Partitioned Fixed-Priority Scheduling of Parallel Tasks Without Preemptions

Daniel Casini

Alessandro Biondi

Geoffrey Nelissen*

Giorgio Buttazzo

*CISTER Research Centre

CISTER-TR-180806

2018/12/11

Partitioned Fixed-Priority Scheduling of Parallel Tasks Without Preemptions

Daniel Casini, Alessandro Biondi, Geoffrey Nelissen*, Giorgio Buttazzo

*CISTER Research Centre

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, giorgio@sssup.it

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

Abstract

The study of parallel task models executed withpredictable scheduling approaches is a fundamental problem forreal-time multiprocessor systems. Nevertheless, to date limitedefforts have been spent in analyzing the combination of partitionedscheduling and non-preemptive execution, which is arguablyone of the most predictable schemes that can be envisaged tohandle parallel tasks. This paper fills this gap by proposing ananalysis for sporadic DAG tasks under partitioned fixed-priorityscheduling where the computations corresponding to the nodes ofthe DAG are non-preemptively executed. The analysis has beenachieved by means of segmented self-suspending tasks with nonpreemptablesegments, for which a new fine-grained analysis isalso proposed. The latter is shown to analytically dominate state-of-the-art approaches. A partitioning algorithm for DAG tasksis finally proposed. By means of experimental results, the proposedanalysis has been compared against a previously-proposedanalysis for DAG tasks with non-preemptable nodes managed byglobal fixed-priority scheduling. The comparison revealed important improvements in terms of schedulability performance.

Partitioned Fixed-Priority Scheduling of Parallel Tasks Without Preemptions

Daniel Casini*, Alessandro Biondi*, Geoffrey Nelissen†, and Giorgio Buttazzo*
*Scuola Superiore Sant'Anna, Pisa, Italy
†CISTER, ISEP, Polytechnic Institute of Porto, Portugal

***Abstract*—The study of parallel task models executed with predictable scheduling approaches is a fundamental problem for real-time multiprocessor systems. Nevertheless, to date limited efforts have been spent in analyzing the combination of partitioned scheduling and non-preemptive execution, which is arguably one of the most predictable schemes that can be envisaged to handle parallel tasks. This paper fills this gap by proposing an analysis for sporadic DAG tasks under partitioned fixed-priority scheduling where the computations corresponding to the nodes of the DAG are non-preemptively executed. The analysis has been achieved by means of segmented self-suspending tasks with non-preemptable segments, for which a new fine-grained analysis is also proposed. The latter is shown to analytically dominate state-of-the-art approaches. A partitioning algorithm for DAG tasks is finally proposed. By means of experimental results, the proposed analysis has been compared against a previously-proposed analysis for DAG tasks with non-preemptable nodes managed by global fixed-priority scheduling. The comparison revealed important improvements in terms of schedulability performance.**

I. Introduction

Parallel task models have been introduced to cope with the increasing hardware parallelism offered by multiprocessor platforms. Such models are typically expressed with *directed acyclic graphs* (DAGs) where nodes represent sequential computations, also called sub-tasks, and edges represent precedence constraints between sub-tasks.

Among the possible approaches to schedule parallel tasks on a multiprocessor, *global* scheduling approaches are found at one extreme, where sub-tasks are dispatched to the available processors from a centralized (logical) ready queue, whereas *partitioned* scheduling approaches are found at the other extreme, for which each sub-task is statically allocated to one of the available processors. Global scheduling benefits from automatic load balancing of the workload at run-time, but complicates worst-case execution time (WCET) analysis, tends to introduce larger run-time overheads [1], and leads to numerous challenges when predictable access to shared resources (e.g., data objects stored in memories) is required. The main challenge though, is that state-of-the-art analysis techniques for global scheduling are still far from being effective even for simpler sporadic sequential tasks [2]. Conversely, partitioned scheduling requires finding a suitable partitioning of the tasks (typically done off-line), which, if not properly performed, may under-utilize the computing platform. However, it benefits of a simplified WCET analysis, limited run-time overhead, and many approaches to control memory contention on multiprocessors have been developed atop partitioned scheduling [3]–[5].

Besides the different strategies to dispatch sub-tasks to processors, scheduling algorithms can also be differentiated by the way the processors are contended. When priority-based algorithms are adopted, scheduling strategies can be classified into *preemptive* and *non-preemptive*: the former allows a task to take over the execution of another task, and hence tend to contain the latencies incurred by high-priority workload, whereas the latter allows running each sub-task until completion after the access to the processor has been granted. Intuitively, non-preemptive scheduling tends to decrease the system schedulability when long sub-tasks are present [6], [7].

Nevertheless, the combination of partitioned scheduling with non-preemptive execution is likely one of the most predictable solutions that can be envisaged to schedule workload on a multiprocessor. To mention just a relevant characteristic of this scheduling scheme, note that tasks can preload data in local memories (such as scratchpads) before to start executing, and then be sure that such data will not be evicted since preemptions are forbidden (e.g., as proposed in [8]).

To better understand what could be a suitable choice to schedule realistic complex parallel workloads, we performed an analysis of the parallel execution graph generated by Tensorflow, a popular framework for machine learning developed by Google, when inferring a state-of-the-art deep neural network named InceptionV3 [9] on a 8-core Intel i7 machine running at 3.5GHz[1]. The resulting graph is composed of more than 34000 nodes (i.e., unitary sequential computations), where only about 400 nodes (less than the 1.2% of the total nodes) have execution times larger than 100 microseconds. In this particular context of extreme parallelism, dispatching the computations of such large parallel graphs by following a global scheduling policy may result in large scheduling overheads, especially if worst-case latencies are concerned. Furthermore, the fact that such graphs tend to be composed of a large number of nodes where most of them have a very small execution time, suggests that a non-preemptive execution may not be harmful for the system schedulability.

Unfortunately, besides being a fundamental problem of high practical relevance, no results are available to analyze a set of parallel tasks with complex precedence constraints under partitioned fixed-priority scheduling without preemptions.

Paper contributions. This paper fills this gap by proposing an analysis for the sporadic DAG task model [10] under partitioned fixed-priority scheduling where each node (i.e., a sub-task) executes in a non-preemptive manner. As it is illustrated in Figure 1, the proposed analysis studies each

[1]The stock Tensorflow configuration relying on the `eigen` library has been used.

DAG task as a set of segmented self-suspending tasks [11] where each execution segment is non-preemptively executed. For this reason, a new fine-grained analysis for non-preemptive self-suspending tasks is also proposed, which analytically dominates the only existing and very recent result for this model [12].

Furthermore, a partitioning algorithm is proposed to found suitable allocations of the tasks to the available processors. The algorithm is general enough for being integrated with multiple fitting and task selection heuristics.

Finally, the proposed analysis approach has been compared in an experimental study with previous work targeting global scheduling of DAG tasks without preemptions. The collected results show huge improvements in terms of schedulability.

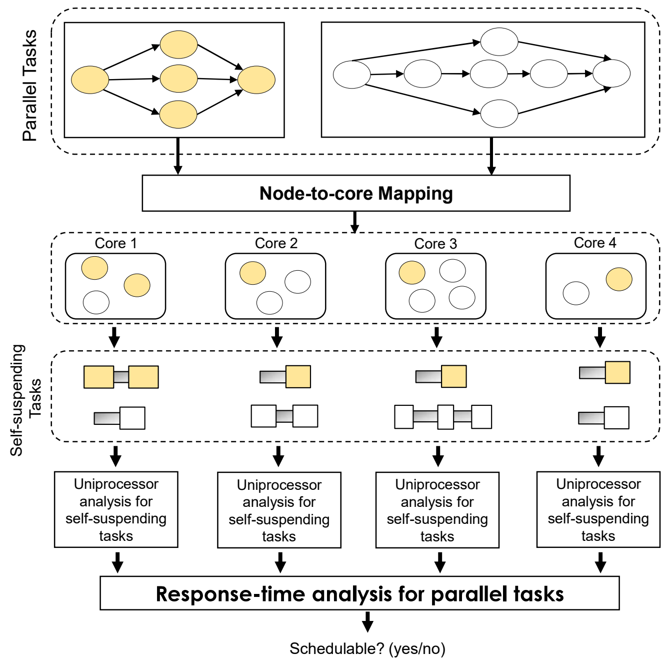
Parallel Tasks
Node-to-core Mapping
Core 1
Core 2
Core 3
Core 4
Self-suspending Tasks
Uniprocessor analysis for self-suspending tasks
Uniprocessor analysis for self-suspending tasks
Uniprocessor analysis for self-suspending tasks
Uniprocessor analysis for self-suspending tasks
Response-time analysis for parallel tasks
Schedulable? (yes/no)

Figure 1. Illustration of the analysis framework proposed in this paper.

Paper Organization. The rest of this paper is organized as follows. Section II presents the system model. Section III reviews the related work. Section IV presents a schedulability test for non-preemptive segmented self-suspending tasks, developed by combining two different analyses. Section V shows how to analyze DAG tasks by means of self-suspending tasks leveraging the results of the previous section. Section VI presents the partitioning algorithm. Section VII reports the experimental results and Section VIII concludes the paper.

II. System Model

The system considered in this paper consists of a set Γ of n real-time tasks, each modelled as a directed acyclic graph (DAG) [10]. Tasks are released sporadically and are executed on a multiprocessor platform composed of M identical processors $p_1, \ldots, p_M$. Each task $\tau_i = (V_i, E_i, T_i, D_i, \pi_i)$ is characterized by a set V_i of nodes (or vertices), a set of directed edges E_i, a minimum inter-arrival time T_i, a relative deadline $D_i \leq T_i$, and a priority π_i. The j-th node of task τ_i is denoted by $v^i_j \in V_i$ and is statically allocated to processor $\mathcal{P}(v^i_j)$. The subset of nodes allocated to p_k is denoted with $V_i(k) \subseteq V_i$. A node v^i_j is a sub-task of τ_i and is characterized by a worst-case execution time (WCET) C^i_j. Nodes are connected by edges. Edge $e^i_{j,z} \in E_i$ connects node v^i_j to node v^i_z of τ_i and defines a *precedence constraint* between the two nodes, i.e., v^i_z can start executing only after the completion of v^i_j. A task is said to be *pending* when it has at least one released but uncompleted node, while it is said to be *self-suspended* on a core p_k when it is pending and none of the uncompleted nodes of τ_i allocated to p_k have their precedence constraints satisfied.

Each task τ_i releases an infinite sequence of instances (jobs) with a minimum inter-arrival time T_i. Each of such jobs must execute all nodes in V_i within D_i units of time after its release. Each job of task τ_i must respect the precedence constraints between its nodes given by the edges in E_i. All the nodes execute with the same priority π_i and are simultaneously released when their job is released, although some of them may not be ready due to precedence constraints.

Tasks are executed under *partitioned fixed-priority* scheduling, where each node executes in a *non-preemptive* fashion. Note that the fact that nodes are non-preemptive does not prevent a high priority task to delay the execution of a lower priority task.

The WCET of each node is assumed to include a bound on the worst-case delay incurred by the node in accessing shared memories or to load/unload local memories (such as scratch-pads). Note that the combination of partitioned scheduling with non-preemptive execution significantly simplifies the WCET analysis and allows deriving tighter bounds on memory-related delays [13] with respect to other scheduling approaches (e.g., preemptive scheduling [14]–[16]).

The set of tasks that have at least one node allocated to p_k is denoted by Γ_k. Tasks are independent, i.e., they do not access mutually-exclusive shared resources. Mutually-exclusive resources shared by nodes of the same tasks are handled with precedence constraints or wait-free mechanisms[2].

For each node, the set of immediate predecessors is defined as $\mathsf{ipred}(v^i_s) = \{v^i_j \in V_i : \exists\ (v^i_j, v^i_s) \in E_i\}$, whereas the set of immediate successors is defined as $\mathsf{isucc}(v^i_s) = \{v^i_j \in V_i : \exists\ (v^i_s, v^i_j) \in E_i\}$. Similarly, the sets of predecessors $\mathsf{pred}(v^i_s)$ and successors $\mathsf{succ}(v^i_s)$ denote precedence relations that are either direct (i.e., by means of an edge) or transitive (i.e., by means of a set of edges involving intermediate nodes).

A node without incoming edges is referred to as a *source node*, whereas a node without outgoing edges is denoted as a *sink node*. For the sake of simplicity, this paper assumes a single sink and source node. Whenever this assumption does not hold, any DAG with multiple source/sink nodes can always be transformed into a DAG with a single source/sink node by adding an extra *dummy* source/sink node with computation time equal to zero.

A formal definition of *path* is also required.

Definition 1: A path $\lambda_{i,z} = (v^i_s, \ldots, v^i_e)$ of a DAG task τ_i is an ordered sequence of nodes in V_i where v^i_s and v^i_e

[2]Note that the relaxation of this assumption requires support for locking protocols, which is left as future work.

are source and sink nodes, respectively, and (i) $\forall v_j^i \in \lambda_{i,z} \setminus v_e^i, \exists! v_a^i \in \lambda_{i,z}$ such that $(v_a^i, v_j^i) \in E_i$; and (ii) $\forall v_j^i \in \lambda_{i,z} \setminus v_s^i, \exists! v_a^i \in \lambda_{i,z}$ such that $(v_j^i, v_a^i) \in E_i$.

Informally, a path is an ordered sequence of nodes starting from a source and ending in a sink where there is a direct precedence constraint between any two adjacents nodes. The set of all paths of a task is denoted by $\mathsf{paths}(\tau_i)$.

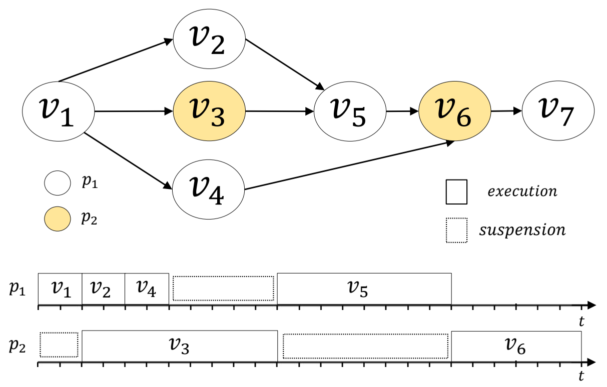

Figure 2. Example scheduling of a DAG task partitioned onto two processors. Nodes v_1, v_2, v_4, v_5, and v_7 are allocated to p_1, while nodes v_3 and v_6 are allocated to p_2. Note that the task is self-suspended on a processor p_k whenever, due to uncompleted predecessors on remote processors, there are no nodes allocated to p_k that are ready for being executed.

Figure 2 reports a sample schedule of a single task τ_i with 6 nodes partitioned onto two processors. Note that the task is self-suspended in a processor whenever there are no ready nodes to be executed due to unsatisfied precedence constraints.

To help the reader in following the adopted notation, Table I reports the main symbols introduced in the system model.

Table I. TABLE OF SYMBOLS

Symbol	**Description**
p_k	k-th processor
Γ_k	set of tasks with at least one node allocated on p_k
τ_i	i-th task
π_i	priority of τ_i
D_i	relative deadline of τ_i
T_i	min. inter-arrival time of τ_i
v_j^i	j-th node of τ_i
C_j^i	WCET of v_s^i
V_i	set of nodes of τ_i
$V_i(k)$	subset of nodes of τ_i allocated to p_k
$\lambda_{i,z}$	z-th path of τ_i
$\mathsf{paths}(\tau_i)$	set of all paths of τ_i
$\mathcal{P}(v_s^i)$	processor in which v_s^i is allocated to
$\mathsf{ipred}(v_s^i)$	immediate predecessors of v_s^i
$\mathsf{isucc}(v_s^i)$	immediate successors of v_s^i
$\mathsf{pred}(v_s^i)$	predecessors of v_s^i
$\mathsf{succ}(v_s^i)$	successors of v_s^i

III. RELATED WORK

The problem of executing real-time parallel tasks upon a multiprocessor platform has been addressed in many works during the last decade, also considering different task models. For instance, in the fork-join task model [17], [18], tasks are composed of interleaved sequences of sequential and parallel segments, where each segment has a precedence constraint with respect to preceding one. The DAG task model, introduced by Saifullah et al. [10], reduces the above restriction by representing each task by means of a direct acyclic graph. Most of the works targeting the DAG model addressed global preemptive scheduling. Three notable examples are the works by Bonifaci et al. [19], Baruah [20] and Fonseca et al. [21], in which a schedulability test for EDF and Deadline Monotonic is proposed and improved, respectively. To the best of our knowledge, the only (not flawed) work explicitly targeting DAG tasks scheduled under partitioned fixed-priority policy has been recently proposed by Fonseca et al. [22]. In their work, the authors proposed a response time analysis based on self-suspending task theory. Concurrently, Fonseca et al. [23], Melani et al. [24] and Baruah et al. [25] introduced the conditional DAG model, in which each task can consider different execution flows.

Much closer to this work, Serrano et al. [26] provided an analysis for limited-preemptive globally-scheduled DAG tasks. Note that the authors denoted with limited-preemptive scheduling the case in which each node of the DAG tasks is non-preemptively executed (i.e., the same execution model assumed here). The main difference with respect to this paper is that they considered global scheduling in place of partitioned scheduling.

Concerning non-preemptive uniprocessor scheduling, Davis et al. [27] proposed a revised analysis for the CAN bus (a notable example of non-preemptive scheduling), individuating the self-pushing phenomenon. Later, Bril et al. [28] found the same phenomenon when studying deferred preemptions, providing a revised response time analysis. Yao et al. [29] considered the problem of analyzing the feasibility of a task set under uniprocessor fixed priority scheduling with fixed preemption points. Nasri and Brandenburg [30] presented an effective and practical implementation of non-preemptive scheduling. Nasri et al. [7] proposes a schedulability test for job sets scheduled under global non-preemptive scheduling. It drastically improves the accuracy of the analysis in comparison to the state-of-the-art, but requires to know the arrival times of each jobs. Further information about limited preemptive scheduling is reported in the survey by Buttazzo et al. [31].

The problem of scheduling parallel tasks has also been addressed in the context of distributed systems, which mandates partitioned scheduling. One of the first work is due to Fohler and Ramamritham [32], which proposed an approach to achieve a static schedule composed of tasks with precedence constraints. In the context of non-static scheduling, a seminal work is due to Tindell and Clark [33], who presented a holistic schedulability analysis for sequences of events called transactions, preemptively scheduled with fixed priorities. Later, Palencia et al. refined their analysis, considering offsets [34] and precedence relations [35]. Finally, Jayachandran and Abdelzaher [36] addressed the problem of transforming a single DAG task scheduled in a distributed system in an equivalent uniprocessor task set, providing a job-level analysis for the end-to-end delay.

Note that this work also builds upon results for self-suspending tasks. The interested reader can refer to the survey of Chen et al. [37], while specific references are accordingly provided in the following sections.

IV. Analyzing Non-Preemptive Self-Suspending Tasks

Following the results presented in [22], a parallel task scheduled under partitioned scheduling can be analyzed by means of segmented self-suspending tasks. For instance, consider the example shown in Figure 2. By looking at the scheduling of the parallel task separately on each core, it is easy to observe that the resulting behavior is essentially the one of a self-suspending task, i.e., a computational activity that alternates execution phases (in this case, the execution of a node) with self-suspension phases (in this case, the waiting for a precedence constraint to be satisfied). The analysis of self-suspending tasks is therefore fundamental for analyzing the timing behavior of parallel tasks under partitioned scheduling.

Unfortunately, all results in the literature concerning the analysis of self-suspending tasks targeted preemptive scheduling, with the only notable exception being a very recent work by Dong et al. [12]. Analyzing tasks in the presence of both self-suspensions and non-preemptive execution poses significant challenges that require to be addressed with new specialized techniques that exploit the specific characteristics of the combination of these two features. The objective of this section is to propose fine-grained analysis techniques for segmented self-suspending tasks where each execution segment is *non-preemptable*, which will be used in the next section as the foundation for analyzing parallel tasks.

To make the paper self-consistent, it is necessary to begin by introducing the segmented self-suspending task model. A segmented self-suspending task τ_i^{ss} is characterized by an ordered sequence of N_i^S execution segments alternated by self-suspensions, both with bounded duration and represented by the tuple $\langle C_{i,1}, S_{i,1}, \ldots, S_{i,N_i^S-1}, C_{i,N_i^S}\rangle$, where $C_{i,j}$ denotes the WCET of the j-th execution segment of τ_i^{ss} and $S_{i,j}$ denotes the maximum duration of the j-th self-suspension of τ_i^{ss}. Like the parallel tasks introduced in Section II, each self-suspending task τ_i^{ss} is released sporadically with minimum inter-arrival time T_i, relative deadline $D_i \leq T_i$, and fixed priority π_i. The symbol $lp(i)$ denotes the set of tasks with priority lower than π_i, whereas $hep(i)$ denotes the set of tasks with priority higher than or equal to π_i. Once a self-suspending task is released, the first execution segment is also released. If the j-th segment of τ_i^{ss} completes its execution at time t, the $(j+1)$-th segment is released after time t and no later than time $t + S_{i,j}$.

The analysis techniques presented in the following sections assume the knowledge of a vector $\overline{\mathbf{R}}$ of safe response-time bounds for each execution segment. Since the response time of the last segment corresponds to the response-time of the task itself, the notation $\overline{R}_i = \overline{R}_{i,N_i^S}$ is adopted to reduce clutter.

Section IV-A presents an approach to compute the set of low-priority execution segments that can block a task. Then, two different approaches are presented to compute response-time bounds of self-suspending tasks with non-preemptive execution segments. The first one aims at bounding the response time of the whole task with an holistic approach (Section IV-B), while the second one is based on bounding the worst-case response time of each individual segment (Section IV-C). These two approaches are then combined in an hybrid response-time analysis algorithm (Section IV-D). Finally, Section IV-E demonstrates that the analysis proposed in this paper analytically dominates the one proposed in [12].

A. Computing non-preemptive blocking

Since preemptions are forbidden, a task τ_i^{ss} can be blocked by an execution segment of a lower-priority task whenever it releases a new execution segment. To cope with this blocking phenomenon, we identify a superset of the lower-priority execution segments that can potentially block task τ_i^{ss}. To this end, let w be a window of length t at the beginning of which τ_i^{ss} is released and in which τ_i^{ss} is pending. Let $LPS(t)$ be the multiset of segments of tasks in $lp(i)$ that can overlap with w and that may block τ_i^{ss} for their WCET. The following lemma establishes how to compute a multiset that includes the WCETs of such segments.

Lemma 1: The WCET of each execution segment into $LPS(t)$ is included into the multiset[3]

$$\mathcal{C}_i(t, \overline{\mathbf{R}}) = \biguplus_{\tau_l^{\text{ss}} \in lp(i)} \biguplus_{j=1}^{N_l^S} \{C_{l,j}\} \otimes \eta_{l,j}(t, \overline{\mathbf{R}}), \tag{1}$$

where $\eta_{l,j}(t, \overline{\mathbf{R}}) = 1 + \left\lfloor \frac{t + \overline{R}_{l,j} - C_{l,j}}{T_l} \right\rfloor$.

Proof: Let $X_{l,j}$ denote the j-th execution segment of τ_l^{ss}. Assume that $X_{l,j}$ starts executing a first job ϵ time units before the start of w, where ϵ is infinitesimally small. Clearly, that first job of $X_{l,j}$ may block τ_i^{ss} for $C_{l,j} - \epsilon$ time units, which tends to $C_{l,j}$ when ϵ tends to 0. Now, because $X_{l,j}$'s worst-case response time is upper bounded by $\overline{R}_{l,j}$, and because the minimum inter-release time of $X_{l,j}$ is T_l, the next job of $X_{l,j}$ is released at the earliest $T_l - (\overline{R}_{l,j} - C_{l,j}) - \epsilon$ time units away from the beginning of w. Since w has length t and $X_{l,j}$ releases jobs with minimum inter-release time T_l, execution segment $X_{l,j}$ may start executing x additional jobs in w, where x is computed as follows $x = \left\lceil \frac{t - T_l + (\overline{R}_{l,j} - C_{l,j}) + \epsilon}{T_l} \right\rceil = \left\lceil \frac{t + (\overline{R}_{l,j} - C_{l,j}) + \epsilon}{T_l} \right\rceil - 1 = \left\lfloor \frac{t + (\overline{R}_{l,j} - C_{l,j})}{T_l} \right\rfloor$.

Each of such x jobs may execute for their WCET. Therefore, each execution segment $X_{l,j}$ may have $(x+1) = 1 + \left\lfloor \frac{t + (\overline{R}_{l,j} - C_{l,j})}{T_l} \right\rfloor$ jobs whose execution overlap with w and that may block τ_i^{ss} for their WCET. This proves the lemma. ■

Since a self-suspending task can be blocked by lower-priority segments for a bounded number of times, it is convenient to define $\mathcal{B}_i(k, t, \overline{\mathbf{R}})$ as the multiset that includes the k *largest* elements into $\mathcal{C}_i(t, \overline{\mathbf{R}})$ padded with zeros if $|\mathcal{C}_i(t, \overline{\mathbf{R}})| \leq k$. The multiset $\mathcal{B}_i(k, t, \overline{\mathbf{R}})$ will be used as a building block for the analysis techniques presented in the following sections.

[3]The operator $\uplus$ represents the union between multisets, e.g., $\{1,1\} \uplus \{1,2\} = \{1,1,1,2\}$, and the product operator $\otimes$ multiplies the number of instances of every element in the multiset to which it is applied, e.g., $\{1,2,3\} \otimes 3 = \{1,1,1,2,2,2,3,3,3\}$.

B. Approach A: holistic analysis

A first lemma is provided to compute the maximum lower-priority blocking incurred by a self-suspending task.

Lemma 2: The maximum blocking incurred by a non-preemptive self-suspending task τ_i^{ss} in an arbitrary time window of length t is bounded by

$$\sum_{b \in \mathcal{B}_i(N_i^S, t, \overline{\mathbf{R}})} b.$$

Proof: A self-suspending task τ_i^{ss} can be blocked by a lower-priority task every time it releases an execution segment, and hence no more than N_i^S times. By Lemma 1 and the definition of $\mathcal{B}_i(k, t, \overline{\mathbf{R}})$, multiset $\mathcal{B}_i(N_i^S, t, \overline{\mathbf{R}})$ includes the N_i^S largest execution segments of lower-priority tasks that can overlap with τ_i^{ss} while the latter is pending in a time window of length t. Therefore, the sum of the elements in $\mathcal{B}_i(N_i^S, t, \overline{\mathbf{R}})$ yields a safe bound on the blocking incurred by τ_i^{ss}. ■

Then, a bound on the high-priority interference suffered by a self-suspending task in a time window of length t is also established.

Lemma 3: The maximum interference $I_i(t)$ suffered by a non-preemptive self-suspending task τ_i^{ss} in an arbitrary time window of length t is bounded by

$$I_i(t) = \min\left\{ \sum_{\tau_h^{\text{ss}} \in hep(i)} \sum_{r=1}^{N_h^S} \left(\left\lfloor \frac{t + \overline{R}_{h,r} - C_{h,r}}{T_h} \right\rfloor + 1 \right) C_{h,r} \ ; \ \sum_{\tau_h^{\text{ss}} \in hep(i)} \left(\left\lfloor \frac{t + \overline{R}_h - C_h}{T_h} \right\rfloor + 1 \right) C_h \right\}. \quad (2)$$

where $C_h = \sum_{r=1}^{N_h^S} C_{h,r}$.

Proof: For fixed priority non-preemptive and limited-preemptive task sets, the worst-case interference generated by a higher-priority task τ_h in a window of length t is upper-bounded by the maximum amount of workload that τ_h may execute in a window of length $(t+\epsilon)$ where ϵ is an arbitrarily small positive number [27], [28].

Let $\overline{R}_h$ be an upper bound on the worst-case response time (WCRT) of a higher priority self-suspending task τ_h^{ss} with minimum inter-arrival time T_h. An upper-bound on the maximum workload it may execute in a time window of length $(t+\epsilon)$ is given by $W_h(t) = \left\lceil \frac{t+\epsilon+\overline{R}_h - C_h}{T_h} \right\rceil C_h$. This happens when (1) the first job of τ_h^{ss} has its start time aligned with the start of the window; (2) that job starts executing exactly C_h time units before reaching its WCRT; (3) the next jobs of τ_h^{ss} are released as early as possible; and (4) all jobs execute for their WCET.

Using properties of the floor and ceil operators, we can rewrite that equation as $W_h(t) = \left(\left\lfloor \frac{t+\overline{R}_h - C_h}{T_h} \right\rfloor + 1 \right) C_h$ and getting rid of the ϵ term. Summing the contribution of all higher-priority tasks, this proves the second term in the min operator of Equation (2).

Similarly, each execution segment $X_{h,r}$ of a task τ_h^{ss} may be modelled as an independent sporadic task. Therefore, applying the same argument then above, we have that the maximum workload $X_{h,r}$ may execute in a window of length $(t+\epsilon)$ is given by $\left(\left\lfloor \frac{t+\overline{R}_{h,r} - C_{h,r}}{T_h} \right\rfloor + 1 \right) C_{h,r}$, where $\overline{R}_{h,r}$ is an upper bound on the WCRT of $X_{h,r}$.

Therefore, summing the contribution of all execution segments of all higher priority tasks, we prove the first term of Equation (2). Hence concluding the proof. ■

Finally, the results of Lemma 2 and Lemma 3 can be combined in the following theorem to bound the response time of a self-suspending task.

Theorem 1: The response-time of a self-suspending task τ_i^{ss} is bounded by

$$R_i = R_i' + C_{i,N_i^S}, \quad (3)$$

where R_i' is given by the least positive fixed point of the following recursive equation:

$$R_i' = \sum_{j=1}^{N_i^S - 1} (C_{i,j} + S_{i,j}) + \sum_{b \in \mathcal{B}_i(N_i^S, R_i', \overline{\mathbf{R}})} b + I_i(R_i'), \quad (4)$$

where $I_i(t)$ is given by Lemma 3.

Proof: Since segments are executed in a non-preemptive manner, the response time of a task is equivalent to the latest release time of its last segment (relative to the task release), plus the WCET of that segment. Note that the release time of the last segment is determined by the sum of (i) the WCETs of the preceding segments, (ii) the sum of the maximum duration of the task suspensions, and (iii) the maximum cumulative delay incurred by all segments. The first term in Equation (4) accounts for (i) and (ii). Then, analogously to standard response-time analysis, a recursive equation can be formulated to cope with (iii), provided that safe bounds are adopted for the low-priority blocking and the high-priority interference that the task can incur.

Given a tentative response-time R_i' for τ_i^{ss}, by Lemma 2 the second term of Equation (4) provides a bound on the maximum low-priority blocking. By Lemma 3, the third term of Equation (4) provides a bound on the maximum high-priority interference. The WCET of the last segment is finally accounted for in Equation (3). Hence the theorem follows. ■

Corollary 1: The response time of the j-th execution segment of task τ_i^{ss} is upper-bounded by

$$R_{i,j} = R_i - \sum_{k=j+1}^{N_i^S} C_{i,k} - \sum_{k=j}^{N_i^S - 1} S_{i,k} \quad (5)$$

where R_i is given by Theorem 1.

Proof: After the j-th execution segment of task τ_i^{ss} completes, τ_i^{ss} must still execute all subsequent suspension and execution segments. Therefore, if by contradiction we were to assume that the j-th execution segment of τ_i^{ss} were to complete at time $t > R_{i,j}$, then τ_i^{ss} would complete no earlier than $t + \sum_{k=j+1}^{N_i^S} C_{i,k} + \sum_{k=j}^{N_i^S - 1} S_{i,k} > R_i$ in the worst case. Therefore, R_i would not be a WCRT for τ_i^{ss}, a contradiction. ■

C. Approach B: segment-level analysis

Differently from the approach presented above, this section aims at deriving an individual response-time bound for each execution segment. To precisely identify the analysis window for each segment, the following lemma establishes a bound on the maximum amount of time it can last from the release of a task and the release of one of its segments.

Lemma 4: The latest release time of the k-th segment of τ_i^{ss}, relative to the release time of the task itself, is upper-bounded by

$$r_{i,k}(\overline{\mathbf{R}}) = \begin{cases} 0 & \text{if } k = 0, \\ \overline{R}_{i,k-1} + S_{i,k-1} & \text{otherwise.} \end{cases}$$

Proof: The first segment ($k = 0$) is released when the task is released, hence $r_{i,k}(\overline{\mathbf{R}}) = 0$. For the arbitrary k-th segment to be released, with $k > 0$, the $(k-1)$-th segment must be completed and the suspension between the $(k-1)$-th and the k-th segment must be elapsed. The lemma follows by noting that the $(k-1)$-th segment completes no later than $\overline{R}_{i,k-1}$ as the latter is by definition an upper bound on its response time, and that the above mentioned suspension has a duration bounded by $S_{i,k-1}$. ■

Then, Lemma 5 below, is provided to bound the delay incurred by a segment in the presence of both low-priority blocking and high-priority interference.

Lemma 5: Assuming that an execution segment of τ_i^{ss} is blocked by a low-priority task for b time units, a bound $\Delta_i(b, \overline{\mathbf{R}})$ on the maximum time it can be delayed from its release up to the time it starts executing is given by the least positive fixed point of the following recursive equation:

$$\Delta_i = b + I_i(\Delta_i), \tag{6}$$

where $I_i(t)$ is given by Lemma 3.

Proof: It directly follows from the standard response-time analysis for non-preemptive tasks [27] after recalling Lemma 3. ■

Note that Lemma 5 is independent from the actual segment of τ_i^{ss} that is delayed. It is because segments of τ_i^{ss} are non-preemptive. Hence, different from the preemptive case, the delay they suffer does not depend on their execution time.

Lemmas 4 and 5 are then combined in the following theorem to bound the response time of an execution segment.

Theorem 2: The response-time of the k-th segment of a self-suspending task τ_i^{ss} is bounded by

$$R_{i,k} = \sum_{j=1}^{k} C_{i,j} + \sum_{j=1}^{k-1} S_{i,j} + \sum_{b \in \mathcal{B}_i(k, r_{i,k}(\overline{\mathbf{R}}), \overline{\mathbf{R}})} \Delta_i(b, \overline{\mathbf{R}}), \tag{7}$$

where $r_{i,k}(\overline{\mathbf{R}})$ is given by Lemma 4 and $\Delta_i(b, \overline{\mathbf{R}})$ is the fixed-point given by Lemma 5.

Proof: The response-time of the k-th segment is given by the sum of (i) the WCET of the k first segments of τ_i^{ss}, (ii) the sum of the maximum duration of the suspensions preceding the k-th segment, and (iii) the maximum delay incurred by the first k segments. The first two terms in the equation above account for (i) and (ii). By Lemma 4, $r_{i,k}(\overline{\mathbf{R}})$ upper-bounds the latest release of the k-th segment, hence by Lemma 1 multiset $\mathcal{B}_i(k, r_{i,k}(\overline{\mathbf{R}}), \overline{\mathbf{R}})$ includes the k largest WCETs of the low-priority execution segments that can potentially block the first k segments of τ_i^{ss}. By leveraging the results of Lemma 5 applied for each blocking term into $\mathcal{B}_i(k, r_{i,k}(\overline{\mathbf{R}}), \overline{\mathbf{R}})$ we get a bound on the delay incurred by each of the k first execution segments. These delays are summed up in the third term of the above equation. Hence the theorem follows. ■

Clearly, to compute a bound on the response-time of a self-suspending task τ_i^{ss} it is sufficient to use the above theorem for its last segment (i.e., the N_i^S-th execution segment). However, $r_{i,N_i^S}(\overline{\mathbf{R}})$ is recursively defined (see Lemma 4) and depends on the response time of all other segments of τ_i^{ss}. Therefore, Theorem 2 must be called for all execution segments of τ_i^{ss} starting from the first one.

Both Theorem 1 and Theorem 2 offer safe response-time bounds for each task τ_i^{ss} and its execution segments: hence, the minimum of the two is still a safe response-time bound.

It is worth mentioning that the analysis of non-preemptive periodic/sporadic tasks *without* self-suspensions usually requires to deal with the (so-called) *self-pushing* phenomenon [27]. Yet, none of the above theorems make specific assumptions concerning self-pushing: this is due to the fact that Lemma 3 provides a conservative bound on the number of interfering high-priority jobs that holds as long as fixed-priority scheduling is used, i.e., independently of whether preemption is enabled or not. More precisely, this is possible due to the use of the jitter-like term $R_{i,k} - C_{i,k}$, which, in this case, is conservative enough to cope with both (i) the self-suspending behavior [37] and (ii) possible task self-pushing.

D. Response-time analysis algorithm

This section shows how Theorem 1 and Theorem 2 can be combined to implement a schedulability test for segmented self-suspending tasks whose segments are non-preemptively executed. The main idea is to increasingly refine the response-time bounds in $\overline{\mathbf{R}}$ with an iterative algorithm that always takes the minimum of the response-time bounds provided by the two theorems.

Algorithm 1 reports the pseudo-code for the proposed approach. First, the response-time bounds $\overline{\mathbf{R}}$ are initialized (line 3) for each segment to ensure a sufficient condition for schedulability, i.e., by setting them equal to the deadline of the task minus the sum of the WCETs of the following segments. If a segment was to finish later than that time, then the system would not be schedulable since the task WCRT would then be larger than its deadline (see Corollary 1). Then, within a while loop the algorithm computes the response time of each task by leveraging Theorem 1, and for each segment by leveraging Theorem 2. The minimum between that last bound and the bound defined in Corollary 1 is then taken as an upper bound for the response time of each segment (line 12). The algorithm terminates as soon as the obtained response-time bounds allow deeming the system schedulable (line 16). If at least one of the new response-time bounds obtained from this step is lower than those in $\overline{\mathbf{R}}$, then the algorithm continues to iterate; otherwise no further improvement is possible and hence the system is deemed unschedulable. Finally, at every iteration, the algorithm updates the bounds in $\overline{\mathbf{R}}$ for which an improved bound has been found (line 21). Convergence of the algorithm is guaranteed by the fact that all the formulas involved in Theorems 1 and 2 are monotone in the components of vector $\overline{\mathbf{R}}$.

Algorithm 1 Schedulability test for segmented self-suspending tasks with non-preemptable execution segments.

```
1:  procedure isSchedulable($\Gamma^{ss}$)
2:      $\forall \tau_i^{ss} \in \Gamma^{ss}, \forall j = 1, \dots, N_i^S$,
3:          $\overline{R}_{i,j} = D_i - \sum_{k=j+1}^{N_i^S} C_{i,k} - \sum_{k=j}^{N_i^S-1} S_{i,k}$.
4:      atLeastOneUpdate $\leftarrow$ TRUE
5:      while (atLeastOneUpdate==TRUE) do
6:          atLeastOneUpdate $\leftarrow$ FALSE
7:          for all $\tau_i^{ss} \in \Gamma^{ss}$ do
8:              $R_i^A \leftarrow$ Theorem 1
9:              for $j = 1, \dots, N_i^S$ do
10:                 $R_{i,j}^B \leftarrow$ Theorem 2
11:                 $R_{i,j} = \min\Big\{R_{i,j}^B,\ R_i^A - \sum_{k=j+1}^{N_i^S} C_{i,k}$
12:                                 $- \sum_{k=j}^{N_i^S-1} S_{i,k}\Big\}$
13:                 if $R_{i,j} < \overline{R}_{i,j}$ then
14:                     atLeastOneUpdate $\leftarrow$ TRUE
15:                 end if
16:             end for
17:         end for
18:         if $\forall \tau_i \in \Gamma^{ss},\ R_{i,N_i^S} \leq D_i$ then
19:             return TRUE
20:         end if
21:         $\forall \tau_i \in \Gamma^{ss}, \forall j = 1, \dots, N_i^S,\ \overline{R}_{i,j} \leftarrow \min\{R_{i,j}, \overline{R}_{i,j}\}$
22:     end while
23:     return FALSE
24: end procedure
```

E. Analytical dominance

This section demonstrates that the response-time bound provided by Theorem 1 analytically dominates the jitter-based analysis recently proposed by Dong et al. [12], which is reported in the following lemma.

Lemma 6: [Lemma 4 in [12]] The response time of a self-suspending task is bounded by the solution of the following recursive equation:

$$R_i = C_i + S_i + N_i^S \times C_{\text{MAX}}^{\text{LP}} + \sum_{\tau_h \in hep(i)} \left\lceil \frac{R_i + D_h - C_h}{T_h} \right\rceil C_h, \quad (8)$$

where $C_i = \sum_{j=1}^{N_i^S} C_{i,j}$, $S_i = \sum_{j=1}^{N_i^S-1} S_{i,j}$, and $C_{\text{MAX}}^{\text{LP}}$ the maximum amount of delay incurred by τ_i when it is blocked by a low-priority task.

Lemma 7 proves that Theorem 1 dominates Lemma 6.

Lemma 7: The response time bound that can be obtained by solving Equation (3) is always smaller than or equal to the bound obtained with Equation (8).

Proof: Let us rewrite R_i by injecting Equation (4) into Equation (3), we obtain

$$R_i = C_{i,N_i^S} + \sum_{j=1}^{N_i^S-1} (C_{i,j} + S_{i,j}) + \sum_{b \in \mathcal{B}_i(N_i^S, R_i', \overline{\mathbf{R}})} b + I_i(R_i - C_{i,N_i^S}). \quad (9)$$

To prove the lemma we show that all the terms of Equation (9) are lower than or equal to those of Equation (8), hence establishing a dominance relationship between the corresponding fixed points.

Since $C_i = \sum_{j=1}^{N_i^S} C_{i,j}$ and $S_i = \sum_{j=1}^{N_i^S-1} S_{i,j}$, note that the first two terms in Equation (9) are equal to the first two terms in Equation (8). Furthermore, since a task is schedulable only if its response time is smaller than D_i, we have $\overline{R}_h \leq D_h$. Assuming $C_{i,N_i^S} > 0$, we thus have $\sum_{\tau_h \in hep(i)} \left\lceil \frac{t+D_h-C_h}{T_h} \right\rceil C_h \geq \sum_{\tau_h \in hep(i)} \left\lceil \frac{t+\overline{R}_h-C_h}{T_h} \right\rceil C_h \geq \sum_{\tau_h \in hep(i)} \left(\left\lfloor \frac{t+\overline{R}_h-C_h-C_{i,N_i^S}}{T_h} \right\rfloor + 1 \right) C_h \geq I_i(t - C_{i,N_i^S})$, where $I_i(\cdot)$ is defined as in Equation (2) (note the second term in the minimum). Finally, by definition $\mathcal{B}_i(N_i^S, R_i', \overline{\mathbf{R}})$, is the multiset that contains the WCETs of the N_i^S largest execution segments that can block τ_i. Recalling the definition of $C_{\text{MAX}}^{\text{LP}}$ provided by Dong et al. (Lemma 6), it follows that the sum of the elements into $\mathcal{B}_i(N_i^S, R_i', \overline{\mathbf{R}})$ is lower than or equal to $N_i^S \times C_{\text{MAX}}^{\text{LP}}$. Hence the lemma follows. ■

V. Analysis of DAG tasks without preemptions

As discussed in Section IV, when partitioned scheduling is adopted, each core perceives the execution of a DAG task as an interleaved sequence of execution and suspension regions. Indeed, each path $\lambda_{i,z}$ (i.e., a valid sequence of nodes) of the DAG describing τ_i can easily be mapped to a corresponding self-suspending task, provided that the first and the last node of the path are allocated to the same core p_k. In this case, nodes in $\lambda_{i,z}$ executing on p_k can be seen as execution regions, while the execution of the nodes mapped on other cores correspond to suspension regions.

The major issue in following this approach resides in the fact that the inter-core dependencies among nodes must correctly be reflected to the parameters of self-suspending tasks—specifically to their suspension times. In fact, given a path with nodes that span across different processors, the suspension times seen on a processor p_k depend on the response times of the nodes allocated to the other processors $\neq p_k$, which in turn may be delayed by nodes executing on p_k hence originating a circular dependency.

A methodology to overcome such dependency has been recently proposed by Fonseca et al. [22], who studied this problem in the context of *preemptive* DAG tasks. In this paper we build upon the results presented in [22] to enable the analysis of DAG tasks without preemptions.

To make the paper self-consistent, it is necessary to concisely recall the method presented in [22].

A. Summary of the method by Fonseca et al. [22]

The method proposed in [22] separately computes the response time $R(\lambda_{i,z})$ of each path $\lambda_{i,z}$, and derives the overall response time of each DAG task τ_i as:

$$R_i = \max_{\lambda_{i,z} \in \mathsf{paths}(\tau_i)} \{R(\lambda_{i,z})\}. \qquad (10)$$

Each path is independently studied with the recursive algorithm reported in Appendix A (Algorithm 5). At a high level, the algorithm recursively divides a path into smaller ones for which the suspension time is calculated first. The recursion stops (base case) when the analyzed sub-path contains only nodes allocated to a single processor (line 7). The response time for such sub-path can be directly computed as it can be modelled as self-suspending tasks with null suspension times. In our case, Theorems 1 and 2 can be applied to analyze the resulting self-suspending tasks. Each time the response time of a sub-path is found, it is propagated in a global data structure (denoted by RTs) that will be used to assign suspension times (see Section V-B) of sub-paths analyzed at a shallower level of the call tree of recursive Algorithm 5. Then, whenever a sub-path that represents a valid self-suspending task is found (i.e., the first and the last node are allocated to the same processor), the corresponding response time can be computed. The last step is reached by the algorithm only when all sub-paths have been explored, and hence the corresponding suspension times of the initially analyzed path $\lambda_{i,z}$ are certainly known. The WCRT of $\lambda_{i,z}$ can then be computed with Theorems 1 and 2.

B. Generating self-suspending tasks from paths

Our objective is to integrate the analysis presented in Section IV with the path analysis algorithm discussed above, which must also be properly wrapped within the iterative response-time procedure reported by Algorithm 1. These contributions were not part of [22] as the authors considered preemptive scheduling and different analysis techniques for self-suspending tasks (i.e., using mathematical programming).

As a prerequisite, it is necessary to formalize how a self-suspending task can be instantiated from a given path of a DAG task, paying careful attention to the fact that nodes are executed non-preemptively. The adopted strategy is summarized by the pseudo-code reported in Algorithm 2. The algorithm takes as input a path $\lambda_{i,z}$ that must begin and end with nodes allocated to the same processor p_k (also specified as input). It then returns a self-suspending task modelling the execution of $\lambda_{i,z}$ on p_k, and an upper bound on the total suspension time that the self-suspending task may incur.

Algorithm 2 is used in Algorithm 5 to create a self-suspending task whenever a sub-path that begins and end on the same processor is found. It then computes the WCRT of that equivalent self-suspending task using Theorems 1 and 2.

Algorithm 2 works as follows. For each node v^i_j in the path (iterated at line 5), if v^i_j is allocated on p_k then the algorithm creates a corresponding execution segment with length equal to the WCET C^i_j of the node (line 11); otherwise, it creates a suspension region with length equal to the response-time bound of the node (line 14). Furthermore, if two consecutive nodes in the path are allocated on the same processor, the algorithm inserts a suspension region with length zero (line 9): under non-preemptive scheduling this case is important as it will correspond to a scheduling event in which the execution of the path can be delayed by high-priority workload.

Finally, to bound the cumulative suspension time that the generated task can incur (i.e., an upper bound on the total suspension time $S = \sum_{j=1}^{N^S-1} S_j$), the remote interference suffered by $\lambda_{i,z}$ from each processor $\neq p_k$ is computed at lines 19–23 of Algorithm 2. In Algorithm 2, $\mathcal{P}(\lambda_{i,z})$ denotes the set of processors to which at least one node of path $\lambda_{i,z}$ is assigned.

Algorithm 2 Constructing the self-suspending task τ^{ss} corresponding to path $\lambda_{i,z}$.

```
1: global variables $RTs(v^i_j, v^i_s)$, bound on the remote interference
   suffered by sub-paths starting with $v^i_j$ and ending with $v^i_s$
2: end global variables
3: procedure DeriveSSTask($\lambda_{i,z}, \tau_i, p_k$)
4:     flag ← false
5:     for each $v^i_j \in \lambda_{i,z}$ do
6:         $p_j \leftarrow \mathcal{P}(v_{i,j})$
7:         if $p_j = p_k$ then
8:             if flag then
9:                 Add a suspension region to $\tau^{\text{ss}}$ of length 0
10:            end if
11:            Add an execution region to $\tau^{\text{ss}}$ of length $C^i_j$
12:            flag ← true
13:        else
14:            Add a suspension region to $\tau^{\text{ss}}$ of length $RTs(v^i_j, v^i_j)$
15:            flag ← false
16:        end if
17:    end for
18:    $S^{\text{UB}} \leftarrow 0$
19:    for each $p_j \in \mathcal{P}(\lambda_{i,z}) \setminus p_k$ do
20:        $v^i_s \leftarrow$ first node of $\lambda_{i,z}$ allocated to $p_j$
21:        $v^i_e \leftarrow$ last node of $\lambda_{i,z}$ allocated to $p_j$
22:        $S^{\text{UB}} \leftarrow S^{\text{UB}} + RTs(v^i_s, v^i_e)$
23:    end for
24:    return $(S^{\text{UB}}, \tau^{\text{ss}})$
25: end procedure
```

C. Integrating with the analysis of self-suspending tasks

After a self-suspending task is constructed with Algorithm 2, it remains to show how both Theorems 1 and 2 can be applied to study the response time of a path. Two major aspects must be considered: **(i)** how to account for the interference and blocking incurred by a path (and hence by the corresponding self-suspending task); and **(ii)** how to exploit the cumulative suspension bound S^{UB} offered by Algorithm 2.

A first observation for (i) is that the nodes that are either predecessors or successors of those in the path under analysis cannot interfere with the path itself. The remaining nodes constitute the (so-called) *self interference* of a DAG task [22], whose contribution can be bounded as follows.

Lemma 8: Given a path $\lambda_{i,z}$ of task τ_i that starts and ends with nodes v^i_s and v^i_e, respectively, both allocated onto the same processor p_k, let $\mathsf{self}(\lambda_{i,z})$ be the set of self-interfering nodes defined as

$$\mathsf{self}(\lambda_{i,z}) = V_i(k) \setminus \lambda_{i,z} \setminus \{prec(v^i_s) \cup succ(v^i_e)\}.$$

The self interference of $\lambda_{i,z}$ on p_k is then bounded by

$$SI(\lambda_{i,z}) = \sum_{v_j^i \in \mathsf{self}(\lambda_{i,z})} C_j^i.$$

Proof: Since tasks have constrained deadlines, only *one* instance of τ_i can be pending at a time. Hence at most one instance of the nodes of τ_i allocated to p_k (i.e., in the set $V_i(k)$) can interfere with $\lambda_{i,z}$ when executing on p_k. Furthermore, the nodes preceding the first node in $\lambda_{i,z}$ are already completed when $\lambda_{i,z}$ executes, while the successors of the last node in $\lambda_{i,z}$ cannot execute while $\lambda_{i,z}$ executes since their precedence constraints are not yet satisfied. Finally, $\lambda_{i,z}$ does not generate interference on itself. Hence, $\mathsf{self}(\lambda_{i,z})$ establishes a super-set of the nodes that can interfere with $\lambda_{i,z}$ on p_k and the lemma follows. ■

Besides self-interference, a path executing on a processor p_k can also suffer the interference generated by nodes of higher-priority tasks and be blocked by nodes of lower-priority tasks. Coping with these two phenomena does not require introducing additional terms with respect to those reported in Theorems 1 and 2. In fact, note that both the blocking and interference terms used in those two theorems do not rely on the structure of high- and low-priority tasks, but rather on their WCET, response-time bound, and period only. Consequently, when analyzing a path of a DAG task τ_i, we only need to keep track of a safe upper bounds $\overline{R}_{k,l}$ on the response time of each node of the other DAG tasks $\neq \tau_i$.

As detailed in [22], each node v_s^i of high-priority DAG-tasks can be modeled as a self-suspending task with a single execution region, whose interference is accounted by means of Equation (2). Such nodes can be subject to release jitter due to the precedence constraints in the corresponding DAGs: note that this phenomenon is already accounted in Equation (2) by means of the jitter terms $R_{i,j} - C_{i,j}$. Nodes of low-priority DAG-tasks can be modeled in the same manner, so that the corresponding blocking times can be accounted by means of Lemma 1. It is worth observing that both such ways of accounting for high-priority interference and low-priority blocking are conservative, but they can carry pessimism in the analysis as they do not exploit the information provided by the precedence constraints into the DAGs. Indeed, due to precedence constraints, there may exist some mutually-exclusive groups of nodes that can delay the path under analysis, while in our case all of them are considered to contribute to the worst-case response-time. The exploitation of this observation is left as future work.

Overall, to study the response time of a path $\lambda_{i,z}$ as a self-suspending task, the only extension required to the analysis presented in Section IV consists in coping with the self-interference $SI(\lambda_{i,z})$, which corresponds to workload with the same priority of the task under analysis. To support arbitrary priority tie-breaking, self-interference can conservatively accounted as high-priority interference, hence summing $SI(\lambda_{i,z})$ to both Equations (4) and (6). Also in this case, note that the analysis could be further improved by deriving segment-specific bounds for the self-interference to be used in Theorem 2: this improvement is not discussed here due to lack of space and is left as future work.

Finally, recalling aspect (ii) mentioned above, it is worth observing that the suspension bound S^{UB} offered by Algorithm 2 can be used to tighten the response-time of a path by simply taking the minimum between S^{UB} and the sum of the suspension times considered by Theorems 1 and 2.

D. Response-time analysis for a DAG

The analysis of each path described in the previous sections can be now integrated within the iterative scheme proposed in Algorithm 3. The main idea is to traverse the paths of the DAG tasks under analysis multiple times, iteratively refining the response-time bounds of the nodes. First, the matrix $\overline{R}_{i,j}$ containing response time upper bounds for each node $v_j^i \in V_i$ is initialized to $D_i - \sum_{v_k^i \in \mathsf{succ}(v_j^i): \mathcal{P}(v_j^i) = \mathcal{P}(v_k^i)} C_{i,k}$ (line 3). Note that it is a safe necessary condition for schedulability, as in the worst-case the successors of v_k^i running on the same processor have to be sequentially executed for their WCETs. Then, within a loop, the algorithm starts analyzing all paths of all DAG tasks into Γ (line 9). Each time a path is analyzed, the maximum response-time bound of the corresponding task is updated (line 12), as the WCRT of a DAG task is determined by the maximum response times among its paths (see Eq. (10)).

Note that the same node v_j^i can be encountered multiple times when traversing all paths of a DAG task. To update the corresponding response-time bound $\overline{R}_{i,j}$, it is then necessary to keep track of the maximum response-time $R_{i,j}$ of v_j^i experienced in all paths, which can be extracted from the global data structure RTs populated by Algorithm 5. This is performed with auxiliary variables $R_{i,j}^*$ managed at line 13.

Similarly as discussed in Section IV-D, the algorithm terminates as soon as the response-time bounds obtained at one iteration allow deeming the task set schedulable. If the task set cannot be deemed schedulable but at least one of the response-time bounds $\overline{R}_{i,j}$ can be improved (line 19), then the algorithm continues to iterate, otherwise no improvements are possible and the task set is deemed unschedulable.

VI. Task partitioning

This section presents an algorithm for partitioning parallel tasks upon the various cores of a multicore platform. The algorithm leverages the analysis presented in the previous section and is general enough to be combined with several partitioning heuristics such as First-Fit, Worst-Fit, and Best-Fit, and different orderings with which the tasks are selected.

Starting from a task set Γ under analysis, the main idea behind the algorithm consists in incrementally testing the schedulability of a task subset $\Gamma' \subseteq \Gamma$ in which each node of Γ is added one at time and assigned to processors by following a fitting heuristic.

The pseudo-code of the proposed algorithm is reported in Algorithm 4. The algorithm takes as input the task set Γ, a strategy Θ to order the tasks in Γ, and a strategy Ψ to order the M processor cores. Tasks are selected for being partitioned by following the order provided by Θ (line 3). Then, for each of such tasks τ_i, a corresponding task τ_i' is created with the same parameters of τ_i but with an empty graph (line 4). Subsequently, the nodes of τ_i are incrementally assigned to τ_i', having care of preserving the corresponding precedence constraints. Every time a node ${v_j^i}'$ is added, the algorithm tries

Algorithm 3 Schedulability test for DAG tasks with non-preemptable nodes.

```
1: procedure isSchedulableDAG($\Gamma$)
2:     $\forall \tau_i \in \Gamma, \forall v_j^i \in \tau_i$,
3:         $\overline{R}_{i,j} \leftarrow D_i - \sum_{v_k^i \in \mathsf{succ}(v_j^i): \mathcal{P}(v_j^i)=\mathcal{P}(v_k^i)} C_{i,k}$.
4:     $\forall \tau_i \in \Gamma,\ R_i \leftarrow 0$
5:     atLeastOneUpdate $\leftarrow$ TRUE
6:     while (atLeastOneUpdate=TRUE) do
7:         atLeastOneUpdate $\leftarrow$ FALSE
8:         $\forall \tau_i \in \Gamma, \forall v_j^i \in \tau_i,\ R^*_{i,j} \leftarrow 0$
9:         for all $\tau_i \in \Gamma$ do
10:            for all $\lambda_{i,z} \in \mathsf{paths}(\tau_i)$ do
11:                $RT_{i,z} \leftarrow PathAnalysis(\lambda_{i,z}, \tau_i, \Gamma,$ TRUE$)$
12:                $R_i \leftarrow \max(R_i, RT_{i,z})$
13:                $\forall v_{i,j} \in \lambda_{i,z},\ R^*_{i,j} \leftarrow \max(R^*_{i,j}, R_{i,j}\}$
14:            end for
15:        end for
16:        if $\forall \tau_i \in \Gamma,\ R_i \leq D_i$ then
17:            return TRUE
18:        end if
19:        if $\exists R^*_{i,j} : R^*_{i,j} < \overline{R}_{i,j}$ then
20:            atLeastOneUpdate $\leftarrow$ TRUE
21:        end if
22:        $\forall \tau_i \in \Gamma, \forall\ v_j^i \in \tau_i,\ \overline{R}_{i,j} \leftarrow \min\{R^*_{i,j}, \overline{R}_{i,j}\}$
23:    end while
24:    return FALSE
25: end procedure
```

Algorithm 4 Schedulability test integrated with task partitioning.

```
1: procedure isSchedulableWithPartitioning($\Gamma, \Theta, \Psi$)
2:     $\Gamma' = \emptyset$
3:     for each $\tau_i \in \Gamma$ ordered by following $\Theta$ do
4:         $\tau_i' = (\emptyset, \emptyset, T_i, D_i, \pi_i)$
5:         for each $v_j^i \in V_i$ do
6:             ${v_j^i}' = v_j^i$
7:             $V_i' = V_i' \cup {v_j^i}'$
8:             Connect nodes in $V_i'$ according to $E_i$
9:             found = FALSE
10:            for each $p_k$ ordered by following $\Psi$ do
11:                $\mathcal{P}({v_j^i}') \leftarrow p_k$
12:                if isSchedulableDAG($\Gamma' \cup \tau_i'$) then
13:                    $\mathcal{P}(v_j^i) \leftarrow p_k$
14:                    found = TRUE
15:                    break
16:                end if
17:            end for
18:            if not found then
19:                return FALSE
20:            end if
21:        end for
22:        $\Gamma' = \Gamma' \cup \tau_i$
23:    end for
24:    return TRUE
25: end procedure
```

to see whether the partial task set obtained up to the current iteration (i.e., $\Gamma' \cup \tau_i'$) is schedulable by assigning ${v_j^i}'$ to a processor p_k (Algorithm 3 is used). In this step, processors are selected according to the order provided by Ψ (line 10). If the partial task set is schedulable by assigning ${v_j^i}'$ to a particular processor p_k, then the corresponding node in the original task τ_i is assigned to the same processor (line 13). Otherwise, if no allocations for ${v_j^i}'$ are found such that the partial task set is schedulable, then the system is deemed unschedulable and the algorithm terminates (line 18). Finally, if the algorithm succeeds in allocating all the nodes, then the system is schedulable with the partitioning found during its execution.

Possible options for strategy Θ are order tasks by (i) increasing or decreasing priorities, (ii) increasing or decreasing deadlines, or (iii) increasing or decreasing utilization. Concerning strategy Ψ, effective options include the adoption of the First-fit, Worst-Fit, and Best-Fit heuristics with respect to the processors utilizations. These options are explored in the experimental results presented in Section VII.

VII. Experimental results

This section reports on an experimental study we conducted to assess the performance of the proposed schedulability test and partitioning algorithm for DAG tasks. Our test has been compared against the only, at least to the best of our records, schedulability test available for limited-preemptive parallel tasks, which has been proposed by Serrano et al. [26][4].

DAG tasks have been generated using the generator designed by the authors of [24], and originally made available online.[5] The generator presented in [24] is quite flexible and allows to tune different parameters. For this reason, it has also been adopted for the experimental evaluation of other recent papers targeting the analysis of DAG tasks including the one of Serrano et al. considered here, hence enabling a fair comparison with respect to the results of [26]. The topology of each DAG is originated from a simple fork-join task composed of two nodes connected by an edge. A recursive procedure is triggered in a probabilistic manner to expand nodes by parallel graphs. The probability for a node to fork is set to 0.8. The maximum number of nested forks is limited to two for all the presented experiments. Each fork generates a number of branches uniformly chosen in the interval $[2, n_{par}]$, where n_{par} is a generation parameter. The resulting fork-join tasks is then converted into a DAG by randomly adding edges between nodes with probability 0.2, avoiding the introduction of cycles. A very similar setting of the generator has been also used in [26]. The interested reader can refer to [24] for a more detailed description of the generator.

The computation time of each node $C_{i,j}$ is randomly generated in the range $[1, 100]$ with uniform distribution. Individual task utilizations are generated with the UUnifast algorithm [38], starting from a fixed number of tasks and a target total system utilization $U = \sum_{\tau_i \in \Gamma} C_i / T_i$, where C_i is the sum of the WCETs of the nodes of a DAG task. Minimum inter-arrival times are computed as $T_i = C_i \cdot U_i$. All the generated tasks have an implicit deadline, i.e., $D_i = T_i$.

Several partitioning heuristics have been implemented and tested. In particular, we tested the case in which the nodes of the DAG tasks are partitioned by using the Worst-Fit,

[4]Note that [26] corrects a previous work by the same authors that has been found to be flawed.

[5]The generator is no more available at the link reported in [24], but the interested reader will find it at: https://retis.sssup.it/~d.casini/resources/DAG_Generator/cptasks.zip

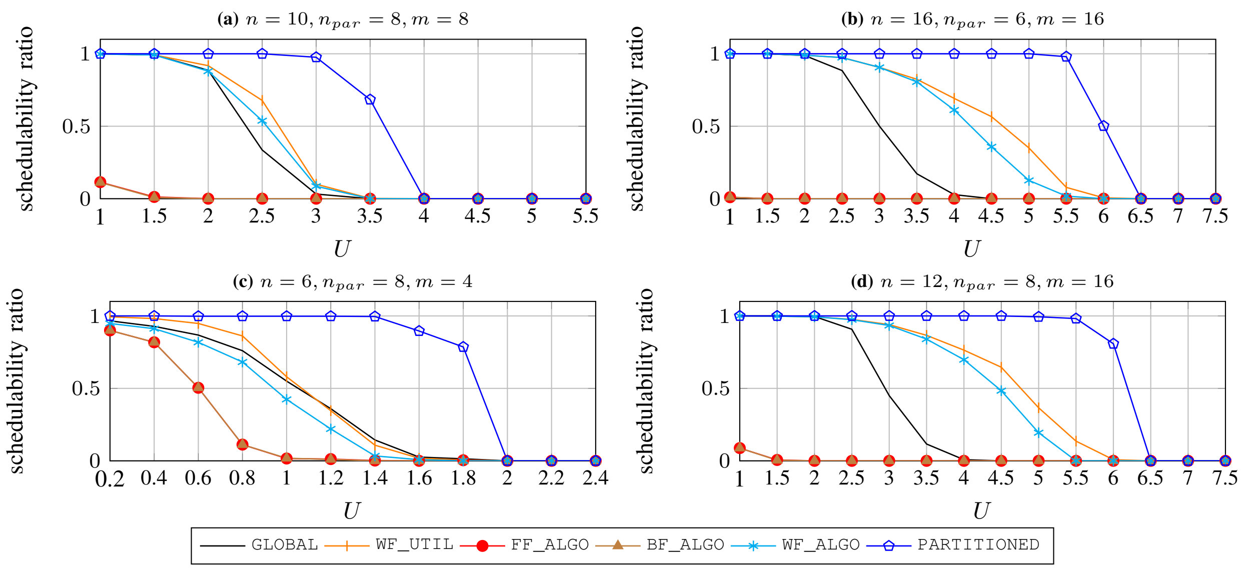

Figure 3. Schedulability ratio obtained with different scheduling approaches (global vs. partitioned) and partitioning heuristics, as function of the overall system utilization. The results are related to four representative configurations identified in the caption of each graph.

First-Fit, and Best-Fit w.r.t. to the individual utilization of the nodes. Then, the resulting allocation is tested with the analysis proposed in Section V. Since only the Worst-Fit heuristic led to reasonable schedulability performance, it is the only utilization-based partitioning scheme that is considered in this section. It is denoted as `WF_UTIL` in the following. Furthermore, we also tested the algorithm presented in Section VI. In the following, we report the results for the cases where the algorithm has been applied with tasks sorted by decreasing priorities (strategy Θ for Algorithm 4) and the First-Fit, Best-Fit, and Worst-Fit heuristics with respect to the processor utilizations (strategies Ψ for Algorithm 4). Such approaches are referred to as `FF_ALGO`, `BF_ALGO`, and `WF_ALGO`, respectively. To better explore the schedulability performance that can be achieved with a typical off-line partitioning of DAG tasks, we also consider the union of all the partitioning schemes (i.e., the logic OR of all the corresponding schedulability tests): this approach is denoted as `PARTITIONED`. Finally, the approach of [26] is referred to as `GLOBAL`.

Figure 3 reports the results of four representative configurations in which the total system utilization U has been varied. For each tested value of U, 500 task sets have been randomly generated and analyzed. The generation parameters corresponding to each configuration are reported in the captions above the graphs. Running times are discussed in Appendix B.

All plots in Figure 3 show that `WF_ALGO` performs always better than `BF_ALGO` and `FF_ALGO` (whose curves are always overlapped in the plots). Also note that `BF_ALGO` and `FF_ALGO` by themselves achieve worse performance than those achieved by `GLOBAL`. Conversely, `WF_ALGO` and `WF_UTIL` tend to perform way better, already improving on `GLOBAL` in many cases (e.g., see Fig. 3(b) and Fig. 3(d)) reaching a performance gap up to 70%. Finally, the obtained results show that the combination of all these partitioning approaches is winning. Indeed, as it can be noted from all the graphs, `PARTITIONED` achieves huge performance improvements over `GLOBAL` in all the tested scenarios, reaching up to 100% improvements in some of the tested configurations (e.g., see Fig. 3(b) for $U = 5$). From the whole set of experiments we conducted, we observed that such improvements tend to increase as the number of processors increases.

VIII. Conclusion and Future Work

This paper presented a schedulability analysis for DAG tasks scheduled under partitioned fixed-priority scheduling where each node is non-preemptively executed. The proposed analysis has been built upon a fine-grained analysis for segmented self-suspending tasks, also proposed in this work, which analytically dominates the state-of-the-art analyses. A partitioning algorithm has been also presented. The algorithm is general enough for being integrated with several task orderings and fitting heuristics. Experimental results have finally been presented comparing our partitioned approaches against an analysis for DAG tasks under global scheduling without preemptions. The results showed very large improvements with performance gaps up to 100% under several configurations.

Future work should target a deeper investigation of partitioning strategies for parallel tasks and possibly integrate memory contention and data communication delays in the analysis. Furthermore, the authors believe that it is worth studying whether the analysis precision can be improved by carefully refining some of the approaches presented in this work.

Appendix A

For the sake of completeness, the algorithm to analyze the paths of a DAG task presented in [22], and summarized in Section V-A, is reported in Algorithm 5. The algorithm takes as input the path $\lambda_{i,k}$, the related DAG task τ_i, and the task set to which it belongs. Algorithm 5 is a recursive function. Therefore, the parameter `isRoot` is a boolean variable used to distinguish recursive calls from the

Algorithm 5 Computes the WCRT of a path $\lambda_{i,k}$ of τ_i

```
1: global variables $RTs(v^i_j, v^i_s)$, bound on the remote interference
   suffered by sub-paths starting with $v^i_j$ and ending with $v^i_s$
2: end global variables
3: procedure PathAnalysis($\lambda_{i,k}, \tau_i, \Gamma$, isRoot)
4:     $v^i_{first} \leftarrow$ first node in $\lambda_{i,k}$
5:     $v^i_{last} \leftarrow$ last node in $\lambda_{i,k}$
6:     $p_{ss} \leftarrow \mathcal{P}(v_{i,first})$
7:     if $|\mathcal{P}(\lambda_{i,k})|$ = 1 then                          ▷ Base-case
8:         $(S^{UB}, \tau^{SS}) \leftarrow$ deriveSSTask($\lambda_{i,k}, \tau_i, p_{ss}$)
9:         $R(\lambda_{i,k}) \leftarrow$ WCRT($\tau^{SS}, \lambda_{i,k}, \tau_i, \Gamma, S^{UB}, p_{ss}$)
10:        $RTs(v^i_{first}, v^i_{last}) \leftarrow R(\lambda_{i,k})$
11:    else
12:        if $\mathcal{P}(v^i_{first}) = \mathcal{P}(v^i_{last})$ then              ▷ Valid SS task
13:            $\lambda' \leftarrow \lambda_{i,k} \setminus \{v^i_{first}, v^i_{last}\}$
14:            PathAnalysis($\lambda', \tau_i, \Gamma, false$)
15:            $(S^{UB}, \tau^{SS}) \leftarrow$ deriveSSTask($\lambda_{i,k}, \tau_i, p_{ss}$)
16:            $R(\lambda_{i,k}) \leftarrow$ WCRT($\tau^{SS}, \lambda_{i,k}, \tau_i, \Gamma, S^{UB}, p_{ss}$)
17:            $RTs(v^i_{first}, v^i_{last}) \leftarrow R(\lambda_{i,k}) - S^{UB}$
18:        else                                ▷ Analyze sub-paths containing SS tasks
19:            $p_{last} \leftarrow \mathcal{P}(v^i_{last})$
20:            $v^i_j \leftarrow$ first node $v^i_j \in \lambda_{i,k} : \mathcal{P}(v^i_j) = p_{last}$
21:            $\lambda' \leftarrow$ nodes of $\lambda_{i,k}$ from $v^i_j$ to $v^i_{last}$
22:            PathAnalysis($\lambda', \tau_i, \Gamma, false$)
23:            $\lambda'' \leftarrow \lambda_{i,k} \setminus \{v^i_{last}\}$
24:            PathAnalysis($\lambda'', \tau_i, \Gamma, false$)
25:        end if
26:    end if
27:    if isRoot = TRUE then
28:        $RT \leftarrow 0$
29:        for each $p_j \in \mathcal{P}(\lambda_{i,k})$ do
30:            $v^i_{pfirst} \leftarrow$ first node allocated to $p_j \in \lambda_{i,k}$
31:            $v^i_{plast} \leftarrow$ last node allocated to $p_j \in \lambda_{i,k}$
32:            $RT \leftarrow RT + RTs(v^i_{pfirst}, v^i_{plast})$
33:        end for
34:    end if
35: end procedure
```

first call to PathAnalysis. Starting from a path $\lambda_{i,k}$, a bound for each of its suspension regions is derived by recursively calling PathAnalysis on smaller paths (lines 14, 22, and 24). It is worth noting that the recursive call at line 14 ensures that the length of the suspension regions are known at lines 15 and 16 to convert the sub-path $\lambda_{i,k}$ into an equivalent self-suspending task and compute its WCRT, respectively. All WCRT values computed at deeper levels of the recursive calls of Algorithm 5 are stored into the global data structure $RTs(v^i_j, v^i_s)$, which stores the cumulative response time of all execution regions of a self suspending task that starts at node v^i_j and terminates at node v^i_s. The reason to subtract S^{UB} from $R(\lambda_{i,k})$ is explained in [22]. DeriveSSTask is given in Algorithm 2 and explained in Section V-B. It converts each node $v^i_j \in \lambda_{i,k}$ allocated to processor p_{ss} into an equivalent execution region of a self-suspending task, while all nodes executing on other processors are treated as suspension regions. The duration of each suspension region is given by the WCRT of the corresponding node in $\lambda_{i,k}$ and is therefore obtained from the global data structure RTs. The WCRT function uses Theorems 1 and 2 to compute the WCRT of the equivalent self-suspending task τ^{ss}, where the nodes of the other DAG tasks are modeled as detailed in Section V-C. Finally, lines 28-32 are used to derive an upper-bound on the overall response time of the path, which is given by the sum of the cumulative response time bounds on the nodes of the inner self-suspending tasks allocated on each processor $p_j \in \mathcal{P}(\lambda_{i,k})$.

Appendix B
Running times

The analysis and partitioning algorithms proposed in this work have been implemented in C++ and compiled with the Microsoft VC++2015 compiler (with default compiling options in release mode). The experiments have been executed on an Intel i7 machine running at 4 GHz with 32 GB of RAM. Literal and sequential implementations have been realized, i.e., not optimized. In the tested configurations discussed in Section VII we measured the running time of the WF_ALGO approach, hence accounting for the execution of the partitioning algorithm (Algorithm 4) and the analysis of DAG tasks (Algorithm 3 called multiple times). We observed a maximum running time below 10 seconds, and an average running time below 2 seconds. These measurements show that our approach is compatible with the typical time-frame of off-line design activities.

Acknowledgment

The authors would like to thank José Fonseca for providing the code of [22]. This work has been partially supported by the RETINA Eurostars Project E10171 and by National Funds through FCT (Portuguese Foundation for Science and Technology) within the CISTER Research Unit (CEC/04234).

References

[1] A. Bastoni, B. B. Brandenburg, and J. H. Anderson, “An empirical comparison of global, partitioned, and clustered multiprocessor EDF schedulers,” in *31st IEEE Real-Time Systems Symposium (RTSS)*, Nov. 2010.

[2] A. Biondi and Y. Sun, “On the ineffectiveness of 1/m-based interference bounds in the analysis of global EDF and FIFO scheduling,” *Real-Time Systems*, vol. 54, no. 3, pp. 515 – 536, Mar 2018.

[3] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, “Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms,” in *19th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2013, pp. 55–64.

[4] A. Biondi and M. D. Natale, “Achieving predictable multicore execution of automotive applications using the LET paradigm,” in *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2018.

[5] R. Mancuso, R. Pellizzoni, M. Caccamo, L. Sha, and H. Yun, “WCET(m) estimation in multi-core systems using Single Core Equivalence,” in *27th Euromicro Conference on Real-Time Systems (ECRTS)*, 2015.

[6] M. Nasri and G. Fohler, “Non-work-conserving non-preemptive scheduling: Motivations, challenges, and potential solutions,” in *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, July 2016.

[7] M. Nasri, G. Nelissen, and B. B. Brandenburg, “A response-time analysis for non-preemptive job sets under global scheduling,” in *Proceedings of the 30th Euromicro Conference on Real-Time Systems (ECRTS 2018)*, July 2018.

[8] R. Tabish, R. Mancuso, S. Wasly, A. Alhammad, S. S. Phatak, R. Pellizzoni, and M. Caccamo, “A real-time scratchpad-centric OS for multi-core embedded systems,” in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2016.

[9] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, “Rethinking the inception architecture for computer vision,” in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[10] A. Saifullah, K. Agrawal, C. Lu, and C. Gill, “Multi-core real-time scheduling for generalized parallel task models,” in *2011 IEEE 32nd Real-Time Systems Symposium*, 2011.

[11] G. Nelissen, J. Fonseca, G. Raravi, and V. Nelis, “Timing analysis of fixed priority self-suspending sporadic tasks,” in *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)*, July 2015.

[12] Z. Dong, C. Liu, S. B. K.-H. Chen, J.-J. Chen, G. von der Bruggen, and J. Shi, “Shared-resource-centric limited preemptive scheduling: A comprehensive study of suspension-based partitioning approaches,” in *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2018.

[13] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley, “A predictable execution model for COTS-based embedded systems,” in *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, April 2011.

[14] S. Altmeyer, R. I. Davis, and C. Maiza, “Improved cache related preemption delay aware response time analysis for fixed priority preemptive systems,” *Real-Time Systems*, vol. 48, no. 5, pp. 499–526, 2012.

[15] S. A. Rashid, G. Nelissen, D. Hardy, B. Akesson, I. Puaut, and E. Tovar, “Cache-persistence-aware response-time analysis for fixed-priority preemptive systems,” in *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, July 2016.

[16] S. A. Rashid, G. Nelissen, S. Altmeyer, R. I. Davis, and E. Tovar, “Integrated analysis of cache related preemption delays and cache persistence reload overheads,” in *2017 IEEE Real-Time Systems Symposium (RTSS)*, Dec 2017.

[17] K. Lakshmanan, S. Kato, and R. Rajkumar, “Scheduling parallel real-time tasks on multi-core processors,” in *2010 31st IEEE Real-Time Systems Symposium*, Nov 2010.

[18] G. Nelissen, V. Berten, J. Goossens, and D. Milojevic, “Techniques optimizing the number of processors to schedule multi-threaded tasks,” in *2012 24th Euromicro Conference on Real-Time Systems*, July 2012.

[19] V. Bonifaci, A. Marchetti-Spaccamela, S. Stiller, and A. Wiese, “Feasibility analysis in the sporadic DAG task model,” in *2013 25th Euromicro Conference on Real-Time Systems*, July 2013.

[20] S. Baruah, “Improved multiprocessor global schedulability analysis of sporadic DAG task systems,” in *2014 26th Euromicro Conference on Real-Time Systems*, July 2014.

[21] J. Fonseca, G. Nelissen, and V. Nélis, “Improved response time analysis of sporadic DAG tasks for global FP scheduling,” in *Proceedings of the 25th International Conference on Real-Time Networks and Systems*, ser. RTNS ’17, 2017.

[22] J. Fonseca, G. Nelissen, V. Nelis, and L. M. Pinho, “Response time analysis of sporadic DAG tasks under partitioned scheduling,” in *2016 11th IEEE Symposium on Industrial Embedded Systems (SIES)*, May 2016.

[23] J. C. Fonseca, V. Nélis, G. Raravi, and L. M. Pinho, “A multi-DAG model for real-time parallel applications with conditional execution,” in *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, ser. SAC ’15, 2015.

[24] A. Melani, M. Bertogna, V. Bonifaci, A. Marchetti-Spaccamela, and G. C. Buttazzo, “Response-time analysis of conditional DAG tasks in multiprocessor systems,” in *2015 27th Euromicro Conference on Real-Time Systems*, July 2015.

[25] S. Baruah, V. Bonifaci, and A. Marchetti-Spaccamela, “The global EDF scheduling of systems of conditional sporadic DAG tasks,” in *2015 27th Euromicro Conference on Real-Time Systems*, July 2015.

[26] M. A. Serrano, A. Melani, S. Kehr, M. Bertogna, and E. Quiñones, “An analysis of lazy and eager limited preemption approaches under DAG-based global fixed priority scheduling,” in *2017 IEEE 20th International Symposium on Real-Time Distributed Computing (ISORC)*, May 2017.

[27] R. I. Davis, A. Burns, R. J. Bril, and J. J. Lukkien, “Controller area network (can) schedulability analysis: Refuted, revisited and revised,” *Real-Time Systems*, vol. 35, no. 3, pp. 239–272, Apr 2007.

[28] R. J. Bril, J. J. Lukkien, and W. F. Verhaegh, “Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption,” *Real-Time Systems*, vol. 42, no. 1-3, pp. 63–119, 2009.

[29] G. Yao, G. Buttazzo, and M. Bertogna, “Feasibility analysis under fixed priority scheduling with fixed preemption points,” in *2010 IEEE 16th International Conference on Embedded and Real-Time Computing Systems and Applications*, Aug 2010.

[30] M. Nasri and B. B. Brandenburg, “Offline equivalence: A non-preemptive scheduling technique for resource-constrained embedded real-time systems (outstanding paper),” in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2017 IEEE*, 2017.

[31] G. C. Buttazzo, M. Bertogna, and G. Yao, “Limited preemptive scheduling for real-time systems. a survey,” *IEEE Transactions on Industrial Informatics*, Feb 2013.

[32] G. Fohler and K. Ramamritham, “Static scheduling of pipelined periodic tasks in distributed real-time systems,” in *Proceedings 9th Euromicro Workshop on Real Time Systems*, Jun 1997.

[33] K. Tindell and J. Clark, “Holistic schedulability analysis for distributed hard real-time systems,” *Microprocess. Microprogram.*, vol. 40, no. 2-3, p. 117–134, 2012.

[34] J. C. Palencia and M. G. Harbour, “Schedulability analysis for tasks with static and dynamic offsets,” in *Proceedings 19th IEEE Real-Time Systems Symposium*, 1998.

[35] ——, “Exploiting precedence relations in the schedulability analysis of distributed real-time systems,” in *Proceedings 20th IEEE Real-Time Systems Symposium*, 1999.

[36] P. Jayachandran and T. Abdelzaher, “Transforming distributed acyclic systems into equivalent uniprocessors under preemptive and non-preemptive scheduling,” in *2008 Euromicro Conference on Real-Time Systems*, July 2008.

[37] J.-J. Chen, G. Nelissen, W.-H. Huang, M. Yang, B. Brandenburg, K. Bletsas, C. Liu, P. Richard, F. Ridouard, Neil, Audsley, R. Rajkumar, and D. de Niz, “Many suspensions, many problems: A review of self-suspending tasks in real-time systems,” Faculty of Informatik, TU Dortmund, Tech. Rep. 854, 2016.

[38] E. Bini and G. Buttazzo, “Measuring the performance of schedulability tests,” *Real-Time Systems*, vol. 30, no. 1, pp. 129 – 154, May 2005.