Conference Paper

Response Time Analysis of Sporadic DAG Tasks under Partitioned Scheduling

Jose Fonseca
Geoffrey Nelissen
Vincent Nelis
Luís Miguel Pinho

CISTER-TR-160502
Response Time Analysis of Sporadic DAG Tasks under Partitioned Scheduling

Jose Fonseca, Geoffrey Nelissen, Vincent Nelis, Luís Miguel Pinho

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

Abstract

Several schedulability analyses have been proposed for a variety of parallel task systems with real-time constraints. However, these analyses are mostly restricted to global scheduling policies. The problem with global scheduling is that it adds uncertainty to the lower-level timing analysis which on multicore systems are heavily context-dependent. As parallel tasks typically exhibit intense communication and concurrency among their sequential computational units, this problem is further exacerbated. This paper considers instead the schedulability of partitioned parallel tasks. More precisely, we present a response time analysis for sporadic DAG tasks atop multiprocessors under partitioned fixed-priority scheduling. We assume the partitioning to be given. We show that a partitioned DAG task can be modeled as a set of self-suspending tasks. We then propose an algorithm to traverse a DAG and characterize such worst-case scheduling scenario. With minor modifications, any state-of-the-art technique for sporadic self-suspending tasks can thus be used to derived the worst-case response time of a partitioned DAG task. Experiments show that the proposed approach significantly tightens the worst-case response time of partitioned parallel tasks comparatively to the state-of-the-art when the most accurate technique is chosen.
Response Time Analysis of Sporadic DAG Tasks under Partitioned Scheduling

José Fonseca, Geoffrey Nelissen, Vincent Nélis and Luís Miguel Pinho
CISTER/INESC-TEC, ISEP-IPP, Porto, Portugal

Abstract—Several schedulability analyses have been proposed for a variety of parallel task systems with real-time constraints. However, these analyses are mostly restricted to global scheduling policies. The problem with global scheduling is that it adds uncertainty to the lower-level timing analysis which on multicore systems are heavily context-dependent. As parallel tasks typically exhibit intense communication and concurrency among their sequential computational units, this problem is further exacerbated.

This paper considers instead the schedulability of partitioned parallel tasks. More precisely, we present a response time analysis for sporadic DAG tasks atop multiprocessors under partitioned fixed-priority scheduling. We assume the partitioning to be given. We show that a partitioned DAG task can be modeled as a set of self-suspending tasks. We then propose an algorithm to traverse a DAG and characterize such worst-case scheduling scenario. With minor modifications, any state-of-the-art technique for sporadic self-suspending tasks can thus be used to derived the worst-case response time of a partitioned DAG task. Experiments show that the proposed approach significantly tightens the worst-case response time of partitioned parallel tasks comparatively to the state-of-the-art when the most accurate technique is chosen.

I. INTRODUCTION

The introduction of multi- and many-core architectures in the embedded domain has set up the basic environment for the deployment of modern applications, sharing both real-time and high-performance strict requirements. In order to exploit the high computation power provided by those architectures, these applications are typically implemented using parallel programming models (such as OpenMP). Such implementations, combined with the inherent complexity of the hardware, challenge the traditional timing analysis techniques that have been designed primarily in the embedded domain to analyze simple software codes, which were meant to run on simple and predictable hardware architectures.

Recently, the real-time community has started to actively study the timing behavior of parallel tasks under well-established scheduling algorithms. Several parallel task models and schedulability tests have been proposed for multiprocessor systems [1]–[3]. A common assumption among most of these works is that the task system is globally scheduled. Although global scheduling theoretically allows for an overall higher performance, it adds uncertainty and variability to the lower-level timing analysis, which then become overly pessimistic.

As noted in [4], on a multicore system there are strong inter-dependencies between timing and schedulability analysis, since the worst-case execution times are heavily dependent on the amount of cross-core interference generated on shared resources. This phenomenon is further exacerbated with parallel tasks due to the intense communication and concurrency between their sequential computational units. Thus, it is our belief that partitioned scheduling is the most promising approach to support parallel tasks in hard real-time systems.

Partitioned scheduling is a well-studied topic in real-time distributed systems. Different response time analyses, priority assignment techniques and mapping heuristics that allow for task parallelism have been proposed in the past years [5]–[8]. Although these works provide a strong understanding of the worst-case behavior of parallel tasks, they are inevitably less effective when applied to multicore systems due to the absence of the network component and local release jitters. Results concerning the schedulability of partitioned parallel tasks in multiprocessors are very limited. In [9], the authors presented a response time analysis for sporadic fork-join tasks with arbitrary deadlines under fixed-priority scheduling. Unfortunately, the paper was found flawed. Therefore, to the best of our knowledge only the results transposed from distributed systems provide an answer to the problem of scheduling partitioned parallel tasks atop multiprocessors.

This work. We consider a sporadic DAG model, where each sequential computational unit of a DAG task is assigned to a specific core. That is, multiple computational units can run in parallel over the multiprocessor platform but they are not allowed to migrate. We assume that such mapping is given. Under fixed-priority partitioned scheduling, we present a novel response time analysis for this task model. We observe that a partitioned DAG task can be modeled as a set of self-suspending tasks. Such self-suspending tasks depend on each other due to the precedence constraints defined in the DAG. To overcome this problem, we propose an algorithm to traverse a DAG and characterize the worst-case scheduling scenario. Moreover, we show how to transform existing response time analyses for sporadic self-suspending task in uniprocessors to analyze partitioned DAG tasks. In comparison to [6], the evaluation results show that our approach obtains substantial gains in terms of computed worst-case response time (WCRT).

II. RELATED WORK

The problem of scheduling parallel tasks atop multiprocessor platforms has been receiving considerable attention from the real-time community. However the majority of the works have focused on global scheduling. Several parallel task models have been proposed to cope with the different forms of task parallelism generated by commonly used parallel programming models. In the fork-join model [9], [10], a task is represented as an interleaved sequence of sequential and parallel segments. Typically, each parallel segment contains the same number of subtasks, which in turn may not exceed the number of cores in the platform. The synchronous parallel model [1], [2], [11] extends the fork-join model by allowing
successive parallel segments and an unconstrained number of subtasks within each segment. Nonetheless, synchronization is still assumed at every segment’s boundary, so that no subtask is ready for execution unless all the subtasks of the previous segment have completed. A less restrictive parallel structure is supported by the DAG model [3], [12]–[14], where each task is instead characterized by a directed acyclic graph. Nodes represent subtasks and edges define precedence constraints between two nodes. In this sense, a subtask becomes ready for execution as soon as all its precedences constraints are satisfied. Recently, researchers have started addressing conditional parallel tasks [15]–[17] by taking into consideration the different flows of execution that a parallel task may experience.

Although partitioned scheduling of sequential tasks is a well-studied topic (see [18] for a comprehensive survey), results for parallel tasks are limited [5]–[9], [19]. In the context of distributed systems, Tindell and Clark [5] introduced an end-to-end schedulability analysis for a sequence of events called transactions, which was later refined by Palencia et al. [6]. Parallelism is expressed through these transactions. Enhancements to this analysis were then proposed in [7] by considering offsets and in [8] by considering precedence relations. EDF systems have also been addressed [19]. Axer et al. [9] derived a response time analysis for fork-join tasks (Path). Multiple source and sink nodes are allowed.

We now present additional notations and terminologies.

**Definition 1** (Path). For a given DAG task $\tau$, a path $\lambda_{i,k}$ is a sequence of subtasks $v_{i,\ell} \in E_i$ where (1) $\forall v_{i,\ell} \in \lambda_{i,k} \setminus \{v_{i,z}\}$ such that $(v_{i,s}, v_{i,\ell}) \in E_i$, (2) $\forall v_{i,\ell} \in \lambda_{i,k} \setminus \{v_{i,a}\}$, $\exists ! v_{i,s} \in \lambda_{i,k}$ such that $(v_{i,\ell}, v_{i,s}) \in E_i$, (3) $v_{i,\ell}$ is a source node, and (4) $v_{i,z}$ is a sink node.

Formally, a path is a sequence of subtasks where there is a direct precedence constraint between any two adjacent subtasks. We denote by $\ell_i$ the number of different paths $\lambda_{i,k}$ that can be extracted from task $\tau$, i.e., $k \in \{1, 2, \ldots, \ell_i\}$. The set $\lambda_{i,k}$ contains all the distinct cores associated to a path $\lambda_{i,k}$, whereas $v_{i,a}$ and $v_{i,z}$ represent the first and the last subtasks in $\lambda_{i,k}$ assigned to core $p$, respectively. We further define the length of a path $\lambda_{i,k}$ as the sum of the WCET of all its subtasks. Formally, $\lambda_{i,k} = \sum_{v_{i,\ell} \in \lambda_{i,k}} C_{i,\ell}$.

**Definition 2** (Length). The length $\lambda_{i,k}$ of a DAG task $\tau$ is the maximum length among all the $\lambda_i$ paths, i.e., $\lambda_{i,k} = \max_{\ell_i} (\sum_{v_{i,\ell} \in \lambda_{i,k}} C_{i,\ell})$.

Note that, under partitioned scheduling, $\lambda_{i,k}$ may not represent the WCET of $\tau$ (also its WCRT in isolation) when the number of cores available is infinite. Indeed, the degree of parallelism is constrained by the subtask-to-core mapping and thus two independent subtasks assigned to the same core are forced to execute sequentially. Nevertheless, a typical necessary condition for the feasibility of $\tau$ is $\lambda_{i,k} \leq D_i$. Partitioned scheduling allows us to define further feasibility conditions. Hence, we introduce the notion of $p$-workload.

**Definition 3** ($p$-Workload). The worst-case workload of a $p$-Workload of a DAG task $\tau$ on a core $p$, denoted by $W_{\tau}$, is the maximum processing time that any instance of $\tau$ requires from $p$, i.e., $W_{\tau} = \sum_{i=1}^{m} \{C_{i,\ell} | \lambda_{i,k} = p\}$.

The following additional relations must then hold for the feasibility of a DAG task system: $W_{\tau} \leq D_i$ and $\sum_{i=1}^{m} W_{\tau} \leq 1, \forall p \in [1, m]$.

**Definition 4** (Workload). The worst-case workload (or simply “workload”) $W_{\tau}$ of a DAG task $\tau$ is the sum of the WCET of

\[ W_{\tau} = \sum_{i=1}^{m} \{C_{i,\ell} | \lambda_{i,k} = p\}. \]

1A transitive predecessor of $v_{i,b}$ is a subtask that must complete execution before a direct predecessor of $v_{i,b}$ can begin execution.
indicates the WCET and the core affinity of the corresponding node, respectively.

Fig. 1: An example DAG task with twelve subtasks. Each label indicates the WCET and the core affinity of the corresponding node, respectively.

all its subtasks irrespective of their mapping to the cores, i.e., $W_i = \sum_{l=1}^{n_i} C_{i,l}$.

Example. Fig. 1 illustrates our model for a DAG task $\tau_i$ comprised of twelve subtasks ($n_i = 12$) $V_i = \{v_1, \ldots, v_{12}\}$ and twelve precedence constraints. For simplicity, hereinafter we omit the subscript $i$ on the subtasks of $\tau_i$. The label next to each node represents the 2-tuple $v = (C_{i,l}, P_l)$. For instance, subtask $v_0$ has a WCET of $C_0 = 4$ and is assigned to core $P_0 = 2$. There is a total of ten paths ($\ell = 10$) in the DAG. The critical path length of $\tau_i$ equals to $\text{len}(\tau_i) = 26$ and is found on the path $\lambda_{i,k} = (v_2, v_3, v_7, v_8, v_9, v_6, v_{11})$. Its workload is $W_i = 45$, whereas the maximum p-workload resides on core $p = 3$ with the value $W_i^3 = 16$, which results from the subset of subtasks $\{v_5, v_7, v_{11}\}$. Note that this subset imposes no sequencing of subtasks and thus is not necessarily a path on the DAG. To clarify, both $v_5$ and $v_7$ are transitive predecessors of $v_{11}$, but $v_5$ and $v_7$ are independent of each other.

IV. WCRT ANALYSIS OF PARTITIONED DAG TASKS

In this section, we present a schedulability analysis for sporadic DAG tasks with constrained deadlines scheduled in a partitioned fashion with any fixed-priority scheduling algorithm. The schedulability analysis is based on the notion of per-path interference. Unlike its sequential counterpart, partitioned DAG tasks may allocate subtasks to different cores, potentially creating cross-core dependencies. Unless each path is entirely allocated to a single core, the traditional uniprocessor analysis for fixed-priority sequential tasks that relies on the concept of critical instant [20] cannot be applied on a per-path basis. Therefore, we introduce a novel WCRT analysis to cope with the notion of partitioned DAG tasks.

Let $\tau_i$ be the DAG task under analysis. For simplicity and clarity of presentation, we assume that all the higher priority tasks are sequential. That is, $\tau_j \overset{\text{def}}{=} (C_j, P_j, D_j, J_j, F_j)$, $\forall j < i$; meaning $\tau_j$ has a release jitter $J_j$, and each of its jobs can execute only on core $P_j$ and for at most $C_j$ time units. In Section VI, we generalize the analysis for the case where every task in the system is a DAG task.

We seek to derive an upper-bound on the WCRT of each path $\lambda_{i,k}$ of $\tau_i$. To do so, we first introduce some definitions to characterize the different worst-case interference contributions.

Definition 5 (Inter-Task Interference). The inter-task interference $I_i(\lambda_{i,k})$ imposed by a higher priority task $\tau_j$ on the $k^{th}$ path of task $\tau_i$ is the maximum cumulative time during which any subtask $v \in \lambda_{i,k}$ is ready but cannot execute because $\tau_j$ is executing on the same core.

Definition 6 (Self-Interference). The self-interference $I_i(\lambda_{i,k})$ imposed by task $\tau_i$ on its own $k^{th}$ path is the maximum cumulative time during which any subtask $v \in \lambda_{i,k}$ is ready to execute but cannot because access to the same core has been granted to subtasks of $\tau_i$ that do not belong to $\lambda_{i,k}$.

Definitions 5 and 6 provide a characterization of the maximal overall interference exerted on a path $\lambda_{i,k}$ of task $\tau_i$ during the execution of any of its jobs. A response time equation for $\lambda_{i,k}$ can then be expressed as follows.

Theorem 1. The worst-case response time of a path $\lambda_{i,k}$ of a partitioned DAG task $\tau_i$ is given by

$$R(\lambda_{i,k}) = \text{len}(\lambda_{i,k}) + I_i(\lambda_{i,k}) + \sum_{\forall j < i} I_j(\lambda_{i,k})$$

Proof. Let $r_a$ be the release time of the first subtask in the path $\lambda_{i,k}$. In the scheduling $[r_a, r_a + R(\lambda_{i,k})]$, the entire path requires at most $\text{len}(\lambda_{i,k})$ time units of execution. The total interference caused by self-interfering subtasks on $\lambda_{i,k}$ is $I_i(\lambda_{i,k})$ according to Definition 6. By Definition 5, the maximum interference exerted on $\lambda_{i,k}$ by each higher priority task (i.e., tasks $\tau_j$ such that $j < i$) is $I_j(\lambda_{i,k})$. The theorem follows by noting that the set of lower priority tasks cannot influence the schedule of $\tau_i$ in fixed-priority preemptive scheduling.

To obtain the WCRT of $\tau_i$ we apply Equation 1 to each of its paths. The next corollary directly follows from Theorem 1.

Corollary 1. The worst-case response time of a DAG task $\tau_i$ is given by

$$R(\tau_i) = \max_{k=1}^{n_i} R(\lambda_{i,k})$$

As a direct consequence of Corollary 1, DAG task $\tau_i$ is deemed schedulable if $R(\tau_i) \leq D_i$.

A. On the self-interference

Conceptually, the self-interference corresponds to the delay on the completion time of task $\tau_i$ caused by the concurrency among its own subtasks. Since the computing resources are typically scarce, subtasks of $\tau_i$ that could execute in parallel (i.e., they share no direct or transitive precedence constraint) will eventually contend for core-access, thus increasing the response time of $\tau_i$ itself. Under partitioned scheduling this phenomenon is easier to observe as the subtask-to-core assignment dictates which independent subtasks may interfere with each other.

On a particular path $\lambda_{i,k}$, $I_i(\lambda_{i,k})$ accounts for all the time intervals where any subtask $v \in V_i \setminus \lambda_{i,k}$ is executing, while there is a ready subtask $v \in \lambda_{i,k}$ with pending work on the same core $P_k$. Thus, if $P_k \notin \text{proc}(\lambda_{i,k})$, then $v \in \lambda_{i,k}$ can never interfere with $\lambda_{i,k}$. Contrary to the inter-task interference, only one instance of such self-interfering subtasks have to be accounted in $I_i(\lambda_{i,k})$ because they belong to the same job of $\tau_i$. The absence of individual priorities assigned to the
subtasks, together with variability in their execution times, makes self-interference a problem mutual to all the paths of \( \tau_i \). That is, if a path \( A \) interferes with a path \( B \), then it is also possible to construct a scenario where path \( B \) interferes with path \( A \). Furthermore, we observe that when a subtask \( v_{\ell} \in V_i \setminus \lambda_i,k \) shares no independent constraints with any subtask \( v_{\ell} \in \lambda_i,k \), where \( P_{\ell} = P_{v_{\ell}} \), \( v_{\ell} \) cannot interfere with \( \lambda_i,k \). This allows us to derive a less conservative upper-bound on \( I_i(\lambda_i,k) \) comparatively to the global approaches.

**Lemma 1.** For any partitioned constrained-deadline DAG task \( \tau_i \), partitioned on \( m \) cores, an upper-bound on the interfering workload imposed by \( \tau_i \) on its path \( \lambda_i,k \) is given by

\[
I_i(\lambda_i,k) \leq \sum_{v_{\ell} \in \text{proc}(\lambda_i,k)} W_{p} - \text{len}(\lambda_i,k) - \sum_{v_{\ell} \in \Theta_{i,k}} C_{\ell}, \tag{3}
\]

where \( \Theta_{i,k} \) is the set of subtasks of \( \tau_i \) that cannot interfere with \( \lambda_i,k \) and is defined as

\[
\Theta_{i,k} \overset{\text{def}}{=} \bigcup_{v_{\ell} \in \text{proc}(\lambda_i,k)} \text{pred}(v_{\ell}^p,p) \cup \text{succ}(v_{z}^p,p). \tag{4}
\]

**Proof.** By definition of a constrained-deadline task, two instances of \( \tau_j \) cannot interfere with each other when \( D_j \) is met. Therefore, the maximum interference that \( \tau_j \) can impose on its path \( \lambda_j,k \) is upper-bounded by the sum of the WCET of all subtasks assigned to a core \( p \in \text{proc}(\lambda_i,k) \) excluding the subtasks in \( \lambda_i,k \), i.e.,

\[
I_i(\lambda_i,k) \leq \sum_{v_{\ell} \in \text{proc}(\lambda_i,k)} W_{p} - \text{len}(\lambda_i,k). \]

Any subtask \( v_{\ell} \) that is a successor, either directly or transitively, of \( v_{p} \) (i.e., the last subtask of \( \lambda_i,k \) assigned to core \( p \) \( \), where \( \lambda_i,k \) cannot interfere with \( \lambda_i,k \) because \( v_{\ell} \) becomes ready only after all subtasks in \( \lambda_i,k \) assigned to \( P_{\ell} \) complete. Similarly, any subtask \( v_{\ell} \) that is a predecessor, either directly or transitively, of \( v_{p} \) (i.e., the first subtask of \( \lambda_i,k \) assigned to core \( p \) \( \), where \( \lambda_i,k \) cannot interfere with \( \lambda_i,k \) because \( v_{\ell} \) cannot interfere with \( v_{p} \)), cannot be safely discarded because any delay caused by \( v_{\ell} \) on the start of \( v_{p} \) will be accounted in any path \( \lambda_i,k', \) where \( k' \neq k \) and \( \{v_{\ell}, v_{p}\} \subseteq \lambda_i,k' \) (from Equation 2). As given by Equation 4, the set \( \Theta_{i,k} \) contains all those non-interfering subtasks. Hence,

\[
I_i(\lambda_i,k) \leq \sum_{v_{\ell} \in \text{proc}(\lambda_i,k)} W_{p} - \text{len}(\lambda_i,k) - \sum_{v_{\ell} \in \Theta_{i,k}} C_{\ell},
\]

which concludes the proof. \( \square \)

Henceforth, let \( \text{self}(\lambda_i,k) \) denote the set of self-interfering subtasks with \( \lambda_i,k \).

**B. On the inter-task interference**

The inter-task interference \( I_j(\lambda_i,k) \) accounts for all the time intervals during which a subtask \( v_{\ell} \in \lambda_i,k \) is ready but it cannot execute since higher priority task \( \tau_j \) is holding the processor. Hence, \( \tau_j \) may interfere with multiple subtasks in \( \lambda_i,k \). Despite \( \tau_j \) being any task with higher priority than \( \tau_i \), if \( P_j \notin \text{proc}(\lambda_i,k) \), then \( \tau_j \) cannot interfere with \( \lambda_i,k \). We denote by \( \text{hp}(\lambda_i,k) \) the set of higher priority tasks that can effectively interfere with \( \lambda_i,k \).

Although in global multiprocessor scheduling one must consider carry-in jobs as part of the individual interference contributions [18], it is not the case for the fixed-priority scheduling analysis of a partitioned DAG task when the higher priority tasks are sequential and also partitioned as the problem boils down to single core scheduling. In this context, \( I_j(\lambda_i,k) \) is a function of the maximum number of interfering jobs released by \( \tau_j \) in a scheduling window \([r_i^p, f_p] \), where \( r_i^p \) is the release time of subtask \( v_i^p \in \lambda_i,k \), as \( \tau_j \) cannot interfere with subtasks on a core different than \( P_j \). No job released by \( \tau_j \) at any instant prior to \( r_i^p \) interferes with \( \lambda_i,k \) because otherwise there would exist a valid scheduling window \([r_i^p - t, f_p] \) that would increase the response time.

A simple solution to upper-bound \( I_j(\lambda_i,k) \) is to assume that \( \tau_j \) is allowed to release jobs synchronously with every subtask \( v_{\ell} \in \lambda_i,k \) such that \( P_j = P_{v_{\ell}} \). Subsequent job releases are then separated by \( T_{\ell} \) time units. A similar technique to this independent worst-case scenario for each subtask in \( \lambda_i,k \) was adopted in [6]. Clearly, it is not always possible for any \( \tau_j \) to interfere with all the workload of \( \lambda_i,k \) assigned to \( P_j \), thus this technique is often overly pessimistic.

Unfortunately, finding tight upper-bounds on the interfering workload of the higher priority tasks is difficult. The challenge comes from the fact that the workload of \( \lambda_i,k \) on a core \( p \) may not be continuous, as some of the intermediate subtasks are assigned to different cores. As a result, cross-core dependencies on the execution flow of the path exist. The length of these discontinuous intervals is not fixed since the release of a transitive successor on \( p \) depends on the response time of the intermediate subtasks mapped to other cores. Therefore, \( I_j(\lambda_i,k) \) is not only a function of the response time of the subtasks of path \( \lambda_i,k \) on core \( P_j \) = \( p \), but it also has to account for the gaps within the different parts of the workload, and for the relation between the duration of the gaps and the response time of subtasks on any core \( p \neq P_j \).

To overcome this problem, we propose in the next section an alternative technique to Equation 1, for computing an upper-bound on the response time of any path \( \lambda_i,k \), which is based on the self-suspending tasks theory.

**V. Response time of a path \( \lambda_i,k \)**

In this section, we explain how to compute an upper-bound on the worst-case response time of each path \( \lambda_i,k \) of task \( \tau_i \). To capture the worst-case interference suffered by a path on the different cores to which it is mapped, we model a path as a set of self-suspending tasks. More precisely, one self-suspending task for each core. For simplicity, hereinafter, we assume that any two consecutive subtasks assigned to the same core are merged into a single subtask with a WCET equal to the sum of their individual WCETs. Because this section refers to only one specific path \( \lambda_i,k \), for the sake of notation conciseness, we let \( \lambda \) denote that path \( \lambda_i,k \).

**A. Intuition**

In the literature, a self-suspending task is often described as an interleaved sequence of execution and suspension regions, where an execution region is a portion of a sequential task that needs to be processed, and a suspension region corresponds to a period of time during which the task voluntarily yields the processor to perform a remote operation. The suspension regions are assumed to have given bounded durations.

Regarding partitioned DAG tasks, we observe that the existence of precedence constraints between subtasks assigned
to different cores leads to a similar behavior. This is easier to see on a path, since there is a direct precedence constraint between every two consecutive subtasks. As depicted in Fig. 2, consider an example of partitioned schedule for the path highlighted in the DAG of Fig. 1. Although only subtask \( v_4 \) is assigned to core \( p = 1 \), it cannot start executing before the joint subtask \( v_1 + v_3 \) completes execution on core \( p = 2 \). In turn, the execution of \( v_4 \) delays the release of the last portion of workload assigned to core \( p = 2 \) (i.e., subtask \( \tau_p \)), creating an intermediate interval of 3 time units where core \( p = 2 \) is free to execute lower priority tasks. Such time intervals can be seen as suspensions within the path on a certain core \( p \).

In this sense, a path \( \lambda \) can be modeled as a set of sporadic self-suspending tasks, one for each core reached by the path, i.e. \( \forall p \in \text{proc}(\lambda) \). The trick is to treat the subtasks of \( \lambda \) assigned to the current core under analysis as execution regions and the response time of all the remaining subtasks as suspension regions. The problem of computing the response time of \( \lambda \) on a multicore platform becomes then equivalent to the analysis of \( |\text{proc}(\lambda)| \) self-suspending tasks in a uniprocessor system. However, unlike previous works, the duration of the suspension regions are not known beforehand as they are in fact computations to be executed on different cores.

Subtasks assigned to a core \( p' \neq p \) and released before the first execution region or after the last execution region of a self-suspending task formed on core \( p \) must not be considered as part of its suspension regions, since they do not contribute to an increase on the interfering workload\(^3\). For instance, the suspensions on core \( p = 1 \) in Fig. 2 do not influence the response time of \( v_4 \). Every other subtask assigned to a core \( p' \neq p \) is henceforth called "remote subtask".

**B. The self-suspending task model**

For ease of understanding, we start with a generic model that closely relates to the ones considered in the literature. Let \( \tau^p \) denote the self-suspending task formed by a path \( \lambda \) on core \( p \). Each self-suspending task \( \tau^p \), \( \forall p \in \text{proc}(\lambda) \), consists of \( q \geq 1 \) execution regions and \( q - 1 \) suspension regions such that any two consecutive execution regions are separated by a suspension region. Formally, \( \tau^p \triangleq \{ (C^p_1, S^p_1), (C^p_2, S^p_2), \ldots, (C^p_{q-1}, S^p_{q-1}), (C^p_q, S^p_{ub}) \} \), where \( C^p_q \) is the WCET of the \( h^{th} \) execution region of \( \tau^p \), while an upper-bound on the duration of the \( h^{th} \) suspension region of \( \tau^p \) is given by \( S^p_h \). The parameter \( S^p_{ub} \) denotes an upper-bound on the overall suspension time.

Note that \( S^p_{ub} \) is not necessarily equal to the maximum cumulative suspension time. That is, \( S^p_{ub} \leq \sum_{h=1}^{q} S^p_h \). While it is easy to see that each \( C^p_h \) corresponds to the WCET of the \( h^{th} \) subtask of \( \lambda \) assigned to core \( p \), the values of each \( S^p_h \) and \( S^p_{ub} \) cannot be directly derived from \( \lambda \) and must therefore be computed. We show next how these values can be expressed as functions of the WCRT of the remote subtasks of \( \tau^p \).

Additionally, we represent by \( E^p_h \) the \( h^{th} \) execution region of \( \tau^p \), \( \forall h \in [1, q] \). A sequential task is a self-suspending task with no suspension regions. In this particular case, \( \tau^p \) is represented as: \( \tau^p \triangleq \{ (C^p_1, S^p_{ub}) \} \).

**C. Methodology**

The purpose of the above model is to not tie the analysis of a partitioned DAG task to a particular work on self-suspending tasks. Thus, any existing uniprocessor timing analysis for fixed-priority sporadic self-suspending tasks can be used to derive the WCRT of a self-suspending task \( \tau^p \). As the WCRT of each \( \tau^p \) encompasses the WCRT of all the subtasks assigned to core \( p \), the WCRT of a path \( \lambda \) can be expressed by its self-suspending tasks. Consequently, Eq. 1 is rewritten as follows.

**Theorem 2.** The WCRT of any path \( \lambda \) is upper-bounded by

\[
R(\lambda) \leq \sum_{\forall p \in \text{proc}(\lambda)} R(\tau^p) - S^p_{ub} \tag{5}
\]

**Proof.** Each self-suspending task \( \tau^p \) models the worst-case request of the path \( \lambda \) on core \( p \). By definition, \( R(\tau^p) \) upper-bounds the cumulative response time of its execution and suspension regions. Thus \( R(\tau^p) \) also upper-bounds the sum of the response time of the subtasks of \( \lambda \) assigned to core \( p \). By summing the WCRT of each \( \tau^p \), \( \forall p \in \text{proc}(\lambda) \), we therefore upper-bound the sum of the response time of all the subtasks in \( \lambda \). Furthermore, because by definition of \( \tau^p \) the suspension time of a task \( \tau^p \) corresponds to the processing time of \( \lambda \) on other cores than \( p \), the suspension time \( S^p_{ub} \) should not be considered as part of \( R(\tau^p) \) as it is already accounted by the other self-suspending tasks \( \tau^p \), where \( \forall p' \neq p \).

An important aspect to consider is that the self-suspending tasks depend on each other through the suspension regions. Bounds on the suspension regions are necessary to analyze the WCRT of the execution regions, but the suspension regions are indeed execution regions on other cores. In the following we explain how to break this circular dependency by capturing the worst-case behavior of the remote subtasks of \( \tau^p \).

A well-known result from the sporadic self-suspending tasks theory is that larger suspensions cannot lead to a decrease in the interference suffered by the task under analysis. Therefore, we are interested in finding the WCRT of all the remote subtasks of \( \tau^p \) as a way to characterize the worst-case request of \( \tau^p \). We first present how \( S^p_{ub} \) can be computed.

Whenever multiple remote subtasks of \( \tau^p \) are assigned to the same core \( p' \), assuming an independent WCRT for each one of them is overly pessimistic. In fact, they form an inner self-suspending task within \( \tau^p \). Let \( R_{ss} \) denote that inner self-suspending task. Then, the WCRT of all such remote subtasks on \( p' \) together is given by \( R(R_{ss}) - S^p_{ub} \).
A special case happens when there is only one remote subtask of $\tau^p$ allocated to core $P_1 \neq p$. Let $v_\ell$ be that subtask. If no other remote subtask of $\tau^p$ is assigned to $P_1$, then the worst-case duration of the suspension region generated by $v_\ell$ is given by the worst-case response time $R(v_\ell)$ of $v_\ell$. Subtask $v_\ell$ can then be considered an independent sequential task for which the critical instant in uniprocessor holds.

Let $R(\tau^p_{ss})$ denote the upper-bound on the WCRT of the remote subtasks of $\tau^p$ assigned to a core $p$. Then, the maximum overall suspension time experienced by $\tau^p$ is

$$S^{p, ub} = \sum_{v_\ell \in proc(\lambda), v_\ell \neq p} R(\tau^p_{ss}) - S^{o, ub}_{ss}. \quad (6)$$

Although parameter $S^{p, ub}_h$, $\forall h \in \{1, q-1\}$ provides no information about the relation between the different suspension regions, it can seamlessly serve as input to any self-suspending analysis that disregard the placement and duration of each suspension region specifically.

We now present how to compute an upper-bound on $S^{p, ub}_h$, $\forall h \in \{1, q-1\}$. A simple solution is to compute an independent WCRT for each of the remote subtasks of $\tau^p$ and sum the resulting values of those that belong to the same suspension region. Thus, if the remote subtasks $v_\ell$ and $v_{\ell+1}$ separate the execution regions $E^{p}_1$ and $E^{p}_2$, then $S^{p}_h = R(v_\ell) + R(v_{\ell+1})$ (similarly to [6]). The pessimism can be reduced by considering the fact that if a inner self-suspending task $\tau_{ss}$ resides inside a single suspension region of $\tau^p$, the cumulative WCRT of the remote subtasks of $\tau^p$ on $p'$ that correspond to execution regions in $\tau_{ss}$ can be replaced by $R(\tau_{ss}) - S^{o, ub}_{ss}$. Assume that only remote subtask $v_\ell$ separates such remote subtasks and $p \neq P_1 \neq p'$. Accordingly, $S^{p}_h = R(\tau_{ss}) - S^{o, ub}_{ss} + R(v_\ell)$.

Example. Fig. 3 depicts how each core perceives the path $\lambda$ of task $\tau$, highlighted in Fig. 1. We describe through this example how to characterize each self-suspending task $\tau^p$ for that particular path. On core $p = 3$, all the remote subtasks appear before and after the execution region $E^{p}_2$ constituted of $v_5$, so $\tau^3$ is a sequential task with $C^{\tau^3}_1 = C_5$. On core $p = 2$, $\tau^2$ has $q = 2$ execution regions with $C^{\tau^2}_1 = C_3$ and $C^{\tau^2}_2 = C_4$ and a single suspension region comprised of two remote subtasks. The maximum duration of the suspension region is given by the independent upper-bounds on $v_4$ and $v_5$, i.e. $S^{\tau^2}_{ss} = R(v_4) + R(v_5)$. Since there is only one suspension region, $S^{2, ub}_{ss} = S^{\tau^2}_{ss}$. Finally, $\tau^1$ has $q = 3$ execution regions with $C^{\tau^1}_1 = C_1$, $C^{\tau^1}_2 = C_4$ and $C^{\tau^1}_3 = C_3$, and two suspension regions for which $S^{\tau^1}_1 = R(v_3)$ and $S^{\tau^1}_2 = R(v_3) + R(v_6)$. Note, however, that suspending subtasks $v_3$ and $v_6$ form in fact $\tau^2$. Thus, $S^{1, ub}_{ss} = R(\tau^2) - S^{2, ub}_{ss} + R(v_5)$.

Given the above relations, it becomes clear that to capture the worst-case request of a self-suspending task $\tau^p$ many inner self-suspending tasks must be characterized and analyzed. This implies that an effective order of computations in necessary to drive the whole algorithm.

D. Unfolding the path

To overcome the dependency problem, we propose an algorithm that recursively divides a path into smaller ones, creating a tree of subpaths which represent self-suspending tasks. The tree reflects the hierarchy of dependencies. When a leaf is reached, the corresponding task has no suspension regions (i.e., it is a sequential task), thus its WCRT does not depend on anything else other than the interfering workload on that core and can be computed immediately. The computed values are then back propagated to the self-suspending tasks on the upper levels, so that their suspension time is no longer unknown. After setting the appropriate bounds on the suspension regions, the WCRT of the execution regions of the next self-suspending task is derived. The process continues level-by-level until the root is revisited. At this point, the WCRT of the original path can be computed thanks to Theorem 2.

Algorithm 1 shows the pseudo-code of the recursive algorithm for unfolding any path $\lambda$ and computing the required WCRTs. It takes as input a path $\lambda_{ss}$, and both the set of higher priority tasks $hp(\lambda)$ and the set of self-interfering subtasks $self(\lambda)$ regarding the path $\lambda$. Initially, $\lambda_{ss} = \lambda$. The algorithm starts by finding the number of subtasks in the path $\lambda_{ss}$ (line 2). If the path spans over more than one core (lines 10-30), there are self-suspending tasks with suspension regions to be identified. Here, we consider two cases.

First (lines 11-16), if both the first and the last subtask ($v^{1st}$ and $v^{last}$ respectively) of $\lambda_{ss}$ reside on the same core (let $p$ be that core), then $\lambda_{ss}$ constitutes one of the self-suspending tasks\(^4\) on core $p$. However, its response time cannot be computed straight away because the bounds on its suspension regions are not known yet. Thus, we exclude $v^{1st}$ and $v^{last}$ from $\lambda_{ss}$ and invoke the algorithm for the resulting subpath. The second case deals with paths that start and end on different cores (lines 17-29). In this situation, we just split the path $\lambda_{ss}$ in two subpaths to be analyzed: i) from the first subtask of $\lambda_{ss}$ on the same core as $v^{last}$ to $v^{last}$ (lines 18-26), and ii) $\lambda_{ss}$ except $v^{last}$ (lines 27-28).

The recursion on a path stops when there is only one subtask in $\lambda_{ss}$. The WCRT of a single subtask can then be computed by adding the self-interfering workload to the traditional equation for fixed-priority sequential tasks in uniprocessors. That is, the WCRT $R(\lambda_{ss})$ when $\lambda_{ss}$ contains only one subtask is given by Eq. 7 and reflected in line 8 of Algorithm 1.

$$R(\lambda_{ss}) = C^{1st}_{ss} + \sum_{\forall v_\ell \in hp(\lambda_{ss})} \left[ R(\tau_{ss}) + J_{i} \right] \times C_{j} + \sum_{\forall v_\ell \in self(\lambda_{ss})} C_{k} \quad (7)$$

\(^4\)Note that a self-suspending task $\tau^p$ may be decomposed into multiple inner self-suspending tasks, some of which constituting remote subtasks for a self-suspending task $\tau^{p'}$ where $p \neq p'$. 

---

Fig. 3: Overview of the path $\lambda$ highlighted in Fig. 1 as a set of self-suspending tasks.
As soon as all the subpaths of a self-suspending task $\tau_{ss}$ on $p$ have been analyzed, the bound on the total duration of its suspensions $S_{ss}$ is given by the sum of the WCRT of its inner self-suspending tasks that are not on $p$ (as discussed in Section V-C). All of these values have already been computed and available in the matrix $RT_s$ returned by Algorithm 1. Every $C_{ss,h}$ is assigned with the WCET of the $h$th subtask in $\lambda_{ss}$ on $p$. Apart from the overall bound on the suspension time, an individual upper-bound $S_{ss,h}$ on each suspension region is also required. While parsing the path $\lambda_{ss}$, if the WCRT of a remote subtask is missing in $RT_s$, we apply Eq. 7 to that particular subtask. This allows us to compose the values $S_{ss,h}$ according to the relations of the remote subtasks with each suspension region. All such operations are performed by the function $setSuspendingTask$ in line 14. With all the parameters in $\tau_{ss}$ defined, the WCRT of the self-suspending task is compute at line 15. Details about the timing analysis are provided in the subsection below. Finally, the procedure is repeated until all the self-suspending tasks are analyzed.

Applying Algorithm 1 to a path $\lambda$ guarantees that the WCRT of every self-suspending task $\tau^p$ defined from $\lambda$ has been correctly computed.

**E. WCRT of a self-suspending task**

We now show how three different state-of-the-art response time analyses [21], [22] for sporadic self-suspending tasks can be extended to cope with both the dependent suspension regions and the self-interfering workload discussed in the previous sections. The worst-case release pattern for the higher priority tasks follows directly from the results of these analyses. Without loss of generality, we consider the WCRT of a self-suspending task $\tau^p$. We start by proving the worst-case release pattern for the self-interfering subtasks in $self(\tau^p)$.

**Lemma 2.** The contribution of $self(\tau^p)$ to the WCRT of a self-suspending task $\tau^p$ is upper-bounded by releasing a single instance of each self-interfering subtask in $self(\tau^p)$ synchronously with any one execution region $E_h^p$ of $\tau^p$.

**Proof.** Following Lemma 1, a self-interfering subtask $v_t \in self(\tau^p)$ can delay the execution of a self-suspending task $\tau^p$ at most once for $C_t$ time units. Since all subtasks in $\tau_t$ have the same priority, they cannot preempt each other. Thus, $v_t$ cannot execute when an execution region of $\tau^p$ is active. The maximum interference imposed by $self(\tau^p)$ on $\tau^p$ happens when each $v_t \in self(\tau^p)$ is released synchronously with an execution region $E_h^p$. Although such self-interference is upper-bounded by $\sum_{v_t \in self(\tau^p)} C_t$ as a whole, the number of higher priority interfering jobs is influenced by the exact placement of each self-interfering subtask, as assuming that $v_t$ interferes with the execution region $E_h^p$ is equivalent to increase $C_h$ by $C_t$ time units. In general, enlarging different execution regions leads to different response times. Hence, allowing each of these self-interfering subtasks to be released with any one execution region of $\tau^p$ captures the worst-case contribution of $self(\tau^p)$ to $R(\tau^p)$.

Suspension-oblivious analyses [21] treat suspension regions as part of the computations to be processed, making suspensions subject to the same sources of interference than the execution regions. In this sense, the entire self-suspending task is modeled as a single sequential task. Consequently, an upper-bound on the WCRT of $\tau^p$ is found when the overall suspension time is the largest (i.e. $S_{ss}^{ub}$) and the self-interfering subtasks are released synchronously with $E_h^p$. That is,

$$R(\tau^p) = \sum_{h=1}^{q} C_h^p + S_{ss}^{ub} + \sum_{v_t \in self(\tau^p)} \frac{[R(\tau^p) + J_t]}{T_j} \times C_j + \sum_{v_t \in self(\tau^p)} C_t$$

**(8)**

The simplest suspension-aware analysis [21] focuses on upper-bounding the WCRT of each execution region independently. That is, the problem is reduced to a set of smaller sequential tasks by assuming that all the interfering workload releases jobs synchronously with each and every execution region. In this case, the self-interfering subtasks must be accounted once in each execution region $E_h^p$. Moreover, the suspension time has no influence on the interference generated. By using Eq. 7 to compute each $R(E_h^p)$, the WCRT of $\tau^p$ is given by

$$R(\tau^p) = S_{ss}^{ub} + \sum_{h=1}^{q} R(E_h^p)$$

**(9)**

As soon as all the subpaths of a self-suspending task $\tau_{ss}$ on $p$ have been analyzed, the bound on the total duration of its suspensions $S_{ss}$ is given by the sum of the WCRT of its inner self-suspending tasks that are not on $p$ (as discussed in Section V-C). All of these values have already been computed and available in the matrix $RT_s$ returned by Algorithm 1. Every $C_{ss,h}$ is assigned with the WCET of the $h$th subtask in $\lambda_{ss}$ on $p$. Apart from the overall bound on the suspension time, an individual upper-bound $S_{ss,h}$ on each suspension region is also required. While parsing the path $\lambda_{ss}$, if the WCRT of a remote subtask is missing in $RT_s$, we apply Eq. 7 to that particular subtask. This allows us to compose the values $S_{ss,h}$ according to the relations of the remote subtasks with each suspension region. All such operations are performed by the function $setSuspendingTask$ in line 14. With all the parameters in $\tau_{ss}$ defined, the WCRT of the self-suspending task is compute at line 15. Details about the timing analysis are provided in the subsection below. Finally, the procedure is repeated until all the self-suspending tasks are analyzed.

Applying Algorithm 1 to a path $\lambda$ guarantees that the WCRT of every self-suspending task $\tau^p$ defined from $\lambda$ has been correctly computed.

**E. WCRT of a self-suspending task**

We now show how three different state-of-the-art response time analyses [21], [22] for sporadic self-suspending tasks can be extended to cope with both the dependent suspension regions and the self-interfering workload discussed in the previous sections. The worst-case release pattern for the higher priority tasks follows directly from the results of these analyses. Without loss of generality, we consider the WCRT of a self-suspending task $\tau^p$. We start by proving the worst-case release pattern for the self-interfering subtasks in $self(\tau^p)$.

**Lemma 2.** The contribution of $self(\tau^p)$ to the WCRT of a self-suspending task $\tau^p$ is upper-bounded by releasing a single instance of each self-interfering subtask in $self(\tau^p)$ synchronously with any one execution region $E_h^p$ of $\tau^p$.

**Proof.** Following Lemma 1, a self-interfering subtask $v_t \in self(\tau^p)$ can delay the execution of a self-suspending task $\tau^p$ at most once for $C_t$ time units. Since all subtasks in $\tau_t$ have the same priority, they cannot preempt each other. Thus, $v_t$ cannot execute when an execution region of $\tau^p$ is active. The maximum interference imposed by $self(\tau^p)$ on $\tau^p$ happens when each $v_t \in self(\tau^p)$ is released synchronously with an execution region $E_h^p$. Although such self-interference is upper-bounded by $\sum_{v_t \in self(\tau^p)} C_t$ as a whole, the number of higher priority interfering jobs is influenced by the exact placement of each self-interfering subtask, as assuming that $v_t$ interferes with the execution region $E_h^p$ is equivalent to increase $C_h$ by $C_t$ time units. In general, enlarging different execution regions leads to different response times. Hence, allowing each of these self-interfering subtasks to be released with any one execution region of $\tau^p$ captures the worst-case contribution of $self(\tau^p)$ to $R(\tau^p)$.

Suspension-oblivious analyses [21] treat suspension regions as part of the computations to be processed, making suspensions subject to the same sources of interference than the execution regions. In this sense, the entire self-suspending task is modeled as a single sequential task. Consequently, an upper-bound on the WCRT of $\tau^p$ is found when the overall suspension time is the largest (i.e. $S_{ss}^{ub}$) and the self-interfering subtasks are released synchronously with $E_h^p$. That is,

$$R(\tau^p) = \sum_{h=1}^{q} C_h^p + S_{ss}^{ub} + \sum_{v_t \in self(\tau^p)} \frac{[R(\tau^p) + J_t]}{T_j} \times C_j + \sum_{v_t \in self(\tau^p)} C_t$$

**(8)**

The simplest suspension-aware analysis [21] focuses on upper-bounding the WCRT of each execution region independently. That is, the problem is reduced to a set of smaller sequential tasks by assuming that all the interfering workload releases jobs synchronously with each and every execution region. In this case, the self-interfering subtasks must be accounted once in each execution region $E_h^p$. Moreover, the suspension time has no influence on the interference generated. By using Eq. 7 to compute each $R(E_h^p)$, the WCRT of $\tau^p$ is given by

$$R(\tau^p) = S_{ss}^{ub} + \sum_{h=1}^{q} R(E_h^p)$$

**(9)**
Both of the aforementioned tests run in pseudo-polynomial time but are substantially pessimistic. The pessimism is further aggravated because multiple self-suspending tasks need to be analyzed to bound the suspensions regions of $\tau^p$. Nelissen et al. [22] proposed a MILP formulation that finds tight upper-bounds (the best known) on the WCRT of a sporadic self-suspending task with multiple suspension regions. The formulation exploits the duration of the suspension regions to accurately upper-bound the number of jobs released by each higher priority task in each execution region. Therefore, a weak characterization of the relation between the different suspension regions, and also their own bounds, compromises the quality of the solution. As it is implicit in the relation $S_{p,ub}^p \leq \sum_{h=1}^{q-1} S_h^p$, some upper-bounds on the duration of the suspension regions of $\tau^p$ are over-estimated. Hence we discuss the limitations of our generic model and propose a more robust one that adheres to such formulation.

Consider the following example: $\tau^p$ has two suspension regions, each one of them comprised of a single remote subtask assigned to the same core; let $v_t$ and $v_{t+1}$ represent those remote subtasks, while $\tau_{ss}$ denotes the inner self-suspending task formed by them. If $R(\tau_{ss}) - \tau_{ss}^{ub} < R(v_t) + R(v_{t+1})$, then the first suspension region plus the second suspension region cannot exceed $R(\tau_{ss}) - \tau_{ss}^{ub}$. Consequently, one must find the trade off that satisfies $R(\tau_{ss}) - \tau_{ss}^{ub}$, while still representing a worst-case suspension pattern for $\tau^p$.

To address this issue, let $S_h^p$ denote instead the $h^{th}$ suspension region of $\tau^p$ and be characterized by the 2-tuple $(S_{h,ss}^p, S_{h,ub}^p)$, $\forall h \in [1, q-1]$. The parameter $S_{h,ss}^p$ is a lower-bound on the duration of the suspension region $S_h^p$, whereas $S_{h,ub}^p$ is an upper-bound. We now prove the values for the individual lower-bounds.

**Lemma 3.** Let $V_{h}^p$ denote the set of remote subtasks within $E_h^p$ and $E_{h+1}^p$ of $\tau^p$. A lower-bound on the suspension region $S_h^p$ is given by $S_{h,ss}^p = \sum_{v \in V_h^p} C_t$.

**Proof.** The WCRT of $\tau^p$ is found when the response time of its remote subtasks is maximized. We must therefore prove that the WCRT of each inner self-suspending tasks $\tau_{ss}$ ($\forall o \in \text{proc}(\lambda), o \neq p$) within $\tau_p$ cannot be met if a remote subtask $v \in V_h^p$ executes for less than $C_t$ time units. The proof is by contradiction. Consider a release pattern $\sigma$ for any inner self-suspending task $\tau_{ss}$ that maximizes $R(\tau_{ss}^o)$ but where $R(E_{ss,ss}^o) < C_{ss,ss}^o$. The execution region $E_{ss,ss}^o$ of $\tau_{ss}$ corresponds to a remote subtask $v \in V_h^p$. If $E_{ss,ss}^o$ executes for its WCET $C_{ss,ss}^o = C_t$, an equivalent release pattern $\sigma'$ can be obtained by delaying each subsequent interfering job release by $C_{ss,ss}^o - R(E_{ss,ss}^o)$ time units. This means that the entire window $[R(E_{ss,ss}^o), R(\tau_{ss}^o)]$ in $\sigma$ is repeated after the initial $C_{ss,ss}^o$ time units in $\sigma'$. Clearly, $R(\tau_{ss}^o)$ is not the worst-case response time of $\tau_{ss}$ which invalidates the hypothesis. \[\Box\]

If an inner self-suspending task $\tau_{ss}^o$ is comprised of a single remote subtask $v_t \in V_h^p$ (i.e., $\tau_{ss}^o$ is sequential), than the value of $S_{h,ss}^p$ can be improved by replacing the contribution of that particular remote subtask with $R(v_t)$ instead of $C_t$. Irrespective of the type of the inner self-suspending tasks, the upper-bound $S_{h,ub}^p$ is computed as explained in Section V-C. Based on these individual bounds on the suspension regions, the following property holds.

**Property 1.** Let $R(\tau_{ss}^o)$ denote the total WCRT of the remote subtasks within $E_h^p$ and $E_{h+2}^p$ of $\tau^p$ assigned to core $o$, $\forall o \in \text{proc}(\lambda), o \neq p$. Admitting any solution, such that (1) $S_h^p + S_{h+1}^p + S_{h+2}^p \leq \sum_{\forall o \in \text{proc}(\lambda), o \neq p} R(\tau_{ss}^o)$, (2) $S_{h,ss}^p \leq S_{h,ub}^p$, and (3) $S_{h+1}^p \leq S_{h+1}^p \leq S_{h+1}^p$, upper-bounds the worst-case suspension behavior of the suspension regions $S_h^p$ and $S_{h+1}^p$ of $\tau^p$.

The reasoning behind Property 1 is that, in the general case, no technique exists yet to deem how the suspension time should be distributed between the suspension regions so that $R(\tau^p)$ is maximized. Hence, all the possible combinations should be considered. Property 1 must be applied to every sequence of suspensions regions in which an inner self-suspending task (not sequential) of $\tau^p$ can be identified. We denote by $\psi^p$ the set of constraints that restrict the duration of multiple suspension regions of $\tau^p$ together (i.e., constraints as $S_h^p + S_{h+1}^p \leq \sum_{\forall o \in \text{proc}(\lambda), o \neq p} R(\tau_{ss}^o)$). Note that an optimization problem is clearly adequate to address the complexity exposed by Lemma 2 and Property 1. Therefore, we describe how to integrate them in the formulation of [22].

In the extended MILP formulation, the duration of each suspension region in $\tau^p$ is a real variable denoted $S_h^p$, while $Y_{t,h}$ is a binary variable that indicates whether $(Y_{t,h} = 1)$ or not $(Y_{t,h} = 0)$ a self-interfering subtask $v_t \in \text{selff}(\tau^p)$ is released synchronously with the execution region $E_h^p$. Lemma 2 is formalized by Constraint 10.

\[\forall v \in \text{selff}(\tau^p) : \sum_{h=1}^{q} Y_{t,h} \leq 1, \ (10)\]

which can then be integrated in the WCRT computation of each execution region as in Constraint 11, where $NI_{j,h}$ is the number of interfering jobs of $t_j$ in $E_h^p$.

\[\forall E_h^p \in \tau^p : \ (11)\]

That is, each self-interfering subtask interferes with exactly one execution region for its WCET. The individual bounds on the duration of each suspension region are expressed by Constraint 12.

\[\forall h \in [1, q-1] : \ S_{h,ss}^p \leq S_h^p \leq S_{h,ub}^p \ (12)\]

Constraint 13 enforces that the sum of the suspension regions cannot exceed the upper-bound on the overall suspension time. For a more accurate analysis, Constraint 13 should be replaced with all the global constraints defined in $\psi^p$.

\[
\sum_{h=1}^{q-1} S_h^p \leq S_{p,ub} \ (13)
\]

The reasoning behind Property 1 is that, in the general case, no technique exists yet to deem how the suspension time should be distributed between the suspension regions so that $R(\tau^p)$ is maximized. Hence, all the possible combinations should be considered. Property 1 must be applied to every sequence of suspensions regions in which an inner self-suspending task (not sequential) of $\tau^p$ can be identified. We denote by $\psi^p$ the set of constraints that restrict the duration of multiple suspension regions of $\tau^p$ together (i.e., constraints as $S_h^p + S_{h+1}^p \leq \sum_{\forall o \in \text{proc}(\lambda), o \neq p} R(\tau_{ss}^o)$). Note that an optimization problem is clearly adequate to address the complexity exposed by Lemma 2 and Property 1. Therefore, we describe how to integrate them in the formulation of [22].
VI. Higher Priority DAG Tasks

In this section, we extend our analysis to cope with multiple partitioned DAG tasks interfering with each other. As the WCRT of a DAG task $\tau_i$ is ultimately derived through a collection of self-suspending tasks $\tau_i^p$, we restrict our attention to the worst-case interference imposed by the higher priority tasks on any $\tau_i^p$. Let $V_j^p$ denote the set of subtasks of the higher priority DAG task $\tau_j$ assigned to core $p$. We prove below that each $\tau_j$ can safely be replaced in the response time analysis of $\tau_i^p$ by a set $V_j^p$ of $n_j^p$ sequential tasks, where $n_j^p = |V_j^p|$.

Let a sequential task $\tau_{j,o} \in V_j^p$ correspond to the subtask $v_o \in V_j^p$. The task $\tau_{j,o}$ upper-bounds the worst-case request of the subtask $v_o$, $\forall o \in [1,n_j^p]$, and is defined as $\tau_{j,o} \overset{\text{def}}{=} (C_{j,o}, D_{j,o}, T_{j,o}, J_{j,o})$. The worst-case execution time $C_{j,o}$ of $\tau_{j,o}$ is given by the WCET of $v_o \in V_j^p$, that is, $C_{j,o} \overset{\text{def}}{=} C_o$. Both the deadline $D_{j,o}$ and the period $T_{j,o}$ are inherited from $\tau_j$. The parameter $J_{j,o}$ denotes the release jitter of $\tau_{j,o}$ and is defined as the difference between the WCRT of $\tau_j$ and the WCET of $v_o \in V_j^p$. Formally, $J_{j,o} \overset{\text{def}}{=} R(\tau_j) - C_o$. This transformation eliminates the dependencies regarding the interfering workload of $\tau_j$ towards $\tau_i^p$ by assuming that each subtask of $\tau_j$ assigned to core $p$ is released independently. A similar method was already proposed in [6]. Note that the method in [6] could be used here too. The results would in fact be more precise, but at the cost of additional computation complexity.

**Theorem 3.** The interference exerted by a DAG task $\tau_j \in \text{hp}(\tau_i)$ on a self-suspending task $\tau_i^p$ is upper-bounded by the sum of the interferences imposed on $\tau_i^p$ by each sequential task $\tau_{j,o} \in \tau_j^p$, where $\tau_{j,o} \overset{\text{def}}{=} (C_{j,o}, D_{j,o}, T_{j,o}, J_{j,o})$ as defined above.

**Proof.** The interference exerted by $\tau_j$ on $\tau_i^p$ is equal to the sum of the interference caused by each of the subtasks $v_o \in V_j^p$. We must therefore prove that the interference caused by each task $\tau_{j,o} \in V_j^p$ upper-bounds the interference generated by $v_o \in V_j^p$. The proof is by contradiction. Let us assume that $v_o$ causes more interference than $\tau_{j,o}$. There might be only two reasons for this to be true: (i) a job released by $v_o$ creates more interference than a job released by $\tau_{j,o}$, or (ii) $v_o$ releases more jobs than $\tau_{j,o}$ in a given time window.

Since $\tau_{j,o}$ and $v_o$ are both non-self-suspending and the WCET of $\tau_{j,o}$ is equal to the WCET of $v_o$, (i) cannot be true. As the minimum inter-arrival time of $v_o$ is identical to that of its corresponding sequential task in $V_j^p$, only their jitters may cause (ii) to be true. Now, let us compute the maximum jitter that can be experienced by the subtask $v_o$. Let $\alpha_j$ denote the arrival time of any job of $\tau_j$. Since $R(\tau_j)$ assumes that each subtask of $\tau_j$ executes for its WCET, it means that a subtask $v_o$ of $\tau_j$ cannot start executing later than $R(\tau_j) - C_o$ after $\alpha_j$ (otherwise it would complete later than $\alpha_j + R(\tau_j)$ and $R(\tau_j)$ would not be the WCRT of $\tau_j$). The release jitter of $v_o$ is therefore upper-bounded by $J_o \overset{\text{def}}{=} R(\tau_j) - C_o$. This contradicts (ii) and hence proves the lemma.

VII. Experiments

In this section, we describe experiments conducted through randomly generated task sets to evaluate (i) the performance of the different tests proposed in Section V, and (ii) the gain in terms of WCRT in comparison with an existing analysis for partitioned parallel tasks derived in the context of distributed systems. Although most of the available results for distributed systems have been implemented in the MAST tool chain [23], only the holistic analysis originally developed by Tindell and Clark [5] and refined by Palencia et al. [6] has support for multi-path constructs in the form of fork-join tasks (not general DAGs). Thus, we restrict our attention to task sets comprised of $n-1$ sequential tasks and 1 fork-join task, where the fork-join task is the task under analysis.

All the task sets were generated using the randfixedsum algorithm [24], allowing us to choose a constant total task set utilization for a given number of tasks and bounded per-task utilization. The total utilization was set to 50% of the platform capacity. For the sequential tasks, the per-task utilization ranged from [0.05, 0.70], while periods were uniformly distributed over [100, 1000]. The task execution requirements were calculated from the respective periods and utilizations. For the fork-join task, it is workload was set to half of the maximum period (i.e., 500), whereas WCET of each subtask was uniformly distributed over [1, 100]. By default, the fork-join task had 2 parallel segments with 4 subtasks within each segment. All the mapping decisions were completely random. For this reason, we study the computed WCRT and not the schedulability of the task. Thus the period of the fork-join task was arbitrarily large. We generated 100 task sets per combination of parameters, while ensuring that all the sequential tasks were always schedulable.

For the first set of experiments, we fixed the number of cores to $m = 4$ and the number of tasks to $n = 12$, while varying the number of parallel segments from 1 to 4 and the number of subtasks within a parallel segment from 2 to 8. Fig. 4a–b show the average gain (w.r.t. the WCRT) attained by our analysis comparatively to the holistic analysis, when using the three different tests for self-suspending tasks. These tests are referred to as Joint, Split and MILP, respectively to the order they were presented in Section V–E. MILP is the only test that outperforms entirely the holistic analysis, with average gains within 20 to 50%. Interestingly, Split exhibits a considerable gain when the number of parallel segments is minimized but ends up in deficit. This behavior can be justified by a higher number of self-interfering subtasks, since they are accounted once in each execution region. Joint has a drastic performance degradation when the varying parameters are increased because the number of self-suspending tasks observed in the paths grows significantly.

We then study the importance of light and heavy tasks to the inter-task interference. Hence, the second set of experiments had the number of parallel segments fixed to 2 and the number of subtasks in a parallel segment fixed to 4, while we varied the number of tasks in the range [5, 20] and the number of cores in the range [4, 10]. The results are depicted in Fig. 4c–d. Both MILP and Split outperform the holistic analysis with average gains close to 30% and 10%, respectively. The holistic
analysis was derived for arbitrary deadline systems, thus assumes that some subtasks may interfere with themselves. An additional source of pessimism is the individual worst-case scenarios assumed for the subtasks. As the utilization of the interfering tasks increase, Split becomes more competitive mainly because the upper-bounds on suspension time tend to also increase. Inversely, Joint performs very poorly with average losses within 2 to 30% as more workload is assumed to interfere with the suspensions. Although not reported here, Joint becomes a reasonable alternative solution to the MILP when the ratio between the workload of the parallel task and the periods of the interfering tasks is smaller.

VIII. CONCLUSIONS

Although parallel tasks have received recently considerable attention from the real-time community, most of the available results focus on multiprocessor global scheduling. Instead, in this paper, we studied parallel tasks under partitioned scheduling as a way to minimize the negative effects of such highly parallel models on the lower-level context-dependent timing analysis. We proposed a response time analysis for sporadic DAG tasks to be fixed-priority scheduled on a multicore platform in a partitioned fashion. As the analysis is based on the self-suspending tasks theory, we derived a method to model and characterize the worst-case scheduling scenario of a partitioned DAG task as a set of self-suspending tasks. Furthermore, we showed how to transform existing response time analysis for sporadic self-suspending tasks in uniprocessors to analyze partitioned DAG tasks; both simple and more complex techniques. Experiments among randomly generated task sets show that our approach obtains a significant gain in terms of computed WCRT comparatively to the state-of-the-art. As future work, we will consider the problem of how to map parallel tasks to cores in order to improve schedulability.

ACKNOWLEDGMENTS

This work was 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 the CISTER Research Unit (CEC/04234); also by the European Union under the Seventh Framework Programme (FP7/2007-2013), grant agreement nr. 611016 (P-SOCRATES), and the EU ARTEMIS JU within project ARTEMIS/0003/2012 - JU grant nr. 333053 (CONCERTO).

REFERENCES