

# A Unifying Response Time Analysis Framework for Dynamic Self-Suspending Tasks

Jian-Jia Chen\*, Geoffrey Nelissen<sup>§</sup>, Wen-Hung Huang\*

\* TU Dortmund University, Germany

<sup>§</sup> CISTER/INESC-TEC, ISEP, Polytechnic Institute of Porto, Portugal

Emails: {jian-jia.chen, wen-hung.huang}@tu-dortmund.de, grrpn@isep.ipp.pt

**Abstract**—For real-time embedded systems, self-suspending behaviour can cause substantial performance/schedulability degradation. In this paper, we focus on preemptive fixed-priority scheduling for the dynamic self-suspension task model on a uniprocessor platform. In this model, a job of a sporadic task can suspend itself dynamically under a specified upper bound on the self-suspension across all of its suspension phases. To quantify the higher-priority interferences, the results in the literature for this task model can be classified into three classes (i) suspension as computation, (ii) suspension-induced release jitter, and (iii) suspension as blocking. However, the suspension-induced release jitter has to be carefully specified, and the concept to model suspension as blocking was never proven to be correct in the past. This paper presents a unifying response time analysis framework for the dynamic self-suspending task model. We provide a rigorous proof and show that all the three classes mentioned above are analytically dominated by this unifying framework. Therefore, all of them are in fact correct, and are inferior to the proposed response time analysis in this paper. The evaluation results show that the analysis framework can have a huge improvement (an increase of up to 50% of the number of task sets deemed schedulable) over these state-of-the-art analyses.

## I. INTRODUCTION

The periodic/sporadic task model has been recognized as the basic model for real-time systems with recurring executions. A sporadic real-time task  $\tau_i$  is characterized by its *worst-case execution time*  $C_i$ , its *minimum inter-arrival time*  $T_i$  and its *relative deadline*  $D_i$ . A sporadic task defines an infinite sequence of task instances, also called *jobs*, that arrive with the minimum inter-arrival time constraint. When a job of task  $\tau_i$  arrives at time  $t$ , the job should finish no later than its *absolute deadline*  $t + D_i$ , and the next job of task  $\tau_i$  can only be released no earlier than  $t + T_i$ . For the periodic task model, the next job is released at time  $t + T_i$ , in which  $T_i$  is also referred to as the *period* of task  $\tau_i$ .

The seminal work by Liu and Layland [16] considered the scheduling of periodic tasks and presented the schedulability analyses based on utilization bounds to verify whether the deadlines are met or not. For over decades, researchers in real-time systems have devoted themselves to effective design and efficient analyses of different recurrent task models to ensure that tasks can meet their specified deadlines. In most of these studies, *a task usually does not suspend itself*. That is, after a job is released, the job is either executed or stays in the ready queue, but it is not moved to the suspension state. Such an assumption is valid only under the following conditions: (1) the latency of the memory accesses and I/O

peripherals is considered to be part of the worst-case execution time of a job, (2) there is no external device for accelerating the computation, and (3) there is no synchronization between different tasks on different processors in a multiprocessor or distributed computing platform.

If a job can suspend itself before it finishes its computation, self-suspension behaviour has to be considered. Due to the interaction with other system components and synchronization, self-suspension behaviour has become more visible in designing real-time embedded systems. Typically, the resulting suspension delays range from a few microseconds (e.g., a write operation on a flash drive [11]) to a few hundreds of milliseconds (e.g., offloading computation to GPUs [12], [18]).

There are two typical models for self-suspending sporadic task systems: 1) the dynamic self-suspension task model, and 2) the segmented self-suspension task model. In the *dynamic* self-suspension task model, in addition the worst-case execution time  $C_i$  of sporadic task  $\tau_i$ , we have also the worst-case self-suspension time  $S_i$  of task  $\tau_i$ . In the *segmented* self-suspension task model, the execution behaviour of a job of task  $\tau_i$  is specified by interleaved computation segments and self-suspension intervals. From the system designer's perspective, the dynamic self-suspension model provides a simple specification by ignoring the juncture of I/O access, computation offloading, or synchronization. However, if the suspending behaviour can be characterized by using a segmented pattern, the segmented self-suspension task model can be more appropriate.

In this paper, we focus on preemptive fixed-priority scheduling for the dynamic self-suspension task model on a uniprocessor platform. To verify the schedulability of a given task set, this problem has been specifically studied in [1], [2], [9], [14], [19]. The recent report by Chen et al. and the report by Bletsas et al. [3] have shown that the analysis by introducing the suspension time of a higher-priority task as its arrival jitter in [1], [2], [14], [19] is unsafe. This misconception was unfortunately adopted in [4], [5], [8], [13], [15], [21]–[23] to analyze the worst-case response time for partitioned multiprocessor real-time locking protocols.

Moreover, one concept to consider suspension-time as blocking time was used by Jane W. S. Liu in her book titled "Real-Time Systems" [17, Pages 164-165], and was also implicitly used by Rajkumar, Sha, and Lehoczy [20, Page 267] for analyzing the self-suspending behaviour due to synchronization protocols in multiprocessor systems. However, there is no proof in [17], [20] to support the correctness of the

provided schedulability tests.

**Contributions.** In this paper, we propose the following contributions:

- We provide a new response analysis framework for dynamic self-suspending sporadic real-time tasks executing on a uniprocessor platform. The key observation in that analysis is that the *interference from higher-priority self-suspending tasks can be arbitrarily modelled as jitter or carry-in terms*.
- We prove that the new analysis analytically dominates all the existing results in the state-of-the-art, excluding the flawed ones.
- We prove the correctness of the analysis initially proposed in [17, p. 164-165] and [20, p. 267], which were never proven correct in the state-of-the-art<sup>1</sup>.
- The evaluation results presented in Section VIII show the huge improvement (an increase of up to 50% of the number of task sets that are deemed schedulable) over the state-of-the-art.

## II. TASK MODEL

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

Each task  $\tau_i$  is characterized by the tuple  $(C_i, S_i, D_i, T_i)$ , where  $T_i$  is the period (or minimum inter-arrival time) of  $\tau_i$  and  $D_i$  is its relative deadline.  $T_i$  specifies the minimum time between two consecutive job releases of  $\tau_i$ , while  $D_i$  defines the maximum amount of time a job can take to complete its execution after its release. It results that for each job  $\tau_{i,j}$ ,  $d_{i,j} = r_{i,j} + D_i$  and  $r_{i,j+1} \geq r_{i,j} + T_i$ . In this paper, we focus on constrained-deadline tasks, for which  $D_i \leq T_i$ . The utilization of a task  $\tau_i$  is defined as  $U_i = C_i/T_i$ .

The worst-case response time  $R_i$  of a task  $\tau_i$  is the maximum response time among all its jobs. A schedulability test for a task  $\tau_k$  is therefore to verify whether its worst-case response time is no more than its associated relative deadline  $D_k$ .

In this paper, we only consider *preemptive fixed-priority scheduling running on a single processor platform*, in which each task is assigned with a unique priority level. We assume that the priority assignment is given beforehand and that the tasks are numbered in a decreasing priority order. That is, a task with a smaller index has a higher priority than any task

with a higher index, i.e., task  $\tau_i$  has a higher-priority than task  $\tau_j$  if  $i < j$ .

When performing the schedulability analysis of a specific task  $\tau_k$ , we will implicitly assume that all the higher priority tasks (i.e.,  $\tau_1, \tau_2, \dots, \tau_{k-1}$ ) are already verified to meet their deadlines, i.e., that  $R_i \leq D_i, \forall \tau_i \mid 1 \leq i \leq k-1$ .

## III. BACKGROUND

To analyze the worst-case response time (or the schedulability) of a task  $\tau_k$ , one usually needs to quantify the worst-case interference exerted by the higher-priority tasks on the execution of any job of task  $\tau_k$ . In the ordinary sequential sporadic real-time task model, i.e., when  $S_i = 0$  for every task  $\tau_i$ , the so-called critical instant theorem by Liu and Layland [16] is commonly adopted. That is, the worst-case response time of task  $\tau_k$  (if it is less than or equal to its period) happens for the first job of task  $\tau_k$  when (i)  $\tau_k$  and all the higher-priority tasks release their first job synchronously and (ii) all their subsequent jobs are released as early as possible (i.e., with a rate equal to their period). However, this definition of the critical instant does not hold for self-suspending sporadic tasks.

The analysis of self-suspending task systems requires to model the self-suspending behavior of both the task  $\tau_k$  under analysis and the higher priority tasks that interfere with  $\tau_k$ . The techniques employed to model the self-suspension are usually different for  $\tau_k$  and the higher priority tasks. The worst-case for  $\tau_k$  happens when its jobs suspend whenever there is no higher-priority job in the system. The resulting behavior is therefore similar as if the suspension time  $S_k$  of task  $\tau_k$  was converted into computation time (see [10] for more detailed explanations). Second, for the higher-priority tasks, we need to consider the self-suspension behaviour that may result in the largest possible interference for task  $\tau_k$ . There exist three different approaches in the state-of-the-art that are potentially sound to perform the schedulability analysis of self-suspending tasks:

- modeling the suspension as execution, also known as the suspension-oblivious analysis (see Section III-A);
- modeling the suspension as a release jitter (see Section III-B);
- modeling the suspension as blocking time (see Section III-C).

We later prove in Section VI that all these approaches are analytically correct.

### A. Suspension-Oblivious Analysis

The simplest analysis consists in converting the suspension time  $S_i$  of each task  $\tau_i$  as a part of its computation time. Therefore, a constrained-deadline task  $\tau_k$  can be feasibly scheduled by a fixed-priority scheduling algorithm if

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

<sup>1</sup>A simplified version of the proof of Theorem 1 to support the correctness of [17, p. 164-165] and [20, p. 267] is provided in [6].

### B. Modeling the Suspension as a Release Jitter

Another approach consists in modeling the impact of the self-suspension  $S_i$  of each higher priority task  $\tau_i$  as a release jitter. Several works in the state-of-the-art [1], [2], [14], [19] upper bounded the release jitter with  $S_i$ . However, it has been recently shown in [3] that this upper bound is unsafe and the release jitter of task  $\tau_i$  can in fact be larger than  $S_i$ .

Nevertheless, it was proven in the same document [3] that the jitter of a higher-priority task  $\tau_i$  can be safely upper bounded by  $R_i - C_i$ . It results that a task  $\tau_k$  with a constrained deadline can be feasibly scheduled under fixed-priority if

$$\exists t \mid 0 < t \leq D_k, \quad C_k + S_k + \sum_{i=1}^{k-1} \left\lceil \frac{t + R_i - C_i}{T_i} \right\rceil C_i \leq t. \quad (2)$$

### C. Modeling the Suspension as Blocking Time

In [17, p. 164-165], Liu proposed a solution to study the schedulability of a self-suspending task  $\tau_k$  by modeling the extra delay suffered by  $\tau_k$  due to the self-suspension behavior of each task in  $\tau$  as a blocking time. This blocking time has been defined as follows:

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

An upper bound on the blocking time is therefore given by:

$$B_k = S_k + \sum_{i=1}^{k-1} \min(C_i, S_i). \quad (3)$$

In [17], the blocking time is then used to derive a utilization-based schedulability test for rate-monotonic scheduling. Namely, it is stated that, if  $T_i = D_i$  for every task  $\tau_i \in \tau$  and  $\frac{C_k + B_k}{T_k} + \sum_{i=1}^{k-1} U_i \leq k(2^{\frac{1}{k}} - 1)$ , then  $\tau_k$  can be feasibly scheduled with rate-monotonic scheduling.

The same concept was also implicitly used by Rajkumar, Sha, and Lehoczky in [20, p. 267] for analyzing the impact of the self-suspension of a task due to the utilization of synchronization protocols in multiprocessor systems. The statement in [20] reads as follows:<sup>2</sup>

*“For each higher priority job  $\tau_{i,j}$  that suspends on global semaphores or for other reasons, add the term  $\min(C_i, S_i)$  to  $B_k$ , where  $S_i$  is the maximum duration that  $\tau_{i,j}$  can suspend itself. [...] The sum [...] yields  $B_k$ , which in turn can be used in  $\frac{C_k + B_k}{T_k} + \sum_{i=1}^{k-1} U_i \leq k(2^{\frac{1}{k}} - 1)$  to determine whether the current task allocation to the processor is schedulable.”*

<sup>2</sup>We rephrased the wording and notation in order to be consistent with the rest of this paper. Moreover, the multiprocessor scheduling in such a case is based on partitioned scheduling. Therefore, the schedulability analysis of a task set on a processor is the same as the uniprocessor problem by additionally considering the self-suspending behaviour due to the synchronization with other tasks on other processors.

If the above argument is correct, we can further prove that a constrained-deadline task  $\tau_k$  can be feasibly scheduled under fixed-priority scheduling if

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

However, there is no proof in [17] nor in [20] to support the correctness of those tests. Therefore, in Section VI, we provide a proof (see Theorem 3) of the correctness of Equation (4).

## IV. RATIONALE

Even though it can be proven that the response time analysis associated with Eq.(4) dominates the suspension oblivious one (see Lemma 3 in Section VI), none of the analyses presented in Section III dominates all the others. Hence, Eqs. (2) and (4) are incomparable. That is, in some cases Eq. (4) performs better than Eq. (2), while in others Eq. (2) outperforms Eq. (4).

**Example 1.** Consider the two tasks  $\tau_1 = (4, 5, 10, 10)$  and  $\tau_2 = (6, 1, 19, 19)$ . The worst-case response time of  $\tau_1$  is obviously 9 whatever the analysis employed. However, the upper bound on the WCRT of  $\tau_2$  obtained with Eq. (2) is 15, while it is 19 with Eq. (4). The solution obtained with Eq. (2) is therefore tighter.

Now, let us consider one more task  $\tau_3 = (4, 0, 50, 50)$ . Using Eq. (2), the WCRT of task  $\tau_3$  is upper bounded by the smallest  $t > 0$  such that  $t = 4 + \left\lceil \frac{t+9-4}{10} \right\rceil 1 + \left\lceil \frac{t+15-6}{19} \right\rceil 6$ , which turns out to be 42. With Eq. (4) though,  $B_3 = 4 + 1 = 5$  (Eq. (3)) and an upper bound on the WCRT of  $\tau_3$  is given by the smallest  $t > 0$  such that  $C_3 + B_3 + \sum_{i=1}^2 \left\lceil \frac{t}{T_i} \right\rceil C_i \leq t$ . The solution to this last equation is  $t = 37$ . Therefore, Eq. (4) provides a tighter bound on the WCRT of  $\tau_3$  than Eq. (2), while the opposite was true for  $\tau_2$ .  $\square$

In addition to the fact that Eqs. (2) and (4) are incomparable, there might be task sets for which both equations overestimate the WCRT. One such example is given below.

**Example 2.** Consider the same three tasks than in Example 1. As explained in Section III-B, the extra interference caused by the self-suspending behavior of  $\tau_1$  can be safely modeled by a release jitter equal to  $R_1 - S_1 = 5$ . Similarly, the extra interference caused by the self-suspension of  $\tau_2$  can be modeled by a blocking time equal to  $\min(C_2, S_2) = 1$  (see Section III-C). Hence, the WCRT of  $\tau_3$  is upper bounded by the smallest  $t > 0$  such that  $t = 4 + 1 + \left\lceil \frac{t+5}{10} \right\rceil 1 + \left\lceil \frac{t}{19} \right\rceil 6$ , which turns out to be 33. This bound on the WCRT is smaller than the estimates obtained with both Eqs. (2) and (4) (see Example 1).  $\square$

Example 2 shows that a tighter bound on the WCRT of a task can be obtained by combining the properties of the analyses discussed in both Section III-B and III-C. Therefore, in this paper, we derive a response time analysis that draws inspiration from both Eqs. (2) and (4), combining the best of each of them. As further proven in Section VI, the resulting schedulability test dominates all the tests discussed in Section III.

## V. A UNIFYING ANALYSIS FRAMEWORK

As already discussed in Section III, one can greedily convert the suspension time of task  $\tau_k$  in computation time. Let  $\tau'_k$  be this converted version of task  $\tau_k$ , i.e.,  $\tau'_k = (C_k + S_k, 0, D_k, T_k)$ . Suppose that  $R'_k$  is the worst-case response time of  $\tau'_k$  in the modified task set  $\{\tau_1, \tau_2, \dots, \tau_{k-1}, \tau'_k\}$ . It was already shown in previous works, e.g., Lemma 3 in [18], that  $R'_k$  is a safe upper bound on the worst-case response time of task  $\tau_k$  in the original task set.

Note that in all this section, we implicitly assume that  $R_i \leq D_i, \forall \tau_i \mid 1 \leq i \leq k-1$ . Our key result in this paper is the following theorem:

**Theorem 1.** *Suppose that  $R_k \leq T_k$ , then for any arbitrary vector assignment  $\vec{x} = (x_1, x_2, \dots, x_{k-1})$ , in which  $x_i$  is either 0 or 1, the worst-case response time  $R_k$  of  $\tau_k$  is upper bounded by the minimum  $t$  larger than 0 such that*

$$C_k + S_k + \sum_{i=1}^{k-1} \left\lceil \frac{t + Q_i^{\vec{x}} + (1 - x_i)(R_i - C_i)}{T_i} \right\rceil C_i \leq t \quad (5)$$

where  $Q_i^{\vec{x}} \stackrel{\text{def}}{=} \sum_{j=i}^{k-1} (S_j \times x_j)$ .

One can directly derive the following schedulability test from Theorem 1.

**Corollary 1.** *If there is a vector  $\vec{x} = (x_1, x_2, \dots, x_{k-1})$  with  $x_i \in \{0, 1\}$ , such that*

$$\exists t \mid 0 < t \leq D_k, \\ C_k + S_k + \sum_{i=1}^{k-1} \left\lceil \frac{t + Q_i^{\vec{x}} + (1 - x_i)(R_i - C_i)}{T_i} \right\rceil C_i \leq t \quad (6)$$

where  $Q_i^{\vec{x}} \stackrel{\text{def}}{=} \sum_{j=i}^{k-1} (S_j \times x_j)$ , then the constrained-deadline task  $\tau_k$  is schedulable under fixed-priority.

The proof of correctness of Theorem 1 and hence Corollary 1 is provided in Section V-A. Moreover, we will later prove in Section VI, that Corollary 1 in fact dominates all the analyses discussed in Section III.

We now use the same example as in Section IV, to demonstrate how Corollary 1 can be applied.

**Example 3.** *Consider the same three tasks used in Examples 1 and 2, i.e.,  $\tau_1 = (4, 5, 10, 10)$ ,  $\tau_2 = (6, 1, 19, 19)$  and  $\tau_3 = (4, 0, 50, 50)$ . There are four possible vector assignments  $\vec{x}$  when considering the schedulability of task  $\tau_3$  with Corollary 1:*

**Case 1.**  $\vec{x} = (0, 0)$ : *In this case, Theorem 1 states that  $R_k$  is upper bounded by the minimum  $t$  under  $0 < t \leq D_3$  such that  $4 + \lceil \frac{t+5}{10} \rceil 4 + \lceil \frac{t+9}{19} \rceil 6 \leq t$ . Note that this equation is identical to the schedulability test discussed in Section III-B, and hence, as shown in Example 1, we get that  $R_k \leq 42$ .*

**Case 2.**  $\vec{x} = (0, 1)$ : *In this case, Theorem 1 states that  $R_k$  is upper bounded by the minimum  $t$  under  $0 < t \leq D_3$  that satisfies  $4 + \lceil \frac{t+6}{10} \rceil 4 + \lceil \frac{t+1}{19} \rceil 6 \leq t$ . As a solution, we get that  $R_k \leq 32$ .*

**Case 3.**  $\vec{x} = (1, 0)$ : *In this case, we look for the minimum  $t$  such that  $4 + \lceil \frac{t+5}{10} \rceil 4 + \lceil \frac{t+9}{19} \rceil 6 \leq t$ . Hence, we get  $R_k \leq 42$ .*

**Case 4.**  $\vec{x} = (1, 1)$ : *In this case, Theorem 1 states that  $R_k$  is upper bounded by the minimum  $t$  under  $0 < t \leq T_3$  such that  $4 + \lceil \frac{t+6}{10} \rceil 4 + \lceil \frac{t+1}{19} \rceil 6 \leq t$  leading to  $R_k \leq 32$ .*

Among the above four cases, the tests in Cases 2 and 4 are the tightest. Therefore, by Corollary 1,  $\tau_3$  is schedulable under fixed-priority.  $\square$

Note also that the upper bound on  $\tau_3$ 's WCRT computed in Example 3, is lower than the WCRT estimate obtained in Example 2. The response time analysis presented in Corollary 1 is therefore tighter than the simple combination of existing analysis techniques proposed in Example 2.

### A. Proof of Correctness

We now provide the proof to support the correctness of the response time analysis presented in Theorem 1, whatever the binary values used in vector  $\vec{x}$ .

Throughout the proof, we consider any arbitrary assignment  $\vec{x}$ , in which  $x_i$  is either 0 or 1. For the sake of clarity, we classify the  $k-1$  higher-priority tasks into two sets:  $\mathbf{T}_0$  and  $\mathbf{T}_1$ . A task  $\tau_i$  is in  $\mathbf{T}_0$  if  $x_i$  is 0; otherwise, it is in  $\mathbf{T}_1$ .

Our analysis is also based on very simple properties and lemmas enunciated as follows:

**Property 1.** *In a preemptive fixed-priority schedule, the lower-priority jobs do not impact the schedule of the higher-priority jobs.*

**Lemma 1.** *In a preemptive fixed-priority schedule, if the worst-case response time of task  $\tau_i$  is no more than its period  $T_i$ , preventing the release of a job of task  $\tau_i$  does not affect the schedule of any other job of task  $\tau_i$ .*

*Proof:* Since, by assumption, the worst-case response time of task  $\tau_i$  is no more than its period, any job  $\tau_{i,j}$  of task  $\tau_i$  completes its execution before the release of the next job  $\tau_{i,j+1}$ . Hence, the execution of  $\tau_{i,j}$  does not directly interfere with the execution of any other job of  $\tau_i$ , which then depends only on the schedule of the higher priority jobs. Furthermore, as stated in Property 1, the removal of  $\tau_{i,j}$  has no impact on the schedule of the higher-priority jobs, thereby implying that the other jobs of task  $\tau_i$  are not affected by the removal of  $\tau_{i,j}$ .  $\blacksquare$

With the above properties, we can present the detailed proof of Theorem 1. Since the proof is pretty long, we will also provide some examples to demonstrate the steps in the proof.

**Proof of Theorem 1.** Consider the modified task set  $\tau'$  composed of  $\{\tau_1, \tau_2, \dots, \tau_{k-1}, \tau'_k, \tau_{k+1}, \dots\}$  where  $\tau'_k = (C_k + S_k, 0, D_k, T_k)$ . Let  $\Psi$  be a schedule of  $\tau'$  such that  $R'_k \leq T_k$ . Suppose that a job  $J_k$  of task  $\tau'_k$  arrives at time  $r_k$  and finishes at time  $f_k$ . We prove that Eq. (5) gives us a safe upper bound on  $f_k - r_k$  for any job  $J_k$  in  $\Psi$ .

The proof is built upon the three following steps:

- 1) We discard all the jobs that do not contribute to the response time of  $J_k$  in the schedule  $\Psi$ . We follow an inductive strategy by iteratively inspecting the schedule of the higher priority tasks in  $\Psi$ , starting with  $\tau_{k-1}$  until the highest priority task  $\tau_1$ . At each

iteration, a time instant  $t_j$  is identified such that  $t_j \leq t_{j+1}$  ( $1 \leq j < k$ ). Then, all the jobs of task  $\tau_j$  released before  $t_j$  are removed from the schedule and, if needed, replaced by an artificial job mimicking the interference caused by the residual workload of task  $\tau_j$  at time  $t_j$ .

- 2) The final reduced schedule is analyzed so as to characterize the worst-case response time of  $\tau'_k$  in  $\Psi$ .
- 3) We then prove that the response time analysis in Eq. (5) is indeed an upper bound on the worst-case response time  $R'_k$  of  $\tau'_k$ .

### Step 1: Reducing the schedule $\Psi$

During this step, we iteratively build an artificial schedule  $\Psi^j$  from  $\Psi^{j+1}$  (with  $1 \leq j < k$ ) so that the response time of  $\tau'_k$  remains identical. At each iteration, we define  $t_j$  for task  $\tau_j$  in the schedule  $\Psi^{j+1}$  (with  $j = k-1, k-2, \dots, 1$ ) and build  $\Psi^j$  by removing all the jobs released by  $\tau_j$  before  $t_j$ .

*Basic step (definition of  $\Psi^k$  and  $t_k$ ):*

Recall that the job  $J_k$  of task  $\tau'_k$  arrives at time  $r_k$  and finishes at time  $f_k$  in schedule  $\Psi$ . We know by Property 1 that the lower priority tasks  $\tau_{k+1}, \tau_{k+2}, \dots, \tau_n$  do not impact the response time of  $J_k$ . Moreover, since we assume that the worst-case response time of task  $\tau'_k$  is no more than  $T_k$ , Lemma 1 proves that removing all the jobs of task  $\tau'_k$  but  $J_k$  has no impact on the schedule of  $J_k$ . Therefore, let  $\Psi^k$  be a schedule identical to  $\Psi$  but removing all the jobs released by the lower priority tasks  $\tau_{k+1}, \dots, \tau_n$  as well as all the jobs released by  $\tau'_k$  at the exception of  $J_k$ . The response time of  $J_k$  in  $\Psi^k$  is identical to the response time of  $J_k$  in  $\Psi$ .

We define  $t_k$  as the release time of  $J_k$  (i.e.,  $t_k = r_k$ ).

*Induction step (definition of  $\Psi^j$  and  $t_j$  with  $1 \leq j < k$ ):*

Let  $r_j$  be the arrival time of the last job released by  $\tau_j$  before  $t_{j+1}$  in  $\Psi^{j+1}$  and let  $J_j$  denote that job. Removing all the jobs of task  $\tau_j$  arrived before  $r_j$  has no impact on the schedule of any other job released by  $\tau_j$  (Lemma 1) or any higher priority job released by  $\tau_1, \dots, \tau_{j-1}$  (Property 1). Moreover, because by the construction of  $\Psi^{j+1}$ , no task with a priority lower than  $\tau_j$  executes jobs before  $t_{j+1}$  in  $\Psi^{j+1}$ , removing the jobs released by  $\tau_j$  before  $t_{j+1}$  does not impact the schedule of the jobs of  $\tau_{j+1}, \dots, \tau_k$ . Therefore, we can safely remove all the jobs of task  $\tau_j$  arrived before  $r_j$  without impacting the response time of  $J_k$ . Two cases must then be considered: (a)

- 1)  $\tau_j \in \mathbf{T}_1$ . In this case, we analyze two different subcases:
  - $J_j$  did not complete its execution by  $t_{j+1}$ . For such a case,  $t_j$  is set to  $r_j$  and hence  $\Psi^j$  is built from  $\Psi^{j+1}$  by removing all the jobs released by  $\tau_j$  before  $r_j$ .
  - $J_j$  completed its execution before or at  $t_{j+1}$ . By Lemma 1 and Property 1, removing all the jobs of task  $\tau_j$  arrived before  $t_{j+1}$  has no impact on the schedule of the higher-priority jobs (jobs released by  $\tau_1, \dots, \tau_{j-1}$ ) and the jobs of  $\tau_j$  released after or at  $t_{j+1}$ . Moreover, because no task with lower priority

than  $\tau_j$  executes jobs before  $t_{j+1}$  in  $\Psi^{j+1}$ , removing the jobs released by  $\tau_j$  before  $t_{j+1}$  does not impact the schedule of the jobs of  $\tau_{j+1}, \dots, \tau_k$ . Therefore,  $t_j$  is set to  $t_{j+1}$  and  $\Psi^j$  is generated by removing all the jobs of task  $\tau_j$  arrived before  $t_{j+1}$ . The response time of  $J_k$  in  $\Psi^j$  thus remains unchanged in comparison to its response time in  $\Psi^{j+1}$ .

- 2)  $\tau_j \in \mathbf{T}_0$ . For such a case, we set  $t_j$  to  $t_{j+1}$ . Let  $c_j^*$  be the remaining execution time for the job of task  $\tau_j$  at time  $t_j$ . By definition,  $c_j^* \geq 0$ , and we also know that  $c_j^*$  is at most  $C_j$ . Since by the construction of  $\Psi^j$ , all the jobs of  $\tau_j$  released before  $t_j$  are removed, the job of task  $\tau_j$  arrived at time  $r_j$  ( $< t_j$ ) is replaced by a new job released at time  $t_j$  with execution time  $c_j^*$  and the same priority than  $\tau_j$ . Clearly, this has no impact on the execution of any job executed after  $t_j$  and thus on the response time of  $J_k$ . The remaining execution time  $c_j^*$  of  $\tau_j$  at time  $t_j$  is called the *residual workload* of task  $\tau_j$  in the rest of the proof.

This iterative process is repeated until producing  $\Psi^1$ . The procedures are well-defined and it is therefore guaranteed that  $\Psi^1$  can be constructed. Note that after each iteration, the number of jobs considered in the resulting schedule has been reduced, yet without affecting the response time of  $J_k$ .

**Example 4.** Consider the 4 tasks  $\tau_1 = (1, 1, 6, 6)$ ,  $\tau_2 = (1, 6, 10, 10)$ ,  $\tau_3 = (4, 1, 18, 18)$  and  $\tau_4 = (5, 0, 20, 20)$ .

Figure 1 depicts a possible schedule of those tasks. We assume that the first job of task  $\tau_1$  arrives at time  $4 + \epsilon$  with a very small  $\epsilon > 0$ . The first job of task  $\tau_2$  suspends itself from time 0 to time  $5 + \epsilon$ , and is blocked by task  $\tau_1$  from time  $5 + \epsilon$  to time  $6 + \epsilon$ . After some very short computation with  $\epsilon$  amount of time, the first job of task  $\tau_2$  suspends itself again from time  $6 + 2\epsilon$  to 7.

In the schedule illustrated in Figure 1,  $f_4$  is  $20 - \epsilon$ . We define  $t_4$  as 7. Then, we set  $t_3$  to 6. When considering task  $\tau_2$ , since it belongs to  $\mathbf{T}_0$ , we greedily set  $t_2$  to  $t_3 = 6$  and the residual workload  $c_2^*$  is 1. Then,  $t_1$  is set to  $4 + \epsilon$ . In the above schedule, the idle time from  $4 + \epsilon$  to  $20 - \epsilon$  is at most  $2 = S_1 + S_3$ . We have to further consider one job of task  $\tau_2$  arrived before time  $t_1$  with execution time  $C_2$ .

**Lemma 2.** Let  $\sigma_j$  be the amount of time during which the processor remains idle within  $[t_j, t_{j+1})$  in  $\Psi^1$ . It holds that  $\sum_{j=1}^{i-1} \sigma_j \leq \sum_{j=1}^{i-1} x_j S_j$ .

*Proof:* If  $t_j = t_{j+1}$ , which is the case when  $x_j = 0$  (see Case (b) of the reduction process), then  $\sigma_j$  is obviously equal to 0.

If  $t_j \neq t_{j+1}$ , then we are in subcase 1 of Case (a) of the schedule reduction process (i.e., when  $x_j = 1$ ). In that case,  $t_j = r_j$  and the job  $J_j$  did not complete its execution yet. Therefore, the amount of time during which the processor may remain idle within  $[t_j, t_{j+1})$  is at most  $S_j$  time units.

From those two cases, it results that  $\sum_{j=1}^{i-1} \sigma_j = \sum_{j=1}^{i-1} x_j S_j \leq \sum_{j=1}^{i-1} x_j S_j$ .  $\blacksquare$

### Step 2: Analyzing the final reduced schedule $\Psi^1$

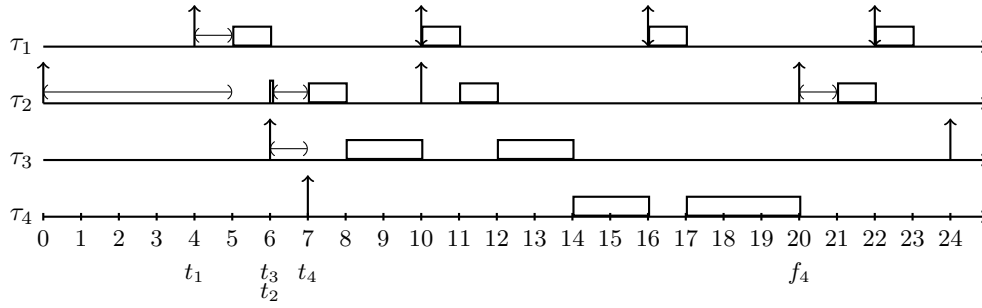


Fig. 1: An illustrative example of Step 1 in the proof of Theorem 1.

We now analyze the properties of the final schedule  $\Psi^1$  in which all the unnecessary jobs have been removed. The proof is based on the fact that for any interval  $[t_1, t)$ , there is

$$\text{idle}(t_1, t) + \text{exec}(t_1, t) = (t - t_1) \quad (7)$$

where  $\text{exec}(t_1, t)$  is the amount of time during which the processor executes tasks within  $[t_1, t)$ , and  $\text{idle}(t_1, t)$  is the amount of time during which the processor remains idle within the interval  $[t_1, t)$ .

From Lemma 2, it holds that

$$\text{idle}(t_1, t) \leq \sum_{j=1}^{i-1} x_j S_j. \quad (8)$$

Because there is no job released by lower priority tasks than  $\tau'_k$  in  $\Psi^1$ , we only focus on the execution patterns of the tasks  $(\tau_1, \tau_2, \dots, \tau_{k-1}, \tau'_k)$ . According to Step 1, we should consider two cases:

- $\tau_j \in \mathbf{T}_1$ . This corresponds to Case (a) in Step 1, which tells us that there is no job of task  $\tau_j$  arrived before  $t_j$  in  $\Psi^1$ . In this case, for any  $\Delta \geq 0$ , the workload  $W_j^1(\Delta)$ , executed by  $\tau_j$  on the processor from  $t_j$  to  $t_j + \Delta$  is upper bounded by

$$W_j^1(\Delta) = \left\lfloor \frac{\Delta}{T_j} \right\rfloor C_j + \min \left\{ \Delta - \left\lfloor \frac{\Delta}{T_j} \right\rfloor T_j, C_j \right\}. \quad (9)$$

- $\tau_j \in \mathbf{T}_0$ . This corresponds to case (b) in Step 1, which tells us that there might be a job arrived before  $t_j$  with a residual workload  $c_j^*$  at time  $t_j$ . Let  $c_j^*$  be the residual workload of task  $\tau_j$  at  $t_j$ . Since by assumption  $\tau_j$  respects all its deadlines, the absolute deadline of the job of  $\tau_j$  active at  $t_j$  must be at least  $t_j + c_j^*$ ; otherwise that job would miss its deadline. Therefore, the earliest arrival time of task  $\tau_j$  arriving *strictly after*  $t_j$  is at least  $t_j + (T_j - D_j + c_j^*)$ . For notational brevity, let  $\rho_j$  be  $(T_j - D_j + c_j^*)$ . In this case, for any  $\Delta \geq 0$  and  $c_j^* > 0$ , the workload  $\widehat{W}_j^0(\Delta, c_j^*)$  executed by  $\tau_j$  from  $t_j$  to  $t_j + \Delta$  is upper bounded by

$$\widehat{W}_j^0(\Delta, c_j^*) = \begin{cases} \Delta & \text{if } \Delta \leq c_j^* \\ c_j^* & \text{if } c_j^* < \Delta \leq \rho_j \\ c_j^* + W_j^1(\Delta - \rho_j) & \text{otherwise.} \end{cases} \quad (10)$$

It is easy to see that Eq. (10) is maximized when  $c_j^*$  is maximum, that is, when  $c_j^* = C_j$ . It results that for all  $\Delta \geq 0$ , we have  $\widehat{W}_j^0(\Delta, C_j) \geq \widehat{W}_j^0(\Delta, c_j^*)$ . For the sake of notational brevity, for the rest of this proof, we use the notation  $W_j^0(\Delta)$  to denote  $\widehat{W}_j^0(\Delta, C_j)$ .

Summing the workloads of the tasks in  $\mathbf{T}_0$  and  $\mathbf{T}_1$ , we have for  $i = 2, 3, \dots, k-1$  that  $\forall t \mid t_{i-1} \leq t < t_i$

$$\text{exec}(t_1, t) \leq \sum_{j=1}^{i-1} x_j \cdot W_j^1(t - t_j) + (1 - x_j) \cdot W_j^0(t - t_j). \quad (11)$$

Injecting Eqs. (8) and (11) in Eq. (7), we have for  $i = 2, 3, \dots, k-1$  that  $\forall t \mid t_{i-1} \leq t < t_i$

$$\sum_{j=1}^{i-1} x_j \cdot (W_j^1(t - t_j) + \sigma_j) + (1 - x_j) \cdot W_j^0(t - t_j) \geq t - t_1. \quad (12)$$

Moreover, since  $\tau'_k$  does not complete its execution before  $f_k$  and because, by definition,  $\tau'_k$  does not self-suspend, it holds that  $\forall t \mid t_k \leq t < f_k$ ,

$$C'_k + \sum_{j=1}^{k-1} x_j \cdot (W_j^1(t - t_j) + \sigma_j) + (1 - x_j) \cdot W_j^0(t - t_j) > t - t_1. \quad (13)$$

**Example 5.** Consider the same 4 tasks as in Example 4, for which a possible schedule was depicted in Figure 1. We have  $\sigma_1 = 1$ ,  $\sigma_2 = 0$  and  $\sigma_3 = 1$ . The corresponding functions  $W_1^1(t - t_1)$ ,  $W_2^0(t - t_2)$ ,  $W_3^1(t - t_3)$  are illustrated in Figure 2. As can be seen in the figure, the inequalities of Eqs. (12) and (13) clearly hold.

### Step 3: Creating a Safe Response-Time Upper Bound

This step constructs a safe response-time analysis based on the conditions specified by Eqs. (12) and (13). To do so, we construct another release pattern which moves  $t_i$  to  $t_i^*$  for  $i = 2, 3, \dots, k$  such that  $t_i^* \leq t_i$  and the corresponding conditions in Eqs. (12) and (13) will become worse when we use  $t_i^*$ . We start the procedure as follows:

- Initial step: Let  $t_1^*$  be  $t_1$ .
- Iterative steps ( $i = 2, 3, \dots, k$ ): Let  $t_i^*$  be  $t_{i-1}^* + x_{i-1} \cdot \sigma_{i-1}$ .



Fig. 3: An illustrative example of Step 3 in the proof of Theorem 1 based on an *imaginary* schedule.

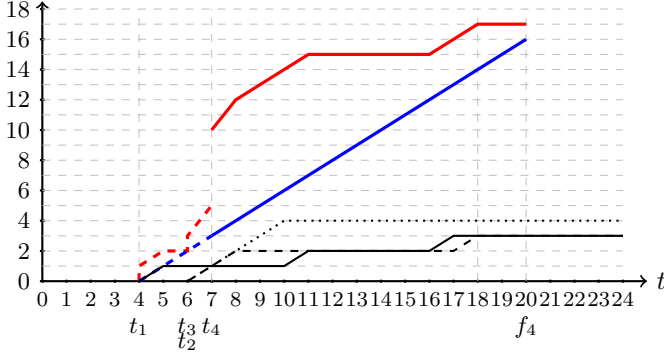


Fig. 2: The workload function for the three higher-priority tasks in Figure 2. Solid black line:  $W_1^1(t - t_1)$ , Dashed black line:  $W_2^1(t - t_2)$ , Dotted black line:  $W_3^1(t - t_3)$ , where the functions are 0 if  $t - t_j < 0$  for  $j = 1, 2, 3$ , Blue line:  $t - t_1$ , Red line: left-hand side of Eq. (12) when  $t < 7$  and left-hand side of Eq. (13) when  $7 \leq t < 20$ .

This results in  $t_i^* \leq t_i$  for  $i = 2, 3, \dots, k$ . Moreover, by definition,  $t_j^*$  is  $t_1^* + \sum_{i=1}^{j-1} x_i \cdot \sigma_i$  for  $j = 2, 3, \dots, k$ . For any task  $\tau_j$  in  $\mathbf{T}_1$ ,  $\forall \Delta \geq 0$ , since  $t_j \geq t_j^*$ , we have

$$W_j^1(\Delta) \leq W_j^1(\Delta + (t_j - t_j^*)). \quad (14)$$

For any task  $\tau_j$  in  $\mathbf{T}_0$ ,  $\forall \Delta \geq 0$ , since  $t_j \geq t_j^*$ , we have

$$W_j^0(\Delta) \leq W_j^0(\Delta + (t_j - t_j^*)). \quad (15)$$

Therefore, for any  $j = 1, 2, \dots, k - 1$ , the contribution  $W_j^1(t - t_j) \leq W_j^1(t - t_j^*)$  and  $W_j^0(t - t_j) \leq W_j^0(t - t_j^*)$  for any  $t \geq t_j$ . Putting these into Eqs. (12)  $\forall t \mid t_k^* \leq t < t_k$  leads to

$$\begin{aligned} & \sum_{j=1}^{k-1} x_j \cdot (W_j^1(t - t_j^*) + \sigma_j) + (1 - x_j) \cdot W_j^0(t - t_j^*) \geq t - t_1, \\ \Rightarrow & \sum_{j=1}^{k-1} x_j \cdot W_j^1(t - t_j^*) + (1 - x_j) \cdot W_j^0(t - t_j^*) \geq t - t_k^*. \end{aligned} \quad (16)$$

Similarly, putting these into Eqs. (13)  $\forall t \mid t_k \leq t < f_k$  leads

to

$$C'_k + \sum_{j=1}^{k-1} x_j \cdot W_j^1(t - t_j^*) + (1 - x_j) \cdot W_j^0(t - t_j^*) > t - t_k^*. \quad (17)$$

By the fact that  $C'_k \geq C_k > 0$ , we can unify the above inequalities in Eq. (16) and Eq. (17) as follows:  $\forall t \mid t_k^* \leq t < f_k$

$$C'_k + \sum_{j=1}^{k-1} x_j \cdot W_j^1(t - t_j^*) + (1 - x_j) \cdot W_j^0(t - t_j^*) > t - t_k^*. \quad (18)$$

By the definition of  $t_j^*$ ,  $\forall t \mid t_k^* \leq t < f_k$ , we have  $t - t_j^* = t - t_k^* + \sum_{\ell=j}^{k-1} x_\ell \sigma_\ell$  for every  $j = 1, 2, \dots, k - 1$ . Therefore, we know that  $W_j^1(t - t_j^*) \leq \left\lceil \frac{t - t_k^* + \sum_{\ell=j}^{k-1} x_\ell \sigma_\ell}{T_j} \right\rceil C_j$  for task  $\tau_j$  in  $\mathbf{T}_1$ . Moreover,  $\forall t \mid t_k^* \leq t < f_k$ , we have  $W_j^0(t - t_j^*) \leq \left\lceil \frac{t - t_k^* + \sum_{\ell=j}^{k-1} x_\ell \sigma_\ell + (1 - x_j)(D_j - C_j)}{T_j} \right\rceil C_j$  for task  $\tau_j$  in  $\mathbf{T}_0$ . Therefore, we can conclude that  $\forall t \mid t_k^* \leq t < f_k$

$$C'_k + \sum_{j=1}^{k-1} \left\lceil \frac{t - t_k^* + X_j + (1 - x_j)(D_j - C_j)}{T_j} \right\rceil C_j > t - t_k^*, \quad (19)$$

where  $X_j$  is  $\sum_{\ell=j}^{k-1} x_\ell \sigma_\ell$ . We replace  $t - t_k^*$  with  $\theta$ . The above inequation implies that the minimum  $\theta$  with  $\theta > 0$  such that  $C'_k + \sum_{j=1}^{k-1} \left\lceil \frac{\theta + X_j + (1 - x_j)(D_j - C_j)}{T_j} \right\rceil C_j = \theta$  is larger than or equal to  $f_k - t_k^* \geq f_k - t_k$ .

However, the above condition requires the knowledge of  $\sigma_i$ . It is straightforward to see that  $\sum_{j=1}^{k-1} \left\lceil \frac{\theta + X_j + (1 - x_j)(D_j - C_j)}{T_j} \right\rceil C_j$  reaches the worst case if  $X_j$  is the largest. Since  $X_j$  is upper bounded by  $Q_j^x$  defined in Theorem 1, we reach the conclusion.

**Example 6.** This can be demonstrated in Figure 3 based on the previous example in Figure 1. Figure 3 provides the imaginary workload and an imaginary execution plan based on the test behind the condition in Eq. (18). Note that this is not an actual schedule since task  $\tau_2$  is artificially alerted to release two jobs within a short time interval. This is only for illustrative purposes. For such a case,  $t_1^* = 4, t_2^* = 5, t_3^* = 5$ , and  $t_4^* = 6$ . The two idle time units are used between time 4 and time 6. The accumulated workload is then started to be executed at

time 6 and the processor does not idle after time 6. Over here, we see that two jobs of task  $\tau_2$  are executed back to back from time 7 to time 9. As shown in the imaginary schedule in Figure 3, the processor is busy executing the workload from time 6 to time 21, which is more pessimistic than the actual in Figure 1. The conclusion we have in the final statement of the theorem is that  $20 - 7 = f_k - r_k \leq 21 - 6$ .

□

## VI. DOMINANCE OVER THE STATE OF THE ART

In this section, we prove that the schedulability test presented in Corollary 1 dominates all the existing tests in the state-of-the-art, in the sense that if a task set is deemed schedulable by either of the tests presented in Section III, then it is also deemed schedulable by Corollary 1.

**Lemma 3.** *The schedulability test of task  $\tau_k$  provided by Eq. (4) dominates that of Eq. (1).*

*Proof:* It is straightforward to see that

$$\begin{aligned} & C_k + S_k + \sum_{i=1}^{k-1} \left\lceil \frac{t}{T_i} \right\rceil (C_i + S_i) \\ & \geq C_k + S_k + \sum_{i=1}^{k-1} S_i + \sum_{i=1}^{k-1} \left\lceil \frac{t}{T_i} \right\rceil C_i \\ & \geq C_k + S_k + \sum_{i=1}^{k-1} \min(C_i, S_i) + \sum_{i=1}^{k-1} \left\lceil \frac{t}{T_i} \right\rceil C_i \end{aligned}$$

and by using the definition of  $B_k$  (i.e., Equation (3)), we get

$$C_k + S_k + \sum_{i=1}^{k-1} \left\lceil \frac{t}{T_i} \right\rceil (C_i + S_i) \geq C_k + B_k + \sum_{i=1}^{k-1} \left\lceil \frac{t}{T_i} \right\rceil C_i$$

Therefore, Eq. (4) will always have a solution which is smaller than or equal to the solution of Eq. (1). This proves the lemma. ■

**Lemma 4.** *The schedulability test presented in Corollary 1 dominates the schedulability test provided by Eq. (2).*

*Proof:* Consider the case where  $x_1 = x_2 = \dots = x_{k-1} = 0$ . Eq. (6) becomes identical to Eq. (2) for this particular vector assignment. Therefore, if Eq. (2) deems a task set as being schedulable, so does Corollary 1. This proves the lemma. ■

**Lemma 5.** *The schedulability test presented in Corollary 1 dominates the schedulability test provided by Eq. (4).*

*Proof:* In this proof, we first transform the worst-case response time analysis presented in Corollary 1 in a more pessimistic analysis. We then prove that this more pessimistic version of Corollary 1 provides the same solution as Eq. (4), which then proves the lemma.

Since  $Q_i^{\vec{x}} \stackrel{\text{def}}{=} \sum_{j=i}^{k-1} S_j \times x_j$ , it holds that  $Q_i^{\vec{x}} \leq Q_1^{\vec{x}}$  for

$i = 1, 2, \dots, k-1$ . It follows that

$$\begin{aligned} & C_k + S_k + \sum_{i=1}^{k-1} \left\lceil \frac{t + Q_i^{\vec{x}} + (1 - x_i)(R_i - C_i)}{T_i} \right\rceil \\ & \stackrel{(Q_i^{\vec{x}} \leq Q_1^{\vec{x}})}{\leq} C_k + S_k + \sum_{i=1}^{k-1} \left\lceil \frac{t + Q_1^{\vec{x}} + (1 - x_i)(R_i - C_i)}{T_i} \right\rceil \\ & \stackrel{(R_i \leq D_i \leq T_i)}{\leq} C_k + S_k + \sum_{i=1}^{k-1} \left\lceil \frac{t + Q_1^{\vec{x}} + (1 - x_i)T_i}{T_i} \right\rceil \\ & \stackrel{(x_i \in \{0,1\})}{\leq} C_k + S_k + \sum_{i=1}^{k-1} (1 - x_i)C_i + \sum_{i=1}^{k-1} \left\lceil \frac{t + Q_1^{\vec{x}}}{T_i} \right\rceil \end{aligned}$$

Therefore, the smallest positive value  $t$  such that

$$C_k + S_k + \sum_{i=1}^{k-1} (1 - x_i)C_i + \sum_{i=1}^{k-1} \left\lceil \frac{t + Q_1^{\vec{x}}}{T_i} \right\rceil \leq t \quad (20)$$

is always larger than or equal to the solution of Eq. (6).

Substituting  $(t + Q_1^{\vec{x}})$  by  $\theta$  in Eq. (20), we get that  $R_k$  is upper bounded by the minimum value  $(\theta - Q_1^{\vec{x}})$  greater than 0 (and therefore by the smallest  $\theta > 0$ ) such that

$$\begin{aligned} & C_k + S_k + \sum_{i=1}^{k-1} (1 - x_i)C_i + \sum_{i=1}^{k-1} \left\lceil \frac{\theta}{T_i} \right\rceil \leq \theta - Q_1^{\vec{x}} \\ \Leftrightarrow & C_k + S_k + Q_1^{\vec{x}} + \sum_{i=1}^{k-1} (1 - x_i)C_i + \sum_{i=1}^{k-1} \left\lceil \frac{\theta}{T_i} \right\rceil \leq \theta \\ \Leftrightarrow & C_k + S_k + \sum_{i=1}^{k-1} (x_i S_i + (1 - x_i)C_i) + \sum_{i=1}^{k-1} \left\lceil \frac{\theta}{T_i} \right\rceil C_i \leq \theta. \end{aligned} \quad (21)$$

Now, consider the particular vector assignment  $\vec{x}$  in which

$$x_i = \begin{cases} 1 & \text{if } S_i \leq C_i \\ 0 & \text{otherwise,} \end{cases}$$

for  $i = 1, 2, \dots, k-1$ . By the definition of  $B_k$  (i.e., Eq. (3)), we get that

$$B_k = S_k + \sum_{i=1}^{k-1} \min(C_i, S_i) = S_k + \sum_{i=1}^{k-1} (x_i S_i + (1 - x_i)C_i)$$

Eq. (21) thus becomes identical to Eq. (4). Therefore, if Eq. (4) deems a task set as being schedulable, so does Corollary 1. ■

**Theorem 2.** *The schedulability test presented in Corollary 1 dominates the schedulability tests provided by Equations (1), (2), and (4).*

*Proof:* It is a direct application of Lemmas 3, 4 and 5. ■

As a corollary of this theorem, it directly follows that all the response time analyses discussed in Section III are in fact correct. This provides the first proof of correctness for Eq. (4), which was initially presented in [17] but never proven correct.

**Theorem 3.** *The schedulability tests provided by Eqs (1), (2), and (4) are all correct.*



*Proof:* It directly results from the two following facts, leftmargin=0

- (i) by Theorem 2, the schedulability test presented in Corollary 1 dominates the schedulability tests provided by Equations (1), (2), and (4);
- (ii) as proven in Section V-A, Corollary 1 is correct. ■

## VII. LINEAR APPROXIMATION

To test the schedulability of a task  $\tau_k$ , Corollary 1 implies to test all the possible vector assignments  $\vec{x} = (x_1, x_2, \dots, x_{k-1})$ . Therefore,  $2^{k-1}$  possible combinations must therefore be tested, implying exponential time complexity. In this section, we thus provide a solution to reduce the time complexity associated to Corollary 1. Indeed, using a linear approximation of the test in Eq. (6), a good vector assignment can be derived in linear time.

By the definition of the ceiling operator, it holds that:

$$\begin{aligned} & C_k + S_k + \sum_{i=1}^{k-1} \left\lceil \frac{t + \sum_{\ell=i}^{k-1} x_\ell S_\ell + (1-x_i)(R_i - C_i)}{T_i} \right\rceil C_i \\ & \leq C_k + S_k + \sum_{i=1}^{k-1} \left( \frac{t + \sum_{\ell=i}^{k-1} x_\ell S_\ell + (1-x_i)(R_i - C_i)}{T_i} + 1 \right) C_i \\ & = C_k + S_k + \sum_{i=1}^{k-1} \left( U_i \cdot t + C_i + U_i(1-x_i)(R_i - C_i) + U_i \sum_{\ell=i}^{k-1} x_\ell S_\ell \right) \end{aligned} \quad (22)$$

Moreover, using the simple algebra property that for any two vectors  $\vec{a}$  and  $\vec{b}$  of size  $(k-1)$  there is  $\sum_{i=1}^k a_i \sum_{j=i}^k b_j = \sum_{j=1}^k b_j \sum_{i=1}^j a_i$ , we get that  $\sum_{i=1}^{k-1} U_i \sum_{\ell=i}^{k-1} x_\ell S_\ell = \sum_{i=1}^{k-1} x_i S_i \sum_{\ell=1}^i U_\ell$ . Hence, injecting this last expression in Eq. (22), it holds that

$$\begin{aligned} & C_k + S_k + \sum_{i=1}^{k-1} \left\lceil \frac{t + \sum_{\ell=i}^{k-1} x_\ell S_\ell + (1-x_i)(R_i - C_i)}{T_i} \right\rceil C_i \\ & \leq C_k + S_k + \sum_{i=1}^{k-1} \left( U_i \cdot t + C_i + U_i(1-x_i)(R_i - C_i) + x_i S_i \sum_{\ell=1}^i U_\ell \right) \end{aligned}$$

It results that the minimum positive value for  $t$  such that

$$C_k + S_k + \sum_{i=1}^{k-1} \left( U_i \cdot t + C_i + U_i(1-x_i)(R_i - C_i) + x_i S_i \sum_{\ell=1}^i U_\ell \right) \leq t \quad (23)$$

is an upper bound on the WCRT of  $\tau_k$ .

Observing Eq. (23), the contribution of  $x_i$  can be individually determined as  $U_i(R_i - C_i)$  when  $x_i$  is 0 or  $S_i(\sum_{\ell=1}^i U_\ell)$  when  $x_i$  is 1. Therefore, whether  $x_i$  should be set to 0 or 1 can be decided by individually comparing the two constants  $U_i(R_i - C_i)$  and  $S_i(\sum_{\ell=1}^i U_\ell)$ . Eq. (23) is therefore minimized when  $x_i = 1$  if  $U_i(R_i - C_i) > S_i(\sum_{\ell=1}^i U_\ell)$  and when  $x_i = 0$  otherwise. We denote the resulting vector by  $\vec{x}^{lin}$ , where, for each higher-priority task  $\tau_i$ ,

$$x_i^{lin} = \begin{cases} 1 & \text{if } U_i(R_i - C_i) > S_i(\sum_{\ell=1}^i U_\ell) \\ 0 & \text{otherwise} \end{cases} \quad (24)$$

The following properties directly follow.

**Property 2.** For any  $t > 0$ , the vector assignment  $\vec{x}^{lin}$  minimizes the solution to Eq. (23) among all  $2^{k-1}$  possible vector assignments.

**Theorem 4.** Let  $rbf_k^{lin}(t, \vec{x})$  be the left hand side of Eq. (23). Task  $\tau_k$  is schedulable under fixed-priority if

$$rbf_k(D_k, \vec{x}^{lin}) \leq D_k. \quad (25)$$

*Proof:* It directly follows from Corollary 1 and the fact that, by construction, Eq. (23) upper bounds Eq. (5). ■

**Property 3.** The time complexity of both deriving  $\vec{x}^{lin}$  and testing Eq. (23) is  $O(k)$ .

## VIII. EXPERIMENTS

In this section, we present experiments conducted on randomly generated task sets. Five schedulability tests for dynamic self-suspending tasks are compared, namely, the suspension oblivious approach (Section III-A), the modeling of suspension as a release jitter (Section III-B), the analysis proposed by Jane W.S. Liu and proven correct in this paper that models the suspension as blocking (Section III-C), the generic framework of Corollary 1 (called ECRTS 16 in the plots) and the schedulability test used Theorem 1 based on the vector defined in Eq. (24) in Section VII (called ECRTS 16 linear in the plots). In those experiments, the tasks are assumed to be scheduled with rate monotonic and have implicit deadlines (i.e.,  $D_i = T_i$ ).

The task sets were generated using the `randfixedsum` algorithm presented in [7]. Let  $C'_i$  denote the sum of  $C_i$  and  $S_i$  (i.e.,  $C'_i \stackrel{\text{def}}{=} C_i + S_i$ ). That is,  $C'_i$  is the WCET of a task for which the suspension time would be considered as execution time like it is the case in the test of Section III-A. The modified utilization of  $\tau_i$  is then given by  $U'_i \stackrel{\text{def}}{=} \frac{C'_i}{T_i}$  and the total modified utilization is  $U' \stackrel{\text{def}}{=} \sum_{i=1}^n U'_i$ . The task generator uses the `randfixedsum` algorithm to generate  $n$  values  $U'_i$  (one for each task) with total modified utilization  $U'$ . A period  $T_i$  is then randomly generated from a uniform distribution spanning from 100 to 10000. The resulting value  $C'_i = U'_i \times T_i$  is then divided in the two components  $C_i$  and  $S_i$  using a random ratio  $r_i$  obtained from a uniform distribution between a value  $r_{\min}$  and  $r_{\max}$  depending of the specific experiment performed. That is,  $S_i \stackrel{\text{def}}{=} r_i \times C'_i$  and  $C_i = (1 - r_i) \times C'_i$ .

Each point in the plots of Figure 4 represents the number of task sets that were deemed schedulable by the respective algorithm over 1000 different experiments.

Four different types of experiments are reported in this paper. The first one is presented in Figure 4a. It presents the evolution of the number of task sets deemed schedulable when the number of self-suspending tasks increases. The number of tasks  $n$  is varied from 4 to 10 for a total modified utilization  $U'$  of 0.95. As can be seen in Figure 4a, at the exception of the suspension oblivious analysis, the performances of the tests are barely influenced by the number of tasks. In fact, the number of task sets found schedulable by the test of Corollary 1 and the linear test of Section VII slightly increases with the number of tasks. It is the opposite behavior than the suspension oblivious

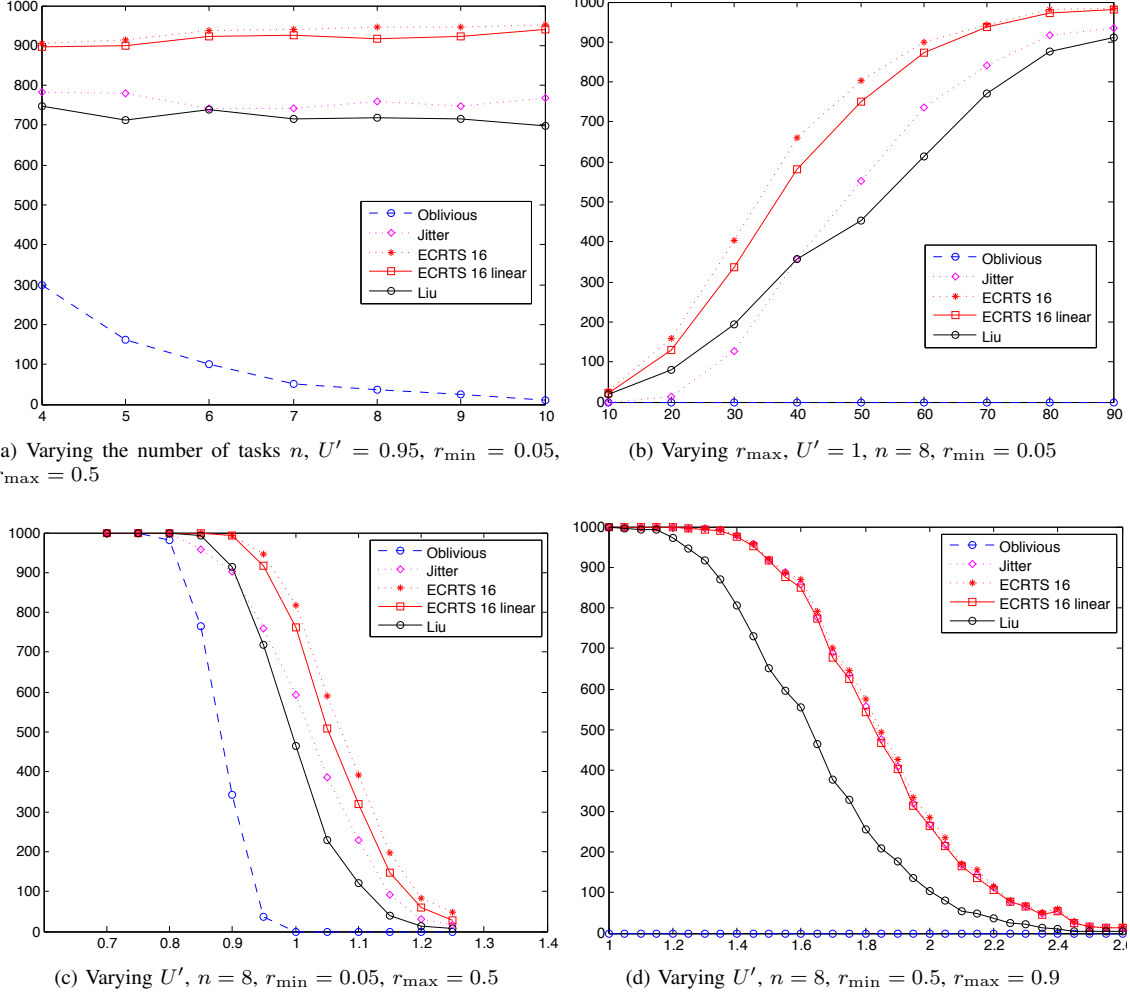


Fig. 4: Number of schedulable task sets over 1000 randomly generated task sets.

approach. One can already conclude from this plot that the tests developed in this paper perform way better than the state-of-the-art. Furthermore, the difference between the performances of the complete test of Corollary 1 and its linear version are quite small, thereby making the linear test a practical and useful analysis.

The second experiment is presented in Figure 4b and shows the evolution of the performances of the tests with respect to the length of the total suspension time of a task when the total modified utilization  $U'$  and the number of tasks are kept constant. The value of  $r_{\max}$  is then varied from 10% to 90%, hence increasing the number of tasks with high suspension times. The value  $r_{\min}$  is kept constant at 5%, so as to keep a certain diversity in the suspension behavior of each task. As expected, the suspension oblivious approach does not accept any task set since the total modified utilization is equal to 100%. For the other tests however, the number of schedulable task sets increases when the suspension times become larger. Indeed, the actual workload, which accounts only for the WCET  $C_i$ , decreases when  $S_i$  increases. Again, one can see the improvement of the tests of this paper over the state-of-the-art. Interestingly, one can also witness the incomparability of

the jitter based and the blocking based schedulability tests. The jitter based test performs better for large blocking times while the blocking based test has better results for lower blocking times.

The last two plots (Figures 4c and 4d), present the results obtained when the total modified utilization increases but the distribution of suspension times and the number of tasks remain identical. As expected, the number of schedulable task sets decreases when the utilization increases. Again, the incomparability of the jitter- and blocking-based tests can be seen. The test based on blocking usually performs better for lower utilizations. The improvement of Corollary 1 over the state-of-the-art is still high when suspension times are in average smaller than the execution times of the tasks (see Figures 4c). However, when the suspension time becomes larger than the execution time of the task (see Figures 4d), the release jitter-based test performs almost as well as Corollary 1.

## IX. CONCLUSION

**Acknowledgement:** This paper is supported by DFG, as part of the Collaborative Research Center SFB876 (<http://sfb876.tu-dortmund.de/>). This work was also partially supported by National Funds through FCT/MEC (Portuguese Foundation for Science and Technology) and co-financed by ERDF (European Regional Development Fund) under the PT2020 Partnership, within project UID/CEC/04234/2013 (CISTER); also by FCT/MEC and the EU ARTEMIS JU within project(s) ARTEMIS/0003/2012 - JU grant nr. 333053 (CONCERTO) and ARTEMIS/0001/2013 - JU grant nr. 621429 (EMC2).

## REFERENCES

- [1] N. C. Audsley and K. Bletsas. Fixed priority timing analysis of real-time systems with limited parallelism. In *16th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 231–238, 2004.
- [2] N. C. Audsley and K. Bletsas. Realistic analysis of limited parallel software / hardware implementations. In *10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 388–395, 2004.
- [3] K. Bletsas, N. Audsley, W.-H. Huang, J.-J. Chen, and G. Nelissen. Errata for three papers (2004-05) on fixed-priority scheduling with self-suspensions. Technical Report CISTER-TR-150713, CISTER, July 2015.
- [4] B. Brandenburg. Improved analysis and evaluation of real-time semaphore protocols for P-FP scheduling. In *RTAS*, 2013.
- [5] A. Carminati, R. de Oliveira, and L. Friedrich. Exploring the design space of multiprocessor synchronization protocols for real-time systems. *Journal of Systems Architecture*, 60(3):258–270, 2014.
- [6] J.-J. Chen, W.-H. Huang, and G. Nelissen. A note on modeling self-suspending time as blocking time in real-time systems. Technical report, 2015.
- [7] P. Emberson, R. Stafford, and R. I. Davis. Techniques for the synthesis of multiprocessor tasksets. In *WATERS Workshop*, pages 6–11, 2010.
- [8] G. Han, H. Zeng, M. Natale, X. Liu, and W. Dou. Experimental evaluation and selection of data consistency mechanisms for hard real-time applications on multicore platforms. *IEEE Transactions on Industrial Informatics*, 10(2):903–918, 2014.
- [9] W.-H. Huang, J.-J. Chen, H. Zhou, and C. Liu. PASS: Priority assignment of real-time tasks with dynamic suspending behavior under fixed-priority scheduling. In *Design Automation Conference (DAC)*, 2015.
- [10] W.-H. Huang, J.-J. Chen, H. Zhou, and C. Liu. PASS: Priority assignment of real-time tasks with dynamic suspending behavior under fixed-priority scheduling. In *Proceedings of the 52nd Annual Design Automation Conference on - DAC15*. Association for Computing Machinery (ACM), 2015.
- [11] W. Kang, S. Son, J. Stankovic, and M. Amirjoo. I/O-Aware Deadline Miss Ratio Management in Real-Time Embedded Databases. In *Proc. of the 28th IEEE Real-Time Systems Symp.*, pages 277–287, 2007.
- [12] S. Kato, K. Lakshmanan, A. Kumar, M. Kelkar, Y. Ishikawa, and R. Rajkumar. RGEM: A Responsive GPGPU Execution Model for Runtime Engines. In *2011 IEEE 32nd Real-Time Systems Symposium*, 2011.
- [13] H. Kim, S. Wang, and R. Rajkumar. vMPCP: a synchronization framework for multi-core virtual machines. In *RTSS*, 2014.
- [14] I. Kim, K. Choi, S. Park, D. Kim, and M. Hong. Real-time scheduling of tasks that contain the external blocking intervals. In *RTCSA*, pages 54–59, 1995.
- [15] K. Lakshmanan, D. De Niz, and R. Rajkumar. Coordinated task scheduling, allocation and synchronization on multiprocessors. In *RTSS*, 2009.
- [16] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20(1):46–61, jan 1973.
- [17] J. W. S. W. Liu. *Real-Time Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2000.
- [18] W. Liu, J.-J. Chen, A. Toma, T.-W. Kuo, and Q. Deng. Computation Offloading by Using Timing Unreliable Components in Real-Time Systems. In *Proceedings of the The 51st Annual Design Automation Conference on Design Automation Conference (DAC)*, 2014.
- [19] L. Ming. Scheduling of the inter-dependent messages in real-time communication. In *Proc. of the First International Workshop on Real-Time Computing Systems and Applications*, 1994.
- [20] R. Rajkumar, L. Sha, and J. P. Lehoczky. Real-time synchronization protocols for multiprocessors. In *Proceedings of the 9th IEEE Real-Time Systems Symposium (RTSS '88)*, pages 259–269, 1988.
- [21] M. Yang, H. Lei, Y. Liao, and F. Rabee. PK-OMLP: An OMLP based k-exclusion real-time locking protocol for multi- GPU sharing under partitioned scheduling. In *DASC*, 2013.
- [22] M. Yang, H. Lei, Y. Liao, and F. Rabee. Improved blocking time analysis and evaluation for the multiprocessor priority ceiling protocol. *Journal of Computer Science and Technology*, 29(6):1003–1013, 2014.
- [23] H. Zeng and M. Natale. Mechanisms for guaranteeing data consistency and flow preservation in AUTOSAR software on multi-core platforms. In *SIES*, 2011.