A Mixed-Criticality Approach to Fault Tolerance: Integrating Schedulability and Failure Requirements

Federico Reghenzani*§, Zhishan Guo†, Luca Santinelli‡, and William Fornaciari*
*Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Italy, [name].[surname]@polimi.it
†Department of Electrical and Computer Engineering, University of Central Florida, USA, zsguo@ucf.edu
‡Airbus Defense & Space, Germany, luca.santinelli@airbus.com
§ESTEC, European Space Agency, Netherlands

***Abstract*—Mixed-Criticality (MC) systems have been widely studied in the past decade, majorly due to their potential to consolidate applications with different criticality levels onto the same platform. In the original design proposed by Vestal, a target probability of failure per hour specified by certification requirements is assigned to each criticality level. These requirements have been mainly conceived for hardware faults. Software fault tolerance techniques are available to mitigate hardware faults, but their adaptation to real-time systems is challenging due to the introduced overhead. This paper proposes an extension to the traditional MC scheduling theory to implement fault tolerance strategies against transient faults, with the goal of complying with both failure and timing requirements. In particular, we introduce the dropping relationships that generalize the concept of criticality and allow, on the one hand, to improve the schedulability analysis, on the other, to control the dependency between tasks satisfying the certification requirements. The simulation study shows a schedulability ratio improvement of 20-30% compared to classical scheduling while maintaining compliance with failure requirements.**

***Index Terms*—Real-Time, Mixed-Criticality, Fault-Tolerance, SIHFT.**

I. Introduction

The resilience to faults is an essential property of most safety-critical systems. Hardware is inevitably subject to both transient and permanent faults; their probability can be reduced by appropriate manufacturing processes, but not zeroed. Unfortunately, chip miniaturization causes modern architectures to be more susceptible to transient faults. It has been observed [1] that in modern memories, the probability of experiencing a transient fault is in the order of 10^{-5} per hour. The problem is amplified when considering aerospace applications, because the atmospheric and magnetic shields of the Earth have a reduced effect at high altitudes. This is especially concerning for reconfigurable architectures: recent studies [2], [3] showed how FPGA devices could exhibit a fault rate larger than 10^{-2}/h in the space environment.

According to the IEEE glossary [4], a *fault* is a defect in the functioning of a hardware device or component. In the case of transient faults, they are usually modeled as Single Event Upsets (SEUs), i.e., bit-flips occurring in memory regions (including CPU registers). The *error* is the effect caused by the presence of faults to some computation, e.g., a miscalculated value. The error can, in turn, cause the *failure*, i.e., the inability of the system to perform the required function according to the original requirements. Fault tolerance techniques can be implemented both at the hardware and software levels. Hardware solutions are traditionally used in safety-critical systems, and many state-of-the-art techniques exist. However, they are usually expensive in terms not only of financial costs but also of energy, space, and weight of the final system [5]. For this reason, software mechanisms are attractive to implement the necessary fault tolerance strategies. They are usually called with the umbrella term *Software-Implemented Hardware Fault Tolerance (SIHFT)*. The mainly used SIHFT approaches are: (1) the re-execution of a failed task, (2) checkpoint/restore mechanisms, (3) the execution of recovery blocks, and (4) the task replication. All of these software techniques introduce, however, challenges for hard real-time systems, where the timing requirements must be satisfied.

A. Overview of Standards

We contextualize this paper in the avionics and automotive domains. The standard ecosystems describe processes and methods for performing and validating the safety assessment for certification. In particular, for avionics, the AC 25.1309-1/AMC 25.1309 establishes the principle that the more severe the hazard resulting from a system or equipment failure, the less likely that failure must be. The safety assessment is defined in the ARP-4761 for the system level, while the analysis of failure effects at equipment level is recognized in DO-254 for hardware items and in DO-178C for software items. The aircraft system functions are systematically analyzed for failure conditions, and each failure condition is assigned a hazard classification, called Design Assurance Level (DAL), with associated maximum allowable failure probability per flight hour: 10^{-9} for DAL A (Catastrophic), 10^{-7} for DAL B (Hazardous), 10^{-5} for DAL C (Major), 10^{-3} for DAL D (Minor), and no probability associated for DAL E (No Safety). Similarly, in the automotive domain, the ISO 26262 and IEC 61508 define a similar classification named Automotive Safety Integrity Levels (ASIL). However, the failure requirements defined by the standards are for hardware failures, while software failures are considered only as systematic (i.e., bugs and defects), therefore only qualitative assessments of failure conditions are possible. This concept limits the applicability

TABLE I

Example of a task set, where T_i is the period, D_i is the deadline, C_i is the WCET, and p_i^{req} is the required failure rate.

Task	$T_i = D_i$	C_i	DAL	p_i^{req}
τ_1	50	10	A	10^{-9}/h
τ_2	1000	75	A	10^{-9}/h
τ_3	250	50	B	10^{-7}/h
τ_4	100	25	D	10^{-3}/h

of some scheduling algorithms and software fault tolerance techniques, as described in the subsequent paragraphs.

B. Mixed-Criticality and Its Compliance with Standards

In the last decade, Mixed-Criticality (MC) systems emerged to improve the scheduling capacity of real-time systems. The Worst-Case Execution Time (WCET) estimations in modern architectures are often overly pessimistic, making the schedulability of hard real-time systems with limited resources challenging. To overcome this problem, in 2007, Vestal [6] proposed the MC model, where each task is assigned a criticality level that conceptually corresponds to the aforementioned DALs. The criticality levels are interpreted as the levels of assurance of the WCET estimation. One or more WCET values estimated at different levels of assurance is assigned to each task, depending on the criticality level. For instance, according to the DAL characterization, the Vestal's model assigns five different WCET estimations to a task belonging to DAL A, from the most pessimistic, but safe, to the less pessimistic, but possibly unsafe. When a task overruns one of its WCET estimations, a *mode switch* occurs. To deal with mode switches, traditional methods drop tasks of lower criticality than the WCET criticality of the task that overran. However, several discussions emerged recently on this strategy [7]–[9], which is considered far from the industry perspective. In fact, dropping tasks during mode-switch creates a "dependency" between the tasks, i.e., the functional correctness of a job of the task does not depend on the job itself, but it is influenced by the execution of other tasks. This is a violation of the independence property of safety-critical systems. For example, IEC61508 states: *"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."* Similar requirements can be found in the other standards mentioned above. Traditional MC approaches that drop tasks do not comply with the standards because the introduced dependence among the tasks makes condition (1) false, and the unpredictability of mode switches makes also condition (2) false.

To solve this issue, we propose to satisfy the condition (2) of IEC61508-3, by "controlling" the dependence violation. In particular, we show how to perform a joint scheduling and failure analysis, able to obtain quantitative data on the effects of dropping jobs due to the WCET overruns of other tasks, which is, in turn, caused by the triggering of fault tolerance mechanisms.

C. A Motivational Example

To give an insight of the proposed approach, we begin with a motivational example based on the small task set presented in Table I. Tasks τ_1 and τ_2 belong to the highest level DAL A, τ_3 to the DAL B, and τ_4 to DAL D, according to the safety integrity levels of DO-178C and DO-254. According to the failure rate, we can consider τ_1 and τ_2 as HI-criticality, τ_3 as MI-criticality, and τ_4 as LO-criticality. However, this example shows the limit of the majority of MC approaches: we cannot drop the MI-criticality task τ_3, because of a mode switch caused by the overrun of the MI-criticality WCET of one of the HI-criticality tasks. Indeed, the task τ_3 still provides an essential and inalienable feature with a well-defined requirement of failure rate. For example, if τ_1 overruns its MI-criticality WCET with a rate $p = 10^{-5}$/h, then this triggers the drop of τ_3 with a rate of at least 10^{-5}/h, violating, in this way, τ_3's failure requirement. By allowing other tasks to trigger the drop of τ_3, we are creating a dependence between tasks and violating the requirement of the previously mentioned standards. Moreover, we cannot *control* the probability that a mode switch occurs, because p is usually unknown[1]. Realistic MC task sets may be composed of over 200 different tasks [11], exacerbating the problem.

The task set has an utilization of $U = 0.725 < 1$, thus it is schedulable on a single processor – e.g., with the Earliest Deadline First (EDF) scheduler. However, let us consider that the system suffers from independent transient faults with a fault rate of 10^{-4}/h. With such a value, failure requirements of τ_1, τ_2, and τ_3 could not be met[2]. One possible solution is to budget for re-executing a failed job, thus decreasing the failure rate to 10^{-8}/h (because two faults have to happen). This is sufficient for τ_3 but not for τ_1 and τ_2. Thus, for these two tasks we need to re-execute a failed job 3 times (if necessary) in order to meet the failure rate requirements (becoming 10^{-12}/h). Unfortunately, allocating time to execute all the re-execution jobs implies an utilization $U = 1.475$ that is not schedulable on a uni-processor. A possible solution to this problem is to drop the task τ_4 if a fault occurs in τ_1, τ_2, or τ_3. In this case, even if the re-execution job is introduced, the task set remains schedulable. This, at first glance, is a violation of the task-independence requirement. Nevertheless, contrarily to the traditional MC problem, the probability of mode-switch is *known*. In fact, even if the failure rate of τ_4 is no longer 10^{-4}/h but it also includes the effect of other tasks – becoming $1-(1-10^{-4})^4 \approx 3 \cdot 10^{-4}$/h –, it still meets the failure rate requirement of τ_4. Similarly, to allocate the 2nd re-execution of τ_1 and τ_2, τ_3 can be dropped, still meeting its failure requirement: 10^{-7}/h.

Unlike traditional MC, where the probability of exceeding WCET thresholds is unknown and the resulting dependence uncontrolled, we can mathematically bound the failure rate

[1]Probabilistic-WCET analyses can estimate p, but their safe use in the critical systems is currently disputed and yet to be formally demonstrated [10].

[2]For the sake of this example, the computation of fault/failure rates are simplified for brevity and clarity purposes. How to derive the exact values is described later in the manuscript.

caused by mode switches, provided that a careful calculation is done. In the previous example, the failure rate of τ_4 including other task effects is $3 \cdot 10^{-4}$/h, satisfying the DAL D requirement. As previously mentioned in Section I-B, IEC61508 requires that any violation of independence must be controlled. The standard does not precisely specify what "controlled" means. *We believe that a mathematically computed failure rate upper-bounding the requirement is sufficient to adhere to the "controlled" condition*. The availability of a safe bound allows us to go beyond the non-quantifiability of software failure rates specified by the standards.

D. Contribution

In this work, we propose a generalization of the traditional MC setup with the so-called Dropping Relations (DRs). Theoretical tools to compute the fault/failure rate and verify the requirements in presence of the DRs are developed and presented. The use of DRs allows us to improve the schedulability with respect to hard real-time constraints in presence of SIHFT algorithms for transient faults occurring in memories or CPU cores. This paper advances the current MC theory by pursuing the following goals: 1) generalize the concept of system modes with the DRs; 2) improve the schedulability of real-time tasks when fault tolerance tasks are introduced (in particular, with the re-execution mechanism); 3) guarantee the "controlled dependency" requirement of the standards even if task dropping occurs. Besides the theoretical achievement, we performed an experimental simulation to quantify the improvement with respect to traditional techniques.

II. Model and Assumptions

A. Task and Failure Model

The set of tasks is identified by $\Gamma = \{\tau_1, \tau_2, ..., \tau_n\}$. Each task is modeled by the following tuple:

$$\tau_i = (C_i, D_i, T_i, p_i^{\text{req}})$$

where C_i is the WCET, D_i is the deadline, T_i is the period or inter-arrival time, p_i^{req} is the required maximum task failure rate coming from the safety requirement analysis. All deadlines are assumed to be *constrained* ($D_i \leq T_i$).

The fault tolerance mechanism we consider in this work is the *re-execution* mechanism: if a fault is detected during a job execution, the same computation is restarted. We model this behavior as a set of extra tasks activated on-demand. The stricter the failure requirement needed, the higher the number of re-execution tasks that are potentially activable. We define Δ as the set of all tasks including the re-execution tasks: $\Delta = \{\tau_{1(0)}, \tau_{1(1)}, ..., \tau_{2(0)}, ..., \tau_{n(N_n^{\text{re-exec}})}\}$ where $\tau_{i(k)}$ represents the k-th re-execution of task τ_i, or, if $k = 0$, the primary execution of τ_i. For example, $\tau_{1(0)}$ is the "normal" execution of task τ_1, $\tau_{1(1)}$ is the task spawning the first re-execution job, etc. The re-execution tasks $\tau_{i(k)}$ are intrinsically sporadic, because the jobs of $\tau_{i(k)}$ for $k > 0$ are activated only if the corresponding job for task $k-1$, i.e., $\tau_{i(k-1)}$, suffered a fault. The total number of re-execution tasks $N_i^{\text{re-exec}}$ is computed per-task depending on its failure requirement. Further details of the computation of $N_i^{\text{re-exec}}$ are described later in the manuscript (Eq. (7)). All the re-execution tasks share the same WCET value C_i being exactly the same workload. In order to guarantee the temporal correctness, the jobs of re-execution tasks $\tau_{i(k)}$ have the same absolute deadlines of the jobs of $\tau_{i(0)}$.

Task failure model. A job of a task is considered to have a failure when it does not produce the correct results according to its specification. In real-time systems, the results' correctness is defined in both logical and temporal senses. Faults can affect both: for example, an SEU can cause an error that changes the result value, possibly violating the logical correctness, or increasing the counter variable of a loop, possibly violating the temporal correctness. The fault detection is possible via any state-of-the-art techniques (e.g., acceptance tests, watchdog, or control-flow-graph signatures — see [12] for a comprehensive survey). The choice of a particular fault-detection algorithm does not affect the discussions in this work. Any overhead caused by fault detection is considered as part of the WCET C_i. No perfect fault-detection algorithm can be built, but we assume their failure probability as already included in the overall system failure analysis. Similarly, the re-execution mechanism cannot resolve all the possible errors caused by transient faults. For instance, an SEU occurring in the memory region of the state or input data persisting across job execution cannot be fixed with re-executing the job. We assume the failure probability of such situations as already included in the overall system failure analysis or their tolerance provided by other mechanisms, which overhead is included in the WCETs of the tasks. Standards already consider the imperfection of the fault detection and recovery algorithms: ISO 26262, for instance, defines the *level of coverage* of each detection/recovery mechanism, which must meet well-defined requirements.

When the output of a job is not correctly produced, either in logical or timing sense, the task is immediately considered to have a failure. More precisely,

Definition 1 (Task failure)**.** A task τ_i is considered to have a *failure* – and by extension the whole system in case τ_i is critical, i.e., $p_i^{\text{req}} < 1$ – if any job of task $\tau_{i(N_i^{\text{re-exec}})}$ is activated but failed to produce a correct result by the absolute deadline of the corresponding job of the original task $\tau_{i(0)}$.

This definition implicitly means that the previous jobs of all the tasks $\tau_{i(k)}$ with $k < N_i^{\text{re-exec}}$ have failed. In fact, if any job of the task $\tau_{i(k)}$ does not fail, the job of the task $\tau_{i(k+1)}$ would have never been activated and, therefore, not even $\tau_{i(N_i^{\text{re-exec}})}$. This definition is an obvious consequence of the classical definition of hard real-time systems: no deadline misses are tolerated and, consequently, the interest from the failure standpoint is related solely to the first failure, i.e., the *dangerous failure* as called in safety engineering.

B. System Model

The system is identified by a set of resources $\mathcal{R} = \{R_1, R_2, ..., R_{m_R}\}$ assignable to tasks. Each resource can be either a CPU core or a memory. The resource manager or the scheduling policy assigns to each task one or more resources in $\mathcal{R}$. We then define the resource assignment function $r : \Delta \times \mathcal{R} \to [0; 1]$ as follows: $r(\tau, R) = 1$ if the task is assigned to the resource, $r(\tau, R) = 0$ if the task is not assigned to the resource, and a value in the interval $(0, 1) \in \mathbb{Q}$ if the task partially uses the resource. The latter case refers to memories, which are space-assigned, and not CPU cores, which are, instead, time-assigned. Please note that, in this notation, there is no concept of *how much time* a task uses a resource—it will be introduced later in Section III-B. The resource assignment function $r(\cdot)$ for CPU cores, is either 0 or 1; this is because, at a given time instant, the resource can be only used by one task. Each task is also considered single-thread, so there is only one computing unit assigned to each task. Even if multi-threaded applications are not the subject of this work, they can be represented as multiple tasks, provided that an analysis of the dependency for both timing and failure standpoints is performed.

In order to properly formulate the subsequent analysis and in common with almost all of the state-of-the-art works, we assume the Operating System (OS), or at least the scheduling and fault detection algorithms, as non-affectable by faults. This assumption can be implemented in a real system in different manners, e.g., redundancy in the OS routines, specialized hardware implementations [13], [14], or by considering their failure probability in the safety analysis.

C. Fault Model

In this paper, we restrict the discussion on transient faults, while the analysis of permanent faults and fault bursts [15] is left as future work. The probability of observing a transient fault within a given duration $\mathfrak{F}$ is often referenced as the *fault rate*. The fault rate is typically expressed per-hour, i.e., $\mathfrak{F} = 1$ h. Each resource is characterized by a different fault rate λ_i, computed by using estimation methods developed since decades ago [16]. This value refers to the whole resource: it represents the probability of observing a fault in the timeframe $\mathfrak{F}$ in any component of the resource R_i. For example, if R_i is a memory, then λ_i refers to the probability of observing a fault in any memory location of R_i in $\mathfrak{F}$. The probability of observing a hardware fault in a given time instant is considered *independent and identically distributed (iid)*. This assumption is realistic and widely adopted in industrial contexts [17]. In the case of a resource having different sub-components with different fault rates, this model considers them as separated resources.

Table II summarizes the introduced notations.

D. The Analysis Flow

In order to fulfill both temporal and failure requirements in an integrated manner, the proposed offline analysis is composed of a mix of failure and schedulability analyses. In particular, the following steps, also depicted in Figure 1, are performed:

1) Preliminary failure analysis: the failure rate of tasks is computed, and the maximum number of re-execution tasks to admit to scheduling is selected according to the failure requirements.
2) If the resulting task set Δ is trivially schedulable (e.g., by EDF), then the process successfully terminates.
3) If not, a joint scheduling and failure analysis is performed to find a set of DRs, as subsequently defined in Section IV-A, to improve its schedulability.
4) The two analyses are performed again to ensure that requirements are still valid even after the introduction of the DRs: they improve the schedulability, but they can negatively affect the fulfillment of failure requirements.

TABLE II
SUMMARY OF KEY NOTATIONS.

Symbol	Description
Γ	All tasks w/o re-execution tasks
Δ	All tasks w/ re-execution tasks
$\tau_{i(j)}$	The j-th re-execution task of the i-th task
$N_i^{\texttt{re-exec}}$	Number of max re-execution tasks
$R_i \in \mathcal{R}$	A computing resource
λ_i	Fault rate associated to i-th resource
$r(\cdot, \cdot)$	Resource assignment function
$p_i^{\texttt{req}}$	Max failure rate (requirement)

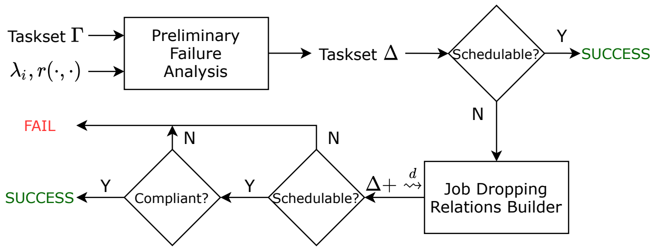


Fig. 1. The high-level flowchart of the integrated analyses.

III. Failure analysis

The failure analysis aims to compute the probability of a transient fault to occur for a single job and derive the minimum number of re-execution tasks necessary to comply with the failure rate requirements.

A. Probability of An Event for A Given Time Frame

In probability, the outcome of a sequence of trials is expressed as a sequence of *random variables* $E_1, E_2, ..., E_n$. In our problem, the random variables represent the absence "0" or the presence "1" of a fault at a given time point. The statistical process generating a sequence of Boolean variables is the *Bernoulli process*. Since we consider the time as a discrete variable (because at the finest scale it corresponds to the clock cycles), their statistical law is a binomial distribution:

$$P(E = k) = \binom{n}{k} p^k (1-p)^{n-k} \tag{1}$$

that corresponds to the probability of observing k faults in a sequence of n trials, i.e., $E = \sum_i^n E_i$ and $p = P(E_i = 1)$. According to our failure model described in Section II, we are interested in the probability of observing at least one fault: $P(E > 0)$. We derive this probability from its complementary $P(E > 0) = 1 - P(E = 0)$:

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

For the sake of completeness, it should be noted that for large n and small p, this probability converges to the Poisson distribution with $k = 0$, i.e., $P(E > 0) \rightarrow 1 - e^{-np}$. This property is frequently used in safety analyses [17] and it is called *Homogeneous Poisson Process (HPP)* hypothesis.

By exploiting the formulas of the previous paragraph, it is possible to change the time frame of a given probability of failure. For example, if the requirement failure rate is expressed *per-hour*, it is possible to divide, thanks to the *iid* hypothesis, this time frame *per-minute* by setting $n = 60$ and by computing the inverse of Eq. (2).

B. Probability of a Fault to Occur for a Given Job

From the fault rate λ_i of the resource R_i expressed in the time frame $\mathfrak{F}$, we scale the λ_i to become the probability of a fault to occur in one time instant, defined as the time unit used in the task model. The time unit is often expressed as the number of clock cycles. Let f^{-1} be the period of the clock cycles, then we can compute how many clock cycles are in $\mathfrak{F}$: $k = \frac{\mathfrak{F}}{f^{-1}}$. Having this number, for each resource R_i, we compute the fault probability at any time instant by inverting Eq. (2):

$$\lambda'_i = 1 - (1 - \lambda_i)^{\frac{1}{k}}. \tag{3}$$

To compute the probability that a fault affects a job, we have to consider the resources it uses (and in which percentage, i.e., the $r(\tau_i, R_k)$ value) and the exposure time, i.e., the time interval in which a fault occurring in the resource can affect the job execution. The probability that a fault affects a job of the task τ_i is:

$$p_i^{\mathtt{F}} = 1 - \prod_{\forall k: R_k \in \mathcal{R}} (1 - r(\tau_i, R_k)\lambda'_k)^{\epsilon_{i,k}} \tag{4}$$

where $\epsilon_{i,k}$ is the exposure time of the task τ_i for the resource R_k over its period T_i.

Note 1. The term λ'_k is the probability of a fault to occur in a given resource R_k at any time instant. Multiplying this term by $r(\tau_i, R_k)$, we obtain the fault probability for the share the task τ_i uses the resource R_k (if > 0). Let us call this value $x = r(\tau_i, R_k)\lambda'_k$. Being this term the probability in one time instant, we find the probability {the fault occurs at time $t = 0$} $\cup$ {the fault occurs at time $t = 1$} $\cup$ etc. This union of events is repeated for the whole exposure time $\epsilon_{i,k}$. The fault rate for the whole job, similar to Eq. (1), becomes: $p_{i,k}^{\mathtt{F}} = 1 - \prod_{t=0}^{\epsilon_{i,k}}(1 - x) = 1 - (1 - x)^{\epsilon_{i,k}}$. Then, we compute the joint fault rate for all the resources by applying one more time the union probability formula: $p_i^{\mathtt{F}} = 1 - \prod_{\forall k: R_k \in \mathcal{R}}(1 - p_{i,k}^{\mathtt{F}})$. Both steps in which we applied the union formula are possible because the events are *iid*. Replacing the $p_{i,k}^{\mathtt{F}}$ and applying simple algebra, we obtain Eq. (4).

The value $\epsilon_{i,k}$ is expressed with a worst-case value depending on the resource type. For example, for a processor R_k, $\epsilon_{i,k}$ is equal to the WCET C_i, because a task can use the processor for at most C_i time units. A memory area active from the start to the end of the job is exposed in the worst-case for $C_i + I_i$ time units, where I_i is the maximum interferences from higher-priority tasks (preemption time). In first approximation, $\epsilon_{i,k} = T_i$ is a valid, but pessimistic, value for any resource under constrained deadline assumption. Indeed, $\epsilon_{i,k}$ cannot be larger than T_i, because it would be larger than D_i and therefore non-schedulable.

Example 1. Let τ_1 be a task that uses one core (R_1) and two memory address spaces in two memory nodes (R_2) and (R_3). Each memory is used by the task at the 20% of the whole size, consequently: $r(\tau_1, R_1) = 1, r(\tau_1, R_2) = 0.2, r(\tau_1, R_3) = 0.2$. Its WCET and period are $C_1 = 1\,000$ and $T_1 = 10\,000$ (clock cycles). The fault rates are $\lambda_{R_1} = 10^{-9}/h, \lambda_{R_2} = 10^{-8}/h, \lambda_{R_3} = 10^{-8}/h$. Computing the scaled probability of failure at any time instant (by assuming $f = 100MHz$), we obtain with Eq. (3): $\lambda'_{R_1} \approx 4 \cdot 10^{-15}, \lambda_{R_2} \approx 4 \cdot 10^{-14}, \lambda_{R_3} \approx 4 \cdot 10^{-14}$. Then, assuming $\epsilon_{1,1} = C_1$, $\epsilon_{1,2} = T_1$, $\epsilon_{1,3} = T_1$, we get the following job failure rate thanks to Eq. (4): $p_i^{\mathtt{F}} \approx 2 \cdot 10^{-10}/h$. □

C. From the Safety Requirement to the Job Failure Bound

The safety requirement of each task $p_i^{\mathtt{req}}$ is often expressed as PFH (average frequency of dangerous failure[3]) or PFD (Probability of Failure on Demand) [17]. The PFH is used for components that provide the safety operations in continuous mode, while PFD is related to on-demand functions. In the continuous case, a fault immediately triggers a hazardous event, while the on-demand case requires both the fault and the activation of the function (demand) to occur simultaneously. The book of Rausand et al. [17] explains in details the PFD and PFH concepts. These requirements can be mapped to the real-time model by categorizing the tasks respectively in periodic and sporadic ones.

According to the task and failure models of Section II-A, we define the following system property:

Definition 2 (Compliant task set)**.** A task set Δ is said to be *compliant with respect to the failure requirements* (shorten, *compliant*) if the following condition holds for all tasks:

$$p_i^{\mathtt{J}} \leq p_i^{\mathtt{req}} \quad \forall \tau_i$$

where $p_i^{\mathtt{J}}$ is the total failure rate of a single job of the i-th task and $p_i^{\mathtt{req}}$ is the given maximum failure rate for the i-th task (requirement). □

If no fault tolerance mechanisms are available (i.e., $\Delta = \Gamma$), the value of $p_i^{\mathtt{J}}$ is the same of the single job failure rate: $p_i^{\mathtt{J}} = p_i^{\mathtt{F}}$. The PFH is frequently expressed as per-hour, while the PFD requirement is, instead, expressed as the probability of failure when a demand (i.e., a job spawn in sporadic task model) occurs: PFD already represents the value $p_i^{\mathtt{req}}$, while PFH must be shifted to the WCET timescale in a similar way we did for the $p_i^{\mathtt{F}}$. Let $\mathfrak{F}$ be the PFH time frame, then $N_i^{\mathtt{act}} = \left\lceil \frac{\mathfrak{F}}{T_i} \right\rceil$ is the number of jobs of $\tau_{i(0)}$ activated in the time frame $\mathfrak{F}$. By inverting Eq. (2), we obtain:

[3]The acronym refers to the original name "Probability of Failure per Hour" of the IEC 61508 standard, later changed in the subsequent version of the standard to reflect any $\mathfrak{F}$ value.

$$p_i^{\texttt{req}} = 1 - (1 - PFH)^{1/N_i^{\text{act}}} \quad (5)$$

It may happen that the failure rate of a given job execution $p_i^{\texttt{F}}$ is larger than the requirement $p_i^{\texttt{req}}$. This is the reason for the necessity of fault tolerance algorithms. When they are correctly applied, the total failure rate of a task ($p_i^{\texttt{J}}$) decreases depending on the type of the algorithm. For the re-execution paradigm, this rate can be computed from the number of re-execution tasks $N_i^{\texttt{re-exec}}$:

$$p_i^{\texttt{J}} = \left(p_i^{\texttt{F}}\right)^{N_i^{\texttt{re-exec}}} \quad (6)$$

When a job is detected as failed, the job is restarted if $N_i^{\texttt{re-exec}} \geq 1$. If the re-started job fails, then the job is restarted again if $N_i^{\texttt{re-exec}} \geq 2$, and so on for a maximum of $N_i^{\texttt{re-exec}}$ times. The minimum number of re-execution necessary to meet the requirements is:

$$N_i^{\texttt{re-exec}} = \max\left(0, \left\lceil \log_{p_i^{\texttt{F}}}\left(p_i^{\texttt{req}}\right)\right\rceil - 1\right) \quad (7)$$

The maximization is necessary because if the task is non-critical, i.e., $p_i^{\texttt{req}} = 1$, then the second term is negative and $N_i^{\texttt{re-exec}}$ would not be meaningful. This parameter and the consequent new task set Δ represent the output of the preliminary failure analysis.

Example 2. Considering the task of Example 1 having a period of $T_1 = 10\,000$ clock cycles and a required $PFH = 10^{-9}/h$, the number of job in $\mathfrak{F}$ is $N_i^{\texttt{act}} = 36 \cdot 10^6$ and the requirement per job is consequently: $p_i^{\texttt{req}} = 1 - (1 - 10^{-9})^{1/36\cdot 10^6} \approx 2.7 \cdot 10^{-17}$. Since $p_i^{\texttt{F}} \approx 2 \cdot 10^{-10} > p_i^{\texttt{req}}$, re-execution tasks are mandatory and their minimum number is: $N_i^{\texttt{re-exec}} = \lceil 1.71 \rceil - 1 = 1$. With one re-execution task, the total task failure rate becomes: $p_i^{\texttt{F}} \approx 4 \cdot 10^{-20} < p_i^{\texttt{req}}$. □

D. Extension to the k-out-of-n case

Even if not directly exploited in this work, it is worth to cite the k-out-of-n failure model and how it can be integrated with the proposed analysis. This approach is part of the weakly hard real-time computing [18] and its characteristic is to tolerate the deadline miss of k-out-of-n jobs. This failure model is opposed to the dangerous failure model presented in Section II-A and is more linked to the *availability* concept of safety-engineering rather than *reliability* (see, for instance, the recent work by Zhou et al. [19] on the availability of real-time systems). The probability of a task to be considered failed – Eq. (6) – when k-out-of-n job failures are tolerated is:

$$p_i^{\texttt{J}(k\text{-out-of-}n)} = 1 - \prod_{s=k+1}^{n} (1 - p_i^{\texttt{J}})^{\frac{n!}{(n-s)!s!}} \quad (8)$$

This probability has been derived similarly to the proof of Eq. (4), by considering that $\frac{n!}{(n-s)!s!}$ is the number of failure situations (from the formula of combination of events). To verify the failure requirement, we can use the obtained probability $p_i^{\texttt{J}(k\text{-out-of-}n)}$, instead of $p_i^{\texttt{J}}$, in the condition of Definition 2.

IV. Scheduling Analysis and Dropping Relations

Our approach does not limit the choice of the scheduler and the related scheduling analysis. The scheduling algorithm does not necessarily need to be fault-aware. The task set Δ can, in fact, be considered as a normal task set in which tasks $\tau_{i(k)}$, with $k > 0$, are sporadic tasks. Consequently, any algorithm, including EDF, can be used for scheduling. Independently from the choice of the scheduling test, even if the original task set Γ is schedulable, the task set Δ may result non-schedulable due to the addition of re-execution tasks needed to meet the failure requirements. However, we can exploit additional information to improve the analysis, for instance, the fact that $\tau_{i(k)}$ can be activated only if $\tau_{i(k-1)}$ was activated and this event is unlikely to occur. Hence, to improve the schedulability, we propose the following MC generalization.

A. Dropping Relations

Definition 3 (Dropping Relation (DR))**.** The *Dropping Relation (DR)* of a task $\tau_{i(k)} \in \Delta$, for $k > 0$, is defined as:

$$\tau_{i(k)} \overset{d}{\rightsquigarrow} \{\tau_{a(b)}, \tau_{c(d)}, ...\} \quad \text{s.t.} \quad \tau_{a(b)} \in \Delta$$

A task may or may not have a defined DR. □

This relation states which tasks to drop ($\{\tau_{a(b)}, \tau_{c(d)}, ...\}$) when the re-execution task $\tau_{i(k)}$ is activated. The time instant at which the effects of the DR ceases after a fault occurrence depends on the adopted scheduling algorithm (similarly to the problem of when switch back a traditional MC system to a lower criticality mode). As a first approximation and if the scheduler is work-conservative, we can consider that the DR ceases its effect at the end of the hyperperiod or at the first idle time. This choice does not impact the failure requirements because, for the dangerous failure model, only the first failure is relevant.

Example 3. Let $\Delta = \{\tau_{1(0)}, \tau_{1(1)}, \tau_{2(0)}, \tau_{2(1)}, \tau_{3(0)}, \tau_{4(0)}\}$. The following DR for task $\tau_{1(1)}$:

$$\tau_{1(1)} \overset{d}{\rightsquigarrow} \{\tau_{2(1)}, \tau_{3(0)}\}$$

means that if the first re-execution task of τ_1 (i.e., $\tau_{1(1)}$) is triggered (because of the fault of $\tau_{1(0)}$), then the re-execution task $\tau_{2(1)}$ is dropped, or never activated, and the task $\tau_{3(0)}$ is dropped, or never activated. □

A DR $\tau_{i(k)} \overset{d}{\rightsquigarrow} \{\tau_{a(b)}, \tau_{c(d)}, ...\}$ is said to be correct if the following condition holds: $\nexists j : \tau_{i(j)} \in \{\tau_{a(b)}, \tau_{c(d)}, ...\}$. This condition requires that a DR does not exist from a task to itself: it has no sense that a task causes the drop of a re-execution task of itself.

B. Effects on the Scheduling Analysis

The effects of the DRs on the schedulability analysis depend on the analysis itself. We can expect that the introduction of DRs improves the task set schedulability. For example, in single-core scheduling, DRs reduce the system utilization after the occurrence of a fault, increasing the schedulability. If the scheduling test is based on utilization, the utilization improvement of a DR is the sum of utilization improvement for each dropped task in the DR.

It is important to remark that all the probabilities considered in this work do not affect the correctness of the scheduling analysis. They are used to model the fault rate, verify the failure requirements and configure the fault tolerance algorithms. The schedulability test runs on statically computed WCET, making the proposed approach in any case safe for the timing correctness standpoint.

C. Effects on the Failure Analysis

The introduction of DRs adds a dependency among the tasks and, in particular, on their failure rate. The *iid* assumption makes the computation of the additional failure rate of a job of $\tau_{i(k)}$ due to a DR possible:

$$p^{\mathtt{F'}}_{i(k)} = 1 - (1 - p^{\mathtt{F}}_i) \prod_{(j,m)\in D(i,k)} \left(1 - p^{\mathtt{F}}_j\right) \tag{9}$$

where $D(i,k) = \{(a,b) : \exists A \tau_{a(b)} \overset{d}{\rightsquigarrow} A \wedge \tau_{i(k)} \in A\}$. This formula comes from the union of the independent events of observing the natural failure of a job of $\tau_{i(k)}$ *or* observing the failure of a job of $\tau_{i(k)}$ due to the activation of $\tau_{j(m)}$ according to the defined DRs.

For each task, once computed the failure rate increment, we must re-verify the condition of Definition 2, i.e., that the task dependence is controlled and task failure rate remains within limits. In particular, Eq. (6) becomes $p^{\mathtt{J'}}_i = \prod_{k=1}^{N^{\mathtt{re-exec}}_i} p^{\mathtt{F'}}_{i(k)}$ and the condition of Definition 2 must still hold: $p^{\mathtt{J'}}_i \leq p^{\mathtt{req}}_i$.

V. Computation of Dropping Relations

To build the DRs, we propose two approaches: one inspired by traditional MC categorization of tasks, and the other a novel tree-exploration based approach.

A. Mapping to the Traditional Mixed-Criticality Setup

In the traditional MC setup, a criticality level $L_i \in \{1, ..., n_L\}$ is assigned to each task (assuming smaller values of L_i indicates lower criticality). In the original MC paper by Vestal [6], this value is derived from the safety requirements of the task: Instead of having one single WCET value, the tasks have several WCETs depending on their own criticality level. This is usually expressed by making C_i a vector $\bar{C}_i$ of dimension L_i, with the condition that $\bar{C}_i(j) \geq \bar{C}_i(k)$ for any $j > k$. Most of the papers in the literature based on the Vestal model uses the *mode switch* concept: when a task with criticality level l overruns its $\bar{C}_i(k)$, with $k < l$, the system is said to switch to the higher criticality mode $k+1$. When a system is in criticality mode k, then all tasks of lower criticalities ($L_i < k$) are either dropped or scheduled with a best-effort policy.

We map the traditional MC to our approach as follows:

- We order the tasks according to their $N^{\mathtt{re-exec}}_i$ and we assign a criticality level for each value, i.e., $L_i = N^{\mathtt{re-exec}}_i + 1$.
- A task with L_i criticality level is considered to tolerate a minimum of $L_i - 1$ faults: Each criticality level of a task τ_i represents the re-execution of the previous criticality level, guaranteeing a total of $L_i - 1$ re-execution of the original workload of τ_i. The WCET vector $\bar{C}_i(j)$ is interpreted as the following: $\bar{C}_i(1) = C_i$, $\bar{C}_i(2) = C_i + C_i(1)$, etc. The re-execution tasks have the same WCET of the original task, thus: $\bar{C}_i(j) = j \cdot C_i$.
- We then create the following DRs according to the aforementioned MC system mode behaviour:

$$\tau_{i(j)} \overset{d}{\rightsquigarrow} \{\tau_{a(b)}, ...\} \forall a, b : L_a < j \leq L_i. \tag{10}$$

Informally, we drop any task of lower criticality than the current system mode criticality.

Accordingly, Eq. (9) is modified as follow:

$$p^{\mathtt{F'}}_{i(k)} = 1 - (1 - p^{\mathtt{F}}_i) \prod_{j:L_j<L_i} \left(1 - p^{\mathtt{F}}_j\right)^{N^{\mathtt{re-exec}}_j}. \tag{11}$$

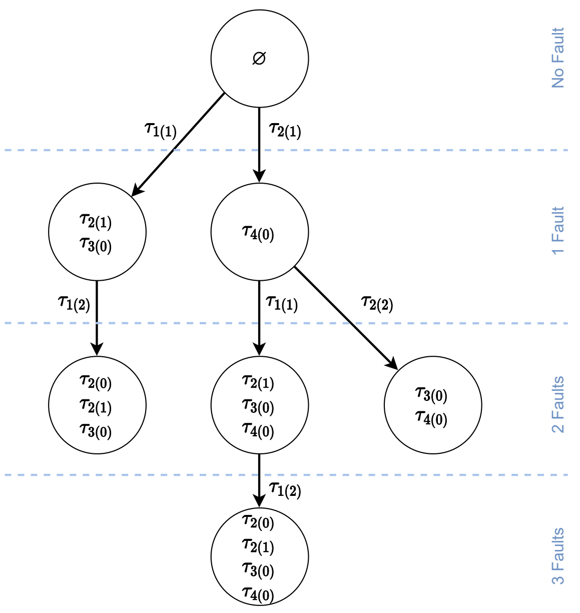
Differently from the traditional use of MC, in our case, a mode switch happens after the occurrence of a hardware real fault, thus the probability of the mode switch and the dropping of jobs is known. The MC approach has, however, the drawback that it drops more jobs than necessary, as later showed in the experimental results. This may impact the feasibility of lower criticality tasks to meet their failure requirement.

B. A Tree Exploration-Based Solution

To construct a more efficient solution than MC-mapping, we introduce a tree-based solution to the DRs creation problem. As the MC method, the tree exploration takes place offline and the created DRs are then enforced online by the scheduler.

The set of DRs can be represented with a tree T, e.g., the one depicted in Figure 2. This representation helps us in describing the DRs generation algorithm based on a coupled scheduling and failure analysis. Each node represents the set of the dropped tasks after the detection, at run-time, of faults. This concept is a generalization of the traditional *system mode* of MC systems. The root node corresponds to the normal execution when no faults have occurred, and all the tasks in Γ are scheduled. When a fault is detected, the system switches to another node following the edge corresponding to the newly activated task (i.e., the re-execution task of the failed task). The content of this node is the set of dropped tasks according to the triggered DR. If another re-execution task must be activated – i.e., another fault occurs – the system switches to a new node with a larger number of dropped tasks.

Formal definition. Let T be a *tree* composed of $V = \{v_0, v_1, v_2, ..., v_n\}$ nodes and $E = \{e_1, e_2, ..., e_n\}$ edges, where each edge is defined as a pair of nodes $e_k = (v_i, v_j)$. A *path* of a tree is defined as a sequence of connected and distinct nodes and edges. In this paper, we identify with $P(v_x) = (e_a, e_b, ..., e_m) = ((v_0, v_a), (v_b, v_c), ..., (v_w, v_x))$ the path having the first node as the root node (v_0) and the last node in the edge e_m is v_x. From the definition of tree and path, it is possible to state that $P(v_x)$ is unique. To each node is assigned a label $\delta(v_i) \subset \Delta$ and to each edge is assigned a label $\delta(e_i) \in \Delta$. The label of the root node v_0 is always



∅
$\tau_{1(1)}$
$\tau_{2(1)}$
No Fault
$\tau_{2(1)}$ $\tau_{3(0)}$
$\tau_{4(0)}$
1 Fault
$\tau_{1(2)}$
$\tau_{1(1)}$
$\tau_{2(2)}$
$\tau_{2(0)}$ $\tau_{2(1)}$ $\tau_{3(0)}$
$\tau_{2(1)}$ $\tau_{3(0)}$ $\tau_{4(0)}$
$\tau_{3(0)}$ $\tau_{4(0)}$
2 Faults
$\tau_{1(2)}$
$\tau_{2(0)}$ $\tau_{2(1)}$ $\tau_{3(0)}$ $\tau_{4(0)}$
3 Faults

Fig. 2. An example of DR tree. When no fault occurred, all the tasks are admitted to scheduling (node ∅). If, for instance, the task τ_1 fails, it activates the task $\tau_{1(1)}$ and the tasks $\tau_{2(1)}$ and $\tau_{3(0)}$ are consequently dropped.

$\delta(v_0) = \emptyset$. The label of each node is a superset of the parent node: $\delta(v_j) \supseteq \delta(v_i), (v_i, v_j) \in E$. An edge $e_k = (v_i, v_j)$ with label $\delta(e_i) = \tau_{a(b)}$ exists in the tree if the following conditions hold:

1) the DR $\tau_{a(b)} \overset{d}{\rightsquigarrow} \tau_{a'(b')}, \tau_{a''(b'')}, ...$ exists;
2) $\tau_{a(b)} \notin \delta(v_i)$ (the task has not been already dropped);
3) $\tau_{a(b)} \notin \{\delta(e_i) : e_i \in P(v_i)\}$ (the re-execution task has not been already activated).

The node v_j is labelled with $\delta(v_j) = \delta(v_i) \cup \{\tau_{a'(b')}\} \cup \{\tau_{a''(b'')}\} \cup \cdots$. If the DR does not exist but the second and third conditions hold, the new edge is added with $\delta(v_j) = \delta(v_i)$ (in this case a fault does not trigger a DR).

Example 4. The tree in Figure 2 corresponds to the following DRs:
$\tau_{1(1)} \overset{d}{\rightsquigarrow} \tau_{2(1)}, \tau_{3(0)}$; $\tau_{1(2)} \overset{d}{\rightsquigarrow} \tau_{2(1)}, \tau_{3(0)}, \tau_{2(0)}$; $\tau_{2(1)} \overset{d}{\rightsquigarrow} \tau_{4(0)}$.
The system *starts* in the root node configuration, i.e., no tasks are dropped. Then, if a fault is detected in $\tau_{1(0)}$ or $\tau_{2(0)}$, the system *switches* to one of the node at the first level below the root. For instance, if a fault is detected in $\tau_{1(0)}$, $\tau_{1(1)}$ is activated and $\tau_{1(1)}$ and $\tau_{3(0)}$ dropped. □

Schedulability. The tree representation is convenient because it is easy to prove schedulability by exploiting pre-existing MC schedulability tests and, in particular, EDF-VD(k) [20]. This is possible thanks to the following lemma:

Lemma 1. The task set Δ is schedulable according to the dropping mechanism imposed by the DRs if both the following conditions are satisfied:

1) $\forall v_l$, such that v_l is a leaf node (i.e., $\nexists v_i : (v_l, v_i) \in E$), $P(v_l)$ is schedulable according to the MC scheduling test for EDF-VD(k).
2) $\exists x$ (scaling parameter of the schedulability test) identical for all $P(v_l)$ satisfying [20, Eq. (6)] and [20, Eq. (7)] $\forall P(v_l)$.

Intuitive explanation. Each root-to-leaf path represents a chain of events that may occur at run-time. According to the tree definition, no other possible chains of events exist other than the set of all paths. Initially, the system is in the root node and all tasks are admitted to be scheduled according to the virtual deadlines computed with the scaling parameter x. Then, if a fault occurs, there is a transition to a node of the next level and, consequently, some tasks are dropped. The failed task requires its re-execution, i.e., its WCET is incremented. This is the same task model of the MC model, thus the schedulability of the single path can be checked with the EDF-VD(k) test by setting a criticality level for each node. □

Proof. The following proof is split in three parts: (1) the conversion to the EDF-VD model; (2) schedulability of the single path; and (3) schedulability of the whole tree.
(1) We convert each path $P(v_x) = (e_a, e_b, ..., e_m)$ to a MC problem as follows. To each task τ_i we assign the following criticality level:

$$L_i = 1 + \sum_{j=1}^{m} \mathbf{1}(\tau_{i(0)} \notin \delta(v_k) : (\cdot, v_k) \in e_j) \tag{12}$$

and the following WCET vector:

$$\bar{C}_i(k) = \begin{cases} C_i & \text{if } k = 1 \\ \bar{C}_i(k-1) + C_i \mathbf{1}(\exists j : \tau_{i(j)} = \delta(e_{k-1})) & \text{if } 1 < k \le L_i \end{cases} \tag{13}$$

where $\mathbf{1}(\cdot)$ is the indicator function[4]. The relation with the model notation of [20, §2] is: $\chi_i := L_i$, $c_i(k) := \bar{C}_i(k)$. The MC model generated from the path complies with the model conditions: $\bar{C}_i(k) \ge \bar{C}_i(k-1)$, and $k \le L_i$. The MC model requires a task $\tau_{i(k)}$ to be scheduled if the current system mode is lower of equal to the criticality of the task, otherwise it may be dropped. Eq. (12) guarantees this behavior: a task is activable if not present in $\delta(v_k)$, and this is occurs until the node in which it is dropped.
(2) Because each path is an MC problem, we apply the EDF-VD(k) schedulability test to verify the schedulability of each path. This sufficient test consists of checking whether a scaling parameter x exists which satisfies [20, Eq. (6)] and [20, Eq. (7)].
(3) Because any of the paths may happen at run-time, all the paths must be schedulable. However, the tasks are scheduled according to the virtual deadlines of EDF-VD(k) computed with the scaling parameter x. A non-clairvoyant does not know which path is selected at run-time, and, consequently, which x to use for computing the virtual deadlines guaranteeing the correctness of the specific MC problem instance. Therefore a common value x must exist for all the possible paths: When a fault occurs, the specific MC instance is considered (given by the new node of the path), and EDF-VD(k) correctly schedules

[4]$\mathbf{1}(\cdot) = \{$ 1 if the condition · is true, 0 otherwise $\}$.

Algorithm 1 Algorithm for the DR tree building.

```
1: function BuildNode($P$, $\tau_{i(j)}$, $A$)
2:     (newN, newP) ← create_node($P$, $\tau_{i(j)}$)
3:     newA ← $A - \{\tau_{i(j)}\}$
4:     if schedulable(newP) then
5:         return NextLevel(newP, newA)
6:     else
7:         FD ← SearchCompliantDroppings(newP, newA)
8:         for all fd ∈ FD do
9:             set_node_content(newN, $fd$)
10:            if schedulable(newP) then
11:                if NextLevel(newP, newA−{fd}) then
12:                    return SUCCESS
13:        return FAILED
14: function NextLevel($P$, $A$)
15:     if $A == \varnothing$ then
16:         return SUCCESS
17:     for all $\tau_{i(j)} \in$ A do
18:         if $\tau_{i(j-1)} \notin$ A then
19:             if not BuildNode($P$, $\tau_{i(j)}$, $A$) then
20:                 return FAILED
21:     return SUCCESS
```

it because the instance has been proved as schedulable and a value x satisfying [20, Eq. (6)] and [20, Eq. (7)] was used. □

Offline Algorithm. We use Algorithm 1 to build the tree. The algorithm starts from the root node (which has an empty label by definition) by calling `NextLevel(∅,Δ \ Γ)` and proceeding with a recursive strategy: `NextLevel` calls `BuildNode` (Line 19) for each possible activable re-execution task. The condition at Line 18 is necessary because the task $\tau_{i(j)}$ is not activable if A still contains the task $\tau_{i(j-1)}$ (it will become activable in the subsequent recursion). This loop verifies that every possible path generating from the current node is both compliant and schedulable (in this case `BuildNode` would always return SUCCESS). `BuildNode` adds a new node and the related edge (Line 2) verifying if the new path is schedulable or not (condition at Line 4). If it is, no task dropping is necessary and we can proceed to the next level (Line 5), otherwise, we need to find a compliant set of tasks to drop. This is done by function `SearchCompliantDroppings` (Line 7), which explores the power set of the activable tasks to find the compliant dropping sets. Then, `BuildNode` recursively verifies that the next levels are schedulable and compliant, and accordingly returns SUCCESS or FAILED (Lines 13–12).

Complexity. To compute the time complexity of the algorithm we proceed as follows. The tree has the maximum number of nodes when all the paths are schedulable without any drop. In this case, the number of nodes in the tree is $O(n^m)$ where n is the cardinality of Γ and m is the cardinality of Δ. For each node, we run the schedulability test (Line 4). The EDF-VD(k) test for multi-critical levels has $O(n_L^2+n)$ complexity [20] and, by assuming $n >> n_L$, it becomes $O(n)$. If the tree is complete, the complexity is $O(n \cdot n^m)$. Otherwise, the condition at Line 4 is false at least one time. The worst-case is when the condition of Line 4 is always false. Starting from the root, we have n edges and n new nodes. For each node, we find the compliant droppings. The set, in the worst-case, has a cardinality of 2^m (power set of Δ), leading to a total worst-case complexity of $O(n^m 2^m)$. This complexity is exponential, however, we can apply the following considerations to reduce the number of explored nodes:

- The complexity does not consider schedulability. Many tasks are likely required to be dropped in each node, considerably reducing the number of subsequent branches;
- The tree height can theoretically be equal the number of re-execution tasks, but can be bounded from an engineering perspective. For example, if we consider a constant fault rate of $p_i^F = 10^{-4}/h$, after level 4 the event rate is lower than $p_i^F = 10^{-16}/h$. Being the rate lower than the DAL A required failure rate, we can prune the tree at this level. Consequently, being $l << m$ the level at which we decided to prune, the complexity becomes $O(n^l 2^m)$.
- It is possible to trade the number of sets of possible dropping tasks of each node with the solution optimality. Limiting the number of tested dropping tasks reduces the execution time of the required analysis with the disadvantage of scarifying a bit of schedulability ratio.

The experiments of the next section resulted certainly affordable in a normal workstation: The analysis's execution time of a task set of 50 tasks ranges from a few milliseconds to some minutes, which is acceptable for an offline analysis.

Online Execution. Having the pre-computed DRs, the scheduler simply applies the relative DR when a fault occurs and behaves like a traditional MC scheduler. Therefore, no significant additional overhead is expected at run-time compared to normal mixed-criticality systems. The complexity is in the worst-case $O(m^2)$ (m DRs to check, m tasks in each DR).

VI. Experimental findings

The methodology proposed in the previous sections (both the MC method of Section V-A and the tree-based method of Section V-B) improves the schedulability by introducing the DR mechanism. The tree-based method also implicitly ensures that the resulting scheduling is compliant with the failure requirements. In this experimental campaign, we run simulations to quantify these improvements, show the limits of traditional approaches, and highlight the benefits of the tree-based method. We limit the following experiments to a single-core machine, in order to avoid adding new assumptions and because of the existence of optimal scheduling algorithms for single-cores. This limitation actually makes the problem more challenging because of the need to drop a larger number of tasks to make space for re-execution tasks. The source code of the experiments and the resulting datasets are open access and available online[5,6].

A. Simulation Setup

Each simulation scenario is defined by the tuple (n, U, λ), where $n \in [1; 50] \subseteq \mathbb{N}$ is the number of tasks, $U \in [0; 1] \subseteq \mathbb{Q}$ is

[5]Dataset: https://doi.org/10.5281/zenodo.6202760

[6]Source code: https://github.com/HEAPLab/mc-fault-simulator

the total utilization of the system without considering the extra utilization of the re-execution tasks, and λ is the fault rate per-hour. For each the tuple (n, U, λ), 1 000 different task sets (100 for the tree) are generated with the following randomization approach:

- The utilization of each task and, consequently, the ratio $r_i = \frac{C_i}{T_i}$ is selected by the UUnifast algorithm [21].
- The period T_i of each task is randomly chosen in the interval $[50; 1000] \subseteq \mathbb{N}$. The C_i is then accordingly computed with the ratio r_i.
- The failure requirement per-hour of each task is randomly chosen (according to an uniform distribution) among the values provided by the DO-178C standard for the criticality levels: $\{10^{-3}, 10^{-5}, 10^{-7}, 10^{-9}\}$/h.

The task set Δ, that includes the re-execution tasks, is computed thanks to the equations of Section III, in particular Eq. (7). We explored three different values for the fault rate: $\lambda \in \{10^{-5}, 10^{-4}, 10^{-3}\}$/h.

B. Experiments

The first set of experiments consists of measuring the ability to schedule the task sets when all the tasks in Δ are admitted to run, including the re-execution ones. This scenario is a traditional real-time scheduling problem, and we, therefore, selected the optimal EDF algorithm. Because all the re-execution tasks are admitted, all the schedulable task sets (i.e., having $U \leq 1$) are also compliant, i.e., they also satisfy the failure requirements – according to Definition 2.

The second set of experiments is to use the traditional MC approach to build the DRs, as explained in Section V-A. Each task requires a number of re-execution tasks in the range $[0; 2]$ in all the considered scenarios. Consequently, the number of considered criticality levels is three: $L_i \in [1; 3]$. In this case, the introduction of DRs may violate the failure requirements.

Finally, the novel tree exploration of Section V-B is tested. In this case, the exploration already takes into account the compliance of the solution: the number of schedulable task sets corresponds to the number of compliant task sets.

In order to make a fair comparison and to guarantee the reproducibility of the experiments, we set the pseudo-random number generator so that all the random task sets and their task properties are identically generated in all the three scenarios. We used the uniform distribution for all the random samplings.

C. Results & Discussion

The result of the EDF schedulability test when all the re-execution tasks are admitted is shown in Figure 3. Since all task sets are compliant by construction (because all the re-execution tasks are admitted), Figure 3 shows the percentage of tasks that are both schedulable and compliant with the failure requirement. Most all of the task sets remain schedulable when the original utilization of Γ is lower than 0.3–0.4. The presence of re-execution tasks compromises, as obvious, the schedulability bound of $U \leq 1$ of EDF. Because of the choice of the uniform distribution among the failure requirements,

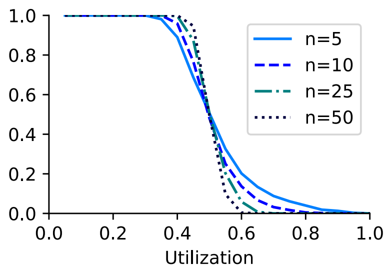


Fig. 3. EDF Schedulability (and compliance) of Δ task sets. The y-axis is the percentage of the task sets schedulable for a total of 1000 simulations. The x-axis refers to the original utilization U_Γ, i.e., without the re-execution tasks, not the utilization of Δ. This plot is not affected by the λ value.

the utilization of Δ is on average $U_\Delta = 2U_\Gamma$ and, in fact, at $U_\Gamma = 0.5$ only half of the task sets are schedulable.

In the MC case (presented in Section V-A), the EDF-VD test for multiple criticality levels [20] has been used. The schedulability improves, as visible in the left column of Figure 4. The improvement is of about +5% thanks to the DRs introduced by the MC approach. Unfortunately, there is no free lunch: introducing DRs inevitably hinders the compliance with the failure requirements. For each task set, we computed the failure rate of the tasks according to the DRs, and we compared it to the required maximum failure rate. The result is depicted in the other two columns of Figure 4. When the fault rate is $\lambda = 10^{-5}$/h (first row) the MC approach guarantees the compliance with the failure requirements for all the task sets. This result deteriorates by increasing the fault rate. For $\lambda = 10^{-4}$/h (second row) the result is still better than the EDF only for $n \leq 10$. As the number of tasks increases, the number of dropped tasks increases as well, violating the failure requirement. This is more evident for the last value of the failure rate, i.e., $\lambda = 10^{-3}$/h (third row). In the MC approach, several task sets are schedulable but not compliant with the failure requirements. In this case, a standard EDF is preferable over the MC approach.

Finally, the results for the tree algorithm of Section V-B are shown in Figure 5. In this case, the figures represent both schedulability and compliance, because they are intrinsically determined during the tree building. It is immediately clear that the number of task sets that are both schedulable and compliant is higher than EDF and MC cases. Moreover, the increasing number of tasks becomes advantageous and not a disadvantage as the MC case: with the tree model we can explore more possibilities and find several possible solutions.

To better compare the results, the percentage of task sets schedulable and compliant in the three cases are summarized in Table III. The effectiveness of the MC approach is quite limited, even considering only schedulability. The $\approx$ 5% improvement is small – probably due to the large differences between the WCETs $\bar{C}_i$ – and only for $\lambda = 10^{-5}$/h the MC approach results a better choice than standard EDF. Instead, the tree-based approach is the best in all the scenarios. Performing a joint scheduling and failure analysis allows us to control the failure rate. The tree-based approach is effective in

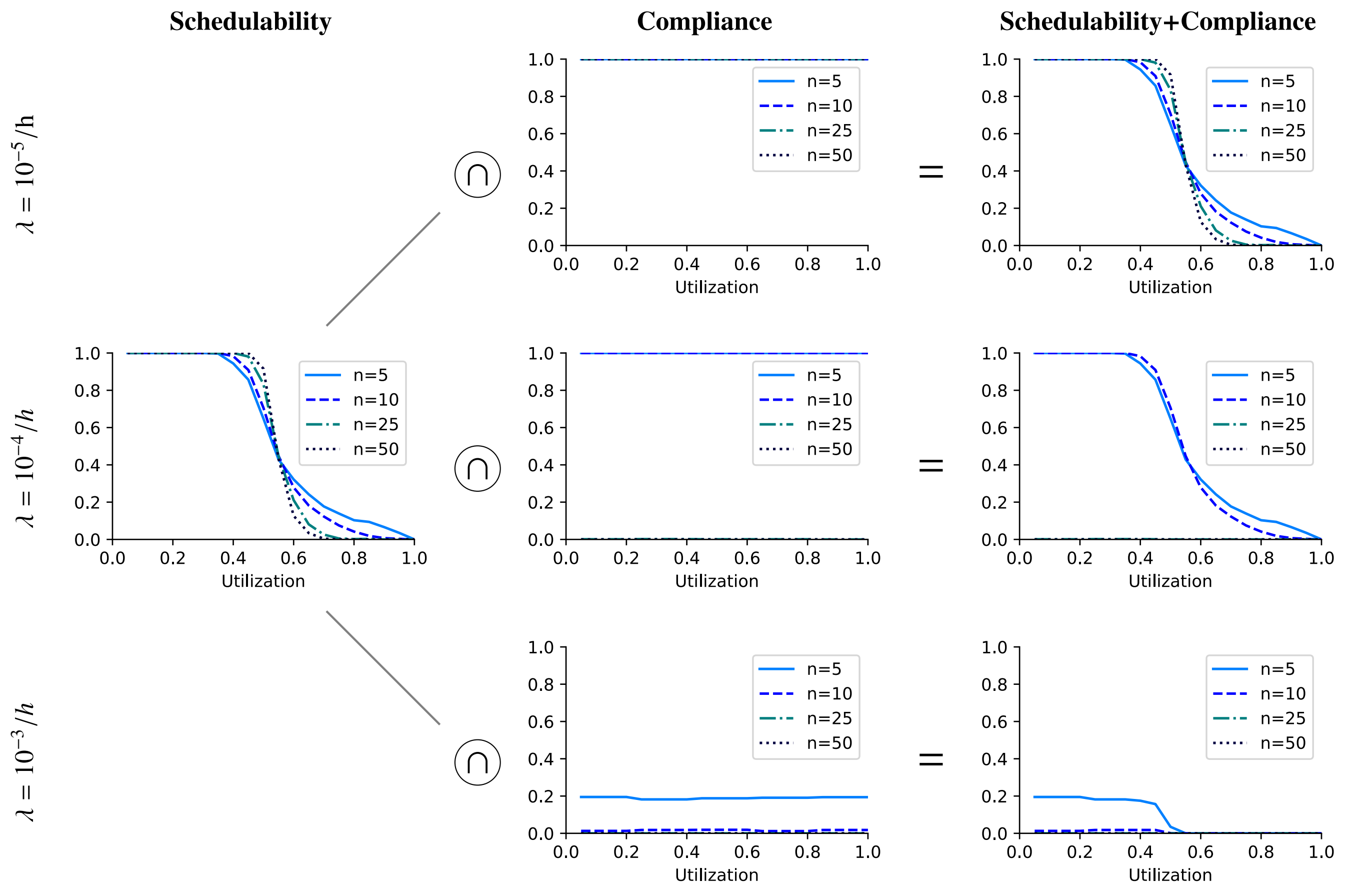


Fig. 4. Schedulability and compliance with failure requirements of the MC approach. The x-axis refers to the utilization of Γ (without re-execution tasks), while the y-axis is the percentage of task-sets Δ (with re-execution tasks) schedulable and/or compliant.

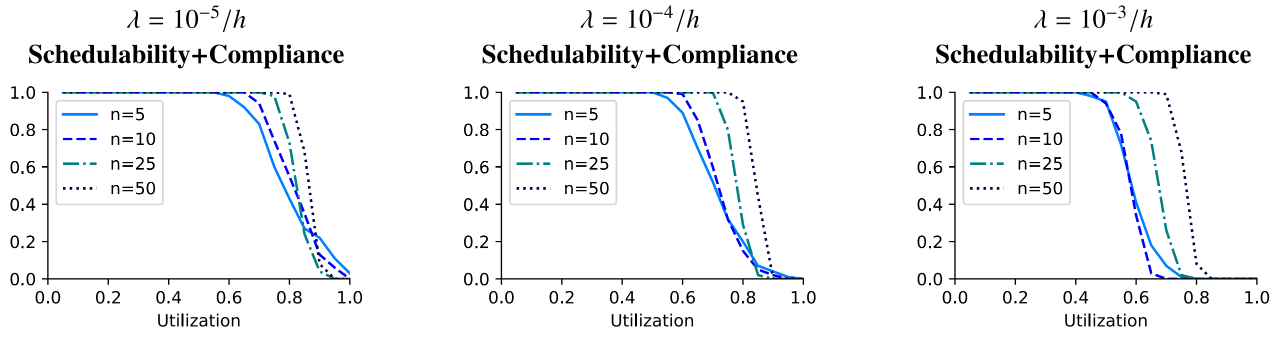


Fig. 5. Schedulability and compliance with failure requirements of the tree-based approach. The x-axis refers to the original utilization of Γ (without re-execution tasks), while the y-axis is the percentage of task-sets Δ (with re-execution tasks) both schedulable and compliant.

all the scenarios and is able to schedule a significant amount of extra task sets compared to the other approaches, while remaining compliant with the failure requirement.

VII. Related Work

The first study on the schedulability analysis that considered fault-recovery tasks has been proposed in 1994 [22]. From that point on, several articles proposed schedulability or response-time analysis with different fault tolerance mechanisms, including re-execution [13], [23], checkpoint/restore [24], [25], recovery blocks [26], [27], and replicas [28], [29]. However, few existing works considered the MC scenario. Huang et al. [30] and Pathan et al. [31] converted the fault tolerance problem into a dual-criticality problem, with re-execution to recover from faults. LO-criticality tasks are dropped or degraded when faults occur. Similarly, Lin et al. [32] proposed an EDF scheduling scheme that guarantees the execution of HI-criticality tasks and minimizes the dropping of LO-criticality tasks. Brüggen et al. [33] modeled a system with *normal* and *abnormal* modes. In normal mode, all the tasks

TABLE III
The overall percentage of schedualability and compliance with failure requirements of all the analyzed scenarios.

Case	λ	Schedulable	Compliant	Schedulable+Compliant
EDF	any	48.58%	100%	48.58%
MC	$1 \cdot 10^{-5}$	53.66%	100%	53.66%
	$1 \cdot 10^{-4}$	53.66%	50.02%	27.30%
	$1 \cdot 10^{-3}$	53.66%	5.15%	2.29%
Tree	$1 \cdot 10^{-5}$	-	-	79.88%
	$1 \cdot 10^{-4}$	-	-	74.00%
	$1 \cdot 10^{-3}$	-	-	62.66%

meet the deadlines, while, when a fault occurs, the abnormal mode guarantees the deadlines for HI-criticality tasks only, while LO-criticality tasks have a bounded worst-case tardiness. Choi et al. [34] proposed a worst-case analysis based on a similar re-execution approach on multi-core systems. Guo et al. [35] studied the schedulability of MC tasks with permitted failure probability, but the focus was only on timing faults (deadline misses). Other works exploited MC for replicas [36], [37], including multi-core setups [38].

All of the previously mentioned works take into account important yet limited aspects of fault tolerance in the MC context. None of them considered the fault probability as caused by both hardware faults and MC droppings due to the task dependency. As we quantified in Section VI, both probabilities are non-negligible, making the compliance with failure requirement of the MC approach very challenging in real applications.

VIII. Extension to Probabilistic DR

As a future work, we plan to extend our approach to manage probabilistic-DR defined as follows:

Definition 4 (Probabilistic Dropping Relation (PDR))**.** The PDR of a task $\tau_{i(k)} \in \Delta$, for $k > 0$, is defined as:

$$\tau_{i(k)} \overset{d}{\rightsquigarrow} A_1, A_2, ... \quad \text{s.t.} \quad A_l \in \mathcal{P}(\Delta), A_l \cap A_m = \varnothing \ \forall l \neq m$$

where $\mathcal{P}(\Delta)$ is the *power set* of Δ, i.e., all possible subsets of Δ. To each task $\tau_{a(b)}$ in $A_1, A_2, ...$ is assigned a probability $\alpha^{i(k)}_{a(b)}$. A task may or may not have a defined PDR. □

The probability $\alpha^{i(k)}_{a(b)}$ represents the probability that the task $\tau_{a(b)}$ is dropped if the task $\tau_{i(k)}$ needs to be re-executed. We represent this probability by writing it on the top of each task in A_i.

Example 5. Let $\Delta = \{\tau_{1(0)}, \tau_{1(1)}, \tau_{2(0)}, \tau_{2(1)}, \tau_{3(0)}, \tau_{4(0)}\}$. The following DR for task $\tau_{1(1)}$:

$$\tau_{1(1)} \overset{d}{\rightsquigarrow} \{\overset{1}{\tau_{2(1)}}\}, \{\overset{0.4}{\tau_{3(0)}}, \overset{0.6}{\tau_{4(0)}}\}$$

means that if the first re-execution task of τ_1 (i.e., $\tau_{1(1)}$) is triggered (because of the fault of $\tau_{1(0)}$), then the re-executed task $\tau_{2(1)}$ is dropped, or never activated, and the task $\tau_{3(0)}$ or $\tau_{4(0)}$ is dropped, or never activated, according to the respective probability of 0.4 and 0.6. □

A probabilistic scheduling analysis can directly exploit the presence of probabilities on the PDR terms. However, if the scheduling analysis is deterministic, we have to consider the worst-case improvement. Nevertheless, the presence of $\alpha^{i(k)}_{a(b)}$ is still advantageous in failure analysis, because the term p^{F}_j of Eq. (9) becomes $\alpha^{j(m)}_{i(k)} \cdot p^{\text{F}}_j$, improving the compliance with failure requirements.

IX. Conclusions

The use of software fault tolerance techniques in real-time systems is challenging due to the need to satisfy both timing and failure requirements, especially nowadays when chip miniaturization exacerbates the susceptibility to transient faults. We proposed a new model and scheduling strategies to guarantee both requirements. In particular, we derived a generalization of the MC framework, i.e., the Dropping Relations, followed by a tree-based algorithm to generate them. Unlike traditional MC, the violations of task independence introduced by the DRs are exactly computed, satisfying the failure requirements of safety-critical standards. The experimental results showed how our approach improves the schedulability by about $20 - 30\%$, while still guaranteeing the failure requirements.

Future research could investigate several aspects: new algorithms to explore the tree, the extension to the k-out-of-n case outlined of Section III-D, the probabilistic model of Section VIII, integrating the effects of power management in the analysis, the analysis of different types resources subject to faults (such as disks, power supply, etc.), and the introduction of permanent faults and fault bursts in the analysis, which may introduce additional overheads.

Acknowledgments

This article has received funding from the EU H2020 Research and Innovation Programme, project TEXTAROSSA (grant agreement No. 956831), and NSF grants CNS-1850851, PPoSS-2028481.

References

[1] X. Li, M. C. Huang, K. Shen, and L. Chu, "A realistic evaluation of memory hardware errors and software system susceptibility," in *USENIX Annual Technical Conference (USENIX ATC '10)*, 2010.

[2] A. Jacobs, G. Cieslewski, A. D. George, A. Gordon-Ross, and H. Lam, "Reconfigurable Fault Tolerance: A Comprehensive Framework for Reliable and Adaptive FPGA-Based Space Computing," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 5, no. 4, Dec. 2012.

[3] L. A. Tambara, P. Rech, E. Chielle, J. Tonfat, and F. L. Kastensmidt, "Analyzing the Impact of Radiation-Induced Failures in Programmable SoCs," *IEEE Transactions on Nuclear Science*, vol. 63, no. 4, pp. 2217–2224, 2016.

[4] IEEE, "IEEE Standard 610.12-1990 Glossary of Software Engineering Terminology (Reaffirmed 2002)," IEEE, New York, USA, Standard, 1990.

[5] O. Goloubeva, M. Rebaudengo, M. S. Reorda, and M. Violante, *Software-Implemented Hardware Fault Tolerance*, Springer, Ed. Morgan Kaufmann, 2006.

[6] S. Vestal, "Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance," *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*, pp. 239–243, 2007.

[7] K. Bletsas, M. Ali Awan, P. Souto, B. Åkesson, A. Burns, and E. Tovar, "Decoupling criticality and importance in mixed-criticality scheduling," in *6th International Workshop on Mixed Criticality Systems (WMC 2018)*, 2018, pp. 25–30.

[8] R. Ernst and M. Di Natale, “Mixed criticality systems—a history of misconceptions?” *IEEE Design Test*, vol. 33, no. 5, pp. 65–74, 2016.

[9] A. Bhuiyan, F. Reghenzani, W. Fornaciari, and Z. Guo, “Optimizing Energy in Non-Preemptive Mixed-Criticality Scheduling by Exploiting Probabilistic Information,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 11, pp. 3906–3917, 2020.

[10] F. Reghenzani, L. Santinelli, and W. Fornaciari, “Dealing with Uncertainty in pWCET Estimations,” *ACM Trans. Embed. Comput. Syst.*, vol. 19, no. 5, sep 2020, doi:10.1145/3396234.

[11] S. Law, I. Bate, and B. Lesage, “Industrial Application of a Partitioning Scheduler to Support Mixed Criticality Systems,” in *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), S. Quinton, Ed., vol. 133. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, pp. 8:1–8:22.

[12] I. Hwang, S. Kim, Y. Kim, and C. E. Seah, “A Survey of Fault Detection, Isolation, and Reconfiguration Methods,” *IEEE Transactions on Control Systems Technology*, vol. 18, no. 3, pp. 636–653, 2010.

[13] F. A. T. Abad, R. Mancuso, S. Bak, O. Dantsker, and M. Caccamo, “Reset-based recovery for real-time cyber-physical systems with temporal safety constraints,” *IEEE International Conference on Emerging Technologies and Factory Automation, ETFA*, vol. 2016-Novem, pp. 1–8, 2016.

[14] J. Song, G. Bloom, and G. Parmer, “SuperGlue: IDL-Based, System-Level Fault Tolerance for Embedded Systems,” in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2016, pp. 227–238.

[15] F. Many and D. Doose, “Scheduling analysis under fault bursts,” *17th IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 113–122, 2011.

[16] S. K. Jain and V. D. Agrawal, “Statistical fault analysis,” *IEEE Design & Test of Computers*, vol. 2, no. 1, pp. 38–44, 1985.

[17] M. Rausand, *Reliability of Safety-Critical Systems*. John Wiley & Sons, Ltd, 2014.

[18] G. Bernat, A. Burns, and A. Liamosi, “Weakly hard real-time systems,” *IEEE Transactions on Computers*, vol. 50, no. 4, pp. 308–321, 2001.

[19] J. Zhou, X. S. Hu, Y. Ma, J. Sun, T. Wei, and S. Hu, “Improving availability of multicore real-time systems suffering both permanent and transient faults,” *IEEE Transactions on Computers*, vol. 68, no. 12, pp. 1785–1801, 2019.

[20] S. Baruah, V. Bonifaci, G. D’angelo, H. Li, A. Marchetti-Spaccamela, S. Van Der Ster, and L. Stougie, “Preemptive Uniprocessor Scheduling of Mixed-Criticality Sporadic Task Systems,” *J. ACM*, vol. 62, no. 2, May 2015.

[21] E. Bini and G. C. Buttazzo, “Measuring the performance of schedulability tests,” *Real-Time Systems*, vol. 30, no. 1, pp. 129–154, 2005.

[22] M. Pandya and M. Malek, “Minimum achievable utilization for fault-tolerant processing of periodic tasks,” *IEEE Transactions on Computers*, vol. 47, no. 10, pp. 1102–1112, published 1998, original 1994.

[23] A. Burns and R. Davis, “Feasibility Anlaysis of Fault-Tolerant Real-Time Task Sets,” in *Proceedings of the Eighth Euromicro Workshop on Real-Time Systems*. IEEE, 1996, pp. 29–33.

[24] A. Burns, S. Punnekkat, L. Strigini, and D. R. Wright, “Probabilistic scheduling guarantees for fault-tolerant real-time systems,” *Dependable Computing for Critical Applications 7*, pp. 361–378, 1999.

[25] P. Axer, M. Sebastian, and R. Ernst, “Reliability analysis for MPSoCs with mixed-critical, hard real-time constraints,” in *2011 Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2011, pp. 149–158.

[26] G. de A Lima and A. Burns, “An optimal fixed-priority assignment algorithm for supporting fault-tolerant hard real-time systems,” *IEEE Transactions on Computers*, vol. 52, no. 10, pp. 1332–1346, 2003.

[27] B. Zhao, H. Aydin, and D. Zhu, “Shared recovery for energy efficiency and reliability enhancements in real-time applications with precedence constraints,” *ACM Transactions on Design Automation of Electronic Systems*, vol. 18, no. 2, 2013.

[28] S. Ghosh, R. Melhem, and D. Mosse, “Fault-tolerance through scheduling of aperiodic tasks in hard real-time multiprocessor systems,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 8, no. 3, pp. 272–284, 1997.

[29] M. Cui, L. Mo, A. Kritikakou, and E. Casseau, “Energy-aware partial-duplication task mapping under real-time and reliability constraints,” in *Embedded Computer Systems: Architectures, Modeling, and Simulation*, A. Orailoglu, M. Jung, and M. Reichenbach, Eds. Cham: Springer International Publishing, 2020, pp. 213–227.

[30] P. Huang, H. Yang, and L. Thiele, “On the scheduling of fault-tolerant mixed-criticality systems,” *Design Automation Conference (DAC)*, pp. 1–6, 2014.

[31] R. M. Pathan, “Fault-tolerant and real-time scheduling for mixed-criticality systems,” *Real-Time Systems*, vol. 50, no. 4, pp. 509–547, Jul 2014.

[32] J. D. Lin, A. M. K. Cheng, D. Steel, and M. Y. Wu, “Scheduling Mixed-Criticality Real-Time Tasks with Fault Tolerance,” *Proc. 2nd Workshop on Mixed Criticality Systems (WMC), RTSS*, pp. 39–44, 2014.

[33] G. von der Brüggen, K.-H. Chen, W.-H. Huang, and J.-J. Chen, “Systems with Dynamic Real-Time Guarantees in Uncertain and Faulty Execution Environments,” in *2016 IEEE Real-Time Systems Symposium (RTSS)*, 2016, pp. 303–314.

[34] J. Choi, H. Yang, and S. Ha, “Optimization of Fault-Tolerant Mixed-Criticality Multi-Core Systems with Enhanced WCRT Analysis,” *ACM Trans. Des. Autom. Electron. Syst.*, vol. 24, no. 1, Dec. 2018.

[35] Z. Guo, L. Santinelli, and K. Yang, “EDF Schedulability Analysis on Mixed-Criticality Systems with Permitted Failure Probability,” in *2015 IEEE 21st International Conference on Embedded and Real-Time Computing Systems and Applications*, 2015, pp. 187–196.

[36] J. Zhou, M. Yin, Z. Li, K. Cao, J. Yan, T. Wei, M. Chen, and X. Fu, “Fault-tolerant task scheduling for mixed-criticality real-time systems,” *Journal of Circuits, Systems and Computers*, vol. 26, no. 1, pp. 1–17, 2017.

[37] S.-h. Kang, H. Yang, S. Kim, I. Bacivarov, S. Ha, and L. Thiele, “Static mapping of mixed-critical applications for fault-tolerant MPSoCs,” in *Proceedings of the 51st Annual Design Automation Conference*, ser. DAC ’14. New York, NY, USA: Association for Computing Machinery, 2014, p. 1–6.

[38] S. Safari, M. Ansari, G. Ershadi, and S. Hessabi, “On the scheduling of energy-aware fault-tolerant mixed-criticality multicore systems with service guarantee exploration,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 10, pp. 2338–2354, 2019.