Computational Complexity and Speedup Factors Analyses for Self-Suspending Tasks

Jian-Jia Chen
Department of Informatics, TU Dortmund University, Germany

***Abstract*—In computing systems, an execution entity (job/process/task) may suspend itself when it has to wait for some activities to continue/finish its execution. For real-time embedded systems, such self-suspending behavior has been shown to cause substantial performance/schedulability degradation in the literature. There are two commonly adopted self-suspending sporadic task models in real-time systems: 1) dynamic self-suspension and 2) segmented self-suspension sporadic task models. A dynamic self-suspending sporadic task is specified with an upper bound on the maximum suspension time for a job (task instance), which allows a job to dynamically suspend itself as long as the suspension upper bound is not violated. By contrast, a segmented self-suspending sporadic task has a predefined execution and suspension pattern in an interleaving manner.**

Even though some seemingly positive results have been reported for self-suspending task systems, the computational complexity and the theoretical quality (with respect to speedup factors) of fixed-priority preemptive scheduling have not been reported. This paper proves that the schedulability analysis for fixed-priority preemptive scheduling even with only one segmented self-suspending task as the lowest-priority task is $co\mathcal{NP}$-hard in the strong sense. For dynamic self-suspending task systems, we show that the speedup factor for any fixed-priority preemptive scheduling, compared to the optimal schedules, is not bounded by a constant or by the number of tasks, if the suspension time cannot be reduced by speeding up. Such a statement of unbounded speedup factors can also be proved for earliest-deadline-first (EDF), least-laxity-first (LLF), and earliest-deadline-zero-laxity (EDZL) scheduling algorithms. However, if the suspension time can be reduced by speeding up coherently or the suspension time of each task is not comparable with (i.e., sufficiently smaller than) its relative deadline, then we successfully show that rate-monotonic scheduling has a constant speedup factor, with respect to the optimal schedules, for implicit-deadline task systems.

1 Introduction

Advanced embedded real-time computing systems for safety-critical applications have timing requirements to ensure the functional correctness and timeliness. The seminal work by Liu and Layland [26] considered the scheduling of periodic tasks. More advanced task models have been designed in the past decades to improve the expressiveness of the task models to match the system behavior. To validate whether their deadlines will be met at run-time in the resulting schedule, the *schedulability* of a set of such tasks has to be provided.

One important assumption in most of these schedulability analyses is that a job does not suspend itself. Such an assumption enables the widely-adopted critical instant theorem [26], the busy-window concept [22], etc. When a task can suspend itself, most of such existing schedulability analyses in many scheduling algorithms cannot be applied without any modifications. Self-suspension can happen due to the following scenarios: (1) the latency of the memory accesses and I/O peripherals is hidden by using direct memory access (DMA), (2) there are external devices for accelerating the computation by using computation offloading, (3) another task on another processor has already held the resource (e. g., locked semaphores) required by the task to finish its computation, etc. In those cases, a job may suspend itself and release the processor to let the processor idle or to run another job (even with lower priorities) to improve the execution efficiency.

Therefore, self-suspension has become increasingly important for many applications. For more details, please refer to the recent review paper [9] for scheduling self-suspending tasks in real-time systems. Although the investigation of the impact of self-suspension behavior in real-time systems has been started since 1990, the literature of this research topic has been seriously flawed, including [1]–[3], [11], [18]–[20], [28], as reported in [9].

We consider a system of n sporadic self-suspending tasks. A sporadic task τ_i releases an infinite number of jobs that arrive with the minimum inter-arrival time constraint. A sporadic real-time task τ_i is characterized by its *worst-case execution time* C_i, its *minimum inter-arrival time* (also known as period) T_i and its *relative deadline* D_i. In addition, each job of task τ_i has also a specified worst-case self-suspension time S_i. If the relative deadline D_i of task τ_i in the task set is always equal to (no more than, respectively) the period T_i, such a task set is called an *implicit-deadline* (a *constrained-deadline*, respectively) task set (system).

There are two models that are widely used in the literature: *dynamic* and *segmented* self-suspension (sporadic) task models. The dynamic self-suspension model allows a job of task τ_i to suspend itself at any moment before it finishes as long as the worst-case self-suspension time S_i is not violated. The *segmented* self-suspension model further characterizes the computation segments and suspension intervals as an array $(C_i^1, S_i^1, C_i^2, S_i^2, ..., S_i^{m_i-1}, C_i^{m_i})$, composed of m_i computation segments separated by $m_i - 1$ suspension intervals.

Both of the above self-suspension models are meaningful and important. The dynamic self-suspension model has high flexibility since it does not need the system designers to detail the suspension and execution behavior very precisely. However, if the suspension patterns are well-defined and can be characterized with known suspension intervals, the flexibility of the dynamic self-suspension model can make the analysis and scheduling design rather pessimistic, whilst the segmented self-suspension model is more appropriate. Therefore, these two models are used for different scenarios.

For self-suspending task systems, there are two separated

This paper has been supported by DFG, as part of the Collaborative Research Center SFB876 (http://sfb876.tu-dortmund.de/).

problems: 1) how to *design scheduling policies* to schedule the self-suspending tasks and 2) how to *validate* the schedulability of a scheduling algorithm. In this paper, the former is referred to as the *scheduler design* problem, whilst the latter is referred to as the *schedulability test* problem.

Computational complexity: It was shown by Ridouard et al. [31] that the scheduler design problem for the segmented self-suspension task model is $\mathcal{NP}$-hard in the strong sense.[1] The proof in [31] only needs each segmented self-suspending task to have one self-suspension interval with two computation segments.

Lakshmanan and Rajkumar [20] proposed a pseudo-polynomial-time worst-case response time analysis, based on a revised critical instant theorem from [26], for a special case, in which there are $n-1$ ordinary sporadic tasks without any self-suspension and one segmented self-suspending task as the lowest-priority task. This has been recently disproved by Nelissen et al. [29]. The sufficient schedulability test by Nelissen et al. [29] requires exponential-time complexity even when the task system has *only one self-suspending task*. The other solutions [14] [30] require pseudo-polynomial time complexity but are only sufficient schedulability tests.

Table I summarizes the computational complexity mentioned above, in which the computational complexity of the schedulability tests for dynamic-priority scheduling can be found in Section 2.2.

Speedup Factors: Since real-time systems focus on the worst-case properties to meet or to miss the deadlines, direct approximation on the schedulability answers is usually not possible. Alternatively, researchers have widely used the resource augmentation bound or the speedup factor to quantify the imperfectness of the scheduling algorithms and the schedulability tests [17]. If an algorithm $\mathcal{A}$ has a *speedup factor* ρ, then it guarantees that the schedule derived from the algorithm $\mathcal{A}$ is always feasible by running at speed ρ, if the input task set admits a feasible schedule on a unit-speed processor.

Under the setting of self-suspending tasks, there are two options for speeding up. If the suspension length cannot be reduced by changing the local execution platform (e.g., due to computation offloading), then speeding up the processor only affects the execution time but the self-suspension time remains the same. If the suspension length can also be *coherently* reduced by changing the local execution platform (e.g., due to multiprocessor synchronization), then we assume that speeding up affects both the execution time and the self-suspension time coherently. The former is termed as the speedup factor as usual, and the latter is termed as the *suspension-coherent speedup factor*.

For the segmented self-suspension task model, Chen and Liu [7] and Huang and Chen [15] proposed to use release time enforcement, termed as fixed-relative-deadline (FRD), to have bounded speedup factors under dynamic-priority scheduling and fixed-priority scheduling, respectively. Specifically, the scheduling algorithm in [7] has a speedup factor 3 when each task can only suspend at most once. The scheduling algorithm in [15] has a speedup factor $(M+1)^2$ when each task can only suspend at most M times. For the dynamic self-suspension task model, Huang et al. [16] provided a fixed-priority scheduling policy to find a good priority assignment. A speedup factor 2 with respect to the *optimal fixed-priority schedule* (instead of the *optimal schedule*) was provided in [16]. However, whether the optimal fixed-priority schedule has a bounded speedup factor with respect to the optimal schedule, for the dynamic self-suspension task model, remains as an open problem.

The above results [7], [15], [16] are the only results (in the literature) that provide speedup factor upper bounds for self-suspending sporadic task systems. There have been also a few lower bounds in the literature. For both self-suspension task models, EDF and Rate-Monotonic (RM) scheduling algorithms (known as very good traditional real-time scheduling algorithms when there is no self-suspension) do not have any speedup factor bound as shown in [7], [16], [31].

Contributions: This paper provides the following *negative* results of the scheduler design problem and the schedulability test problem for self-suspending sporadic task systems:

- In Section 3, we prove that the schedulability analysis for fixed-priority (FP) preemptive scheduling even with only one segmented self-suspending task as the lowest-priority task is $co\mathcal{NP}$-hard in the strong sense when there are more than one self-suspension interval (or equivalently more than two computation segments). The computational complexity analysis is valid for both implicit-deadline and constrained-deadline cases, when the priority assignment is given. Our proof also shows that validating whether there exists a feasible priority assignment is $co\mathcal{NP}$-hard in the strong sense for constrained-deadline segmented self-suspending task systems.
- For dynamic self-suspending task systems, Section 4 shows that the speedup factor for any FP preemptive scheduling, compared to the optimal schedules, is not bounded by a constant if the suspension time cannot be reduced by speeding up. Such a statement of unbounded (by a constant or by the number of tasks) speedup factors can also be proved for earliest-deadline-first (EDF), least-laxity-first (LLF), and earliest-deadline-zero-laxity (EDZL) scheduling algorithms. How to design good schedulers with a constant speedup factor remains as an open problem.

Moreover, the following *positive* result for implicit-deadline task systems is provided for speedup factors in Section 5.

- If the suspension time can also be reduced by speeding up coherently, then we show that RM scheduling has a constant suspension-coherent speedup factor. Furthermore, if S_i/D_i is small for each task τ_i, we also show that the speedup factor can be bounded.

2 System Models and Preliminary Results

We assume a system $\mathbf{T}$ composed of n sporadic self-suspending tasks. A sporadic task τ_i is released repeatedly, with each such invocation called a job. The j^{th} job of τ_i, denoted by $\tau_{i,j}$, is released at time $r_{i,j}$ and has an absolute deadline at time $d_{i,j}$. Each job of task τ_i is assumed to have a worst-case execution time C_i. Furthermore, a job of task τ_i may suspend itself for at most S_i time units (across all of its suspension phases). When a job suspends itself, it releases the processor and another job can be executed. The response time of a job is defined as its finishing time minus its release time.

[1] Ridouard et al. [31] termed this problem as the feasibility problem for the decision version to verify the existence of a feasible schedule.

Task Model	Scheduler design problem	Schedulability test problem		
Segmented self-suspension	strongly $\mathcal{NP}$-hard [31]	Fixed-Priority Scheduling	Dynamic-Priority Scheduling	
			Constrained Deadlines	Implicit Deadlines
		strongly $co\mathcal{NP}$-hard (this paper)	strongly $co\mathcal{NP}$-hard (special case is [12])	strongly co$\mathcal{NP}$-hard (explained in this paper)
Dynamic self-suspension	unbounded (by a constant or by the number of tasks) speedup factors in any FP scheduling, EDF, EDZL, etc. (this paper) unknown computational complexity	unknown	strongly $co\mathcal{NP}$-hard (special case is [12])	unknown

TABLE I: The computational complexity classes (and some speedup factors) of the scheduler design problem and the schedulability test problem for self-suspending tasks.

Each task τ_i is characterized by the tuple (C_i, S_i, D_i, T_i), where T_i is the period (or minimum inter-arrival time) of τ_i and D_i is its relative deadline. T_i specifies the minimum time between two consecutive job releases of τ_i, while D_i defines the maximum amount of time a job can take to complete its execution after its release. It results that for each job $\tau_{i,j}$, $d_{i,j} = r_{i,j} + D_i$ and $r_{i,j+1} \geq r_{i,j} + T_i$. In this paper, we focus on constrained-deadline tasks, for which $D_i \leq T_i$. The utilization of a task τ_i is defined as $U_i = C_i/T_i$. Without loss of generality, we also implicitly assume that $0 < C_i$, $0 < S_i < D_i$ and $C_i + S_i \leq D_i$ for every task τ_i in $\mathbf{T}$.

The *dynamic* self-suspension task model allows a job of task τ_i to suspend at any moment before it finishes as long as the worst-case self-suspension time S_i is not violated. The *segmented* self-suspension task model further characterizes the computation segments and suspension intervals as an array $(C_i^1, S_i^1, C_i^2, S_i^2, ..., S_i^{m_i-1}, C_i^{m_i})$, composed of m_i computation segments separated by $m_i - 1$ suspension intervals.

We consider uniprocessor preemptive scheduling. If there is no self-suspension, the preemptive earliest-deadline-first (EDF) algorithm (as a dynamic-priority scheduling policy) is optimal [26] to meet the timing constraints. Due to the high overhead of EDF scheduling, fixed-priority (FP) scheduling has been also introduced to reduce the scheduling overhead by assigning a task with a fixed-priority level. FP scheduling has been widely adopted and also supported in most real-time operating systems. If there is no self-suspension, rate-monotonic (RM) scheduling [26] and deadline-monotonic (DM) scheduling [24] are optimal for uniprocessor FP scheduling for implicit-deadline and constrained-deadline task systems, respectively.

We say that a schedule is *feasible* if all the temporal characteristics and timing constraints are respected and satisfied. Moreover, a task system (set) is *schedulable* by a scheduling algorithm if the resulting schedule is always feasible. A *schedulability test* of a scheduling algorithm for a given task system is to validate whether the task system is schedulable by the scheduling algorithm. A *sufficient* schedulability test provides only sufficient conditions for validating the schedulability of a task system. A *necessary* schedulability test provides only necessary conditions to allow the schedulability of a task system. An *exact* schedulability test provides necessary and sufficient conditions for validating the schedulability.

If a scheduling algorithm $\mathcal{A}$ has a *speedup factor* ρ, then it guarantees that the schedule derived from the scheduling algorithm $\mathcal{A}$ is always feasible by running at speed ρ, if the input task set admits a feasible schedule on a unit-speed processor. The negation of the above statement is therefore useful to quantify the theoretical quality of the scheduling algorithm $\mathcal{A}$ as follows: If the algorithm $\mathcal{A}$ fails to provide a feasible schedule for an input task system, then there does not exist any feasible schedule (in the worst cases) for the task set when the execution time of each task τ_i in $\mathbf{T}$ (running at speed $1/\rho$) becomes $\rho \cdot C_i$.

The above definition does not allow the self-suspension time to be also reduced by speeding up. As explained in Section 1, we also consider the *suspension-coherent speedup factor* defined as follows: If the algorithm $\mathcal{A}$ fails to provide a feasible schedule for an input task system, then there does not exist any feasible schedule (in the worst cases) for the task set when the execution time of each task τ_i in $\mathbf{T}$ (running at speed $1/\rho$) becomes $\rho \cdot C_i$ and the self-suspension time of each task τ_i in $\mathbf{T}$ (suspending at speed $1/\rho$) becomes $\rho \cdot S_i$.

2.1 Scheduler Design Problem

The scheduler design problem is to design a scheduling algorithm to handle self-suspending tasks. For the segmented self-suspension task model, this problem is $\mathcal{NP}$-hard in the strong sense [31]. For the dynamic self-suspension task model, the computational complexity of this problem remains unknown. To validate whether the resulting schedules are feasible or not, sufficient or exact schedulability tests for the scheduling algorithm should also be provided.

For FP scheduling, the scheduler design problem is to define the corresponding fixed-priority assignment. Without self-suspension, it has been shown by Davis et al. [10] that the speedup factors of RM and DM are 1.4427 and 1.7322, respectively. However, with self-suspension, RM and DM do not have any speedup factor bound as shown in [7], [16], [31].

2.2 Schedulability Test Problem

To analyze the speedup factors of scheduling algorithms for dynamic self-suspension task systems, the following necessary conditions will be used in Section 4.

Lemma 1 (Huang et al. [16]): (**necessary condition for FP scheduling**) If a constrained-deadline dynamic self-suspension sporadic task system $\mathbf{T}$ is schedulable under a fixed-priority preemptive scheduling algorithm, then, for each task $\tau_k \in \mathbf{T}$

$$\exists 0 < t \leq D_k, \text{ s. t. } C_k + S_k + \sum_{\tau_i \in hp(\tau_k)} \left\lceil \frac{t}{T_i} \right\rceil C_i \leq t, \quad (1)$$

where $hp(\tau_k)$ is the set of the tasks with higher-priority than task τ_k in the fixed-priority scheduling algorithm.

Proof: This was proved by Huang et al. in [16, Theorem 3]. ■

Lemma 2: (**necessary condition for any scheduling**) If a constrained-deadline dynamic self-suspension sporadic task system $\mathbf{T}$ is schedulable under a scheduling algorithm, then

$$\forall 0 < t, \qquad \sum_{\tau_i \in \mathbf{T}} \max\left\{0, \left(\left\lfloor \frac{t-(D_i-S_i)}{T_i} \right\rfloor + 1\right) \cdot C_i\right\} \leq t. \tag{2}$$

Proof: The proof is in Appendix A. ■

Notably, for self-suspending task systems, the computational complexity of the schedulability test problem has not been explicitly explored in the literature. Here are a few cases for dynamic-priority scheduling (as shown in Table I) that can be concluded by using the result by Ekberg and Wang [12].

Dynamic-priority scheduling for constrained-deadline self-suspending tasks: For this case, the complexity class of this problem is at least co$\mathcal{NP}$-hard in the strong sense, since a special case of this problem is co$\mathcal{NP}$-complete in the strong sense [12]. Ekberg and Wang [12] proved that verifying uniprocessor feasibility of ordinary sporadic tasks (under EDF) with constrained deadlines is strongly co$\mathcal{NP}$-complete. Therefore, when we consider constrained-deadline self-suspending task systems, for both dynamic and segmented self-suspension task models, the complexity class is at least co$\mathcal{NP}$-hard in the strong sense.

Dynamic-priority scheduling for implicit-deadline segmented self-suspending tasks: A special case of the segmented self-suspending task system is to allow each task τ_i having exactly one self-suspension interval with a *fixed* length S_i and one computation segment with (worst-case) execution time C_i. Therefore, the relative deadline of the computation segment of task τ_i (after it is released to be scheduled) is $D_i = T_i - S_i$. For such a special case, the optimal scheduling policy is EDF. Similarly, such a case is also co$\mathcal{NP}$-hard in the strong sense.

3 Computational Complexity

In this section, we will prove that the schedulability test problem for FP preemptive scheduling even with only one *segmented* self-suspending task as the lowest-priority task in the task system is *co*$\mathcal{NP}$-hard in the strong sense. Specifically, we will also show that our reduction implies that finding whether there exists a feasible priority assignment (the scheduler design problem) under FP scheduling for constrained-deadline task systems is also *co*$\mathcal{NP}$-hard in the strong sense. We will first consider constrained-deadline task systems and then revise the reduction to consider implicit-deadline task systems.

Our reduction is from the 3-PARTITION problem [13]:[2]

Definition 1 (3-PARTITION Problem): We are given a positive integer V, a positive integer M, and a set of $3M$ integer numbers $\{v_2, v_3, \ldots, v_{3M+1}\}$ with the condition $\sum_{i=2}^{3M+1} v_i = MV$, in which $V/4 < v_i < V/2$ and $M \geq 3$. Therefore, $V \geq 3$.
Objective: The problem is to partition the given $3M$ integer numbers into M disjoint sets $\mathbf{V}_1, \mathbf{V}_2, \ldots, \mathbf{V}_M$ such that the sum of the numbers in each set $\mathbf{V}_i$ for $i = 1, 2, \ldots, M$ is V, i.e., $\sum_{v_j \in \mathbf{V}_i} v_j = V$. □

[2]For notational consistency and brevity in our reduction, we index the $3M$ integer numbers from 2.

The decision version of the 3-PARTITION problem to verify whether such a partition into M disjoint sets exists or not is known $\mathcal{NP}$-complete in the strong sense [13] when $M \geq 3$.

3.1 Constrained-Deadline Task Systems

Definition 2 (Reduction to a constrained-deadline system): For a given input instance of the 3-PARTITION problem, we construct $n = 3MV + 2$ sporadic tasks as follows:

- For task τ_1, we set $C_1 = V, S_1 = 0, D_1 = V, T_1 = 3V$.
- For task τ_i with $i = 2, 3, \ldots, 3M+1$, we set $C_i = v_i, S_i = 0, T_i = 21MV$ and $D_i = 3MV/2$ if M is an even number or $D_i = 3MV/2 + V/2$ if M is an odd number.
- For task τ_{3M+2}, we create a segmented self-suspending task with M computation segments separated by $M-1$ self-suspension intervals, i.e., $m_{3M+2} = M$, in which $C^j_{3M+2} = V + 1$ for $j = 1, 2, \ldots, M$, $S^j_{3M+2} = 6V$ for $j = 1, 2, \ldots, M-1$, $D_{3M+2} = M(4V+1) - V + 6V(M-1) = 10MV + M - 7V$, and $T_{3M+2} = 21MV$.

Due to the stringent relative deadline of task τ_1, it must be assigned as the highest-priority task. Moreover, the $3M$ tasks, i.e., $\tau_2, \tau_3, \ldots, \tau_{3M+1}$, created by using the integer numbers from the 3-PARTITION problem instance are assigned lower priorities than task τ_1 and higher priorities than task τ_{3M+2}. Since the integer numbers in the 3-PARTITION problem instance are given in an arbitrary order, without loss of generality, we index the tasks in $\tau_2, \tau_3, \ldots, \tau_{3M+1}$ by the given priority assignment, i.e., a lower-indexed task has higher priority. (In fact, we can also assign all these $3M$ tasks with the same priority level.) □

For the rest of the proof, the task set created in Definition 2 is referred to as $\mathbf{T}^{red}$.

Lemma 3: Tasks $\tau_1, \tau_2, \ldots, \tau_{3M+1}$ in $\mathbf{T}^{red}$ can meet their deadlines under the specified FP scheduling.

Proof: In FP scheduling, the segmented self-suspending task τ_{3M+2} in $\mathbf{T}^{red}$ has no impact on the schedule of the higher-priority tasks. Therefore, we can use the standard schedulability test for FP scheduling to verify their schedulability. The schedulability of task τ_1 is obvious since $C_1 \leq D_1$. For $i = 2, 3, \ldots, 3M+1$, task τ_i is schedulable under FP scheduling since $C_i + \sum_{j=1}^{i-1} \left\lceil \frac{D_i}{T_j} \right\rceil C_j = \left\lceil \frac{D_i}{T_1} \right\rceil C_1 + \sum_{j=2}^{i} C_j \leq \left\lceil \frac{D_i}{3V} \right\rceil V + MV = D_i$, where the last equality is due to

- $\left\lceil \frac{D_i}{3V} \right\rceil = \left\lceil \frac{3MV/2}{3V} \right\rceil = M/2$ when M is an even number;
- $\left\lceil \frac{D_i}{3V} \right\rceil = \left\lceil \frac{3MV/2+V/2}{3V} \right\rceil = \left\lceil \frac{M}{2} + \frac{1}{6} \right\rceil = (M+1)/2$ when M is an odd number.

■

The worst-case response time of task τ_{3M+2} happens by using one of the release patterns with the conditions in Lemma 4:

Lemma 4: The worst-case response time of task τ_{3M+2} in $\mathbf{T}^{red}$ under FP scheduling happens under the following necessary conditions:

1) Task τ_{3M+2} releases a job at time 0. This job requests the worst-case execution time per computation segment and suspends in each self-suspension interval exactly equal to its worst case.

2) Task τ_1 always releases one job together with each computation segment of the job (released at time 0) of task τ_{3M+2}, and releases the subsequent jobs strictly periodically with period $3V$ until a computation segment of task τ_{3M+2} finishes. Task τ_1 never releases any job when task τ_{3M+2} suspends itself.

3) For $i = 2, 3, \ldots, 3M+1$, task τ_i only releases one job together with one of the M computation segments of the job (released at time 0) of task τ_{3M+2}.

All the jobs and all the computation segments are executed with their worst-case execution time specifications.

Proof: For completeness, the proof for the release pattern is in Appendix A and [4]. ∎

For the j-th computation segment of task τ_{3M+2}, suppose that $\mathbf{T}_j \subseteq \{\tau_2, \tau_3, \ldots, \tau_{3M+1}\}$ is the set of the tasks released together with C^j_{3M+2} (under the third condition in Lemma 4). For notational brevity, let w_j be $\sum_{\tau_i \in \mathbf{T}_j} C_i$. By definition, w_j is a non-negative integer. Together with the second condition in Lemma 4, we can use the standard time demand analysis to analyze the worst-case response time R_j of the j-th computation segment of task τ_{3M+2} (after it is released) under the higher-priority interference contributed from $\{\tau_1\} \cup \mathbf{T}_j$. The response time R_j of a computation segment C^j_{3M+2} is defined as the finishing time of the computation segment minus the arrival time of the computation segment.

For a given task set $\mathbf{T}_j$ (i.e., a given non-negative integer w_j), R_j is the minimum t with $t > 0$ such that

$$C^j_{3M+2} + (\sum_{\tau_i \in \mathbf{T}_j} C_i) + \left\lceil \frac{t}{T_1} \right\rceil C_1 = V + 1 + w_j + \left\lceil \frac{t}{3V} \right\rceil V = t.$$

Since R_j only depends on the non-negative integer w_j, we use $R(w_j)$ to represent R_j for a given $\mathbf{T}_j$. We know that $V + 1 + w_j + \lceil \frac{t}{3V} \rceil V = t$ happens with $\ell \cdot 3V < t \leq (\ell+1) \cdot 3V$ for a certain non-negative integer ℓ. That is, $V+1+w_j+\lceil \frac{t}{3V} \rceil V > t$ when t is $\ell \cdot 3V$ and $V + 1 + w_j + \lceil \frac{t}{3V} \rceil V \leq t$ when t is $(\ell+1)3V$. We know that ℓ is $\left\lceil \frac{V+1+w_j}{2V} \right\rceil - 1$. Moreover,

$$\begin{aligned} R(w_j) &= \ell \cdot 3V + V + (V + 1 + w_j - \ell \cdot 2V) \\ &= 2V + 1 + w_j + \ell \cdot V \\ &= V + 1 + w_j + \left\lceil \frac{V+1+w_j}{2V} \right\rceil V. \end{aligned}$$

This leads to three cases that are of interest:

$$R(w_j) = \begin{cases} 2V + 1 + w_j & \text{if } w_j \leq V - 1 \\ 4V + 1 & \text{if } w_j = V \\ V + 1 + w_j + \left\lceil \frac{V+1+w_j}{2V} \right\rceil V & \text{if } w_j > V \end{cases} \tag{3}$$

For example, if $w_j = 3V - 1$, then $R(w_j)$ is $6V$; if w_j is $3V$, then $R(w_j)$ is $7V + 1$.

With the above discussions, we can now conclude the unique condition when task τ_{3M+2} misses its deadline in the following lemma.

Lemma 5: Suppose that $\mathbf{T}_j \subseteq \{\tau_2, \tau_3, \ldots, \tau_{3M+1}\}$ and $\mathbf{T}_i \cap \mathbf{T}_j = \emptyset$ when $i \neq j$. Let $w_j = \sum_{\tau_i \in \mathbf{T}_j} C_i$. If a task partition $\mathbf{T}_1, \mathbf{T}_2, \ldots, \mathbf{T}_M$ exists such that $\sum_{j=1}^{M} R(w_j) > M(4V+1) - V$ with $R(w_j)$ defined in Eq. (3), then task τ_{3M+2} misses its deadline in the worst case; otherwise, task τ_{3M+2} always meets its deadline.

Proof: By Lemma 4, task τ_{3M+2} in $\mathbf{T}^{red}$ is not schedulable under the fixed-priority preemptive scheduling if and only if there exists a task partition $\mathbf{T}_1, \mathbf{T}_2, \ldots, \mathbf{T}_M$ such that $\sum_{j=1}^{M} R(w_j) + \sum_{j=1}^{M-1} S^j_{3M+2} = 6(M-1)V + \sum_{j=1}^{M} R(w_j) > D_{3M+2} = M(4V+1) + 6V(M-1) - V$. This concludes the proof. ∎

Instead of investigating the combinations of the task partitions, we analyze the corresponding total worst-case response time $\sum_{j=1}^{M} R(w_j)$ for the M computation segments of task τ_{3M+2} (by excluding the self-suspension time) by considering different *non-negative integer assignments* $w_1, w_2, \ldots, w_M$ with $\sum_{i=1}^{M} w_i = MV$ and $w_i \geq 0$ in the following lemmas.

Lemma 6: If $w_1 = w_2 = \cdots = w_M = V$, then

$$\sum_{j=1}^{M} R(w_j) = M(4V+1),$$

where $R(w_j)$ is defined in Eq. (3).

Proof: This comes directly by Eq. (3). ∎

Lemma 7: For any non-negative integer assignment for $w_1, w_2, \ldots, w_M$ with $\sum_{i=1}^{M} w_i = MV$, if there exists a certain index j with $w_j \neq V$, then

$$\sum_{j=1}^{M} R(w_j) \leq M(4V+1) - V,$$

where $R(w_j)$ is defined in Eq. (3).

Proof: Let $\mathbf{X}$ be the set of indexes such that $0 \leq w_j < V$ for any $j \in \mathbf{X}$. Similarly, let $\mathbf{Y}$ be the set of indexes such that $V < w_j$ for any $j \in \mathbf{Y}$. If $j \notin \mathbf{X} \cup \mathbf{Y}$, then w_j is V. (A concrete example of the following steps can be found in Appendix B.)

If there exists j in $\mathbf{Y}$ with $w_j > 2V$, since $\sum_{i=1}^{M} w_i = MV$, there must exist an index i in $\mathbf{X}$ with $w_i < V$. We can increase w_i to $w'_i = V$, which increases the worst-case response time $R(w_i)$ by $2V - w_i$ (i.e., from $2V+1+w_i$ to $4V+1$). Simultaneously, we reduce w_j to $w'_j = w_j - (V - w_i) > V$. Therefore, $w_i + w_j = w'_i + w'_j$. Moreover, the reduction of w_j to w'_j also reduces the worst-case response time $R(w_j)$ by case 1) $V - w_i$ if $\left\lceil \frac{V+1+w'_j}{2V} \right\rceil$ is equal to $\left\lceil \frac{V+1+w_j}{2V} \right\rceil$, and by case 2) $V - w_i + V$ if $\left\lceil \frac{V+1+w'_j}{2V} \right\rceil$ is not equal to $\left\lceil \frac{V+1+w_j}{2V} \right\rceil$. In both cases, we can easily see that the worst-case response time is not decreased in the new integer assignment. Moreover, the index j remains in $\mathbf{Y}$ and the index i is removed from set $\mathbf{X}$. We repeat the above step until all the indexes j in $\mathbf{Y}$ are with $w_j \leq 2V$.

It is clear that $\mathbf{X}$ and $\mathbf{Y}$ are both non-empty after the above step. For the rest of the proof, let $\mathbf{X}$ and $\mathbf{Y}$ be defined after finishing the above step. Therefore, the condition $w_j \leq 2V$ holds for any $j \in \mathbf{Y}$. Due to the pigeon-hole principle, when $\mathbf{Y}$ is not an empty set, $\mathbf{X}$ is also not an empty set. Moreover, for an element i in $\mathbf{X}$, there must be a subset $\mathbf{Y}' \subseteq \mathbf{Y}$ and an index $\ell \in \mathbf{Y}'$ such that

$$\sum_{j \in \mathbf{Y}'} (w_j - V) \geq V - w_i > \sum_{j \in \mathbf{Y}' \setminus \{\ell\}} (w_j - V).$$

That is, we want to adjust w_i to V (i.e., w_i is increased by $V-w_i$), and the set $\mathbf{Y}'\setminus\{\ell\}$ is not enough to match the integer adjustment $V-w_i$ and the set $\mathbf{Y}'$ is enough to match the integer adjustment $V-w_i$. We now increase w_i to V, which increases the worst-case response time $R(w_i)$ by $2V-w_i$. Simultaneously, we reduce w_j to V for every $j \in \mathbf{Y}' \setminus \{\ell\}$ and reduce w_ℓ to $w'_\ell = w_\ell - (V - w_i - \sum_{j\in\mathbf{Y}'\setminus\{\ell\}}(w_j - V))$. Since $V < w_j \leq 2V$ for any $j \in \mathbf{Y}'$ before the adjustment, the adjustment reduces the worst-case response time $R(w_j)$ by $w_j - V$ if $j \neq \ell$ and reduces $R(w_\ell)$ by $w_\ell - w'_\ell$. Therefore, the adjustment reduces $\sum_{j\in\mathbf{Y}'} R(w_j)$ by exactly $V - w_i$. Therefore, the adjustment in this step to change w_i in $\mathbf{X}$ and w_j in $\mathbf{Y}'$ increases the overall worst-case response time by exactly V time units.

By adjusting with the above procedure repeatedly, we will reach the integer assignment $w_1 = w_2 = \cdots = w_M = V$ with bounded increase of the worst-case response time. As a result, we can conclude that $\sum_{j=1}^{M} R(w_j) \leq M(4V+1) - |\mathbf{X}|V$. By the assumption $\sum_{i=1}^{M} w_i = MV$ and the existence of $w_j \neq V$ for some j, we know that $|\mathbf{X}|$ must be at least 1. Therefore, we reach the conclusion. ■

We can now conclude the $co\mathcal{NP}$-hardness.

Theorem 1: The schedulability analysis for FP scheduling even with only one segmented self-suspending task as the lowest-priority task in the sporadic task system is $co\mathcal{NP}$-hard in the strong sense, when the number of self-suspending intervals in the self-suspending task is more than or equal to 2 and $D_i \leq T_i$ for every task τ_i.

Proof: The reduction in Definition 2 requires polynomial time. Moreover, by Lemmas 4, 5, 6, and 7, a feasible solution of the 3-PARTITION problem for the input instance exists if and only if task τ_{3M+2} is not schedulable by the FP scheduling when $M \geq 3$. Therefore, this concludes the proof. ■

Corollary 1: Validating whether there exists a feasible priority assignment is $co\mathcal{NP}$-hard in the strong sense for constrained-deadline segmented self-suspending task systems.

Proof: This comes directly from Theorem 1 and the only possible priority level for task τ_{3M+2} to be feasible in $\mathbf{T}^{red}$. ■

Remarks for Computational Complexity Although the reduction in this section is limited to the special case with only a segmented self-suspending task as the lowest-priority task, the reduction can be easily extended for more than one self-suspending task, e.g., by allowing task τ_1 to suspend. However, the above proof of the strong $co\mathcal{NP}$-hardness does not hold when the number of tasks is a constant or there is only one self-suspension interval per task.

3.2 Implicit-Deadline Task Systems

The $co\mathcal{NP}$-hardness in the strong sense for testing the schedulability of task τ_n under FP scheduling can be easily proved with the same input as in $\mathbf{T}^{red}$ by changing the periods of the tasks as follows:

- For task τ_1, we set $D_1 = 3V, T_1 = 3V$.
- For task τ_i with $i = 2, ..., 3M+1$, we set $T_i = D_i = 10MV + M - 7V$.
- For task τ_{3M+2}, we set $T_{3M+2} = D_{3M+2} = 10MV + M - 7V$.

Assume that τ_{3M+2} is the lowest-priority task. It is not difficult to see that all the conditions in Lemma 4 still hold for testing whether task τ_{3M+2} can meet its deadline or not (but not for the worst-case response time if task τ_{3M+2} misses the deadline). Therefore, the schedulability analysis for FP scheduling even with only one segmented self-suspending task as the lowest-priority task in the sporadic task system is $co\mathcal{NP}$-hard in the strong sense, when the number of self-suspending intervals in the self-suspending task is more than or equal to 2 and $D_i = T_i$ for every task τ_i.

However, the above argument does not hold if we assign task τ_{3M+2} to the highest-priority level. Therefore, the above proof does not support a similar conclusion for implicit-deadline task systems to that for constrained-deadline task systems in Corollary 1.

4 Negative Results for Speedup Factors

This section presents a concrete input task system to show that any FP scheduling as well as several other scheduling algorithms do not admit any constant speedup factors for the dynamic self-suspension task model if self-suspension time cannot be reduced by speeding up. We consider the following specific task set with two implicit-deadline tasks:

- For task τ_1, we set $T_1 = D_1 = 1$, $S_1 = 0$, and $C_1 = B$.
- For task τ_2, we set $T_2 = D_2 = \frac{1}{B^2}$, $S_2 = \frac{1}{B^2}(1-B)$, and $C_2 = 1$.

In the above setting, we will implicitly assume that $0 < B \leq 0.25$ and $1/B$ is a positive integer. For notational brevity, let $\mathbf{T}^{negative}$ be the above task set, consisting of these two tasks. To prove the lower bounds of speedup factors, two steps will be involved:

1) We will first show that this task set $\mathbf{T}^{negative}$ can be in fact feasibly scheduled by a simple heuristic algorithm when the system runs at any speed faster than or equal to $2B(\frac{1/B-0.5}{1/B-1})$ in Section 4.1.
2) We will then show that this task set is not schedulable by several typical (well-motivated) scheduling algorithms at the original speed 1 in Sections 4.2 and 4.3.

By these, we will conclude that the speedup factor of these scheduling algorithms is at least $\frac{1}{2B}(\frac{1/B-1}{1/B-0.5})$, formally stated in Section 4.4. Since B can be arbitrarily small, this section concludes that the typical scheduling algorithms in real-time systems, including EDF, LLF, and FP, have unbounded speedup factors (by a constant or by the number of tasks) for the dynamic self-suspension task model.

4.1 How Much Can We Slow Down?

To prove the lower bound s of the speedup factor, we will first show that the above task system $\mathbf{T}^{negative}$ can still be feasibly scheduled by running at a processor with speed $\frac{1}{s}$. We can start with the necessary condition stated in Lemma 2. The necessary condition to have a feasible schedule is basically identical to consider the corresponding ordinary sporadic task set with τ_1 and τ'_2 in which $C'_2 = C_2$ and $D'_2 = T_2 - S_2 = \frac{1}{B}$ and $T'_2 = T_2$. By considering $t = \frac{1}{B}$ in Eq. (2), we know that

$$\sum_{\tau_i\in\mathbf{T}^{negative}} \max\left\{0, \left(\left\lfloor \frac{1/B - (D_i - S_i)}{T_i} \right\rfloor + 1\right)\cdot C_i\right\} = 2.$$

Therefore, we can conclude that this task set $\mathbf{T}^{negative}$ cannot be feasibly scheduled for any speed slower than $\frac{2}{1/B} = 2B$.

Since the necessary condition in Lemma 2 looks pretty pessimistic, it may leave a hope to rule out some speed higher than $2B$ that is in fact not able to admit any feasible schedule. We will design a specific scheduling algorithm for task system $\mathbf{T}^{negative}$ that can feasibly schedule tasks in $\mathbf{T}^{negative}$ under any speed higher than or equal to $2B(\frac{1/B-0.5}{1/B-1})$. This shows that the necessary condition provided in Lemma 2 is actually pretty tight for analyzing the speedup factors when B is small.

Towards this, we consider the following scheduling algorithm that is specifically designed for the above two tasks:

- An unfinished job of task τ_1 is executed at time t if 1) task τ_2 suspends at time t or 2) the *laxity* of the job of task τ_1 at time t equals to 0;
- otherwise an unfinished job of task τ_2 is executed at time t.

Here, the *laxity* of a job of task τ_1 at time t is defined as $d_j - t - c^*$, where d_j is the absolute deadline of the job and c^* is the remaining execution time of the job. The above algorithm is denoted as Algorithm $\mathcal{R}$.

For the rest of the proof, we will implicitly assume that the actual-case execution time and the actual-case suspension time are the same as their corresponding worst cases. Figure 1 provides an illustrative example of task set $\mathbf{T}^{negative}$ by using Algorithm $\mathcal{R}$ at speed $\frac{1}{s} = 0.25$ when $B = 0.1$. The job of task τ_2 finishes at time $95.6 \leq D_2 = 100$ in this concrete schedule. We have the following properties of Algorithm $\mathcal{R}$.

Lemma 8: Suppose that a job of task τ_1 arrives at time a_j on a processor with speed $\frac{1}{s}$ and the unfinished job of task τ_2 (arrived before a_j) has not yet finished at time $a_j + 1$ in the schedule by using Algorithm $\mathcal{R}$. Then, in the schedule by using Algorithm $\mathcal{R}$,

- **Case 1**: either task τ_2 is executed for exactly $1 - Bs$ amount of execution time in the time interval $[a_j, a_j + 1)$ and task τ_2 is executed again right at time $a_j + 1$,
- **Case 2**: or task τ_2 is executed for y amount execution time in the time interval $[a_j, a_j + 1)$ and suspends itself for $1 - y$ amount of time for some y with $0 \leq y \leq 1 - Bs$.

Proof: The execution time of task τ_1 is Bs at speed $1/s$. Therefore, from a_j to $a_j + 1$, task τ_2 can be executed by at most $1 - Bs$ amount of time. If the job of task τ_1 has to preempt task τ_2 since its laxity becomes 0 at some time $a_j + 1 - Bs \leq t < a_j + 1$, then we reach Case 1. If the job of task τ_1 does not have to preempt task τ_2, we reach Case 2. ■

Lemma 9: Suppose that a job of task τ_1 arrives at time a_j on a processor with speed $\frac{1}{s}$ and the unfinished job of task τ_2 (arrived before a_j) finishes at time f in time interval $[a_j, a_j + 1)$ in the schedule by using Algorithm $\mathcal{R}$. Then, in the schedule by using Algorithm $\mathcal{R}$,

- **Case 3**: task τ_2 is executed for y amount execution time in the time interval $[a_j, a_j + 1)$ and suspends itself for $f - a_j - y$ amount of time for some y with $0 \leq y < 1 - Bs$.

Proof: This is similar to Case 2 in Lemma 8. ■

In the example provided in Figure 1, we know that Case 1 holds for intervals $[3, 4)$, $[4, 5)$, $[91, 92)$, and $[93, 94)$, and Case 3 holds for interval $[95, 96)$. Case 2 holds for all the other intervals $[a_j, a_j + 1)$ with $a_j \neq 3, 4, 91, 93, 95$ with $a_j < 96$.

Theorem 2: Algorithm $\mathcal{R}$ can feasibly schedule task τ_1 and task τ_2 on a processor with speed $2B(\frac{1/B-0.5}{1/B-1})$.

Proof: According to Algorithm $\mathcal{R}$, any job of task τ_1 can always be finished in time when its laxity at time t becomes 0. Therefore, task τ_1 can be feasibly scheduled if $Bs \leq 1$.

Now, we examine the schedulability of task τ_2 under Algorithm $\mathcal{R}$. Suppose that a job of task τ_2 arrives at time r and finishes at time $f \leq r + T_2$ by using Algorithm $\mathcal{R}$. We do not have to consider any job of task τ_2 arrived before r if we can guarantee that the worst-case response time is no more than the minimal inter-arrival time. Suppose that task τ_1 releases its jobs at time $a_1, a_2, a_3, \ldots, a_\ell$ such that $r \leq a_1 \leq a_2 \leq \cdots \leq a_\ell < f$. By the definition of sporadic tasks, we also know that $a_i + 1 \leq a_{i+1}$. Moreover, we also know that there may be a job of task τ_1 arrived before r but still has not finished yet at time r. We use a_0 to denote the arrival time of this job of task τ_1. By definition $r \geq a_0 \geq r - 1$.

Now, suppose that the processor is with speed $\frac{1}{s}$, i.e., the execution time of task τ_2 is s at this speed. By Lemma 8, let $\mathbf{I}$ be the sub set of the arrival times $a_1, a_2, \ldots, a_\ell$ in which **Case 1** takes place. That is, the overal execution time of task τ_2 executed in the intervals $\cup_{i \in \mathbf{I}}[a_i, a_i + 1)$ is exactly $(1 - Bs)|\mathbf{I}|$. Moreover, in the other time intervals from $a_0 + 1$ to f, i.e., $[a_0 + 1, f) \setminus (\cup_{i \in \mathbf{I}}[a_i, a_i + 1))$, task τ_2 has to either execute or suspend (i.e., **Case 2** in Lemma 8 and **Case 3** in Lemma 9).

By the definition of $\mathbf{I}$, we know that $0 \leq |\mathbf{I}| \leq \left\lceil \frac{s}{1-Bs} \right\rceil - 1 < \frac{s}{1-Bs}$.[3] Therefore, the maximum overall suspension time and execution time allowed for task τ_2 in the time interval $[a_0 + 1, a_0 + \frac{1}{B^2}) \subseteq [r, r + \frac{1}{B^2})$ is at least

$$|\mathbf{I}|(1-Bs) + \frac{1}{B^2} - 1 - |\mathbf{I}| = \frac{1}{B^2} - 1 - |\mathbf{I}|Bs > \frac{1}{B^2} - 1 - \frac{Bs^2}{1 - Bs}.$$

As a result, if

$$\frac{1}{B^2} - 1 - \frac{Bs^2}{1 - Bs} \geq C_2 s + S_2 = s + \frac{1}{B^2}(1 - B),$$

then task τ_2 can be feasibly scheduled. That is, if

$$s \leq \frac{1/B - 1}{1 + B(1/B - 1)} = (\frac{1}{2B}) \left(\frac{1/B - 1}{1/B - 0.5} \right),$$

i.e., if $\frac{1}{s} \geq (2B) \left(\frac{1/B - 0.5}{1/B - 1} \right)$, then task τ_2 can be feasibly scheduled by Algorithm $\mathcal{R}$ under this reduced platform speed. ■

4.2 Fixed-Priority Scheduling

For task set $\mathbf{T}^{negative}$, setting task τ_2 as the higher-priority task makes task τ_1 miss the deadline since $C_2 + C_1 > T_1$. Setting task τ_1 as the higher-priority task also makes task τ_2 miss the deadline. This can be proved by using the necessary condition for fixed-priority scheduling provided in Lemma 1. For any $0 < t \leq T_2 = \frac{1}{B^2}$, we have

$$C_2 + S_2 + \left\lceil \frac{t}{T_1} \right\rceil C_1 - t = 1 + \frac{1}{B^2}(1 - B) + \lceil t \rceil \cdot B - t$$
$$\geq 1 + \frac{1}{B^2}(1 - B) - t(1 - B) \geq_a 1 + \frac{1}{B^2}(1 - B) - \frac{1}{B^2}(1 - B) > 0,$$

[3] The reason to have $\left\lceil \frac{s}{1-Bs} \right\rceil - 1$ instead of $\left\lfloor \frac{s}{1-Bs} \right\rfloor$ is due to the fact that task τ_2 has not yet finished its execution at the end of an interval with Case 1 in Lemma 8 and has to be resumed and executed. Therefore, if $\frac{s}{1-Bs}$ is an integer, there are at most $\frac{s}{1-Bs} - 1$ such intervals in set $\mathbf{I}$.

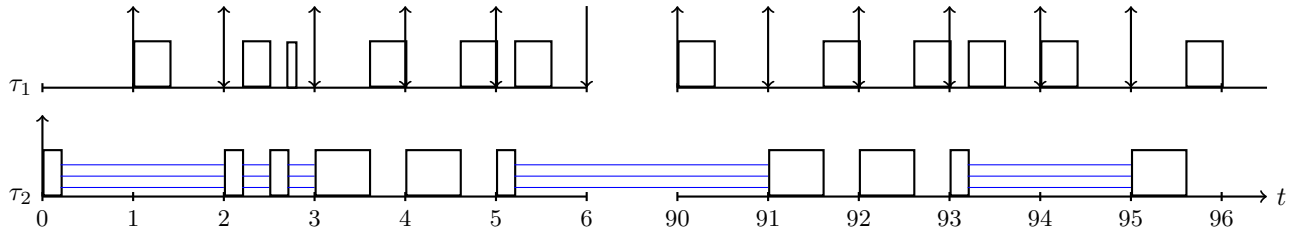

Fig. 1: An example of task set $\mathbf{T}^{negative}$ by using Algorithm $\mathcal{R}$ at speed $\frac{1}{s} = 0.25$ when $B = 0.1$. That is, the execution time of task τ_1 is 0.4 at this platform speed and the execution time of task τ_2 is 4 at this platform speed. Each smaller rectangle for task τ_2 is for executing task τ_2 for 0.2 time unit. Each larger rectangle for task τ_2 is for executing task τ_2 for 0.6 time unit. The job of task τ_2 finishes at time 95.6.

where $\geq_a$ is due to the assumption that $1 - B > 0$. By the above inequality, the necessary condition in Lemma 1 does not hold for scheduling task τ_2 as the lower-priority task. Therefore, we know that assigning task τ_1 as the higher-priority task makes task τ_2 miss the deadline.

Theorem 3: The speedup factor of any FP scheduling, compared to the optimal scheduling policy, for dynamic self-suspending task systems is not bounded by a constant.

Proof: By Theorem 2 and the fact that there does not exist any feasible FP scheduling for $\mathbf{T}^{negative}$ in the original speed, the speedup factor of any FP scheduling is at least $(\frac{1}{2B})\left(\frac{1/B-1}{1/B-0.5}\right)$, converging to ∞ when B approaches 0. ■

4.3 Unbounded Scheduling Algorithms

For the rest of this section, we provide a concrete job release pattern of task set $\mathbf{T}^{negative}$ to demonstrate the unbounded speedup factors of several algorithms. Task τ_1 releases jobs periodically from time 0 every one time unit. One job of task τ_2 arrives at time 0 and has the following execution/suspension pattern:

- It first requests to run on the processor for ϵ amount of time, and then it suspends for $1-B-\epsilon$ amount of time; the above execution/suspension pattern is repeated for $\frac{1}{B^2}(1-B)$ iterations (times).
- Then, it requests $1-\frac{\epsilon}{B^2}$ to run on the processor, followed by a suspension interval length $\frac{\epsilon}{B^2}(1-B)+\frac{1}{B}-1$.
- Then, it requests $\frac{\epsilon}{B}$ to run on the processor.

We assume that $0 < \epsilon < B^3$. Recall that $0 < B \leq 0.25$. Therefore, the overall execution time of the job of task τ_2 is $\frac{\epsilon}{B^2}(1-B)+1-\frac{\epsilon}{B^2}+\frac{\epsilon}{B} = 1$, and the overall self-suspension time of the job of task τ_2 is $\frac{1-B-\epsilon}{B^2}(1-B)+\frac{\epsilon}{B^2}(1-B)+\frac{1}{B}-1 = \frac{1}{B^2}(1-B)$. Hence, the above execution/suspension pattern of task τ_2 is valid. Our analyses for the unbounded speedup factors of several algorithms are mainly based on the observation made in the following lemma:

Lemma 10: For task set $\mathbf{T}^{negative}$, from time 0 to time $\frac{1}{B^2}(1-B)$, if the jobs of task τ_1 are always with higher priority than the job of task τ_2, then either task τ_1 or task τ_2 misses its deadline.

Proof: By the defined job release pattern, whenever task τ_1 is executed from time 0 to time $\frac{1}{B^2}(1-B)$, task τ_2 is blocked/preempted and τ_2 does not suspend itself. The schedule when $t = 0, 1, 2, \ldots, \frac{1}{B^2}(1-B)-1$ is as follows:

- $t = 0$: the job of task τ_1 finishes at B. Task τ_2 executes from B to $B+\epsilon$, suspends from $B+\epsilon$ to 1, and resumes at time 1.
- $t = 1, 2, \ldots, \frac{1}{B^2}(1-B)-1$: the above pattern repeats.

There are two cases for the execution behavior in time interval $[\frac{1}{B^2}(1-B), \frac{1}{B^2}(1-B)+1)$:

- Case 1 - the job of task τ_1 arrived at time $\frac{1}{B^2}(1-B)$ **does not complete its execution of B time units**: This concludes the deadline miss of task τ_1.
- Case 2 - the job of task τ_1 arrived at time $\frac{1}{B^2}(1-B)$ **completes its execution of B time units**: At time $\frac{1}{B^2}(1-B)$, the requested execution time of task τ_2 is $1-\frac{\epsilon}{B^2} > 1-B$, due to the assumption $0 < \epsilon < B^3$. The completion of the job of task τ_1 arrived at time $\frac{1}{B^2}(1-B)$ means that task τ_2 is blocked by task τ_1 for B time units, and, for its requested $1-\frac{\epsilon}{B^2}$ execution time units, only $1-B$ time units are executed from time $\frac{1}{B^2}(1-B)$ to time $\frac{1}{B^2}(1-B)+1$. The sum of the remaining execution time and the remaining suspension time of task τ_2 at time $\frac{1}{B^2}(1-B)$ is $1-\frac{\epsilon}{B^2}+\frac{\epsilon}{B^2}(1-B)+\frac{1}{B}-1+\frac{\epsilon}{B} = \frac{1}{B}$. Therefore, the sum of the remaining execution time and the remaining suspension time of task τ_2 at time $\frac{1}{B^2}(1-B)+1$ is $\frac{1}{B}-(1-B)$, which is strictly greater than the remaining time (to the deadline), i.e., $D_2-(\frac{1}{B^2}(1-B)+1) = \frac{1}{B}-1$. Therefore, task τ_2 misses its deadline.

■

By the following theorem, if a scheduling algorithm always gives the jobs of task τ_1 higher priority than the job of task τ_2 from time 0 to time $\frac{1}{B^2}(1-B)$, then the algorithm has an unbounded speedup factor.

Theorem 4: For task set $\mathbf{T}^{negative}$, from time 0 to time $\frac{1}{B^2}(1-B)$, if a scheduling algorithm always gives the jobs of task τ_1 higher priority than the job of task τ_2, then the speedup factor of the algorithm, compared to the optimal scheduling policy, for dynamic self-suspending task systems is not bounded by a constant or by the number of tasks.

Proof: By Theorem 2 and the fact that these schedules are not feasible for $\mathbf{T}^{negative}$ in the original speed due to Lemma 10, the speedup factor of these algorithms must be at least $(\frac{1}{2B})\left(\frac{1/B-1}{1/B-0.5}\right)$, which converges to ∞ when B is close to 0. ■

EDF. Earliest-Deadline-First (EDF) scheduling gives the job with the earliest absolute deadline the highest priority. For EDF, the unbounded speedup factor for dynamic self-suspension task systems was already provided by Huang et al. [16]. The EDF schedule of our defined job release pattern also fits the condition in Lemma 10 as well. A *seemingly* reasonable extension of EDF is the suspension-aware EDF (SAEDF) as follows:

- Suspension-aware absolute deadline at time t: the absolute

deadline of the job minus S^*, where S^* is the remaining upper bound on the suspension time of the job.

In SAEDF, the job with the earliest suspension-aware absolute deadline is scheduled.

Under SAEDF, we have to calculate the suspension-aware absolute deadlines. Since task τ_1 in $\mathbf{T}^{negative}$ does not suspend, its job arriving at time t has suspension-aware absolute deadline $t+1$. The suspension-aware absolute deadline of task τ_2 can change over time. Here, we detail the calculation when $t = 0, 1, 2, \ldots, \frac{1}{B^2}(1-B) - 1$ and the resulting schedule:

- $t = 0$: the job of task τ_1 has the earliest suspension-aware absolute deadline since $D_2 - S_2 = \frac{1}{B^2} - \frac{1}{B^2}(1-B) = 1/B \geq 4$ (due to our assumption $B \leq 0.25$). The first job of task τ_1 finishes at time B. Task τ_2 executes from B to $B+\epsilon$, suspends, and resumes at time $t = 1$.
- $t = 1, 2, \ldots, \frac{1}{B^2}(1-B) - 1$: the above pattern repeats since the suspension-aware absolute deadline of task τ_2 at time t is $D_2 - (S_2 - t\cdot(1-B-\epsilon)) > \frac{1}{B} + t - t\cdot(B+B^3) \geq t + \frac{1}{B} - (\frac{1}{B^2}(1-B) - 1)\cdot(B+B^3) = t + \cancel{\frac{1}{B}} - (\cancel{\frac{1}{B}} - 1 - \cancel{B}) - (\cancel{B} - B^2 - B^3) = t + 1 + B^2 + B^3 > t + 1$.

Therefore, the schedule of SAEDF from time 0 to time $\frac{1}{B^2}(1-B)$ is the same as the schedule defined in Lemma 10. As a result, the schedule produced by SAEDF for our defined job release pattern is not feasible.

Laxity-Based Scheduling Algorithms. We consider two definitions of the laxity of a job at time t:

- Suspension-unaware laxity at time t: the absolute deadline of the job minus $(t + e^*)$.
- Suspension-aware laxity at time t: the absolute deadline of the job minus $(t + e^* + S^*)$.

In the above definitions, e^* is the (worst-case) remaining execution time of a job, and S^* is the (worst-case) remaining suspension time.

The least-laxity-first (LLF) scheduling algorithm [23] is a dynamic-priority scheduling algorithm. It gives the job with the least laxity the highest priority. If the laxity is defined by using the suspension-unaware laxity, the scheduling algorithm is called suspension-unaware least-laxity-first (SULLF) scheduling algorithm. If the laxity is defined by using the suspension-aware laxity, the scheduling algorithm is called suspension-aware least-laxity-first (SALLF) scheduling algorithm. Before $t \leq \frac{1}{B^2}(1-B)$, the suspension-unaware laxity is trivially greater than or equal to $D_2 - t - C_2 \geq \frac{1}{B^2} - (\frac{1}{B^2}(1-B)+1) - 1 = \frac{1}{B} - 2 \geq 2$ since $\frac{1}{B} \geq 4$ in our assumption. Therefore, since the laxity of the jobs of task τ_1 is no more than 1, the schedule of SULLF from time 0 to time $\frac{1}{B^2}(1-B)$ is the same as the schedule defined in Lemma 10. However, we are not able to construct a speedup factor lower bound of SALLF by using the task set $\mathbf{T}^{negative}$, since the resulting schedule is different from that in Lemma 10.

EDZL (earliest deadline zero laxity) is a preemptive hybrid-priority scheduling policy in which the jobs with zero laxity are given the highest priority and the other jobs are ranked by their absolute deadlines [21]. If we adopt suspension-unaware laxity for EDZL, the resulting schedule from time 0 to time $\frac{1}{B^2}(1-B)$ is the same as the schedule defined in Lemma 10 (since the suspension-unaware laxity is still positive).

For EDZL under suspension-aware laxity, the resulting schedule remains the same as EDF/SAEDF from time 0 to time $\frac{1}{B^2}(1-B)$ since the suspension-aware laxity of task τ_2 is greater than 0 (strictly) before $t = \frac{1}{B^2}(1-B) - 1 + B$.

4.4 Unbounded Speedup Factors

We conclude this section with the following theorem.

Theorem 5: The speedup factor of FP, EDF, suspension-aware EDF, EDZL, suspension-aware EDZL, and SULLF scheduling algorithms, compared to the optimal scheduling policy, for dynamic self-suspending task systems is not bounded by a constant or by the number of tasks.

Proof: This comes from Theorem 4 and the discussions in Section 4.3. ■

5 Utilization Bounds and Speedup Factors

In this section, we provide the utilization bounds and speedup factors for RM scheduling. Due to space limitations, we only consider implicit-deadline task systems here. Similar arguments can be made for constrained-deadline task systems under DM scheduling. For brevity, we sort the tasks by their priority levels. Task τ_1 has the highest priority, whereas task τ_n has the lowest priority. For the rest of this section, suppose that $\tau_1, \tau_2, \ldots, \tau_{k-1}$ have been already verified to meet their deadlines. We are interested to analyze whether task τ_k can meet its deadline or not.

In [27, p. 164-165], Liu proposed a solution to study the schedulability of a self-suspending task τ_k by modeling the extra delay suffered by τ_k due to the self-suspension behavior of each higher-priority task as a blocking time. This blocking time has been defined as follows:

- The blocking time contributed from task τ_k is S_k.
- A higher-priority task τ_i can block the execution of task τ_k for at most $\min(C_i, S_i)$ time units.

The correctness of the above analysis has been recently proved by Chen et al. [8] as a special case in a unified response-time analysis framework.[4] Suppose that $\gamma_k = \max_{i=1}^{k-1}\left\{\max\{1, \frac{S_i}{C_i}\}\right\}$. For testing whether task τ_k is schedulable by RM scheduling, as proved by Chen et al. [8], we can validate whether

$$\exists t \mid 0 < t \leq T_k, \quad C_k + S_k + \sum_{i=1}^{k-1}\left(\left\lceil \frac{t}{T_i} \right\rceil + \gamma_k\right) C_i \leq t. \quad (4)$$

Since $\gamma_k \leq 1$, this can be considered as if the first job from the higher-priority task τ_i under the schedulability analysis doubles its execution time. Liu and Chen [25] showed that the above schedulability test leads to the utilization-based hyperbolic test $\left(\frac{C_k+S_k}{T_k} + 2\right)\prod_{i=1}^{k-1}(1+U_i) \leq 3$. A more precise analysis in the following lemma comes from the utilization-based schedulability test framework **k2U** [5].

Lemma 11: An implicit-deadline system is schedulable by using RM scheduling if for each task τ_k in $\mathbf{T}$,

$$\left(\frac{C_k + S_k}{T_k} + 1 + \gamma_k\right)\prod_{i=1}^{k-1}(1+U_i) \leq 2 + \gamma_k, \quad (5)$$

[4] The reader may wonder why this is called blocking time. The terminology by modeling self-suspension time as blocking time was introduced by Liu in [27, p. 164-165]. Although the correctness of the schedulability test has been recently provided in [8], there is no clear evidence or logical explanation why we can call this as *blocking time*.

where $\gamma_k = \max_{i=1}^{k-1} \left\{ \max\{1, \frac{S_i}{C_i}\} \right\}$.

Proof: This comes directly by using the utilization-based schedulability test framework **k2U** [5]. The sketched proof is in Appendix A. ■

Recall that the example $\mathbf{T}^{negativ}$ used in Section 4 shows that the utilization bound of any FP scheduling is close to 0 if there is a task with very long self-suspension time. Therefore, there is no constant utilization bound or constant speedup factor for any FP scheduling algorithm neither if we do not constrain the self-suspension time. For the rest of this section, we demonstrate some positive results when the self-suspension time is constrained. The following theorems provide the utilization bounds and the speedup factors under different conditions by using the hyperbolic bound in Eq. (5):

- Theorem 6 shows that the speedup factor (respectively, the utilization bound) is $1/\ln(\frac{2+\lambda}{1+\lambda})$ (respectively, $\ln(\frac{2+\lambda}{1+\lambda})$) *if* $S_i \leq \lambda C_i$ *for each task* τ_i *in* $\mathbf{T}$.
- Theorem 7 shows that the speedup factor (respectively, the utilization bound) is $1/\ln(\frac{3}{2+\sigma})$ (respectively, $\ln(\frac{3}{2+\sigma})$) *if* $S_i + C_i \leq \sigma T_i$ *for each task* τ_i *in* $\mathbf{T}$.

Therefore, if the suspension time of each task is upper bounded by its worst-case execution time, Theorem 6 shows that the utilization bound can still be $\ln(3/2) \approx 0.4054$. Moreover, if $(C_i + S_i)/T_i$ is small enough for each τ_i in $\mathbf{T}$, i.e., σ is small enough, the utilization bound of RM is still $\ln(\frac{3}{2+\sigma})$.

Theorem 6: If $S_i \leq \lambda C_i$ for each task $\tau_i \in \mathbf{T}$, then task τ_k is schedulable by RM scheduling if $\sum_{i=1}^{k} U_i \leq \ln(\frac{2+\lambda}{1+\lambda})$. Moreover, the speedup factor of RM scheduling for such a case is $\frac{1}{\ln(\frac{2+\lambda}{1+\lambda})}$.

Proof: By the definition, we know $S_k/C_k \leq \lambda$ and $\gamma_k \leq \lambda$. Therefore, we can reformulate the schedulability test in Lemma 11 into $\left(\frac{C_k+S_k}{T_k} + 1 + \lambda\right) \leq (U_k + 1)(1 + \lambda)$. If task τ_k is not schedulable by using the schedulability test in Lemma 11, we know that $(U_k + 1)(1 + \lambda)\prod_{i=1}^{k-1}(U_i + 1) > (2+\lambda)$, which implies that $\prod_{i=1}^{k}(U_i+1) > (2+\lambda)/(1+\lambda)$. The infimum $\sum_{i=1}^{k} U_i$ such that $\prod_{i=1}^{k}(U_i + 1) > (2 + \lambda)/(1 + \lambda)$ happens when $U_i = \left(\frac{(2+\lambda)}{(1+\lambda)}\right)^{\frac{1}{k}} - 1$. Therefore, task τ_k is schedulable under RM if the utilization bound $\sum_{i=1}^{k} U_i \leq k\left(\left(\frac{(2+\lambda)}{(1+\lambda)}\right)^{\frac{1}{k}} - 1\right)$ holds, which converges to $\ln(\frac{2+\lambda}{1+\lambda})$ when k is sufficiently large. Therefore, we reach the conclusion for the utilization bound. The speedup factor is simply the reciprocal of the utilization bound. ■

Theorem 7: Suppose that $\sigma = \max_{\tau_i \in \mathbf{T}} \frac{C_i+S_i}{T_i}$. The speedup factor of RM scheduling for such a case is $\frac{1}{\ln(\frac{3}{2+\sigma})}$.

Proof: The proof is similar to Theorem 6 and left to Appendix A. ■

If speeding up the platform can also reduce the self-suspension time, we show that the suspension-coherent speedup factor of RM scheduling for implicit-deadline dynamic self-suspending task systems is a constant.

Theorem 8: The suspension-coherent speedup factor of RM scheduling for implicit-deadline dynamic self-suspending task systems is 3.621432.

Proof: The proof is similar to Theorem 7. We use the inequality in Eq. (11) by taking the worst-case setting of γ_k as 1. If we are also able to reduce the suspension time by speeding up, the execution time plus the suspension time becomes $\frac{C_k+S_k}{s}$ by running the platform at speed s. Therefore, we are interested to know the minimum $\frac{C_k+S_k}{T_k}$ and $\sum_{i=1}^{k-1} U_i \leq \sum_{i=1}^{k} U_i$ for the condition in Eq. (11) when $\gamma_k = 1$. This is equivalent to solving the equality $(x + 2)e^x = 3$. The equality holds when $x = x^* = lambertw(3 \cdot e^2) - 2$, where $lambertw(z)$ is the Lambert W function, i.e., $z = lambertw(z)e^{lambertw(z)}$. Therefore, either $\frac{C_k+S_k}{T_k} > x^*$ or $\sum_{i=1}^{k-1} U_i > x^*$. The suspension-coherent speedup factor is hence $1/x^* \approx 3.621432$. ■

6 Conclusion and Discussions

The $co\mathcal{NP}$-hardness of the (exact) schedulability test problem under FP scheduling shows that adding a segmented self-suspending sporadic task makes the schedulability analysis much more complicated than the ordinary sporadic task systems without self-suspension. There have been a significant amount of efforts analyzing FP scheduling for dynamic self-suspending sporadic task systems, e.g. [1]–[3], [16], [18], [25], [28]. However, the gap between the optimal fixed-priority schedule and the optimal schedule (with respect to the speedup factor) can be unbounded, as shown in this paper. We also show how to extend the classical dynamic-priority scheduling algorithms, i.e., EDF, LLF, and EDZL, to be suspension-aware. Unfortunately, all these algorithms, except suspension-aware LLF, have also unbounded speedup factors.

Interestingly, the difficulty of the scheduler design problem in dynamic self-suspending task systems may simply come from the definition of the quantification metrics: *the speedup factor*. If the suspension time of each task is upper bounded by its worst-case execution time, then, we show that RM scheduling has a constant speedup factor and a constant utilization bound. If $\sigma = \max_{\tau_i \in \mathbf{T}} \frac{C_i+S_i}{T_i}$ is small enough, the speedup factor of RM scheduling for such a case is $1/\ln(\frac{3}{2+\sigma})$, which is far below the worst-case scenario demonstrated in Section 4. Moreover, if the suspension time can also be reduced by speeding up coherently, then we show that RM scheduling has a constant suspension-coherent speedup factor.

However, the negative example used in Section 4 does not invalidate the effectiveness (with respect to the speedup factor) of SALLF (suspension-aware least laxity-first) or FPZL (fixed-priority zero-laxity) scheduling algorithms. These two scheduling algorithms may be useful, but their corresponding schedulability analyses are also unknown when there are dynamic self-suspending tasks.

References

[1] N. C. Audsley and K. Bletsas. Fixed priority timing analysis of real-time systems with limited parallelism. In *16th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 231–238, 2004.

[2] N. C. Audsley and K. Bletsas. Realistic analysis of limited parallel software / hardware implementations. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 388–395, 2004.

[3] K. Bletsas and N. C. Audsley. Extended analysis with reduced pessimism for systems with limited parallelism. In *11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 525–531, 2005.

[4] J.-J. Chen. A note on the exact schedulability analysis for segmented self-suspending systems. *Computing Research Repository (CoRR)*, 2016. http://arxiv.org/abs/1605.00124.

[5] J.-J. Chen, W.-H. Huang, and C. Liu. k2U: A general framework from k-point effective schedulability analysis to utilization-based tests. In *Real-Time Systems Symposium (RTSS)*, 2015.
[6] J.-J. Chen, W.-H. Huang, and C. Liu. Automatic parameter derivations in *k2U* framework. *Computing Research Repository (CoRR)*, 2016. http://arxiv.org/abs/1605.00119.
[7] J.-J. Chen and C. Liu. Fixed-relative-deadline scheduling of hard real-time tasks with self-suspensions. In *Real-Time Systems Symposium (RTSS)*, pages 149–160, 2014. **A typo in the schedulability test in Theorem 3 was identified on 13, May, 2015**.
[8] J.-J. Chen, G. Nelissen, and W.-H. Huang. A unifying response time analysis framework for dynamic self-suspending tasks. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2016.
[9] J.-J. Chen, G. Nelissen, W.-H. Huang, M. Yang, B. Brandenburg, K. Bletsas, C. Liu, P. Richard, F. Ridouard, Neil, Audsley, R. Rajkumar, and D. de Niz. Many suspensions, many problems: A review of self-suspending tasks in real-time systems. Technical Report 854, Faculty of Informatik, TU Dortmund, 2016.
[10] R. I. Davis, A. Burns, S. Baruah, T. Rothvoß, L. George, and O. Gettings. Exact comparison of fixed priority and EDF scheduling based on speedup factors for both pre-emptive and non-pre-emptive paradigms. *Real-Time Systems*, 51(5):566–601, 2015.
[11] S. Ding, H. Tomiyama, and H. Takada. Effective scheduling algorithms for I/O blocking with a multi-frame task model. *IEICE Transactions*, 92-D(7):1412–1420, 2009.
[12] P. Ekberg and W. Yi. Uniprocessor feasibility of sporadic tasks with constrained deadlines is strongly coNP-Complete. In *27th Euromicro Conference on Real-Time Systems, ECRTS*, pages 281–286, 2015.
[13] M. R. Garey and D. S. Johnson. *Computers and intractability: A guide to the theory of NP-completeness*. W. H. Freeman and Co., 1979.
[14] W.-H. Huang and J.-J. Chen. Schedulability and priority assignment for multi-segment self-suspending real-time tasks under fixed-priority scheduling. Technical report, Technical University of Dortmund, Dortmund, Germany, 2015.
[15] W.-H. Huang and J.-J. Chen. Self-suspension real-time tasks under fixed-relative-deadline fixed-priority scheduling. In *Design, Automation, and Test in Europe (DATE)*, 2016.
[16] W.-H. Huang, J.-J. Chen, H. Zhou, and C. Liu. PASS: Priority assignment of real-time tasks with dynamic suspending behavior under fixed-priority scheduling. In *DAC*, 2015.
[17] B. Kalyanasundaram and K. Pruhs. Speed is as powerful as clairvoyance. *Journal of ACM*, 47(4):617–643, July 2000.
[18] I. Kim, K. Choi, S. Park, D. Kim, and M. Hong. Real-time scheduling of tasks that contain the external blocking intervals. In *RTCSA*, pages 54–59, 1995.
[19] J. Kim, B. Andersson, D. de Niz, and R. Rajkumar. Segment-fixed priority scheduling for self-suspending real-time tasks. In *Real-Time Systems Symposium, (RTSS)*, pages 246–257, 2013.
[20] K. Lakshmanan and R. Rajkumar. Scheduling self-suspending real-time tasks with rate-monotonic priorities. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 3–12, 2010.
[21] S. K. Lee. On-line multiprocessor scheduling algorithms for real-time tasks. In *IEEE Region 10's Ninth Annual International Conference. Theme: Frontiers of Computer Technology*, pages 607–611 vol.2, 1994.
[22] J. P. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *RTSS*, pages 201–209, 1990.
[23] J. Y.-T. Leung. A new algorithm for scheduling periodic real-time tasks. *Algorithmica*, 4(2):209–219, 1989.
[24] J. Y.-T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Perform. Eval.*, 2(4):237–250, 1982.
[25] C. Liu and J.-J. Chen. Bursty-Interference Analysis Techniques for Analyzing Complex Real-Time Task Models. In *2014 IEEE Real-Time Systems Symposium*, pages 173–183, 2014.
[26] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20(1):46–61, 1973.
[27] J. W. S. Liu. *Real-Time Systems*. Prentice Hall PTR, 1st edition, 2000.
[28] L. Ming. Scheduling of the inter-dependent messages in real-time communication. In *Proc. of the First International Workshop on Real-Time Computing Systems and Applications*, 1994.
[29] G. Nelissen, J. Fonseca, G. Raravi, and V. Nélis. Timing Analysis of Fixed Priority Self-Suspending Sporadic Tasks. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 80–89, 2015.
[30] J. C. Palencia and M. G. Harbour. Schedulability analysis for tasks with static and dynamic offsets. In *Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS)*, pages 26–37, 1998.
[31] F. Ridouard, P. Richard, and F. Cottet. Negative results for scheduling independent hard real-time tasks with self-suspensions. In *RTSS*, pages 47–56, 2004.

Appendix

Appendix A: Proofs

Proof of Lemma 2. Assume that the condition in Eq. (2) is not satisfied. That is, there exists a certain $t^* > 0$ such that $\sum_{\tau_i \in \mathbf{T}} \max\left\{0, \left(\left\lfloor \frac{t^* - (D_i - S_i)}{T_i} \right\rfloor + 1\right) \cdot C_i\right\} > t^*$. Then, we prove that there exists a valid release pattern that cannot be schedulable by using any scheduling algorithm. Let a be a positive number. We release the first job of each task τ_i at time $a - S_i$, and each task suspends from $a - S_i$ to a. After the first job of task τ_i, the subsequent jobs are released periodically at time $a - S_i + T_i, a - S_i + 2T_i, \ldots$ with the same suspension and execution pattern.

For this specific release pattern, the task set can be actually imagined as if each self-suspending sporadic task τ_i becomes an ordinary sporadic task without self-suspension. The equivalent sporadic task of task τ_i has an effective relative deadline $D_i - S_i$, execution time C_i, and period T_i. The existence of t^* implies that the demand $\sum_{\tau_i \in \mathbf{T}} \max\left\{0, \left(\left\lfloor \frac{t^* - (D_i - S_i)}{T_i} \right\rfloor + 1\right) \cdot C_i\right\}$ of task execution from a to $a + t^*$ (usually called the demand bound function in the literature) is strictly greater than the interval length t^*. Therefore, it is not possible to feasibly schedule this release pattern of task set $\mathbf{T}$. □

Proof of Lemma 4.

There are a few properties for the necessary condition of the worst-case release pattern:

1) We only have to check one job execution of task τ_n.
2) All the higher-priority tasks only release their jobs between the arrival time and the finishing time of each computation segment of task τ_n.
3) All the self-suspension intervals of task τ_n always take the worst case. All the jobs and all the computation segments are executed with their worst-case execution time specifications.

The proof of the above properties can be found in [4]. These three properties were also implicitly used in [29]. We can now use the above properties to prove Lemma 4. For task τ_1 in $\mathbf{T}^{red}$, since $T_1 < S_n^j$ for any $j = 1, 2, \ldots, M-1$, the release pattern of task τ_1 is independent from the computation segments. This is formally proved in Property 2 in [4]. Moreover, since $T_i > D_n$ for $i = 2, 3, \ldots, 3M+1$, such a higher-priority task τ_i in $\mathbf{T}^{red}$ only releases one job together with one of the M computation segments of the job (under analysis) of task τ_n. This is formally proved in Property 3 in [4]. By putting all the above conditions together, we reach the conclusion for Lemma 4. □

Proof of Lemma 11. This comes directly from Lemma 13, that can be proved by using the **k2U** framework in [5]. The automatic parameter derivation procedure for the **k2U** framework has been recently developed by Chen et al. [6], i.e., Corollary 1 in [6], as a powerful tool for the **k2U** framework. The details can be found in [5], [6]. Here, we sketch the proof for completeness in Lemma 13. We can directly link the schedulability test in Eq. (4) to Eq. (8), by setting C'_k as $C_k + S_k$ and γ as γ_k. This results in the test in Eq. (5). □

Before proving Lemma 13, we give one definition and one important lemma from the **k2U** framework [5].[5]

[5]Here, we use C'_k instead of C_k.

w_1	$R(w_1)$	w_2	$R(w_2)$	w_3	$R(w_3)$	w_4	$R(w_4)$	w_5	$R(w_5)$	w_6	$R(w_6)$	$\mathbf{X}$	$\mathbf{Y}$	$\sum_{j=1}^{6} R(w_j)$
0	$2V+1$	$3.5V$	$7.5V+1$	$0.4V$	$2.4V+1$	$0.6V$	$2.6V+1$	$1.5V$	$4.5V+1$	0	$2V+1$	$\{1,3,4,6\}$	$\{2,5\}$	$21V+6$
V	$4V+1$	$2.5V$	$5.5V+1$	–	–	–	–	–	–	–	–	$\{3,4,6\}$	$\{2,5\}$	$21V+6$
–	–	$1.5V$	$4.5V+1$	–	–	–	–	–	–	V	$4V+1$	$\{3,4\}$	$\{2,5\}$	$22V+6$
–	–	V	$4V+1$	V	$4V+1$	–	–	$1.4V$	$4.4V+1$	–	–	$\{4\}$	$\{5\}$	$23V+6$
–	–	–	–	–	–	V	$4V+1$	$1V$	$4V+1$	–	–	$\emptyset$	$\emptyset$	$24V+6$

TABLE II: An example of Lemma 7

Definition 3 (Chen, Huang, and Liu [5]): A k-point effective schedulability test is a sufficient schedulability test of a fixed-priority scheduling policy, that verifies the existence of $t_j \in \{t_1, t_2, \dots t_k\}$ with $0 < t_1 \le t_2 \le \cdots \le t_k$ such that

$$C'_k + \sum_{i=1}^{k-1} \alpha_i t_i U_i + \sum_{i=1}^{j-1} \beta_i t_i U_i \le t_j, \tag{6}$$

where $C'_k > 0$, $\alpha_i > 0$, $U_i > 0$, and $\beta_i > 0$ are dependent upon the setting of the task models and task τ_i. □

Lemma 12 (Chen, Huang, and Liu [5]): For a given k-point effective schedulability test of a scheduling algorithm, defined in Definition 3, in which $0 < t_k$ and $0 < \alpha_i \le \alpha$, and $0 < \beta_i \le \beta$ for any $i = 1, 2, \dots, k-1$, task τ_k is schedulable by the scheduling algorithm if the following condition holds

$$\frac{C'_k}{t_k} \le \frac{\frac{\alpha}{\beta} + 1}{\prod_{j=1}^{k-1}(\beta U_j + 1)} - \frac{\alpha}{\beta}. \tag{7}$$

Lemma 13: Suppose that $T_i \le T_k$, for each task $\tau_i \in hp(\tau_k)$, where $hp(\tau_k)$ is the set of the tasks with higher-priority than task τ_k in the fixed-priority scheduling algorithm, i.e., rate-monotonic (RM). For a schedulability test

$$\exists 0 < t \le T_k \text{ s.t. } C'_k + \sum_{\tau_i \in hp(\tau_k)} \left(\left\lceil \frac{t}{T_i} \right\rceil C_i + \gamma C_i \right) \le t, \tag{8}$$

where $\gamma \ge 0$, task τ_k is schedulable if

$$\left(\frac{C'_k}{T_k} + (1+\gamma) \right) \prod_{\tau_i \in hp(\tau_k)} (U_i + 1) \le 2 + \gamma. \tag{9}$$

Proof: The proof is a special case of Corollary 1 in [6]. Let g_i be $\left\lfloor \frac{T_k}{T_i} \right\rfloor$ and t_i be $g_i T_i$. By $T_i \le T_k$ for $i = 1, 2, \dots, k$, we know that $g_i \ge 1$. We index the tasks in $hp(\tau_k) \cup \{\tau_k\}$ according to non-decreasing t_i. Therefore, the left-hand side of Eq. (8) at time $t = t_j$ is upper bounded by

$$\begin{aligned}
& C'_k + \sum_{i=1}^{k-1} \left(\left\lceil \frac{t_j}{T_i} \right\rceil C_i + \gamma C_i \right) \\
\le_0 \; & C'_k + \sum_{i=1}^{j-1} \left(\left\lceil \frac{T_k}{T_i} \right\rceil C_i + \gamma C_i \right) + \sum_{i=j}^{k-1} \left(\left\lceil \frac{t_i}{T_i} \right\rceil C_i + \gamma C_i \right) \\
\le_1 \; & C'_k + \sum_{i=1}^{j-1} ((g_i + 1) C_i + \gamma C_i) + \sum_{i=j}^{k-1} (g_i C_i + \gamma C_i) \\
= \; & C'_k + \sum_{i=1}^{k-1} (g_i C_i + \gamma C_i) + \sum_{i=1}^{j-1} C_i \\
=_2 \; & C'_k + \sum_{i=1}^{k-1} \frac{(g_i + \gamma)}{g_i} t_i U_i + \sum_{i=1}^{j-1} \frac{1}{g_i} t_i U_i,
\end{aligned} \tag{10}$$

where the inequality $\le_0$ comes from $t_1 \le t_2 \le \cdots \le t_j \le t_{j+1} \le \cdots \le t_k = T_k$ in our index rule, the inequality $\le_1$ comes from $\left\lceil \frac{T_k}{T_i} \right\rceil \le \left\lfloor \frac{T_k}{T_i} \right\rfloor + 1 = g_i + 1$, and $=_2$ comes from the setting that $C_i = U_i T_i = \frac{1}{g_i} t_i U_i$. That is, the test in Eq. (8) can be safely rewritten as

$$(\exists t_j | j = 1, 2, \dots, k), \qquad C'_k + \sum_{i=1}^{k-1} \alpha_i t_i U_i + \sum_{i=1}^{j-1} \beta_i t_i U_i \le t,$$

where $U_i = C_i / T_i$, $\alpha_i = \frac{g_i + \gamma}{g_i} \le 1 + \gamma$ and $\beta_i = \frac{1}{g_i} \le 1$ for $i = 1, 2, \dots, k-1$. Therefore, we use $\alpha = 1 + \gamma$ and $\beta = 1$ and apply Lemma 12 to reach the conclusion of this lemma. ■

Proof of of Theorem 7. Suppose that task τ_k is not schedulable by RM scheduling due to the failure to pass the schedulability test in Lemma 11. Without loss of generality, we assume $C_k + S_k \le T_k$ here for proving the speedup factor. By the fact that the arithmetical mean is greater than or equal to the geometric mean, we know that $(\prod_{i=1}^{k-1}(1 + U_i))^{\frac{1}{k-1}} \le (1 + \frac{\sum_{i=1}^{k-1} U_i}{k-1})$, which implies that $\prod_{i=1}^{k-1}(1 + U_i) \le (1 + \frac{\sum_{i=1}^{k-1} U_i}{k-1})^{k-1} \le e^{\sum_{i=1}^{k-1} U_i}$. Therefore,

$$\begin{aligned}
& \left(\frac{C_k + S_k}{T_k} + 1 + \gamma_k \right) \prod_{i=1}^{k-1} (1 + U_i) > 2 + \gamma_k \\
\Rightarrow & \left(\frac{C_k + S_k}{T_k} + 1 + \gamma_k \right) e^{\sum_{i=1}^{k-1} U_i} > 2 + \gamma_k. \qquad (11) \\
\Rightarrow & \sum_{i=1}^{k-1} U_i > \ln \left(\frac{2 + \gamma_k}{\frac{C_k + S_k}{T_k} + 1 + \gamma_k} \right) \ge_1 \ln \left(\frac{3}{\frac{C_k + S_k}{T_k} + 2} \right).
\end{aligned}$$

The inequality $\ge_1$ is due to the fact that $0 \le \gamma_k \le 1$ and the implicit assumption $C_k + S_k \le T_k$. Therefore, by contrapositive, we reach the conclusion of the utilization bound $\ln \left(\frac{3}{\frac{C_k + S_k}{T_k} + 2} \right)$ for RM scheduling. The speedup factor is simply the reciprocal of the utilization bound. □

Appendix B: An example of Lemma 7

We use an example to illustrate how the procedure in Lemma 7 operates. Suppose that $w_1 = 0, w_2 = 3.5V, w_3 = 0.4V, w_4 = 0.6V, w_5 = 1.5V, w_6 = 0$ when $M = 6$ and V is an integer multiple of 10. We will start from $\mathbf{X} = \{1, 3, 4, 6\}$ and $\mathbf{Y} = \{2, 5\}$. As shown in Table II, the operation makes $\sum_{j=1}^{6} R(w_j)$ increase. Note that the conclusion $\sum_{j=1}^{M} R(w_j) \le M(4V + 1) - |\mathbf{X}|V$ in Lemma 7 was for $|\mathbf{X}| = \{3, 4\}$ in this example when $w_j < 2V$ for any $j \in \mathbf{Y}$.