How do we exploit joint scheduling and failure analysis to explore the trade-off between schedulability and compliance with failure requirements in the mixed-criticality context?

### 5.4 The Effect of Power Management Techniques

Dynamic Voltage and Frequency Scaling (DVFS) is the most common approach to optimizing the trade-off between performance and power/energy consumption. Frequency scaling directly impacts timing requirements; vast literature is available on this topic (see the survey of Bambagini et al. [8]). Reducing power consumption also has side effects, first and foremost thermal effects and, in turn, effects on reliability. Reducing the average temperature and thermal stress decreases the probability of permanent faults. However, frequency and voltage cannot be selected arbitrarily without considering their relation. Usually, the lower the frequency, the lower the voltage. This relation leads to the trade-off between permanent and transient faults: while reducing the frequency/voltage is beneficial for permanent faults, low voltages increase the probability of transient faults to occur [114]. It has been observed [26] that in cache memories the SEU probability increases by one order of magnitude when running in low power mode. The optimal DVFS operating point with respect to energy does not correspond to the optimal operating point with respect to transient fault probability [28]. The cross-links between permanent faults, transient faults, and real-time requirements have not been thoroughly studied.

Another interesting aspect for future research is the fact that the use of lower frequencies increases the execution time and, in turn, the exposure time of our tasks $a_T(\tau, R)$, as defined in our previous model of Section 4.1. Two opposing goals are then present: (1) the traditional use of DVFS for improving the reliability in terms of permanent faults; and (2) the minimization of exposure time to transient faults. To model such a trade-off, let us begin with stating the Arrhenius equation for reliability according to the standard JESD91A [51]:

$$A_T = \exp\left[-\frac{E_{aa}}{k}\left(\frac{1}{T_s} - \frac{1}{T_r}\right)\right] \tag{6}$$

$$\lambda_s = \lambda_r \cdot A_T, \tag{7}$$

where

- $\lambda_s$ is the system fault rate at temperature $T_s$ (in K).
- $\lambda_r$ is the reference system fault rate at the reference temperature $T_r$ (in K).
- $E_{aa}, k$ are constants, the apparent activation energy and the Boltzmann's constant, respectively.

During system runtime, the fault rate equation becomes a monotonically non-decreasing function of only the temperature, $\lambda_s = f(T_s)$, according to the Arrhenius equation. The steady-state temperature is a monotonically non-decreasing function of the power consumption, that is, in turn, a monotonically non-decreasing function of the voltage/frequency assigned to the processor. This explains how DVFS strategies have a direct impact on reliability concerning the fault rate for permanent faults. Decreasing the voltage/frequency decreases the fault rate; on the other hand, it increases the execution time of the task. This has a side impact on the reliability of the overall system when we also consider transient faults: decreasing the frequency has the effect of increasing the execution time and, then, the exposure time of our tasks, that is, the $a_T(\tau, R)$ value, increasing the probability of a transient fault to occur.

*Research Question 13.* How does energy-aware real-time scheduling impact the reliability problem with respect to both permanent and transient fault rates? Can mixed-criticality be exploited for this optimization problem?

*Research Question 14.* Which is the optimal DVFS strategy that, while guaranteeing the hard real-time constraints, minimizes the failure rate of the overall system, taking into account all factors (thermal, transient faults susceptibility, and exposure time) that affect reliability?

### 5.5 The Implementation of the OS and Scheduler

*Quis custodiet ipsos custodes?*[6] This historical Latin quote perfectly describes the problem of realistic implementations of software fault tolerance: How can we be sure that a fault does not happen in critical system software, such as the scheduler, the fault-detection mechanism, or, in general, any other operating system component? Most of the literature presented in Section 3 assumed the system software and fault tolerance routines to be flawless, neglecting this aspect. Dealing with system software faults has non-trivial theoretical and engineering challenges.

The first solution to address these challenges is to compute the probability that a fault happens in critical system software and add it to the failure analysis. Depending on the number of tasks, the complexity of fault tolerance/detection techniques, and the services offered by the operating system, this solution is potentially feasible or not. Suppose that the system software has a small memory footprint and executes fast. In that case, the values of $a_S^A(\tau, R)$ and $a_T(\tau, R)$ are usually very limited. Consequently, the probability that a transient fault affects a system software is small. Applying the previous model to system software allows the system integrator to compute the failure probability for the whole system, which is, hopefully, compliant with the failure requirement.

The second solution is the use of hardware-based voter or recovery strategies. For example, Yim et al. [107] implemented a programmable hardware voter that takes the decision on the NMR software tasks outputs. In such a way, the risk of a fault happening in the voting procedure is moved to the hardware, which is implemented to be fault tolerant. While the concept of applying this scheme to a general application is straightforward, how to propagate the result of OS routines back from the hardware to the software is non-trivial and still subject to faults. Future works can investigate how to implement an OS capable of exploiting this mixed software–hardware fault tolerance via hardware voters. However, the fact that these solutions require special-purpose processors or, in general, custom hardware should also be considered, cancelling out one of the major advantages of software fault tolerance techniques, that is, their possible use in COTS hardware.

The third solution is to implement SIFT mechanisms for system software. For example, in a multicore system, the scheduler could run with the NMR fault-tolerance strategies simultaneously on different cores. Then, the scheduling decision is compared and applied via a voting procedure. While the idea looks quite trivial, the actual implementation and the computation of failure probability are not trivial. To the best of our knowledge, no works have addressed software fault tolerance concerning system software.

*Research Question 15.* How do we develop system software to reduce its failure probability by improving $a_S^I(\tau, R)$, $a_S^A(\tau, R)$, and $a_T(\tau, R)$?

*Research Question 16.* How to exploit integrated software-hardware fault tolerance mechanisms (such as software NMR linked to a hardware voter) to improve system software reliability?

*Research Question 17.* How do we implement SIFT for system software?

[6]Who guards the guards?

### 5.6 Exploiting Probabilistic Information

The model to compute the failure rate presented in Section 4 uses the WCET to compute the maximum exposure time of a given job, that is, the function $a_T(\tau_i, R_j)$ is computed by using $C_i$. Therefore, the value we obtain for the failure rate $\lambda$ is a *worst-case* failure rate. This approach is not typical in reliability engineering, which instead tends to use the average value when a failure rate cannot be considered constant [53]. Conversely, in our approach, we assume that every job instance of a task is exposed for at least its WCET in addition to the preempted time. While this is a safe assumption, this is also very pessimistic because running until the WCET is a rare condition [64]. Moreover, the WCET in modern architectures is difficult to estimate statically, and the estimation is often overly pessimistic [61].

A possible solution to reducing such pessimism is to estimate an average value for the failure rate using probabilistic information. The most straightforward approach is to exploit the central limit theorem to compute the average execution time and its confidence interval. A similar solution is to use the measurement-based probabilistic-WCET (pWCET). The pWCET represents the distribution of large values of execution time and can considerably reduce the overestimation of the traditional WCET analysis techniques. However, while some recent works addressed the estimation uncertainty of the pWCET [84, 86], the safety of pWCET results is still controversial. This is mainly due to the well-known representativity problem of the observed execution time, which intrinsically contains epistemic uncertainty. The same problem also affects the central limit theorem and, in general, any statistical technique. However, epistemic uncertainty is also present in many estimation methods for reliability engineering because the fault models are never perfect. Its presence is, to some extent, accepted, and some theories have been developed to handle the epistemic uncertainty [52].

*Research Question 18.* Can we trust probabilistic information on the execution time for the sake of failure rate computation? If not, which hardware/software strategies and analyses can be employed to reduce the epistemic uncertainty and improve the representativity of the observed execution time?

### 5.7 Other Research Directions

The (k,n) failure model was briefly discussed at the end of Section 4.2. In such a case, the different failure requirement plays a key role in the schedulability analysis because dropping jobs becomes a permitted action (under the specified limits). In safety-engineering terms, the (k,n) failure model is linked to the *availability* concept rather than *reliability*. While many works exist on (k,n) schedulability, the failure analysis of Section 4.3 must be reshaped to allow job failures. Similar to the model considered in this article, the failure analysis may impact the scheduling decision and vice versa.

Similarly, approximate computing can be considered in this context. Once a fault occurs, the overhead introduced by time redundancy techniques may require the system to degrade the application quality-of-service, for instance, by reducing the result accuracy. The presence of approximate computing adds a new dimension to be explored for trade-offs, that is, the output accuracy is traded for real-time and/or fault-tolerant guarantees.

Finally, we would like to put a focus on security, in particular, on malicious faults. Attackers can act in several ways on the system: software or hardware, remote or local, with contact or without contact, system violation, or service degradation (Denial-of-Service [DoS]). However, even if we find a way to model these malicious faults, they are usually characterized by one unknown or human-dependent parameter: for example, the probability that an attack occurs or the probability that a vulnerability is present. It is difficult, if not impossible, to precisely estimate these quantities.

Even if some metrics exist and they are already used in the security world – for example, the CVSS score [69] – their application to safety-critical systems looks limited due to their expert-provided nature. This is a crucial difference compared with natural faults, which instead follow well-known or estimable distributions. Additionally, not all SIFT techniques may be the best solutions to tackle malicious faults: for instance, re-executing a workload may worsen a DoS scenario.

*Research Question 19.* How does the (k,n) failure model impact the failure analysis and the interactions between scheduling and fault tolerance?

*Research Question 20.* Approximate computing can be exploited to carry out less-precise results during/after a fault event. How do fault tolerance, real-time scheduling, and output precision interact in this context? Can schedulability be linked to the result precision and fault tolerance requirements?

*Research Question 21.* Is it possible to model the probability of malicious faults to occur such that hard real-time requirements are guaranteed and a sound failure analysis is possible?

## 6 CONCLUSIONS

Real-time scheduling, failure analysis, and software fault tolerance techniques have been thoroughly studied in the past few decades. However, most studies treat them separately, and their interactions still present numerous challenges. This article surveyed state-of-the-art scientific works analyzing the SIFT mechanisms and their real-time schedulability, including in the mixed-criticality context. We proposed a new joint model that integrates failure and timing requirements and highlighted a series of related open research questions.

## REFERENCES


[1] M. V. Aarset. 1987. How to identify a bathtub hazard rate. *IEEE Transactions on Reliability* R-36, 1 (1987), 106–108. https://doi.org/10.1109/TR.1987.5222310

[2] Fardin Abdi Taghi Abad, Renato Mancuso, Stanley Bak, Or Dantsker, and Marco Caccamo. 2016. Reset-based recovery for real-time cyber-physical systems with temporal safety constraints. *IEEE International Conference on Emerging Technologies and Factory Automation (ETFA'16)*. Berlin, 1–8. https://doi.org/10.1109/ETFA.2016.7733561

[3] Fatimah Adamu-Fika and Arshad Jhumka. 2015. An investigation of the impact of double bit-flip error variants on program execution. In *8th International Conference on Dependability, Lecture Notes in Computer Science*, Vol. 9531. Springer, Venice, Italy, 15–22. https://doi.org/10.1007/978-3-319-27140-8_55

[4] AE-4 Electromagnetic Compatibility (EMC) Committee. 2010. *Guide to Certification of Aircraft in a High-Intensity Radiated Field (HIRF) Environment.* Technical Report. SAE International. 130 pages. https://doi.org/10.4271/ARP5583A

[5] Algirdas Avižienis, Jean-Claude Laprie, and Brian Randell. 2004. Dependability and its threats: A taxonomy. In *Building the Information Society*, Renè Jacquart (Ed.). Springer US, Boston, MA, 91–120. https://doi.org/10.1007/978-1-4020-8157-6_13

[6] P. Axer, M. Sebastian, and R. Ernst. 2011. Reliability analysis for MPSoCs with mixed-critical, hard real-time constraints. In *2011 Proceedings of the 9th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'11)*. IEEE, Taipei, Taiwan, 149–158. https://doi.org/10.1145/2039370.2039396

[7] Fatemeh Ayatolahi, Behrooz Sangchoolie, Roger Johansson, and Johan Karlsson. 2013. A study of the impact of single bit-flip and double bit-flip errors on program execution. In *Computer Safety, Reliability, and Security*, Friedemann Bitsch, Jérémie Guiochet, and Mohamed Kaâniche (Eds.). Springer, Berlin, 265–276. https://doi.org/10.1007/978-3-642-40793-2_24

[8] Mario Bambagini, Mauro Marinoni, Hakan Aydin, and Giorgio Buttazzo. 2016. Energy-aware scheduling for real-time systems: A survey. *ACM Trans. Embed. Comput. Syst.* 15, 1, Article 7 (Jan. 2016), 34 pages. https://doi.org/10.1145/2808231

[9] G. Bernat, A. Burns, and A. Liamosi. 2001. Weakly hard real-time systems. *IEEE Trans. Comput.* 50, 4 (April 2001), 308–321. https://doi.org/10.1109/12.919277

[10] Anand Bhat, Soheil Samii, and Ragunathan Raj Rajkumar. 2017. Practical task allocation for software fault-tolerance and its implementation in embedded automotive systems. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'17)*. IEEE, Pittsburgh, PA, 87–97. https://doi.org/10.1109/RTAS.2017.33

[11] Ashikahmed Bhuiyan, Zhishan Guo, Abusayeed Saifullah, Nan Guan, and Haoyi Xiong. 2018. Energy-efficient real-time scheduling of DAG tasks. *ACM Trans. Embed. Comput. Syst.* 17, 5, Article 84 (Sept. 2018), 25 pages. https://doi.org/10.1145/3241049

[12] A. Bhuiyan, F. Reghenzani, W. Fornaciari, and Z. Guo. 2020. Optimizing energy in non-preemptive mixed-criticality scheduling by exploiting probabilistic information. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 11 (2020), 3906–3917. https://doi.org/10.1109/TCAD.2020.3012231

[13] Konstantinos Bletsas, Muhammad Ali Awan, Pedro Souto, Benny Åkesson, Alan Burns, and Eduardo Tovar. 2018. Decoupling criticality and importance in mixed-criticality scheduling. In *6th International Workshop on Mixed Criticality Systems (WMC'18)*. WMC, Nashville, TN, 25–30.

[14] Cristiana Bolchini and Antonio Miele. 2013. Reliability-driven system-level synthesis for mixed-critical embedded systems. *IEEE Trans. Comput.* 62, 12 (2013), 2489–2502. https://doi.org/10.1109/TC.2012.226

[15] Alan Burns. 2020. Multi-model systems — an MCS by any other name. In *Workshop on Mixed Criticality Systems (WMC'21)*. RTSS, Virtual, 4.

[16] Alan Burns and Robert Davis. 1996. Feasibility analysis of fault-tolerant real-time task sets. In *Proceedings of the 8th Euromicro Workshop on Real-Time Systems*. IEEE, L'Aquila, Italy, 29–33. https://doi.org/10.1109/EMWRTS.1996.557785

[17] A. Burns and R. I. Davis. 2017. A survey of research into mixed criticality systems. *ACM Computer Surveys* 50, 6 (2017), 1–37.

[18] Alan Burns and Robert Davis. 2019. Mixed Criticality Systems — A Review. (2019), 81 pages. https://www-users.cs.york.ac.uk/burns/review.pdf.

[19] Alan Burns, Robert I. Davis, Sanjoy Baruah, and Iain Bate. 2018. Robust mixed-criticality systems. *IEEE Trans. Comput.* 67, 10 (2018), 1478–1491. https://doi.org/10.1109/TC.2018.2831227

[20] A. Burns, S. Punnekkat, L. Strigini, and D. R. Wright. 1999. Probabilistic scheduling guarantees for fault-tolerant real-time systems. In *Dependable Computing for Critical Applications 7*. IEEE, San Jose, CA, 361–378. https://doi.org/10.1109/DCFTS.1999.814306

[21] Ramon Canal, Carles Hernandez, Rafa Tornero, Alessandro Cilardo, Giuseppe Massari, Federico Reghenzani, William Fornaciari, Marina Zapater, David Atienza, Ariel Oleksiak, Wojciech Piundefinedtek, and Jaume Abella. 2020. Predictive reliability and fault management in exascale systems: State of the art and perspectives. *ACM Comput. Surv.* 53, 5, Article 95 (Sept. 2020), 32 pages. https://doi.org/10.1145/3403956

[22] K. Chen, G. v. der Brüggen, and J. Chen. 2018. Reliability optimization on multi-core systems with multi-tasking and redundant multi-threading. *IEEE Trans. Comput.* 67, 4 (2018), 484–497. https://doi.org/10.1109/TC.2017.2769044

[23] Nan Chen, Shuai Zhao, Ian Gray, Alan Burns, Siyuan Ji, and Wanli Chang. 2022. MSRP-FT: Reliable resource sharing on multiprocessor mixed-criticality systems. In *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS'22)*. IEEE, Milano, Italy, 201–213. https://doi.org/10.1109/RTAS54340.2022.00024

[24] Cristian Constantinescu. 2007. Intermittent faults in VLSI circuits. *IEEE Workshop on Silicon Errors in Logic-System Effects*. 7183 (2007), 1–3.

[25] Cristian Constantinescu. 2008. Intermittent faults and effects on reliability of integrated circuits. In *2008 Annual Reliability and Maintainability Symposium*. IEEE, Las Vegas, NV, 5. https://doi.org/10.1109/RAMS.2008.4925824

[26] V. Degalahal, Lin Li, V. Narayanan, M. Kandemir, and M. J. Irwin. 2005. Soft errors issues in low-power caches. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 13, 10 (2005), 1157–1166. https://doi.org/10.1109/TVLSI.2005.859474

[27] Paul Emberson and Iain Bate. 2008. Extending a task allocation algorithm for graceful degradation of real-time distributed embedded systems. In *Proceedings - Real-Time Systems Symposium*. IEEE, Barcelona, Spain, 270–279. https://doi.org/10.1109/RTSS.2008.24

[28] D. Ernst, S. Das, S. Lee, D. Blaauw, T. Austin, T. Mudge, Nam Sung Kim, and K. Flautner. 2004. Razor: Circuit-level correction of timing errors for low-power operation. *IEEE Micro* 24, 6 (2004), 10–20. https://doi.org/10.1109/MM.2004.85

[29] Rolf Ernst and Marco Di Natale. 2016. Mixed criticality systems – a history of misconceptions? *IEEE Design & Test* 33, 5 (2016), 65–74. https://doi.org/10.1109/MDAT.2016.2594790

[30] ESA-ESTEC. 2016. *Techniques for Radiation Effects Mitigation in ASICs and FPGAs Handbook*. Handbook ECSS-Q-HB-60-02A. European Cooperation for Space Standardization, Noordwijk, The Netherlands.

[31] European Committee for Electrotechnical Standardization. 2011. *Railway Applications - Communication, Signalling and Processing Systems - Software for Railway Control and Protection Systems*. Standard EN50128. CENELEC.

[32] Thomas Huining Feng and Edward A. Lee. 2008. Real-time distributed discrete-event execution with fault tolerance. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS*. IEEE, St. Louis, MO, 205–214. https://doi.org/10.1109/RTAS.2008.22

[33] J. Ferreira, A. Oliveira, P. Fonseca, and J. Fonseca. 2004. An experiment to assess bit error rate in CAN. In *Proceedings of 3rd International Workshop of Real-Time Networks*. IEEE, Seoul, Korea, 15–18.
[34] Erol Gelenbe. 1979. On the optimum checkpoint interval. *J. ACM* 26, 2 (April 1979), 259–270. https://doi.org/10.1145/322123.322131
[35] S. Ghosh, R. Melhem, and D. Mosse. 1995. Enhancing real-time schedules to tolerate transient faults. In *Proceedings of the 16th IEEE Real-Time Systems Symposium*. IEEE, Pisa, Italy, 120–129. https://doi.org/10.1109/REAL.1995.495202
[36] Sunondo Ghosh, Rami Melhem, Daniel Mossé, and Joydeep Sen Sarma. 1998. Fault-tolerant rate-monotonic scheduling. *Real-Time Systems* 15, 2 (1998), 149–181. https://doi.org/10.1023/A:1008046012844
[37] O. Goloubeva, M. Rebaudengo, M. S. Reorda, and M. Violante. 2006. *Software-Implemented Hardware Fault Tolerance*. Springer, Boston, MA.
[38] O. Gonzalez, H. Shrikumar, J. A. Stankovic, and K. Ramamritham. 1997. Adaptive fault tolerance and graceful degradation under dynamic hard real-time scheduling. In *Proceedings of the Real-Time Systems Symposium*. IEEE, San Francisco, CA, 79–89. https://doi.org/10.1109/REAL.1997.641271
[39] J. Gracia-Moran, D. Gil-Tomas, L. J. Saiz-Adalid, J. C. Baraza, and P. J. Gil-Vicente. 2010. Experimental validation of a fault tolerant microcomputer system against intermittent faults. In *2010 IEEE/IFIP International Conference on Dependable Systems & Networks (DSN'10)*. IEEE, Chicago, IL, 413–418. https://doi.org/10.1109/DSN.2010.5544288
[40] R. Guerraoui and A. Schiper. 1997. Software-based replication for fault tolerance. *Computer* 30, 4 (1997), 68–74. https://doi.org/10.1109/2.585156
[41] Haryadi S. Gunawi, Riza O. Suminto, Russell Sears, Casey Golliher, Swaminathan Sundararaman, Xing Lin, Tim Emami, Weiguang Sheng, Nematollah Bidokhti, Caitie McCaffrey, Deepthi Srinivasan, Biswaranjan Panda, Andrew Baptist, Gary Grider, Parks M. Fields, Kevin Harms, Robert B. Ross, Andree Jacobson, Robert Ricci, Kirk Webb, Peter Alvaro, H. Birali Runesha, Mingzhe Hao, and Huaicheng Li. 2018. Fail-slow at scale: Evidence of hardware performance faults in large production systems. *ACM Transactions on Storage* 14, 3 (2018), 15. https://doi.org/10.1145/3242086
[42] Z. Guo, S. Vaidhun, L. Satinelli, S. Arefin, J. Wang, and K. Yang. 2021. Mixed-criticality scheduling upon permitted failure probability and dynamic priority. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 40, 5 (2021), 1–1. https://doi.org/10.1109/TCAD.2021.3053232
[43] M. A. Haque, H. Aydin, and D. Zhu. 2014. Real-time scheduling under fault bursts with multiple recovery strategy. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS'14)*. IEEE, Berlin, Germany, 63–74. https://doi.org/10.1109/RTAS.2014.6925991
[44] Tino Heijmen. 2002. *Radiation-induced Soft Errors in Digital Circuits*. Technical Report 828. Philips Electronics. 19 pages. Retrieved April 6, 2023 from https://www.nowpublishers.com/article/DownloadSummary/EDA-018.
[45] Mei Chen Hsueh, Timothy K. Tsai, and Ravishankar K. Iyer. 1997. Fault injection techniques and tools. *Computer* 30, 4 (1997), 75–82. https://doi.org/10.1109/2.585157
[46] Pengcheng Huang, Hoeseok Yang, and Lothar Thiele. 2014. On the scheduling of fault-tolerant mixed-criticality systems. In *Proceedings of the Design Automation Conference*. IEEE, New York, NY, 1–6. https://doi.org/10.1145/2593069.2593169
[47] I. Hwang, S. Kim, Y. Kim, and C. E. Seah. 2010. A survey of fault detection, isolation, and reconfiguration methods. *IEEE Transactions on Control Systems Technology* 18, 3 (2010), 636–653. https://doi.org/10.1109/TCST.2009.2026285
[48] IEEE. 1990. *IEEE Standard 610.12-1990 Glossary of Software Engineering Terminology (Reaffirmed 2002)*. Standard. IEEE, New York, NY,. https://doi.org/10.1109/IEEESTD.1990.101064
[49] International Electrotechnical Commission. 2010. IEC 61508 - Functional Safety of Electrical/electronic/programmable Electronic Safety-related Systems. (Apr 2010).
[50] International Standard Organization. 2018. *Road Vehicles – Functional Safety*. Standard ISO-26262. ISO.
[51] JEDEC. 2003. JESD91A - Method for Developing Acceleration Models for Electronic Component Failure Mechanisms. (Aug 2003). Global Standards for the Microelectronics Industry.
[52] C. Jiang, Z. Zhang, X. Han, and J. Liu. 2013. A novel evidence-theory-based reliability analysis method for structures with epistemic uncertainty. *Computers & Structures* 129 (2013), 1–12. https://doi.org/10.1016/j.compstruc.2013.08.007
[53] M. P. Kaminskiy. 2012. *Reliability Models for Engineers and Scientists*. Taylor & Francis, Boca Raton. https://doi.org/10.1201/b13701
[54] S. Kang, Hoeseok Yang, Sungchan Kim, I. Bacivarov, Soonhoi Ha, and L. Thiele. 2014. Static mapping of mixed-critical applications for fault-tolerant MPSoCs. In *51st ACM/EDAC/IEEE Design Automation Conference (DAC'14)*. IEEE, New York, NY, 1–6. https://doi.org/10.1145/2593069.2593221
[55] J. Karlsson, P. Liden, P. Dahlgren, R. Johansson, and U. Gunneflo. 1994. Using heavy-ion radiation to validate fault-handling mechanisms. *IEEE Micro* 14, 1 (Feb 1994), 8–23. https://doi.org/10.1109/40.259894
[56] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. 2014. Flipping bits in memory without accessing them. In *ISCA'14 Proceedings of the 41st Annual International Symposium on Computer Architecture*. IEEE, Minneapolis, MN, 361–372. https://doi.org/10.1145/2678373.2665726

[57] G. A. Klutke, P. C. Kiessler, and M. A. Wortman. 2003. A critical look at the bathtub curve. *IEEE Transactions on Reliability* 52, 1 (2003), 125–129. https://doi.org/10.1109/TR.2002.804492

[58] Angeliki Kritikakou, Panagiota Nikolaou, Ivan Rodriguez-Ferrandez, Joseph Paturel, Leonidas Kosmidis, Maria K. Michael, Olivier Sentieys, and David Steenari. 2022. Functional and timing implications of transient faults in critical systems. In *IEEE 28th International Symposium on On-Line Testing and Robust System Design (IOLTS'22)*. IEEE, Torino, Italy, 1–10. https://doi.org/10.1109/IOLTS56730.2022.9897537

[59] Seong Woo Kwak, Byung Jae Choi, and Byung Kook Kim. 2001. An optimal checkpointing-strategy for real-time control systems under transient faults. *IEEE Transactions on Reliability* 50, 3 (2001), 293–301. https://doi.org/10.1109/24.974127

[60] G. Latif-Shabgahi, J. M. Bass, and S. Bennett. 2004. A taxonomy for software voting algorithms used in safety-critical systems. *IEEE Transactions on Reliability* 53, 3 (2004), 319–328. https://doi.org/10.1109/TR.2004.832819

[61] E. A. Lee. 2008. Cyber physical systems: Design challenges. In *11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'08)*. IEEE, Orlando, FL, 363–369. https://doi.org/10.1109/ISORC.2008.25

[62] Aiguo Li and Bingrong Hong. 2007. Software implemented transient fault detection in space computer. *Aerospace Science and Technology* 11, 2 (2007), 245–252. https://doi.org/10.1016/j.ast.2006.06.006

[63] Xin Li, Michael C. Huang, Kai Shen, and Lingkun Chu. 2010. A realistic evaluation of memory hardware errors and software system susceptibility. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*. ACM, Boston, MA, 6. http://dl.acm.org/citation.cfm?id=1855840.1855846.

[64] Yau-Tsun Steven Li and Sharad Malik. 1995. Performance analysis of embedded software using implicit path enumeration. *SIGPLAN Not.* 30, 11 (Nov. 1995), 88–98. https://doi.org/10.1145/216633.216666

[65] Jian Denny Lin, Albert M. K. Cheng, Douglas Steel, and Michael Yuchi Wu. 2014. Scheduling mixed-criticality real-time tasks with fault tolerance. In *Proceedings of the 2nd Workshop on Mixed Criticality Systems (WMC'14), RTSS*. WMC, Rome, Italy, 39–44.

[66] Andreas Löfwenmark and Simin Nadjm-Tehrani. 2018. Fault and timing analysis in critical multi-core systems: A survey with an avionics perspective. *Journal of Systems Architecture* 87 (2018), 1–11. https://doi.org/10.1016/j.sysarc.2018.04.001

[67] Florian Many and David Doose. 2011. Scheduling analysis under fault bursts. In *Real-Time Technology and Applications — Proceedings*. IEEE, Chicago, IL, 113–122. https://doi.org/10.1109/RTAS.2011.19

[68] Rami Melhem, Daniel Mossé, and Elmootazbellah Elnozahy. 2004. The interplay of power management and fault recovery in real-time systems. *IEEE Trans. Comput.* 53, 2 (2004), 217–231. https://doi.org/10.1109/TC.2004.1261830

[69] P. Mell, K. Scarfone, and S. Romanosky. 2006. Common vulnerability scoring system. *IEEE Security Privacy* 4, 6 (2006), 85–89. https://doi.org/10.1109/MSP.2006.145

[70] ESA COTS WG 2/3 members. 2020. *ESA COTS Initiative, WG 2/3 Work Synthesis*. Technical Report. European Space Agency, Noordwijk, Netherlands.

[71] Microsemi. 2011. *FPGA Reliability and the Sunspot Cycle*. Technical Report. Microsemi. Retrieved April 6, 2023 from https://www.microsemi.com/document-portal/doc.

[72] D. Mosse, R. Melhem, and S. Ghosh. 1994. Analysis of a fault-tolerant multiprocessor scheduling algorithm. In *Proceedings of IEEE 24th International Symposium on Fault-Tolerant Computing*. IEEE, Austin, TX, 16–25. https://doi.org/10.1109/FTCS.1994.315661

[73] R. Natella, D. Cotroneo, J. A. Duraes, and H. S. Madeira. 2013. On fault representativeness of software fault injection. *IEEE Transactions on Software Engineering* 39, 1 (2013), 80–96. https://doi.org/10.1109/TSE.2011.124

[74] Linwei Niu, Jonathan Musselwhite, and Wei Li. 2018. Work-in-progress: Enhanced energy-aware standby-sparing techniques for fixed-priority hard real-time systems. *Proceedings of the Real-Time Systems Symposium*, 165–168. https://doi.org/10.1109/RTSS.2018.00031

[75] ISO/TC 199 Safety of machinery. 2015. ISO 13849-1:2015 — Safety of Machinery — Safety-related Parts of Control Systems, Part 1: General Principles for Design Hardware. International Organization for Standardization. (Dec 2015). Retrieved April 6, 2023 from https://www.iso.org/standard/69883.html.

[76] N. Oh, P. P. Shirvani, and E. J. McCluskey. 2002. Control-flow checking by software signatures. *IEEE Transactions on Reliability* 51, 1 (2002), 111–122. https://doi.org/10.1109/24.994926

[77] Sam K. Oh and Glenn H. Macewen. 1992. *Toward Fault-tolerant Adaptive Real-time Distributed Systems*. Technical Report. Department of Computing and Information Science, Queen's University, Kingston, Ontario, CA. Retrieved April 6, 2023 from https://research.cs.queensu.ca/TechReports/Reports/1992-325.pdf.

[78] Mihir Pandya and Miroslaw Malek. 1998. Minimum achievable utilization for fault-tolerant processing of periodic tasks. *IEEE Trans. Comput.* 47, 10 (1998), 1102–1112. https://doi.org/10.1109/12.729793

[79] Risat Mahmud Pathan. 2014. Fault-tolerant and real-time scheduling for mixed-criticality systems. *Real-Time Systems* 50, 4 (01 Jul 2014), 509–547. https://doi.org/10.1007/s11241-014-9202-z

[80] Jonathan Pellish. 2018. *Commercial Off-The-Shelf (COTS) Electronics Reliability for Space Applications.* Technical Report. NASA/Goddard Space Flight Center, Greenbelt, MD. Retrieved April 6, 2023 from https://ntrs.nasa.gov/api/citations/20180002659/downloads/20180002659.pdf.

[81] Sasikumar Punnekkat, Alan Burns, and Robert Davis. 2001. Analysis of checkpointing for real-time systems. *Real-Time Systems* 20, 1 (01 Jan 2001), 83–102. https://doi.org/10.1023/A:1026589200419

[82] Marvin Rausand. 2014. *Reliability of Safety–Critical Systems.* John Wiley & Sons, Ltd, Hoboken, NJ. 25–51 pages. https://doi.org/10.1002/9781118776353

[83] Federico Reghenzani, Zhishan Guo, Luca Santinelli, and William Fornaciari. 2022. A mixed-criticality approach to fault tolerance: Integrating schedulability and failure requirements. In *IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS'22).* IEEE, Milan, Italy, 27–39. https://doi.org/10.1109/RTAS54340.2022.00011

[84] Federico Reghenzani, Giuseppe Massari, and William Fornaciari. 2020. Probabilistic-WCET reliability: Statistical testing of EVT hypotheses. *Microprocessors and Microsystems* 77 (2020), 103135. https://doi.org/10.1016/j.micpro.2020.103135

[85] Federico Reghenzani, Gianmario Pozzi, Giuseppe Massari, Simone Libutti, and William Fornaciari. 2016. The MIG framework: Enabling transparent process migration in open MPI. In *Proceedings of the 23rd European MPI Users' Group Meeting (EuroMPI'16).* ACM, New York, NY, 64–73. https://doi.org/10.1145/2966884.2966903

[86] Federico Reghenzani, Luca Santinelli, and William Fornaciari. 2020. Dealing with uncertainty in pWCET Estimations. *ACM Trans. Embed. Comput. Syst.* 19, 5, Article 33 (Sept. 2020), 23 pages. https://doi.org/10.1145/3396234

[87] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. 2005. SWIFT: Software implemented fault tolerance. In *International Symposium on Code Generation and Optimization.* IEEE, San Jose, CA, 243–254. https://doi.org/10.1109/CGO.2005.34

[88] RTCA/EUROCAE. 1992. *DO-178B - Software Considerations in Airborne Systems and Equipment Certification.* Standard. RTCA/EUROCAE.

[89] RTCA/EUROCAE. 2000. DO-178C — Design Assurance Guidance for Airborne Electronic Hardware. (Apr 2000).

[90] S. Safari, M. Ansari, G. Ershadi, and S. Hessabi. 2019. On the scheduling of energy-aware fault-tolerant mixed-criticality multicore systems with service guarantee exploration. *IEEE Transactions on Parallel and Distributed Systems* 30, 10 (2019), 2338–2354. https://doi.org/10.1109/TPDS.2019.2907846

[91] Goutam Kumar Saha. 2006. Software based fault tolerance: A survey. *Ubiquity* 2006, July, Article 1 (July 2006), 1 pages. https://doi.org/10.1145/1149633.1147995

[92] F. SalarKaleji and A. Dayyani. 2013. A survey on fault detection, isolation and recovery (FDIR) module in satellite onboard software. In *6th International Conference on Recent Advances in Space Technologies (RAST'13).* IEEE, Istanbul, Turkey, 545–548. https://doi.org/10.1109/RAST.2013.6581270

[93] Mohammad Salehi, Mohammad Khavari Tavana, Semeen Rehman, Muhammad Shafique, Alireza Ejlali, and Jörg Henkel. 2016. Two-state checkpointing for energy-efficient fault tolerance in hard real-time systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 24, 7 (2016), 2426–2437. https://doi.org/10.1109/TVLSI.2015.2512839

[94] Tianning She, Sudharsan Vaidhun, Qijun Gu, Sajal Das, Zhishan Guo, and Kecheng Yang. 2021. Precise scheduling of mixed-criticality tasks on varying-speed multiprocessors. In *29th International Conference on Real-Time Networks and Systems.* RTNS, Virtual, 10.

[95] K. G. Shin, T. Lin, and Y. Lee. 1987. Optimal checkpointing of real-time tasks. *IEEE Trans. Comput.* C-36, 11 (1987), 1328–1341. https://doi.org/10.1109/TC.1987.5009472

[96] Lucas Antunes Tambara, Paolo Rech, Eduardo Chielle, Jorge Tonfat, and Fernanda Lima Kastensmidt. 2016. Analyzing the impact of radiation-induced failures in programmable SoCs. *IEEE Transactions on Nuclear Science* 63, 4 (2016), 2217–2224. https://doi.org/10.1109/TNS.2016.2522508

[97] Abhilash Thekkilakattil, Radu Dobrin, and Sasikumar Punnekkat. 2015. Fault tolerant scheduling of mixed criticality real-time tasks under error bursts. *Procedia Computer Science* 46 (2015), 1148–1155. https://doi.org/10.1016/j.procs.2015.01.027
Proceedings of the International Conference on Information and Communication Technologies, ICICT 2014, 3-5 December 2014 at Bolgatty Palace & Island Resort, Kochi, India.

[98] Abhilash Thekkilakattil, Radu Dobrin, Sasikumar Punnekkat, and Huseyin Aysan. 2012. Resource augmentation for fault-tolerance feasibility of real-time tasks under error bursts. In *Proceedings of the 20th International Conference on Real-Time and Network Systems (RTNS'12).* ACM, New York, NY, 41–50. https://doi.org/10.1145/2392987.2392992

[99] Massimo Tipaldi and Bernhard Bruenjes. 2015. Survey on fault detection, isolation, and recovery strategies in the space domain. *Journal of Aerospace Information Systems* 12, 2 (2015), 235–256. https://doi.org/10.2514/1.I010307 arXiv:https://doi.org/10.2514/1.I010307

[100] Steve Vestal. 2007. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *Proceedings of the Real-Time Systems Symposium.* IEEE, Tucson, AZ, 239–243. https://doi.org/10.1109/RTSS.2007.47

[101] G. von der Brüggen, K. H. Chen, W. H. Huang, and J. J. Chen. 2016. Systems with dynamic real-time guarantees in uncertain and faulty execution environments. In *2016 IEEE Real-Time Systems Symposium (RTSS'16)*. IEEE, Porto, Portugal, 303–314. https://doi.org/10.1109/RTSS.2016.037

[102] C. Wang, H. Kim, Y. Wu, and V. Ying. 2007. Compiler-managed software-based redundant multi-threading for transient fault detection. In *International Symposium on Code Generation and Optimization (CGO'07)*. IEEE, San Jose, CA, 244–258. https://doi.org/10.1109/CGO.2007.7

[103] C. Wang, F. Mueller, C. Engelmann, and S. L. Scott. 2008. Proactive process-level live migration in HPC environments. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (SC'08)*. IEEE, Austin, TX, 1–12. https://doi.org/10.1109/SC.2008.5222634

[104] J. H. Wensley, L. Lamport, J. Goldberg, M. W. Green, K. N. Levitt, P. M. Melliar-Smith, R. E. Shostak, and C. B. Weinstock. 1978. SIFT: Design and analysis of a fault-tolerant computer for aircraft control. *Proc. IEEE* 66, 10 (1978), 1240–1255. https://doi.org/10.1109/PROC.1978.11114

[105] Geoff Whittington. 2016. A critical look at the IASB. *Pioneers of Critical Accounting* 52, 1 (2016), 179–200. https://doi.org/10.1057/978-1-137-54212-0_10

[106] K. Yang, A. Bhuiyan, and Z. Guo. 2020. F2VD: Fluid rates to virtual deadlines for precise mixed-criticality scheduling on a varying-speed processor. In *2020 IEEE/ACM International Conference on Computer Aided Design (ICCAD'20)*. IEEE/ACM, San Diego, CA, 1–9. https://doi.org/10.1145/3400302.3415716

[107] K. S. Yim, V. Sidea, Z. Kalbarczyk, D. Chen, and R. K. Iyer. 2012. A fault-tolerant programmable voter for software-based N-modular redundancy. In *2012 IEEE Aerospace Conference*. IEEE, Big Sky, MT, 1–20. https://doi.org/10.1109/AERO.2012.6187253

[108] John W. Young. 1974. A first order approximation to the optimum checkpoint interval. *Commun. ACM* 17, 9 (Sept. 1974), 530–531. https://doi.org/10.1145/361147.361115

[109] Ying Zhang and K. Chakrabarty. 2006. A unified approach for fault tolerance and dynamic power management in fixed-priority real-time embedded systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 25, 1 (2006), 111–125. https://doi.org/10.1109/TCAD.2005.852657

[110] Yi-Wen Zhang and Rong-Kun Chen. 2022. A survey of energy-aware scheduling in mixed-criticality systems. *Journal of Systems Architecture* 127 (2022), 102524. https://doi.org/10.1016/j.sysarc.2022.102524

[111] Baoxian Zhao, Hakan Aydin, and Dakai Zhu. 2013. Shared recovery for energy efficiency and reliability enhancements in real-time applications with precedence constraints. *ACM Transactions on Design Automation of Electronic Systems* 18, 2 (2013), 21. https://doi.org/10.1145/2442087.2442094

[112] Z. Zhengyong, P. Liping, and Y. Fumin. 2014. Schedulability analysis for fault tolerance real-time system under fault bursts. In *2014 IEEE 7th Joint International Information Technology and Artificial Intelligence Conference*. IEEE, Chongqing, China, 20–27. https://doi.org/10.1109/ITAIC.2014.7064998

[113] Junlong Zhou, Min Yin, Zhifang Li, Kun Cao, Jianming Yan, Tongquan Wei, Mingsong Chen, and Xin Fu. 2017. Fault-tolerant task scheduling for mixed-criticality real-time systems. *Journal of Circuits, Systems and Computers* 26, 1 (2017), 1–17. https://doi.org/10.1142/S0218126617500165

[114] D. Zhu and H. Aydin. 2009. Reliability-aware energy management for periodic real-time tasks. *IEEE Trans. Comput.* 58, 10 (2009), 1382–1397. https://doi.org/10.1109/TC.2009.56

[115] Haissam Ziade, Rafic Ayoubi, and Raoul Velazco. 2004. A survey on fault injection techniques. *The International Arab Journal of Information Technology* 1, 2 (2004), 171–186.

Received 23 April 2021; revised 19 January 2023; accepted 23 March 2023