Schedulability and Priority Assignment for Multi-Segment Self-Suspending Real-Time Tasks under Fixed-Priority Scheduling

Wen-Hung Huang and Jian-Jia Chen
Department of Computer Science
TU Dortmund University, Germany
{wen-hung.huang, jia.chen}@tu-dortmund.de

Abstract—**Self-suspension is becoming an increasingly prominent characteristic in real-time systems such as: (i) I/O-intensive systems (ii) multi-core processors, and (iii) computation offloading systems with coprocessors, like Graphics Processing Units (GPUs). In this paper, we study the schedulability of multi-segment self-suspension tasks under fixed-priority scheduling, where the executions of a multi-segment self-suspension task alternate between per-defined computation segments and suspension intervals. In particular, we do not use any enforcement to control the releases of computation segments and suspension intervals. Such an enforcement can prevent jitter but may incur non-negligible overheads. This work presents a combined method using the proposed multi-segment workload function to compute the upper bound on the worst-case response time (WCRT) of multi-segment tasks. To the best of our knowledge, this is the first study that successfully provides a pseudo-polynomial-time test for multi-segment self-suspension, hard real-time systems under fixed-priority scheduling without any additional execution control. We also show that the proposed analysis is compatible with Audsley's Priority Assignment. Our empirical investigations show that the proposed approach is highly effective in terms of the number of task sets deemed to be schedulable.**

I. Introduction

In many real-time and embedded systems, tasks may be suspended by the operating system when accessing external devices such as disks, graphical processing units (GPUs), or synchronizing with other tasks. This behavior is often known as *self-suspension*. Self-suspensions are even more pervasive in many emerging embedded cyber-physical systems in which the computation components frequently interact with external and physical devices [16], [17]. Typically, the resulting suspension delays range from a few microseconds (e.g., a write operation on a flash drive [16]) to a few hundreds of milliseconds (e.g., offloading computation to GPUs [17], [25]). Such suspension delays have negative impact on the timing predictability and cause intractability in hard real-time (HRT) schedulability analysis [26].

The unsolved problem of efficiently supporting self-suspensions in real-time systems has impeded research progress on many related research topics such as predictably supporting I/O-intensive applications and computation offloading. In fact, the problem of scheduling HRT self-suspension task systems on a uniprocessor has been proved to be $\mathcal{NP}$-hard in the strong sense [26]. Therefore, there cannot exist a polynomial-time scheduling algorithm for this problem, unless $\mathcal{P} = \mathcal{N}P$ [26].

Related work. Recently, there have been some results [14], [20]–[22] on the *dynamic* self-suspending task model [24]. This model characterizes each task as a 4-tuple (C_i, S_i, T_i, D_i): T_i denotes the minimum inter-arrival time of τ_i, each job of τ_i has a relative deadline D_i, C_i denotes the upper bound on total execution time of each job of τ_i, and S_i denotes the upper bound on total suspension time of each job of τ_i. For such a model, a utilization bound for self-suspending task is derived under *rate-monotonic* (RM) scheduling in [22]. In [14] the proposed approach is guaranteed to find a feasible fixed-priority assignment on a speed-2 uniprocessor, if one exists on a unit-speed processor.

From the system designer's perspective, such a model provides an easy way to specify self-suspending systems without considering the juncture of I/O access or computation offloading. However, such a model essentially suffers from poor schedulability, as explained in Appendix B.

In terms of schedulability, it is more desirable to adopt the *multi-segment* self-suspension task model [26] where each task's computation segments and suspension intervals are specified as an array $(C_i^0, S_i^0, C_i^1, S_i^1, ..., S_i^{M_i-2}, C_i^{M_i-1})$ composed of M_i computation segments separated by $M_i - 1$ suspension intervals. For dynamic-priority scheduling, Chen and Liu [8] show that fixed-relative-deadline scheduling can yield non-trivial resource augmentation (speed-up factor) performance. Fixed-relative-deadline scheduling assigns each computation segment of a task with a fixed relative deadline, and adopts *earliest-deadline-first* (EDF) scheduling by giving the highest priority to the job with the earliest *absolute* deadline. However, the result in [8] is only applicable when each task has at most one suspension interval.

Several approaches [6], [11], [18], [19] have been reported to analyze and design fixed-priority scheduling for the multi-segment self-suspension task model. Specifically, Bletsas and Audsley [6] study the schedulability for systems with limited parallelism, which can also be modeled by the multi-segment self-suspension model. However, the analysis provided by Bletsas and Audsley [6] is flawed. We present a counter example in Appendix A. Lakshmanan and Rajkumar [19] study a special case of such a problem, in which the system has only one self-suspension task as the lowest-priority task. Unfortunately, the critical-instant analysis in [19] has also been shown flawed in a very recent paper by Nelissen et al. [11]. Moreover, Nelissen et al. [11] propose a method based on a mixed integer linear programming (MILP) formulation to calculate the *exact* worst-case response time of a self-suspension task with *only one* suspension interval. However, in the general cases, the time complexity rapidly grows, and the MILP-based approach requires exponential time complexity. Furthermore, how to assign the task priority remains open for such a model.

Kim et al. [18] also analyze special cases when there is only one self-suspension task. Moreover, an MILP-based algorithm for segment-fixed priority scheduling is developed in [18], in which each computation segment of a task is assigned with a specific priority level, so to speak segment-level scheduling. In their analysis, all the higher-priority computation segments are assumed to be able to interfere with a lower-priority computation segment at the same time. However, it is an evidence that two computation segments within one job of a higher-priority self-suspension task are temporarily separated by a suspension interval, and hence their formulation of the MILP is pessimistic. Moreover, in [18], [19] the static/dynamic slack enforcement is proposed to ensure that each task imposes no more than its sporadic non-suspending interference on the lower-priority task. Such a scheme requires an additional execution control. The slack enforcement is introduced in [18], [19] to make the analysis easier, but its advantages to the system are not discussed.

In summary, several promising results have been successfully proposed for the dynamic self-suspension task model [14], [20]–[22]. However, in the presence of multi-segment self-suspension task models, such results are too pessimistic to be seamlessly adopted. The best-known result presented for multi-segment self-suspension task models under fixed-priority scheduling has worst-case exponential time complexity [11]. There is neither polynomial-time nor pseudo-polynomial-time algorithm available to provide efficient schedulability analysis and feasible priority assignments for multi-segment self-suspension tasks.

The significance of this paper is that by means of a sophisticated schedulability analysis, we do not need any additional execution control on the releases of suspension intervals and computation segments, called *enforcement*, by which some additional overhead may incur, as used in the concurrent submission [12]. Besides, the approach in this paper achieves significantly high performance in terms of schedulability when there are many suspension intervals, whereas the result [12] essentially suffers from such a case. One may conclude that the result in this paper is more applicable if there are many suspension intervals, whereas the concurrent submission [12] is recommended if there are a few suspension intervals.

Contribution. We propose a pseudo-polynomial-time approach for assigning priority levels and analyzing the multi-segment self-suspension system under fixed-priority scheduling without any additional execution control. Our contributions are as follows:

- We derive a workload function to safely bound the workload from higher-priority tasks, to be detailed in Section III. The workload function considers the worst-case execution and release scenarios including early computation and suspension completions. As the workload function provides more precise information in the multi-segment self-suspension model than the dynamic self-suspension model, our response time analysis is superior to the analysis in [14], [22].
- The derived workload function can be utilized in two ways to analyze the worst-case response time of the task τ_k under analysis, to be detailed in Section IV: (1) It can be used to analyze the worst-case response time of a computation segment directly. (2) Alternatively, it can also be used by treating τ_k's suspension as computation. We prove that both of the above treatments can be adopted by extending the standard response time analysis (RTA). This results in pseudo-polynomial time complexity for deriving the worst-case response time safely. Such tests can also be converted to polynomial-time with some errors by using the tricks in [1], [7].
- We also show, in Section V, that the proposed response time analysis is compatible with Audsley's priority assignment [3], [4], [9]. Therefore, the optimal priority ordering (under our response time analysis) can be determined efficiently.
- Our empirical investigations, in Section VI, show that the proposed approach is highly effective in terms of the number of task sets deemed to be schedulable. We significantly improve the performance over the best-known test for dynamic self-suspension model presented in [14]. The self-suspension task is deemed to reach the poor schedulability when tasks have a long suspension interval. The proposed test fully tackles such a case, and our empirical results show that our proposed test is able to accept task sets with utilization up to 75% in such a case with noticeable acceptance ratios.
- Evaluation results also provide very strong evidence to control the suspension length to improve the schedulability, which is discussed in Section VII. However, controlling the phases of the computation segments (with constant offsets with respect to the first computation segment) is not necessary.

To the best of our knowledge, this is the first work that successfully provides schedulability analysis with pseudo-polynomial-time complexity for multi-segment self-suspension systems under fixed-priority scheduling without any additional execution control.

II. System Model and Notations

We consider a real-time system to execute a set of n independent, preemptive, self-suspension real-time tasks $\tau = \{\tau_1, \tau_2, ..., \tau_n\}$ on a uniprocessor. Each task can release an infinite number of jobs under minimum inter-arrival time (temporal) constraints. The *self-suspending sporadic* (SSS) task model extends the conventional sporadic task model by allowing tasks to suspend themselves. Similar to sporadic tasks, a self-suspending sporadic task releases jobs sporadically, but the execution of each job of τ_i is composed of M_i *computation* segments separated by $M_i - 1$ *suspension* intervals. A computation segment is eligible to execute only after the completion of the previous suspension interval. A self-suspension task τ_i is characterized by a 3-tuple:

$$\tau_i = \Big((C_i^0, S_i^0, C_i^1, S_i^1, ..., S_i^{M_i-2}, C_i^{M_i-1}), T_i, D_i\Big)$$

where T_i denotes the minimum inter-arrival time of τ_i, D_i denotes the relative deadline of task τ_i, C_i^j denotes the upper bound on execution times of the $(j+1)$-th computation segment, and S_i^j denotes the $(j+1)$-th suspension interval. If M_i is 1, there is only one computation segment of task τ_i, which is the conventional sporadic task model. It is possible that a task can start or complete its job with a suspension interval. This can be easily covered by the above model by placing one virtual computation segment at the beginning or at the end in the model. Therefore, such cases are not explicitly discussed.

Each S_i^j ranges in interval $[\check{S}_i^j, \hat{S}_i^j]$ where $\check{S}_i^j$ ($\hat{S}_i^j$, respectively) denotes the lower (upper, respectively) bound on the $(j+1)$-th suspension time of task τ_i. Note that it is not required to obtain the lower bound on suspension intervals (if unknown, then $\check{S}_i^j = 0$). However, as will be seen in Section VI, the interference from higher-priority tasks can be more precisely calculated if the lower bound is provided.

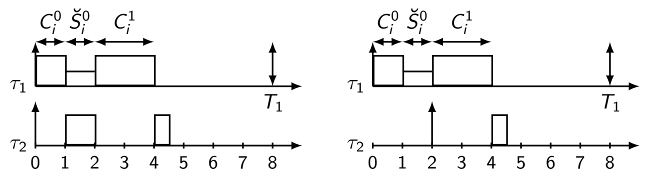
We denote the amount of total computation segment lengths $\sum_{j=0}^{M_i-1} C_i^j$ as C_i and the amount of total minimum suspension interval times $\sum_{j=0}^{M_i-2} \check{S}_i^j$ as $\check{S}_i$ and the amount of total maximum suspension interval times $\sum_{j=0}^{M_i-2} \hat{S}_i^j$ as $\hat{S}_i$. Note that when $M_i = 1$, both $\check{S}_i$ and $\hat{S}_i$ are 0. We assume that $C_i + \hat{S}_i \leq D_i$ for any task $\tau_i \in \tau$. The utilization of task τ_i is defined as $U_i = C_i/T_i$. We further assume that $\sum_{i=1}^{n} U_i \leq 1$.

A task system τ is said to be an *implicit-deadline* system if it is guaranteed that each task has its relative deadline equal to its period, and a *constrained-deadline* system if the relative deadline of each task is no larger than its period. Otherwise, task system τ is said to be an *arbitrary-deadline* system. In this work, we restrict our attention to *constrained-deadline* task systems.

The response time of a job is defined as the completion time of the last computation segment minus the release time of the job. The worst-case response time of task τ_i is defined as the longest response time among all the jobs released. A system τ is said to be *feasible* if there exists a scheduling algorithm that can schedule the system without any deadlines being missed.

In this paper we focus on fixed-priority scheduling, in which each task is associated with a unique priority. More precisely, all the jobs of a task have the same priority level, and the system always selects the job in the ready queue with the highest-priority level to execute. Clearly, if a job suspends itself, it is no longer in the ready queue. On the other hand, when a job resumes from its self-suspension, it is put into the ready queue again.

The *multi-segment* self-suspension model is a restrictive model extended from the *dynamic* self-suspension model where the suspension can occur at any time-instant with any suspension size. Hence the dynamic self-suspension model can be thought of as a relaxation of the multi-segment self-suspension model. The dynamic model in turn alleviates the effort of finding the location of suspension intervals. However, for such a model, oblivious to the releasing pattern of computation segments, the schedulability is deemed more pessimistic. In several approaches [14], [20]–[22] reported on this model we can see that the suspension intervals of the analyzed task are modeled as computation segments. Comparing to the multi-segment self-suspension model, such a transformation leads to rather poor performance. In detail, we explain this in Appendix B.



(a) The critical instant for 1.5 execution time units

(b) The critical instant for 0.5 execution time units

Fig. 1: The dependency of the critical instant on the parameters of task τ_2

III. Multi-Segment Workload Function

Before analyzing the worst-case response time of a task, we have to quantify the interference from the higher-priority tasks. Most of the existing workload functions that quantify such interferences [14], [22] are derived based on the dynamic self-suspension model, which lacks of the characteristic of multi-segment self-suspension tasks studied in this paper. The workload functions from [19] and [6] have been shown to be flawed (in Appendix A for [6] and in [11] for [19]). This section presents a workload function that provides some insights on the multi-segment self-suspension task and results in tighter analysis.

Most of the response time analysis relies on the *critical instant* for a task, which is defined to be an instant at which an execution of that task will have the longest response time [23]. For sporadic tasks without self-suspension, it is proven that the critical instant for a task occurs when the analyzed task and all higher priority tasks are released simultaneously and all the jobs are released as early as possible [23]. This critical instant is unique and independent of the parameters of the higher-priority tasks and the analyzed task.

Unfortunately, for self-suspension tasks, there is a dependency on the parameters of tasks for the critical instant. For example, consider a self-suspension real-time system consisting of one SSS task $\tau_1 = ((1, 1, 2), 8, 8)$ and one sporadic task τ_2 with both period and deadline ≥ 8, as shown in Figure 1. The SSS task has higher-priority than the sporadic task. When the execution time of task τ_2 is 1.5, the critical instant for task τ_2 occurs when task τ_2 is released simultaneously with the first computation segment of task τ_1. On the other hand, when the execution time of task τ_2 is 0.5, the critical instant occurs when task τ_2 is released simultaneously with the second computation segment of task τ_1.

Instead of investigating the critical instant, we will safely bound the higher-priority interferences by quantifying the *carry-in* jobs. As a naive approach, we can simply count all the computation segments of the carry-in job, which is released prior to the interval of our interest. This is referred as a *burst* in [22]. However, this certainly induces too much pessimism for multi-segment self-suspension systems where the pattern of computation segments is given.

To quantify the interference from the higher-priority tasks more precisely, we here give some insights in the multi-segment self-suspension systems. For such systems, no matter how the computation segments of a multi-segment self-suspension task are executed in the schedule, there will be a minimum *inter-arrival* time between two computation segments within one job, i.e., the minimum time on suspension intervals $\breve{S}_i^j$ plus the computation segment C_i^j. In addition, the time between the finish time of the last computation segment and the release time of the first computation segment of the next job release cannot be less than $T_i - D_i$. Otherwise, task τ_i already misses its deadline. Informally speaking, we can shift the computation segments of the first release to the right so that consecutive releases can run nearly back-to-back while satisfying the minimum inter-arrival time requirement. (Also see Figure 2.)

Suppose that given a time-instant t_0, the interval $[t_0, t_0+t]$ with a length of t is of our interest. The rest of this section is to quantify an upper bound of the execution of task τ_i in the time interval $[t_0, t_0+t)$ under the assumption that task τ_i can meet its deadline and $D_i \leq T_i$. By the assumption that $D_i \leq T_i$, there is at most one job of an SSS task that can be carried into the interval of our interest, i.e. the so-called *carry-in* job. The remaining question is to answer what is the worst case of interferences from an SSS task that characterizes such a carry-in job (may including several computation segments).

Multi-segment workload function. Multi-segment workload function is an upper bound on the amount of execution that the jobs of task τ_i can perform in the time interval of duration t where h states the phasing (also see Figure 2). More formally, let h be a *candidate* computation segment in the carry-in job of task τ_i.

Definition 1: Given an index h and an interval length t, the multi-segment workload function $W_i(t)$ of task τ_i is defined as follows:

$$W_i^h(t) = \sum_{j=h}^{\ell} C_i^{j \ mod \ M_i} + min\left(C_i^{(\ell+1) \ mod \ M_i}, t - \sum_{j=h}^{\ell}\left(C_i^{j \ mod \ M_i} + S_i(j)\right)\right) \tag{1}$$

where ℓ is the *maximum* integer satisfying the following condition:

$$\sum_{j=h}^{\ell}\left(C_i^{j \ mod \ M_i} + S_i(j)\right) \leq t \tag{2}$$

and

$$S_i(j) \equiv \begin{cases} \breve{S}_i^{j \ mod \ M_i} & \text{if } (j \ mod \ M_i) \neq (M_i - 1), \\ T_i - D_i & \text{else if } \ j \leq M_i, \\ T_i - (C_i + \breve{S}_i) & \text{otherwise.} \end{cases}$$

□

Roughly speaking, the work from task τ_i can be considered as two parts: (*i*) the last computation segment that arrives before $t_0 + t$ and (*ii*) the other computation segments released before the last computation segment. The segments of task τ_i's first release to be counted as the carry-in job are released as late as possible. Every computation segment is treated as it is released immediately, regardless of the schedule. The modulo operation *mod* is used to indicate which computation segment to be counted. Hence, the term $\sum_{j=h}^{\ell} C_i^{j \ mod \ M_i}$ counts those computation segments that can be fully executed prior to the last computation segment's release within time interval $[t_0, t_0+t)$, starting from the h-th computation segment. For the last computation segment released prior to $t_0 + t$, we use $min\left(C_i^{(\ell+1) \ mod \ M_i}, t - \sum_{j=h}^{\ell}\left(C_i^{j \ mod \ M_i} + \breve{S}_i(j)\right)\right)$ to quantify its *interfering* execution time instead of the request execution time, since the latter may not be effectively executed within the interval of length t. Moreover, $S_i(j)$ can be thought of as the minimum interval-arrival time between the $(j-1)$-th and the j-th released computation segments. Therefore, if the $(j-1)$-th and the j-th computation segments belong to the same job of task τ_i, then $S_i(j) = \breve{S}_i^{j \ mod \ M_i}$. Otherwise, the j-th computation segment belongs to the next released job of task τ_i. In this case, the minimum inter-arrival time between the $(j-1)$-th and the j-th computation segments may depend on whether the j-th computation segment comes from the carry-in job, that is, $T_i - D_i$ for the carry-in job; otherwise, $T_i - (C_i + \breve{S}_i)$.

Note that the derivation of the workload function is based on the setting $D_i \leq T_i$ and the assumption that task τ_i can meet its deadline, without referring to any properties of the scheduling policy. We only quantify the worst-case accumulated execution time of task τ_i that can interfere lower-priority tasks, within an interval of length t. For the rest of this section, we will show that given a candidate computation segment C_i^h, the multi-segment workload function above is the upper bound on the interference from high-priority self-suspension task τ_i.

In contrast with the conventional sporadic task, an early completion on a computation segment (if the execution time is less than the worst-case execution time) of a self-suspension task will advance the following releases of suspension intervals and computation segments. This may seem at first glance bring some additional interferences into the interval of interest. On the other hand, each suspension interval executed for longer than its lower bound $\breve{S}_i^j$ time-units may postpone the following releases of computation segments. In summary, we have the following two observations, to be proven in Lemma 1: for any interval length of t and index h

- The multi-segment workload function of task τ_i is non-increasing when each suspension interval is *increased.*
- The multi-segment workload function of task τ_i is non-increasing when each computation segment *decreases* its execution time.

Let $W_i^h(t)$ and $\bar{W}_i^h(t)$ denote the multi-segment workload function of task τ_i and $\bar{\tau}_i$, respectively, such that $T_i = \bar{T}_i, D_i = \bar{D}_i$, for each computation segment, $\bar{C}_i^j \leq C_i^j$, and, for each suspension interval, $\bar{S}_i^j \geq \breve{S}_i^j$.

Lemma 1: Given h, for all $t > 0$,

$$W_i^h(t) \geq \bar{W}_i^h(t) \tag{3}$$

Proof: We know that each computation segment is eligible to be executed only after the completion of the previous suspension interval. Hence, an increase on any suspension interval will postpone the following execution and thus the interference cannot be increased.

Also, a decrease Δ on any computation segment will bring *at most* an increase by Δ execution of the following computation segments within the same release into the interval

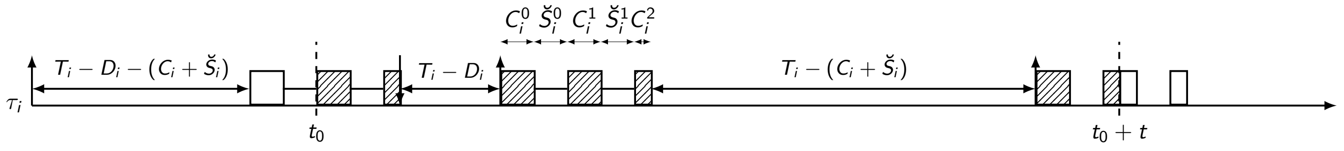
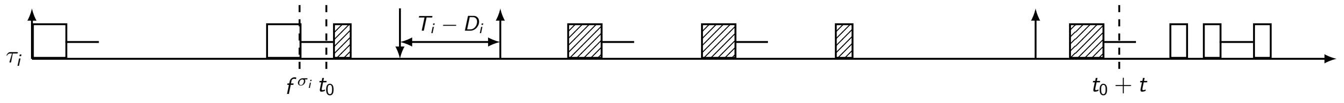
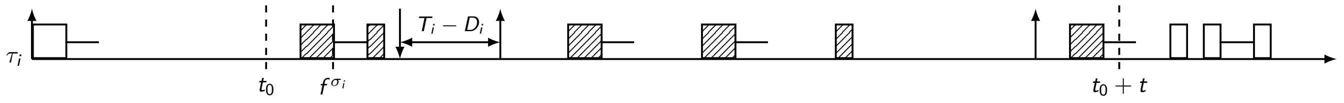

C_i^0 $\breve{S}_i^0$ C_i^1 $\breve{S}_i^1$ C_i^2
$T_i - D_i - (C_i + \breve{S}_i)$
$T_i - D_i$
$T_i - (C_i + \breve{S}_i)$
τ_i
t_0
$t_0 + t$

Fig. 2: Multi-segment workload function for $h = 1$, where the striped rectangles indicate the workload in this interval.



(a) The schedule when $t_0 > f^{\sigma_i}$. In this case the work contributing over $[t_0, t_0 + t)$ is bounded by the workload function for $\sigma_i = 2$.



(b) The schedule when $t_0 \leq f^{\sigma_i}$. In this case the work contributing over $[t_0, t_0 + t)$ is bounded by the workload function for $\sigma_i = 1$.

Fig. 3: The illustration in the schedule for two different cases in the proof of Theorem 1.

length of t. Hence, this lemma is proved. ∎

We have defined the multi-segment workload function in Eq. (1) by assuming that a computation segment of task τ_i arrives at time t_0. The following theorem shows that this function indeed bounds the maximum interference for a given interval length t resulting from task τ_i if h is chosen properly.

Theorem 1: Suppose that task τ_i can meet its deadlines under the given scheduling policy. For any interval length t, there exists an index of computation segment $h_i \in [0, M_i - 1]$ such that the work contributing to the interval $[t_0, t_0 + t)$ from SSS task τ_i is bounded from above by $W_i^{h_i}(t)$.

Proof: Consider the work contributing over the interval $[t_0, t_0 + t)$ by a legal sequence of jobs executed by task τ_i that is assumed to be schedulable under the scheduling policy. Let $C_i^{\sigma_i}$ be the *latest* computation segment released prior to t_0 and f^{σ_i} denote its completion time. We now consider two separate cases:

- $f^{\sigma_i} < t_0$. In this case, we can see that by the choice of the latest segment $C_i^{\sigma_i}$ prior to t_0, there is no work from task τ_i executing over $[f^{\sigma_i}, t_0]$. Hence, we can use the workload function $\bar{W}_i^h(t)$ for $h = ((\sigma_i + 1) \ mod \ M_i)$ to bound the cumulative execution in the interval $[t_0, t_0 + t)$.
- $f^{\sigma_i} \geq t_0$. In this case, we can use the workload function $\bar{W}_i^h$ for $h = \sigma_i$ to bound the cumulative execution in the interval $[t_0, t_0 + t)$.

The above two cases are also illustrated in Figures 3a and 3b, respectively. In either case we may see that the work contributing over $[t_0, t_0 + t)$ is bounded by its corresponding multi-segment function $\bar{W}_i^h(t)$. By Lemma 1, we know that the maximum cumulative execution over $[t_0, t_0 + t)$ where the computation segment may be less than the upper bound on execution time and where the suspension interval length may be larger than its minimum time on suspension is bounded from above by the multi-segment workload function $W_i^h(t)$ defined in Eq. (1). Thus, there must exist $h_i \in [0, M_i - 1]$ such that the work contributing over $[t_0, t_0 + t)$ is bounded by the multi-segment workload function $W_i^{h_i}(t)$ defined in Eq. (1) for any interval length of t. ∎

Figure 2 also demonstrates that each job of task τ_i that arrives in and has its absolute deadline within the interval $[t_0, t_0 + t)$, called a *body job*, must contribute to the interval for the amount C_i of execution times. This implies that we can reduce the time complexity to evaluate Eq. (1) by skipping these body jobs and counting entirely all their computation segments, i.e., $\sum_{j=0}^{M_i-1} C_i^j = C_i$. Therefore, Eq. (1) can be equivalently decomposed as follows:

$$W_i^h(t) = W_i^h\left(t - \left[\left\lfloor \frac{t - 2T_i}{T_i} \right\rfloor T_i\right]^\dagger\right) + \left[\left\lfloor \frac{t - 2T_i}{T_i} \right\rfloor C_i\right]^\dagger \tag{4}$$

where $[x]^\dagger$ is defined as $max(0, x)$.

As a result, only those computation segments that belong to (*i*) the carry-in job: released prior to t_0 and have absolute deadline within $[t_0, t_0 + t)$ and (*ii*) the tail job: the last job released within $[t_0, t_0 + t)$ and has absolute deadline after $t_0 + t$ have to be evaluated, in total at most $2M_i$ computation segments. The above multi-segment workload function $W_i^h(t)$ can thus be computed in linear-time in the size of M_i, for a given interval length of t.

Maximum workload function. Without knowing the worst-case scenario, the interference from higher-priority tasks can only be quantified by enumerating all possible release sequences $h_i \in [0, M_i - 1]$ for all the higher-priority tasks according to Theorem 1 but it is computationally intractable since the time complexity is $O(M_1 \times M_2 \times ...M_n)$. Therefore, we here introduce the maximum workload function of task τ_i, $W_i(t)$, for providing an amenable solution.

$$W_i(t) = max_{h_i \in [0, M_i - 1]}\{W_i^{h_i}(t)\} \tag{5}$$

The interference from higher-priority suspending tasks can be thereafter bounded above by the summation of $W_i(t)$ for all the tasks, in the interval $[t_0, t_0 + t]$. Note that the maximum workload function $W_i(t)$ for a given t can be computed in polynomial-time $O(M_i^2)$.

In addition to the computational tractability, the sum of the maximum workload functions of the higher-priority tasks are fully independent from each other. This attribute is important for being compatible with *Optimal Priority Assignment (OPA)*

Algorithm, and allows us to use OPA to achieve high schedulability (to be discussed in Section V).

IV. Our Proposed Response Time Analysis

In this section we derive a response time analysis for an analyzed SSS task τ_k based on the maximum workload function established in Section III, under the assumption that all the higher-priority tasks are already verified to be schedulable. We will show that the response time analysis for task τ_k can be successfully done by two methods:

- *SC* (Suspension as Computation): This method models every suspension interval of task τ_k as computation segments. Informally speaking, task τ_k can be thought of as a dynamic SSS task in this method, like in [14], [22]. However, since our maximum workload function that characterizes the interference of a higher-priority task τ_i in Section III is tighter for a multi-segment SSS task than the interference used in PASS in [14] for a dynamic SSS task, this method is still better than the state-of-the-art analysis for dynamic SSS systems.
- *AIR* (As Interference Restarts): This method treats each of the computation segments of task τ_k as if the worst-case higher-priority interference that it suffers restarts, regardless of its previous computation segments.

The above two methods can be easily compared for their advantages and disadvantages. SC is very useful when the suspension length of task τ_k is short enough, wheres AIR does not work well for such cases. In constrast, AIR is very useful when the length of a suspension interval is long enough, wheres SC does not work well for such cases.

Before we proceed to present these two methods, it is also useful to explain why calculating the response time is non-trivial, even if the maximum workload function derived in Section IV is available. Intuitively, one may release task τ_k along with all maximum workload (interference) function $W_i(t)$ simultaneously and calculate its corresponding response time from the resulting schedule. But, unfortunately, this may lead to an overoptimistic worst-case response time. We show this in the following example:

Example. Consider a self-suspension real-time system consisting of two self-suspension tasks $\tau = \{\tau_1 = ((0.5, 3, 0.5), 4, 4), \tau_2 = ((6, 2, 1), T, T)\}$ where T is any number larger than 12, as shown in Figure 4. Task τ_1 has higher-priority than task τ_2. Notice that there is no slack for task τ_1. By making use of the multi-segment workload function, it is not different to see that the interference starting from the second computation segment ($h = 1$) dominates that from the first one ($h = 0$). Thus, aligning task τ_2's release with task τ_1's second computation segment results in the most unfavorable interference, as shown in Figure 4a. Consequently, it results in a worst-case response time of 11. However, it can also be easily seen that releasing task τ_2 at time 1.5 results in a response time of 12, shown in Figure 4b. Therefore, the maximum workload function developed in Section III has to be used very carefully. Releasing synchronously the worst-case interference only from the first computation segment is not safe enough. □

To ensure the upper bound on the response time of the analyzed task τ_k, a conservative way is to model every suspension interval as a computation segment released with its upper bound, i.e., $\hat{S}_k^j$. Then, we can safely calculate the response time of the equivalent non-suspending task by releasing higher-priority tasks with their maximum workload functions simultaneously. That is, the standard Response Time Analysis (RTA) [2], [15] approach first computes the worst-case response time of the equivalent sporadic non-suspending task, and then compares this value with the tasks deadline D_i. We state this with the following theorem

Theorem 2 (SC): If the worst-case response time of task τ_k is $\leq T_k$, then the worst-case response time of multi-segment SSS task τ_k is upper bounded by the smallest R_k satisfying the following recurrence:

$$R_k = C_k + \hat{S}_k + \sum_{\tau_i \in hp(k)} W_i(R_k) \tag{6}$$

where $W_i(t)$ is defined in Eq. (5) and $hp(k)$ denotes the set of the tasks with higher priority than task τ_k.

Proof: Similar to the proof of Lemma 3 in [22] for dynamic SSS tasks, converting the maximum self-suspension time of multi-segment SSS task τ_k to computation time makes the worst-case response time of task τ_k larger (if the worst-case response time after converting is no more than T_k). Let τ_k' be a computation task without any self-suspension, in which its worst-case execution time is $C_k' = C_k + \hat{S}_k$ and minimal inter-arrival time is $T_k' = T_k$.

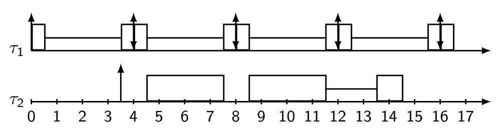
Now, we consider a fixed-priority schedule of $hp(k)$ and τ_k' under the same priority ordering, i.e., τ_k' has lower priority than $hp(k)$. Let t_0 be the arrival time of a job of task τ_k'. By Theorem 1 and Eq. (5) and the assumption that all the higher-priority tasks in $hp(k)$ can meet their deadlines in an SSS task system with constrained deadlines, we know that the maximum workload executed from $hp(k)$ in the interval $[t_0, t_0 + t)$ is at most $\sum_{\tau_i \in hp(k)} W_i(t)$. To finish C_k' amount of workload of task τ_k from t_0 to $t_0 + t$, we need $C_k' + \sum_{\tau_i \in hp(k)} W_i(t) \leq t$. Therefore, the minimum t with $C_k' + \sum_{\tau_i \in hp(k)} W_i(t) \leq t$ is the worst-case response time, provided that $t \leq T_k$ to ensure that any job of task τ_k' does not have remaining execution time before the next release. ■

The accuracy of Theorem 2 (i.e., SC) depends on the ratio of the suspension length $\hat{S}_k$ to the total computation time C_k of task τ_k. That is, it becomes too pessimistic when $\frac{\hat{S}_k}{C_k}$ is large enough. This is also shown in the example in Appendix B where the suspension interval is modeled as the computation computation segment when a dynamic self-suspension task is analyzed. To overcome this problem, the alternative is to individually compute the response time of each computation segment that undergoes the maximum interference, called AIR. Towards this end, we need the following lemma.

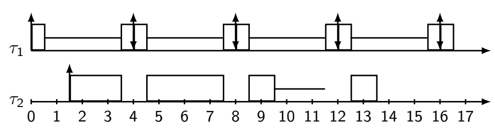
Lemma 2: Let R_k^j denote the smallest value t satisfying the following recurrence:

$$t = C_k^j + \sum_{\tau_i \in hp(k)} W_i(t), \tag{7}$$

where $W_i(t)$ is defined in Eq. (5) and $hp(k)$ denotes the set of the tasks with higher priority than task τ_k. The response time of computation segment C_k^j of SSS task τ_k is at most R_k^j if the worst-case response time of task τ_k is no more than T_k.



(a) The synchronous release results in a response time of 11



(b) The asynchronous release results in a response time of 12

Fig. 4: Different response times for synchronous and asynchronous releases of the analyzed task

Proof: We prove this lemma by contradiction: *the response time of computation segment C_k^j is $R > R_k^j$.*

Let t_0 be the release time of computation segment C_k^j in the schedule. Due to the SSS model and the assumption that the worst-case response time of task τ_k is no more than T_k, task τ_k does not have any other demand ready to be executed at time t_0, except C_k^j. Since the response time of computation segment C_k^j is greater than R_k^j, the interval $[t_0, t_0 + R_k^j]$ must be busy, and computation segment C_k^j has not finished within the interval. By Theorem 1 and the definition of the maximum workload function, the maximum cumulative execution of each task is bounded by $W_i(t)$ with any interval length of t. Since $t_0 + R_k^j$ is the time-instant at which $R_k^j = C_k^j + \sum_{\tau_i \in hp(k)} W_i(t)$, we know that computation segment C_k^j has completed its execution at time $t_0 + R_k^j$ with at most a response time R_k^j, which contradicts to our assumption. ■

Once obtaining the response time of every computation segment, we can calculate the total response time by summing them up in addition to the total suspension length. We state this with the following theorem:

Theorem 3 (AIR): If the worst-case response time of task τ_k is $\leq T_k$, then the worst-case response time of task τ_k is upper bounded by R_k:

$$R_k = \hat{S}_k + \sum_{j=0}^{M_k-1} R_k^j \tag{8}$$

where R_k^j is the smallest value t satisfying Eq. (7).

Proof: This comes directly from Lemma 2. ■

In either way, we can see that the response time of SSS task τ_k is upper bounded. Hence, Theorem 4 follows immediately:

Theorem 4 (SCAIR): A sporadic self-suspension task τ_k is schedulable under a fixed-priority scheduling policy if the smaller of the two upper bounds for R_k using Eq. (6) and (8) is less than or equal to D_k.

Proof: This comes directly from Theorems 2 and 3. ■

Computational complexity. As discussed in the previous section, the maximum workload function $W_i(t)$ can be computed in polynomial-time $O(M_i^2)$ for a given time t. Checking the schedulability of an SSS task τ_k can be done in pseudo-polynomial time, i.e., $O(D_k \times \sum_{\tau_i \in hp(\tau_k)} M_i^2)$.

V. Priority Assignment and Proposed Approach

In this section we will present how a feasible priority assignment in a multi-segment self-suspension system under fixed-priority scheduling can be determined in polynomial time. With respect to schedulability to meet the deadlines, *deadline-monotonic* (DM) scheduling is known to be an optimal fixed-priority scheduling for a sporadic constrained-deadline real-time system *without* self-suspensions. For self-suspending systems it has been shown in [8] that no lower bound on processor speed-up factor is guaranteed for RM scheduling in implicit-deadline SSS systems.

A priority assignment determines the priority levels of the given tasks, defined as follows:

Definition 2 (priority assignment): Let π be a priority assignment as a bijective function $\pi : \tau \rightarrow \{1, 2, \ldots, n\}$ to define the priority level of task $\tau_i \in \tau$. Priority levels are numbered from 1 to n where 1 is the highest and n the lowest. □

As there are $n!$ possible priority orderings, testing all of them to find a feasible priority assignment is computationally intractable. For arbitrary-deadline real-time systems without self-suspensions, fortunately, it has been shown in [4] that it suffices to examine a polynomial number of priority orderings to find a feasible priority ordering, if one exists. The method is called *Optimal Priority Assignment (OPA) Algorithm* [3], [4], [9]. The OPA algorithm assigns each priority level k to one of the unassigned tasks that has no deadline miss along with the other unassigned tasks, assumed to have higher priorities. The iterative priority assignment terminates as soon as either no unassigned task can be assigned at the priority level k or all priority levels are assigned.

The OPA algorithm was originally proposed for handling arbitrary-deadline real-time systems without self-suspensions [3], [4], [9]. It has been recently shown in [9] that the OPA algorithm can always find a feasible priority assignment if one exists by the sufficient schedulability test that complies with three conditions. For completeness, we state these three conditions as follows:

- **Condition 1.** The schedulability of a task τ_k may, according to schedulability test $\mathcal{S}$, depend on any independent properties of tasks with priorities higher than τ_k, but not on any properties of those tasks that relate to their relative priority ordering.
- **Condition 2.** The schedulability of a task τ_k may, according to schedulability test $\mathcal{S}$, depend on any independent properties of tasks with priorities lower than τ_k, but not on any properties of those tasks that relate to their relative priority ordering.
- **Condition 3.** When the priorities of any two tasks of adjacent priority are swapped, the task being assigned the higher priority cannot become unschedulable according to schedulability test $\mathcal{S}$, if it was previously schedulable at the lower priority.

If the above three conditions can be satisfied for a schedulability test, then the test is called *OPA-compatible*. It is not difficult to see that the RTA by Theorem 4 complies with the required conditions for the OPA algorithm. We state this with the following lemma.

Algorithm 1: SCAIR-OPA

```
input : A set of multi-segment self-suspending tasks $\tau$
output: Priority assignment $\pi$ and the feasibility of system $\tau$
$\pi \leftarrow \emptyset$;
for each priority $k$ from $|\tau|$ to 1 do
    for each unassigned task $\tau_i$ do
        if task $\tau_i$ is schedulable at priority $k$ with all unassigned
        tasks (assume them as higher-priority tasks) according to
        Theorem 4 then
            $\pi(\tau_i) \leftarrow k$ // assign task $\tau_i$ to priority $k$
            break (continue the outer loop) ;
    if priority level $k$ is not assigned with any task then
        return "unscheduable";
return $\pi$ as a "scheduable priority assignment";
```

Lemma 3: The sufficient schedulability test by Theorem 4 is OPA-compatible.

Proof: Eqs. (6) and (8) for the schedulability of task τ_k depend only on the set of higher-priority tasks but not on their relative priority ordering. Hence, Condition 1 holds. Similarly, they are independent on the set of lower-priority tasks, and hence Condition 2 holds. Consider two tasks τ_a and τ_b initially at priorities k and $k+1$, respectively. If task τ_b is schedulable, it is still schedulable when it is shifted one priority level up to priority level k, since the only change of higher-priority task demand is the removal of task τ_a from the tasks that are assigned higher priority than task τ_b. Hence, Condition 3 holds. ■

Algorithm 1 shows the proposed approach adopting the OPA algorithm, called SCAIR-OPA, for the feasible priority assignment in self-suspending systems. Lemma 3 suggests the following theorem.

Theorem 5: If there exist feasible priority assignments by adopting Theorem 4 as the schedulability test, the proposed *SCAIR-OPA* returns one of them.

Proof: From Lemma 3 the sufficient test by Theorem 4 is compliant with the above conditions. Following the proof of Theorem 3 in [9], we here conclude this theorem. ■

VI. Experimental Results

In this section, we conduct extensive experiments using synthesized task sets for evaluating the proposed test. To the best of our knowledge, there was no polynomial-time or pseudo-polynomial-time schedulability test ([6] is flawed) for the multi-segment self-suspension task model. To analyze the multi-segment SSS tasks, we can still adopt the known schedulability tests for dynamic self-suspension systems [14], [22]. These evaluated tests are listed as follows:

- *XDM*: the pseudopolynomial-time analysis where we transform every task into a non-suspending task by modeling every suspension interval as computation segments and use the standard RTA under *deadline-monotonic* (DM) scheduling.
- *Idv-Burst-RM*: the polynomial-time utilization-based test for dynamic self-suspension tasks in Corollary 2 in [22].
- *PASS-OPA*: the pseudopolynomial-time algorithm and analysis presented in [14]. Note that PASS-OPA is the best known analysis handling the general dynamic self-suspension system under fixed-priority scheduling.
- *SCAIR-OPA*: Algorithm 1 in this paper, with pseudopolynomial-time complexity.

The metric to compare the results is to measure the *acceptance ratio* of the above tests with respect to a given goal of task set utilization. We generate 100 task sets for each utilization level. The acceptance ratio of a level is said to be the number of task sets that are schedulable divided by the number of task sets for this level, i.e., 100.

We first investigate the impact of suspension interval lengths and suspension types, of SSS tasks, on the acceptance ratio, by assuming that the upper and lower bounds on suspension intervals are equal. In the second experiment we further explore the impact of the lower bound of suspension interval times on the acceptance ratio.

A. Simulation Setup- & Result I

We first generated a set of sporadic tasks. The cardinality of the task set was 10. The UUniFast method [5] was adopted to generate a set of utilization values with the given goal. We here used the approach suggested by Davis and Burns [10] to generate the task period according to an exponential distribution. The distribution is of two orders of magnitude, i.e., $[1ms - 100ms]$. The execution time was set accordingly, i.e., $C_i = T_i U_i$. Task relative deadlines were implicit, i.e., $D_i = T_i$.

We then converted all of the sporadic tasks to SSS tasks. Suspension lengths of the tasks were then generated in a similar manner to the method used in [21]. Suspension lengths of the tasks were generated according to a uniform random distribution, in one of three ranges depending on the self-suspension length (sslen): $[0.01(T_i - C_i), 0.1(T_i - C_i)]$ (short suspension, sslen=S), $[0.1(T_i - C_i), 0.6(T_i - C_i)]$ (medium suspension, sslen=M), and $[0.6(T_i - C_i), T_i - C_i]$ (long suspension, sslen=L). We assume $\hat{S}_i^j = \check{S}^j$ for each suspension interval for all SSS tasks.

The number of computation segments M_i was set depending on the following types of self-suspensions: 2 (rare suspension, sstype=R), 5 (moderate suspension, sstype=M), and 10 (frequent suspension, sstype=F). We then generated every computation segment C_i^j and suspension interval $\check{S}_i^j$ with the given C_i and S_i, according to a uniform distribution, like the UUniFast method.

Results. Figure 5 presents the result for the performance in terms of the acceptance ratio. Not surprisingly, the naive approach XDM is only effective when the suspension length is short enough, i.e., sslength=S. Also, as shown earlier, RM scheduling is not optimal in self-suspension systems. Hence, Idv-Burst-RM is hardly to be effective even the state-of-the-art utilization-based analysis has been shown in [22]. It is clear that our proposed SCAIR-OPA is far more effective to PASS-OPA. For all the tests, the acceptance ratio decreases when the number of suspension intervals, for all different suspension types, increases.

For the cases with short suspension lengths, i.e., Figure 5a, 5b, and 5c, PASS-OPA and SCAIR-OPA are able to admit the task sets with a utilization of up to 75% with noticeable acceptance ratios. For such cases, the improvement by considering multi-segment SSS tasks over dynamic SSS tasks is not significant but still visible. For the other cases, the

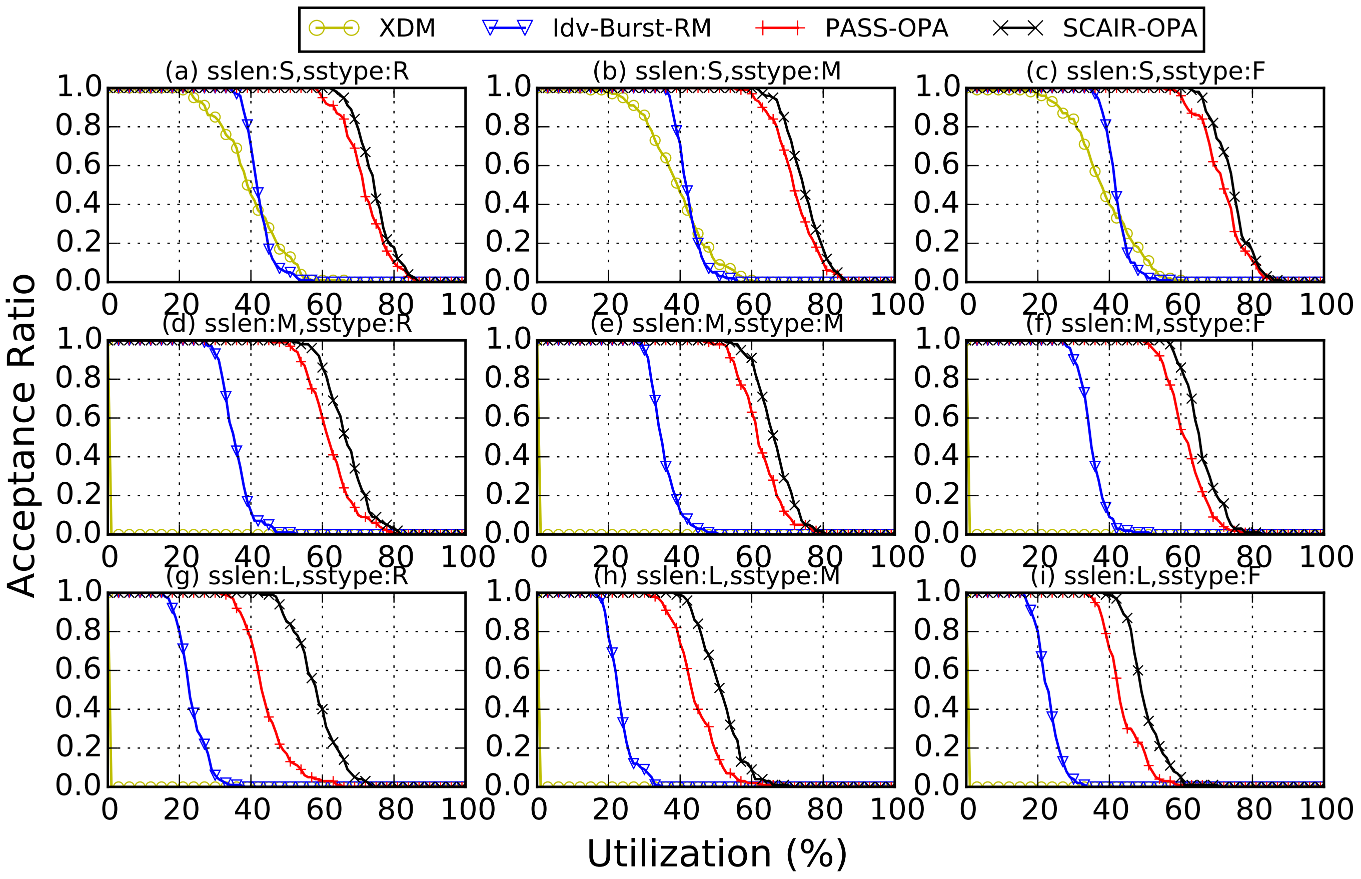


Fig. 5: Comparison with different types of suspension lengths (sslength) and different types of suspension frequency (sstype) of SSS tasks. In the first (respectively, second and third) column in the figure, rare (respectively, moderate and frequent) self-suspension frequency is assumed. In the first (respectively, second and third) row in the figure, short (respectively, medium and long) suspension length is assumed.

proposed SCAIR-OPA can achieve significant improvement over PASS-OPA, especially for the cases with long suspension lengths (Figure 5g, 5h, and 5i).

It is noticeable that the acceptance ratio of the proposed SCAIR-OPA still drops when the number of suspension segments increases, especially for long suspension lengths (Figure 5g, 5h, and 5i). In such cases, the *SC* test by Theorem 2 can hardly be better than the *AIR* test by Theorem 3, since the conversion of suspensions to computations, of task τ_k, is overly pessimistic. However, the pessimism on calculating the response time of each computation segment by the *AIR* test in Theorem 3 propagates when the number of suspension intervals increases.

Furthermore, for rare suspensions (Figure 5a, 5d, and 5g), we can see that the case of long suspension lengths (Figure 5g) can be successfully scheduled with a utilization of up to 40% with noticeable acceptance ratios. In conclusion, SCAIR-OPA significantly improves the schedulability by the best-known test for dynamic self-suspension tasks.

B. Simulation Setup- & Result II

In this experiment we evaluated how the variability of the lower bounds of the suspension interval lengths affects the schedulability. We generated a set of sporadic self-suspension tasks in the same manner in Section VI-A. The suspension interval generated by UUniFast was set to its upper bound time on the suspension interval. Then, we use a scaling factor b to assign the lower bound time on the suspension interval, i.e., $\check{S}_i^j = b\hat{S}_i^j$. The scale factor b is chosen from one of four values: $[0, 0.25, 0.5, 0.75, 1]$, denoted by SCAIR-OPA-0, SCAIR-OPA-25, SCAIR-OPA-50, SCAIR-OPA-75, and SCAIR-OPA-100, respectively.

Theoretically, SCAIR-OPA for multi-segment self-suspension tasks outperforms PASS-OPA for dynamic self-suspension tasks where the pattern of suspension intervals is oblivious. As shown in Figure 5, the gaps in acceptance ratio between these two tests are small for the cases of short and medium suspension lengths. Hence, we only report the effect of the variability of lower bounds for the case of long suspension lengths.

Results. Figure 6 shows that performance comparison when the suspension length is long. The schedulability decreases when the lower bound of the suspension lengths decreases. For PASS-OPA and SCAIR-OPA-0, even though the lower bound of a suspension interval can be 0, SCAIR-OPA-0 is better than PASS-OPA due to its awareness of the computation segments characterized in the multi-segment self-suspension task model.

VII. Remarks on Early Suspension Completions

With the above discussions, we notice that the worst-case interference of a self-suspension task is evaluated by considering the lower bound of the self-suspension lengths. However, when analyzing the worst-case response time of a

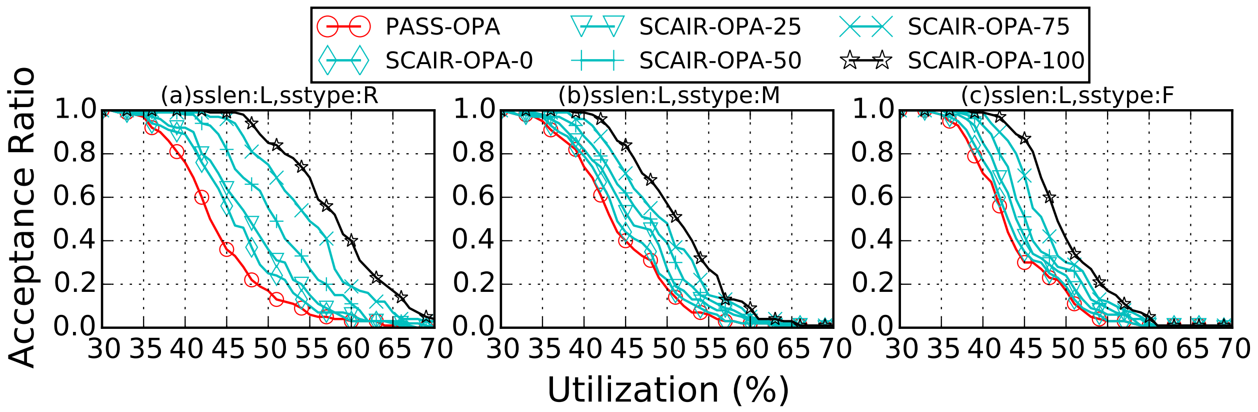


Fig. 6: Comparison for different types of suspension frequency, with long suspension length

task, we have to consider its worst-case suspension interval lengths. Therefore, it is more difficult to be schedulable if $\check{S}_i$ is small and $\hat{S}_i$ is large for every task τ_i.

A very clear message from the above analysis is "**Do not allow early suspension completions, if possible!**". The early suspension completion may simply result in unnecessary jitters and bursts. The task τ_i with early suspension completion may finish its execution earlier in the average case, but it creates more interference to lower-priority task τ_k. From the perspective of schedulability, such flexibility does not provide any gain in the worst cases but decreases the schedulability.

Therefore, if possible, it is recommended to make the suspension interval length deterministic without any flexibility by taking the worst case. This can be done by designing the suspensions more properly. For example, every suspension has to initialize a timer (interrupt) to set a fixed suspension length. Even if the operation of the self-suspension is done, the following computation segment is not placed back to the ready-queue until the timer is triggered. As shown in Section VI-B, this may improve the schedulability significantly.

VIII. Conclusions

In this paper we address the problem of scheduling and the priority assignment for multi-segment self-suspension tasks under fixed-priority scheduling. We derive the multi-segment workload function that bounds the maximum cumulative execution of a self-suspension task. Based on this function, we present two methods for analyzing the response time of a self-suspension task. Empirical results show that our analysis together with the optimal priority assignment is highly effective in terms of schedulability. The self-suspension task is deemed to reach the poor schedulability when a long suspension interval exists. The proposed test fully tackles this case, and empirical results show that our proposed test is able to accept a task set with utilization up to 75% in such a case with noticeable acceptance ratios. Nevertheless, the worst-case quality of the proposed test is not studied. In future work it is interesting to see whether there is a processor speed so that the proposed test will lead to a feasible schedule if one exists upon a unit-speed processor.

References

[1] K. Albers and F. Slomka. An event stream driven approximation for the analysis of real-time systems. In *ECRTS*, pages 187–195, 2004.

[2] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, 1993.

[3] N. C. Audsley. *Optimal priority assignment and feasibility of static priority tasks with arbitrary start times*. Citeseer, 1991.

[4] N. C. Audsley. On priority assignment in fixed priority scheduling. *Information Processing Letters*, 79(1):39–44, 2001.

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

[6] K. Bletsas and N. C. Audsley. Extended analysis with reduced pessimism for systems with limited parallelism. In *RTCSA*, pages 525–531, 2005.

[7] S. Chakraborty, S. Künzli, and L. Thiele. Approximate schedulability analysis. In *IEEE Real-Time Systems Symposium*, pages 159–168, 2002.

[8] J.-J. Chen and C. Liu. Fixed-relative-deadline scheduling of hard real-time tasks with self-suspensions. In *Real-Time Systems Symposium (RTSS)*, 2014.

[9] R. I. Davis and A. Burns. Improved priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems. *Real-Time Systems*, 47(1):1–40, 2011.

[10] R. I. Davis, A. Zabos, and A. Burns. Efficient exact schedulability tests for fixed priority real-time systems. *Computers, IEEE Transactions on*, 57(9):1261–1276, 2008.

[11] G. R. Geoffrey Nelissen, Jos Fonseca and V. Nelis. Timing analysis of fixed priority self-suspending sporadic tasks. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2015.

[12] W.-H. Huang and J.-J. Chen. Self-suspending real-time tasks under fixed-relative-deadline fixed-priority scheduling. Submitted to RTNS15.

[13] 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, Dortmund, Germany, 2015.

[14] W.-H. Hung, J.-J. Chen, H. Zhou, and C. Liu. PASS: Priority assignment of real-time tasks with dynamic suspending behavior under fixed-priority scheduling. In *Design Automation Conference (DAC)*, 2015.

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

[16] W. Kang, S. Son, J. Stankovic, and M. Amirijoo. I/O-Aware Deadline Miss Ratio Management in Real-Time Embedded Databases. In *Proc. of the 28th IEEE Real-Time Systems Symp. (RTSS)*, pages 277–287, 2007.

[17] S. Kato, K. Lakshmanan, A. Kumar, M. Kelkar, Y. Ishikawa, and R. Rajkumar. RGEM: A responsive GPGPU execution model for runtime engines. In *Real-Time Systems Symposium (RTSS)*, pages 57–66, 2011.

[18] J. Kim, B. Andersson, D. d. Niz, and R. R. Rajkumar. Segment-fixed priority scheduling for self-suspending real-time tasks. In *Real-Time Systems Symposium (RTSS)*, pages 246–257, 2013.

[19] K. Lakshmanan and R. Rajkumar. Scheduling self-suspending real-time tasks with rate-monotonic priorities. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 3–12, 2010.

[20] C. Liu and J. H. Anderson. Supporting sporadic pipelined tasks with early-releasing in soft real-time multiprocessor systems. In *RTCSA*, pages 284–293, 2009.

[21] C. Liu and J. H. Anderson. Task scheduling with self-suspensions in soft real-time multiprocessor systems. In *Real-Time Systems Symposium*, pages 425–436, 2009.

[22] C. Liu and J. Chen. Bursty-interference analysis techniques for analyzing complex real-time task models. In *Real-Time Systems Symposium (RTSS)*, pages 173–183, 2014.

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

[24] J. W. Liu. *Real-time systems. 2000.* Prentice Hall.

[25] W. Liu, J. Chen, A. Toma, T. Kuo, and Q. Deng. Computation offloading by using timing unreliable components in real-time systems. In *Design Automation Conference (DAC)*, 2014.

[26] F. Ridouard, P. Richard, and F. Cottet. Negative results for scheduling independent hard real-time tasks with self-suspensions. In *Real-Time Systems Symposium (RTSS)*, pages 47–56, 2004.

Appendix A
Counter Example for the Test in [6]

Consider a self-suspension system consisting of three tasks: sporadic task $\tau_1 = ((1), 4, 4)$, SSS task $\tau_2 = ((1, 2, 1), 6, 6)$ with one suspension interval and two computation segments, and sporadic task $\tau_3 = ((1), T, 3)$ where $T \geq 4$. We assume that the minimum and maximum times on the suspension interval of task τ_2 are same. Task τ_1 is assigned with the highest priority whereas task τ_3 with the lowest priority. It is clear that task τ_1 and task τ_2 are schedulable. We then check the schedulability of task τ_3. The response time analysis proposed by Bletsas and Audsley [6] is shown as follows:

$$R_k = C_k + \hat{S}_k + \sum_{i \in hp(k)} \sum_{j=0}^{n(\tau_i)} \left\lceil \frac{R_k - O_i^j + A_i}{T_i} \right\rceil u(R_k - O_i^j) C_i^j \tag{9}$$

where

$$O_i^j = \sum_{j=0}^{M_i - 1} (C_i^j + \breve{S}_i^j) \tag{10}$$

$$A_i = S_i - \breve{S}_i \tag{11}$$

and

$$u(t) = \begin{cases} 1 & \text{if } t \geq 0, \\ 0 & \text{otherwise.} \end{cases} \tag{12}$$

According to their analysis, a transformation, called *synthetic* distribution, has to be done first to provide an upper bound on the worst-case interference. Due to the equal size of the computation segments and suspension intervals of task τ_2 in our example, such a transformation has no impact in our example. In addition, since the maximum and minimum times on suspension intervals are the same, the jitter A_i is set to zero. Consequently, the worst-case scenario provided by Eq. (9) is to release all the higher-priority tasks as synchronous arrival releases and the following jobs as early as possible. By using the response time analysis by Bletsas and Audsley [6] a response time of 3 for task τ_3 is answered, whereas the asynchronous release, arriving at time-instant 4, results in a response time of 4, as shown in Figure 7. Hence, the analysis proposed by Bletsas and Audsley in [6] guarantees the schedulability of task τ_3 that in fact misses its deadline in the worst-case.

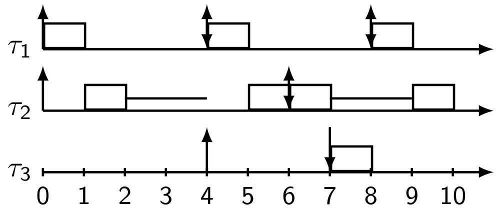


Fig. 7: A counterexample for the analysis in [6]

Appendix B
Pessimism by Converting Multi-Segment to Dynamic Self-Suspension Model

We demonstrate the pessimism by presenting the resulting speed-up factor in the following example. If an algorithm $\mathcal{A}$ has a speed-up factor α, then it guarantees that the schedule derived from the algorithm $\mathcal{A}$ is always feasible by running at speed α, if the input task set admits a feasible schedule on a unit-speed processor. With such a transformation on a speed-α uniprocessor, the computation segment C_i^j becomes $\frac{C_i^j}{\alpha}$; however, $\hat{S}_i^j$ remains the same.

Example. Consider a self-suspension real-time system consisting of two tasks $\tau = \{\tau_1 = ((0.5, F-1, 0.5), F, F), \tau_2 = ((\epsilon, T-2-2\epsilon, \epsilon), T, T)\}$, where F is an integer larger than 1, T is an integer larger than 2 and divisible by F, and ϵ can be an arbitrarily small positive real number, as shown in Figure 8. Task τ_1 has higher priority than task τ_2. Clearly, task τ_1 will meet its deadline. The worst-case scenario of τ_2 occurs when task τ_2 and the second computation segment of τ_1 are released simultaneously. Consequently, the first computation segment of task τ_2 finishes with a response time of $1 + \epsilon$, and task τ_2 completes its execution with a response time of $T - 1$ while its second computation segment experiences no interference from task τ_1. Therefore, we can conclude that task τ_2 is schedulable. Finding the worst-case scenario for a self-suspension task is not trivial. Alternatively, we can pessimistically assume that every computation segment suffers the possible maximum interference from the high-priority tasks, and then calculate the response time of the task accordingly, as presented in Section IV. In the example, each computation segment of task τ_1 can experience at most one time-unit interference from task τ_1 before its completion. Consequently, the response time of task τ_2 is at most $1+1+2\epsilon+(T-2-2\epsilon) \leq T$, which is in fact schedulable. However, the transformed self-suspension task τ_2,

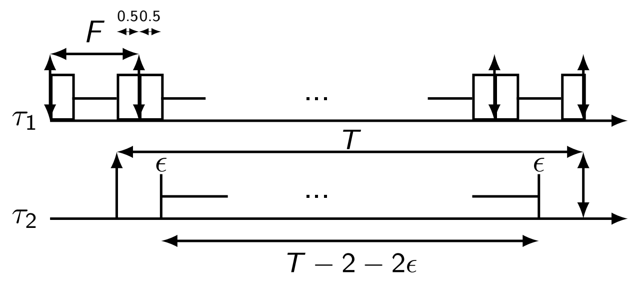


Fig. 8: The ineffectiveness of modeling suspension intervals as computation time

which is equivalent to a non-suspending task $((T-2), T, T)$, will yield no speedup factor: it is necessary for a deadline to be met on α-speed processor that task τ_2 is schedulable when the second computation segment of task τ_1 and task τ_2 are released simultaneously. Consequently, the following inequality must hold[1]:

$$\frac{2\epsilon}{\alpha} + (T - 2 - 2\epsilon) + \frac{T/F}{\alpha} \leq T$$

[1]Task τ_1 can be considered as an equivalent sporadic non-suspending task $((1), F, F)$.

After reformulation, we have $\alpha \geq \frac{T/F+2\epsilon}{2+2\epsilon}$. Thus, $\alpha \rightarrow \infty$ as $T/F \rightarrow \infty$. $\square$

One can conclude that the current state-of-the-art tests for dynamic self-suspension models are not accurate enough to be seamlessly applied for the multi-segment self-suspension model. There still exists a large gap between the best-known self-suspension tests for the dynamic self-suspension model and what may be possible schedulability tests for the multi-segment one.