2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)

Job-Class-Level Fixed Priority Scheduling of Weakly-Hard Real-Time Systems

Hyunjong Choi[†], Hyoseung Kim[†], and Qi Zhu[‡]
[†]*University of California, Riverside*, [‡]*Northwestern University*
{hchoi036, hyoseung}@ucr.edu, qzhu@northwestern.edu

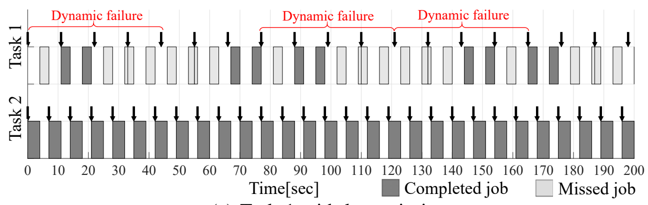
***Abstract*—Many cyber-physical applications including sensing and control operations can tolerate a certain degree of timing violations as long as the number of the violations are predictably bounded. The notion of *weakly-hard* real-time systems has been studied to capture this effect, but existing work reveals limitations for practical use due the restrictions imposed on timing model and the high complexity of analysis. In this paper, we propose a new *job-class-level fixed-priority preemptive scheduler* and its schedulability analysis framework for sporadic tasks with weakly-hard real-time constraints. Our proposed scheduler employs the *meet-oriented classification* of jobs of a task in order to reduce the worst-case temporal interference imposed on other tasks. Under this approach, each job is associated with a "job-class" that is determined by the number of deadlines previously met (with a bounded number of consecutively-missed deadlines). This approach also allows decomposing the complex weakly-hard schedulability problem into two sub-problems that are easier to solve: (1) analyzing the response time of a job with each job-class, which can be done by an extension of the existing task-level analysis, and (2) finding possible job-class patterns, which can be modeled as a simple reachability tree. Experimental results indicate that our scheduler outperforms prior work in terms of task schedulability and analysis time complexity. We have also implemented a prototype of a job-class-level scheduler in the Linux kernel running on Raspberry Pi with acceptably-small runtime overhead.**

I. Introduction

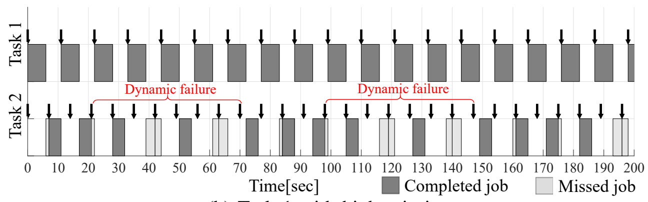
The performance and stability of embedded and cyber-physical applications depends not only on the precision of computation but also on the physical instant at which the output is generated [23, 24]. Since Liu and Layland's seminal work [28], real-time systems with hard deadlines have been extensively studied and have shown their effectiveness in satisfying all deadlines under any circumstance. However, in practical systems, there are many components that are tolerant to some deadline misses without affecting their functional correctness, if the number of misses is predictably controlled and bounded. This observation has motivated the development of *weakly-hard real-time systems* to improve resource efficiency [5], and this paper focuses on the scheduling problem of tasks with weakly-hard constraints.

The common notation of a weakly-hard constraint is the (m, K) form, which specifies that among any K consecutive task instances (jobs), at most m instances can miss their deadlines. Prior work with predictable (m, K) guarantees [5, 6, 14, 18, 20, 38] has focused on reducing the pessimism of schedulability analysis under traditional task-level fixed-priority scheduling, such as Rate Monotonic (RM) [28], by making strong assumptions on task timing behavior, e.g., fixed phasing (initial release offset) and fixed period with no release jitter. However, we believe that such assumptions limit their applicability to recent cyber-physical systems that require flexibility and adaptability. The high complexity of the existing analysis also makes it difficult to use in runtime admission control, which is required by systems running in a changing environment.

Moreover, we find that task-level fixed-priority scheduling cannot take full advantage of m permitted deadline misses in K consecutive job executions. As an example, let us consider a uniprocessor system with two periodic tasks. Task 1 has $(m, K) = (2, 4)$ with period of 11 and execution time of 6 time units. Task 2 has $(4, 7)$ with period of 7 and execution time of 4 units. Hence, Task 1 may miss up to 2 deadlines in 4 consecutive jobs; Task 2 may miss up to 4 in 7 jobs. As the total utilization of the two tasks exceeds 1, they cannot meet their deadlines all the time. However, there *may* exist a feasible weakly-hard schedule as the minimum utilization demand to meet the weakly-hard constraints is only $(6/11) \cdot (2/4) + (4/7) \cdot (3/7) \approx 0.52$.



(a) Task 1 with low priority



(b) Task 1 with high priority

Fig. 1: Dynamic failures under conventional task-level fixed-priority scheduling

Under task-level fixed-priority scheduling, only two priority assignments can be found for the above taskset: (1) low priority to Task 1 and high priority to Task 2, which is also obtainable by RM, and (2) high to Task 1 and low to Task 2. Fig. 1 illustrates task scheduling timeline with these two priority assignments. In both cases, the taskset falls into

978-1-7281-0678-6/19/$31.00 ©2019 IEEE
DOI 10.1109/RTAS.2019.00028

a *dynamic failure* [17], where a task experiences more than m deadline misses in a window of K jobs. Thus, we can conclude that despite the very low minimum utilization demand of this taskset, none of the task-level fixed-priority assignments yields a feasible schedule.

In this paper, we take a completely different approach that significantly improves the scheduling efficiency and flexibility of weakly-hard real-time tasks. We propose a new scheduling policy, called *job-class-level fixed-priority preemptive scheduling*, for periodic and sporadic tasks with (m, K) constraints. This scheduler supports arbitrary initial offsets and non-zero release jitters. The running time of the resulting schedulability analysis is much shorter than that of the latest work [38] that uses mixed integer linear programming. The key to our scheduler lies in the classification of the jobs of each task and the assignment of fixed priority to each class of jobs. We will show in Section IV that the proposed job-class-level scheduling can successfully schedule the above taskset example.

Contributions. This paper makes the following contributions:

- We propose a new job-class-level fixed-priority scheduler for sporadic real-time tasks with weakly-hard constraints. Specifically, our scheduler is based on the *meet-oriented classification* of jobs of tasks, which effectively reduces the worst-case temporal interference imposed on other weakly-hard tasks.
- We present a schedulability analysis framework for weakly-hard tasks under our scheduler. It consists of two steps: (1) analyzing the response time of a job with each job-class, which is done by an extension of the existing task-level analysis, and (2) finding possible job-class patterns of a task, which is solved by constructing a reachability tree based on the response time of individual job-classes.
- We find the schedulability analysis of tasks with $\frac{m}{K} \geq 0.5$ can be tested by using a single sufficient condition, without exploring all possible execution patterns. In other words, the time complexity of analyzing such tasks is much smaller than that for tasks with $\frac{m}{K} < 0.5$. We show this property with analytical and empirical results.
- We prove that our proposed job-class-level fixed priority scheduler is a generalization of task-level fixed-priority scheduling. We also show that our scheduling framework can be used to upper bound the number of consecutive deadline misses.
- Experimental results demonstrate that the proposed scheduler outperforms the prior work [25, 38] in both task schedulability and analysis running time. In addition, we have implemented our scheduler in the Linux kernel running on Raspberry Pi with acceptably-small runtime overhead.

Organization. The rest of the paper is organized as follows. Section II discusses prior work on weakly-hard systems. The task model and notation used in this paper are presented in Section III. We then propose job-class-level scheduling in Section IV, and present our schedulability analysis in Section V. The evaluation of our work is given in Section VI. Section VII concludes the paper and discusses future work.

II. Related Work

The notion of (m, K) constraints was first introduced by Hamdaoui et al. [17]. They focused on reducing the probability of timing violations with dynamic priority assignment, and showed its positive impact in simulation. However, no analytical bound on the number of deadline misses was given. Bernat et al. [5] formally defined weakly-hard real-time systems, and proposed the first work on the schedulability of periodic tasks with weakly-hard constraints under fixed-priority scheduling. Extensions of this schedulability work have been studied, such as for bi-modal execution [6, 35] and non-preemptive tasks [27]. The former ones, however, assume that the initial release offset of each task is statically fixed and exactly known ahead of time, which is not always possible especially in an open system. The latter assumes that all jobs have zero release jitter. Recent work [38] relaxed these assumptions but at the expense of high analysis complexity, e.g., taking more than 10 minutes for 20 tasks on an Intel Xeon 8-core processor, which makes it difficult to use for online admission control in embedded platforms. Goossens [15] presented an exact schedulability test for periodic tasks with zero jitter and offset under distance-based dynamic-priority scheduling. This is done by thoroughly enumerating all task schedules over multiple hyperperiods and checking if there is a repeated feasible schedule.

Weakly-hard constraints have also been studied to bound the temporal violations of overloaded systems [19, 33, 34, 41]. They use typical worst-case analysis (TWCA), which assumes the exact arriving patterns of task instances with occasional overloads are given in the form of arrival curves. Variants of TWCA have been studied for CAN bus analysis [32], tasks with dependencies [18], tasks with varying execution time [1, 39], budget assignment [20]. While these TWCA-based approaches made significant contributions to weakly-hard systems, the precise identification and description of task arrival patterns is much harder than the use of periodic [28] or sporadic task models [30], as discussed in [38].

Besides schedulability, preserving control stability has been studied in the context of weakly-hard systems. Rainer et al. [8] used weakly-hard constraints to capture the failure of unstable feedback control systems in a deterministic way. In [31], a state-based methodoldy is presented to analyze the performance of a control application using weakly-hard constraints. Frehse et al. [12] analyzed the closed-loop properties of control software based on TWCA. In [35], periodic task instances are classified into mandatory and optional ones based on (m, K) constraints, and only the mandatory ones are guaranteed to complete in time. Gaid et al. [13] and Marti et al. [29] extended this work to consider optional instances and non-periodic execution, respectively. The (m, k) model has been further investigated for control-schedule co-design [10, 11, 36].

The recent work of Lee and Shin [26] focused on bounding consecutive deadline misses of periodic tasks for cyber-physical systems. They classified each job of tasks based on the number of consecutive deadline misses happened just before its arrival, and associated this number with control stability. While this job-level classification has inspired our work, there are two major differences. First, our work focuses on the (m, K) model which is a superset of the model used in [26]. Specifically, x consecutive deadline misses are a special case of the weakly-hard constraint $(x, x+1)$ in the (m, K) form and thus can be safely bounded by our work. Second, our work uses a meet-oriented classification which will be detailed in Section IV.

III. System Model

This paper considers a uniprocessor system where the CPU runs at a fixed clock frequency. The system executes a taskset consisting of N periodic or sporadic real-time tasks with constrained deadlines.

Task model. Each task τ_i is characterized as follows:

$$\tau_i := (C_i, D_i, T_i, (m_i, K_i))$$

- C_i: The worst-case execution time of each job of a task τ_i.
- D_i: The relative deadline of each job of τ_i ($D_i \leq T_i$)
- T_i: The minimum inter-arrival time between consecutive jobs of τ_i. If τ_i is a periodic task, T_i is the period of τ_i.
- (m_i, K_i): The weakly-hard constraints of τ_i ($m_i < K_i$). If τ_i is a hard real-time task, $m_i = 0$ and $K_i = 1$.

Each task τ_i can have a non-zero initial release offset o_i, and a release jitter $\mathcal{J}_i$ ($\mathcal{J}_i \leq D_i - C_i$). The j-th job of a task τ_i is denoted as $J_{i,j}$.

Utilization. To represent the resource usage and the effective performance of a weakly-hard system, we define a set of utilization metrics below.

Def. 1. *The maximum utilization of a task τ_i, U_i^M, is the maximum amount of CPU resource that τ_i can utilize. It is defined as $U_i^M = \frac{C_i}{T_i}$, which is in fact the same as the common task utilization. The maximum total utilization is defined as the sum of the maximum utilization of all tasks, i.e., $U^M = \sum_{i=1}^{N} \frac{C_i}{T_i}$, where N is the number of tasks.*

Note that the maximum utilization U_i^M is the value that τ_i can achieve when it always meets the deadline.

Def. 2. *The minimum utilization of a task τ_i, U_i^m, is the CPU resource used by τ_i when it experiences the maximum deadline misses allowed by its (m_i, K_i) constraint, i.e., $U_i^m = \frac{C_i}{T_i} \times \frac{K_i - m_i}{K_i}$. The minimum total utilization is defined as $U^m = \sum_{i=1}^{N} \frac{C_i}{T_i} \times \frac{K_i - m_i}{K_i}$.*

Each task requires at least U_i^m of CPU resource to be schedulable w.r.t. the weakly-hard constraint. Note that the minimum total utilization should be less than or equal to 1 for a taskset to be schedulable, i.e., $U^m \leq 1$.

Deadline-missed Jobs. If a job of a task misses its deadline, there are two approaches to handle this job: (i) letting it continue to execute beyond the deadline, and (ii) dropping (or descheduling) it immediately. If the output of a job has some usefulness even after the deadline, the first approach can be considered better than the second one. Otherwise, the second approach is more appealing as it can prevent the deadline-missed job blocking its next job and unnecessarily wasting CPU cycles. Therefore, this paper uses the second approach and shows that it can be implemented on embedded platforms with small overhead. Note that job dropping has also been used in other real-time contexts, e.g., shared resource access [2, 4] and mixed-criticality systems [16].

IV. Job-class-level Fixed-Priority Scheduling

Unlike task-level fixed-priority scheduling, our work classifies the jobs of each task into *job-classes* and assigns priorities to individual job-classes. With this approach, a task can have as many priority levels as the number of job-classes it has, and the priority of each job is determined by the priority of its corresponding job-class.

A. Meet-oriented job classification

Bernat et al. [5] discussed that weakly-hard constraints can be categorized into four types based on the following criteria: consecutive vs. any order, and met vs. missed deadlines. In line with this idea, one may consider the following four classification approaches for a job $J_{i,j}$ based on the execution results of its prior jobs: the number of previous deadlines (i) met, (ii) missed, (iii) consecutively met, and (iv) consecutively missed. In this paper, we specifically use a *meet-oriented classification* to define a job-class.

Def. 3. *A job-class J_i^q includes a job whose nearest previous jobs have consecutively met q deadlines, where q has the range of $[0,\ K_i - m_i]$ and $m_i \geq 1$.*

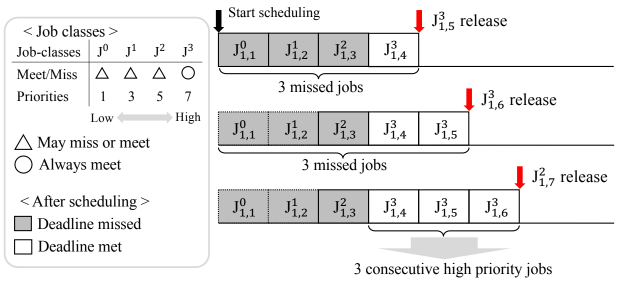
The superscript of J_i^q is referred to as the index of that job-class. Due to the range of job-class indices, any job that follows more than $K_i - m_i$ consecutively-met deadlines belongs to a job-class $J_i^{K_i - m_i}$. If $m_i = 0$, meaning that no deadline miss is allowed, i.e., a hard real-time task, the number of job-classes for that task is always one. Note that a job-class is determined by the number of *nearest* deadlines consecutively met. If a job follows two distinct intervals of q and q' consecutively-met deadlines and q is the more recent one, this job gets the job-class index of q. More precisely, given the k-th job of a task τ_i, $J_{i,k}$, its nearest previous jobs with q consecutively met deadlines are $J_{i,x\ldots y}$, where $x \leq y < k$ and there is no other job $J_{i,z} : y < z < k$ that has met the deadline.

Each job-class J_i^q is assigned a fixed priority, which is denoted as π_i^q. Since the job-class index of a job indicates the number of consecutive deadlines met just before its arrival, the higher index likely means the less urgent the job is (w.r.t. weakly-hard constraints). Therefore, we propose that the priority of a job-class decreases monotonically as a job-class index increases. For instance, a task τ_1 with $(m_1, K_1) = (2, 4)$ can have three job-classes, J_1^0, J_1^1, and J_1^2, with their priorities of 6, 4, and 2, respectively. More details on job-class priority assignment will be given in Section IV-B.

To better represent a sequence of job execution results, we introduce the following two patterns: *μ-pattern* and *$\mathcal{C}$-pattern*.

Def. 4 ([5, 17])**.** *A μ-pattern represents a sequence of deadline met and missed jobs of a task. For example,* `MMmMMmMM`*, where* `M` *and* `m` *represent a met and a missed deadline, respectively.*

Def. 5. *A $\mathcal{C}$-pattern represents job-class indices of a sequence of jobs released by a task with a weakly-hard constraint. For example,* 0120101*, where each digit means a job-class index used.*[1].



< Job classes >
Job-classes J^0 J^1 J^2 J^3
Meet/Miss △ △ △ ○
Priorities 1 3 5 7
Low High
△ May miss or meet
○ Always meet
< After scheduling >
▨ Deadline missed
☐ Deadline met
Start scheduling
$J^3_{1,5}$ release
$J^0_{1,1}$ $J^1_{1,2}$ $J^2_{1,3}$ $J^3_{1,4}$
3 missed jobs
$J^3_{1,6}$ release
$J^0_{1,1}$ $J^1_{1,2}$ $J^2_{1,3}$ $J^3_{1,4}$ $J^3_{1,5}$
3 missed jobs
$J^2_{1,7}$ release
$J^0_{1,1}$ $J^1_{1,2}$ $J^2_{1,3}$ $J^3_{1,4}$ $J^3_{1,5}$ $J^3_{1,6}$
3 consecutive high priority jobs

Fig. 2: Consecutive execution of high-priority jobs under *miss-oriented* job classification

Benefits of meet-oriented classification. The rationale behind the meet-oriented job classification is that it can reduce the worst-case interference imposed on lower-priority jobs by modulating the execution of jobs with high priority. For comparison, let us consider the opposite approach where a job-class is defined by the number of deadline misses. In this *miss-oriented* approach, a job-class with a higher index means more deadlines missed previously and thus gets a higher priority. Fig. 2 shows an example of a task τ_1 with $(m_1, K_1) = (3, 6)$ under the miss-oriented approach. On the left side of the figure, a circle indicates a job-class that always meets the deadline due to its high priority, and a triangle indicates a job-class that does not guarantee meeting the deadline. The task τ_1 has 4 job-classes and the highest-index job-class, J^3, has the highest priority.

The right side of Fig. 2 illustrates a possible scheduling result of the task τ_1. Assume that the first three jobs of τ_i, $J^0_{1,1}$, $J^1_{1,2}$ and $J^2_{1,3}$, miss the deadline due to the low priorities of their job-classes. Then, the 4th, 5th and 6th jobs of τ_1 get the highest job-class index, i.e., $J^3_{1,4}$, $J^3_{1,5}$ and $J^3_{1,6}$, because they have three missed deadlines in the window of $K_1 = 6$ prior jobs. This results in three consecutive execution of the highest-priority jobs of τ_1, thereby resulting in consecutive interference to the lower-priority jobs of other tasks. Under the miss-oriented classification, it is very hard (or may be impossible) to avoid such interference. On the other hand, in our *meet-oriented* approach, such consecutive execution of the highest-priority jobs is effectively prevented, because once the highest-priority job meets the deadline, its next job will be assigned a job-class with a lower priority. Hence, the time interval between highest-priority jobs becomes longer and other tasks likely experience less interference during their job execution.

[1]If there is a job-class index greater than 9, one can use a delimiter between each index, e.g., $0.1 \cdots 11.12.13$.

Bounding consecutive deadline misses. The above Def. 3 does not specify a *distance* from the current job to the nearest previous deadline-met jobs. If we limit this distance to zero, only immediately-previous jobs will be checked. For example, for a job-class J^q_i, there will be no deadline miss allowed between the current job and the window of q prior jobs; if a job $J_{i,j}$ misses its deadline, the job-class index of its next job $J_{i,j+1}$ will be immediately demoted to zero, i.e., $J^0_{i,j+1}$. If we do not limit the distance, an unbounded number of consecutive deadline misses will be allowed at each job-class. For example, if a job of τ_i gets a job-class of J^q_i and its subsequent jobs continuously miss the deadline, they all will belong to the same job-class. Therefore, we define a *miss threshold* to limit the distance and bound the number of consecutive deadline misses at each job-class.

Def. 6. *A miss threshold w_i is the maximum number of consecutive deadline misses that a task τ_i can experience at any job-class J^q_i with $q > 0$. If a job of τ_i follows w_i consecutive deadline misses, this job is assigned the lowest-index job-class J^0_i. The value of w_i is given by:*

$$w_i = \max\left(\left\lfloor \frac{K_i}{K_i - m_i} \right\rfloor - 1, 1\right)$$

The reasoning behind this w_i value is to ensure that a task τ_i can have an enough number of jobs running with τ_i's highest-priority job-class, J^0_i. By the definition of the miss threshold, w_i cannot be smaller than 1, and as w_i goes larger, it takes more periods for τ_i to regain J^0_i. The way w_i is determined allows τ_i to run at least $K_i - m_i$ jobs with J^0_i (denominator in the equation) during K_i consecutive invocations (numerator) when the other m_i jobs all miss their deadlines.

Example. A task τ_i with a weakly-hard constraint $(m_i, K_i) = (5, 7)$ is assigned 3 job-classes and a miss threshold size of $w_i = 2$, based on Defs. 3 and 6. Suppose that the very first two jobs of τ_i have met their deadlines (τ_i's μ-pattern is `MM`). As a result, the job-class index of the 3rd job becomes 2, i.e., $J^2_{i,3}$. If $J^2_{i,3}$ misses the deadline (`MMm`), the 4th job continues to get $J^2_{i,4}$ as the number of consecutive misses is less than the miss threshold ($w_i = 2$). If $J^2_{i,4}$ also misses the deadline (`MMmm`), the task τ_i has reached the miss threshold and the 5th job of τ_i is assigned $J^0_{i,5}$. On the other hand, if $J^2_{i,4}$ meets the deadline (`MMmM`), the 5th job is assigned $J^1_{i,5}$ because the number of nearest consecutively-met deadlines is 1 (as given by Def. 3).

Relations to other scheduling approaches. Our job-class-level fixed-priority scheduling model is a generalization of the conventional task-level fixed-priority scheduling [28] and can represent the temporal constraints of the CFP scheduling model [26][2] that upper bounds the number of consecutive

[2]CFP stands for Cyber subsystem's state-level Fixed Priority (scheduling).

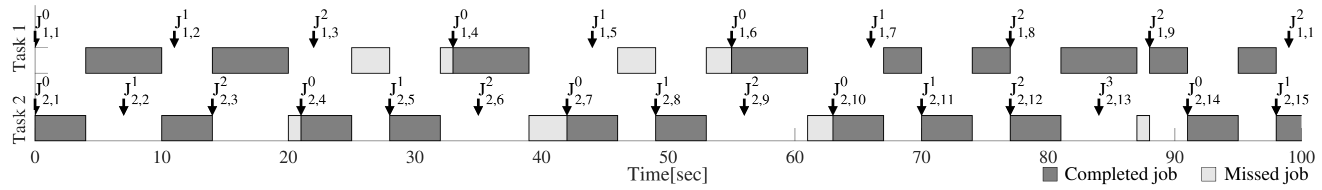


Fig. 3: Execution timeline of weakly-hard tasks under job-class-level scheduling

Algorithm 1 Job-class priority assignment

```
Input: Γ: Taskset
 1: $N \leftarrow |\Gamma|$
 2: Sort $\tau_i$ in $\Gamma$ in ascending order of deadline
 3: for all $\tau_i \in \Gamma$ do
 4:     $l_i \leftarrow K_i - m_i + 1$          ▷ $l_i$: number of job-classes for $\tau_i$
 5: end for
 6: $prio \leftarrow \sum_{\tau_i \in \Gamma} l_i$          ▷ Priority to be assigned next
 7: if $\Gamma$ is schedulable by DM then
 8:     for all $\tau_i \in \Gamma$ do
 9:         ▷ Assign the same priority to all job-classes of $\tau_i$
10:         for all $q \leftarrow 0$ to $l_i - 1$ do
11:             $\pi_i^q \leftarrow prio$
12:         end for
13:         $prio \leftarrow prio - 1$
14:     end for
15: else
16:     $L \leftarrow \max_{\tau_i \in \Gamma} l_i$
17:     for $q \leftarrow 0$ to $L - 1$ do
18:         if $q > 0$ then
19:             Sort $\tau_i \in \Gamma$ in ascending order of $w_i$ and deadline
20:         end if
21:         for all $\tau_i \in \Gamma$ do
22:             if $q < l_i$ then          ▷ Check if $q$ is a valid index
23:                 $\pi_i^q \leftarrow prio$
24:                 $prio \leftarrow prio - 1$
25:             end if
26:         end for
27:     end for
28: end if
```

deadline misses of periodic tasks.

Lemma 1. *The job-class-level fixed-priority scheduling subsumes the task-level fixed-priority scheduling.*

Proof: If we assign the same priority to all jobs of τ_i (e.g., $\forall q : 0 < q \leq K_i - m_i, \pi_i^q = \pi_i^0$), the job-class-level scheduling can yield the same task schedule as the task-level fixed-priority scheduling. ■

Lemma 2. *The task model of the job-class-level fixed-priority scheduling subsumes that of the CFP scheduling [26].*

Proof: The task model of the CFP scheduling uses a single parameter m' for each task, which means at most m' consecutive deadline misses are allowed for that task. If the (m, K) constraint of a task under the job-class-level scheduling is set to $(m', m' + 1)$, it represents the maximum of m' deadline misses permitted in any $m' + 1$ consecutive periods, which captures the case for m' consecutive deadline misses. Therefore, the job-class-level scheduling can represent any constraint imposed by the CFP scheduling model. ■

B. *Priority assignment to job-classes*

Priority assignment can affect the overall performance of the job-class-level scheduler. An optimal priority assignment can be found by a brute-force method, but due to its extremely high computational complexity, it is not practically usable. Therefore, we propose a heuristic priority assignment scheme given in Alg. 1.

The proposed algorithm is an extension of the deadline-monotonic (DM) priority assignment policy [3]. The algorithm first checks the schedulability of a given taskset Γ under the task-level DM assignment (line 7), which can be done by the conventional iterative response-time test [22]. If the taskset is schedulable by DM, the algorithm simply follows the task-level DM assignment. Hence, a task with a shorter deadline is assigned a higher priority and all job-classes of each task gets the same priority (lines 8-14). If the taskset is not schedulable by DM, the algorithm assigns priorities to individual job classes. Basically, it has two-level iterations. The outer loop (line 17) iterates over job-class indices from 0 to L, where L is the maximum number of job-classes for each task. The inner loop (line 21) iterates over all tasks in ascending order of deadlines when $q = 0$ (sorted in line 2) and in ascending order of miss thresholds (w_i) with deadlines for tie-breaking when $q > 0$ (line 19). Hence, the job-class priorities assigned by the algorithm have the following properties: if $D_i < D_j$, $\pi_i^0 > \pi_j^0$, and if $w_i < w_j$, $\pi_i^q > \pi_j^q$ for $q > 0$.

Lemma 3. *The proposed job-class-level priority assignment algorithm subsumes the task-level DM priority assignment.*

Proof: Obvious as shown by lines 8-14 of Alg. 1. ■

C. *Example of job-class-level scheduling*

Recall the example taskset of Fig. 1 which is unschedulable by any task-level fixed-priority scheduling, as discussed in Section I. Table I gives the detailed parameters of this taskset. However, the proposed job-class-level scheduling and priority assignment schemes satisfy the weakly-hard constraints of this taskset. Fig. 3 illustrates the execution timeline of the taskset under our scheduler, and Fig. 4 depicts the priority changes of the two tasks of this taskset. As can be seen, each task uses the priority levels of its all job-classes and the relative priority ordering of the tasks changes over time.

V. Schedulability Analysis

This section presents the schedulability analysis of tasks with weakly-hard constraints under job-class-level scheduling. Our analysis consists of two parts: analyzing the response time of a job with each job-class, and finding job-class patterns that

TABLE I: Task set

	Specifications
Task 1	$\tau_1 : (C_1 = 6, T_1 = 11, m_1 = 2, K_1 = 4)$
Task 2	$\tau_2 : (C_2 = 4, T_2 = 7, m_2 = 4, K_2 = 7)$

	Priority
Task 1	$\pi_1^0 = 6, \pi_1^1 = 4, \pi_1^2 = 2$
Task 2	$\pi_2^0 = 7, \pi_2^1 = 5, \pi_2^2 = 3, \pi_2^3 = 1$

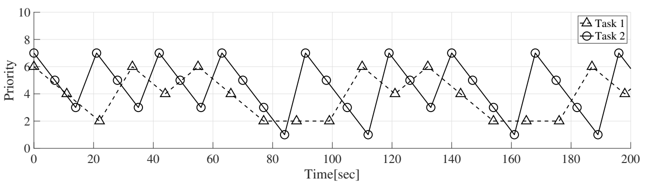


Fig. 4: Priority changes under job-class-level scheduling

can possibly happen at runtime. We will describe each part and explain how the schedulability test is done.

A. Minimum job-class inter-arrival time

In order to analyze the worst-case response time (WCRT) of a job with a specific job-class, we need to upper bound the maximum interference imposed by the jobs of other tasks with higher-priority job-classes. Such interference can be captured by finding the minimum arrival time between any two jobs of the same job-class. Hence, we begin with analyzing this interval and call it the minimum job-class inter-arrival time.

We first analyze the minimum inter-arrival time of a job-class J_i^q whose index q is the highest index of a task τ_i, i.e., $q = K_i - m_i$.

Lemma 4. *The minimum inter-arrival time of a job-class J_i^q where $q = K_i - m_i$ is given by:*

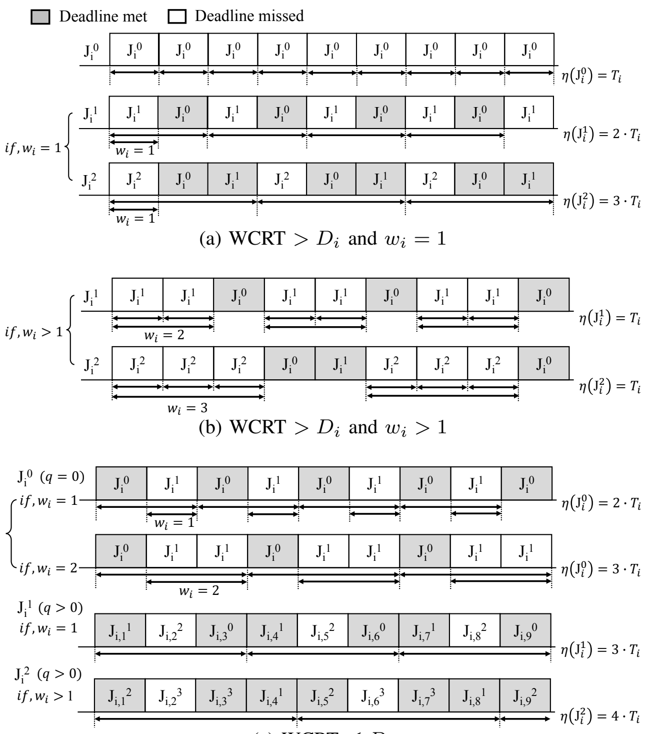
$$\eta(J_i^q) = 1 \cdot T_i$$

Proof: If the WCRT of J_i^q is less than or equal to its relative deadline D_i, the job-class index of the next released job is always $q+1$. However, since q is the highest index of τ_i, the next released job maintains the current index even if the job meets its deadline. Thus, the minimum time for τ_i to regain $J_i^{K_i - m_i}$ is $1 \cdot T_i$. If the WCRT of J_i^q is greater than D_i, the job may or may not meet the deadline when it is scheduled at runtime. If the job meets the deadline, the minimum time to regain $J_i^{K_i - m_i}$ is the same as the case when the WCRT $\leq D_i$. If the job misses the deadline, the next job may get J_i^0, which causes longer time to regain J_i^q. Therefore, in the worst case, $\eta(J_i^q)$ is $1 \cdot T_i$. ■

Lemma 5. *The minimum inter-arrival time of J_i^q where $q < K_i - m_i$ and the WCRT of J_i^q is greater than D_i is given by:*

$$\eta(J_i^q) = \begin{cases} (q+1) \cdot T_i & \text{, if } w_i = 1 \\ 1 \cdot T_i & \text{, if } w_i > 1 \end{cases}$$

Proof: The proof is done by contradiction. Assume that when $w_i = 1$, the minimum inter-arrival time of J_i^q is less



(a) WCRT $> D_i$ and $w_i = 1$

(b) WCRT $> D_i$ and $w_i > 1$

(c) WCRT $\leq D_i$

Fig. 5: The minimum time interval between any two jobs of the same job-class

than $(q+1) \cdot T_i$. Since the WCRT of J_i^q is greater than D_i, the job-class index q' of the next released job can be either 0 or $q+1$ at runtime. If $q' = 0$, by Def. 3, at least q subsequent jobs should meet their deadlines in order to get the job-class index of q again, giving $(q+1) \cdot T_i$ as the minimum inter-arrival time. If $q' = q+1$, it requires at least $q+1$ subsequent jobs (1 miss and q meets) to regain the job-class index q, resulting in $(q+2) \cdot T_i$. These contradict the assumption. Hence, the inter-arrival time of J_i^q is greater than or equal to $(q+1) \cdot T_i$ when $w_i = 1$. When $w_i > 1$, jobs can continue to have J_i^q as long as w_i permits, and thus the minimum inter-arrival time of J_i^q is $1 \cdot T_i$. The examples of these two cases, $w_i = 1$ and $w_i > 1$, are illustrated in Fig. 5a and 5b, respectively. ■

Lemma 6. *The minimum inter-arrival time of J_i^q where $q < K_i - m_i$ and the WCRT of J_i^q is less than or equal to D_i is:*

$$\eta(J_i^q) = \begin{cases} (w_i + 1) \cdot T_i & \text{, if } q = 0 \\ (q+2) \cdot T_i & \text{, if } q > 0 \end{cases}$$

Proof: If $q = 0$, the proof is done by contradiction. Assume that the minimum inter-arrival time of $J_i^{q=0}$ is less than $(w_i + 1) \cdot T_i$. Since the job-class index q' of the next job is always $q + 1 = 1$ by Def. 3 ($\because$ WCRT of $J_i^q \leq D_i$), the inter-arrival time of J_i^0 is at least $2 \cdot T_i$. This contradicts the assumption because $w_i \geq 1$ by Def. 6. Therefore, the inter-arrival time of $J_i^{q=0}$ is greater than or equal to $(w_i + 1) \cdot T_i$. If $q > 0$, the minimum time for τ_i to get a job-class J_i^0 is

$2 \cdot T_i$ because the next job $J_{i,k+1}$ always gets the job-class index of $q+1$ by Def. 3 and only the second next job $J_{i,k+2}$ may be able to get the index 0. Then, at least $q \cdot T_i$ is required for τ_i to get the job-class index q. Therefore, the minimum inter-arrival time of J_i^q for $q > 0$ is $(q+2) \cdot T_i$. Fig. 5c shows the examples for both cases. ■

B. Worst-case response time of job-classes

In order to capture the worst-case temporal interference from other tasks, we exploit the notion of the minimum job-class inter-arrival time given by Lemmas 4, 5 and 6.

Lemma 7. *The worst-case temporal interference imposed on a job-class J_i^q by the higher-priority jobs of another task τ_k during an arbitrary time window t, which starts with the release of J_i^q, is upper-bounded by:*

$$W_i^q(t, \tau_k) = \begin{cases} 0 & , \text{if } \nexists p : \pi_i^q < \pi_k^p \\ \min\Big(\sum_{\forall p: \pi_i^q < \pi_k^p} \left\lceil \frac{t + \mathcal{J}_k}{\eta(J_k^p)} \right\rceil \cdot C_k, \left\lceil \frac{t + \mathcal{J}_k}{T_k} \right\rceil \cdot C_k \Big) & , o.w. \end{cases} \quad (1)$$

where $\mathcal{J}_k$ is a release jitter of τ_k.

Proof: If τ_k does not have any job-class with a priority level higher π_i^q, it obviously will not cause any interference to J_i^q. This is captured by the first condition of Eq. (1). Otherwise, Eq. (1) computes the interference from the higher-priority jobs of τ_k by extending the conventional iterative response time test [21]. Instead of using a task-level period (T_k) in the ceiling function, we use the minimum inter-arrival time of any two jobs of a job-class ($\eta(J_k^p)$) because a job with the priority of J_k^p cannot repeat more often than $1/\eta(J_k^p)$. With a jitter, interference from a high-priority job is increased because a job can be released earlier by the amount of a jitter [9]. This is exactly captured by the first part of the min term. The amount of interference from τ_k can be also bounded by using its period T_k, which is the same as the task-level analysis [21]. Hence, W_i^q can be safely upper-bounded by the minimum of these two approaches. ■

Theorem 1. *The worst-case response time of J_i^q, denoted by R_i^n, is bounded by the following recurrence:*

$$R_i^{q,n+1} \leftarrow C_i + \sum_{\tau_k \in \Gamma - \tau_i} W_i^q(R_i^{q,n}, \tau_k) \quad (2)$$

where Γ is the entire taskset. The recurrence starts with $R_i^{q,0} = C_i$ and terminates when $R_i^{q,n} + \mathcal{J}_i > D_i$ or $R_i^{q,n+1} = R_i^{q,n}$.

Proof: Obvious from Lemma 7. ■

Lemma 8. *The job-class-level response time test for weakly-hard tasks given in Eq.* (2) *is a generalization of the task-level iterative response time test for hard real-time tasks [21].*

Proof: Any hard real-time task τ_k has only one job-class (J_k^0). Thus, the minimum inter-arrival of this job-class is $\eta(J_k^p) = T_k$ and there is only priority level for τ_k.

Algorithm 2 WCRT for all job-classes of a taskset Γ

```
Input: Γ: Taskset
 1: procedure WCRT(Γ)
 2:     $\digamma \leftarrow$ all job-classes of a taskset $\Gamma$
 3:     ▷ $\pi_i^q$ is a priority of a job-class index $q$ of task $\tau_i$
 4:     for all job-classes $\in \digamma$ in descending order of priority do
 5:         $i \leftarrow$ a task index of a job-class in $\digamma$
 6:         $q \leftarrow$ a job-class index of a task $\tau_i$
 7:         $R_i^q \leftarrow C_i$
 8:         while $R_i^{q,n+1} > R_i^{q,n}$ do
 9:             $W_i^q \leftarrow 0$
10:             for $k = 1$ to $N$ do                    ▷ Check all tasks.
11:                 $v \leftarrow 0$
12:                 if $k \neq i$ then
13:                     for $p = 0$ to $K_k - m_k$ do
14:                         if $\pi_k^p > \pi_i^q$ then
15:                             if WCRT of $J_k^p \leq D_k$ then
16:                                 if $p == 0$ then
17:                                     $\eta(J_k^p) \leftarrow (w_k + 1) \cdot T_k$
18:                                 else if $q > 0$ then
19:                                     $\eta(J_k^p) \leftarrow (p + 2) \cdot T_k$
20:                                 end if
21:                             else
22:                                 if $w_k == 1$ then
23:                                     $\eta(J_k^p) \leftarrow (p + 1) \cdot T_k$
24:                                 else if $w_k > 1$ then
25:                                     $\eta(J_k^p) \leftarrow 1 \cdot T_k$
26:                                 end if
27:                             end if
28:                             if $p == K_k - m_k$ then
29:                                 $\eta(J_k^p) \leftarrow 1 \cdot T_k$
30:                             end if
31:                             $v \leftarrow v + \left\lceil \frac{R_i^{q,n} + \mathcal{J}_k}{\eta(J_k^p)} \right\rceil \times C_k$
32:                         end if
33:                     end for
34:                 end if
35:                 $W_i^q \leftarrow W_i^q + min\Big(v, \left\lceil \frac{R_i^{q,n} + \mathcal{J}_k}{T_k} \right\rceil \times C_k\Big)$
36:             end for
37:             $R_i^{q,n+1} \leftarrow C_i + W_i^q$
38:             $n \leftarrow n + 1$
39:         end while
40:         WCRT of $J_i^q \leftarrow R_i^{q,n+1} + \mathcal{J}_i$
41:     end for
42:     WCRT of all job-classes in $\digamma$
43: end procedure
```

Then, the first part of the min term of Eq. (1) is reduced to $\lceil (t + \mathcal{J}_k)/(T_k) \rceil \times C_k$, which is the same as the second part, and expanding Eq. (2) with this reduced W_i^q gives the same analysis as the conventional response time test. ■

Based on Eqs. (1) and (2), one can compute the WCRT of the job-classes of all tasks, in descending order of job-class priority. This is because the analysis needs the WCRT of higher-priority job-classes to get their inter-arrival time. The detailed procedure for doing this can be found in Algorithm 2. All job-classes are sorted in descending order of their priorities in line 4 so that the $\eta(J_k^p)$ of any higher-priority job-classes can be considered by the WCRT calculation done in the inner *while* loop (from lines 8 to 39). The worst-case interference W_i^q is found by Lemma 7 in line 35 and the WCRT of a job-class J_i^q is computed by Theorem 1 in line 40.

C. Schedulability test for tasks with weakly-hard constraints

After checking the WCRT of all job-classes, now we can test the schedulability of individual tasks w.r.t. weakly-hard constraints. Note that if a task τ_i has no job-class J_i^q with $R_i^q \leq D_i$, the task cannot be said schedulable in our job-class-level analysis framework.

Lemma 9 (Prerequisite for our schedulability analysis)**.** *For a task τ_i to be schedulable by our job-class-level schedulability analysis, the WCRT of the lowest-indexed job-class (J_i^0) should be less than or equal to its deadline D_i.*

Proof: Suppose that the WCRT of the lowest-indexed job-class J_i^0 is greater than its deadline. Then the WCRT of other job-classes (e.g., J_i^1 and J_i^2) of τ_i is always greater than the deadline because J_i^0 has the highest priority in τ_i. Thus, τ_i cannot be guaranteed to be schedulable by our analysis. ■

If a task does not satisfy the above prerequisite, it is immediately considered unschedulable by our analysis. If it satisfies, we next check the m/K ratio of the task, which greatly reduces the time complexity of schedulability analysis.

Lemma 10. *A task τ_i is always schedulable if the ratio of m_i/K_i is greater than or equal to 0.5 and it satisfies the prerequisite given by Lemma 9.*

Proof: Recall the definition of a miss threshold w_i. If τ_i misses w_i deadlines consecutively, the next job of τ_i is assigned the lowest-index job-class J_i^0 whose WCRT is $\leq D_i$ (guaranteed by the prerequisite). This means that, even if τ_i's all other job-classes have WCRT $> D_i$, τ_i can have at least 1 deadline met every $w_i + 1$ periods. Based on this property, the proof can be done in two steps as follows.

Step 1. We prove that there are one or more occurrences of the w_i+1 periods within the K_i window, by showing $\exists \alpha \in \mathbb{Z}$ that satisfies the following equation:

$$(w_i + 1) \cdot \alpha \leq K_i \tag{3}$$

By Def. 6, w_i+1 can be substituted by $\left\lfloor \frac{K_i}{K_i - m_i} \right\rfloor$ as $m_i/K_i \geq 0.5$. With $m_i \leq K_i - 1$, the upper bound of the left-hand side is $\lfloor K_i \rfloor \cdot \alpha$. The inequality $\lfloor K_i \rfloor \cdot \alpha \leq K_i$ is always true for α, and thus it is proved.

Step 2. The inequality $(w_i + 1) \cdot \alpha \leq K_i$ means that there are at least α deadlines met in the K_i window. Hence, if we prove that α is greater than or eqaul to $K_i - m_i$ in the K_i window, the task is always schedulable as long as the condition of Lemma 9 is satisfied. Hence, we prove:

$$\frac{1}{w_i + 1} \geq \frac{K_i - m_i}{K_i} \tag{4}$$

Since $w_i = \left\lfloor \frac{K_i}{K_i - m_i} \right\rfloor - 1$ for $m_i/K_i \geq 0.5$ by Def. 6, Eq. (4) is rewritten as follows:

$$\frac{1}{\left\lfloor \frac{K_i}{K_i - m_i} \right\rfloor} \geq \frac{K_i - m_i}{K_i} \tag{5}$$

$$\Rightarrow \left\lfloor \frac{K_i}{K_i - m_i} \right\rfloor \leq \frac{K_i}{K_i - m_i}$$

This inequality is always true as $m_i \leq K_i - 1$, and thus completes the proof. ■

When the ratio of m_i/K_i is less than 0.5, we check all possible μ-patterns that can happen at runtime. Thus, a reachability tree to be introduced next assumes that the ratio of $m_i/K_i < 0.5$, which also means $w_i = 1$. Moreover, since a single consecutive missed job is allowed, we find the exact upper-bound of the complexity of each tree, which is detailed in Section V-E

D. Reachability trees

A reachability tree consists of a *node*, which indicates a job-class, and a *branch*, which represents deadline missed or met of the node, as depicted in Fig. 6. Two types of nodes exist based on the WCRT. If the WCRT $\leq D_i$, the node has a single meet branch, e.g., nodes 1 and 4 in Fig. 6. Otherwise, a node has two branches (miss and meet), e.g., nodes 2, 3, and 5. Thus, the proposed tree model is generated by the following two Lemmas.

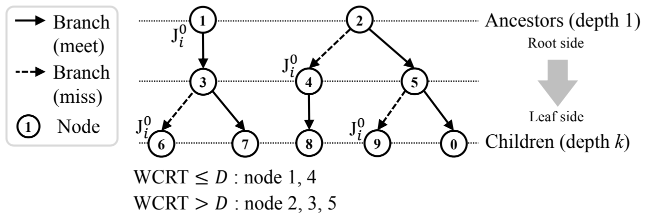


Fig. 6: A reachability tree model

Lemma 11. *If a node is from its parent's miss branch, the node always generates a single meet branch.*

Proof: Since a miss threshold $w_i = 1$, the node from a miss branch always belongs to the lowest-indexed job-class J_i^0. By Lemma 9, the WCRT of the node is less than the deadline. Thus, it only generates a meet branch. ■

Lemma 12. *If a node is from its parent's meet branch, this node generates two sub-branches, miss and meet, in the worst case.*

Proof: When a node is generated by a meet branch, two cases exist based on the WCRT of its parent node: its parent's WCRT $\leq D_i$ and WCRT $> D_i$. In the former case, the node may generate one or two sub-branches based on its own WCRT, as depicted node 3 in Fig. 6. The node always generates two sub-branches when its parent's WCRT $> D_i$ because the WCRT of its parent node, which has a higher priority, is greater than the deadline (e.g., node 5 in Fig. 6). ■

Note that the number of reachability trees of a task is equal to the number of job-classes of the task. This is because $w_i = 1$ and there are no other cases that a task can take as initial conditions. Each tree has the following properties.

1) There exist $K_i - m_i + 1$ trees for a task and the root node of each tree represents a different job-classes of the task.

2) Each tree has K_i depth in order to check the satisfiability of the (m_i, K_i) constraint of a task τ_i.
3) Each node has a μ-pattern, which indicates a series of deadline met or missed jobs from the root to its parent node.
4) The leaf nodes have $\mathcal{C}$-patterns that represent the indices of job-classes for the K_i execution window of a task.

Hence, the proposed reachability tree model of a task generates all possible job-class patterns that happens at runtime, as shown by the following lemma.

Lemma 13. *The reachability trees of a task τ_i represent all possible job-class patterns that the task can experience at its runtime for any execution window of K_i jobs.*

Proof: By the property 1) of the reachability tree, the trees cover all the starting job-classes that the task has. Each node creates all possible cases, i.e., deadline met and missed jobs, by Lemmas 11 and 12. Besides, since individual trees have K_i-depth as stated in property 2), our reachability trees represent all possible job-class patterns for K_i job executions. ■

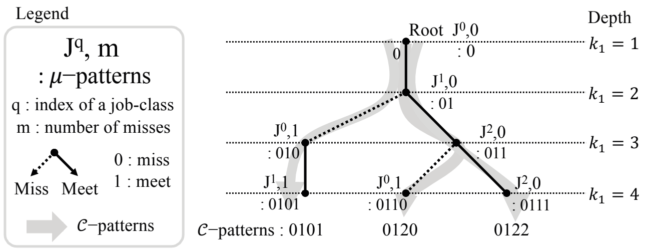


Fig. 7: Examples of a reachability tree

Fig. 7 illustrates an example of a reachability tree from a taskset in Table I. In Fig. 3, we observe that the indices of the first four jobs of Task 1 are $J^0_{1,1}$, $J^1_{1,2}$, $J^2_{1,3}$, and $J^0_{1,4}$ respectively, which is depicted as a $\mathcal{C}$-pattern 0120 in Fig. 7.

Theorem 2. ***Schedulability.*** *A task is guaranteed to be schedulable if the μ-patterns at all leaf nodes in its reachability trees satisfy the weakly-hard constraint.*

Proof: The proof can be done by contradiction. Suppose that all μ-patterns of a task satisfy the constraints, but the task is unschedulable. Since the reachability trees of a task represent all possible job-class patterns by Lemma 13, this means that there exist other job-class patterns that make the task unschedulable. This, however, contradicts the Lemma 13, thus completing the proof. ■

E. Complexity of a reachability tree

The proposed reachability tree model has a bounded computational complexity and is faster than other weakly-hard schedulability methods, which will be shown in the evaluation section. Moreover, as can be seen in Figs. 6 and 7, the number of nodes from the root to leaf nodes forms the Fibonacci sequence.

Theorem 3. *In a reachability tree, the upper-bound on the number of nodes follows the Fibonacci sequence, which starts from the root to the leaf node.*

Proof: By Lemmas 11 and 12, a node in a reachabiity tree can generate only the patterns illustrated in Fig. 6. By these patterns, the number of nodes at depth i of a tree is the sum of the number of its parents and grandparents. Thus, the number of nodes at depth i is bounded as the Fibonacci sequence, and given as $f_{i+2} = f_{i+1} + f_i$. ■

Moreover, for a task, the total complexity of reachabiity trees can be bounded as follows.

Theorem 4. *The complexity of computing all the reachability trees of a task τ_i, O_i, is upper-bounded by*

$$O_i \leq (K_i - m_i + 1) \times \frac{\rho^{K_i+1} - (1-\rho)^{K_i+1}}{\sqrt{5}} \tag{6}$$

where $\rho = \frac{1+\sqrt{5}}{2}$, which is the golden ratio, and $K_i - m_i + 1$ is the number job-classes.

Proof: By Theorem 3, each reachability tree follows the Fibonacci sequence, where the total number of nodes is bounded by $\frac{\rho^{K_i+1}-(1-\rho)^{K_i+1}}{\sqrt{5}}$, as already proved in [40]. Since one reachability tree is created for one distinct job-class, there are at most $K_i - m_i + 1$ trees for τ_i, and thus Eq. 6 holds. ■

VI. Evaluation

We first check the runtime overhead of the proposed job-class-level scheduler by using a prototype implementation in the Linux kernel. We then perform schedulability experiments to compare it with other existing approaches and to explore its performance characteristics in various scenarios.

A. Implementation cost

We have implemented the proposed scheduler in the Linux kernel v4.9.80 running on a Raspberry Pi 3 equipped with a quad-core ARM Cortex-A53 processor. Since our work focuses on uniprocessor scheduling, the implemented scheduler operates on a per-core basis with no task migration. The implementation largely consists of two parts: updating task priority and handling deadline-missed jobs. To update task priority, the scheduler first keeps track of individual job executions and updates the μ-pattern. It then determines the job-class index of a newly released job based on the μ-pattern, and finally sets the task's priority according this job-class. If a job misses the deadline, the scheduler drops this job immediately by stopping its execution, in accordance to our system mode, and rolls the task's state back to the previous clean state. Such a *rollback* is required for the correct operation of the next job as the previous job might have been dropped before releasing a lock or finishing memory writes.

In our implementation, we used a user-level checkpointing technique for the task rollback mechanism. This technique performs the following three steps: 1) creating a checkpoint of a task, 2) notifying a deadline miss from the kernel to the user space, and 3) recovering from the checkpoint. For step 1, a task calls `sigsetjmp` to store its status at the beginning of each job execution. For step 2, the kernel sends a signal to the task when its deadline is missed and the task implements the corresponding signal handler. For step 3, the task's signal handler calls `siglongjmp` to restore the status. If the task has other resources to be restored, such as lock releases, relevant code can be added to the signal handler before calling `siglongjmp`.

Overhead measurement. We measured the runtime overhead of our scheduler implementation. To minimize measurement errors, dynamic frequency scaling was disabled and the processor was configured to use its maximum clock frequency, 1.2 GHz. The overhead was measured by running a taskset consisting of five tasks with periods of 20 ms to 40 ms, and a total of 118,569 jobs were generated during 10 minutes of running time.

TABLE II: Runtime overhead [μs]

Type		Mean	Max	Min	99%th
Updating μ-pattern		0.3002	1.1460	0.1040	0.6250
Updating job-class index		1.5035	11.8750	0.5210	2.5000
Changing task priority		4.7633	28.9580	3.0210	11.3020
Rollback	Checkpointing	1.9413	9.3230	1.2500	3.2290
	Recovery	6.1257	24.8430	0.4680	8.3146

Table II reports the measurement results. In the rollback mechanism, checkpointing is the time to execute `sigsetjmp` and recovery is the time from the kernel sending a signal to the completion of the user-level signal handler. Changing priority and recovery are the two most costly operations. The sum of all entries in each column indicates the total amount of overhead imposed on each job invocation. Since the maximum total overhead per job is observed to be much less than 100 μs, we conclude that the runtime overhead of our scheduler is acceptably small on commodity embedded platforms like Raspberry Pi.

B. Schedulability experiments

This subsection is organized in two parts. The first part presents a comparative evaluation with the two other weakly-hard scheduling schemes [25, 38], and the second part examines the detailed behavior of our job-class-level scheduler. All experiments are conducted on a machine equipped with an Intel Core-i7 2.3 GHz processor and 8GB of memory.

Taskset generation. We use 1,000 randomly-generated weakly-hard tasksets for each experimental setting, e.g., each point on the x-axis of figures. For each taskset, task utilization is obtained by the UUniFast algorithm [7]. Task period is chosen randomly in $[10, 1000]$ ms. Task deadline is set equal to the period, i.e., $T_i = D_i$. Unless otherwise mentioned, release jitter is set to 0. Motivated by a recent study [42] that empirically discovered reasonable weakly-hard constraints from a practical application, the K value is selected from the set of $\{5, 10, 15\}$.

Comparison of schedulability tests. We compare our work with the two other existing approaches. The first one is the offset-free weakly-hard schedulability analysis for fixed-priority scheduling [38]. Although it uses a different method from ours to handle deadline-missed jobs, i.e., jobs continue to execute even after the deadline, we chose this work because it is the latest study on weakly-hard scheduling. The second one is the Red-Task-Only version of the skip-over algorithm [25], and it was chosen as it drops deadline-missed jobs, same as our work. In summary, the following three methods are compared:

- **JCLS**: the proposed Job-Class-Level Scheduler with reachability tree analysis (our work)
- **WSA**: the Weakly-hard Schedulability Analysis for offset-free periodic tasks [38]
- **RTO-RM**: the Red-Task-Only approach for periodic tasks with RM priority assignment [25]

We used the source code of WSA provided in [37] by the authors of [38] and our own implementation of RTO-RM. For our experimental conditions to be consistent with [38], all tasks in the same taskset are set to use a common (m, K) constraint. This is because the currently available WSA source code does not support testing a taskset with various (m_i, K_i) constraints, but we will examine such cases for JCLS in the later part of this subsection. The weakly-hard constraint K for each taskset is set to 10 and m is chosen randomly in the range of $[1, 9]$. Following these rules, we generated 1,000 tasksets with 20 tasks each at each level of the total maximum utilization U^M.

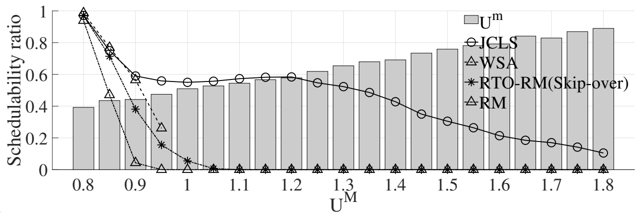


Fig. 8: Schedulability under JCLS, WSA, and RTO-RM

Fig. 8 shows the ratio of weakly-hard schedulable tasksets under the three approaches. The schedulability results under hard RM scheduling is also shown for comparison purpose. At $U^M = 0.95$, JCLS schedules 56% of tasksets while WSA and RTO-RM schedule only 26% and 16% of tasksets, respectively. WSA and RTO-RM do not dominate each other. When $U^M < 1$, WSA outperforms RTO-RM, but when $U^M \geq 1$, WSA cannot find any schedulable taskset while RTO-RM still schedules some tasksets. JCLS shows much higher schedulability ratios than WSA and RTO-RM, e.g., even at $U^M = 1.8$, JCLS schedules 11% of tasksets. This result demonstrates that our JCLS scheduler utilizes the CPU resource more efficiently than the other two prior approaches and the benefit is significant especially when the system is overloaded.

Comparison of analysis running time. In this experiment, we evaluate the analysis running time of JCLS and WSA, which is the time to determine the schedulability of a given taskset. It is obviously affected by the number of tasks in the taskset. We thus consider three cases, where the number of tasks per taskset is 10, 30, and 50, respectively, and generate 1,000 tasksets for each case. The weakly-hard constraint K is set to 10 and m is randomly selected from $[1, 9]$. The analysis running time of JCLS is measured on Raspberry Pi 3 running at 1.2 GHz. On the other hand, WSA is measured on an Intel Core-i7 system running at 2.3 GHz because the CPLEX Optimizer required by the WSA program could not be installed on Raspberry Pi.

TABLE III: Analysis running time of JCLS and WSA [sec]

Number of tasks	Approach	Mean	Max
10	JCLS	0.0010	0.0046
	WSA	0.2739	114.2892
30	JCLS	0.0112	0.0432
	WSA	25.7284	1800.5996
50	JCLS	0.0331	0.1463
	WSA	78.5982	3002.5189

Table III shows the mean and maximum running time of JCLS and WSA. Although JCLS is measured on a much resource-constrained platform, its analysis time is significantly shorter than that of WSA. We observed that the analysis time of JCLS becomes even shorter when it runs on the same x86 platform. Therefore, we conclude that our schedulability analysis using reachability trees is much faster than WSA and is applicable to runtime admission control.

Various (m_i, K_i) constraints. We now use various (m_i, K_i) constraints for tasks in each taskset. Since the WSA implementation does not support this, we will limit our focus to JCLS. We first evaluate the impact of the K_i parameter on weakly-

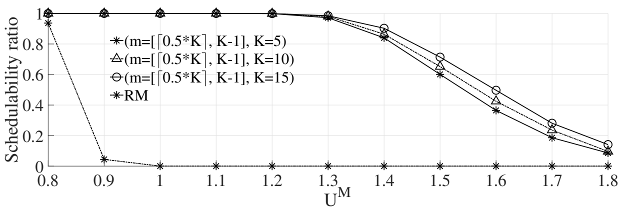


Fig. 9: Schedulability results with different K_i values

hard schedulability in Fig. 9. Three different K_i values are considered under JCLS: 5, 10, and 15. The range of m_i used is $[\lceil 0.5 \times K_i \rceil, K_i - 1]$, and each task's m_i is randomly chosen in this range. Hence, the average ratio of m_i/K_i is similar in all the three K_i cases. Each generated taskset has 20 tasks. As can be seen in the figure, the ratio of schedulable tasksets slightly increases with K_i under JCLS. This is due to that a larger K_i can give more chances for tasks to satisfy their weakly-hard constraints.

We then investigate in Fig. 10 the impact of the m_i/K_i ratio under JCLS, by using different ranges of m_i for a fixed K_i value. The schedulability of JCLS decreases as m_i/K_i reduces. Specifically, we observe that the schedulability of JCLS drops drastically when the m_i/K_i ratio is less than 0.5. This is because if tasks with small m_i have short periods, the

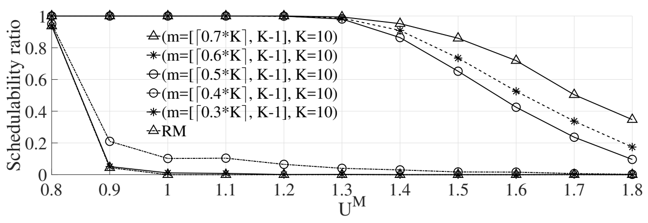
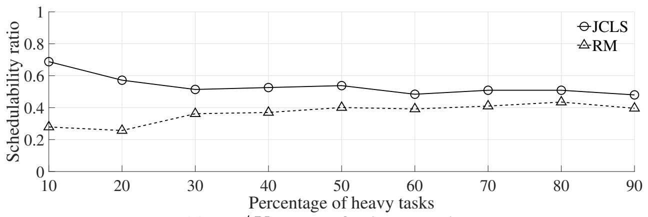
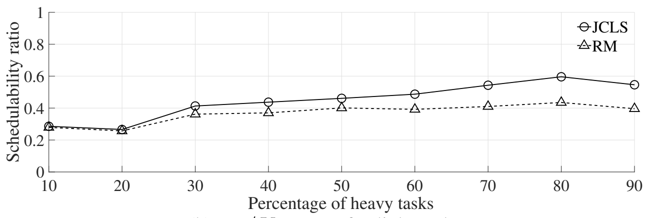


Fig. 10: Schedulability results with different m_i/K_i ratios

use of (m_i, K_i) does not help much in reducing interference to other tasks with larger m_j, compared to hard real-time RM. Furthermore, the higher number of tasks with small m_i in a taskset, the lower the schedulability ratio is. We'll discuss this trend in detail with the following experiments.



(a) $m_i/K_i < 0.5$ for heavy tasks

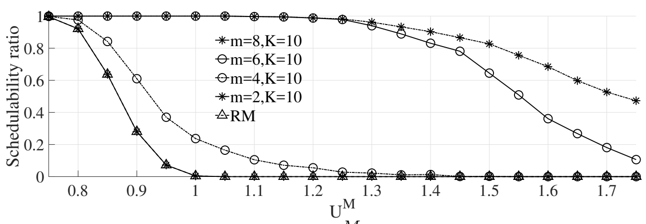


(b) $m_i/K_i < 0.5$ for light tasks

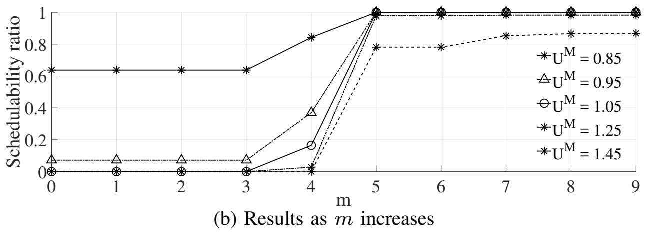
Fig. 11: Schedulability results with bimodal tasksets

Bimodal tasksets with $m_i/K_i < 0.5$ tasks. In order to better understand the schedulability characteristics of JCLS for tasks with $m_i/K_i < 0.5$, we use bimodal tasksets consisting of light and heavy tasks. The utilization ranges for light and heavy tasks are $[0.01, 0.15]$ and $[0.2, 0.4]$, respectively. Either heavy tasks or light tasks are assigned $m_i/K_i < 0.5$ and the other $m_i/K_i \geq 0.5$ depending on experimental settings. For each taskset generation, light and heavy tasks are generated according to their given percentages until the taskset's total maximum utilization exceeds the target U^M, and then the last task's utilization is reduced to meet U^M. The (m_i, K_i) constraints are set to $(4, 10)$ for the $m_i/K_i < 0.5$ case and $(9, 10)$ for the other case. The target maximum total utilization U^M is set to 0.9. Fig. 11 shows the results of JCLS and RM schedulability with bimodal tasksets as the percentage of heavy tasks increases. As expected, we can observe that the difference between JCLS and RM reduces as the percentage of tasks with $m_i/K_i < 0.5$ increases, but in both heavy- and light-task cases, JCLS yields better results than RM.

Varying m_i for a fixed (m_i, K_i) constraint. We also conduct experiments with bimodal tasksets by varying the m_i parameter for a fixed (m_i, K_i) constraint. Fig. 12 shows the results



(a) Results as U^M increases



(b) Results as m increases

Fig. 12: Schedulability results with different m_i values

of this experiment. The percentages of light tasks and heavy tasks are 80% and 20%, respectively, and the m_i parameter of heavy tasks is varied as shown in the legend of Fig. 12a and on the x-axis of Fig. 12b. Other task parameters remain the same as before. Similar to the trends observed in previous experiments, the schedulability decreases as m_i gets smaller.

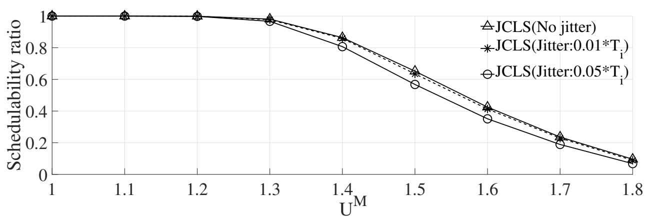


Fig. 13: Schedulability results with release jitters

Release jitters. One of the advantages of the proposed analysis is that it can analyze tasks with non-zero release jitters. In this experiment, we check the schedulability characteristics of JCLS in the presence of jitters that are proportional to the period of each task, e.g., 1% and 5% of T_i. Tasksets are generated in the same way as in Fig. 9. As can be seen in Fig. 13, schedulability decreases slightly as the amount of jitter increases but there is no drastic reduction. This result demonstrates that our approach can be applied to realistic applications where release jitters exist.

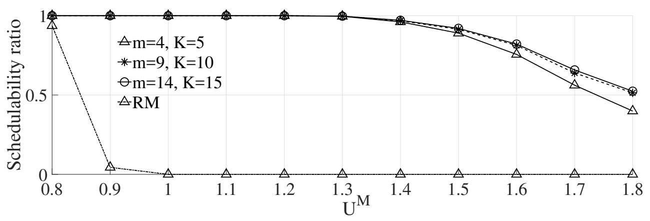


Fig. 14: Schedulability results for consecutive deadline misses

Consecutive deadline misses. As discussed in Section IV, our work can safely bound the number of consecutively missed deadlines, m_i, by modeling $K_i = m_i + 1$ in the (m_i, K_i) form. Tasksets are generated in the same way as in Fig. 9, except for the (m_i, K_i) parameters. Fig. 14 shows that the schedulability of tasksets slightly increases with a larger K_i value. This trend is similar to that in Fig. 9, but one difference here is that the ratio of m_i/K_i reduces as K_i increase, which helps improve schedulability.

VII. Conclusion

In this work, we propose a job-class-level fixed-priority scheduling and a schedulability analysis framework for weakly-hard systems. Our scheduler employs a meet-oriented classification of jobs of tasks and can support the scheduling of periodic and sporadic tasks with arbitrary initial offsets and release jitters. The schedulability analysis for our proposed scheduler consists of two steps: analysis of the worst case response time for individual job-classes, and finding all possible scheduling patterns using the reachability trees. Our evaluation results have demonstrated that the proposed scheduler and its schedulability analysis outperforms prior work with respect to the taskset schedulability and the analytical complexity. It has been also shown that tasksets with the maximum utilization higher than 1 is schedulable under our scheduler. However, we assume that the scheduler is aware of deadline misses of jobs and cancel it immediately when a job missed the deadline. We will investigate the impact of relaxing these assumptions under the job-class-level scheduling framework. Also, some results indicate that our schedulability analysis is pessimistic when the ratio of m_i/K_i is less than 0.5. We leave addressing this pessimism as part of our future work.

References

[1] L. Ahrendts, S. Quinton, and R. Ernst. Exploiting execution dynamics in timing analysis using job sequences. *IEEE Design Test*, PP(99):1–1, 2017.

[2] M. Asberg, T. Nolte, and M. Behnam. Resource sharing using the rollback mechanism in hierarchically scheduled real-time open systems. In *IEEE Real-Time Technology and Applications Symposium (RTAS)*, 2013.

[3] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Hard real-time scheduling: The deadline-monotonic approach. *IFAC Proceedings Volumes*, 24(2):127–132, 1991.

[4] M. Behnam, I. Shin, T. Nolte, and M. Nolin. SIRAP: a synchronization protocol for hierarchical resource sharingin real-time open systems. In *ACM International Conference on Embedded Software (EMSOFT)*, 2007.

[5] G. Bernat, A. Burns, and A. Liamosi. Weakly hard real-time systems. *IEEE transactions on Computers*, 50(4):308–321, 2001.

[6] G. Bernat and R. Cayssials. Guaranteed on-line weakly-hard real-time systems. In *IEEE Real-Time Systems Symposium (RTSS)*, 2001.

[7] E. Bini and G. C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1-2):129–154, 2005.

[8] R. Blind and F. Allgöwer. Towards networked control systems with guaranteed stability: Using weakly hard real-time constraints to model the loss process. In *Decision and Control (CDC), 2015 IEEE 54th Annual Conference on*, pages 7510–7515. IEEE, 2015.

[9] R. J. Bril, J. J. Lukkien, and R. H. Mak. Best-case response times and jitter analysis of real-time tasks with arbitrary deadlines. In *Proceedings of the 21st International conference on Real-Time Networks and Systems*, pages 193–202. ACM, 2013.

[10] T. Bund and F. Slomka. Controller/platform co-design of networked control systems based on density functions. In *ACM SIGBED Interna-*

tional Workshop on Design, Modeling, and Evaluation of Cyber-Physical Systems, pages 11–14. ACM, 2014.

[11] H. S. Chwa and J. L. Kang G. Shin. Closing the gap between stability and schedulability: A new task model for cyber-physical systems. In *IEEE Real-Time Technology and Applications Symposium (RTAS)*, 2018.

[12] G. Frehse, A. Hamann, S. Quinton, and M. Woehrle. Formal analysis of timing effects on closed-loop properties of control software. In *IEEE Real-Time Systems Symposium (RTSS)*, 2014.

[13] M. B. Gaid, D. Simon, and O. Sename. A design methodology for weakly-hard real-time control. *IFAC Proceedings Volumes*, 41(2):10258 – 10264, 2008. 17th IFAC World Congress.

[14] O. Gettings, S. Quinton, and R. I. Davis. Mixed criticality systems with weakly-hard constraints. In *International Conference on Real Time and Networks Systems (RTNS)*, 2015.

[15] J. Goossens. (m, k)-firm constraints and dbp scheduling: impact of the initial k-sequence and exact schedulability test. In *16th International Conference on Real-Time and Network Systems (RTNS)*, 2008.

[16] Z. Guo and S. Baruah. The concurrent consideration of uncertainty in WCETs and processor speeds in mixed-criticality systems. In *International Conference on Real Time and Networks Systems (RTNS)*, pages 247–256. ACM, 2015.

[17] M. Hamdaoui and P. Ramanathan. A dynamic priority assignment technique for streams with (m, k)-firm deadlines. *IEEE Transactions on Computers*, 44(12):1443–1451, 1995.

[18] Z. A. Hammadeh, R. Ernst, S. Quinton, R. Henia, and L. Rioux. Bounding deadline misses in weakly-hard real-time systems with task dependencies. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, 2017.

[19] Z. A. H. Hammadeh, S. Quinton, and R. Ernst. Extending typical worst-case analysis using response-time dependencies to bound deadline misses. In *ACM International Conference on Embedded Software (EMSOFT)*, 2014.

[20] Z. A. H. Hammadeh, S. Quinton, M. Panunzio, R. Henia, L. Rioux, and R. Ernst. Budgeting Under-Specified Tasks for Weakly-Hard Real-Time Systems. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2017.

[21] M. Joseph and P. Pandya. Finding response times in a real-time system. *The Computer Journal*, 29(5):390–395, 1986.

[22] M. Joseph and P. K. Pandya. Finding response times in a real-time system. *Comput. J.*, 29(5):390–395, 1986.

[23] J. Kim, H. Kim, K. Lakshmanan, and R. Rajkumar. Parallel scheduling for cyber-physical systems: Analysis and case study on a self-driving car. In *International Conference on Cyber-Physical Systems (ICCPS)*, 2013.

[24] H. Kopetz. *Real-time systems: design principles for distributed embedded applications*. Springer Science & Business Media, 2011.

[25] G. Koren and D. Shasha. Skip-over: Algorithms and complexity for overloaded systems that allow skips. In *Real-Time Systems Symposium, 1995. Proceedings., 16th IEEE*, pages 110–117. IEEE, 1995.

[26] J. Lee and K. G. Shin. Development and use of a new task model for cyber-physical systems: A real-time scheduling perspective. *Journal of Systems and Software*, 126:45–56, 2017.

[27] J. Li, Y. Song, and F. Simonot-Lion. Providing real-time applications with graceful degradation of QoS and fault tolerance according to (m, k)-firm model. *IEEE Transactions on Industrial Informatics*, 2(2):112–119, 2006.

[28] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973.

[29] P. Marti, A. Camacho, M. Velasco, and M. E. M. B. Gaid. Runtime allocation of optional control jobs to a set of CAN-based networked control systems. *IEEE Transactions on Industrial Informatics*, 6(4):503–520, Nov 2010.

[30] A. K. Mok. Fundamental design problems of distributed systems for the hard real-time environment. *PhD Thesis, Massachusetts Institute of Technology*, 1983.

[31] P. Pazzaglia, L. Pannocchi, A. Biondi, and M. Di Natale. Beyond the weakly hard model: Measuring the performance cost of deadline misses. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 106. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[32] S. Quinton, T. T. Bone, J. Hennig, M. Neukirchner, M. Negrean, and R. Ernst. Typical worst case response-time analysis and its use in automotive network design. In *Design Automation Conference (DAC)*, 2014.

[33] S. Quinton, M. Hanke, and R. Ernst. Formal analysis of sporadic overload in real-time systems. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, 2012.

[34] S. Quinton, M. Negrean, and R. Ernst. Formal analysis of sporadic bursts in real-time systems. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, 2013.

[35] P. Ramanathan. Overload management in real-time control applications using (m, k)-firm guarantee. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):549–559, Jun 1999.

[36] D. Soudbakhsh, L. T. Phan, A. M. Annaswamy, and O. Sokolsky. Co-design of arbitrated network control systems with overrun strategies. *IEEE Transactions on Control of Network Systems*, 2016.

[37] Y. Sun and M. D. Natale. m-k-wsa. https://github.com/m-k-wsa/, 2017.

[38] Y. Sun and M. D. Natale. Weakly hard schedulability analysis for fixed priority scheduling of periodic real-time tasks. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(5s):171, 2017.

[39] S. Tobuschat, R. Ernst, A. Hamann, and D. Ziegenbein. System-level timing feasibility test for cyber-physical automotive systems. In *IEEE Symposium on Industrial Embedded Systems (SIES)*, 2016.

[40] J. Verner E. Hoggatt. *Fibonacci and Lucas Numbers*. Boston: Houghton Mifflin Co., 1969.

[41] W. Xu, Z. A. H. Hammadeh, A. Kröller, R. Ernst, and S. Quinton. Improved deadline miss models for real-time systems using typical worst-case analysis. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2015.

[42] M. Yayla, K.-h. Chen, and J.-j. Chen. Fault Tolerance on Control Applications : Empirical Investigations of Impacts from Incorrect Calculations. In *International Workshop on Emerging Ideas and Trends in Engineering of Cyber-Physical Systems (EITEC)*, 2018.