Achieving System-level Fault-tolerance with Controlled Resets

Fardin Abdi, Renato Mancuso, Rohan Tabish, Marco Caccamo Department of Computer Science
University of Illinois at Urbana-Champaign
{abditag2, rmancus2, rtabish, mcaccamo}@illinois.edu

Abstract

Embedded systems in safety-critical environments are continuously required to deliver more performance and functionality, leading to increased complexity and connectivity. Despite the fast growing complexity, guaranteeing safety is of the utmost importance. Nonetheless, platform-wide software verification is often expensive. Therefore, design methods that enable utilization of components such as real-time operating systems (RTOS), without requiring their correctness to guarantee safety, is necessary.

In this paper, we propose a design approach to deploy safe-by-design embedded systems. To attain this goal, we rely on a small core of verified software to detect faults in applications and RTOS and recover from them while ensuring that timing constraints of safety-critical tasks are always satisfied. Faults are detected by monitoring the state of the physical plant and application timing. Fault-recovery is achieved via full platform restart and software reload, enabled by the short restart time of embedded systems. Schedulability analysis is used to ensure that the timing constraints of critical plant control tasks are always satisfied in spite of faults and consequent restarts. We derive schedulability results for four restart-tolerant task models. We use a simulator to evaluate and compare the performance of the considered scheduling models. Finally, we implement a controller for a 3 degree of freedom (3DOF) helicopter using one of such models on real hardware and demonstrate that the system remains safe despite faults in the RTOS and applications.

I. Introduction

Embedded controllers with smart capabilities are being increasingly used to implement safety-critical cyber-physical systems (SC-CPS). In fact, modern medical devices, avionic and automotive systems, to name a few, are required to deliver increasingly high performance without trading off in robustness and assurance. Unfortunately, satisfying the increasing demand for smart capabilities and high performance means deploying increasingly complex systems. Even seemingly simple embedded control systems often contain a multitasking real-time kernel, support networking, utilize open source libraries [1], and a number of specialized hardware components (GPUs, DSPs, DMAs, *etc.*). As systems increase in complexity, however, the cost of formally verifying their correctness can easily explode.

Testing alone is insufficient to guarantee the correctness of a (complex) systems, and unverified software may violate system safety in multiple ways, for instance: (i) the control application may contain unsafe logic that guides the system towards hazardous states; (ii) the logic may be correct but incorrectly implemented thereby creating unsafe commands at runtime (application-level faults); (iii) even with logically safe, correctly implemented control applications, faults in underlying software layers (e.g. RTOS and device drivers) can prevent the correct execution of the controller and jeopardize system safety (system-level faults). Due to the limited feasibility and high cost of platform-wide formal verification, we take a different approach. Specifically, we propose a software/hardware co-design methodology to deploy SC-CPS that (i) provide strong safety guarantees; and (ii) can utilize unverified software components to implement complex safety-critical functionalities.

Our approach relies on a key observation: by performing careful boot-sequence optimization, many embedded platforms and RTOS utilized in automotive industry, avionics, and manufacturing can be **entirely restarted** within a very short period of time. Restarting a computing system and reloading a fresh image of all the software (*i.e.,* RTOS, and applications) from a read-only source appears to be an effective approach to recover from unexpected faults. Thus, we propose the following: as soon as a fault that disrupts the execution of critical components is detected, the entire system is restarted. After a restart, all the safety-critical applications that were impacted by the restart are re-executed. If restart and re-execution of critical tasks can be performed *fast enough*, i.e. such that timing constraints are always met is spite of task re-executions, the physical system will remain oblivious to and will not be impacted by the occurrence of faults.

The effectiveness of the proposed restart-based recovery relies on timely detection of faults to trigger a restart. Since detecting logical faults in complex control applications can be challenging, we utilize Simplex Architecture [2]–[4] to construct control software. Under Simplex, safety of the system relies solely on correct execution of safety controller tasks. From a scheduling perspective, safety is guaranteed if critical tasks have enough CPU cycles to re-execute and finish before their deadlines in spite of restarts. In this paper, we analyze the conditions for a periodic task set to be schedulable in the presence of restarts and re-executions. We assume that when a restart occurs, the task instance executing on the CPU and any of the tasks that were preempted before their completion will need to re-execute after the restart. In particular, we make the following contributions:

- We propose a Simplex Architecture that can be recovered via restarts and implemented on a **single processing unit**;
- We derive the response time analysis under fixed-priority with fully preemptive and fully non-preemptive disciplines in presence of restart-based recovery and discuss pros and cons of each one;
- We propose response time analysis of fixed-priority scheduling in presence of restarts for tasks with preemption thresholds [5] and non-preemptive ending intervals [6] to improve feasibility of task sets;
- To evaluate the practicality of our restart-based recovery, we perform a proof-of-concept implementation on a 3 degree of freedom (3DOF) helicopter and test the system against various types of system and application faults.

The rest of this paper is organized in the following order. Section II provides a background on Simplex Architecture and reviews the most relevant literature. In Section III, the core assumptions on the fault model, as well as the recovery approach is discussed. Section IV explains our fault-detection using HW watchdogs. In Section V, the feasibility analysis of taskset under preemptive and non-preemptive schemes are provided. In Section VI, we analyze two hybrid scheduling disciplines that improve feasibility of task sets in the presence of restarts. In Section VII, we explain various types of faults tested on our proof-of-concept implementation. Section VIII concludes the paper.

II. Background & Related Work

A. Simplex Architecture

Simplex architecture was initially proposed in [2]–[4] to provide formal safety guarantees using only a small verified core. Under Simplex Architecture, each controlled component requires a safety controller (SC), a complex controller (CC) and a decision module (DM). SC is simple and formally verified (i.e. it has safe logic and correct implementation) such that it can always stabilize the physical plant within a safety region. CC, on the other hand, is unverified (i.e. it may contain unsafe logic or implementation bugs) and high-performance and drives the system towards the mission set points. DM includes a switching logic that attempts to use CC as much as possible to increase the progress of the system towards the goal. However, when CC does not satisfy the safety requirements, DM chooses the output from SC for actuation. A system that adheres to this architecture is guaranteed to remain safe as long as SC and DM execute correctly and complete within their deadlines.

Most of the previous work on Simplex Architecture [2]–[4], [7], [8] has focused on design of the switching logic of DM or the SC, assuming that the underlying RTOS, libraries and middle-ware will correctly execute the SC and DM. Often however, these underlying software layers are unverified and may contain bugs. Unfortunately, Simplex-based systems are note guaranteed to behave correctly in presence of system-level faults.

System-Level Simplex and its variants [9]–[11] run SC and DM as bare-metal applications on an isolated, dedicated hardware unit. By doing so, the critical components are protected from the faults in the OS or middle-ware of the complex subsystem. However, exercising this design on most multi-core platforms is challenging. The majority of commercial multi-core platforms are not designed to achieve strong inter-core fault isolation due to the high-degree of hardware resource sharing. For instance, a fault occurring in a core with the highest privilege level may compromise power and clock configuration of the entire platform. To achieve full isolation and independence, one has to utilize two separate boards/systems.

Our design enables the system to safely tolerate and recover from application-level and system-level faults that cause silent failures in SC and DM **without utilizing additional hardware**.

B. Restarting

The notion of restarting as a means of recovery from faults and improving system availability was previously studied in the literature. Most of the previous work, however, target traditional *non*-safety-critical computing systems such as servers and switches. Authors in [12] introduce recursively restartable systems as a design paradigm for highly available systems and uses a combination of revival and rejuvenation techniques. Earlier literature [13]–[15] illustrates the concept of micro-reboot which consists of having fine-grain rebootable components and trying to restart them from the smallest component to the biggest one in the presence of faults. The works in [16]–[18] focus on failure and fault modeling and try to find an optimal rejuvenation strategy for various systems.

Additionally, System-Level Simplex [9] and Reset-Based Recovery [10] also propose that the complex subsystem can be restarted upon the occurrence of faults. Restarting is possible, because the safety subsystem runs on a dedicated processing unit and is not impacted by the restarts in the complex subsystem.

C. Scheduling

One way to achieve fault-tolerance in real-time systems is to use time redundancy. Using time redundancy, whenever a fault leads to an error, and the error is detected, the faulty task is either re-executed or a different logic (recovery block) is executed to recover from the error. It is necessary that such recovery strategy does not cause any deadline misses in the task set. Fault tolerant scheduling has been extensively studied in the literature. Hereby we briefly survey those works that are more closely related.

A feasibility test for single fault-tolerant schedulability fashion under EDF is proposed in [19]. A feasibility check algorithm under multiple faults, assuming earliest deadline first scheduling (EDF) for aperiodic preemptive tasks is proposed in [20]. An exact schedulability tests using checkpointing for task sets under fully preemptive model and transient fault that affects one task is proposed in [21]. This analysis is further extended in [22] for the case of multiple faults as well as for the case where the priority of a critical task's recovery block is increased.

In [23], authors propose the exact feasibility test for fixed-priority scheduling of a periodic task set to tolerate multiple transient faults on uniprocessor. In [24] an approach is presented to schedule under fixed priority-driven preemptive scheduling at least one of the two versions of the task; simple version with reliable timing or complex version with potentially faulty.

Authors in [25] consider a similar fault model to ours, where the recovery action is to re-execute all the partially executed tasks at the instant of the fault detection *i.e.,* executing task and all the preempted tasks. This work only considers preemptive task sets under rate monotonic and shows that single faults with a minimum inter-arrival time of largest period in the task set can be recovered if the processor utilization is less than or equal to 50%.

In [26], the authors investigate the feasibility of task sets under fault bursts. Similarly to our work, the recovery action is to re-execute the faulty job along with all the partially completed (preempted) jobs at the time of fault detection. The analysis is limited to preemptive scheduling. The work in [27] addresses the problem of fault-tolerance feasibility under error bursts by performing processor speed-up to re-execute the impacted tasks. The recovery strategy in this work shares a number of similarities with our work.

Most of these works are only applicable to transient faults (*e.g.,* faults that occur due to radiation or short-lived HW malfunctions) that impact the task and do not consider faults affecting the underlying system. Additionally, most of the works assume that online fault detection or acceptance test exists. While this assumption is valid for detecting many transient faults, detecting complex system-level faults in a reliable manner is non-trivial.

Additionally, to the best of our knowledge, our paper is the first one to provide the sufficient feasibility condition in the presence of faults under the preemption threshold model and task sets with non-preemptive ending intervals.

III. System Model and Assumptions

In this paper, the assumption is that a fault and a consequent restart can take place at any point in time throughout execution. Hence, the system needs to be designed to safely tolerate restarts at any point in time. In this section we formalize the considered system and task model, and discuss the assumptions under which our methodology is applicable.

A. Sporadic and Periodic Task

We consider a task set $\mathcal{T}$ composed of n sporadic tasks $\tau_1 \dots \tau_n$ executed on a uniprocessor under fixed priority scheduling. Each task τ_i is assigned a priority level π_i. We will implicitly index tasks in decreasing priority order, *i.e.,*, τ_i has higher priority than τ_k if $i < k$. Each *sporadic task* τ_i is characterized by three positive real values; a worst-case execution time (WCET) C_i, a deadline D_i (relative to the release time), and a minimum inter-arrival time T_i, with $C_i \leq D_i$ and $C_i \leq T_i$. Sporadic tasks may be released at any time, as long as two successive arrival times are separated in time by at least T_i time units. Each instance of a sporadic task is called *job* and $\tau_{i,k}$ denotes the kth job of task τ_i. Once released, each job needs to execute for at most C_i units of time before its deadline D_i. *Periodic task* can be considered a special case of sporadic task, where the inter-arrival time is constant and always equal to the minimum inter-arrival time. In this paper, for simplicity purpose, we assume $D_i = T_i$. This indicates that each job has to finish before the next instance of the task is released.

Moreover, $hp(\pi_i)$ and $lp(\pi_i)$ refer to the set of tasks with higher or lower priority than π_i *i.e.,* $hp(\pi_i) = \{\tau_j \mid \pi_i < \pi_j\}$ and $lp(\pi_i) = \{\tau_j \mid \pi_i > \pi_j\}$.

B. Critical and Non-Critical Workload

In this work, we utilize Simplex Architecture for implementation of each the controller software. As a result, three periodic tasks are associated with every controlled component[1]; a SC task, a CC task, and a DM task. In typical designs, the three tasks that compose the same controller have the same period, deadline, and release time. SC tasks are the only ones whose timely execution is necessary for the safety of the physical plant. Therefore, out of the three tasks, SC must execute first and write its output to the actuator command buffer. On the other hand, DM needs to execute last, after the output of CC is available, to decide if it is safe to replace the SC command already in the actuator buffer. Hence, the priorities of the controller tasks need to be in the following order: $\pi(DM) < \pi(CC) < \pi(SC)$.

The set of all the SC tasks on the system is called *critical* workload. The set of all the CC and DM tasks is referred as *non-critical* workload. Safety is guaranteed if and only if all the critical tasks complete before their deadlines. Whereas, execution of non-critical tasks is not crucial for safety; these tasks are said to be mission-critical but not safety-critical.

Additionally, we assume that the first n_c tasks of $\mathcal{T}$ are critical. Notice that with this indexing strategy, any critical task has a higher priority than any non-critical task.

C. Fault Model

In this paper, we consider two types of fault for the system; application-level faults and system-level faults. We make the following assumptions about the faults that our system safely handles:

- Application faults may only occur in the unverified workload (*i.e.,* all the application-level processes on the system except SC and DM tasks).
- SC and DM tasks are independently verified and fault-free. They might, however, fail silently (no output is generated) due to faults in software layers or other applications on which they depend.
- In this paper we only consider system- and application-level faults that cause SC and DM tasks to fail silently but do not change their logic or alter their output.
- Faults do not alter sensor readings.
- We indicate with T_r the minimum inter-arrival time of the faults and the consequent restarts. We assume that the minimum inter-arrival time of restarts is larger than the least common multiple (hyper-period) of the critical tasks, i.e. $T_r > \text{LCM}\{T_k \mid k \leq n_c\}$[2].

We further assume that task instances (jobs) of SC and CC tasks are independent and individually perform sampling, computation and output of actuation commands. As such, a job remains unaffected by system restarts as long as it has completed or it has not started its execution before a restart occurs.

[1]Usually multiple physical unit/components are controlled via one processing unit.

[2]For a well-tested system, faults that require restarts are typically considered to be rare events [28], [29].

Fig. 1: Example of fully preemptive system with 3 tasks $\tau_1 = (1, 3); \tau_2 = (2, 8); \tau_3 = (4, 22)$, and restart at $t = 10 - \epsilon$ ($C_r = 0$). The taskset is schedulable without restarts, however, restart and task re-execution causes a deadline miss at $t = 22$.

D. Recovery Model

The recovery action we assume in this paper is to restart the entire system, reload all the software (RTOS and applications) from a read-only storage unit, and re-execute all the partially executed jobs (*i.e.,* preempted jobs and the job running on CPU at the restart moment). Priority of re-execution of a job is the same as the priority of the original job at the time of restart. Thus, during the re-execution of a lower priority job, if a higher priority task arrives, the higher priority task will preempt the re-executed instance of the lower priority task. In this paper, C_r refers to the time required to restart the system and reload and run the software (RTOS and applications) from a read-only image. Figure 1 depicts how restart and task re-execution affect the scheduling of 3 real-time tasks ($\tau_1, \tau_2,$ andτ_3). When the restart happens at $t = 10 - \epsilon$, τ_1 was still running. Moreover, τ_2 and τ_3 were preempted at time $t = 9$ and $t = 8$, respectively. Hence all the three task will need to be re-executed after the restart.

System restart is triggered only after a fault is detected. We utilize a reliable timing based fault detection mechanism that uses hardware watchdog (WD) timers to detect when instances of critical tasks have not finished before their worst-case response time. These mechanisms are further elaborated in Section IV.

E. Scheduler State Preservation

We assume that after a restart it is possible to know what tasks were preempted/executing at the time when a restart occurred. This can be performed in two ways: (i) by preserving minimal information about the state of the scheduler across restarts; or by (ii) relying on a monotonic clock source to determine absolute timing after a reset and use the task parameter to select a safe set of tasks for re-execution. In the first case, if scheduling algorithms with job-level static priority are used, the state of the scheduler needs to be saved only at the boundaries of tasks' activation and completion. This can be achieved by saving the schedule status to nonvolatile memory at each job termination point. The fixed overhead of writing can be incorporated in the task's WCET.

F. RBR-Feasibility

A task set $\mathcal{T}$ is said to be feasible under restart based recovery (RBR-Feasible) if the following two conditions are satisfied; *(i)* there exists a schedule such that all jobs of all the critical tasks, or their potential re-executions, can complete successfully before their respective deadlines, even in the presence of a system-wide restart, occurring at any arbitrary time during execution. *(ii)* All jobs, including instances of non-critical tasks, can complete before their deadlines when no restart is performed.

IV. Fault Detection

For a system designed based on Simplex Architecture, safety depends on timely execution of SC tasks which are part of the critical workload of the task set (*i.e.,* tasks with $i \leq n_c$). If faults prevent any of these critical tasks from finishing before the deadline, the system would be at risk. A successful fault-detection approach must be able to detect the fault before the deadline is missed and trigger the recovery procedure. Another challenge to overcome is to ensure that there is enough time after triggering a recovery procedure, such that the re-execution of the jobs can complete before their deadlines.

In this section we discuss a fault detection approach to detect faults before the deadline using a hardware watchdog timer. Later, in section V, we provide the conditions to ensure timing requirements of the critical tasks in presence of restarts.

A. Monitoring with HW WD timers

To explain the detection mechanism, we need to define the *ideal worst-case response time* of a critical task, as the worst-case response time of the task when there are no restarts (and no re-executions) in the system. In other words, it is the longest time elapsed between the arrival time (when task is ready to execute) to the completion time when no restarts occur, over all instances

of the task. We use $\mathcal{R}_i$ to denote the ideal worst-case response time of τ_i. In Section V, we explain how to obtain $\mathcal{R}_i$ for the tasks in a given task set.

If no faults occur in the system, every instance of task τ_i is expected to finish its execution within at most $\mathcal{R}_i$ time units after the arrival time of the job. This can be checked in runtime through a monitoring task that checks the list of active tasks at time instants $kT_i + \mathcal{R}_i$, to ensure that τ_i has terminated and is not in the list of ready tasks. Otherwise, it restarts the system. In each invocation of the monitoring task, following steps are performed.

1) Assuming that j is the task under monitoring, check if τ_j has completed. If τ_j has not terminated, do nothing. As a result, the WD timer will expire and system will restart.
2) If τ_j has terminated, set the next wake up time of the task to t_{next} and set the WD expiration time to $t - t_{\text{next}} + \epsilon$ where t is the current time, ϵ is a small positive value and

$$t_{\text{next}} = \min_{i \leq c_n} \Big(\lfloor t/T_i \rfloor T_i + \phi_i + \mathcal{R}_i \Big). \tag{1}$$

where ϕ_i denotes the phase of task τ_i that is, the release time of its first instance.

Notice that this simple solution utilizes only one WD timer, and handles all the silent failures. The advantage of using hardware WD timers is that if any faults in the OS or other applications, prevent the time monitor task from execution, the WD which is already set, will expire and restart the system.

V. RBR-Feasibility Analysis

As mentioned in Section IV, re-execution of jobs impacted by a restart must not cause any other job to miss a deadline. Also, re-executed jobs need to meet their deadlines as well. The goal of this section is to present a set of sufficient conditions to reason about the feasibility of a given task set $\mathcal{T}$ in presence of restarts (RBR-feasibility). In particular, in Sections V-A and V-B, we present a methodology that provides a sufficient condition for exact RBR-Feasibility analysis of preemptive and non-preemptive task sets.

Definition: *Length of level-i preemption chain* at time t is defined as sum of the executed portions of all the tasks that are in the preempted or running state, and have a priority greater than or equal to π_i at t. *Longest level-i preemption chain* is the preemption chain that has the longest length over all the possible level-i preemption chains.

For instance, consider a fully preemptive task set with four tasks; C_1 = 1, $T_1 = 5$, C_2 = 3, $T_2 = 10$, C_3 = 2, $T_3 = 12$, C_4 = 4, $T_3 = 15$, and $\pi_4 < \pi_3 < \pi_2 < \pi_1$. For this task set, the longest level-3 and level-4 preemption chains are 6 and 10, respectively.

A. Fully Preemptive Task Set

Under fully preemptive scheme, as soon as a higher priority task is ready, it preempts any lower priority tasks running on the processor. To calculate the worst-case response time of task τ_i, we have to consider the case where the restart incurs the longest delay on finishing time of the job. For a fully preemptive task set, this occurs when every task τ_k for $k \in \{2, \ldots, i\}$ is preempted immediately prior to its completion by τ_{k-1} and system restarts right before the completion of τ_1. In other words, when tasks τ_1 to τ_i form the longest level-i preemption chain. An example of this case is depicted in Figure 1. In this case, the restart and consequent re-execution causes a deadline miss at $t = 22$. The example uses only integer numbers for task parameters, hence tasks can be preempted only up to 1 unit of time before their completion. In the rest of the paper, we discuss our result assuming that tasks' WCETs are real numbers.

Theorem 1 provides RBR-feasibility conditions for a fully preemptive task set $\mathcal{T}$, under fixed priority scheduling.

Theorem 1. *A set of preemptive sporadic (periodic) tasks $\mathcal{T}$ is RBR-Feasible under fixed priority algorithm if the response time R_i of each task τ_i satisfies the condition: $\forall \tau_i \in \mathcal{T}, R_i \leq T_i$. R_i is obtained for the smallest value of k for which we have $R_i^{(k+1)} = R_i^{(k)}$.*

$$R_i^{(k+1)} = C_i + \sum_{\tau_j \in hp(\pi_i)} \left\lceil \frac{R_i^{(k)}}{T_j} \right\rceil C_j + \mathcal{O}_i^p \tag{2}$$

where the restart overhead $\mathcal{O}_i^p$ on response time is

$$\mathcal{O}_i^p = \begin{cases} C_r + \sum_{\tau_j \in hp(\pi_i) \cup \{\tau_i\}} C_j & i \leq n_c \\ 0 & i > n_c \end{cases} \tag{3}$$

Proof. First, note that Equation 2 without the overhead term $\mathcal{O}_i^p$, corresponds to the classic response time of a task under fully preemptive fixed priority scheduling [30]. The additional overhead term represents the worst-case interference on the task instance under analysis introduced by restart time and the re-execution of the preempted tasks. We need to show that the overhead term can be computed using Equation 3. Consider the scenario in which every task τ_k is preempted by τ_{k-1} after executing for δ_i time units where $k \in \{2, ..., i\}$. And, a restart occurs after τ_1 executed for δ_1 time units. Due to the restart, all the tasks have to re-execute and the earliest time τ_i can finish its execution is $C_r + \delta_i + ... + \delta_1 + C_i + ... + C_1$. Hence, it is obvious that the later each preemption or the restart in τ_1 occurs, the more delay it creates for τ_i. Once a task has completed, it no longer needs to be re-executed. Therefore, the maximum delay of each task is felt immediately prior to the task's completion instant. Thus, the

overhead is maximized when each τ_k is preempted by τ_{k-1} for $k \in \{2, .., i\}$ and restart occurs immediately before the end of τ_1. □

As seen in this section, the worst-case overhead of restart-based recovery in fully preemptive setting occurs when system restarts at the end of longest preemption chain. Therefore, to reduce the overhead of restarting, length of the longest preemption chain must be reduced. In order to reduce this effect we investigate the non-preemptive setting in the following section.

B. Fully Non-Preemptive Task set

Under this model, jobs are not preempted until their execution terminates. At every termination point, the scheduler selects the task with the highest priority amongst all the ready tasks to execute. The main advantage of non-preemptive task set is that at most one task instance can be affected by restart at any instant of time.

Authors in [31] showed that in non-preemptive scheduling, the largest response time of a task does not necessarily occur in the first job after the critical instant. In some cases, the high-priority jobs activated during the non-preemptive execution of τ_i's first instance are pushed ahead to successive jobs, which then may experience a higher interference. Due to this phenomenon, the response time analysis for a task cannot be limited to its first job, activated at the critical instant, as done in preemptive scheduling, but it must be performed for multiple jobs, until the processor finishes executing tasks with priority higher than or equal to π_i. Hence, the response time of a task needs to be computed within the longest *Level-i Active Period*, defined as follows [32], [33].

Definition: The *Level-i Active Period* L_i is an interval $[a, b)$ such that the amount of processing that still needs to be performed at time t due to jobs with priority higher than or equal to π_i, released strictly before t, is positive for all $t \in (a, b)$ and null in a and b. It can be computed using the following iterative relation:

$$L_i^{(q)} = B_i + C_i + \sum_{j \in hp(\pi_i)} \lceil L_i^{(q-1)}/T_j \rceil C_j + \mathcal{O}_i^{np} \tag{4}$$

Here, $\mathcal{O}_i^{np}$ is the maximum overhead of restart on the response time of a task. In the following we describe how to calculate this value. L_i is the smallest value for which $L_i^{(q)} = L_i^{(q-1)}$. This indicates that the response time of task τ_i must be computed for all jobs $\tau_{i,k}$ with $k \in [1, K_i]$ where $K_i = \lceil L_i/T_i \rceil$.

Theorem 2 describes the sufficient conditions under which a fault and the subsequent restart do not compromise the timely execution of the critical workload under fully non-preemptive scheduling.

Theorem 2. *A set of non-preemptive sporadic (periodic) tasks is RBR-feasible under fixed-priority if the response time R_i of each task τ_i, calculated through following relation, satisfies the condition: $\forall \tau_i \in \mathcal{T}; R_i \leq T_i$.*

$$R_i = \max_{k \in [1, K_i]} \{F_{i,k} - (k-1)T_i\} \tag{5}$$

where $F_{i,k}$ is the finishing time of job $\tau_{i,k}$ given by

$$F_{i,k} = S_{i,k} + C_i \tag{6}$$

Here, $S_{i,k}$ is the start time of job $\tau_{i,k}$, obtained for the smallest value that satisfies $S_{i,k}^{(q+1)} = S_{i,k}^{(q)}$ in the following relation

$$S_{i,k}^{(k+1)} = B_i + \sum_{\tau_j \in hp(\pi_i)} \left(\left\lfloor \frac{S_{i,k}^{(k)}}{T_j} \right\rfloor + 1 \right) C_j + \mathcal{O}_i^{np} \tag{7}$$

In Equation 7, term B_i is the blocking from low priority tasks and is calculated as $B_i = \max_{\tau_j \in lp(\pi_i)} \{C_j\}$. The term $\mathcal{O}_i^{np}$ represents the overhead on task execution introduced by restarts and is calculated as follows:

$$\mathcal{O}_i^{np} = \begin{cases} C_r + max\left\{\{C_j \mid j \in hp(\pi_i)\} \cup C_i\right\} & i \leq n_c \\ 0 & i > n_c \end{cases} \tag{8}$$

Proof. Equation 7 and 6, without the restart overhead term $\mathcal{O}_i^{np}$, are proposed in [32], [33] to calculate the worst-case start time and response time of a task under non-preemptive setting.

We need to show that the overhead term can be computed using Equation 8. Under non-preemptive discipline, restart only impacts a single task executing on the CPU at the instant of restart. There are two possible scenarios that may result in the worst-case restart delay on finish time of task τ_i. First, when τ_i is waiting for the higher priority tasks to finish their execution, a restart can occur during the execution of one of the higher priority tasks τ_j and delay the start time τ_i by $C_r + C_j$. Alternatively, a restart can occur infinitesimal time prior to the completion of τ_i and cause an overhead of $C_r + C_i$. Hence, the worst-case delay due to a restart is caused by the task with the longest execution time among the task itself and the tasks with higher priority (Equation 8). The restart overhead is not included in the response-time of non-critical tasks ($\mathcal{O}_i^{np} = 0$ for $i > n_c$). □

Unfortunately, under non-preemptive scheduling, blocking time due to low priority tasks, may cause higher priority tasks with short deadlines to be non-schedulable. As a result, when preemptions are disabled, there exist task sets with arbitrary low utilization that despite having the lowest restart overhead, are not RBR-Feasible. Figure 2 uses the same task parameters as in Figure 1. The plot shows that the considered task system is not schedulable under fully non-preemptive scheduling when a restart is triggered at $t = 5 - \epsilon$.

Fig. 2: Example of fully non-preemptive system with 3 tasks $\tau_1 = (1, 3); \tau_2 = (2, 8); \tau_3 = (4, 22)$, and restart at $t = 5 - \epsilon$ ($C_r = 0$). Restart and task re-execution causes a deadline miss at $t = 9$.

VI. Limited Preemptions

In the previous section, we analyzed the RBR-Feasibility of task sets under fully preemptive and fully non-preemptive scheduling. Under full preemption, restarts can cause a significant overhead because the longest preemption chain can contains all the tasks. On the other hand, under non-preemptive scheduling, the restart overhead is minimum. However, due to additional blocking on higher priority tasks, some task sets, even with low utilization, are not schedulable.

In this section we discuss two alternative models with limited preemption. Limited preemption models are suitable for restart-based recovery since they enable the necessary preemptions for the schedulability of the task set, but avoid many unnecessary preemptions that occur in fully preemptive scheduling. Consequently, they induce lower restarting overhead and exhibit higher schedulability.

A. Preemptive tasks with Non-Preemptive Ending

As seen in the previous sections, reducing the number and length of preempted tasks in the longest preemption chain, can reduce the overhead of restarting and increase the RBR-Feasibility of task sets. On the other hand, preventing preemptions entirely is not desirable since it can impact feasibility of the high priority tasks with short deadlines. As a result, we consider a hybrid preemption model in which, a job once executed for longer than $C_i - Q_i$ time units, switches to non-preemptive mode and continues to execute until its termination point. Such a model allows a job that has mostly completed to terminate, instead of being preempted by a higher priority task. Q_i is called the size of non-preemptive ending interval of τ_i and $Q_i \leq C_i$. The model we utilize in this section, is a special case of the model proposed in [6] which aims to decrease the preemption overhead due to context switch in real-time operating systems. In Figure 3, we consider a task set with the same parameters as in Figure 1, where in addition task τ_3 has a non-preemptive region of length $Q_3 = 1$. The preemption chain that caused the system in Figure 1 to be non-schedulable cannot occur and the instance of the task becomes schedulable under restarts. With the same setup, Figure 4 considers the case when a reset occurs at $t = 9 - \epsilon$.

1) RBR-Feasibility Analysis: Theorem 3 provides the RBR-feasibility conditions of a task-set with non-preemptive ending intervals. In this theorem, $S_{i,k}$ represents the worst case start time of the non-preemptive region of the re-executed instance of job $\tau_{i,k}$. Similarly, $F_{i,k}$ is used to represent the worst-case finish time. The arrival time of instance k of task $\tau_{i,k}$ is $(k-1)T_i$.

Theorem 3. *A set of sporadic (periodic) tasks $\mathcal{T}$ with non-preemptive ending regions of length Q_i, is RBR-Feasible under a fixed priority algorithm if the worst-case response time R_i of each task τ_i, calculated from Equation 9, satisfies the condition: $\forall \tau_i \in \mathcal{T}, R_i \leq T_i$.*

$$R_i = \max_{k \in [1, K_i]} \{F_{i,k} - (k-1)T_i\} \tag{9}$$

where

$$F_{i,k} = S_{i,k} + Q_i \tag{10}$$

and $S_{i,k}$ is obtained for the smallest value of q for which we have $S_{i,k}^{(q+1)} = S_{i,k}^{(q)}$ in the following

$$S_{i,k}^{(q+1)} = B_i + (k-1)C_i + C_i - Q_i + \sum_{\tau_j \in hp(\tau_i)} \left(\left\lfloor \frac{S_{i,k}^{(q)}}{T_j} \right\rfloor + 1 \right) C_j + \mathcal{O}_i^{npe} \tag{11}$$

Here, the term B_i is the blocking from low priority tasks and is calculated by

$$B_i = \max_{\tau_k \in lp(\pi_i)} \{Q_k\}. \tag{12}$$

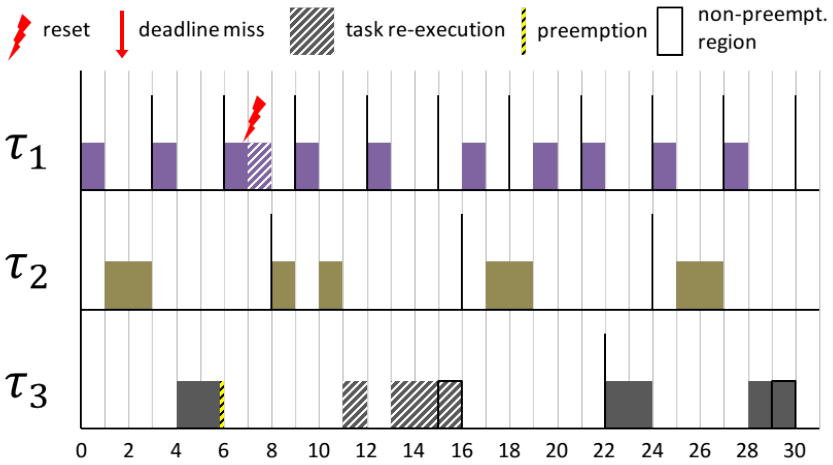


Fig. 3: Example of system with 3 tasks $\tau_1 = (1, 3); \tau_2 = (2, 8); \tau_3 = (4, 22)$, where τ_3 has a non-preemptive region of size $Q_3 = 1$. Restart occurs at $t = 7 - \epsilon$ ($C_r = 0$). The task set is schedulable with restarts.

$\mathcal{O}_i^{npe}$ is the maximum overhead of the restart on the response time and is calculated as follows:

$$\mathcal{O}_i^{npe} = \begin{cases} C_r + \mathcal{WCWE}(i) & i \leq n_c \\ 0 & i > n_c \end{cases} \tag{13}$$

where $\mathcal{WCWE}(i)$ is the worst-case amount of the execution that may be wasted due to the restarts. It is given by the following where $\mathcal{WCWE}(1) = C_1$ and

$$\mathcal{WCWE}(i) = C_i + max\Big(0, \mathcal{WCWE}(i-1) - Q_i\Big) \tag{14}$$

K_i in Equation 9 can be computed from Equation 4 by using $\mathcal{O}_i^{npe}$ instead of $\mathcal{O}_i^{np}$.

Proof. Authors in [5] show that the worst-case response time of task τ_i is the maximum difference between the worst case finish time and the arrival time of the jobs that arrive within the level-i active period (Equation 9).

Hence, we must compute the worst-case finish time of job $\tau_{i,k}$ in the presence of restarts. When a restart occurs during the execution of $\tau_{i,k}$ or while it is in preempted state, $\tau_{i,k}$ needs to re-execute. Therefore, the finish time of the $\tau_{i,k}$ is when the re-executed instance completes. As a result, to obtain the worst-case finish time of $\tau_{i,k}$, we calculate the response time of each instance when a restart with longest overhead has impacted that instance. We break down the worst-case finish time of $\tau_{i,k}$ into two intervals: the worst-case start time of the non-preemptive region of the re-executed job and the length of the non-preemptive region, Q_i (Equation 10). $S_{i,k}$ in Equation 10, is the worst-case start time of non-preemptive region of job $\tau_{i,k}$ which can be iteratively obtained from Equation 11. Equation 11 is an extension of the start time computation from [33]. In the presence of non-preemptive regions, an additional blocking factor B_i must be considered for each task τ_i, equal to the longest non-preemptive region of the lower priority tasks. Therefore, the maximum blocking time that τ_i may experience is $B_i = \max_{\tau_j \in lp(\pi_i)} \{Q_j\}$. B_i is added to the worst-case start time of the task in Equation 11.

For a task τ_i with the non-preemptive region of size Q_i, there are two cases that may lead to the worst-case wasted time. First case is when the system restarts immediately prior to the completion of τ_i, in which case the wasted time is C_i. Second case occurs when τ_i is preempted immediately before the non-preemptive region begins (*i.e.,* at $C_i - Q_i$) by the higher priority task τ_{i-1}. In this case, the wasted execution is $C_i - Q_i$ plus the maximum amount of the execution of the higher priority tasks that may be wasted due to the restarts (*i.e.,* $\mathcal{WCWE}(i-1)$). The worst-case wasted execution is the maximum of these two values *i.e.,* $\mathcal{WCWE}(i) = max(C_i, C_i - Q_i + \mathcal{WCWE}(i-1)) = C_i + max(0, \mathcal{WCWE}(i-1) - Q_i)$. Similarly, $\mathcal{WCWE}(i-1)$ can be computed recursively. □

2) Optimal Size of Non-Preemptive Regions: RBR-Feasibility of a taskset depends on the choice of Q_is for the tasks. In this section, we present an approach to determine the size of non-preemptive regions Q_i for the tasks to maximize the RBR-Feasibility of the task set.

First, we introduce the the notion of *blocking tolerance* of a task β_i. β_i is the maximum time units that task τ_i may be blocked by the lower priority tasks, while it can still meet its deadline. Algorithm 1, uses binary search and the response time analysis of task (from Theorem 3) to find β_i for a task τ_i.

In Algorithm 1, $R_{i,B_i=middle}$ is computed as described in Theorem 3 (Equation 9), where instead of using the B_i from Equation 12, the blocking time is set to the value of $middle$.

Note that, if Algorithm 1 cannot find a β_i for task τ_i, this task is not schedulable at all. This indicates that there is not any selection of Q_is that would make $\mathcal{T}$ RBR-Feasible.

Given that task τ_1 has the highest priority, it may not be preempted by any other task; hence we set $Q_1 = C_1$. The next theorem shows how to drive optimal Q_i for the rest of the tasks in $\mathcal{T}$. The results are optimal, meaning that if there is at least one set of Q_is under which $\mathcal{T}$ is RBR-Feasible, it will find them.

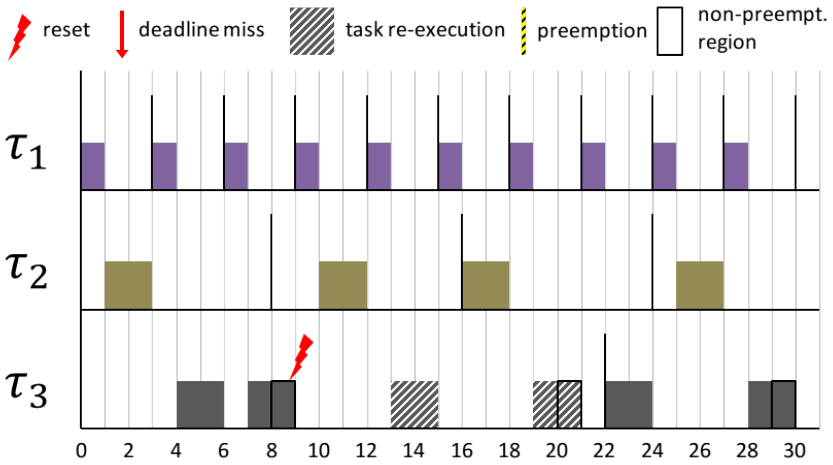


Fig. 4: Example of system with 3 tasks $\tau_1 = (1, 3); \tau_2 = (2, 8); \tau_3 = (4, 22)$, where τ_3 has a non-preemptive region of size $Q_3 = 1$. Restart occurs at $t = 9 - \epsilon$ ($C_r = 0$). The task set is schedulable with restarts.

Algorithm 1: Binary Search for Finding β_i

```
FindBlockingTolerance(τ_i, 𝒯, Q_1, ..., Q_i)
    start = 0; end = T_i /* Initialize the interval */
    if R_i(start) > T_i then return τ_i Not Schedulable;
    while end - start > ϵ do
        middle = (start + end)/2
        if R_{i,B_i=middle} > T_i then end = middle ;
        else start = middle
    end
    return β_i = start;
```

Theorem 4. *The optimal set of non-preemptive interval Q_is of tasks τ_i for $2 \leq i \leq n$ is given by:*

$$Q_i = min\{min\{\beta_j \mid j \in hp(\pi_i)\}, C_i\} \tag{15}$$

assuming that $\beta_j \geq 0$ for $j \in hp(\pi_i)$.

Proof. Increasing the length of Q_i for a task reduces the response time in two ways. First, from Equation 11, increasing Q_i reduces the start time of the job $S_{i,k}$ which reduces the finish time and consequently the response time of τ_i. Second, from Equation 14, increasing Q_i reduces the restart overhead $\mathcal{O}_i^{npe}$ on the task and lower priority tasks which in turn reduces the response time. Thus Q_i may increase as much as possible up to the worst-case execution time C_i; $Q_i \leq C_i$. However, the choice of Q_i must not make any of the higher priority tasks unschedulable. As a result, Q_i must be smaller than the smallest blocking tolerance of all the tasks with higher priority than π_i; $Q_i \leq min\{\beta_j | j \in hp(\pi_i)\}$. Combining these two conditions results in the relation of Equation 15. □

B. Preemption Thresholds

In the previous section, we discussed non-preemptive endings as a way to reduce the length of the longest preemption chain and decrease the overhead of restarts. In this section, we discuss an alternative approach to reduce the number of tasks in the longest preemption chain and thus reduce the overhead of restart-based recovery.

To achieve this goal, we use the notion of preemption thresholds which has been proposed in [5]. According to this model, each task τ_i is assigned a nominal priority π_i and a preemption threshold $\lambda_i \geq \pi_i$. In this case, τ_i can be preempted by τ_h only if $\pi_h > \lambda_i$. At activation time, priority of τ_i is set to the nominal value π_i. The nominal priority is maintained as long as the task is kept in the ready queue. During this interval, the execution of τ_i can be delayed by all tasks with priority $\pi_h > \pi_i$, and by at most one lower priority task with threshold $\lambda_l \geq \pi_i$. When all such tasks complete, τ_i is dispatched for execution, and its priority is raised to λ_i. During execution, τ_i can be preempted by tasks with priority $\pi_h > \lambda_i$. When τ_i is preempted, its priority is kept at λ_i.

Restarts may increase the response time of $\tau_{i,k}$ in one of two ways; A restart may occur after the arrival of the job but before it has started, delaying its start time $S_{i,k}$. Alternatively, the system can be restarted after the job has started. We use $\mathcal{O}_i^{pt,s}$ to denote the worst-case overhead of a restart that occurs before the start time of a job in task sets with preemption thresholds. And, $\mathcal{O}_i^{pt,f}$ is used to represent the worst-case overhead of a restart that occurs after the start time of a job in task sets with preemption thresholds.

In Figure 5, we consider a task set with the same parameters as in Figure 1 where in addition τ_2 and τ_3 have a preemption threshold equal to $\lambda_2 = 1$ and $\lambda_3 = 2$, respectively. This assignment is effective to prevent a long preemption chain, and the jobs do not miss their deadline when the restart occurs at $t = 7 - \epsilon$. Notice that, the task set is still not RBR-Feasible since if the restart occurs at $t = 9 - \epsilon$, some job will miss the deadline, as shown in Figure 6.

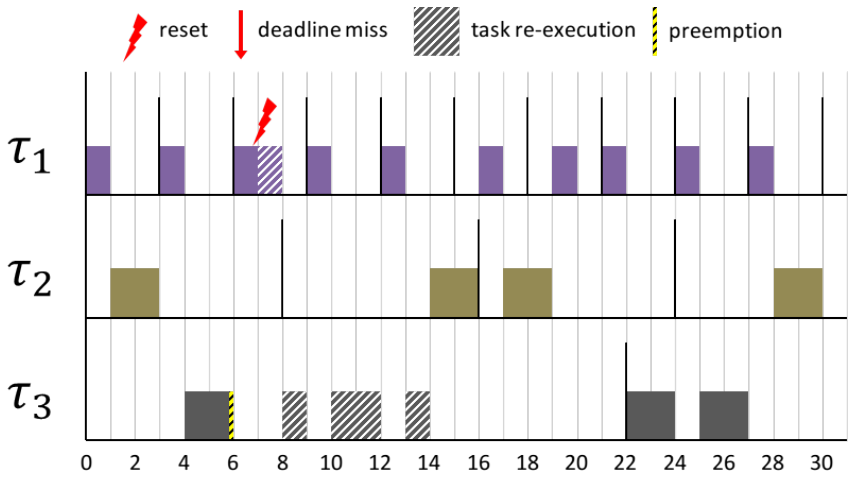


Fig. 5: Example of system with 3 tasks $\tau_1 = (1, 3); \tau_2 = (2, 8); \tau_3 = (4, 22)$, where τ_2 and τ_3 have a preemption threshold of $\lambda_2 = 1$ and $\lambda_3 = 2$, respectively. Restart occurs at $t = 7 - \epsilon$ ($C_r = 0$). In this case, the task set remains schedulable.

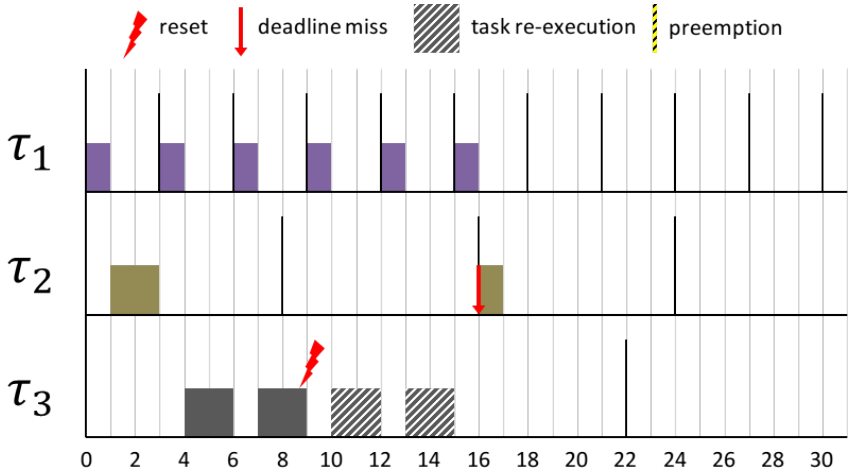


Fig. 6: Example of system with 3 tasks $\tau_1 = (1, 3); \tau_2 = (2, 8); \tau_3 = (4, 22)$, where τ_2 and τ_3 have a preemption threshold of $\lambda_2 = 1$ and $\lambda_3 = 2$, respectively. Restart occurs at $t = 9 - \epsilon$ ($C_r = 0$). The task set is not schedulable.

Theorem 5. *For a task set with preemption thresholds under fixed priority, the worst-case overhead of a restart that occurs after the start of the job $\tau_{i,k}$ is $\mathcal{O}_i^{pt,f} = C_r + \mathcal{WCWE}(i)$ where*

$$\mathcal{WCWE}(i) = C_i + \max\{\mathcal{WCWC}(j) \mid \tau_j \in hp(\lambda_i)\} \tag{16}$$

Here, $\mathcal{WCWC}(1) = C_1$.

Proof. After a job $\tau_{i,k}$ starts, its priority is raised to λ_i. In this case, the restart will create the worst-case overhead if it occurs at the end of longest preemption chain that includes τ_i and any subset of the tasks with $\pi_h > \lambda_i$. Equation 16 uses a recursive relation to calculate the length of longest preemption chain consisting of τ_i and all the tasks with $\pi_h > \lambda_i$. □

Theorem 6. *For a task set with preemption thresholds under fixed priority, a restart occurring before the start time of a job $\tau_{i,k}$, can cause the worst-case overhead of*

$$\mathcal{O}_i^{pt,s} = C_r + \max\{\mathcal{WCWE}(j) \mid \tau_j \in hp(\pi_i)\} \tag{17}$$

where $\mathcal{WCWE}(j)$ can be computed from Equation 16.

Proof. Start time of a task can be delayed by a restart impacting any of the tasks with priority higher than π_i. Equation 17 recursively finds the longest possible preemption chain consisting of any subset of tasks with $\pi_h > \pi_i$. □

Due to the assumption of one fault per hyper-period, each job may be impacted by at most one of $\mathcal{O}_i^{pt,f}$ or $\mathcal{O}_i^{pt,s}$, but not both at the same time. Hence, we compute the finish time of the task once assuming that the restart occurs before the start time *i.e.,* $\mathcal{O}_i^{pt,f} = 0$, and another time assuming it occurs after the start time *i.e.,* $\mathcal{O}_i^{pt,s} = 0$. Finish time in these two cases is referred respectively by $F_{i,k}^s$ (restart before the start time) and $F_{i,k}^f$ (restart after the start time).

We expand the response time analysis of tasks with preemption thresholds from [5], considering the overhead of restarting. In the following, $S_{i,k}$ and $F_{i,k}$ represent the worst case start time and finish time of job $\tau_{i,k}$. And, the arrival time of $\tau_{i,k}$ is $(k-1)T_i$. The worst-case response time of task τ_i is given by:

$$R_i = \max_{k \in [1, K_i]} \left\{ \max\{F^s_{i,k}, F^f_{i,k}\} - (k-1)T_i \right\} \tag{18}$$

Here, K_i can be obtained from Equation 4 by using $max(\mathcal{O}_i^{pt,f}, \mathcal{O}_i^{pt,s})$ instead of $\mathcal{O}_i^{np}$. A task τ_i can be blocked only by lower priority tasks that cannot be preempted by it, that is:

$$B_i = \max_j \{C_j \mid \pi_j < \pi_i \leq \lambda_j\} \tag{19}$$

To compute finish time, $S_{i,k}$ is computed iteratively using the following equation [5]:

$$S^{(q)}_{i,k} = B_i + (k-1)C_i + \sum_{j \in hp(\pi_i)} \left(1 + \left\lfloor \frac{S^{(q-1)}_{i,k}}{T_j} \right\rfloor \right) C_j + \mathcal{O}_i^{pt,s} \tag{20}$$

Once the job starts executing, only the tasks with higher priority than λ_i can preempt it. Hence, the $F_{i,k}$ can be derived from the following:

$$F^{(q)}_{i,k} = S_{i,k} + C_i + \sum_{j \in hp(\lambda_i)} \left(\left\lceil \frac{F^{(q-1)}_{i,k}}{T_j} \right\rceil - \left(1 + \left\lfloor \frac{S_{i,k}}{T_j} \right\rfloor \right) \right) C_j + \mathcal{O}_i^{pt,f} \tag{21}$$

Task set $\mathcal{T}$ is considered RBR-Feasible if $\forall \tau_i \in \mathcal{T}, R_i \leq T_i$.

RBR-Feasibility of a task set depends on the choice of λ_is for the tasks. In this paper, we use a genetic algorithm to find a set of preemption thresholds to achieve RBR-Feasibility of the task-set. Although this algorithm can be further improved to find the optimal threshold assignments, the proposed genetic algorithm achieves acceptable performance, as we show in Section VII.

VII. Evaluation and Case Study

In this section, we compare and evaluate the four fault-tolerant scheduling strategies discussed in this paper. In addition, to demonstrate the practicality of the proposed restart-based fault tolerance, we implemented a controller on a commercial-off-the-shelf (COTS) embedded board for a 3DOF helicopter [34] (Figure 7)and empirically validate the practicality of our approach. We inject faults in the control logic, control application, and the operating system and demonstrate that the system remains safe, despite the occurrence of faults, and restarts.

A. Proof-Of-Concept Implementation

1) 3DOF helicopter and Controllers: The 3DOF helicopter (displayed in figure 7) is a simplified helicopter model, ideally suited to test intermediate to advanced control concepts and theories relevant to real world applications of flight dynamics and control in the tandem rotor helicopters, or any device with similar dynamics [34]. It is equipped with two motors that can generate force in the upward and downward direction, proportionally to the given actuation voltage. It also has three sensors to measure elevation, pitch and travel angle as shown in Figure 7.

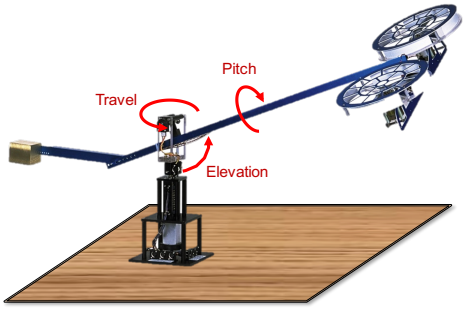


Fig. 7: 3 Degree of freedom (3DOF) helicopter.

Safety for the 3DOF helicopter is achieved if the propellers do not hit the surface underneath, as shown in Figure 7. The subset of the state space in which there is no contact between the helicopter and the surface underneath is given by $-\epsilon - \rho/3 \leq 0.3$ and $-\epsilon + \rho/3 \leq 0.3$ where ϵ and ρ are the elevation and pitch angles of the helicopter. The helicopter is considered safe if it never exits this region. To design the decision logic and the safety controller, we use the Simplex-based approach proposed in [35] and the linear model of 3DOF helicopter obtained from the manufacturer manual [34]. We also design a complex controller that

3DOF Helicopter Fault Injection					
Failure Type	**Fault Category**	Safety			**Restarted**
		Application-Level Simplex (1 processing unit)	System-Level Simplex (2 processing units)	Our Approach (1 processing unit)	
No Output	App.	✔	✔	✔	No
Maximum Voltage	App.	✔	✔	✔	No
Time Degraded Control	App.	✔	✔	✔	No
Resource Blocking - CPU	RTOS/App.	✗	✔	✔	Yes
Resource Blocking - Port	RTOS/App.	✗	✔	✔	Yes
FreeRTOS Freeze	RTOS	✗	✔	✔	Yes
CPU Reboot	RTOS	✗	✔	✔	Yes

TABLE I: Our approach tolerates the system-level faults using only one processing unit. Whereas, System-Level Simplex [9] needs an extra processing unit to tolerate these faults.

makes the helicopter follow a set of circular set points The logic of the complex controller is not safe, i.e. as it control the plant to follow the set points, the helicopter may hit the surface.

2) Controller Implementation: For the prototype of the proposed design, a Freescale i.MX7D application processor is considered. This SoC provides two general purpose ARM Cortex-A7 cores capable of running at the maximum frequency of 1 GHz and one real-time ARM Cortex-M4 core that runs at the maximum frequency of 200 MHz. The real-time core runs from tightly coupled memory to ensure predictable behavior required for real-time applications/tasks. The real-time core on the considered platform runs FreeRTOS [3], a popular open-source real-time operating system.

Due to the real-time constraints of the control tasks, we choose the real-time ARM Cortex-M4 core to run the controller. Faults occurring on ARM Cortex-M4 core trigger a full restart of the ARM Cortex-M4 core followed by a full reload of the image containing the OS (FreeRTOS) and the control software. On the i.MX7D platform, only the Cortex-A7 cores have direct access to the flash memory, and only these two cores are responsible for loading the binary images of the real-time core from flash. The first time after a platform bootstrap, the image of the FreeRTOS and the control software is loaded into the RAM of the Cortex-A7 cores. Every time the real-time core restarts, the FreeRTOS and control applications are loaded into real-time core's tightly coupled memory by any of the Cortex-A7 cores. This procedure also reduces the restart-time of the real-time core as it does not have to read the image from flash memory (which is usually slower than RAM). With the above implementation, the restart time of the real-time core is less than 1ms[4].

The control tasks (SC, CC, and DM tasks) on the ARM Cortex-M4 core run with a frequency of 50 Hz (*i.e.,* period of 20 ms). Our controller interfaces with the 3DOF helicopter through a PCIe-based *Q8 High-Performance H.I.L. Control and data acquisition unit* [36] and an intermediate Linux-based PC. The PC communicates with the i.MX7D through a serial port and uses a custom Linux driver for sending the voltages to the 3DOF helicopter motors and reading the sensor values. Inside FreeRTOS, a task periodically checks if the SC and DM tasks have completed before their response time, and it updates the WD timer accordingly. Otherwise, the WD expires, triggering a restart of real-time core. After the completion instant of every task, the scheduler state is stored in a fixed address of the Cortex-A7 RAM. Notice that the Cortex-A7 RAM is not impacted by restarting the Cortex-M4 core. Therefore, after the restart of Cortext-M4 core, the scheduler state can retrieved from the system RAM of Cortex A7 cores.

B. Fault Injection

Table I presents a list of faults that we tested on our implemented controller. For the investigated system-level faults, we observed that the WD timer restarted the system, the safety controller task did not miss the deadline after restart, and the physical system remained in safe state. For application faults, we observed that when the helicopter's state approached potentially unsafe states, as a result of faults or unsafe logic in complex controller, the DM detected the safety hazard and switched the control to safety controller. This prevented the helicopter from crashing.

To provide a comparison point, we provide the performance of Application-Level Simplex and System-Level Simplex. As shown in Table I, System-Level Simplex is able to tolerate all the system-level and application-level faults because, due to hardware isolation, backup controller is not impacted by the system-level faults on the complex subsystem. Whereas our proposed approach does not require any additional hardware.

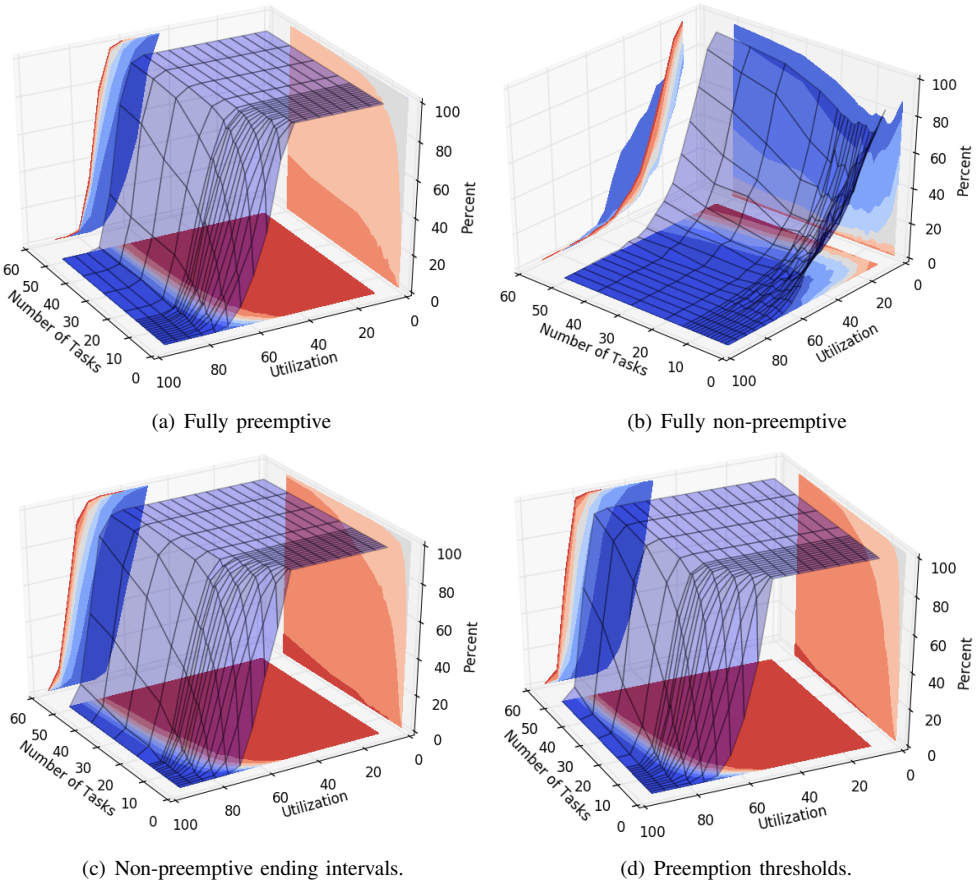
In the rest of this section, we elaborate some of these faults.

1) Maximum Voltage in Wrong Way: The helicopter should not hit the surface even if the CC outputs a voltage that normally would result in a crash. We consider an extreme case of this scenario where the CC generates a voltage that pushes the helicopter towards the surface. The unsafe CC commands were detected by DM, and the control was switched to the SC until the helicopter was in the safety and then control was handed back to CC.

2) Blocking Shared Resource: A faulty process may behave differently in runtime from its expected/reported behavior. For instance, it may lock a particular resource which is also needed by other critical tasks for more than the intended duration. Or, it may execute for more time than its reported WCET that is used for the schedulability test of the system. These faults may also originate from RTOS or driver misbehavior due to race condition or other reasons. If the fault delays/stops the execution of the

[3]http://www.freertos.org

[4]SC task activates one of the GPIO pins immediately after it executes. The restart time is measured externally using the signal on this pin. After multiple experiments, a conservative upper bound was picked for the restart time.



(a) Fully preemptive

(b) Fully non-preemptive

(c) Non-preemptive ending intervals.

(d) Preemption thresholds.

Fig. 8: Minimum Period: 10, Maximum Period: 1000

DM or SC, WD will trigger a system-wide restart. This recovers the system from the fault and keeps the physical system safe. We perform two experiments to test the fault-tolerance against this category of faults.

In the first experiment, we added an additional task to the system that uses the serial port in parallel to the SC task to communicate with the PC. We injected a fault into this task such that in random execution cycles, it holds the lock on the serial port for more than the intended period. This prevents the SC task from sending the control command (for which it requires the serial port). As a result, WD expires and restarts the system. We observed that the system recovered from the fault and remained safe during the restart.

In the second test, we introduced a task that runs at the same priority as the SC and DM. We injected a fault, into the task such that in some cycles, its execution time exceeds its reported WCET. FreeRTOS runs the tasks with equal priority using round-robin scheduling with a context switch at every 1ms. Therefore, the faulty task delays the response time of the DM and SC. If the interference is too long, the output of SC may not be ready by the time, the flushing task needs to update the actuators. When this happened, WD restarted the system.

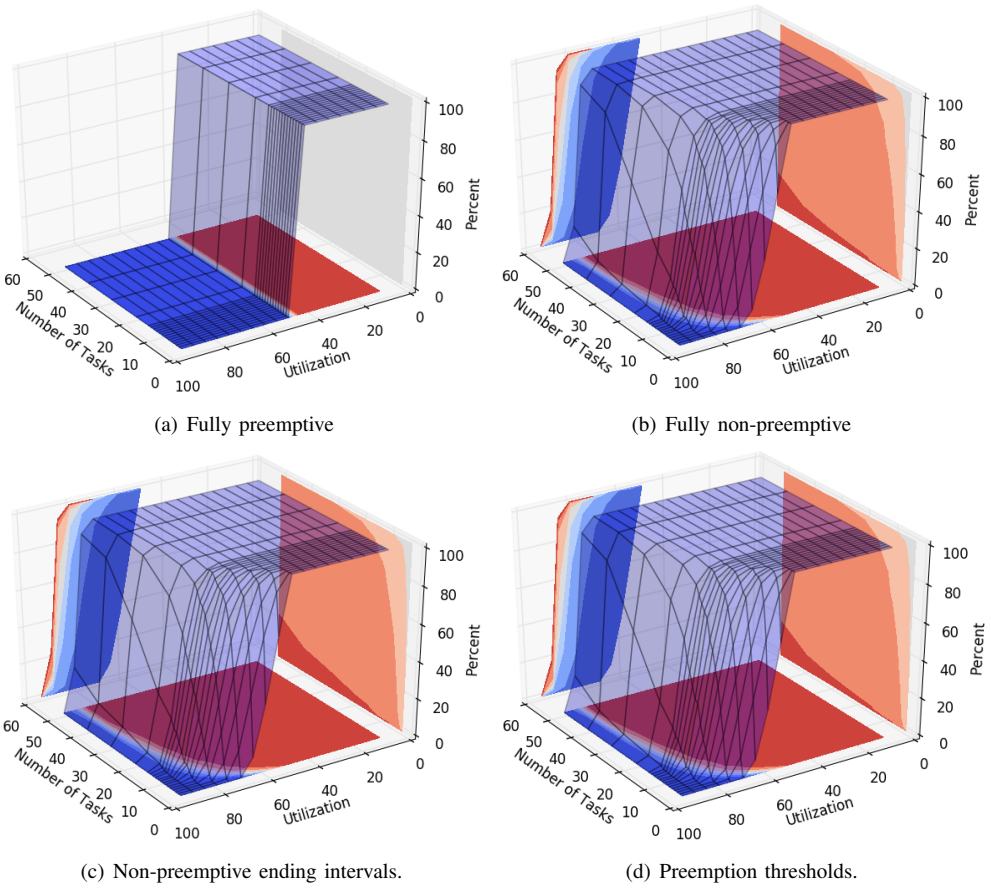
C. Evaluating Performance of Scheduling Schemes

In this section, we evaluate the performance of four fault-tolerant scheduling schemes that are discussed in this paper. For each data point in the experiments, 500 task sets with the specified utilization and number of tasks are generated. Then, RBR-feasibility of the task sets are evaluated under four discussed schemes; fully preemptive, fully non-preemptive, non-preemptive ending intervals, and preemption thresholds. In order to evaluate performance of the scheduling schemes, all the tasks in the analysis are assumed to be part of the critical workload. Priorities of the tasks are assigned according to the periods, so a task with shorter period has a higher priority.

The experiments are performed with two sets of parameters for the periods of the task sets. In the first set of experiments (Figure 8), task sets are generated with periods in the range of 10 to 1000 time units. In the second set (Figure 9), tasks have a period in the range of 900 to 1000 time units. As a result, tasks in the first experiment have more diverse set of periods than the second one.

As shown in Figure 8(a) and 9(a), all the task sets with utilization less than 50% are RBR-feasible under preemptive scheduling. This observation is consistent with the results of [25] which considers preemptive task sets under rate monotonic scheduling with a recovery strategy similar to ours (re-executing all the unfinished tasks), and shows that all the task sets with utilization under 50% are schedulable.

Moreover, a comparison between Figure 8(a) and 9(a) reveals that fully preemptive setting performs better when tasks in the task set have diverse rates. To understand this effect, we must notice that the longest preemption chain for a task in preemptive



(a) Fully preemptive

(b) Fully non-preemptive

(c) Non-preemptive ending intervals.

(d) Preemption thresholds.

Fig. 9: Minimum Period: 900, Maximum Period: 1000

setting, consists of the execution time of all the tasks with a higher priority. Therefore, under this scheduling strategy, tasks with low priority are the bottleneck for RBR-feasibility analysis. When the diversity of the periods is increased, lower priority tasks, on average, have much longer periods. As a result, they have a larger slack to tolerate the overhead of restarts compared to the lower priority tasks in task sets with less diverse periods. Hence, more task sets are RBR-feasible when a larger range of periods is considered.

On the contrary, when tasks have more diverse periods, non-preemptive setting performs worse (Figure 8(b) and 9(b)). This is because, with diverse periods, tasks with shorter periods (and higher priorities) experience longer blocking times due to low priority tasks with long execution times.

As the figures show, scheduling with preemption thresholds and non-preemptive intervals in both experiments yield better performance than preemptive and non-preemptive schemes. This effect is expected because the flexibility of these schemes allows them to decrease the overhead of restarts by increasing the non-preemptive regions, or by increasing the preemption thresholds while maintaining the feasibility of the task sets. Tasks under these disciplines exhibit less blocking and lower restart overhead.

Preemption thresholds and non-preemptive endings in general demonstrate comparable performance. However, in task sets with very small number of tasks (2-10 task), scheduling using non-preemptive ending intervals performs slightly better than preemption thresholds. This is due to the fact that, with small number of tasks, the granularity of the latter approach is limited because few choices can be made on the tasks' preemption thresholds. Whereas, the length of non-preemptive intervals can be selected with a finer granularity and is not impacted by the number of tasks.

VIII. Future Work and Conclusion

Software Faults: Current proposed approach does not handle software faults that modify the program logic or output of the SC and the DM at execution time. Utilizing frameworks such as ARM TrustZone [37] and limiting the access to these critical components can mitigate this issue.

Restart Time: As the restart time of the platform increases, the feasibility of the task set decreases. Therefore, the proposed solution in its current form, even though useful for many platforms, may not suit platforms with a long restart time. However, the current restart time of many platforms is not optimal, simply because reducing the reboot time of the platform has not been investigated. We are actively working on an alternative multi-stage booting solution for multi-core platforms to mitigate this problem. Our main idea is to boot one core with the bare minimum requirements to execute the SC in the shortest time possible. The SC can keep the system safe, while the real-time or general purpose OS boots on the other cores. Once the boot process is complete, the control switches to the controllers running on the OS. As a future extension, we are working on implementing

this solution on i.MX7D platform. We first boot the real-time core with FreeRTOS and SC, then boot an embedded Linux on the general purpose core and move the control to the Linux core.

Conclusion: Restarting is considered a reliable way to recover traditional computing systems from complex software faults. However, restarting safety-critical CPS is challenging. In this work we propose a restart-based fault-tolerance approach and analyze feasibility conditions under various schedulability schemes. We analyze the performance of these strategies for various task sets. Furthermore, we implemented a proof-of-concept prototype and tested it against faults in the application and RTOS. This approach enables us to provide formal safety guarantees in the presence of software faults in the application-layer as well as system-layer faults utilizing only one commercial off-the-shelf processor.

REFERENCES

[1] S. M. Sulaman, A. Orucevic-Alagic, M. Borg, K. Wnuk, M. Höst, and J. L. de la Vara, "Development of safety-critical software systems using open source software–a systematic map," in *2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications*. IEEE, 2014, pp. 17–24.

[2] L. Sha, "Dependable system upgrade," in *Real-Time Systems Symposium, 1998. Proceedings., The 19th IEEE*. IEEE, 1998, pp. 440–448.

[3] L. Sha, "Using simplicity to control complexity." IEEE Software, 2001.

[4] L. Sha, R. Rajkumar, and M. Gagliardi, "Evolving dependable real-time systems," in *Aerospace Applications Conference, 1996. Proceedings., 1996 IEEE*, vol. 1. IEEE, 1996, pp. 335–346.

[5] Y. Wang and M. Saksena, "Scheduling fixed-priority tasks with preemption threshold," in *Real-Time Computing Systems and Applications, 1999. RTCSA'99. Sixth International Conference on*. IEEE, 1999.

[6] S. Baruah, "The limited-preemption uniprocessor scheduling of sporadic task systems," in *17th Euromicro Conference on Real-Time Systems (ECRTS'05)*, July 2005, pp. 137–144.

[7] D. Seto and L. Sha, "An engineering method for safety region development," 1999.

[8] T. L. Crenshaw, E. Gunter, C. L. Robinson, L. Sha, and P. Kumar, "The simplex reference model: Limiting fault-propagation due to unreliable components in cyber-physical system architectures," in *Real-Time Systems Symposium, 2007. RTSS 2007. 28th IEEE International*. IEEE, 2007.

[9] S. Bak, D. K. Chivukula, O. Adekunle, M. Sun, M. Caccamo, and L. Sha, "The system-level simplex architecture for improved real-time embedded system safety," in *Real-Time and Embedded Technology and Applications Symposium, 2009. RTAS 2009. 15th IEEE*. IEEE, 2009, pp. 99–107.

[10] F. Abdi, R. Mancuso, S. Bak, O. Dantsker, and M. Caccamo, "Reset-based recovery for real-time cyber-physical systems with temporal safety constraints," in *IEEE 21st Conference on Emerging Technologies Factory Automation (ETFA 2016)*, 2016.

[11] S. Mohan, S. Bak, E. Betti, H. Yun, L. Sha, and M. Caccamo, "S3a: Secure system simplex architecture for enhanced security and robustness of cyber-physical systems," in *Proceedings of the 2nd ACM international conference on High confidence networked systems*. ACM, 2013.

[12] G. Candea and A. Fox, "Recursive restartability: Turning the reboot sledgehammer into a scalpel," in *Hot Topics in Operating Systems, 2001. Proceedings of the Eighth Workshop on*. IEEE, 2001, pp. 125–130.

[13] G. Candea and A. Fox, "Crash-only software," in *HotOS IX: The 9th Workshop on Hot Topics in Operating Systems*, 2003, pp. 67–72.

[14] G. Candea, E. Kiciman, S. Zhang, P. Keyani, and A. Fox, "Jagr: An autonomous self-recovering application server," in *Autonomic Computing Workshop. 2003. Proceedings of the*. IEEE, 2003, pp. 168–177.

[15] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox, "Microreboot- a technique for cheap recovery," in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, ser. OSDI'04, 2004, pp. 3–3.

[16] K. Vaidyanathan and K. S. Trivedi, "A comprehensive model for software rejuvenation," *Dependable and Secure Computing, IEEE Transactions on*, vol. 2, no. 2, pp. 124–137, 2005.

[17] S. Garg, A. Puliafito, M. Telek, and K. S. Trivedi, "Analysis of software rejuvenation using markov regenerative stochastic petri net," in *Software Reliability Engineering, 1995. Proceedings., Sixth International Symposium on*. IEEE, 1995, pp. 180–187.

[18] Y. Huang, C. Kintala, N. Kolettis, and N. D. Fulton, "Software rejuvenation: Analysis, module and applications," in *Fault-Tolerant Computing, 1995. FTCS-25. Digest of Papers., Twenty-Fifth International Symposium on*. IEEE, 1995, pp. 381–390.

[19] S. Ghosh, R. Melhem, and D. Mosse, "Enhancing real-time schedules to tolerate transient faults," in *Proceedings 16th IEEE Real-Time Systems Symposium*, Dec 1995, pp. 120–129.

[20] F. Liberato, R. Melhem, and D. Mosse, "Tolerance to multiple transient faults for aperiodic tasks in hard real-time systems," *IEEE Transactions on Computers*, vol. 49, no. 9, pp. 906–914, Sep 2000.

[21] S. Punnekkat, A. Burns, and R. Davis, "Analysis of checkpointing for real-time systems," *Real-Time Systems*, vol. 20, no. 1, pp. 83–102, 2001.

[22] G. Lima and A. Burns, *Scheduling Fixed-Priority Hard Real-Time Tasks in the Presence of Faults*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 154–173.

[23] R. M. Pathan and J. Jonsson, "Exact fault-tolerant feasibility analysis of fixed-priority real-time tasks," in *2010 IEEE 16th International Conference on Embedded and Real-Time Computing Systems and Applications*, Aug 2010, pp. 265–274.

[24] C.-C. Han, K. G. Shin, and J. Wu, "A fault-tolerant scheduling algorithm for real-time periodic tasks with possible software faults," *IEEE Transactions on Computers*, vol. 52, no. 3, pp. 362–372, March 2003.

[25] 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, Oct 1998.

[26] M. A. Haque, H. Aydin, and D. Zhu, "Real-time scheduling under fault bursts with multiple recovery strategy," in *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2014.

[27] A. Thekkilakattil, R. Dobrin, S. Punnekkat, and H. Aysan, "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*, ser. RTNS '12. New York, NY, USA: ACM, 2012.

[28] H. Kopetz and G. Bauer, "The time-triggered architecture," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 112–126, 2003.

[29] H. Kopetz, *On the fault hypothesis for a safety-critical real-time system*. Springer, 2004.

[30] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM (JACM)*, vol. 20, no. 1, pp. 46–61, 1973.

[31] R. I. Davis, A. Burns, R. J. Bril, and J. J. Lukkien, "Controller area network (can) schedulability analysis: Refuted, revisited and revised," *Real-Time Systems*, vol. 35, no. 3, pp. 239–272, 2007.

[32] R. J. Bril, J. J. Lukkien, and W. F. J. Verhaegh, "Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption revisited," in *19th Euromicro Conference on Real-Time Systems (ECRTS'07)*, July 2007, pp. 269–279.

[33] G. C. Buttazzo, M. Bertogna, and G. Yao, "Limited preemptive scheduling for real-time systems. a survey," *IEEE Transactions on Industrial Informatics*, vol. 9, no. 1, pp. 3–15, Feb 2013.

[34] Quanser Inc., "3 dof helicopter," http://www.quanser.com/products/3dof_helicopter, accessed: September 2016.

[35] S. Bak, T. T. Johnson, M. Caccamo, and L. Sha, "Real-time reachability for verified simplex design," in *Real-Time Systems Symposium (RTSS), 2014 IEEE*. IEEE, 2014, pp. 138–148.

[36] Quanser Inc., "Q8 data acquisition board," http://www.quanser.com/products/q8, accessed: September 2016.

[37] ARM Inc., "Arm trustzone," https://www.arm.com/products/security-on-arm/trustzone, accessed: September 2016.