On Fixed-Priority Schedulability Analysis of Sporadic Tasks with Self-Suspension

Morteza Mohaqeqi, Pontus Ekberg, Wang Yi
Uppsala University, Sweden
{morteza.mohaqeqi,pontus.ekberg,wang.yi}@it.uu.se

ABSTRACT

We consider the schedulability analysis problem of a set of sporadic tasks which are subject to self-suspension, using a fixed-priority scheduler on a preemptive uniprocessor. We show that this problem is *co*NP-hard in the strong sense even in the simple case when only the lowest-priority task is self-suspending. Also, it is shown that the problem is weakly *co*NP-hard even if that self-suspending task has only a single suspension interval. In addition, we propose an efficient method for schedulability analysis of self-suspending tasks that are subject to interference from higher-priority tasks without self-suspension. The method works on the basis of an iterative approach which begins with an abstraction of the task set and improves the analysis results by refinement steps as needed. Our evaluation shows that this method significantly improves the scalability of the existing approaches.

CCS Concepts

•**Software and its engineering** → **Real-time schedulability;** •**Computer systems organization** → *Real-time systems;*

1. INTRODUCTION

Many real-time and embedded systems involve situations in which a task is suspended until receiving some result from a component which is not necessarily running on the same processor [1]. This scenario arises, for instance, when a task uses an external device, such as a digital signal processor (DSP), to offload some complex computations [2]. During such intervals, the suspended task, which is called *self-suspending*, is blocked until receiving the respective result, while other tasks can continue to be executed normally.

The switching behavior of self-suspending tasks implies a complicated behavior which causes that the existing methods for timing and schedulability analysis of ordinary real-time tasks cannot be easily applied to this task model. For instance, for ordinary (non-self-suspending) sporadic tasks and under fixed-priority scheduling, it is known that the *critical instant* of a task, namely the scheduling instant at which the task has its worst-case response time, happens when all the higher priority tasks are released synchronously with that one and the jobs are separated by the minimum allowed separation time [3]. As a result, by simply simulating the task set behavior for this release scenario, which is called the *synchronous release sequence*, one can perform schedulability analysis in pseudo-polynomial-time. It has been shown that the critical instant of a self-suspending task is not necessarily generated by a synchronous release sequence [4]. In particular, it is shown that the synchronous release pattern does not always lead to the worst-case response time (WCRT) of a self-suspending task even when the task contains only one suspension region and all other tasks are non-self-suspending. In [4], an algorithm with an exponential running time is presented for the problem. Meanwhile, the computational complexity of the schedulability problem is not established.

In this paper, we consider the problem of uniprocessor fixed-priority schedulability analysis of a task set containing one self-suspending task which has the lowest priority. We provide a lower bound for the computational complexity of the problem for the basic case where the self-suspending task consists of only one suspension interval. We show that the problem is (at least) weakly *co*NP-hard through a reduction from the PARTITION problem. We also show that when the self-suspending task is subject to multiple suspension intervals (while other tasks are simple sporadic tasks) the schedulability problem is *co*NP-hard in the strong sense. This is shown by a reduction from the 3-PARTITION problem.

In addition to these complexity results, we propose a method to improve the efficiency of schedulability analysis of the task set by avoiding unnecessary exploration of the problem state space. The method uses an *abstraction refinement* technique which begins with an over-approximation of the task set behavior providing an upper bound for the task response time. In each step, if the calculated upper bound is larger than the deadline, we refine the abstraction one level to obtain more accurate descriptions, and thus, more accurate value of the response time. This procedure is repeated until reaching a *concrete* (namely exact) release sequence which shows unschedulability of the task set or making sure that no possible behavior can lead to a deadline miss.

In the remainder of this section, the previous work related to the current study is reviewed. The system model and the schedulability problem considered in this paper are formally specified in Section 2. Then, the complexity results

RTNS '16, October 19-21, 2016, Brest, France
© 2016 ACM. ISBN 978-1-4503-4787-7/16/10...$15.00
DOI: http://dx.doi.org/10.1145/2997465.2997485

for the case of a single and multiple suspension intervals are presented in Section 3 and Section 4, respectively. Our abstraction refinement approach is presented in Section 5 and is evaluated in Section 6. A summary of the paper as well as a discussion on the related open problems is provided in Section 7.

1.1 Related Work

The problem of response-time analysis for a set of sporadic tasks, which are special case of self-suspending tasks, has been studied by Eisenbrand and Rothvoß [5]. While it is shown that the problem is NP-hard in the weak sense, they did not address the *schedulability* problem for this task model. In the realm of self-suspending tasks, the computational complexity of the feasibility problem for a set of multiple self-suspending tasks has been studied in [6]. In that work, it is shown that the problem is NP-hard in the strong sense. In comparison to that work, in this paper we focus on the fixed-priority schedulability problem and show that it is *co*NP-hard in the strong sense even for a more restricted case of a single self-suspending task.

Regarding timing analysis of self-suspending tasks under fixed-priority scheduling, Palencia and Harbour [12] suggest to model self-suspending tasks as a set of tasks with release jitters and offsets. In this way, they propose a method to compute an over-approximation of the tasks response time. In addition, an approximation method for schedulability analysis of self-suspending tasks has been proposed in [1]. It is shown that the considered fixed-priority scheduling algorithm has a processor speed-up factor of M^2 with $M = \max(2, p_{max})$, where p_{max} is the maximum number of computation segments (which are separated by suspension regions) of the self-suspending tasks. In contrast, our schedulability analysis approach presented in Section 5 treats an exact analysis of the problem.

The task model considered in this paper has been recently studied in [4]. It has been shown, in contrary to ordinary sporadic tasks, that synchronous arrival sequence does not necessarily generate a critical instant even when there is only one self-suspending task in the task set which has the lowest priority. Then, an algorithm with exponential running time is presented for calculating the response-time of a self-suspending task which is scheduled based on a fixed-priority scheduling policy under the interference of a set of normal non-self-suspending sporadic tasks. Meanwhile, the computational complexity of this problem was not studied. In addition, the method suffers from a state-space explosion problem which restricts its scalability. We use the method proposed in [4], which has been proposed for WCRT analysis, in an abstraction and refinement framework to avoid exploring unnecessary branches in the problem search space.

2. TASK MODEL AND PROBLEM DEFINITION

We assume a set of n tasks, denoted by $\{\tau_1, \ldots, \tau_n\}$, in which $\tau_1, \ldots, \tau_{n-1}$ are ordinary (i.e., non-self-suspending) sporadic tasks, while τ_n is a self-suspending one. Each task τ_i, for $1 \leq i \leq n$, is specified by a worst-case execution time (WCET) C_i and a minimum inter-release time (period) T_i. All the tasks are subject to *implicit* deadlines which means that the relative deadlines are equal to the periods. The task set is scheduled with a fixed priority preemptive scheduling policy assuming that τ_n is of the lowest priority.

The execution of the self-suspending task is divided into a set of p computation segments which are separated by $p-1$ suspension intervals. The j-th segment is denoted by $\tau_{n,j}$ and has a WCET of $C_{n,j}$. Also, the length of the j-th suspension interval is denoted by $S_{n,j}$. The first computation segment can be executed as soon as the respective job is released. In contrast, $\tau_{n,j}$, for $2 \leq j \leq p$, becomes eligible for execution (or is *released*) whenever the previous suspension interval, i.e., the $j-1$-th one, is finished. The problem is to determine schedulability of such a task set.

Problem 1 (Schedulability Problem). *Take a task set with the above description with a fixed-priority preemptive scheduler. The problem is to determine whether the task set is schedulable.*

It is supposed that schedulability analysis of the non-self-suspending tasks is accomplished using existing methods [8]. As a result, in the subsequent discussions, we restrict our focus to schedulability analysis of the self-suspending task.

A *critical instant* for the self-suspending task is defined as a release scenario of the higher priority tasks in which τ_n has its maximum response time. Here, we review a known result with regard to the critical instant of a self-suspending task. For this goal, we first define a notation, Sync_j. For a specified release sequence of jobs containing one job of τ_n, we denote by Sync_j the set of those (higher priority) tasks which have a job released synchronously with $\tau_{n,j}$.

Lemma 1 ([4]). *For any self-suspending task set with the above description, there exists at least one release scenario in which any task τ_i, for $1 \leq i \leq n-1$, belongs to at least one Sync_j and the response time of τ_n is maximized.*

3. COMPLEXITY RESULT FOR A BASIC CASE

We show that Problem 1 is (at least) *co*NP-hard in the weak sense even when the self-suspending task contains only one suspension interval. This result implies that the problem for more general cases is also at least weakly *co*NP-hard.

The proof is by a polynomial-time transformation of the well-known Partition problem. In the following, we first review this problem. Then, the transformation method and the hardness proof are presented.

Definition 1 (Partition Problem). *Given is a set of n items $A = \{a_1, \ldots, a_n\}$ and a size function $s : A \mapsto \mathbb{N}$ such that*

$$\sum_{a_i \in A} s(a_i) = 2S. \tag{1}$$

The problem is to determine whether A can be partitioned into two sets A_1 and A_2 such that

$$\sum_{a_i \in A_1} s(a_i) = \sum_{a_i \in A_2} s(a_i) = S. \tag{2}$$

A given instance of the problem is a positive one if such a partition exists, and is a negative one otherwise.

3.1 Transformation

In this section, we describe a method for transforming a given instance of the Partition problem to an instance of the schedulability problem, i.e., Problem 1, assuming that the self-suspending task is subject to only one suspension interval. It will be shown that the PARTITION problem instance is a positive one, namely the set can be partitioned into two sets which satisfy Eq. (2), if and only if the constructed task set is unschedulable.

Let $A = \{a_1, \ldots, a_n\}$ be the given set in the PARTITION problem. We construct a task set with $n+2$ tasks, denoted by $\tau = \{\tau_0, \tau_1, \ldots, \tau_n, \tau_{n+1}\}$, where τ_{n+1} is a self-suspending task. The WCET of each sporadic (non-self-suspending) task is determined as

$$C_i = \begin{cases} 1, & i = 0, \\ s(a_i), & 1 \leq i \leq n. \end{cases} \tag{3}$$

In addition, the minimum inter-release time of these tasks is specified as

$$T_i = \begin{cases} S+2, & i = 0, \\ 2(3S+7), & 1 \leq i \leq n. \end{cases} \tag{4}$$

For τ_{n+1}, we assume two computation segments and one suspension interval. Specifically, the parameters of τ_{n+1} are set as follows

$$\begin{aligned} C_{n+1,1} &= 2, \\ C_{n+1,2} &= 2, \\ S_{n+1,1} &= S, \\ T_{n+1} &= 3S+7, \end{aligned} \tag{5}$$

It is supposed that tasks are indexed according to their priority. As a result, τ_0 has the highest priority, while τ_{n+1} is of the lowest priority.

3.2 Hardness Proof

The goal is to show that the obtained task set is schedulable, if and only if the given instance of the PARTITION problem is negative. First, we note, according to the specified parameters for the tasks, that $\tau_0, \ldots, \tau_n$ are always schedulable. This holds for τ_0 since it is of the highest priority, and thus, its response time is always equal to its execution time which is less than the deadline. Also, to see that other tasks τ_i, for $1 \leq i \leq n$, are schedulable we consider their respective critical instant which is obtained by a synchronous release sequence (since all of them are normal sporadic tasks). According to the assigned period values, each task τ_i, for $1 \leq i \leq n$, observes no more than six instances of τ_0 before its deadline. Also, it will be interfered by one job from each of the higher priority tasks $\tau_1, \ldots, \tau_{i-1}$. Hence, the total interfering workload from the higher priority tasks plus its own execution time is at most $6 + \sum_{1 \leq j \leq i} s(a_j)$, which is upper bounded by $6 + 2S$. Then, since $6 + 2S$ is less than $2(7 + 3S) = T_i$, which is the task relative deadline, the task is schedulable. As a result, in the following, our focus is only on the schedulability of τ_{n+1}.

In order to investigate the schedulability of τ_{n+1} we first present two lemmas. The first one specifies the maximum interfering workload which can be experienced by the computation segments of τ_{n+1} from $\tau_1, \ldots, \tau_n$. The second one describes the respective interfering workload from τ_0.

LEMMA 2. *The maximum overall interference of $\tau_1, \ldots, \tau_n$ on one job of τ_{n+1} is no larger than $2S$.*

PROOF. Based on Lemma 1, only release sequences where each $\tau_1, \ldots, \tau_n$ is in (at least) one of Sync_1 or Sync_2 are considered. Regarding this, each instance of τ_{n+1} can observe at most one instance (i.e., job) of τ_i, for $1 \leq i \leq n$. This is because the relative deadline of τ_{n+1} is $3S+7$, and the minimum inter-release time of τ_i, for $1 \leq i \leq n$, has been set to $2(3S+7)$. As a result, the maximum interfering workload observed by the computation segments of τ_{n+1} from the higher priority tasks $\tau_1, \ldots, \tau_n$ can be computed as (using Eqs. (3) and (1))

$$\sum_{1 \leq i \leq n} C_i = 2S. \tag{6}$$

□

LEMMA 3. *Let W_0 denote the maximum workload from τ_0 that can interfere with one job of τ_{n+1}. The value of W_0 is determined according to the following criteria:*

- *W_0 is equal to 4 if the interfering workload from τ_i, for $1 \leq i \leq n$, calculated in Lemma 2, on the two computation segments of τ_{n+1} are equally dispatched, i.e., the interfering workload on any of the two segments is exactly equal to S.*
- *$W_0 = 3$, otherwise.*

In other words, $W_0 = 4$ if and only if

$$\sum_{\substack{1 \leq i \leq n \\ \tau_i \in Sync_1}} C_i = \sum_{\substack{1 \leq i \leq n \\ \tau_i \in Sync_2}} C_i = S. \tag{7}$$

PROOF. A schematic view of the first case is depicted in Fig. 1. In this case, as the sum of the execution time of tasks $\tau_i \in \mathrm{Sync}_1$ is equal to S, only one unit of time can be allocated to execute the first computation segment of τ_{n+1} before the second release of τ_0. As a result, completion of $\tau_{n+1,1}$ is postponed to the point after execution of the second job of τ_0. As a result, $\tau_{n+1,1}$ observes two instances of τ_0. A similar situation occurs for the second computation segment of τ_{n+1}. Consequently, in total, τ_{n+1} experiences four instances of τ_0 in the specified circumstances. Then, since $C_{n+1} = 1$, it is implied that $W_0 = 4$.

For the second case, it is observed that if the interfering load from $\tau_1, \ldots, \tau_n$ on one of the segments of τ_{n+1} is less than S, then that segment can be affected by at most one instance of τ_{n+1}. Besides, the other segment experiences at most two instances of τ_{n+1}. To see this, consider Fig. 2. This figure shows the extreme case of such scenario. It is seen that even in this case, where all τ_is (for $1 \leq i \leq n$) interfere with the first segment of τ_{n+1}, this segment still cannot observe more than two instances of τ_0 since

$$2 + 2S + 2 = 2S + 4 \leq 2T_0, \tag{8}$$

which means that the execution of the first computation segment of τ_{n+1} finishes before the third release of τ_0. From this, it follows that $W_0 = 3$. □

Now, we elaborate the duality between the scheduling problem of the real-time task set and the PARTITION problem.

LEMMA 4. *In the PARTITION problem, the given set of items can be partitioned into two sets with equal total size of elements if and only if the task set obtained from the transformation is unschedulable.*

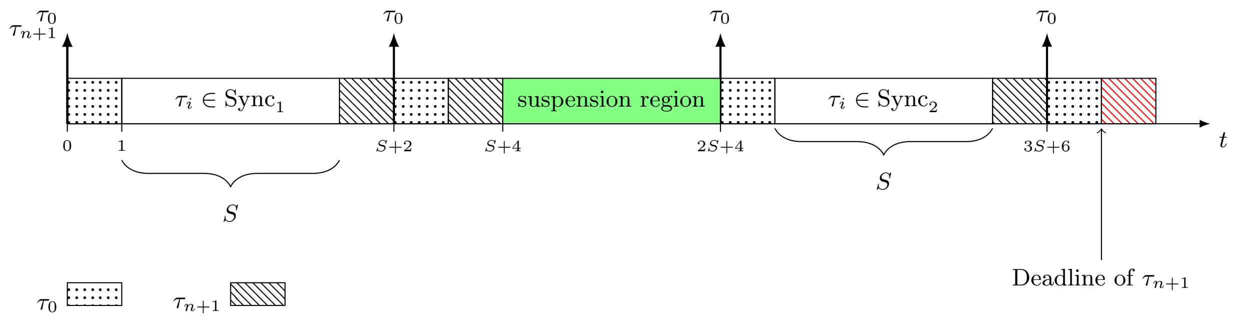

Figure 1: The situation in which $W_0 = 4$, which causes the self-suspending task (i.e., τ_{n+1}) to miss its deadline.

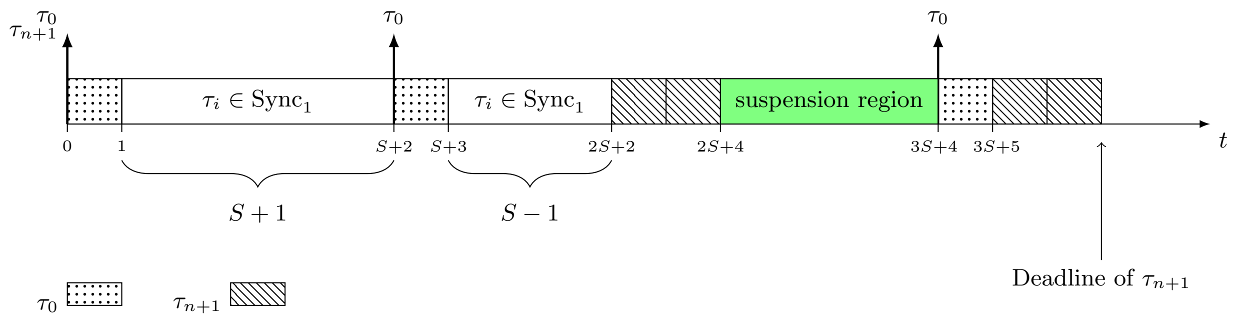

Figure 2: The workload from $\tau_1, \ldots, \tau_n$ on the two segments of τ_{n+1} is not equal. In such situation, the first segment of τ_{n+1} observes at most two instances of τ_0 even in the extreme case, as shown in the figure, which implies $W_1 = 3$.

PROOF. We first assume that in the PARTITION problem instance, the set A can be divided into two sets the sum of elements of which is equal to S. As a result, the set of tasks $\{\tau_1, \ldots, \tau_n\}$ can be divided into two subsets such that the sum of WCET of the tasks in each subset is equal to S. Consider a scenario in which the jobs of one subset are released simultaneously with the first segment of τ_{n+1}, while the jobs of the other subset are released synchronously with the second segment (this scenario is depicted in Fig. 1). According to Lemma 3, this leads to $W_0 = 4$. As a result, the total interference experienced by τ_{n+1} from higher priority tasks will be equal to

$$W_0 + \sum_{1 \le i \le n} C_i = 4 + 2S \qquad (9)$$

As the WCET of τ_{n+1} is assumed to be $2 + 2 = 4$, the total execution demand before the deadline of τ_{n+1} sums up to $2S + 8$. In addition, there is a suspension interval of length S in the middle. The execution demand plus this suspension interval length will be equal to $3S + 8$ which is larger than the relative deadline of τ_{n+1}, namely $3S + 7$. This means that the task set is unschedulable.

Now, we consider the other direction assuming that the set A cannot be partitioned into two sets with equal sum of elements. As a result, the sum of the interfering workload from $\tau_1, \ldots, \tau_n$ on one of the two segments of the self-suspending task is less than S. This leads to the second case in Lemma 3, which results in $W_0 = 3$. Consequently, the total execution requirement in the interval between a release of τ_{n+1} and its deadline is $2S + 3 + 4 = 2S + 7$, which means that τ_{n+1} meets its deadline. As the set of tasks $\{\tau_0, \ldots, \tau_n\}$ is already known to be schedulable, this result leads to the schedulability of the whole task set, which completes the proof. □

THEOREM 1. *The problem of deciding whether a set of sporadic tasks as described in Section 2 is fixed-priority schedulable is at least co*NP*-hard in the weak sense when the self-suspending task contains one suspension interval.*

PROOF. Based on Lemma 4, a PARTITION problem is a positive instance if and only if the corresponding problem of self-suspending task set schedulability is a negative one. As the PARTITION problem is (weakly) NP-hard and the transformation can be computed in polynomial time, the considered schedulability problem turns out to be (weakly) *co*NP-hard. □

4. COMPLEXITY FOR THE CASE OF MULTIPLE SUSPENSION INTERVALS

The goal of this section is to show that the schedulability problem of the task set specified in Section 2 is *co*NP-hard in the strong sense[1]. The proof is based on a polynomial time reduction from the 3-PARTITION problem, reviewed below.

DEFINITION 2 (3-PARTITION PROBLEM). *An instance of the* 3-PARTITION *problem consists of a set of* $3m$ *elements* $A = \{a_1, \ldots, a_{3m}\}$, *a bound* $B \in \mathbb{N}$, *and a size function* $s : A \mapsto \mathbb{N}$ *which satisfies*

- $\sum_{i=1}^{3m} s(a_i) = mB$, *and*
- $\forall a_i \in A : B/4 < s(a_i) < B/2$.

[1] Similar result is obtained in [10] in parallel to our work.

The problem is to partition the elements of A into m sets P_j, $1 \leq j \leq m$, such that for all $j \in \{1, \ldots, m\}$:

$$\sum_{a_i \in P_j} s(a_i) = B. \tag{10}$$

An instance of the problem is a positive one if such a partition exists, and a negative one otherwise.

The 3-Partition problem is known to be NP-hard in the strong sense [9]. In the following, we first provide a brief overview of the proof approach, and then, present the details.

4.1 Proof Overview

The idea is to construct a task set (containing one self-suspending task) for any given instance of 3-Partition such that the task set is unschedulable if and only if the instance of 3-Partition is a positive one. Towards this, consider an arbitrary instance of the 3-Partition problem with a set of $3m$ elements $A = \{a_1, \ldots, a_{3m}\}$. We associate to each element $a_i \in A$ a sporadic task, denoted by τ_i, for $1 \leq i \leq 3m$. The execution time of τ_i is specified as

$$C_i = s(a_i), \quad 1 \leq i \leq 3m \tag{11}$$

We also consider one additional ordinary sporadic task and one self-suspending task with m computation segments which are described shortly.

The periods of the tasks will be assigned such that the following two properties are satisfied:

- at most one job of each task τ_i can interfere with a single job of the self-suspending task,
- τ_i is schedulable for $1 \leq i \leq 3m$.

The first property above guarantees that each task τ_i can be contained by only one Sync_j (recall that Sync_j denotes the set of tasks released synchronously with the j-th segment of the self-suspending task). Assume that the sum of the execution time of the tasks in Sync_j is denoted by B_j, namely,

$$B_j = \sum_{\substack{1 \leq i \leq 3m \\ \tau_i \in \text{Sync}_j}} C_i, \quad \text{for } 1 \leq j \leq m \tag{12}$$

For the proof, we verify two claims:

1. The 3-Partition instance is a positive one if and only if there exists a release sequence of the tasks for which
$$B_j = B, \quad \text{for } 1 \leq j \leq m. \tag{13}$$
2. There exists a release sequence which satisfies (13) if and only if the task set is unschedulable.

In the following, we first present the details of our method for task set construction by determining timing parameters of the constructed tasks in Section 4.2. The validity of the first claim is then established in Section 4.3. The second claim is validated in Section 4.4.

4.2 Task Set Construction

As specified earlier, for each element a_i in the 3-Partition instance, we consider a task τ_i with a WCET of $C_i = s(a_i)$. In addition, we augment the task set with a specific task τ_0 with the WCET and period of

$$\begin{aligned} &C_0 = 1, \\ &T_0 = (m-1)B + 2. \end{aligned} \tag{14}$$

The intuition behind defining such a task is to *control* the workload released before and after the suspension region, as discussed more precisely in the next sections.

We also consider a self-suspending task, denoted as τ_{n+1}, with m computation segments and $m-1$ suspension intervals, with the following characteristics:

$$\begin{aligned} &C_{n+1,j} = (m-2)B + 2, && 1 \leq j \leq m \\ &S_{n+1,j} = (m-1)B, && 1 \leq j \leq m-1 \\ &T_{n+1} = (2m-1)T_0 + 1. \end{aligned}$$

Accordingly, the total execution demand and the total suspension time of any single job of τ_{n+1} are computed, respectively, as

$$\begin{aligned} C_{n+1} &= mC_{n+1,j} \\ &= m(m-2)B + 2m, \end{aligned} \tag{15}$$

and

$$\begin{aligned} S_{n+1} &= (m-1)S_{n+1,j} \\ &= (m-1)^2 B. \end{aligned} \tag{16}$$

In addition, the period of τ_i, for $1 \leq i \leq 3m$, is specified as

$$T_i = 2T_{n+1}, \tag{17}$$

which guarantees the following property.

Property 1. *According to the specified timing parameters, at most one job of each task τ_i can have an interference on a job of τ_{n+1}. In other words, each task τ_i, for $1 \leq i \leq 3m$, can exist in only one Sync_j.*

According to this observation, the worst-case overall interfering workload of the tasks τ_i, for $1 \leq i \leq 3m$, on τ_{n+1}, denoted by I_{3m}, is

$$I_{3m} = \sum_{1 \leq i \leq 3m} C_i = mB. \tag{18}$$

The highest priority is assigned to τ_0, while τ_{n+1} has the lowest priority. The priority of tasks τ_i for $1 \leq i \leq 3m$ is assigned to be lower than τ_0 and higher than τ_{n+1}. The relative priority of τ_i and τ_j for $1 \leq i, j \leq 3m$ with respect to each other has no influence on our results. A graphical illustration of the created task set and a sample schedule is seen in Fig. 3.

4.3 Restating the 3-Partition Problem

In this section, we show that a given 3-Partition instance is positive if and only if the corresponding task set can generate a release sequence which satisfies (13).

Lemma 5. *Any given* 3-Partition *problem instance is a positive one if and only if tasks τ_i can release a job sequence of which the corresponding sets Sync_j satisfy* (13).

Proof. Assume that the given instance of 3-Partition is a positive one, which means that there exists a partition of the items into m *bins* of size B. Let $\tau_i \in \text{Sync}_j$ if and only if a_i is associated with the j-th bin. Then, according to (11), the sum of WCETs of the tasks in each Sync_j for

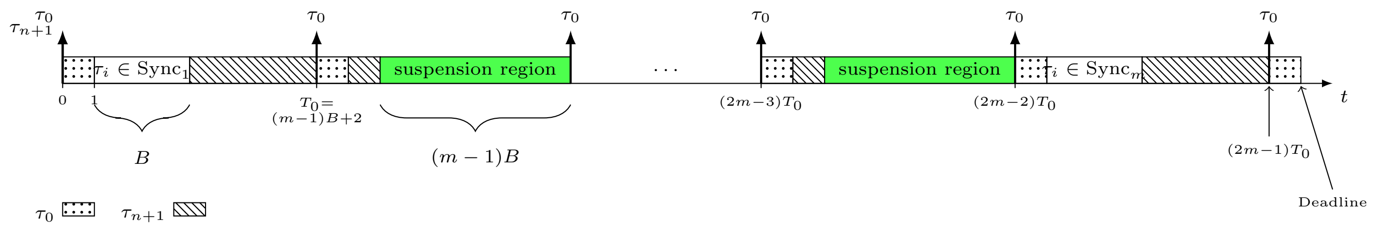

Figure 3: Illustration of the critical instant for the self-suspending task.

$1 \leq j \leq m$ is equal to B which completes the first part of the proof.

In the other direction, if (13) holds, then, according to Property 1, we have a partition of the numbers C_i, or equivalently a partition of $s(a_i)$s, which satisfies (10). By definition, this means that the 3-PARTITION problem is a positive instance, which completes the proof. □

4.4 Duality of the Two Problems

In this section, we show that Condition (13) holds if and only if the self-suspending task (and hence, the whole task set) is unschedulable. For this purpose, we first compute the maximum interference of τ_0 on a single computation segment of the self-suspending task.

LEMMA 6. *Consider an arbitrary computation segment of the self-suspending task, e.g., $\tau_{n+1,j}$. Then, for any release pattern of the higher priority tasks in which $\tau_{n+1,j}$ observes k instances of τ_0, there exists one release pattern in which*

- *all of the higher priority tasks except for τ_0 has the same release pattern,*
- *τ_0 is released synchronously with $\tau_{n+1,j}$, and*
- *$\tau_{n+1,j}$ observes at least k instances of τ_0.*

PROOF. The proof is by construction. Assume that $\tau_{n+1,j}$ is eligible for execution at time t. Then, consider a scenario in which $\tau_{n+1,j}$ has a response time of r and τ_0 is not released synchronously with $\tau_{n+1,j}$. In other words, the first job of τ_0 in the scheduling window of $\tau_{n+1,j}$ is released at time $t+\delta$ for some $\delta > 0$. As τ_0 has the highest priority and its WCET is one unit, the last job of τ_0 released before t has been finished before the release time of $\tau_{n+1,j}$. As a result, it does not have any interference on $\tau_{n+1,j}$. Now, consider a release pattern obtained by the supposed scenario with this modification that all instances of τ_0 which are released in interval $[t, t+r]$ in the initial scenario, now are released δ time units earlier. This means that in the new scenario, τ_0 is released synchronously with $\tau_{n+1,j}$. In this modified release pattern, the number of instances of τ_0 which are released in $[t, t+r]$ is not reduced. Also, the other (non-self-suspending) tasks have exactly the same release pattern. As a result, the observed workload by $\tau_{n+1,j}$ in an interval $[t, t_1]$ for any $t_1 \in [t, t+r]$ is not smaller than the one observed in the original scenario. Thus, $\tau_{n+1,j}$ has at least the same response time in this modified scenario and observes at least the same number of instances of τ_0. Therefore, the described scenario satisfies all the three conditions indicated by the lemma. □

LEMMA 7. *Let B' denote the total interference made by tasks τ_i, for $1 \leq i \leq 3m$, on one computational segment of τ_{n+1}, e.g, $\tau_{n+1,j}$. The maximum number of instances of τ_0 which is observed by $\tau_{n+1,j}$ is 1 if $B' < B$ and is 2 otherwise.*

PROOF. First, let $B' < B$. Accordingly, the total workload demand, assuming one instance of τ_0, will be equal to $C_0 + B' + C_{n+1,j} = 1 + B' + (m-2)B + 2$ which is not larger than $T_0 = (m-1)B + 2$. According to Lemma 6, the maximum number of instances observed by $\tau_{n+1,j}$ during this interval happens when τ_0 is released synchronously with $\tau_{n+1,j}$, which implies that the second instance of τ_0 is released after the completion time of the segment. As a result, the segment observes only one instance of τ_0.

Now, we consider the other case in which $B' \geq B$. First, it is noted that when $B' = B$, according to the above discussion, $\tau_{n+1,j}$ can execute for $(m-2)B + 1$ time units before the release of the second instance of τ_0. The remaining execution demand, however, is executed immediately after the execution of that second instance. As a result, the segment observes exactly two instances of τ_0. On the other side, the maximum value of B' is mB. As a result, with a similar argument used for the case of $B' < B$, it is seen that the total execution demand before the third release of τ_0 is at most $2C_0 + mB + C_{n+1,j}$ which equates to $2 + mB + (m-2)B + 2 = 4 + 2(m-1)B = 2T_0$. Hence, the maximum number of instances of τ_0 seen by $\tau_{n+1,j}$ is 2. □

Now, we elaborate the relationship between the satisfaction of Eq. (13) and unschedulability of the self-suspending task.

LEMMA 8. *Consider a set of tasks obtained by the described reduction method. There exists a release sequence generated by the task set which satisfies Eq.* (13)*, if and only if the self-suspending task is unschedulable.*

PROOF. For the proof, we first show that if the considered condition holds, then the self-suspending task is unschedulable. Next, we show that if the tasks cannot release a sequence of jobs which satisfy (13), then the self-suspending task meets its deadline.

$\Longrightarrow$: Figure 3 shows a situation in which the interference of the higher priority tasks τ_i, for $1 \leq i \leq 3m$, on any segment of τ_{n+1} is exactly equal to B and τ_0 is released synchronously with all computation segments of τ_0. Under such circumstances, according to Lemma 7, the interference of τ_0 on each segment $\tau_{n+1,j}$, for $1 \leq j \leq m$, is equal to 2. As a result, the total response time of τ_{n+1} can be calculated as (I_{τ_0} denotes the total interference of τ_0 on τ_{n+1})

$$\begin{aligned} R(\tau_{n+1}) &= C_{n+1} + S_{n+1} + I_{\tau_0} + I_{3m} \\ &= m(m-2)B + 2m + (m-1)^2 B + 2m + mB \\ &= m(m-1)B + 2m + (m-1)^2 B + 2m \\ &= (2m-1)(m-1)B + 4m \\ &= (2m-1)T_0 + 2 \\ &> T_{n+1} \end{aligned} \quad (19)$$

which means that τ_{n+1} misses its deadline. (recall that we assume implicit deadlines.) As a result, the task set is unschedulable under the assumed condition.

$\Longleftarrow$: For proving the other direction of the lemma, we show that for any release sequence where $B_j \neq B$ for some j, the self-suspending task meets its deadline. We first notice that if $B_j \neq B$, then there exists $k \in \{1, \ldots, m\}$ for which $B_k < B$. Based on Lemma 7, this implies that $\tau_{n+1,k}$ experiences at most one instance of τ_0. Besides, according to the same lemma, the maximum interference of τ_0 on the other computation segments of τ_{n+1} is at most 2. As a result, the total interference of τ_0 on τ_{n+1} in this situation is not larger than $1 + 2(m-1)$, that is $I_{\tau_0} \leq 2m - 1$. Now we calculate the WCRT of τ_{n+1} as

$$\begin{aligned} R(\tau_{n+1}) &= C_{n+1} + S_{n+1} + I_{\tau_0} + I_{3m} \\ &\leq C_{n+1} + S_{n+1} + 2m - 1 + I_{3m}. \end{aligned} \tag{20}$$

Regarding (15), (16), and (18), we have

$$C_{n+1} + S_{n+1} + I_{3m} = B\left[(m-1)(2m-1)\right] + 2m.$$

Using this relation in (20), we get

$$\begin{aligned} R(\tau_{n+1}) &\leq B\left[(m-1)(2m-1)\right] + 4m - 1 \\ &= (2m-1)\left[B(m-1) + 2\right] + 1 \\ &= (2m-1)T_0 + 1, \end{aligned} \tag{21}$$

which means $R(\tau_{n+1}) \leq T_{n+1}$. Hence, the self-suspending task meets its deadline and the proof is completed. □

5. SCHEDULABILITY ANALYSIS USING ABSTRACTION AND REFINEMENT

In this section, we deal with the problem of exact schedulability analysis of the task set described in Section 2 using an abstraction and refinement technique. Schedulability analysis of the non-self-suspending tasks can be done using the well-known response-time analysis methods, such as the one proposed in [8], which run in pseudo-polynomial time worst-case complexity. Hence, we only focus on schedulability analysis of the self-suspending task assuming that the set of higher priority tasks $\{\tau_1, \ldots, \tau_{n-1}\}$ is schedulable.

The idea is based on constructing an abstraction of the task set behavior by which an over-approximation (i.e., an upper bound) for the actual WCRT of the self-suspending task can be calculated. The procedure begins with the highest level of abstraction, which is an over-approximation of *all* possible behaviors. If this abstraction reveals schedulability of the task set, then we can safely conclude that the task set is schedulable. Otherwise, we proceed with one step of refinement by refining the abstraction made for one of the tasks, which reveals a set of less abstract behaviors. Then, we need to check schedulability for all of the refined behaviors to make sure that the task set is schedulable. During this procedure, whenever we reach an abstract behavior under which the self-suspending task is schedulable, then we do not need to perform more refinements for that specific abstraction. On the other hand, if an abstraction implies unschedulability, we need to refine one step more by again selecting a task to be refined. This procedure of refinement and schedulability test is repeated for all of the created branches. If during this exploration we reach the lowest level of abstraction for which the task is unschedulable, then we have reached an actual counter example in which the task misses its deadline. As a result, the procedure is terminated. Meanwhile, if we never reach a concrete scenario under which the task misses its deadline, we can infer schedulability of the task.

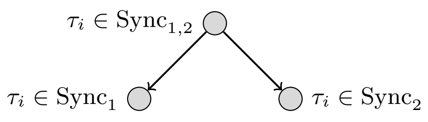
$\tau_i \in \text{Sync}_{1,2}$
$\tau_i \in \text{Sync}_1$
$\tau_i \in \text{Sync}_2$

Figure 4: Refinement of a single task.

In the following, we elaborate on the details of this approach. For simplicity, we first describe our method for a basic case of a self-suspending task with one suspension region in Section 5.1. Then, in Section 5.2, we specify how the method can be used for the case of multiple suspension intervals. Before that, we point out a lemma used later.

Lemma 9. *Take a set of sporadic non-self-suspending tasks τ and also a task set τ' containing the same tasks with this difference that some jobs released by τ' are permitted to violate the minimum inter-release time requirement specified by τ. Then, the WCRT of a self-suspending task under τ' as the set of interfering tasks is larger than or equal to the one obtained under τ.*

Proof. The proof is based on the observation that any release sequence generated by τ can also be generated by τ'. As a result, any critical instant obtained under τ can be achieved by τ', revealing the same response time for the self-suspending task. Therefore, the WCRT of this task when interfered with τ' is at least as large as the one obtained with τ. □

5.1 Analysis Method for the Case of One Suspension Interval

In this section, we assume that the self-suspending task, denoted as τ_n, has one suspension interval and two computation segments $\tau_{n,1}$ and $\tau_{n,2}$. According to Lemma 1, there exists a critical instant for τ_n in which, for any higher priority task τ_i, at least one the following conditions hold: (i) $\tau_i \in \text{Sync}_1$; (ii) $\tau_i \in \text{Sync}_2$. Now, consider a scenario in which at the moment of $\tau_{n,2}$'s release time, we can ignore the minimum inter-release separation requirement of a higher priority task τ_i. As a result, τ_i can be released synchronously with both $\tau_{n,1}$ and $\tau_{n,2}$. We denote this situation by $\tau_i \in \text{Sync}_{1,2}$. According to Lemma 9, the actual interfering workload generated by τ_i is never greater than the workload which can be generated in this imaginary scenario. As a consequence, this release scenario provides an upper bound on the maximum interference of τ_i on τ_n, assuming a fixed behavior for the other tasks. We refer to this scenario as an abstraction of the actual behavior of τ_i.

Referring to the described abstraction method, an abstract scenario for τ_i, which is denoted by $\tau_i \in \text{Sync}_{1,2}$, can be refined into two actual (concrete) scenarios: $\tau_i \in \text{Sync}_1$ and $\tau_i \in \text{Sync}_2$. Figure 4 illustrates the abstraction refinement of a task τ_i based on this method.

Our abstraction refinement approach for schedulability analysis begins with the most abstract level where all the higher priority tasks are in $\text{Sync}_{1,2}$. In each step of refinement, one task is selected and is refined to provide a more accurate description of the task set behavior. The pseudocode of this approach is shown in Algorithm 1. As seen,

Algorithm 1 Schedulability test using abstraction refinement

```
 1: procedure SchedTest($\tau$)
 2:     $A[i] \leftarrow \mathcal{S}_{1,2}, \forall i \in \{1, \dots, n-1\}$    ▷ Set $\tau_i \in \text{Sync}_{1,2}$
 3:     $store.add(A)$
 4:     $N^{up}[i] \leftarrow \left\lceil \frac{WCRT(\tau_{n,1})}{T_i} \right\rceil, \forall i \in \{1, \dots, n-1\}$
 5:     while $store$ is not empty do
 6:         $A \leftarrow store.pop()$
 7:         if $\text{RT}(\tau_n, A, N^{up}) > D$ then
 8:             if $A$ is abstract then    ▷ Do a refinement
 9:                 $A_1 \leftarrow A$
10:                 $A_2 \leftarrow A$
11:                 $i \leftarrow \arg\max_i \{C_i/T_i \mid A[i] = \mathcal{S}_{1,2}\}$
12:                 $A_1[i] \leftarrow \mathcal{S}_1$    ▷ $\tau_i \in \text{Sync}_1$
13:                 $A_2[i] \leftarrow \mathcal{S}_2$    ▷ $\tau_i \in \text{Sync}_2$
14:                 $store.push(A_1)$
15:                 $store.push(A_2)$
16:             else
17:                 return False
18:             end if
19:         end if
20:     end while
21:     return True
22: end procedure
```

Lines 2 and 3 perform an initialization by assigning each task the highest abstraction level ($A[i]$ is used to store the abstraction level of τ_i) and storing this assignment in a stack. Line 4 computes an upper bound on the maximum number of jobs of each higher priority task which can interfere with $\tau_{n,1}$. ($WCRT(\tau_{n,1})$ denotes the WCRT of $\tau_{n,1}$ which can be obtained by conventional WCRT analysis methods for sporadic tasks). This upper bound is given as an input to the response-time analysis algorithm explained later. Line 7 compares the WCRT of the self-suspending task with its relative deadline, denoted as D. If the WCRT is larger than the deadline and A indicates at least one abstracted task, we need to perform a step of refinement. Lines 9 to 15 denote the refinement step in which a selected task, e.g. τ_i, is refined. Different criteria can be used for selecting a task to be refined. We use the task utilization as a measure of determining which task should be picked up. The intuition is that the abstraction of a task with a higher utilization is expected to provide a more inaccurate approximation of the actual task behavior. As a result, refining a task with higher utilization probably helps the method to achieve the ultimate result in earlier steps.

A schematic representation of the refinement process is demonstrated in Fig. 5. The selected task for refinement in each step is colored in red. Further, the newly assigned abstraction level to each task is seen in blue.

For schedulability analysis on the basis of the described approach, we need a method to compute the WCRT of the self-suspending task for a given set of higher priority tasks, some of which are abstracted. For this, we adopt the response-time analysis algorithm proposed in [4]. Algorithm 2 (invoked in Line 7 of Algorithm 1) shows the pseudocode of the algorithm adapted to be used in our framework. As seen, the algorithm first computes the maximum response time of $\tau_{n,1}$ (Line 7) based on a given maximum number of permitted jobs for each higher priority task (in-

$$\begin{tikzcd}
 & & A : [\mathcal{S}_{1,2}, \mathcal{S}_{1,2}, \dots, \mathcal{S}_{1,2}] \arrow[dl] \arrow[dr] & \\
 & {[\mathcal{S}_{1,2}, \mathcal{S}_1, \dots, \mathcal{S}_{1,2}]} \arrow[dl] \arrow[dr] & & {[\mathcal{S}_{1,2}, \mathcal{S}_2, \dots, \mathcal{S}_{1,2}]} \\
{[\mathcal{S}_1, \mathcal{S}_1, \dots, \mathcal{S}_{1,2}]} & & {[\mathcal{S}_2, \mathcal{S}_1, \dots, \mathcal{S}_{1,2}]} & \vdots \\
\vdots & & \vdots &
\end{tikzcd}$$

Figure 5: A part of the hierarchy of abstraction refinement.

dicated by array N^{up}) such that the release restrictions, which are specified by the input argument *Syn*, are preserved. Then, in order to respect the inter-release time requirements, an offset is calculated to determine the release time of the first job of each task in the scheduling window of $\tau_{n,2}$ (Lines 18 to 24). In order to adjust that method to our case, the first job in the mentioned scheduling window of the tasks in $\text{Sync}_{1,2}$ do not need to respect the minimum-inter release time restriction. As a result, the offset of those tasks is set to be zero (Line 20). The remaining part of the algorithm remains unchanged.

5.2 Extension to Multiple Suspension Regions

In this section, we point out how our abstraction refinement approach can be extended to the case of a self-suspending task with multiple suspension regions. The idea is based on the observation that the result of Lemma 9 is valid for the general case of multiple suspension intervals, as specified in the following.

Assume that a higher priority task is allowed to be released synchronously with all of the computation segments of the self-suspending task irrespective of its specified minimum inter-release separation time. When computing the WCRT of the self-suspending task under this assumption, we will obtain an upper bound on its actual WCRT. As a result, a specification of the higher priority task which allows this situation can be considered as an abstraction of that task which can be used in our abstraction refinement framework. In this case, we will denote an abstraction by the notion of $\tau_i \in \text{Sync}_{1,\dots,p}$. Then, each refinement step consists of replacing the abstraction of a task $\tau_i \in \text{Sync}_{1,\dots,p}$ with its p distinct possible choices, i.e., $\tau_i \in \text{Sync}_1, \dots, \tau_i \in \text{Sync}_p$.

5.3 Method Run-Time Complexity

Regarding the computational complexity of the presented method, as argued in [4], there are two sources of exponential complexity. One is related to the exponentially many number of ways that one can partition the sporadic tasks into sets Sync_j. The second one is rooted in different number of jobs from higher priority tasks interfering with $\tau_{n,1}$ for which the response time must be calculated. (explored by the last for loop in Algorithm 2.) Our algorithm also suffers from the same complexity in the worst case. Despite this, as shown in the next section, the method exhibits an acceptable efficiency for moderate-size task sets.

6. EVALUATION

In order to evaluate the proposed method for schedulability analysis we have performed three sets of experiments.

Algorithm 2 Response time analysis (adopted from [4], modified to be used in our schedulability analysis method)

```
1:  procedure RT($\tau$, $Syn$, $N^{up}$)
2:      if already invoked for the given $Syn$ and $N^{up}$ then
3:          return 0
4:      end if
5:      $N \leftarrow N^{up}$
6:      $R^b_{ss,1} \leftarrow 0$
7:      $R_{ss,1} \leftarrow C_{n,1} + \sum_{i=1}^{n-1} N^{up}[i]C_i$
8:      while $R^b_{ss,1} \neq R_{ss,1}$ do
9:          $R^b_{ss,1} \leftarrow R_{ss,1}$
10:         for $i \leftarrow 1$ to $n-1$ do
11:             if $Syn[i] = \mathcal{S}_2$ and $N[i] > \frac{R_{ss,1}+S_{n,1}}{T_i}$ then
12:                 $N[i] \leftarrow N[i] - 1$
13:             end if
14:         end for
15:         $R_{ss,1} \leftarrow C_{n,1} + \sum_{i=1}^{n-1} \min(N[i], \left\lceil \frac{R_{ss,1}}{T_i} \right\rceil)C_i$
16:         $N[i] \leftarrow \min\left(N[i], \left\lceil \frac{R_{ss,1}}{T_i} \right\rceil\right), \forall i \in \{1, \ldots, n-1\}$
17:     end while
18:     for $i \leftarrow 1$ to $n-1$ do
19:         if $Syn = \mathcal{S}_{1,2}$ then
20:             $O[i] \leftarrow 0$
21:         else
22:             $O[i] \leftarrow \max\left(0, N[i]T_i - R_{ss,1} - S_{n,1}\right)$
23:         end if
24:     end for
25:     $R_{ss,2} \leftarrow C_{n,2} + \sum_{i=1}^{n-1} \left\lceil \frac{R_{ss,2}-O[i]}{T_i} \right\rceil C_i$     ▷ Fixed point
26:     $R_{ss} \leftarrow R_{ss,1} + S_{n,1} + R_{ss,2}$
27:     if $R_{ss} < UB_{ss}$ and $R_{ss,2} < UB_{ss,2}$ then
28:         for $i \leftarrow 1$ to $n-1$ do
29:             if $N[i] > 0$ then
30:                 $N' \leftarrow N$
31:                 $N'[i] \leftarrow N[i] - 1$
32:                 $R_{ss} \leftarrow \max(R_{ss}, \mathrm{RT}(\tau, Syn, N'))$
33:             end if
34:         end for
35:     end if
36:     return $R_{ss}$
37: end procedure
```

For generating a set of random utilization values we used the UUniFast algorithm [11]. Further, the period of each task is chosen from $[10, 200]$ with a uniform distribution. The execution time of each task is then calculated based on the corresponding value of its utilization and period. In each task set, a self-suspending task was considered with one suspension interval. Again, we use the UUniFast algorithm for partitioning the calculated execution time for the task into three parts to be assigned to the two computation segments and the single suspension interval. In our experiments, we have considered only those task sets in which the $n-1$ non-self-suspending tasks are schedulable. Each data point in the reported results is obtained by averaging the results for 500 random experiments.

We have compared the results of our method with those obtained by a method, called *Exhaustive*, which searches the concrete combinations without any abstraction. By a concrete combination, we mean any specific partition of the non-self-suspending tasks into two sets of Sync_1 and Sync_2. The Exhaustive method begins with an arbitrary concrete

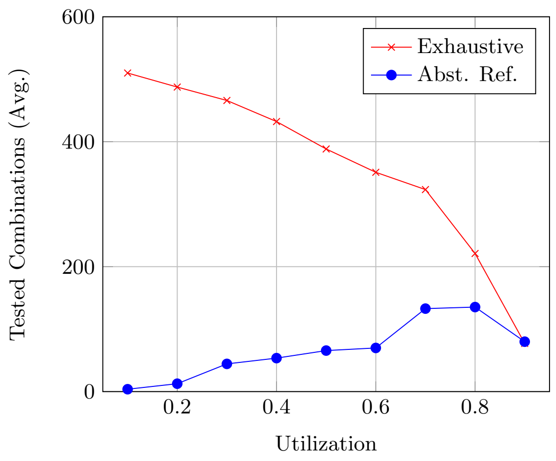

Figure 6: Number of tested combinations for $n = 10$.

case and proceeds until the first counter example showing a deadline miss. Therefore, for a schedulable task set, it will check all the possible concrete cases. It is worth noting that, for a task set of size n, there are 2^{n-1} such concrete cases.

Fig. 6 depicts the average number of tested combinations for task sets of size 10. As seen, the Exhaustive method does not perform well in low utilizations. This is because in this situation, a task set is schedulable with a high probability, the situation in which the method needs to check all possible combinations to make sure that the task set is schedulable. Also, if a task set is unschedulable, it is unlikely to find a counter example in early steps as there are probably not many combinations which lead to a deadline miss.

In contrast, the proposed abstraction refinement method works efficiently in low utilizations. This is because, in low utilizations, the WCRT of the self-suspending task is expected to be far from the relative deadline, introducing a large amount of slack for the task. Hence, even an inaccurate over-approximation of the tasks, which is related to higher abstraction levels, can reveal the schedulability of the task set. This lets the method skip many combinations.

Further, it is seen that the method exhibits a relatively small number of checked combinations in very high utilizations. The reason is that in such circumstances, it turns out to be *easy* to find a counter example. More precisely, in this case, it is highly probable to find a concrete counter example in which the task is unschedulable in early steps of the procedure. As a result, the method does not need to explore a large number of combinations.

In order to study the scalability of the proposed method with respect to the number of tasks, we have reported the average running time of the methods in Fig. 7. As seen, in general, the Exhaustive method is more than one order of magnitude slower than the other method. This result indicates that, while the abstraction refinement method can be as slow as the other method in the worst-case, it is noticeably faster in most cases.

Also, in order to investigate the scalability of the method in different utilizations, we have performed an experiment with changing the utilization for different number of tasks. As shown in Fig. 8, by increasing the number of tasks, the phase change behavior of the method (which occurs around utilization 0.8) becomes sharper. One reason for this can be

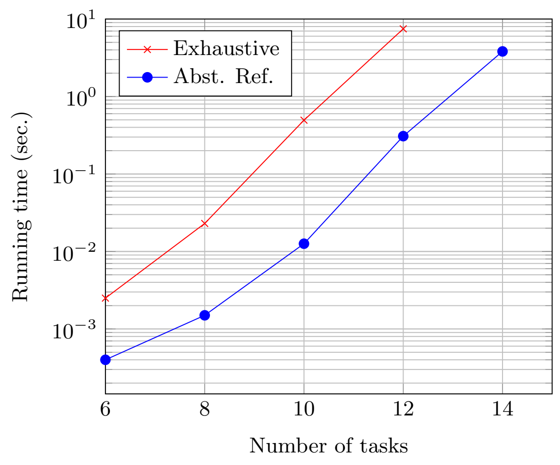

Figure 7: Running time of the two methods when number of tasks ranges from $n = 6$ to $n = 14$.

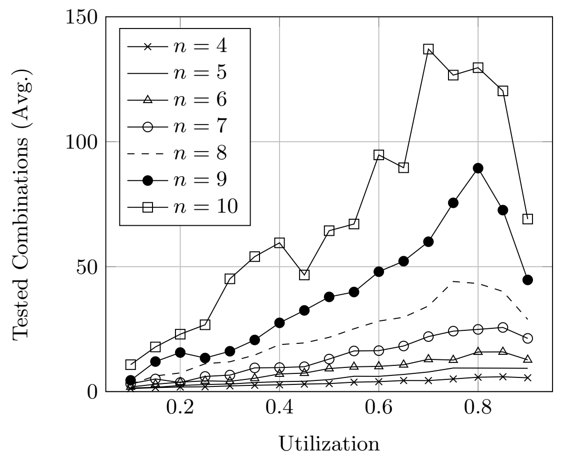

Figure 8: Tested combinations for different number of tasks.

due to the rapidly increased total number of combinations for higher number of tasks. On the other hand, for relatively low or very high utilizations, still there is a good chance for the abstraction refinement method to achieve an ultimate result and terminate in early steps.

7. CONCLUSION

In this paper, we considered a set of sporadic tasks containing a self-suspending task which has the lowest priority. We studied the problem of uniprocessor schedulability analysis of such a task set under the preemptive fixed-priority scheduling policy. We showed that the problem is *co*NP-hard in the weak sense when the self-suspending task contains only one suspension interval and *co*NP-hard in the strong sense if the task is subject to multiple suspension intervals. Our results provide a lower bound on the computational complexity of these problems. Therefore, the obtained results are valid for more general cases including the case in which the self-suspending task is not necessarily the lowest priority one, or task systems with more than one self-suspending task. It is worth mentioning that, for the case of a task set with a single self-suspending task with multiple suspension regions and with the lowest priority, an algorithm with exponential running time has been previously proposed [4]. Consequently, there exists also an upper bound for the problem complexity. Meanwhile, the exact computational complexity of the problem still is not known.

There are a number of problems closely related to this work which are still open. First, whether the obtained bound for the case of a single suspension region is tight is a related question. The same question exists when there are non-self-suspending tasks with priority lower than the self-suspending one.

Also, regarding the abstraction refinement approach, one can think of its extension to more general cases. It is intuitively expected that if there exists some results with respect to the critical instant of the tasks like the one expressed in Lemma 1, then a similar approach can be applied.

8. REFERENCES

[1] 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), pp. 1078–1083, 2016.

[2] 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), pp. 246–257, 2013.

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

[4] G. Nelissen, J. Fonseca, G. Raravi and V. Nelis, *Timing analysis of fixed priority self-suspending sporadic tasks.* In: Euromicro Conference on Real-Time Systems (ECRTS), pp. 80–89, 2015.

[5] F. Eisenbrand and T. Rothvoß, *Static-priority real-time scheduling: response time computation is NP-hard.* In: Real-Time Systems Symposium (RTSS), pp. 397–406, 2008.

[6] 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), pp. 47–56, 2004.

[7] M. Stigge and W. Yi, *Hardness results for static priority real-time scheduling.* In: Euromicro Conference on Real-Time Systems (ECRTS), pp. 189–198, 2012.

[8] R. I. Davis, A. Zabos and A. Burns, *Efficient exact schedulability tests for fixed priority real-time systems.* In: IEEE Transactions on Computers, vol. 57, no. 9, pp. 1261–1276, 2008.

[9] M. R. Garey and D. S. Johnson, *Computers and intractability; A guide to the theory of NP-completeness.* W. H. Freeman & Co., 1990.

[10] J.-J. Chen, *Computational complexity and speedup factors analyses for self-suspending tasks.* In: Real-Time Systems Symposium (RTSS), 2016. To appear.

[11] E. Bini and G. Buttazzo, *Measuring the performance of schedulability tests.* In: Real-Time Systems, vol. 30, no. 1, pp. 129–154, 2004.

[12] J. C. Palencia and M. G. Harbour, *Schedulability analysis for tasks with static and dynamic offsets.* In: Real-Time Systems Symposium (RTSS), pp. 26–37, 1998.