# Toward Practical Weakly Hard Real-Time Systems: A Job-Class-Level Scheduling Approach

Hyunjong Choi, *Student Member, IEEE*, Hyoseung Kim, *Member, IEEE*, and Qi Zhu, *Member, IEEE*

***Abstract*—Recent applications of the Internet of Things and cyber–physical systems require the integration of many sensing and control tasks into resource-constrained embedded devices. Such tasks can often tolerate a bounded number of timing violations. The concept of *weakly hard* real-time systems can effectively improve resource efficiency without sacrificing system safety. However, the existing studies have limitations on their practical use due to the restrictions imposed on the task timing behavior, high analysis complexity, and the lack of multicore support. In this article, we propose a new *job-class-level fixed-priority preemptive scheduler* and its schedulability analysis framework for weakly hard real-time tasks. 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 allows decomposing the complex weakly hard schedulability problem into two subproblems 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. We also present a semipartitioned task allocation method for multicore platforms, which enhances the schedulability of weakly hard tasks under the proposed scheduling framework. Experimental results indicate that our scheduler outperforms the 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.**

***Index Terms*—Cyber–physical systems (CPS), real-time systems, scheduling, weakly hard constraints.**

## I. Introduction

THE PERFORMANCE and stability of the Internet of Things (IoT) and cyber–physical system (CPS) applications depend not only on the precision of computation but also on the time instant at which the output is generated [2], [3]. Since Liu and Layland's seminal work [4], 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.

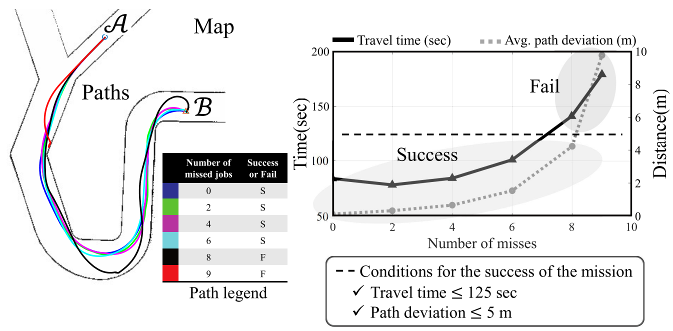



Fig. 1. Autonomous driving of vehicle from point $\mathcal{A}$ (start) to point $\mathcal{B}$ (destination). Paths that the vehicle has driven are depicted in different colors.

*Motivational Example:* We illustrate in Fig. 1 the impact of the number of deadline misses of a periodic task on a self-driving application that is implemented in the robot operating system (ROS). In this example, the mission is that the vehicle drives from point $\mathcal{A}$ (start) to point $\mathcal{B}$ (destination) safely. The success of this mission is determined by whether the vehicle arrives at the destination while satisfying the given conditions of the total travel time[1] and the average deviation from the path.[2] We focused on the control-loop task that sends angular and throttle commands to the vehicle's base at a fixed rate, i.e., 20 Hz by default. For experimental purpose, a specific number of deadline misses were injected to this task every ten periods. As depicted in Fig. 1, the vehicle was able to achieve its mission safely as long as the task missed no more than six deadlines out of ten. However, when eight deadlines were missed (black line in the left part of the figure), the vehicle failed to arrive at the destination within the required travel time (125 s). Furthermore, the vehicle even failed to arrive at the destination when nine deadlines were missed (red line; failure for both criteria). This example motivates the development of *weakly hard real-time systems* [5] for practical IoT and CPS applications in order to enhance resource efficiency within safety boundaries. We believe that the $(m, K)$ notation

Manuscript received July 31, 2020; revised December 31, 2020; accepted January 27, 2021. Date of publication February 9, 2021; date of current version April 7, 2021. This work was supported in part by the Office of Naval Research under Grant N00014-19-1-2496. This article was presented in part at IEEE Real-Time and Embedded Technology and Applications Symposium, 2019. *(Corresponding author: Hyunjong Choi.)*

Hyunjong Choi and Hyoseung Kim are with the Department of Electrical and Computer Engineering, University of California at Riverside, Riverside, CA 92521 USA (e-mail: hchoi036@ucr.edu; hyoseung@ucr.edu).

Qi Zhu is with the Department of Electrical and Computer Engineering and Computer Science, Northwestern University, Evanston, IL 60208 USA (e-mail: qzhu@northwestern.edu).

Digital Object Identifier 10.1109/JIOT.2021.3058215

[1]The travel time is defined as the time from the departure to the complete stop of the vehicle.

[2]This is computed by the average difference between the position data of the vehicle's traveled route and the command route.

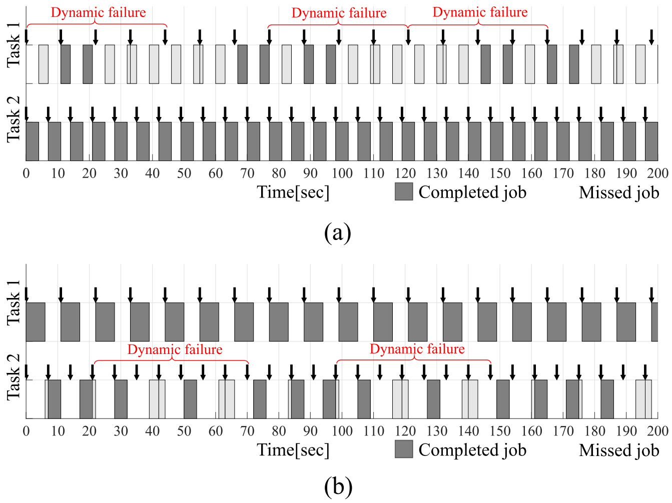



Fig. 2. Dynamic failures under conventional task-level fixed-priority scheduling. (a) Task 1 with low priority. (b) Task 1 with high priority.

of weakly hard constraints, which specifies that at most $m$ task instances (jobs) can miss their deadlines among any $K$ consecutive jobs, can best describe the above system compared with conventional *hard* and *soft* real-time systems due to the following reasons: 1) it is acceptable to miss some deadlines of a task occasionally in the above system but the use of hard real-time constraints strictly disallows any deadline misses and 2) the occurrence of deadline misses needs to be bounded for safety and mission success; whereas, soft real-time constraints do not provide such a rigor.

*Limitation of Existing Methods:* The prior work on predictable $(m, K)$ guarantees [5]–[10] has focused on reducing the pessimism of the schedulability analysis under traditional task-level fixed-priority scheduling, such as rate monotonic (RM) [4], by making strong assumptions on the 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 IoT and CPS applications that require flexibility and adaptability. The high complexity of the existing analysis also makes it difficult to use 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 two deadlines in four consecutive jobs; Task 2 may miss up to four in seven 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 $(6/11) \cdot (2/4) + (4/7) \cdot (3/7) \approx 0.52$.

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. 2 illustrates the task scheduling timeline with these two priority assignments. In both cases, the taskset falls into a *dynamic failure* [11], 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 article, 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 nonzero release jitters. The running time of the resulting schedulability analysis is much shorter than that of the latest work [7] 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 article makes the following contributions.

1) 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.
2) We present a schedulability analysis framework for weakly hard tasks under our scheduler. It consists of two steps: a) analyzing the response time of a job with each job-class, which is done by an extension of the existing task-level analysis and b) 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.
3) We find that the schedulability analysis of tasks with $(m/K) \geq 0.5$ 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 $(m/K) < 0.5$. We show this property with analytical and empirical results.
4) We propose a semipartitioned task allocation scheme for multicore weakly hard systems under the proposed scheduler. The allocation is done in a job-class granularity; hence, the jobs that belong to the same job-class execute on the same processor core. We demonstrate that the proposed scheme yields a significant benefit in schedulability compared to conventional task-level-partitioned scheduling.
5) 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.

6) Experimental results demonstrate that the proposed scheduler outperforms the prior work [7], [12] 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.

This article is an extended version of our prior work [1]. The new contributions include: 1) improved priority assignment to address the schedulability issues of tasks with $(m/K) < 0.5$ constraints; 2) semipartitioned scheduling to enable multicore weakly hard systems; and 3) new experimental results and discussions with the improved priority assignment and the multicore support.

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

## II. Related Work

The notion of $(m, K)$ constraints was first introduced by Hamdaoui and Ramanathan [11]. They focused on reducing the probability of timing violations with the 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 bimodal execution [10], [13] and nonpreemptive tasks [14]. 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 [7] relaxed these assumptions but at the expense of high analysis complexity, e.g., taking more than 10 min for 20 tasks on an Intel Xeon 8-core processor, which makes it difficult to use for the 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. Ismail and Jawawi [16] presented an analysis technique for fixed-priority scheduling in a multicore platform with the same assumptions on initial offset and release jitter. These approaches are done by thoroughly enumerating all task schedules over multiple hyperperiods and checking if there is a repeated feasible schedule. Thus, they can be used for periodic tasks but not for sporadic tasks.

Weakly hard constraints have also been studied to bound the temporal violations of overloaded systems [17]–[19]. 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 the CAN bus analysis [20], tasks with dependencies [6], tasks with varying execution time [21], and budget assignment [8]. 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 [4] or sporadic task models [22], as discussed in [7].

Besides schedulability, preserving control stability has been studied in the context of weakly hard systems. Blind and Allgöwer [23] used weakly hard constraints to capture the failure of unstable feedback control systems in a deterministic way. In [24], a state-based methodology is presented to analyze the performance of a control application using weaklyhard constraints. Frehse *et al.* [25] analyzed the closed-loop properties of control software based on TWCA. In [13], 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.* [26] and Marti *et al.* [27] extended this work to consider optional instances and nonperiodic execution, respectively. Huang *et al.* presented an approach to formally verify the safety of weakly hard systems in [28] and further improved its applicability in [29]. The $(m, k)$ model has been further investigated for control-schedule co-design [30]–[33].

The recent work of Lee and Shin [34] focused on bounding consecutive deadline misses of periodic tasks for CPS. 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 [34]. 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 article considers a multicore system, where all processor cores run at the same fixed clock frequency. The system executes a taskset consisting of $N$ periodic or sporadic real-time tasks with constrained deadlines.

*Task Model:* Each task $\tau_i$ is characterized as follows:

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

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

Each task $\tau_i$ can have a nonzero-initial release offset $o_i$. A release jitter, $\mathcal{J}_i$ ($\mathcal{J}_i \leq D_i - C_i$), represents the maximum time interval between the requested activation and the real-released time of an instance of a task $\tau_i$. The $j$th job of a task $\tau_i$ 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.

*Definition 1:* The maximum utilization of a task $\tau_i$, $U_i^M$, is the maximum amount of CPU resource that $\tau_i$ can utilize. It is defined as $U_i^M = (C_i/T_i)$, 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., $U^M = \sum_{i=1}^{N}(C_i/T_i)$, where $N$ is the number of tasks.

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

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

Each task requires at least $U_i^m$ of CPU resource to be schedulable with respect to (w.r.t.) the weakly hard constraint.

*Deadline-Missed Jobs:* If a job of a task misses its deadline, there are two approaches to handle this job: 1) letting it continue to execute beyond the deadline and 2) dropping (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 [35]. Otherwise, the second approach is more appealing as it can prevent the deadline-missed job from blocking its next job and unnecessarily wasting CPU cycles. Therefore, this article 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 [36], [37] and mixed-criticality systems [38], [39].

## 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. For the ease of presentation, we will assume a uniprocessor system in this section. This assumption will be relaxed in Section VI by introducing our semipartitioned multicore scheduling approach.

### 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: 1) consecutive versus any order and 2) met versus 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: 1) met; 2) missed; 3) consecutively met; and 4) consecutively missed. In this article, we specifically use a *meet-oriented classification* to define a job-class.

*Definition 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 $m_i \geq 1$.

For ease of explanation, we are intentionally being vague about the meaning of the "nearest" previous jobs at this point and will refine it later in this section using Definition 6. 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 $\tau_i$, $J_{i,k}$, its nearest previous jobs with $q$ consecutively met deadlines are $J_{i,x\cdots y}$, 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 with a fixed priority, which is denoted as $\pi_i^q$. 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 $\tau_1$ 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 the 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: 1) *$\mu$-pattern* and 2) $\mathcal{C}$-pattern.

*Definition 4 [5], [11]:* A $\mu$-pattern represents a sequence of deadline met and missed jobs of a task. For example, `MMmMmMM`, where `M` and `m` represent a met and a missed deadline, respectively.

*Definition 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.[3]

*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. 3 shows an example of a task $\tau_1$ 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 $\tau_1$ has four job-classes and the highest index job-class $J^3$ has the highest priority.

The right side of Fig. 3 illustrates a possible scheduling result of task $\tau_1$. Assume that the first three jobs of $\tau_i$, $J_{1,1}^0$, $J_{1,2}^1$, and $J_{1,3}^2$, miss the deadline due to the low priorities of

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

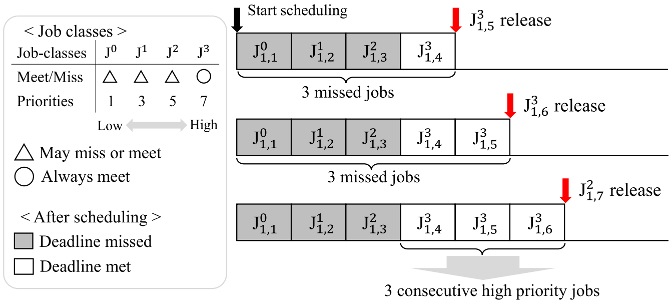



Fig. 3. Consecutive execution of high-priority jobs under *miss-oriented* job classification.

their job-classes. Then, the 4th, 5th and 6th jobs of $\tau_1$ 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 $\tau_1$, 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.

*Bounding Consecutive Deadline Misses:* The above definition (Definition 3) does not specify a *distance* from the current job to the nearest previous deadline-met jobs. If we limit this distance to 0, 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 0, 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 $\tau_i$ 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.

*Definition 6:* A miss threshold $w_i$ is the maximum number of consecutive deadline misses that a task $\tau_i$ can experience at any job-class $J^q_i$ with $q > 0$. If a job of $\tau_i$ 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 task $\tau_i$ can have an enough number of jobs running with $\tau_i$'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 $\tau_i$ to regain $J^0_i$. The way $w_i$ is determined allows $\tau_i$ 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 $\tau_i$ with a weakly hard constraint $(m_i, K_i) = (5, 7)$ is assigned three job-classes and a miss threshold size of $w_i = 2$, based on Definitions 3 and 6. Assume that the very first two jobs of $\tau_i$ have met their deadlines ($\tau_i$'s $\mu$-pattern is `MM`). As a result, the job-class index of the third job becomes 2, i.e., $J^2_{i,3}$. If $J^2_{i,3}$ misses the deadline (`MMm`), the fourth 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`), task $\tau_i$ has reached the miss threshold and the fifth job of $\tau_i$ is assigned $J^0_{i,5}$. On the other hand, if $J^2_{i,4}$ meets the deadline (`MMmM`), the fifth job is assigned $J^1_{i,5}$ because the number of nearest consecutively met deadlines is 1 (as given by Definition 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 [4] and can represent the temporal constraints of the CFP scheduling model [34][4] that upper bounds the number of consecutive 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 $\tau_i$ (e.g., $\forall q : 0 < q \leq K_i - m_i, \pi^q_i = \pi^0_i$), 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 [34].

*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. In this article, we propose two heuristic priority assignment methods: 1) low-index job-class first with miss thresholds (LIF-*w*) and 2) low-index job-class first with a priority holding mechanism (LIF-*h*). Both methods ensure that the priority of a job-class in each task decreases monotonically with increasing job-class index. We will compare the schedulability performance of the two proposed methods in Section VII.

The proposed methods, given in Algorithm 1 (LIF-*w*) and Algorithm 2 (LIF-*h*), are extensions of the deadline-monotonic (DM) priority assignment policy [40]. The algorithms first

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

**Algorithm 1** Job-Class Priority Assignment (LIF-$w$)

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

**Algorithm 2** Job-Class Priority Assignment (LIF-$h$)

**Input:** $\Gamma$: Taskset
Assign priority to all job-classes by calling Alg. 1 (LIF-$w$).
**if** $\Gamma$ is schedulable **then**
$\quad$ **return**
**else**
$\quad \forall i{:}idx_i \leftarrow 0$ $\quad \triangleright$ $idx_i$ is a current job-class index of $\tau_i$.
$\quad$ **for all** $\tau_i \in \Gamma$ **do**
$\quad\quad l_i \leftarrow K_i - m_i + 1$ $\quad \triangleright$ $l_i$: number of job-classes for $\tau_i$
$\quad\quad h_i \leftarrow \lceil \frac{K_i - m_i}{m_i} \rceil$ $\quad \triangleright$ $h_i$: priority holding value of $\tau_i$
$\quad\quad$ **while** $idx_i < l_i$ **do**
$\quad\quad\quad \pi_i^q \leftarrow \pi_i^{idx_i}, \forall q{:}idx_i \leq q < \min\{idx_i + h_i, l_i\}$
$\quad\quad\quad idx_i \leftarrow idx_i + h_i$
$\quad\quad$ **end while**
$\quad$ **end for**
**end if**

check the schedulability of a given taskset $\Gamma$ under the task-level DM assignment (line 7 of Algorithm 1), which can be done by the conventional iterative response-time test [41]. If the taskset is schedulable by DM, the algorithms simply follow the task-level DM assignment. Hence, a task with a shorter deadline is assigned a higher priority and all job-classes of each task get the same priority.

*Lemma 3:* The proposed job-class-level priority assignment algorithms, LIF-$w$ and LIF-$h$, subsume the task-level DM priority assignment.

*Proof:* It is obvious as shown by lines 8-14 of Algorithm 1. ■

*LIF-w:* LIF-$w$ assigns different priority to individual job-classes of tasks when the taskset is not schedulable by DM. By Definition 3, the lowest index job-class of each task implies that there exists no previous job that has met the deadline; thus, the algorithm first assigns priority to the job-classes with the lowest index in ascending order of the deadline. Then, for job-classes with higher indices, those with a lower miss threshold $w$ are assigned higher priority because they can tolerate fewer consecutive deadline misses (see Definition 6). The main part of the LIF-$w$ method consists of 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 $q > 0$.

Under the LIF-$w$ priority assignment, a task cannot continue to use the same high priority for successive jobs. This is because, according to Definition 3, if a job has met its deadline, the priority of the next released job will be demoted by increasing the job-class index. In terms of schedulability, this leads to performance degradation for tasks with weakly hard constraints of $m_i/K_i < 0.5$. As an example, let us consider a task $\tau_i$ with a weakly hard constraint $(m_i, K_i) = (2, 7)$ where $m_i/K_i < 0.5$ and $w_i = 1$. It is assumed that only the lowest index job-class $J_i^0$, i.e., the highest priority job-class, is guaranteed to meet the deadline. As no more than two deadlines can be missed in seven periods, this task requires two or more successive jobs to execute with the highest priority job-class. Based on Definitions 3 and 6, however, $\tau_i$ cannot satisfy this requirement. The $\mu$-pattern of `MmMmMmM` is the sequence that $\tau_i$ can ensure under LIF-$w$, so the task is unschedulable.

*LIF-h:* To overcome the above limitation, LIF-$h$ permits a certain number of low-index job-classes to share the priority of a high-index job-class. We first introduce a priority holding value, $h_i$, which is the number of successive job-classes holding the same priority within a task. From a schedulability perspective, $h_i$ can be determined by the minimum number of consecutive jobs that are required to meet the deadline, in order for task $\tau_i$ to satisfy the given weakly hard constraint $(m_i, K_i)$. Hence, we define $h_i$ as follows:

$$h_i = \left\lceil \frac{K_i - m_i}{m_i} \right\rceil. \tag{1}$$

The value of $h_i$ is always 1 for a task with the weakly hard constraint of $m_i/K_i \geq 0.5$, meaning that the task does not need consecutive high-priority jobs in order to be schedulable. If $m_i/K_i < 0.5$, $h_i$ increases as the weakly hard constraint permits fewer deadline misses (smaller $m_i$), thereby allowing more deadline-meeting jobs to execute with high priority.

By leveraging the value of $h_i$, LIF-$h$ divides the job-classes of a single task into multiple groups, each of which consists of $h_i$ job-classes, and then the first job-class of each group shares its priority with the others in the same group. Therefore, every $h_i$ job-classes of a task $\tau_i$ have the same priority. Note that if the number job-classes of a task is not an integer multiple of $h_i$, the last group consists of job-classes less than $h_i$.

Algorithm 2 describes the LIF-$h$ priority assignment. As an extension of LIF-$w$, LIF-$h$ assigns new priority only when the

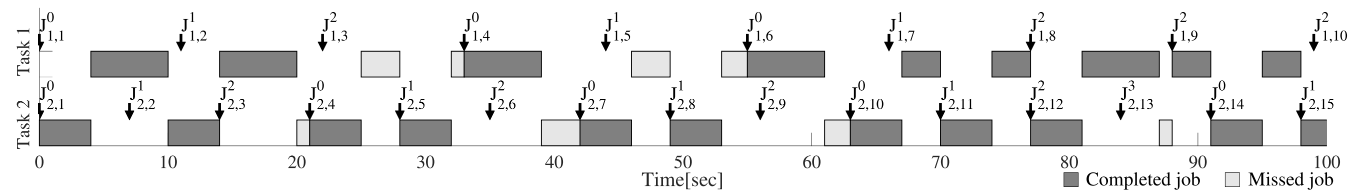



Fig. 4. Execution timeline of weakly hard tasks under job-class-level scheduling.

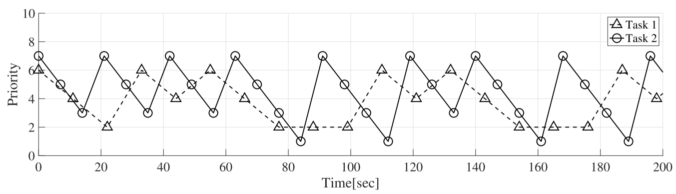



Fig. 5. Priority changes under job-class-level scheduling.

TABLE I
TASK SET I

| | 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 | $\pi_2^0 = 7, \pi_2^1 = 5, \pi_2^2 = 3, \pi_2^3 = 1$ |

given taskset is unschedulable by LIF-$w$. In other words, by the time when the primary part of LIF-$h$ begins (line 5), all job-classes of tasks already have priorities assigned by LIF-$w$. Under LIF-$h$, a group of $h_i$ successive job-classes from the same task inherits the priority from the first job-class ($\pi^{idx_i}$) of that group (line 10). Then, $idx_i$ is updated to indicate the first job-class of the next group (line 11). Hence, $h_i$ successive job-classes starting from the lowest index job-class of each task $\tau_i$ get the same priority.

Note that LIF-$w$ and LIF-$h$ are identical for a task with $w_i = 1$ and $h_i = 1$.

### C. Example of Job-Class-Level Scheduling

Recall the example taskset of Fig. 2, 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. 4 illustrates the execution timeline of the taskset under our scheduler, and Fig. 5 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 weakly hard tasks under job-class-level scheduling. Our analysis can be used for uniprocessor systems as well as semipartitioned multicore systems, which will be presented in the next section. The analysis consists of two parts: 1) analyzing the response time of a job with each job-class and 2) finding job-class patterns that can possibly happen at runtime. We will describe each part and explain how the schedulability test is done.

### A. Minimum Job-Class Interarrival 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 interarrival time.

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

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

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

*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 $\tau_i$, the next released job maintains the current index even if the job meets its deadline. Thus, the minimum time for $\tau_i$ 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 $\le D_i$. If the job misses the deadline, the next job may get $J_i^0$, which causes a 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 interarrival 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

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

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

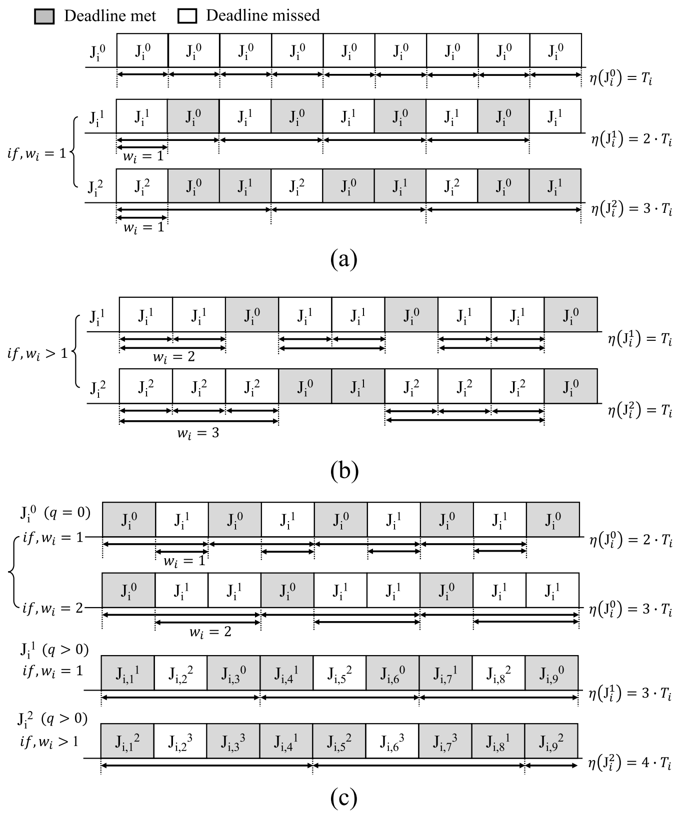


Fig. 6. Minimum time interval between any two jobs of the same job-class. (a) WCRT $> D_i$ and $w_i = 1$. (b) WCRT $> D_i$ and $w_i > 1$. (c) WCRT $\leq D_i$.

long as $w_i$ permits and, thus, the minimum interarrival 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. 6(a) and (b), respectively. ■

*Lemma 6:* The minimum interarrival 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

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

*Proof:* If $q = 0$, the proof is done by contradiction. Assume that the minimum interarrival 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 Definition 3 ($\because$ WCRT of $J_i^q \leq D_i$), the interarrival time of $J_i^0$ is at least $2 \cdot T_i$. This contradicts the assumption because $w_i \geq 1$ by Definition 6. Therefore, the interarrival 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 $\tau_i$ 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 Definition 3 while $q < K_i - m_i$ (the condition of this lemma), 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 $\tau_i$ to get the job-class index $q$. Therefore, the minimum interarrival time of $J_i^q$ for $q > 0$ is $(q + 2) \cdot T_i$. Fig. 6(c) 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 interarrival time given by Lemmas 4–6.

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

$$W_i^q(t, \tau_k) = \begin{cases} 0, & \text{if } \nexists p : \pi_i^q < \pi_k^p \\ \min\left( \sum\limits_{\substack{\forall p: \pi_i^q < \pi_k^p, \\ P(J_k^p) = P(J_i^q)}} \left\lceil \frac{t + \mathcal{J}_k}{\eta\left(J_k^p\right)} \right\rceil \cdot C_k, \left\lceil \frac{t + \mathcal{J}_k}{T_k} \right\rceil \cdot C_k \right), & \text{o.w.} \end{cases} \quad (2)$$

where $\mathcal{J}_k$ is a release jitter of $\tau_k$ and $P(J_i^q)$ is a processor that $J_i^q$ is assigned.

*Proof:* If $\tau_k$ does not have any job-class with a priority level higher $\pi_i^q$, it obviously will not cause any interference to $J_i^q$. This is captured by the first condition of (2). Otherwise, (2) computes the interference from the higher priority jobs of $\tau_k$ by extending the conventional iterative response time test [41]. Instead of using a task-level period ($T_k$) in the ceiling function, we use the minimum interarrival 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 [42]. This is exactly captured by the first part of the min term. The amount of interference from $\tau_k$ can be also bounded by using its period $T_k$, which is the same as the task-level analysis [41]. Hence, $W_i^q$ can be safely upper bounded by the minimum of these two approaches. ■

*Theorem 1:* The WCRT of $J_i^q$, denoted by $R_i^n$, is bounded by the following recurrence:

$$R_i^{q,n+1} \leftarrow C_i + \sum_{\tau_k \in \Gamma - \tau_i} W_i^q\left(R_i^{q,n}, \tau_k\right) \quad (3)$$

where $\Gamma$ 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 (3) is a generalization of the task-level iterative response time test for hard real-time tasks [41].

*Proof:* Any hard real-time task $\tau_k$ has only one job-class ($J_k^0$). Thus, the minimum interarrival of this job-class is $\eta(J_k^p) = T_k$ and there is only priority level for $\tau_k$. Then, the first part of the min term of (2) is reduced to $\lceil (t + \mathcal{J}_k)/(T_k) \rceil \times C_k$, which is the same as the second part, and expanding (3) with this reduced $W_i^q$ gives the same analysis as the conventional response time test. ■

Based on (2) and (3), one can compute the WCRT of the job-classes of all tasks, in descending order of the job-class priority. This is because the analysis needs the WCRT of higher priority job-classes to get their interarrival time. The detailed procedure for doing this can be found in Algorithm 3. All job-classes are sorted in descending order of their priorities in line 4 so that $\eta(J_k^p)$ of any higher priority job-classes assigned to the same processor as $J_i^q$ can be considered by

**Algorithm 3** WCRT for All Job-Classes of Taskset $\Gamma$

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

the WCRT calculation of the *inner* while loop (from lines 9 to 40). The worst case interference $W_i^q$ is found by Lemma 7 in line 36 and the WCRT of a job-class $J_i^q$ is computed by Theorem 1 in line 41.

### *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 task $\tau_i$ 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 $\tau_i$ 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:* Assume 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 $\tau_i$ is always greater than the deadline because $J_i^0$ has the highest priority in $\tau_i$. Thus, $\tau_i$ 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 $\tau_i$ 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 $\tau_i$ misses $w_i$ deadlines consecutively, the next job of $\tau_i$ 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 $\tau_i$'s all other job-classes have WCRT $> D_i$, $\tau_i$ can have at least one 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 $w_i + 1$ periods within the $K_i$ window, by showing $\exists \alpha \in \mathbb{Z}^+$ that satisfies the following equation:

$$(w_i + 1) \cdot \alpha \leq K_i. \tag{4}$$

By Definition 6, $w_i + 1$ can be substituted by $\lfloor [K_i/(K_i - m_i)] \rfloor$ 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 $\alpha$, and thus it is proved.

*Step 2:* The inequality $(w_i + 1) \cdot \alpha \leq K_i$ means that there are at least $\alpha$ deadlines met in the $K_i$ window. Hence, if we prove that $\alpha$ is greater than or equal 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

$$\frac{1}{w_i + 1} \geq \frac{K_i - m_i}{K_i}. \tag{5}$$

Since $w_i = \lfloor [K_i/(K_i - m_i)] \rfloor - 1$ for $m_i/K_i \geq 0.5$ by Definition 6, (5) is rewritten as follows:

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

$$\Leftrightarrow \left\lfloor \frac{K_i}{K_i - m_i} \right\rfloor \leq \frac{K_i}{K_i - m_i}. \tag{6}$$

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 $\mu$-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

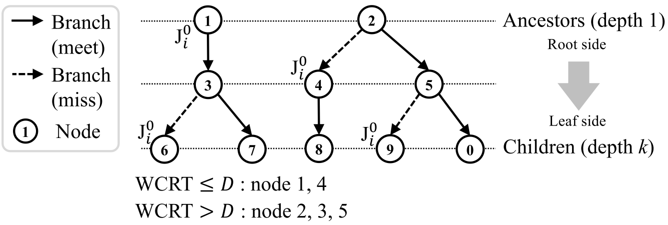



Fig. 7. Reachability tree model.

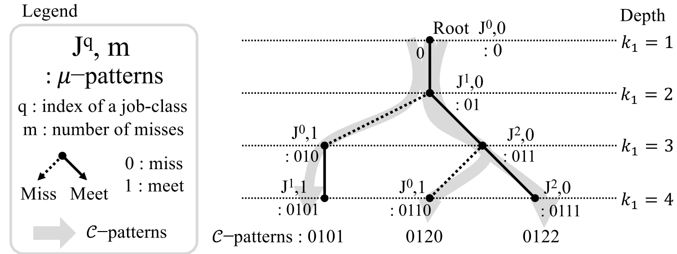



Fig. 8. Examples of the reachability tree.

### 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. 7. 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. 7. 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.

*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 subbranches, 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: 1) its parent's WCRT $\leq D_i$ and 2) WCRT $> D_i$. In the former case, the node may generate one or two subbranches based on its own WCRT, as depicted node 3 in Fig. 7. The node always generates two subbranches 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. 7). ■

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 $\tau_i$.
3) Each node has a $\mu$-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 $\tau_i$ represent all possible job-class patterns that the task can experience at its runtime for any execution window of $K_i$ jobs.

*Proof:* By 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. 8 illustrates an example of a reachability tree from a taskset in Table I. In Fig. 4, we observe that the indices of the first four jobs of task 1 are $J_{1,1}^0$, $J_{1,2}^1$, $J_{1,3}^2$, and $J_{1,4}^0$, respectively, which is depicted as a $\mathcal{C}$-pattern 0120 in Fig. 8.

*Theorem 2 (Schedulability):* A task is guaranteed to be schedulable if the $\mu$-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 $\mu$-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 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. 7 and 8, 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:* Let us denote the number of nodes at depth $i$ as $f_i$, e.g., $f_1 = 1$. In a reachability tree, there are two cases to consider for the root node at depth 1.

First, suppose that the root node has WCRT $\leq D$. At depth 2, $f_2 = 1$ because the root node generates a single meet subbranch to depth 2. At depth 3, $f_3 = 2$ because, by Lemma 12, the node at depth 2 is from the meet branch and creates two subbranches to depth 3. At depth 4, $f_4$ can be obtained considering the two nodes at depth 3: the one from the meet branch of depth 2 generates two subbranches to depth 4 by Lemma 12 (analogous to the case for $f_3$), and the other from the miss branch generates one subbranch by Lemma 11 (analogous to $f_2$). So, $f_4 = f_3 + f_2 = 3$. At depth 5, $f_5 = f_4 + f_3 = 5$ because, among the three nodes of depth 4, two are from the meet and miss branches (the case considered when calculating $f_4$) and one is from the meet branch (the case considered for $f_3$). This pattern continues as the depth increases, and $f_i$ is bounded by the sum of the number of its parents and grandparents, i.e., $f_{i+2} = f_{i+1} + f_i$, following the Fibonacci sequence.

Second, if the root node has WCRT $> D$, it generates two subbranches to depth 2, i.e., $f_2 = 2$. The number of nodes at each subsequent depth follows the same pattern as the above, e.g., $f_3 = f_2 + f_1 = 3$, $f_4 = f_3 + f_2 = 5$, etc. Therefore, the Fibonacci sequence also holds for this case and the proof is done. ■

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

*Theorem 4:* The complexity of computing all the reachability trees of a task $\tau_i$, $O_i$, is upper bounded by

$$O_i \leq (K_i - m_i + 1) \times \frac{\rho^{K_i+1} - (1-\rho)^{K_i+1}}{\sqrt{5}} \tag{7}$$

where $\rho = [(1 + \sqrt{5})/2]$, 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 $[(\rho^{K_i+1} - (1-\rho)^{K_i+1})/\sqrt{5}]$, as already proved in [43]. Since one reachability tree is created for one distinct job-class, there are at most $K_i - m_i + 1$ trees for $\tau_i$ and, thus, (7) holds. ■

**Algorithm 4** Semipartitioned Weakly Hard Task Allocation

**Input:** $\Gamma$: Taskset, $P$: Number of CPUs
1: **function** ASSIGNMENT($\Gamma$, $P$)
2: $\mathcal{F} \leftarrow$ all job-classes of a taskset $\Gamma$
3: $P \leftarrow$ set of all available processor cores
4: **for each** $J_i^q \in \mathcal{F}$ in descending order of priority **do**
5: alloc $\leftarrow$ false
6: **for all** $p \in P$ **do**
7: Allocate $J_i^q$ to $p$ and compute the response time $R_i^q$
8: **if** $R_i^q \leq D_i$ **then**
9: alloc $\leftarrow$ true
10: **break**
11: **else**
12: deallocate $J_i^q$ from $p$
13: **end if**
14: **end for**
15: **if** alloc = false **then**
16: $U_{min} \leftarrow \infty$
17: **for all** $p \in P$ **do**
18: $U_p \leftarrow \sum_{J_j^{q'} \in p} \frac{C_j}{\eta(J_j^{q'})}$
19: **if** $U_p < U_{min}$ **then**
20: $U_{min} \leftarrow U_p$
21: Allocate $J_i^q$ to $p$
22: **end if**
23: **end for**
24: **end if**
25: **end for**
26: **end function**

## VI. Job-Class-Level Scheduling in Multicore Systems

The problem of task scheduling in multicore systems is typically solved by the following three strategies which are classified by how tasks are assigned to processor cores [44]–[46]: 1) partitioned; 2) semipartitioned; and 3) global scheduling. In this work, we propose a semipartitioned task allocation scheme for a multicore platform using our job-class-level scheduler. Our scheme statically allocates individual job-classes of tasks to processor cores. It is semipartitioned in the sense that: 1) a set of jobs belonging to the same job-class always executes on the same core and does not migrate to another core at runtime and 2) another set of jobs belonging to the different job-classes of the same task may execute on different cores.

The proposed semipartitioned allocation scheme has two major benefits. First, our uniprocessor schedulability analysis presented in Section V can be directly used on a per-core basis. Second, it distributes job-classes to processor cores in a way to schedule as many job-classes as possible based on their computed WCRT. We will compare the schedulability results of our proposed semipartitioned scheme and the conventional partitioned scheduling approaches based on bin-packing heuristics, e.g., worst-fit decreasing (WFD), in Section VII.

Algorithm 4 details the procedure of the proposed allocation scheme for all job-classes of a taskset $\Gamma$. First, all job-classes are sorted in descending order of their priorities in line 4. Each job-class $J_i^q$ is assigned to a core where it is schedulable, i.e., the WCRT of $J_i^q$ is less than or equal to $D_i$ (lines 6–14). If a job-class cannot find a core where it is schedulable, it is assigned to a core with the lowest total job-class utilization $U_p$ (lines 17–23). Here, the job-class utilization means the share of the CPU time required for the execution of a job-class. Thus, the utilization of a job-class $J_i^q$ can be estimated by the WCET $C_i$ divided by the minimum job-class arrival time $\eta(J_i^q)$ which is given in Section V-A. The total job-class utilization $U_p$ is then computed by summing the utilization of all job-classes assigned to the processor core $p$ (line 18). It is worth noting that the algorithm uses $U_p$ as a proxy to measure the current load of the core.

## VII. 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. The implementation largely consists of two parts: 1) updating task priority and 2) handling deadline-missed jobs. To update task priority, the scheduler first keeps track of individual job executions and updates the $\mu$-pattern. It then determines the job-class index of a newly released job based on the $\mu$-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*

TABLE II
RUNTIME OVERHEAD [$\mu$S]

| Type | | Mean | Max | Min | 99%th |
|---|---|---|---|---|---|
| Updating $\mu$-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 |

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 of 1.2 GHz. The overhead was measured by running a taskset consisting of five tasks with periods of 20 to 40 ms, and a total of 118 569 jobs were generated during 10 min of running time.

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 $\mu$s, we conclude that the runtime overhead of our scheduler is acceptably small on commodity-embedded platforms like Raspberry Pi.

## B. Uniprocessor Experiments

This section is organized in two parts. The first part presents a comparative evaluation with the two other weakly hard scheduling schemes [7], [12] and the second part examines the detailed behavior of our job-class-level scheduler.

*Taskset Generation:* We use 1000 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 [47]. 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

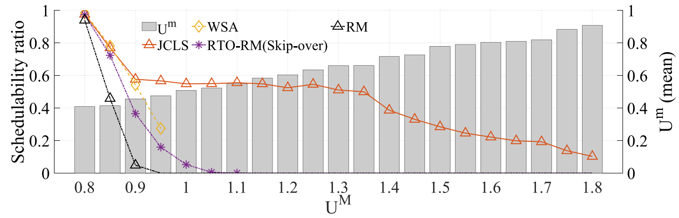



Fig. 9. Schedulability under JCLS, WSA, and RTO-RM.

jitter is set to 0. Motivated by a recent study [48] 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 [7]. 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 [12], and it was chosen as it drops deadline-missed jobs, same as our work. Note that, although [34] has inspired our work, we do not consider it in the evaluation because it is not developed for weakly hard systems and cannot handle tasks with $(m, K)$ constraints. In summary, the following three methods are compared.

1) *JCLS:* The proposed reachability tree analysis under Job-Class-Level Scheduler with the LIF-$h$ priority assignment (our work).
2) *WSA:* The weakly hard schedulability analysis for offset-free periodic tasks [7].
3) *RTO-RM:* The red-task-only approach for periodic tasks with RM priority assignment [12].

We used the source code of WSA provided in [49] by Sun and Natale [7] and our own implementation of RTO-RM. For our experimental conditions to be consistent with [7], 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 section. 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 1000 tasksets with 20 tasks each at each level of the total maximum utilization $U^M$.

Fig. 9 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

TABLE III
ANALYSIS RUNNING TIME OF JCLS AND WSA [S]

| 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 |

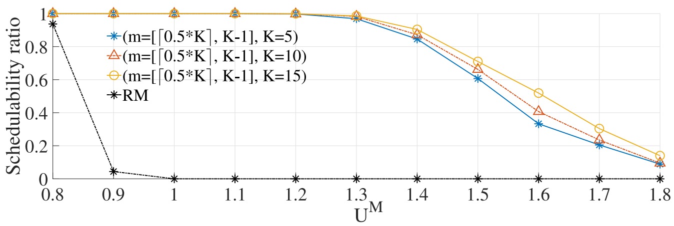



Fig. 10. Schedulability results with different $K_i$ values.

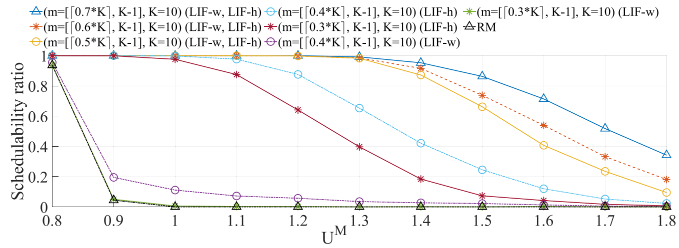



Fig. 11. Schedulability results with different $m_i/K_i$ ratios.

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 1000 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 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 current implementation of WSA does not support this, we will limit our focus to JCLS. Furthermore, we conduct performance evaluation of JCLS using the two proposed priority assignment methods, i.e., LIF-$w$ and LIF-$h$, for all subsequent experiments.

We first evaluate the impact of the $K_i$ parameter on weakly hard schedulability in Fig. 10. Each generated taskset has 20 tasks. 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. Since this condition gives the priority holding value of $h_i = 1$, the results from LIF-$w$ and LIF-$h$ are idential and, thus, shown using the same line. 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. 11 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, for tasks with $m_i/K_i < 0.5$, we observe that the schedulability of JCLS with LIF-$w$ drops drastically. On the other hand, JCLS with LIF-$h$ gives much higher schedulability than LIF-$w$, because LIF-$h$ allows more job-classes of such tasks to execute with high priority.

*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. 12 shows the schedulability results of bimodal tasksets under JCLS (LIF-$w$ and LIF-$h$) and RM 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 with LIF-$h$ yields better results than JCLS with LIF-$w$ and 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. 13 shows the results 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. 13(a) and on the $x$-axis of Fig. 13(b) and (c). Other task parameters remain the same as before. Similar to the trends observed in the previous experiments, the schedulability decreases as $m_i$ gets smaller. LIF-$h$ outperforms LIF-$w$ for tasks with the weakly hard constraint of $m_i/K_i < 0.5$. In particular, for tasksets with $m_i$ is 4 and 2 at $U^M = 0.95$, LIF-$h$ improves schedulability ratio over LIF-$w$ by 59% and 89% points, respectively.

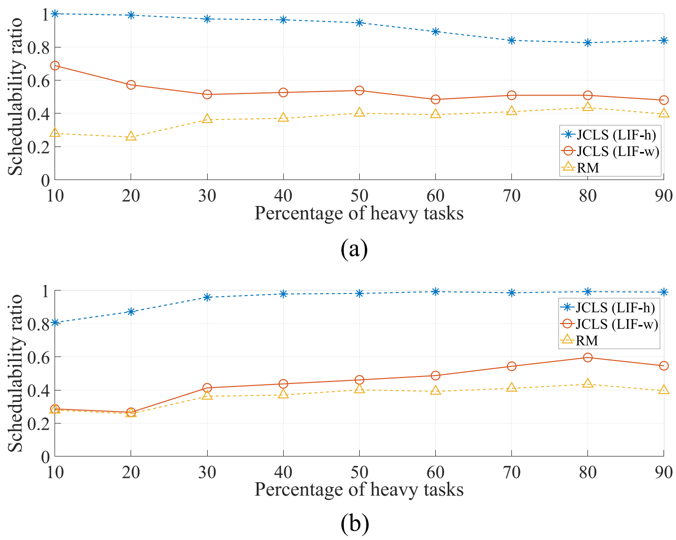



Fig. 12. Schedulability results with bimodal tasksets. (a) $m_i/K_i < 0.5$ for heavy tasks. (b) $m_i/K_i < 0.5$ for light tasks.

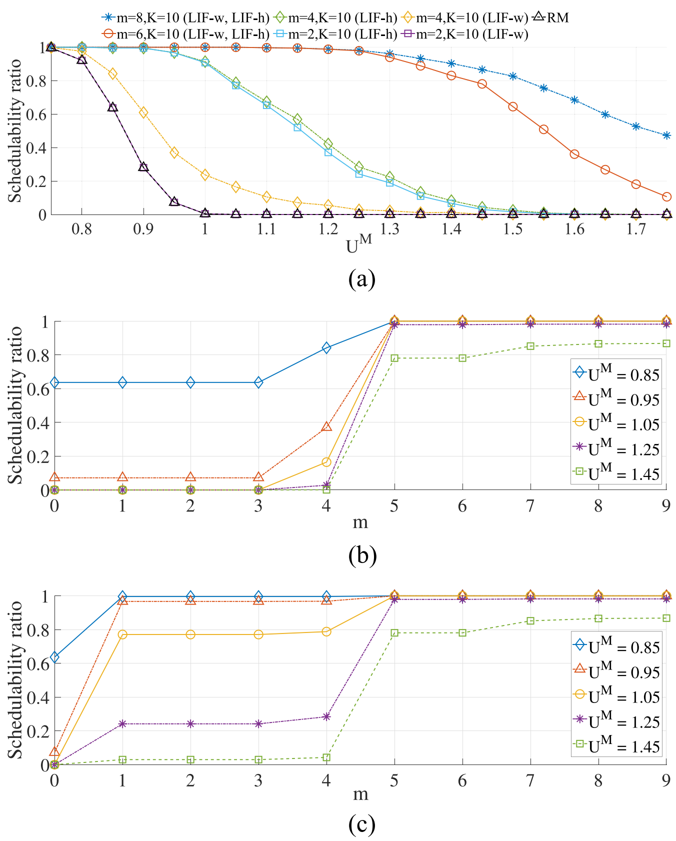



Fig. 13. Schedulability results with different $m_i$ values. (a) Results as $U^M$ increases. (b) Results of JCLS LIF-*w* as $m$ increases. (c) Results of JCLS LIF-*h* as $m$ increases.

*Release jitters:* One of the advantages of the proposed analysis is that it can analyze tasks with nonzero 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. 10, so there is no difference between LIF-*w* and LIF-*h*. As can be seen in

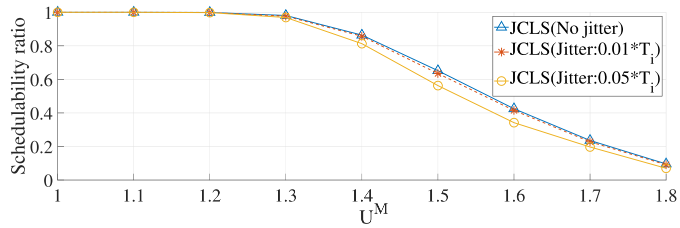



Fig. 14. Schedulability results with release jitters.

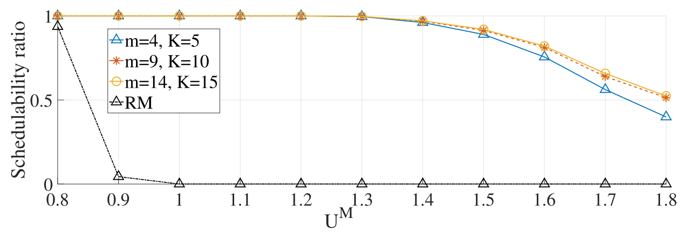



Fig. 15. Schedulability results for consecutive deadline misses.

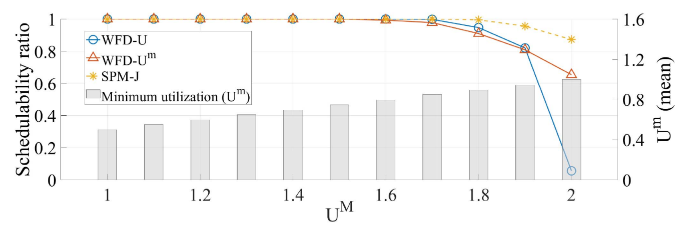



Fig. 16. Schedulability in multicore systems (two CPUs).

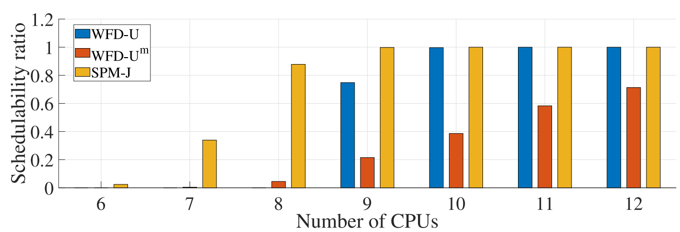



Fig. 17. Schedulability results of tasksets with $U^M = 8$ in multicore systems.

Fig. 14, 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.

*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. 10, except for the $(m_i, K_i)$ parameters. Fig. 15 shows that the schedulability of tasksets slightly increases with a larger $K_i$ value. This trend is similar to that in Fig. 10, but one difference here is that the ratio of $m_i/K_i$ reduces as $K_i$ increase, which helps improve schedulability.

## C. Multicore Experiments

We now evaluate the schedulability of JCLS in multicore systems. In this section, the following three task assignment methods are compared under the JCLS framework.

1) *WFD-U:* A partitioning method based on WFD using the utilization ($U_i$) of each task.

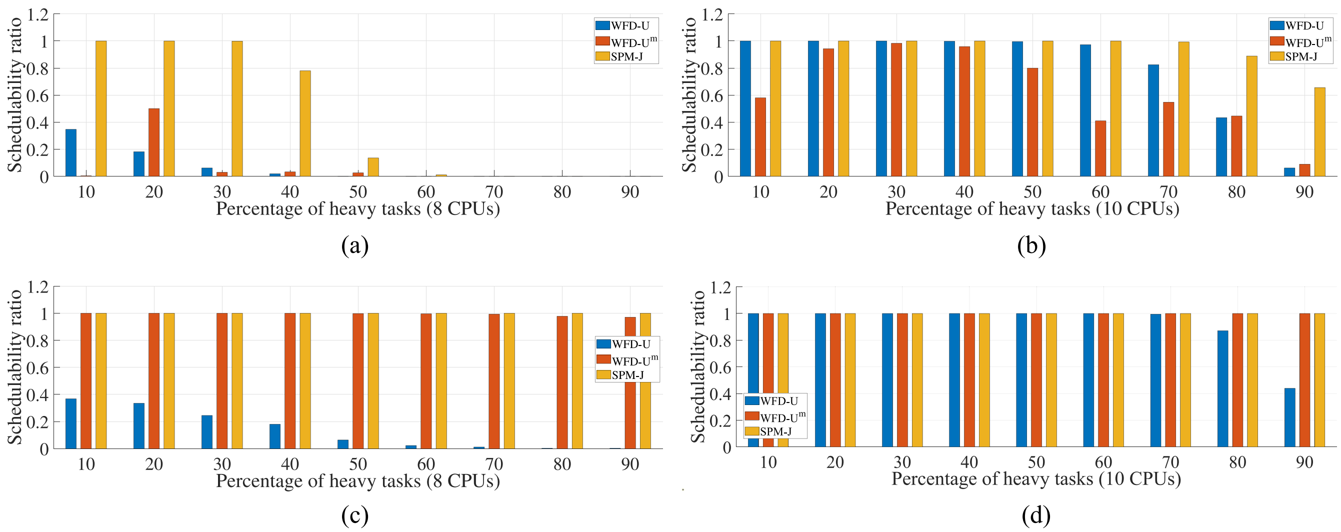



Fig. 18. Schedulability results with bimodal tasksets in multicore systems. (a) $m_i/K_i < 0.5$ for heavy tasks (eight CPUs). (b) $m_i/K_i < 0.5$ for heavy tasks (ten CPUs). (c) $m_i/K_i < 0.5$ for light tasks (eight CPUs). (d) $m_i/K_i < 0.5$ for light tasks (ten CPUs).

2) *WFD-$U^m$:* A partitioning method-based WFD using the minimum utilization ($U_i^m$) of each task.
3) *SPM-J:* A semipartitioning method based on the WCRT of each job-class (Algorithm 4).

Basically, all jobs of each task execute on the same processor under the two partitioned methods (WFD-$\boldsymbol{U}$ and WFD-$\boldsymbol{U^m}$) while SPM-J assigns job-classes of each task to different processors as described in Algorithm 4. It is worth noting that WFD is chosen for load balancing across processor cores, as done in the literature [50].

Fig. 16 shows the schedulability results under WFD-$U$, WFD-$U^m$, and SPM-J when two processors are used. We generated tasksets over the range of the total maximum utilization $U^M$ from 1 to 2 and other task parameters remain the same as before. As can be seen, SPM-J outperforms the others. WFD-$U$ and WFD-$U^m$ do not dominate each other. When $U^M < 2$, WFD-$U$ has slightly higher schedulability ratio than WFD-$U^m$. When $U^M = 2$, however, we observe that the schedulability of WFD-$U$ drops drastically. This is because there exist tasks that cannot be assigned to any processor due to the pessimism of $U_i$ in representing the actual resource demand of weakly hard tasks.

*Number of Processors:* The results of the schedulability ratio according to the given number of processors are shown in Fig. 17. We generated 1000 tasksets with 30 tasks with the maximum utilization $U^M = 8$. The number of processors is chosen as an integer over the range [6, 12]. As expected, we can observe that the schedulability ratio increases in all three methods as the number of processors increases. SPM-J dominates the others for any number of processors. In particular, when the number of processors is 8, SPM-J schedules 88% tasksets while WFD-$U$ and WFD-$U^m$ schedule only 0% and 5% of tasksets, respectively.

*Schedulability of Bimodal Tasksets in Multicore Systems:* Finally, we use bimodal tasksets consisting of light and heavy tasks. We generated bimodal tasksets in the same way as done for the previous experiment of Fig. 12, except that the target maximum total utilization $U^M$ is set to 8. The number of processors is chosen from the set of {8, 10}.

Fig. 18 shows the schedulability results depending on task assignment methods. Similar to the trends observed in the previous experiments, the schedulability ratio decreases as the percentage of tasks with $m_i/K_i < 0.5$ increases. WFD-$U$ and WFD-$U^m$ do not dominate each other. We can observe that the schedulability of WFD-$U$ is overall higher than WFD-$U^m$ in the heavy-task case [Fig. 18(a) and (b)]. On the other hand, WFD-$U^m$ yields better results than WFD-$U$ in the light-task case [Fig. 18(c) and (d)]. Interestingly, WFD-$U^m$ does not exhibit a uniform behavior as shown in Fig. 18(a) and (b). This is because the minimum utilization $U_i^m$ used by WFD-$U^m$ is not a stable metric compared to $U_i$ which is widely used for conventional task allocation approaches. However, SPM-J consistently yields higher schedulability than the others in both light- and heavy-task cases. These results demonstrate the effectiveness of our approach for multicore weakly hard real-time systems.

## VIII. Conclusion

In this article, we proposed a job-class-level fixed-priority scheduling and a schedulability analysis framework for weakly hard real-time systems. Our scheduler employs a meet-oriented classification of jobs of tasks and supports the scheduling of periodic and sporadic tasks with arbitrary initial offsets and release jitters. Besides, we presented two heuristic job-level priority assignment methods that are practically usable. The schedulability analysis framework for our proposed scheduler consists of two steps: 1) analysis of the worst case response time for individual job-classes and 2) finding all possible scheduling patterns using the reachability trees. This framework was directly used for task scheduling in multicore platforms by the proposed semipartitioned scheduling approach. Our scheduler has been implemented in the Linux kernel on Raspberry Pi with acceptably small runtime overhead.

Evaluation results have demonstrated that the proposed scheduler and its schedulability analysis outperforms prior work w.r.t. 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. The schedulability characteristics of two proposed priority assignment methods are explored through experiments of tasksets with various $(m_i, K_i)$ constraints. Furthermore, we have observed that the proposed semipartitioned scheme yields significant benefit over conventional approaches in multicore scheduling. For future work, we are interested in investigating the impact of shared memory contention in order to further enhance the practical usability of multicore weakly hard real-time systems.

## REFERENCES


[1] H. Choi, H. Kim, and Q. Zhu, "Job-class-level fixed priority scheduling of weakly-hard real-time systems," in *Proc. IEEE Real-Time Embedded Technol. Appl. Symp. (RTAS)*, 2019, pp. 241–253.

[2] H. Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications*. New York, NY, USA: Springer, 2011.

[3] 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 *Proc. Int. Conf. Cyber-Phys. Syst. (ICCPS)*, 2013, pp. 31–40.

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

[5] G. Bernat, A. Burns, and A. Liamosi, "Weakly hard real-time systems," *IEEE Trans. Comput.*, vol. 50, no. 4, pp. 308–321, Apr. 2001.

[6] Z. A. H. Hammadeh, R. Ernst, S. Quinton, R. Henia, and L. Rioux, "Bounding deadline misses in weakly-hard real-time systems with task dependencies," in *Proc. Design Autom. Test Europe Conf. Exhibition (DATE)*, 2017, pp. 584–589.

[7] Y. Sun and M. D. Natale, "Weakly hard schedulability analysis for fixed priority scheduling of periodic real-time tasks," *ACM Trans. Embedded Comput. Syst. (TECS)*, vol. 16, no. 5s, p. 171, 2017.

[8] 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 *Proc. Euromicro Conf. Real-Time Syst. (ECRTS)*, 2017, pp. 1–22.

[9] O. Gettings, S. Quinton, and R. I. Davis, "Mixed criticality systems with weakly-hard constraints," in *Proc. Int. Conf. Real Time and Networks Systems (RTNS)*, 2015, pp. 237–246.

[10] G. Bernat and R. Cayssials, "Guaranteed on-line weakly-hard real-time systems," in *Proc. IEEE Real-Time Syst. Symp. (RTSS)*, 2001, pp. 25–35.

[11] M. Hamdaoui and P. Ramanathan, "A dynamic priority assignment technique for streams with (m, k)-firm deadlines," *IEEE Trans. Comput.*, vol. 44, no. 12, pp. 1443–1451, Dec. 1995.

[12] G. Koren and D. Shasha, "Skip-over: Algorithms and complexity for overloaded systems that allow skips," in *Proc. IEEE Real-Time Syst. Symp.*, 1995, pp. 110–117.

[13] P. Ramanathan, "Overload management in real-time control applications using (m, k)-firm guarantee," *IEEE Trans. Parallel Distrib. Syst.*, vol. 10, no. 6, pp. 549–559, Jun. 1999.

[14] 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 Trans. Ind. Informat.*, vol. 2, no. 2, pp. 112–119, May 2006.

[15] J. Goossens, "(m, k)-firm constraints and DBP scheduling: Impact of the initial k-sequence and exact schedulability test," in *Proc. 16th Int. Conf. Real-Time Netw. Syst. (RTNS)*, 2008, pp. 61–65.

[16] H. Ismail and D. N. Jawawi, "A weakly hard scheduling approach of partitioned scheduling on multiprocessor systems," *Universiti Teknologi Malaysia J. Teknologi Sci. Eng.*, vol. 77, no. 9, pp. 179–190, 2015.

[17] S. Quinton, M. Hanke, and R. Ernst, "Formal analysis of sporadic overload in real-time systems," in *Proc. Design Autom. Test Europe Conf. Exhibition (DATE)*, 2012, pp. 515–520.

[18] Z. A. H. Hammadeh, S. Quinton, and R. Ernst, "Extending typical worst-case analysis using response-time dependencies to bound deadline misses," in *Proc. ACM Int. Conf. Embedded Softw. (EMSOFT)*, 2014, pp. 1–10.

[19] 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 *Proc. Euromicro Conf. Real-Time Syst. (ECRTS)*, 2015, pp. 247–256.

[20] 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 *Proc. Design Autom. Conf. (DAC)*, 2014, p. 44.

[21] L. Ahrendts, S. Quinton, and R. Ernst, "Exploiting execution dynamics in timing analysis using job sequences," *IEEE Design Test*, vol. 35, no. 4, pp. 16–22, Aug. 2018.

[22] A. K. Mok, "Fundamental design problems of distributed systems for the hard real-time environment," Ph.D. dissertation, Lab. Comput. Sci., Massachusetts Inst. Technol., Cambridge, MA, USA, 1983.

[23] R. Blind and F. Allgöwer, "Towards networked control systems with guaranteed stability: Using weakly hard real-time constraints to model the loss process," in *Proc. IEEE Annu. Conf. Decis. Control (CDC)*, 2015, pp. 7510–7515.

[24] P. Pazzaglia, L. Pannocchi, A. Biondi, and M. D. Natale, "Beyond the weakly hard model: Measuring the performance cost of deadline misses," in *Proc. Euromicro Conf. Real-Time Syst. (ECRTS)*, 2018, pp. 1–22.

[25] G. Frehse, A. Hamann, S. Quinton, and M. Woehrle, "Formal analysis of timing effects on closed-loop properties of control software," in *Proc. IEEE Real-Time Syst. Symp. (RTSS)*, 2014, pp. 53–62.

[26] M. B. Gaid, D. Simon, and O. Sename, "A design methodology for weakly-hard real-time control," *IFAC Proc. Vol.*, vol. 41, no. 2, pp. 10258–10264, 2008.

[27] 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 Trans. Ind. Informat.*, vol. 6, no. 4, pp. 503–520, Nov. 2010.

[28] C. Huang, W. Li, and Q. Zhu, "Formal verification of weakly-hard systems," in *Proc. ACM Int. Conf. Hybrid Syst. Comput. Control (HSCC)*, 2019, pp. 197–207.

[29] C. Huang, K.-C. Chang, C.-W. Lin, and Q. Zhu, "SAW: A tool for safety analysis of weakly-hard systems," in *Proc. Int. Conf. Comput.-Aided Verification (CAV)*, 2020, pp. 543–555.

[30] D. Soudbakhsh, L. T. Phan, A. M. Annaswamy, and O. Sokolsky, "Co-design of arbitrated network control systems with overrun strategies," *IEEE Trans. Control Netw. Syst.*, vol. 5, no. 1, pp. 128–141, Mar. 2018.

[31] H. S. Chwa, J. Lee, and K. G. Shin, "Closing the gap between stability and schedulability: A new task model for cyber-physical systems," in *Proc. IEEE Real-Time Technol. Appl. Symp. (RTAS)*, 2018, pp. 327–337.

[32] H. Liang, Z. Wang, R. Jiao, and Q. Zhu, "Leveraging weakly-hard constraints for improving system fault tolerance with functional and timing guarantees," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, 2020, p. 101.

[33] H. Liang, Z. Wang, D. Roy, S. Dey, S. Chakraborty, and Q. Zhu, "Security-driven codesign with weakly-hard constraints for real-time embedded systems," in *Proc. IEEE Int. Conf. Comput. Design (ICCD)*, 2019, pp. 217–226.

[34] J. Lee and K. G. Shin, "Development and use of a new task model for cyber-physical systems: A real-time scheduling perspective," *J. Syst. Softw.*, vol. 126, pp. 45–56, Apr. 2017.

[35] H. Choi and H. Kim, "Work-in-progress: A unified runtime framework for weakly-hard real-time systems," in *Proc. Brief Presentation Track RTAS*, 2019, p. 13.

[36] M. Asberg, T. Nolte, and M. Behnam, "Resource sharing using the rollback mechanism in hierarchically scheduled real-time open systems," in *Proc. IEEE Real-Time Technol. Appl. Symp. (RTAS)*, 2013, pp. 129–140.

[37] M. Behnam, I. Shin, T. Nolte, and M. Nolin, "SIRAP: A synchronization protocol for hierarchical resource sharingin real-time open systems," in *Proc. ACM Int. Conf. Embedded Softw. (EMSOFT)*, 2007, pp. 279–288.

[38] Z. Guo and S. K. Baruah, "The concurrent consideration of uncertainty in WCETs and processor speeds in mixed-criticality systems," in *Proc. Int. Conf. Real Time Netw. Syst. (RTNS)*, 2015, pp. 247–256.

[39] D. de Niz, B. Andersson, H. Kim, M. Klein, L. T. X. Phan, and R. Rajkumar, "Mixed-criticality processing pipelines," in *Proc. Design Autom. Test Europe Conf. Exhibition (DATE)*, 2017, pp. 1372–1375.

[40] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings, "Hard real-time scheduling: The deadline-monotonic approach," *IFAC Proc. Vol.*, vol. 24, no. 2, pp. 127–132, 1991.

[41] M. Joseph and P. Pandya, "Finding response times in a real-time system," *Comput. J.*, vol. 29, no. 5, pp. 390–395, 1986.

[42] 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 *Proc. 21st Int. Conf. Real-Time Netw. Syst.*, 2013, pp. 193–202.

[43] V. E. Hoggatt, Jr., *Fibonacci and Lucas Numbers*. Boston, MA. USA: Houghton Mifflin Co., 1969.

[44] Y. Oh and S. H. Son, "Allocating fixed-priority periodic tasks on multiprocessor systems," *Real-Time Syst.*, vol. 9, no. 3, pp. 207–239, 1995.

[45] S. K. Dhall and C. L. Liu, "On a real-time scheduling problem," *Oper. Res.*, vol. 26, no. 1, pp. 127–140, 1978.

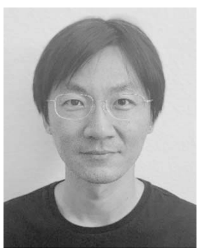
[46] B. Andersson and J. Jonsson, "Fixed-priority preemptive multiprocessor scheduling: To partition or not to partition," in *Proc. 7th Int. Conf. Real-Time Comput. Syst. Appl.*, 2000, pp. 337–346.

[47] E. Bini and G. C. Buttazzo, "Measuring the performance of schedulability tests," *Real-Time Syst.*, vol. 30, nos. 1–2, pp. 129–154, 2005.

[48] M. Yayla, K.-H. Chen, and J.-J. Chen, "Fault tolerance on control applications: Empirical investigations of impacts from incorrect calculations," in *Proc. Int. Workshop Emerg. Ideas Trends Eng. Cyber-Phys. Syst. (EITEC)*, 2018, pp. 17–24.

[49] Y. Sun and M. D. Natale. (2017). *m-k-WSA*. [Online]. Available: https://github.com/m-k-wsa/

[50] S. Hosseinimotlagh and H. Kim, "Thermal-aware servers for real-time tasks on multi-core GPU-integrated embedded systems," in *Proc. IEEE Real-Time Embedded Technol. Appl. Symp. (RTAS)*, 2019, pp. 254–266.

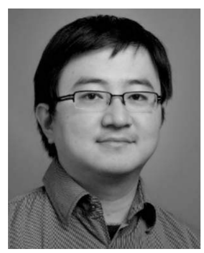
**Hyunjong Choi** (Student Member, IEEE) received the B.S. degree from Yonsei University, Seoul, South Korea, in 2007, and the M.S. degree from the Korea Advanced Institute of Science and Technology, Daejeon, South Korea, in 2009. He is currently pursuing the Ph.D. degree with the Department of Electrical and Computer Engineering, University of California at Riverside, Riverside, CA, USA.

His research interests are real-time embedded systems, cyber–physical systems (CPS), and intelligent autonomous systems including self-driving vehicles, especially the design and implementation of a robust, and predictable scheduling algorithm with an analyzable framework under dynamic timing constraints.

**Hyoseung Kim** (Member, IEEE) received the Ph.D. degree in electrical and computer engineering from Carnegie Mellon University, Pittsburgh, PA, USA, in 2016.

He is an Assistant Professor with the Department of Electrical and Computer Engineering, University of California at Riverside, Riverside, CA, USA. He aims to achieve safe, predictable, and efficient computing under uncertain environments, which is critical for the continued innovation of intelligent autonomous technologies, including self-driving cars and smart health and agriculture. His research interests are in real-time embedded and cyber–physical systems, especially at the intersection of systems software, parallel computing platforms, and analytical techniques.

Dr. Kim is a recipient of the NSF CAREER Award, the Fulbright Scholarship Award, and the Best Paper Awards from RTAS and RTCSA. For more information, please visit http://www.ece.ucr.edu/hyoseung/.



**Qi Zhu** (Member, IEEE) received the Ph.D. degree in electrical engineering and computer science from the University of California Berkeley, Berkeley, CA, USA, in 2008.

He is an Associate Professor with the Department of Electrical and Computer Engineering, Northwestern University, Evanston, IL, USA. His research interests include design automation for intelligent cyber–physical systems and IoT applications, cyber–physical security, machine learning for CPS/IoT, energy-efficient CPS, and system-on-chip design.