CISTER
Research Center in
Real-Time & Embedded
Computing Systems

Conference Paper

Timing Analysis of Fixed Priority Self-Suspending Sporadic Tasks

Geoffrey Nelissen*
José Fonseca*
Gurulingesh Raravi
Vincent Nélis*

*CISTER Research Center

CISTER-TR-150506

2015/07/07

Timing Analysis of Fixed Priority Self-Suspending Sporadic Tasks

Geoffrey Nelissen*, José Fonseca*, Gurulingesh Raravi, Vincent Nélis*

*CISTER Research Center

Polytechnic Institute of Porto (ISEP-IPP)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8321159

E-mail: grrpn@isep.ipp.pt, jcnfo@isep.ipp.pt, guhri@isep.ipp.pt, nelis@isep.ipp.pt

http://www.cister.isep.ipp.pt

Abstract

Many real-time systems include tasks that need to suspend their execution in order to externalize some of their operations or to wait for data, events or shared resources. Although commonly encountered in real-world systems, study of their timing analysis is still limited due to the problem complexity. In this paper, we invalidate a claim made in one of the earlier works [1], that led to the common belief that the timing analysis of one self-suspending task interacting with non-self-suspending sporadic tasks is much easier than in the periodic case. This work highlights the complexity of the problem and presents a method to compute the exact worst-case response time (WCRT) of a self-suspending task with one suspension region. However, as the complexity of the analysis might rapidly grow with the number of tasks, we also define an optimization formulationto compute an upper-bound on the WCRT for tasks with multiple suspendion regions. In the experiments, our optimization framework outperforms all previous analysis techniques and oftenfinds the exact WCRT.

Timing Analysis of Fixed Priority Self-Suspending Sporadic Tasks

Geoffrey Nelissen*, José Fonseca*, Gurulingesh Raravi† and Vincent Nélis*
*CISTER/INESC-TEC, ISEP, Polytechnic Institute of Porto, Portugal
†Xerox Research Center India, Bengaluru
Email: *{grrpn, jcnfo, nelis}@isep.ipp.pt, †gurulingesh.raravi@xerox.com

***Abstract*—Many real-time systems include tasks that need to suspend their execution in order to externalize some of their operations or to wait for data, events or shared resources. Although commonly encountered in real-world systems, study of their timing analysis is still limited due to the problem complexity. In this paper, we invalidate a claim made in one of the earlier works [1], that led to the common belief that the timing analysis of one self-suspending task interacting with non-self-suspending sporadic tasks is much easier than in the periodic case. This work highlights the complexity of the problem and presents a method to compute the exact worst-case response time (WCRT) of a self-suspending task with one suspension region. However, as the complexity of the analysis might rapidly grow with the number of tasks, we also define an optimization formulation to compute an upper-bound on the WCRT for tasks with multiple suspendion regions. In the experiments, our optimization framework outperforms all previous analysis techniques and often finds the exact WCRT.**

I. Introduction

Real-time tasks often involve processing operations which may take considerable time to finish if executed solely on a generic purpose uniprocessor platform. System designers have been achieving significant improvement in the efficiency of these operations by offloading them to specialized hardware accelerators (e.g., Graphical or Network Processing Units), leaving the main processor available for other tasks. The offloading phases represent suspension delays for the task that initiates them. Suspension delays can also be observed when tasks are synchronizing, exchanging data through communication interfaces, or accessing external shared resources such as I/O devices. All such tasks that may at some point in their execution suspend their computation to wait for external data, events or resources are called *self-suspending* tasks.

A self-suspending task is composed of a set of *execution regions* interleaved with *suspension regions*. Traditional real-time systems theory [2] has accounted the duration of suspension regions as part of the task worst-case execution time while doing timing analysis. However, if the suspension regions are lengthy, such suspension-oblivious analysis typically become very pessimistic, leading to severe utilization loss and possibly jeopardizing the schedulability of the system. Hence, recent works [1], [3]–[6] have focused on suspension-aware analysis techniques which explicitly model the suspension placement and durations in order to reduce the pessimism on the calculated worst-case interference exerted by higher priority tasks, thereby offering opportunities fora potential utilization improvement. Nevertheless, suspension-aware analysis are rather complex and sometimes assume the existence of additional operating system facilities (e.g., phase enforcement mechanisms or execution control policies) to ease the analysis and minimize the pessimism while considering the suspension regions.

In this work, we start by studying the timing analysis of self-suspending tasks, under fixed-priority scheduling, assuming that all the interfering tasks are non-self-suspending sporadic tasks. Contrary to what has been claimed in previous works, we show that it is not simple to characterize a critical instant even for this limited model. Based on this key observation, we identify the exponential number of scenarios that need to be considered, and we then present an algorithm to compute the exact worst-case response time (WCRT) for a self-suspending task with one suspension region. As the algorithm becomes intractable in function of the number of higher-priority tasks, we formulate a response time test for self-suspending tasks with multiple suspension regions as an optimization problem, which can be solved in reasonable time. This optimization problem is then extended for the response time analysis of multiple self-suspending sporadic tasks interfering with each other.

Related Work. The problem of analyzing the timing behavior and schedulability of self-suspending tasks has been extensively studied in previous works [1], [3]–[11]. Negative results on the feasibility problem of scheduling *periodic* self-suspending tasks on uniprocessors have been presented in [8], [9]. Schedulability analyses for self-suspending periodic tasks were proposed in [10], [11]. However, the pessimism in those analyses led to significant utilization loss. Recently, new schedulablity tests for synchronous self-suspending tasks with harmonic periods have shown to exhibit low utilization loss under Rate Monotonic (RM) policy [12].

The self-suspending sporadic task model studied in this paper has received considerable attention [1], [3], [7]. In [1], authors attempted to characterize the critical instant for self-suspending tasks with respect to the interference exerted by higher priority non-self-suspending tasks. It builds on the fact that sporadic tasks may delay their job releases, to prove that an *exact* characterization of the worst-case scheduling scenario leading to the WCRT of the self-suspending task is simpler to achieve in comparison to the periodic case. It therefore became a common belief that the exact WCRT of a self-suspending task co-running with sporadic non-self-suspending tasks can be obtained in pseudo-polynomial time. However, as discussed in Section III, the worst-case release pattern identified in [1] is incorrect.

Still on sporadic tasks, early research [7] considered fixed-priority scheduling of limited parallel systems, where parts of a process are executed in parallel in software and hardware, and therefore can be modeled as a set of self-suspending sporadic tasks. The authors introduced the notion of synthetic worst-case execution distribution for higher-priority tasks and derived upper-bounds on the WCRT. Response time analysis for a segment-fixed priority scheduling scheme, which assigns different priorities to each computing segment and enforces phase offsets to predict the different segment's releases, were

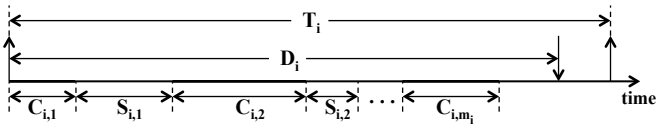


Fig. 1: Constrained-deadline self-suspending sporadic task.

developed in [3]. In the same paper, the effectiveness of RM scheduling for self-suspending periodic tasks with specific properties was demonstrated.

Lately, there has also been relevant work on global scheduling of self-suspending tasks on multiprocessor, particularly for soft real-time systems [4]–[6]. In [6], the first suspension-aware schedulability analysis for self-suspending sporadic tasks in a multi-core hard-real setting was presented. Unfortunately the schedulability test provided in that paper for global fixed-priority scheduling is not entirely correct and may often lead to optimistic (unsafe) results [13].

This Research. In this paper, we consider the problem of computing the WCRT of a self-suspending task on a uniprocessor platform, under a sporadic task system. Towards this end, we make the following contributions: (i) we discuss a couple of misconceptions about the timing analysis of self-suspending tasks by showing that a well-accepted claim from an earlier work [1] is incorrect; (ii) we propose an algorithm for computing the exact WCRT of a self-suspending with a single suspension region; (iii) we propose a Mixed Integer Linear Programming (MILP) based technique for finding an upper-bound on the WCRT of a self-suspending task having multiple suspension regions; (iv) we extend this MILP formulation to analyze the WCRTs of multiple self-suspending sporadic tasks running on the same platform; and (v) we present the evaluation results comparing the performance of our approaches with prior state-of-the-art analyses.

II. System model

Consider a task set $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$ of n constrained-deadline self-suspending sporadic tasks scheduled on a single processor. Each task τ_i releases a (potentially infinite) sequence of *jobs*, with the first job released at any time during the system execution and subsequent jobs released *at least* T_i (referred to as *minimum inter-arrival time*) time units apart. Each job released by τ_i has to complete its execution within $D_i \leq T_i$ (referred to as τ_i's *deadline*) time units from its release.

A task τ_i consists of $m_i \geq 1$ *execution regions* and $m_i - 1$ *suspension regions* such that any two consecutive execution regions are separated by a suspendion region as shown in Fig. 1. Formally, each self-suspending task τ_i is characterized as follows: $\tau_i \stackrel{\text{def}}{=} \langle (C_{i,1}, S_{i,1}, C_{i,2}, S_{i,2}, \ldots, S_{i,m_i-1}, C_{i,m_i},), D_i, T_i \rangle$ where (i) $C_{i,j}$ denotes the worst-case execution time of the jth execution region; (ii) $S_{i,j}$ denotes the worst-case duration of the jth suspension region; (iii) m_i denotes the number of execution regions separated by $m_i - 1$ suspension regions; (iv) $D_i \leq T_i$ denotes the deadline before which all the execution regions must finish their execution and (v) T_i denotes the minimum inter-arrival time between two successive jobs of τ_i. In this paper, we call *non-self-suspending task*, a task with no suspension regions. A non-self-suspending task τ_k is represented as: $\tau_k \stackrel{\text{def}}{=} \langle (C_{k,1}), D_k, T_k \rangle$.

We assume that a fixed-priority scheduling policy is used to schedule the tasks on the processor. For convenience, we denote by $\tau_{i,j}$ the jth execution region of task τ_i, and the overall worst-case execution time of τ_i is defined as $C_i \stackrel{\text{def}}{=} \sum_{j=1}^{m_i} C_{i,j}$. The first execution region $\tau_{i,1}$ of a job of task τ_i becomes ready for execution (also referred to as the *arrival time* of $\tau_{i,1}$ and denoted by $a_{i,1}$) as soon as that job of task τ_i is released. The response time $R_{i,1}$ of the execution region $\tau_{i,1}$ is the difference between its *completion time* (denoted by $f_{i,1}$) and the arrival time of the job; formally, $R_{i,1} = f_{i,1} - a_{i,1}$. For $2 \leq j \leq m_i$, the execution region $\tau_{i,j}$ of a job of task τ_i becomes ready for execution at time $a_{i,j} \stackrel{\text{def}}{=} f_{i,j-1} + s_{i,j-1}$ (where $s_{i,j-1} \leq S_{i,j-1}$ is the duration of the $(j-1)^{\text{th}}$ self-suspension region) and its response time $R_{i,j}$ is given by the difference between its completion time and its arrival time; formally, $R_{i,j} = f_{i,j} - a_{i,j}$. The response time R_i of a job of task τ_i is the sum of the response times of all its execution regions, plus the sum of the duration of all its suspension regions, that is, $R_i \stackrel{\text{def}}{=} \sum_{j=1}^{m_i} R_{i,j} + \sum_{j=1}^{m_i-1} s_{i,j}$. Finally, the *worst-case response time* WCRT_i *of a task* τ_i is defined as the largest response time that any job of τ_i may ever experience.

In Sections III to V, we consider the case in which the task set τ has only one self-suspending task and all the other tasks are non-self-suspending. The self-suspending task is denoted τ_{ss} and has the lowest priority, i.e., all the non-self-suspending tasks in τ have a higher priority than τ_{ss}. We denote the set of higher priority non-self-suspending tasks as $\text{hp}(\tau_{ss})$. The restriction of having only one self-suspending task in the taskset is relaxed in Section VI.

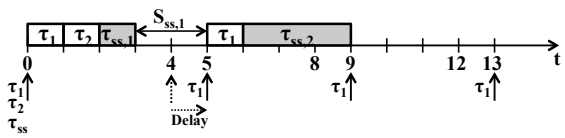
III. Misconceptions about timing analysis of self-suspending tasks

Let us assume that there is only one self-suspending task τ_{ss} in the taskset τ. This task has the lowest priority and suffers interference from a set $\text{hp}(\tau_{ss})$ of higher priority non-self-suspending sporadic tasks. The exact worst-case response time analysis of such a system was studied in [1] and was deemed solved. However, in this section, we prove that those results were based on a couple of wrong observations (which have established themselves as facts over the years), namely (i) that the worst-case interference suffered by τ_{ss} is generated when all the higher priority tasks are released synchronously with τ_{ss}, and (ii) that releasing the jobs of higher priority tasks as often as possible (respecting the minimum inter-arrival times) in each execution region maximizes the overall interference on the self-suspending task. Unfortunately, although intuitive, these observations are incorrect and have led to flawed analyses [1].

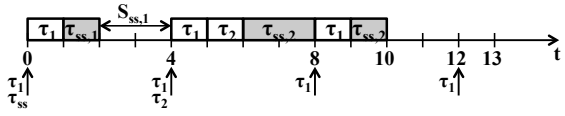
A. On the synchronous release with the first execution region

Exact worst-case response time analysis are based on the notion of *critical instant*. The critical instant for a task τ_i is defined as an instant at which a request for that task will have the largest response time. Since the response time of a task is dependent on the higher priority tasks, a critical instant for a task τ_i is generally concerned with the release pattern of higher priority tasks.

In [1], Lakshmanan et. al. argue that the release pattern Φ_{ss} is a critical instant for a self-suspending task τ_{ss}, where Φ_{ss} is defined as follows:



(a) Scenario 1. Response-time analysis when the job release pattern *is* Φ_{ss}.



(b) Scenario 2. Response-time analysis when the job release pattern *is not* Φ_{ss}.

Fig. 2: Counter-example to Φ_{ss} being the critical instant of τ_{ss}.

- every higher priority non-self-suspending task $\tau_h \stackrel{\text{def}}{=} \langle (C_h), D_h, T_h \rangle$ is released simultaneously with τ_{ss};
- jobs of τ_h eligible to be released during any j^{th} $(1 \leq j < m_i)$ suspension region of τ_{ss} are delayed to be aligned with the release of the subsequent $(j+1)th$ execution region of τ_{ss}; and
- all remaining jobs of τ_h are released every T_h.

We prove with a counter-example that Φ_{ss} is not a critical instant for a self-suspending task τ_{ss}.

Lemma 1. *The worst-case response time of task τ_{ss} is not given by Φ_{ss}.*

Proof: Consider a task set $\tau = \{\tau_1, \tau_2, \tau_{ss}\}$ of three constrained-deadline sporadic tasks scheduled on a single processor. τ_1 and τ_2 are non-self-suspending tasks and τ_{ss} is a self-suspending task. Let the characteristics of these tasks be as follows: $\tau_1 \stackrel{\text{def}}{=} \langle (1), 4, 4 \rangle$; $\tau_2 \stackrel{\text{def}}{=} \langle (1), 100, 100 \rangle$ and $\tau_{ss} \stackrel{\text{def}}{=} \langle (1,2,3), 1000, 1000 \rangle$. Let the priorities of the tasks be assigned using the RM policy (i.e., smaller the period, higher the priority); this implies that task τ_1 has the highest priority and τ_{ss} the lowest. Let us compute the response time of task τ_{ss} considering two different job release patterns: (i) a job release pattern Φ_{ss} compliant with its definition made in [1] and (ii) a job release pattern different than Φ_{ss}. We show that there exists a job release pattern which *is not* Φ_{ss} and for which the response-time of task τ_{ss} is larger than its response time when the job release pattern is Φ_{ss}.

Scenario 1. Let us consider the job release pattern Φ_{ss} as shown in Fig. 2a. Using the standard response-time equation, we obtain $R_{ss,1} = 3$ for the execution region $\tau_{ss,1}$ and $R_{ss,2} = 4$ for the execution region $\tau_{ss,2}$ (see Fig. 2a). Hence, under the release pattern Φ_{ss}, the response-time of task τ_{ss} is given by: $R_{ss} = R_{ss,1} + S_{ss,1} + R_{ss,2} = 3 + 2 + 4 = 9$.

Scenario 2. Let us consider a job release pattern as shown in Fig. 2b. Observe that this release pattern is not Φ_{ss} since task τ_2 is not released synchronously with task τ_{ss}. Using the same standard response-time equation, we obtain a response time $R_{ss,1} = 2$ for the execution region $\tau_{ss,1}$ and $R_{ss,2} = 6$ for the execution region $\tau_{ss,2}$ (see Fig. 2b). Hence, under a release pattern which is not Φ_{ss}, the response-time of task τ_{ss} is given by: $R_{ss} = 2 + 2 + 6 = 10$.

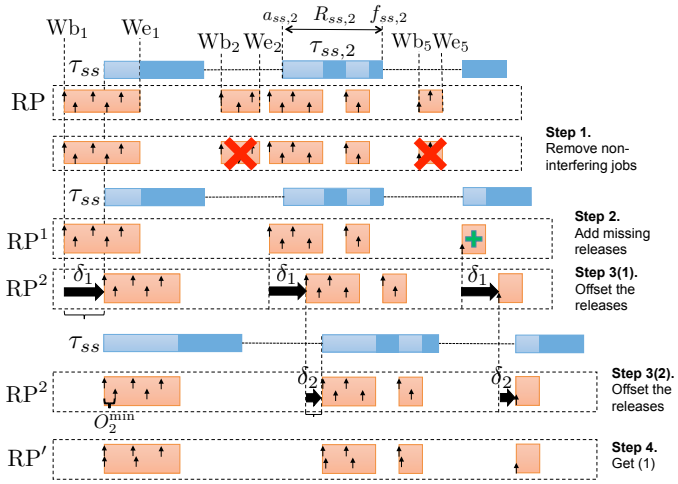


Fig. 3: Illustration of the proof of Lemma 2.

Clearly, the response-time of task τ_{ss} obtained in Scenario 2 is larger than the response-time of τ_{ss} obtained in Scenario 1. Hence, the claim of Lakshmanan et. al. [1] that Φ_{ss} is the critical instant for a self-suspending task τ_{ss} is incorrect. ■

We now prove properties about the job release pattern characterizing the critical instant of task τ_{ss}.

Lemma 2. *From any feasible release pattern* RP *of the tasks in* $\text{hp}(\tau_{ss})$*, we can construct a feasible release pattern* RP' *from* RP *such that:*

(1) *In* RP'*, at least one job of every task in* $\text{hp}(\tau_{ss})$ *is released synchronously with the release of an execution region of* τ_{ss}*;*

(2) RP' *entails a larger (or equivalent) response time of task* τ_{ss} *than* RP.

Proof: Let us assume that τ_{ss} is scheduled to execute concurrently with a set $\text{hp}(\tau_{ss})$ of higher priority tasks and suppose that those tasks are released according to the release pattern RP. We denote by Wb_k and We_k the beginning and end of the k^{th} time window during which only tasks in $\text{hp}(\tau_{ss})$ are executed. That is, τ_{ss} does not execute at all in the time intervals defined by $[\text{Wb}_k, \text{We}_k]$, $\forall k > 0$. Those intervals will be referred to as the *higher priority tasks busy windows*.

Fig. 3 (top part) shows these notations with a simple example that will be used throughout the proof to illustrate the process of creating RP' from RP. This example assumes that $\text{hp}(\tau_{ss})$ consists of three sporadic tasks. The interference by those tasks on the self-suspending task τ_{ss} is represented by light rectangles on the first line of Fig. 3. Dark rectangles correspond to the execution of the execution regions of τ_{ss}. The busy windows generated by tasks in $\text{hp}(\tau_{ss})$ are shown by arrow filled rectangles on the second line of Fig. 3. Note that only the jobs potentially contributing to the response time of τ_{ss} are depicted in Fig. 3.

First of all, we remove from RP all the releases from the tasks in $\text{hp}(\tau_{ss})$ that occur in a busy window $[\text{Wb}_k, \text{We}_k]$ that does not overlap with any execution region of τ_{ss} (see Step 1 in Figure 3). Note that removing these releases along with the execution of the corresponding jobs does not alter the schedule of τ_{ss} (i.e. it does not impact the response time of any of its

execution regions) or that of the jobs of any higher priority task released in any other busy window in RP. As a result, the response time of $\tau_{\rm ss}$ is not impacted by this modification of RP. We define the resulting release pattern as RP^1.

In order to get (1), each task in $\mathrm{hp}(\tau_{ss})$ must release at least one job in RP'. Since there may be some tasks in $\mathrm{hp}(\tau_{ss})$ that do not release a job in RP^1, one job release from each of those tasks is added to RP^1 such that it coincides with the arrival of the last execution region of $\tau_{\rm ss}$ (see Step 2 on Fig. 3). This transformation of RP^1 trivially increases the response time of the last execution region as compared to RP and consequently also increases the overall response time of $\tau_{\rm ss}$.

The next step to construct RP' from RP consists in considering all the execution regions of $\tau_{\rm ss}$ one-by-one, starting from $\tau_{{\rm ss},1}$, and for each execution region $\tau_{{\rm ss},j}$ do the following: if there is a busy window k such that $\mathrm{Wb}_k \leq a_{{\rm ss},j} \leq \mathrm{We}_k$ (i.e. $\tau_{{\rm ss},j}$ is released within $[\mathrm{Wb}_k, \mathrm{We}_k]$), we then compute the distance δ_j between the arrival of $\tau_{{\rm ss},j}$ and Wb_k, i.e. $\delta_j \stackrel{\text{def}}{=} a_{{\rm ss},j} - \mathrm{Wb}_k$. Note that by definition, $\delta_\ell \geq 0$. If such an overlap exists, we postpone all the higher priority job releases that occur at or after Wb_k by δ_j time units. This shift in the job releases makes δ_j additional units of workload from the tasks in $\mathrm{hp}(\tau_{ss})$ interfere with the execution of $\tau_{{\rm ss},j}$. As a consequence, the response time of $\tau_{{\rm ss},j}$ increases by δ_j (i.e. $R_{{\rm ss},j} \leftarrow R_{{\rm ss},j} + \delta_j$), and so does the time $f_{{\rm ss},j}$ at which it finishes its execution ($f_{{\rm ss},j} \leftarrow f_{{\rm ss},j} + \delta_j$) and, in a cascade effect, the times at which the next execution regions are released (i.e., $\forall \ell > j$, $a_{{\rm ss},\ell} \leftarrow a_{{\rm ss},\ell} + \delta_j$). Step 3(1) in Fig. 3) illustrates this process for the first region of $\tau_{\rm ss}$. At that step all the task releases are delayed by δ_1 time units. Then, Step 3(2) illustrates the second and last iteration of that process when the second execution region of $\tau_{\rm ss}$ is considered and all releases occurring at or after Wb_2 get postponed by δ_2 time units. For clarity, we have redrawn the interference pattern on $\tau_{\rm ss}$ resulting from that step.

Note that at each iteration of the transformation described above, the response time of the currently considered region $\tau_{{\rm ss},j}$ of $\tau_{\rm ss}$ increases by δ_j time units. However, given that along with this increase, we also delay by δ_j time units the release of all the subsequent execution regions of $\tau_{\rm ss}$ and the releases of all the jobs of the tasks in $\mathrm{hp}(\tau_{ss})$ that interfere with those regions, there is no variation in the interference suffered by those execution regions and their response time is not impacted by the transformation. After each iteration, the overall response time of $\tau_{\rm ss}$ therefore increases by δ_j time units only. The release pattern from this transformation is now referred to as RP^2. One can notice that all the jobs in RP^2 are released within the execution regions of $\tau_{\rm ss}$.

As already explained, the response time of every region of $\tau_{\rm ss}$ may have only increased (sometimes it remains the same) during the process of constructing RP^2 as described above. Finally, in order to obtain RP', RP^2 is further modified as follows. For each task $\tau_h \in \mathrm{hp}(\tau_{ss})$, let $\mathcal{R}_h$ denote the set of all its release time-instants in the pattern RP^2. We know that for each of these instants $\mathrm{rel}_{h,x}$ there exists a execution region $\tau_{{\rm ss},j}$ of $\tau_{\rm ss}$ such that $a_{{\rm ss},j} \leq \mathrm{rel}_{h,x} < f_{{\rm ss},j}$, i.e. that release of τ_j happens while there is a execution region $\tau_{{\rm ss},j}$ of $\tau_{\rm ss}$ which is running or waiting for the CPU. Now, for each release in $\mathcal{R}_h$, we compute the offset $O_{h,x}$ of $\mathrm{rel}_{h,x}$ relative to the release of the execution region of $\tau_{\rm ss}$ which is active at that time. That is, for each $\mathrm{rel}_{h,x} \in \mathcal{R}_h$ we compute $O_{h,x} \stackrel{\text{def}}{=} \mathrm{rel}_{h,x} - a_{{\rm ss},j}$ where j is such that $a_{{\rm ss},j} \leq \mathrm{rel}_{h,x} < f_{{\rm ss},j}$. We then compute the minimum offset $O_h^{\min}$ for τ_h such that $O_h^{\min} \stackrel{\text{def}}{=} \min_{\forall x}\{O_{h,x}\}$ and shift to the left all the releases of τ_h by that minimum offset, i.e. for all $\mathrm{rel}_{h,x} \in \mathcal{R}_h$, we impose $\mathrm{rel}_{h,x} \leftarrow \mathrm{rel}_{h,x} - O_h^{\min}$. As a result, none of the releases of τ_h exit its "encompassing" $\tau_{\rm ss}$'s execution region and, as a consequence, the interference on $\tau_{\rm ss}$ is not modified when passing from RP^2 to RP'. Moreover, because the releases of all the jobs of τ_h are shifted by the same amount of time, the minimum inter-arrival time between all those jobs is still respected. Finally, at least one job of each task $\tau_k \in \mathrm{hp}(\tau_{ss})$ is now synchronous with the release of an execution region of $\tau_{\rm ss}$ (the one[s] for which the relative offset was minimum, i.e. $O_{h,x} = O_h^{\min}$). This last step of the proof is depicted on the last line of Fig. 3 for the second task in $\mathrm{hp}(\tau_{ss})$. From the entire discussion, it can be seen that (1) and (2) hold true. Hence the lemma. ■

The previous lemma leads to the following corollary.

Corollary 1. *In the critical instant of a self-suspending task $\tau_{\rm ss}$, every higher priority task releases a job synchronously with the arrival of at least one execution region of $\tau_{\rm ss}$, although not all higher priority tasks must release a job synchronously with the same execution region.*

B. On maximizing the number of releases in each execution region

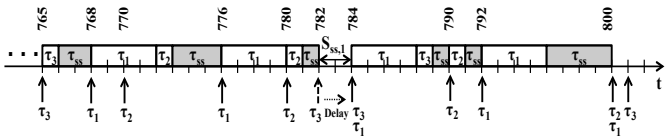
In the previous section, we proved that the critical instant for a self-suspending task $\tau_{\rm ss}$ suffering interference from non-self-suspending sporadic tasks happens when every higher priority task releases a job synchronously with at least one execution region of $\tau_{\rm ss}$. Let us now define a set of synchronous release constraints $\mathrm{Synch}^{\rm ss}$ as follows.

Definition 1 (Set of synchronous release constraints)**.** *Let $\mathrm{Synch}_j^{\rm ss}$ be a set of tasks in $\mathrm{hp}(\tau_{ss})$ that are constrained to release a job synchronously with the j^{th} execution region of $\tau_{\rm ss}$. Then, the set of synchronous release constraints $\mathrm{Synch}^{\rm ss}$ is the composition of the sets $\mathrm{Synch}_j^{\rm ss}$ associated with every execution region $\tau_{{\rm ss},j}$ of $\tau_{\rm ss}$. It thus represents the release constraints imposed to each of the tasks in $\mathrm{hp}(\tau_{ss})$ with respect to the execution regions of $\tau_{\rm ss}$.*

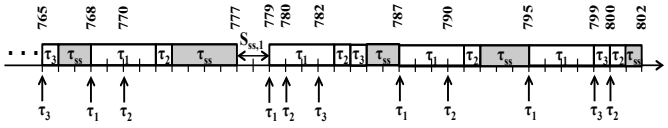
With the above definition, we now prove the counter-intuitive property that, even considering the set of synchronous releases that lead to the critical instant of $\tau_{\rm ss}$, the WCRT of $\tau_{\rm ss}$ is not always obtained when the higher priority tasks release as many jobs as possible in each execution region of $\tau_{\rm ss}$.

Lemma 3. *Let $\mathrm{Synch}^{\rm ss}$ be a set of synchronous release constraints on tasks in $\mathrm{hp}(\tau_{ss})$. Releasing the jobs of the tasks in $\mathrm{hp}(\tau_{ss})$ as often as possible in each execution region of $\tau_{\rm ss}$, while respecting the set of constraints $\mathrm{Synch}^{\rm ss}$, will not always lead to the WCRT of $\tau_{\rm ss}$ under $\mathrm{Synch}^{\rm ss}$.*

Proof: Consider a task set $\tau = \{\tau_1, \tau_2, \tau_3, \tau_{\rm ss}\}$ of 4 tasks in which τ_1, τ_2 and τ_3 are non-self-suspending sporadic tasks and $\tau_{\rm ss}$ is a self-suspending task with the lowest priority. The tasks are characterized as follows: $\tau_1 = \langle (4), 8, 8\rangle$, $\tau_2 = \langle (1), 10, 10\rangle$, $\tau_3 = \langle (1), 17, 17\rangle$ and $\tau_{\rm ss} = \langle (265, 2, 6), 1000, 1000\rangle$. The set $\mathrm{Synch}^{\rm ss}$ imposes τ_1 to release a job synchronously with the second execution region $\tau_{{\rm ss},2}$ of $\tau_{\rm ss}$ while τ_2 and τ_3 must release a job synchronously with $\tau_{{\rm ss},1}$.



(a) Scenario 1. Jobs are released as often as possible while respecting all the constraints on the synchronous releases.



(b) Scenario 2. Jobs are not released as often as possible.

Fig. 4: Example showing that releasing higher priority jobs as often as possible while respecting a set of synchronous release constraints $\mathrm{Synch}^{\mathrm{ss}}$ on tasks in $\mathrm{hp}(\tau_{ss})$ may not always cause the maximum interference on a self-suspending task τ_{ss}.

Consider two scenarios with respect to the job release pattern, always respecting the given synchronous release constraints. In Scenario 1, the jobs of the higher priority non-self-suspending tasks are released as often as possible in each execution region of τ_{ss}. In Scenario 2 however, one less job of task τ_1 is released in (and therefore interfere with) $\tau_{\mathrm{ss},1}$. Showing that the WCRT of the self-suspending task in Scenario 2 is higher than that of Scenario 1 proves the claim.

Scenario 1 is depicted in Fig. 4a, and Scenario 2 in Fig. 4b. The first 765 time units are omitted in both figures. This is mainly due to space constraint. Furthermore, in both scenarios the release and schedule of the jobs is identical in this time window. A first job of τ_1, τ_2 and τ_3 is released synchronously with the arrival of the first execution region of τ_{ss} at time 0. The subsequent jobs of these three tasks are released as often as possible respecting the minimum inter-arrival times of the respective tasks. That is, they are released periodically with periods T_1, T_2 and T_3, respectively. With this release pattern, it is easy to compute that the 97^{th} job of τ_1 is released at time 768, the 78^{th} job of τ_2 at time 770 and the 46^{th} job of τ_3 at time 765. As a consequence, at time 765, τ_{ss} has finished executing 259 time units of its first execution segment out of 265 (indeed, we have $765 - 96 \times 4 - 77 \times 1 - 45 \times 1 = 259$) in both scenarios. From time 765 onwards, we separately consider Scenario 1 and 2.

Scenario 1. Continuing the release of jobs of the non-self-suspending tasks as often as possible without violating their minimum inter-arrival times, the first execution region $\tau_{\mathrm{ss},1}$ of τ_{ss} finishes its execution at time 782 as shown in Fig. 4a. After completion of its first execution region, τ_{ss} self-suspends for two time units until time 784. As τ_3 would have released a job just after the completion of $\tau_{\mathrm{ss},1}$, we delay the release of that job from time 782 to 784 in order to maximize the interference exerted by τ_3 on the second execution region of τ_{ss} as shown in Fig. 4a. Note that, in order to respect its minimum inter-arrival time, τ_2 has an offset of 6 time units with the arrival of the second execution region of τ_{ss}. Upon following the rest of the schedule, it can easily be seen that the job of τ_{ss} finishes its execution at time 800.

Scenario 2. As shown on Fig. 4b, the release of a job of task τ_1 is skipped at time 776 in comparison to Scenario 1. As a result, the execution of $\tau_{\mathrm{ss},1}$ is completed at time 777, thereby causing one job of τ_2 that was released at time 780 in Scenario 1, to *not* be released during the execution of the first execution region of τ_{ss} in Scenario 2. The response time of $\tau_{\mathrm{ss},1}$ is thus reduced by $C_1 + C_2 = 5$ time units in comparison to Scenario 1 (see Fig. 4). Note that this deviation from Scenario 1 still allows us to respect the synchronous release constraints imposed by $\mathrm{Synch}^{\mathrm{ss}}$, as we can release the next job of τ_1 synchronously with the second execution region of τ_{ss} without violating the minimum inter-arrival time of τ_1. The next job of τ_3 however, is not released in the suspension region anymore but 3 time units after the arrival of $\tau_{\mathrm{ss},2}$. Moreover, the offset of τ_2 with respect to the start of the second execution region is reduced by $C_1 + C_2 = 5$ time units. This causes an extra job of τ_2 to be released in the second execution region of τ_{ss}, initiating a cascade effect: an extra job of τ_1 is released in $\tau_{\mathrm{ss},2}$, which in turn causes the release of an extra job of τ_3, itself causing the arrival of one more job of τ_2 in the second execution region of τ_{ss}. Consequently, the response time of $\tau_{\mathrm{ss},2}$ increases by $C_2 + C_1 + C_3 + C_2 = 7$ time units. Overall, the response time of τ_{ss} increases by $7 - 5 = 2$ time units in comparison to Scenario 1. This is reflected in Figure 4b as the job of τ_{ss} finishes its execution at time 802.

This counter-example proves that the response time of a self-suspending task τ_{ss} can be larger when the tasks in $\mathrm{hp}(\tau_{ss})$ do not release jobs as often as possible. ■

Theorem 1. *The WCRT of τ_{ss} is not always obtained when the tasks in $\mathrm{hp}(\tau_{ss})$ release their jobs as often as possible in the execution regions of τ_{ss} under any set of constraints $\mathrm{Synch}^{\mathrm{ss}}$ on their synchronous releases.*

Proof: Using the task set τ of the counter-example provided in Lemma 3, one can check that the response-time obtained for τ_{ss} when releasing jobs as often as possible, while respecting any combination of constraints on the synchronous releases of the tasks in $\mathrm{hp}(\tau_{ss})$, never exceeds 800 (note that 4 of the 8 possible combinations are already covered by Scenario 1 of the Fig. 4 since τ_1 and τ_3 have a synchronous release with both execution regions of τ_{ss}). However, it was shown in the proof of Lemma 3 that a response time of 802 can be experienced by τ_{ss} when the release of one job of τ_1 is delayed. This proves the theorem. ■

IV. Exact WCRT for a self-suspending task with one self-suspending region

In this section, we restrict our analysis to the special case of a self-suspending task τ_{ss} composed of only two execution regions and one suspension region. We propose an algorithm to compute the exact worst-case response time of such a task. Self-suspending tasks with multiple suspension regions will be considered in the next section.

As proven in Lemma 2, the critical instant for τ_{ss} happens when a job of every higher priority task is released synchronously with the release of the first and/or second execution region of τ_{ss}. However, since we do not know a priori which combination of synchronous releases corresponds to the critical instant, there is no other solution for an exact WCRT analysis than considering all the possible combinations of synchronous releases. The exact WCRT for τ_{ss} is thus given by the maximum response time obtained with any of these combinations. Consequently, the WCRT analysis problem boils down to the subproblem of computing the WCRT of τ_{ss} when

the higher priority tasks in $\mathrm{hp}(\tau_{ss})$ are constrained to have a synchronous release with a specific execution region of τ_{ss}. We will later refer to that subproblem as the *"constrained releases subproblem"*.

A. Discussion on the constrained releases subproblem

Let us consider a self-suspending task τ_{ss}, a set of higher priority tasks $\mathrm{hp}(\tau_{ss})$ and a subset $\mathrm{Synch}_2^{\mathrm{ss}}$ of $\mathrm{hp}(\tau_{ss})$ containing tasks constrained to have a synchronous release with the second execution region of τ_{ss}. The WCRT of the first execution region $\tau_{\mathrm{ss},1}$ of τ_{ss} under those circumstances can be computed as follows

$$R_{ss,1} = C_{ss,1} + \sum_{\forall k \in \mathrm{hp}(\tau_{ss})} \mathrm{NI}_k \times C_k \tag{1}$$

where NI_k is the maximum number of jobs of τ_k interfering with τ_{ss}. According to the usual response time analysis for sporadic tasks with fixed priorities, NI_k is subject to the following constraint

$$\mathrm{NI}_k \leq \left\lceil \frac{R_{ss,1}}{T_k} \right\rceil \tag{2}$$

Furthermore, for the higher priority tasks that are constrained to have a synchronous release with $\tau_{\mathrm{ss},2}$, one must ensure that

$$\forall \tau_k \in \mathrm{Synch}_2^{\mathrm{ss}},\ \ \mathrm{NI}_k \times T_k \leq R_{ss,1} + S_{ss,1} \tag{3}$$

in order to respect the minimum inter-arrival time T_k of τ_k. That is, for every higher priority task τ_k that has a synchronous release with $\tau_{\mathrm{ss},2}$, the release of its last job interfering with $\tau_{\mathrm{ss},1}$, happening at time $(\mathrm{NI}_k - 1) \times T_k$, and the beginning of $\tau_{\mathrm{ss},2}$ at time $R_{ss,1} + S_{ss,1}$, have to be separated by at least T_k time units.

As a consequence of those constraints, the following equation can be used for NI_k and substituted in Eq. (1)

$$\mathrm{NI}_k = \begin{cases} \min\left(\left\lceil \frac{R_{ss,1}}{T_k} \right\rceil, \left\lfloor \frac{R_{ss,1}+S_{ss,1}}{T_k} \right\rfloor\right) & \text{if } \tau_k \in \mathrm{Synch}_2^{\mathrm{ss}} \\ \left\lceil \frac{R_{ss,1}}{T_k} \right\rceil & \text{otherwise} \end{cases} \tag{4}$$

When combined with Eq. (4), Eq. (1) becomes recursive. This kind of equation is usually solved using a fixed point iteration on $R_{ss,1}$. However, as shown in the example below, contrary to the traditional WCRT analysis of non-self-suspending sporadic tasks, the first solution found to this equation by increasing the value of $R_{ss,1}$ until it converges may yield to an optimistic (and unsafe) value.

Example 1. *Consider a self-suspending task τ_{ss} such that $C_{ss,1} = 3$ and $S_{ss,1} = 1$, and two higher priority tasks $\tau_1 \stackrel{\mathrm{def}}{=} \langle (1), 5, 5 \rangle$ and $\tau_2 \stackrel{\mathrm{def}}{=} \langle (2), 6, 6 \rangle$. Let us assume that both τ_1 and τ_2 are constrained to have a synchronous release with $\tau_{\mathrm{ss},2}$. It can be verified that the WCRT of $\tau_{\mathrm{ss},1}$ under those constraints is given when both τ_1 and τ_2 release one job in $\tau_{\mathrm{ss},1}$, that is, $R_{ss,1} = 6$. However, using a fixed point iteration with Eq.* (1) *and* (4)*, initiating $R_{ss,1}$ to $C_{ss,1} = 3$, the process immediately converges to $R_{ss,1} = 3$, thereby assuming no job released by the higher priority tasks.* ■

This example shows that the usual fixed-point iteration approach, initiating $R_{ss,1}$ with $C_{ss,1}$, is unsafe. Yet, solving Eq. (1) with $\mathrm{NI}_k = \left\lceil \frac{R_{ss,1}}{T_k} \right\rceil$ for all tasks in $\mathrm{hp}(\tau_{ss})$ — that is, when there is no constraint on the task releases — is known to be an upper-bound on the WCRT of $\tau_{\mathrm{ss},1}$ [10]. Let $\mathrm{UB}_{ss,1}$ be the value of that upper-bound. Based on the observation that increasing $R_{ss,1}$ until its convergence might be optimistic, one might propose to start the fixed point iteration on $R_{ss,1}$ by initiating $R_{ss,1}$ to $\mathrm{UB}_{ss,1}$. Then, the value of $R_{ss,1}$ should decrease over the iterations thanks to the constraint $\mathrm{NI}_k \leq \left\lfloor \frac{R_{ss,1}+S_{ss,1}}{T_k} \right\rfloor$. However, as proven in the example provided below, the new solution output by this second method can over-estimate the WCRT of $\tau_{\mathrm{ss},1}$.

Example 2. *Consider the same set of tasks as in Example 1, but let us assume that τ_1 must have a synchronous release with $\tau_{\mathrm{ss},2}$ and τ_2 with $\tau_{\mathrm{ss},1}$. It can be computed that the WCRT of $\tau_{\mathrm{ss},1}$ under those constraints is given when both τ_1 and τ_2 release one job in $\tau_{\mathrm{ss},1}$, that is, $R_{ss,1} = 6$. Moreover, the WCRT of $\tau_{\mathrm{ss},1}$ is known to be upper-bounded by $\mathrm{UB}_{ss,1} = 9$. If we initiate the fixed point iteration on $R_{ss,1}$ in Eq.* (1) *with $\mathrm{UB}_{ss,1}$, we obtain $R_{ss,1} = 8$. This is impossible since it would mean that τ_2 releases two jobs in $\tau_{\mathrm{ss},1}$, that is ,its second job would be released at time* 6 *when $\tau_{\mathrm{ss},1}$ just completed its execution.* ■

These two examples show that, in the general case, no simple method exists yet to find a exact solution to the constrained releases subproblem.

B. Solution for the constrained releases subproblem

In this section, we propose a method to compute the exact WCRT on the sum of $R_{ss,1}$ and $R_{ss,2}$ under a set of release constraints $\mathrm{Synch}_2^{\mathrm{ss}}$.

The proposed method to compute $R_{ss,1}$ is based on a combination of the two straightforward but inexact solutions investigated in Section IV-A. That is, we simultaneously increase and decrease the value of $R_{ss,1}$ in two different but interdependent iterative processes until they converge to the same value. To do so, Eq. (4) is rewritten as follows

$$\mathrm{NI}_k = \begin{cases} \min\left(\left\lceil \frac{R_{ss,1}}{T_k} \right\rceil, \mathrm{NI}_{\mathrm{k}}^{\max}\right) & \text{if } \tau_k \in \mathrm{Synch}_2^{\mathrm{ss}} \\ \left\lceil \frac{R_{ss,1}}{T_k} \right\rceil & \text{otherwise} \end{cases} \tag{5}$$

where $\mathrm{NI}_{\mathrm{k}}^{\max}$ is an upper-bound on the number of jobs of τ_k interfering with the first execution region of τ_{ss}. This new formulation of Eq. 4 removes the recursiveness in the term enforcing compliance with the constraints imposed by $\mathrm{Synch}_2^{\mathrm{ss}}$. The WCRT can therefore be computed with the usual fixed-point iteration on $R_{ss,1}$ where $R_{ss,1}$ is initialized to $C_{ss,1}$ for the first iteration. During that process, $\mathrm{NI}_{\mathrm{k}}^{\max}$ is assigned a known upper-bound on NI_k. Because $\mathrm{NI}_{\mathrm{k}}^{\max}$ is an upper-bound, the result obtained for $R_{ss,1}$ after convergence of Eq. (5) is also an upper-bound on the actual WCRT of $\tau_{\mathrm{ss},1}$. However, with this value, the constraint expressed by Eq. (3) may not be respected. Therefore, the constraint imposed by Eq. (3) is checked. If violated, the value of $\mathrm{NI}_{\mathrm{k}}^{\max}$ is decreased and Eq. (1) is solved again. Otherwise, an exact WCRT for $\tau_{\mathrm{ss},1}$ has been found.

Lines 2 to 14 of Algorithm 1 present a pseudo-code of that method. Starting with the upper bound $\mathrm{UB}_{ss,1}$ on $R_{ss,1}$ (line 4), it iteratively removes jobs of higher priority tasks interfering with $\tau_{\mathrm{ss},1}$ (lines 7–11) when the condition expressed in Eq. (3) is violated, i.e., $\mathrm{NI}_k > \frac{R_{ss,1}+S_{ss,1}}{T_k}$. With this updated

Algorithm 1: Computation of the WCRT of $\tau_{\rm ss}$ assuming a set of constraints on higher priority tasks releases

1 **Function** *RespTime (* $\mathrm{hp}(\tau_{ss})$, $\mathrm{Synch}_2^{\rm ss}$, $NI^{\rm up}$ *)* **is**
  Inputs : $\mathrm{hp}(\tau_{ss})$ - set of higher priority tasks w.r.t. $\tau_{\rm ss}$
  $\mathrm{Synch}_2^{\rm ss}$ - the set of tasks in $\mathrm{hp}(\tau_{ss})$ with an imposed synchronous release with $\tau_{\rm ss,2}$
  $NI^{\rm up}$ - vector of upper bounds on the number of jobs of each task τ_k, that can interfere with $\tau_{\rm ss,1}$
  Output: R_{ss} - The exact WCRT for $\tau_{\rm ss}$ when respecting the constraints given by $\mathrm{Synch}_2^{\rm ss}$ and $NI^{\rm up}$
2 $R_{ss,1}^{\rm bwd} \leftarrow 0$;
3 $NI \leftarrow NI^{\rm up}$;
4 $R_{ss,1} \leftarrow C_{ss,1} + \sum_{\forall k \in \mathrm{hp}(\tau_{ss})} NI_k^{\rm up} \times C_k$;
5 **while** $R_{ss,1}^{\rm bwd} \neq R_{ss,1}$ **do**
6 $R_{ss,1}^{\rm bwd} \leftarrow R_{ss,1}$;
  /* Update the number of interfering jobs for the tasks synchronous with $\tau_{\rm ss,2}$ */
7 **forall the** $\tau_k \in \mathrm{Synch}_2^{\rm ss}$ **do**
8 **if** $NI_k > \frac{R_{ss,1}+S_{ss,1}}{T_k}$ **then**
9 $NI_k \leftarrow NI_k - 1$;
10 **end**
11 **end**
  // Compute the response time of $\tau_{\rm ss,1}$
12 $R_{ss,1} \leftarrow C_{ss,1} + \sum_{\forall k \in \mathrm{hp}(\tau_{ss})} \min(NI_k; \left\lceil \frac{R_{ss,1}}{T_k} \right\rceil) \times C_k$;
13 **forall the** $\tau_k \in \mathrm{hp}(\tau_{ss})$ **do** $NI_k \leftarrow \left\lceil \frac{R_{ss,1}}{T_k} \right\rceil$;
14 **end**
  // Compute the offsets with $\tau_{\rm ss,2}$
15 **forall the** $\tau_k \in \mathrm{hp}(\tau_{ss})$ **do**
16 $O_{k,2} \leftarrow \max(0; NI_k \times T_k - R_{ss,1} - S_{ss,1})$;
17 **end**
  // Compute the response time of $\tau_{\rm ss,2}$
18 $R_{ss,2} \leftarrow C_{ss,2} + \sum_{\forall k \in \mathrm{hp}(\tau_{ss})} \left\lceil \frac{R_{ss,2}-O_{k,2}}{T_k} \right\rceil \times C_k$;
19 $R_{ss} \leftarrow R_{ss,1} + S_{ss,1} + R_{ss,2}$; ;
  /* Check if there is not a release pattern with one less job in $\tau_{\rm ss,1}$ that increases the overall WCRT */
20 **if** $R_{ss} < \mathrm{UB}_{ss}$ *and* $R_{ss,2} < \mathrm{UB}_{ss,2}$ **then**
21 **forall the** $\tau_k \in \mathrm{hp}(\tau_{ss})$ **do**
22 **if** $NI_k > 0$ **then**
23 $NI' \leftarrow NI$;
24 $NI'_k \leftarrow NI_k - 1$;
25 $R \leftarrow \mathrm{RespTime}(\mathrm{hp}(\tau_{ss}), \mathrm{Synch}_2^{\rm ss}, NI')$;
26 **If** $R > R_{ss}$ **then** $R_{ss} \leftarrow R$;
27 **end**
28 **end**
29 **end**
30 **return** R_{ss};
31 **end**

value, the response time $R_{ss,1}$ is recomputed at line 12 of Algorithm 1. This process iterates until the value computed for $R_{ss,1}$ converges to the exact WCRT of $\tau_{\rm ss,1}$ under the constraints imposed by $\mathrm{Synch}_2^{\rm ss}$.

Once the response time of $\tau_{\rm ss,1}$ has been computed, the offset $O_{k,2}$ can be obtained with Eq. (6). This offset accounts for the difference between the earliest instant at which each task τ_k can release a job in $\tau_{\rm ss,2}$, while respecting its minimum inter-arrival time T_k, and the beginning of $\tau_{\rm ss,2}$.

$$O_{k,2} = \begin{cases} 0 & \text{if } \tau_k \in \mathrm{Synch}_2^{\rm ss} \\ \max\{0,\ \mathrm{NI}_k \times T_k - (R_{ss,1} + S_{ss,1})\} & \text{otherwise} \end{cases} \quad (6)$$

As expressed by Eq. (6), any release of a job of τ_k that should happen within the suspension region of $\tau_{\rm ss}$ is delayed until the beginning of $\tau_{\rm ss,2}$, thereby imposing $O_{k,2} = 0$. This allows us to maximize the interference caused by τ_k to $\tau_{\rm ss,2}$.

$R_{ss,2}$ is given by Eq. (7) and is computed as the traditional response time for non-self-suspending tasks. That is, we seek a minimum response time that satisfies the fixed-point iteration by starting with $R_{ss,2} = C_{ss,2}$. In Algorithm 1, this is reflected in lines 15 to 18.

$$R_{ss,2} = C_{ss,2} + \sum_{\forall k \in hp(\tau_i)} \left\lceil \frac{R_{ss,2} - O_{k,2}}{T_k} \right\rceil \times C_k \quad (7)$$

Line 19 computes the overall response time of $\tau_{\rm ss}$. However, as proven in Section III-B, it might happen that releasing one less job in $\tau_{\rm ss,1}$ allows to increase the response time of $\tau_{\rm ss,2}$ and in turn increase the overall response time of $\tau_{\rm ss}$. Therefore, lines 20 to 29 have been added to consider that case. Those lines call recursively the function RespTime, imposing the upper-bound on the number of interfering jobs with the first execution region to be one less than in the computed solution. Of course, this recursion must not be activated if the overall response time R_{ss} found for $\tau_{\rm ss}$ is already equal to a known upper-bound obtained with simple approximation techniques likes those proposed in [7]. Similarly, there is no need trying to increase the response time of $\tau_{\rm ss,2}$ by reducing the response time of $\tau_{\rm ss,1}$ if $R_{ss,2}$ is already equal to an upper-bound.

It can easily be seen by looking at Algorithm 1, that computing the exact WCRT of $\tau_{\rm ss}$ becomes rapidly intractable. This fact has been confirmed by the experiments reported in Section VII. Therefore, in the next section, we propose a second method, using a MILP formulation, to compute an approximation over the WCRT of $\tau_{\rm ss}$.

V. Upper-bound on the WCRT for a self-suspending task with multiple self-suspending regions

The exact test proposed in the previous section, although restricted to one suspension region, rapidly becomes intractable. In this section, we therefore propose an MILP formulation for computing an upper-bound on the WCRT of a self-suspending task with *multiple* suspension regions when all the interfering tasks are non-self-suspending. This formulation will be extended in the next section to consider the case where multiple self-suspending tasks interfere with each other.

The optimization problem, defined by Expressions (8) to (16) (explained below), has the objective to maximize the sum of the response times of every execution region of $\tau_{\rm ss}$. Its constraints (9)–(16) can all be easily linearized (see [3] for instance). In the proposed problem formulation, the number of jobs $\mathrm{NI}_{k,j}$ of each task $\tau_k \in \mathrm{hp}(\tau_{ss})$ interfering with each execution region of $\tau_{\rm ss}$ are integer variables, while the response time $R_{ss,j}$ of each execution region $\tau_{{\rm ss},j}$ of $\tau_{\rm ss}$ and the offsets $O_{k,j}$ of each task τ_k with each execution region $\tau_{{\rm ss},j}$ are real variables. This MILP formulation is quite simple in comparison to the exact test described in Algorithm 1. As demonstrated in Section VII, this permits a state-of-the-art

MILP solver to output results in an acceptable amount of time for reasonable system sizes.

$$\textbf{Maximize:} \quad \sum_{j=1}^{m_{ss}} R_{ss,j} \tag{8}$$

Subject to:

$$\sum_{j=1}^{m_{ss}} R_{ss,j} + \sum_{j=1}^{m_{ss}-1} S_{ss,j} \leq \mathrm{UB}_{ss}, \tag{9}$$

$$\forall \tau_{\mathrm{ss},j} \in \tau_{\mathrm{ss}} \ : \ R_{ss,j} = C_{ss,j} + \sum_{\tau_p \in \mathrm{hp}(\tau_{ss})} \mathrm{NI}_{p,j} \times C_p \tag{10}$$

$$R_{ss,j} \leq \mathrm{UB}_{ss,j} \tag{11}$$

$$\forall \tau_k \in \mathrm{hp}(\tau_{ss}), \forall \tau_{\mathrm{ss},j} \in \tau_{\mathrm{ss}} :$$

$$O_{k,j} \geq 0 \tag{12}$$

$$O_{k,j+1} \geq O_{k,j} + \mathrm{NI}_{k,j} \times T_k - (R_{ss,j} + S_{ss,j}) \tag{13}$$

$$\mathrm{NI}_{k,j} \geq 0 \tag{14}$$

$$\mathrm{NI}_{k,j} \leq \left\lceil \frac{R_{ss,j} - O_{k,j}}{T_k} \right\rceil \tag{15}$$

The constraints (9)–(15) of the optimization problem are a direct translation of the constraints already discussed in Section IV. That is, Constraint (10) is equivalent to Eq. (1); Constraints (12) and (13) are a generalization of Eq. (6) computing the offsets of the higher priority tasks with each execution region; and Constraints (14) and (15) impose the traditional lower- and upper-bound on the number of interfering jobs of each task τ_k with each execution region $\tau_{\mathrm{ss},j}$ as already discussed for Eq. (7). Constraints (9) and (11) reduce the research space of the problem by stating that the overall response time of τ_{ss} and the response time of each of its execution region, respectively, cannot be larger than known upper-bounds computed with simple methods such as the joint and split methods presented in [7].

The solution of the optimization problem can still be improved by adding the following constraint:

$$\forall \tau_k \in \mathrm{hp}(\tau_{ss}), \forall \tau_{\mathrm{ss},j} \in \tau_{\mathrm{ss}} :$$

$$R_{ss,j} > \mathrm{rel}_{k,j} + \sum_{\tau_p \in \mathrm{hp}(\tau_{ss})} \max\{0, \ \left\lfloor \frac{d_{p,j} - \mathrm{rel}_{k,j}}{T_p} \right\rfloor C_p\} \tag{16}$$

where $\mathrm{rel}_{k,j} \stackrel{\mathrm{def}}{=} O_{k,j} + (\mathrm{NI}_{k,j} - 1) \times T_k$ and $d_{p,j} \stackrel{\mathrm{def}}{=} O_{p,j} + \mathrm{NI}_{p,j} \times T_p$.

The value of $\mathrm{rel}_{k,j}$ gives the release instant of the last job of τ_k released in the execution region $\tau_{\mathrm{ss},j}$, while $d_{p,j}$ gives the deadline of the last job of τ_p released in $\tau_{\mathrm{ss},j}$. Therefore, the term $\left\lfloor \frac{d_{p,j} - \mathrm{rel}_{k,j}}{T_p} \right\rfloor$ provides the number of jobs released by τ_p after $\mathrm{rel}_{k,j}$ and the sum thus gives the total workload released by higher priority tasks after $\mathrm{rel}_{k,j}$. Since τ_{ss} cannot execute when higher priority workload is available and because $\mathrm{rel}_{k,j}$ is an instant in the response time of the execution region, the response time of $\tau_{\mathrm{ss},j}$ cannot be smaller or equal than $\mathrm{rel}_{k,j}$ plus the higher priority workload remaining to execute after $\mathrm{rel}_{k,j}$. This is what Constraint (16) enforces.

Note that because the optimization problem tests all the possible values for the offsets $O_{k,j}$ of each task τ_k with every execution region of τ_{ss}, it also tests all the possible synchronous release combinations. Therefore, there is no need to impose any constraint on the synchronous release patterns, as it was the case in Algorithm 1.

VI. Multiple self-suspending tasks

In this section, we propose a solution to analyse multiple self-suspending tasks interfering together. We prove below that each higher priority self-suspending task τ_k can safely be replaced by a non-self-suspending task $\tau'_k \stackrel{\mathrm{def}}{=} \langle (C_k), D_k, T_k, J_k \rangle$ in the response time analysis. The new parameter J_k is the jitter and is given by $J_k \stackrel{\mathrm{def}}{=} \mathrm{WCRT}_k - C_k$. The worst-case execution time C_k of the equivalent task τ'_k is defined as the sum of the worst-case execution times of all τ_k's execution regions, that is, $C_k \stackrel{\mathrm{def}}{=} \sum_{j=1}^{m_k} C_{k,j}$.

Theorem 2. *The interference caused by $\tau_k \in \mathrm{hp}(\tau_i)$ on a self-suspending task τ_i is upper-bounded by the interference caused by the transformed task $\tau'_k \stackrel{\mathrm{def}}{=} \langle (C_k), D_k, T_k, J_k \rangle$.*

Proof sketch: The proof is by contradiction. Let us assume that τ_k causes more interference than τ'_k. There might be only two reasons for this to be true: (i) some jobs released by τ_k cause more interference than the jobs released by τ'_k, or (ii) τ_k releases more jobs than τ'_k in a given time window.

Since τ_k is self-suspending, the interference caused by each job of τ_k is the sum of the interference caused by each of its execution regions. Therefore, the interference caused by each job of τ_k is upper-bounded by $C_k \stackrel{\mathrm{def}}{=} \sum_{j=1}^{m_k} C_{k,j}$. Because jobs of τ'_k have a WCET of C_k, this contradicts (i).

Since the minimum inter-arrival times of τ_k and τ'_k are identical only their jitters may cause (ii) to be true. Now, let us compute the maximum jitter that can be experienced by the jobs of τ_k. Let $a_{k,1}$ denote the arrival time of a job of τ_k. Since WCRT_k assumes that τ_k executes for its WCET, it means that a job of τ_k cannot start executing later than $\mathrm{WCRT}_k - C_k$ after $a_{k,1}$ (otherwise it would complete after $a_{k,1} + \mathrm{WCRT}_k$ and WCRT_k would not be a worst-case response time). The release jitter of τ_k is therefore upper-bounded by $J_k \stackrel{\mathrm{def}}{=} \mathrm{WCRT}_k - C_k$. This contradicts (ii) and hence proves the lemma. ■

This new model can easily be integrated in the MILP formulation presented in the previous section. Let $J_{k,j}$ represents the jitter experienced by the jobs of τ_k released in the j^{th} execution region of τ_{ss}. In the traditional response time analysis, the jitter can be accounted by subtracting it from the offset of the interfering task [2]. That is, Constraint (15) would become

$$\mathrm{NI}_{k,j} \leq \left\lceil \frac{R_{ss,j} - (O_{k,j} - J_{k,j})}{T_k} \right\rceil$$

However, instead of introducing a new set of variables in the optimization problem and hence increase its complexity, one can simply replace $O_{k,j}$ by $O'_{k,j}$ in Constraints (15) and (16), where $O'_{k,j}$ is defined as $O'_{k,j} \stackrel{\mathrm{def}}{=} O_{k,j} - J_{k,j}$. Because $J_{k,j}$ is upper-bounded by J_k, this variable replacement has for consequence that the bound imposed on the offsets of the tasks in $\mathrm{hp}(\tau_{ss})$ must be modified. Therefore, Constraints (12) and (13) must be replaced by:

$$\forall \tau_k \in \mathrm{hp}(\tau_{ss}), \forall \tau_{\mathrm{ss},j} \in \tau_{\mathrm{ss}} :$$

$$O'_{k,j} \geq -J_k$$

$$O'_{k,j+1} \geq O'_{k,j} + \mathrm{NI}_{k,j} \times T_k - (R_{ss,j} + S_{ss,j}) - J_k$$

Note that those modifications to the MILP formulation do not impact its complexity and therefore the time required

to find a solution to the response time analysis of τ_{ss} in comparison to the case where all interfering tasks are non-self-suspending.

VII. Experiments

In this section, we describe experiments conducted through randomly generated task sets to evaluate (i) the applicability of our exact WCRT computation algorithm, (ii) the performance of the MILP method, and (iii) the respective gain in comparison with the state-of-the-art analysis for self-suspending sporadic tasks. We used Gurobi [14], a state-of-the-art MILP solver, to solve our optimization problem.

All the task sets were generated using the `randfixedsum` algorithm [15], allowing us to choose a constant total task set utilization for a given number of tasks and bounded per-task utilization. Accordingly, the total utilization (U_{tot}) was varied from 0.1 to 1 by 0.1 increments. The per-task utilization ranged from $[0.05, \frac{U_{\text{tot}}}{2}]$, while periods were uniformly distributed over $[10, 100]$. The task execution requirements were calculated from the respective periods and utilizations. Individual values for each of the multiple execution and suspension regions were assigned as fraction of the overall values, C_{ss} and S_{ss} respectively, using again the `randfixedsum` algorithm for a constant total of 1 and a minimum fraction of 0.1. We generated 100 task sets per combination of parameters, while ensuring that task τ_{n-1} was always schedulable.

We evaluated our techniques for computing the WCRT of self-suspending sporadic tasks under fixed-priority scheduling by comparing them to the analysis presented in [3] (denoted "SFP") and the 'Split" and "Joint" analysis presented in [7]. For *SFP*, we provided the task priorities using RM as an input to their optimization problem but let it find the optimal phase assignment. We also removed the constraint $R_i \leq D_i$ in the solver described in [3] since the goal of our experiments is not to check the schedulabity of the system *per se* but comparing the actual worst-case response time computed with each analysis. It is worth mentioning that *SFP* deemed a few task sets infeasible, in which case, in order to maintain a fair comparison, they were discarded in the evaluation. For the task model evaluated in this paper, *Split* boils down to force all higher priority tasks to have synchronous releases with each of the execution regions of the self-suspending task. *Joint* is the traditional suspension-oblivious analysis for fixed-priority scheduling, that is, assuming that the suspension regions are part of the worst-case execution time of the task. Both *Split* and *Joint* are simple response time tests that yield well-known upper bounds and that can be computed straightforwardly. The analysis from [6] was not considered as it was found incorrect [13]. All the plots of Fig. 5 represent the inaccuracy of the previous timing analysis techniques when compared to our optimal method (for Fig. 5a) or our optimization problem (for Fig. 5b–f).

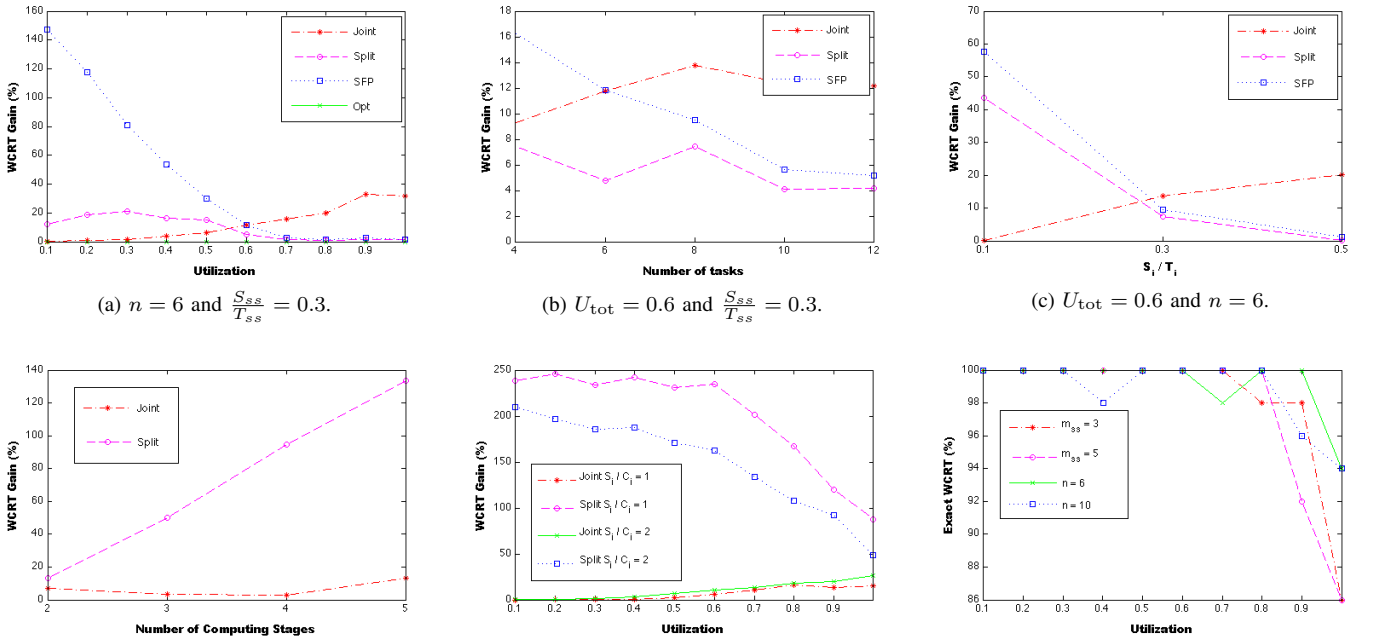
Herein we restrict our attention to problem instances that are representative in size of many real-time systems, in order to study the applicability and trade-offs of the different analysis towards specific parameter intervals. For the first set of experiments presented on Fig. 5a–c, we fixed the number of execution regions of τ_{ss} to 2 and we varied the number of tasks from 4 to 12. The suspension length of τ_{ss} was set as a ratio $\frac{S_i}{T_i}$ with values 0.1, 0.3, and 0.5. Fig. 5a–c show the average gain achieved by our analysis with respect to the WCRT when varying the utilization, the length of the suspension and the number of interfering tasks, respectively. As expected, our MILP formulation (denoted "Opt") and our exact analysis always outperform the other approaches with average gains varying from 1 to 30% relatively to *Joint* and *Split*, and considerably more for *SFP*. The difference with *SFP* can be explained by the fact that the optimization problem of [3] was formulated to find schedulable solutions and not necessarily reduce the response time of the self-suspending tasks. Maximum gains (which is not represented on the plots) observed during our experiments showed that our method can reduce the pessimism on the WCRT over 120% in some cases, but it is approximately 70% for utilizations around 0.7, i.e., close to the schedulability bound of RM for non-self-suspending tasks. However, the gains are highly dependent on the utilization of the system and the length of the suspension region as can be seen on Fig. 5a and 5c.

Joint performs relatively poorly when the suspensions become longer and the utilization increases, but is competitive in the presence of short suspensions and low utilizations. *Split* exhibits the opposite behavior as it is unlikely that the higher priority tasks happen to be released synchronously with both execution regions when the suspension is short and the system is not highly loaded. Although our exact algorithm becomes intractable for $n > 8$ (it is only part of Fig. 5a), *Opt* is able to also provide the exact WCRT for the majority of the task sets as illustrated in Fig. 5f. In order to verify if a given WCRT is the exact solution, one must check if the response time of each execution region converges to the number of interfering jobs assumed to be released inside their window.

We then study the importance of different suspension ratios and how it relates with multiple suspension regions. Accordingly, the second set of experiments had the number of tasks fixed to 6 and the number of execution regions varying from 2 to 5. The suspension length was instead set as a ratio $\frac{S_i}{C_i}$ with values 1 and 2. The results are depicted in Fig. 5d and 5e. *SFP* could not find significant feasible solutions and thus was excluded in the evaluation. It can be observed that an increase in the number of suspension regions, when not accompanied by a substantial increase in the ratio $\frac{S_i}{C_i}$, leads to a severe degradation for the *Split*'s performance, just attenuated for high utilizations which typically result in smaller offsets due to the increase in response time. *Joint* remains very competitive even when the suspension is twice the length of the execution time of the task. Throughout the experiments, it becomes clear that *Joint* and *Split* are not comparable, but also that for such low run-time complexity both approaches may yield tight upper-bounds when applied over this type of task sets. As a last remark, we note that our optimization problem takes in average a few seconds to find its best solution and that the computational time remains acceptable (below 10s) for reasonable dimension of the problem (12 tasks for 2 execution regions, or 6 tasks for 5 execution regions), although under highly loaded circumstances the solver may struggle to improve over the initial solutions for some specific problem instances, in which case a timer may limit the research time.

VIII. Conclusion

In this paper, we have shown that it is simple to characterize a critical instant for sporadic tasks with self-suspensions, thereby invalidating a claim made in earlier works. We highlighted the complexity of the problem and presented an



(a) $n = 6$ and $\frac{S_{ss}}{T_{ss}} = 0.3$.

(b) $U_{\text{tot}} = 0.6$ and $\frac{S_{ss}}{T_{ss}} = 0.3$.

(c) $U_{\text{tot}} = 0.6$ and $n = 6$.

(d) $U_{\text{tot}} = 0.7$, $n = 6$ and $\frac{S_{ss}}{C_{ss}} = 2$.

(e) $n = 6$, $m_{ss} = 5$ and $\frac{S_{ss}}{C_{ss}} = 1$ or 2.

(f) % of exact solutions found by the opt. problem.

Fig. 5: Average WCRT gain (a)–(e) and % of exact solutions found by the optimization problem (f) under various system config.

algorithm to compute the exact WCRT of a lower priority self-suspending tasks when scheduled together with non-self-suspending sporadic tasks. As the algorithm rapidly becomes intractable for a large number of higher priority-tasks due to the exponential number of scenarios that need to be considered, we formulated a response time test for multiple suspension regions as an optimization problem that can be solved by a MILP tool in reasonable time. The optimization problem was then extended to accommodate multiple self-suspending sporadic tasks interfering with each other, still under fixed-priority scheduling. Experiment results showed that the proposed response time tests dominate state-of-the-art techniques, although the WCRT gains highly depend on the peculiarities of the task sets. Experiments also pointed out that the optimization problem finds the exact WCRT solution in the majority of the cases.

Acknowledgment. The authors would like to thank Junsung Kim for providing his simulator for comparing the results of [3] with those obtained in this work. This work was partially supported by National Funds through FCT/MEC (Portuguese Foundation for Science and Technology) and when applicable, co-financed by ERDF (European Regional Development Fund) under the PT2020 Partnership, within project UID/CEC/04234/2013 (CISTER Research Centre); also by, FCT/MEC and the EU ARTEMIS JU within projects ARTEMIS/0003/2012 - JU grant nr. 333053 (CONCERTO) and ARTEMIS/0001/2013 - JU grant nr. 621429 (EMC2), and the European Union under the Seventh Framework Programme (FP7/2007-2013), grant agreement nr. 611016 (P-SOCRATES).

REFERENCES

[1] K. Lakshmanan and R. Rajkumar, "Scheduling self-suspending real-time tasks with rate-monotonic priorities," in *RTAS*, 2010, pp. 3–12.

[2] J. W. S. Liu, *Real-Time Systems*. Prentice Hall, 2000.

[3] J. Kim, B. Andersson, D. de Niz, and R. Rajkumar, "Segment-fixed priority scheduling for self-suspending real-time tasks," in *RTSS*, Dec 2013, pp. 246–257.

[4] C. Liu and J. H. Anderson, "Task scheduling with self-suspensions in soft real-time multiprocessor systems," in *RTSS*, Dec 2009, pp. 425–436.

[5] C. Liu and J. Anderson, "An o(m) analysis technique for supporting real-time self-suspending task systems," in *RTSS*, Dec 2012, pp. 373–382.

[6] C. Liu and J. H. Anderson, "Suspension-aware analysis for hard real-time multiprocessor scheduling," in *ECRTS*, 2013, pp. 271–281.

[7] K. Bletsas, "Worst-case and best-case timing analysis for real-time embedded systems with limited parallelism," Ph.D. dissertation, University of York, Department of Computer Science, 2007, pp. 131–141.

[8] F. Ridouard, P. Richard, F. Cottet, and K. Traoré, "Some results on scheduling tasks with self-suspensions," *Journal of Embedded Computing*, vol. 2, no. 3,4, pp. 301–312, Dec. 2006.

[9] F. Ridouard, P. Richard, and F. Cottet, "Negative results for scheduling independent hard real-time tasks with self-suspensions," in *RTSS*, Dec 2004, pp. 47–56.

[10] K. Tindell and J. Clark, "Holistic schedulability analysis for distributed hard real-time systems," *Microprocess. Microprogram.*, vol. 40, no. 2-3, pp. 117–134, Apr. 1994.

[11] J. Palencia and M. Gonzalez Harbour, "Schedulability analysis for tasks with static and dynamic offsets," in *RTSS*, Dec 1998, pp. 26–37.

[12] C. Liu, J. J. Chen, L. He, and Y. Gu, "Analysis techniques for supporting harmonic real-time tasks with suspensions," in *ECRTS*, July 2014, pp. 201–210.

[13] C. Liu and J. H. Anderson, "Erratum to "suspension-aware analysis for hard real-time multiprocessor scheduling"," https://www.cs.unc.edu/ anderson/papers/ecrts13e_erratum.pdf, 2015.

[14] Gurobi Optimization Inc., "Gurobi optimizer reference manual," http://www.gurobi.com, 2015.

[15] P. Emberson, R. Stafford, and R. I. Davis, "Techniques for the synthesis of multiprocessor tasksets," in *WATERS Workshop*, 2010, pp. 6–11.