Event-Driven Delay-Induced Tasks: Model, Analysis, and Applications

Federico Aromolo*, Alessandro Biondi*, Geoffrey Nelissen†, and Giorgio Buttazzo*
*Scuola Superiore Sant'Anna, Pisa, Italy
†Eindhoven University of Technology, Eindhoven, The Netherlands

***Abstract*—Parallel execution and hardware acceleration involving specialized devices such as GPUs and FPGAs are becoming increasingly relevant in the domain of embedded systems. Communication between jobs dispatched on different cores and hardware accelerators is most often implemented using asynchronous events. Modeling the timing behavior of such systems requires to account for the delays incurred by each task due to the additional time spent waiting for events. This paper presents the *event-driven delay-induced* (EDD) task model to explicitly deal with complex computing workloads that incur such kinds of delays. The EDD task model generalizes several state-of-the-art models, such as the DAG task model and the segmented self-suspending task model, and is particularly suited to analyze parallel tasks that issue asynchronous hardware acceleration requests. Two analysis techniques for EDD tasks executing on single core platforms are first provided. We then extend those approaches to analyze parallel real-time tasks under partitioned multicore scheduling by means of a model transformation. Experimental results are presented to compare the two analysis techniques for EDD tasks proposed in the paper. Finally, we compare the analysis of partitioned parallel tasks modeled with EDD tasks against federated scheduling.**

I. Introduction

Heterogeneous computing platforms, which integrate asymmetric multicore processors together with hardware accelerators, are establishing as the most appropriate solution to implement power-efficient, yet high-performance embedded computing systems. These systems are characterized by complex software workloads that include parallel computations on multiprocessors and hardware acceleration requests of different forms. In such devices, it is often the case that, when dispatching computations over a set of computing resources, some task has to *wait* for a set of *events* to occur before proceeding with its execution, hence introducing delays in the system. In addition, when multiple activities coexist in the system, parts of the workload may be in execution while others are waiting, and their completion may itself constitute an event that is being waited for by some of the other activities.

This kind of behavior is typical whenever execution involves asynchronous hardware acceleration, as in the case of general-purpose graphical processing unit (GPGPU) computing on Nvidia CUDA platforms employing the related CUDA API. Another example consists in heterogeneous platforms integrating field-programmable gate arrays (FPGAs) for asynchronous hardware acceleration of specialized activities. Indeed, in such contexts, a task running on a processor can issue an asynchronous acceleration request, then execute some subtask A on the processor, and finally wait for the accelerated computation to be completed before proceeding by executing another subtask B. Note that the accelerated computation may be completed either before or after the completion of subtask A on the processor. Hence, before executing subtask B, two events must have occurred: (i) the completion of subtask A and (ii) the completion of the accelerated computation.

If multiple accelerators are used or multiple acceleration requests can be pending at the same time, more complex execution behaviors can emerge, with several events to be simultaneously waited for. Similar situations occur in the execution of parallel real-time tasks, in which, due to precedence constraints, some subtask has to wait for the completion of other subtasks, running on either the same or in a different processor, before becoming eligible for execution.

The widespread presence of computational activities that wait for timing events in the domain of embedded systems calls for modeling and analysis techniques that are *explicitly* conceived to deal with *event-driven delays*. Unfortunately, existing models can either deal with these delays at the cost of considerable pessimism in the real-time analysis, or can only accommodate specific categories of workloads.

Contribution and paper organization. This paper presents the *event-driven delay-induced* (EDD) task model to explicitly deal with complex computing workloads that incur delays due to waiting for events (Sec. II). In this model, events can occur after a minimum and a maximum time relative to the completion of some subtask or the release of the task itself. A graph-based structure is used to represent the behavior of the tasks. The EDD task model has relevant practical applications (Sec. III) and also formally generalizes a number of other task models proposed in previous work, including segmented self-suspending tasks and parallel DAG tasks. A closed-form (Sec. IV) and an optimization-based (Sec. V) analysis technique for EDD tasks are presented. Given the relevance of parallel workloads, the paper also shows how to analyze parallel real-time tasks under partitioned scheduling by means of model transformations to an equivalent set of EDD tasks (Sec. VI). Experiments (Sec. VII) finally compare the proposed analysis techniques for EDD tasks and assess the schedulability performance of partitioned scheduling for parallel tasks when analyzed by means of EDD tasks, showing considerable improvements over the case of federated scheduling [1] (selected for comparison for being the closest popular alternative in terms of approach, runtime overhead, and implied predictability).

II. System model

This work considers task sets $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$ of n *event-driven delay-induced* (EDD) tasks to be scheduled on a processor of a multicore platform under preemptive fixed-priority scheduling. Each EDD task τ_i releases a potentially infinite sequence of jobs $J_{i,1}, \ldots, J_{i,j}, \ldots$. For the sake of conciseness, in the following, we use J_i when referring to an arbitrary job of τ_i. Each EDD task τ_i is characterized by a tuple (G_i, T_i, D_i, π_i), where G_i is a *directed acyclic graph* (DAG) representing the computational structure of the task, T_i is the minimum inter-arrival time (or period) between the jobs of τ_i, $D_i \leq T_i$ is the relative deadline to be respected by all jobs of the task, and π_i is the fixed scheduling priority associated to the jobs of τ_i. The DAG $G_i = (V_i, E_i)$ is composed of a set $V_i = \{v_i^1, v_i^2, \ldots, v_i^{n_i}\}$ of n_i nodes (or vertices) and a set E_i of directed edges connecting any two nodes in V_i. Each node $v_i^j \in V_i$ represents a computational unit of task τ_i, referred to as *subtask* (or execution segment). Each subtask $v_i^j \in V_i$ is characterized by a worst-case execution time (WCET) C_i^j.

Let $e_i^{a,b}$ represent a directed edge connecting v_i^a to v_i^b. Each edge $e_i^{a,b} \in E_i$ implies a precedence constraint between subtasks v_i^a and v_i^b, and is labeled with a pair $(\underline{W}_i^{a,b}, W_i^{a,b})$. A precedence constraint between subtasks v_i^a and v_i^b is satisfied when v_i^a has completed its execution and an additional time duration has elapsed since the completion of v_i^a. The length of the additional time duration can be any value within $[\underline{W}_i^{a,b}, W_i^{a,b}]$, and models the waiting time between the completion of v_i^a and the *notification of an event* to v_i^b.

All subtasks of any job J_i of τ_i are simultaneously released as soon as the job is released, and a subtask becomes *ready* for execution as soon as all the precedence constraints corresponding to the incoming edges on the subtask are satisfied. A task τ_i is *suspended* at time t if τ_i released a job J_i at or before t, the execution of J_i is not completed at time t, and J_i does not have any ready subtasks at time t. A task τ_i is said to be *pending* at time t if τ_i released a job J_i at or before t and the execution of J_i is not completed at time t. Analogously, within a job J_i of task τ_i released at or before a given time t, subtask v_i^a is said to be pending at time t if the instance of v_i^a in J_i is not completed at time t. Subtasks inherit the priority of the corresponding task. In case two subtasks share equal priority, the subtask that first became ready for execution has higher priority, as in first-in-first-out (FIFO) ordering.

The *response time* R_i of a job J_i of task τ_i is defined as the difference between its finishing time, that is, the time at which the job completes its execution, denoted by f_i, and its release time, denoted by a_i. The *worst-case response time* (WCRT) of a task τ_i is defined as the maximum response time that any job of τ_i can experience. Analogously, the response time R_i^b of a subtask v_i^b instanced within a job J_i of task τ_i is defined as the difference between its finishing time, denoted by f_i^b, and the release time a_i of the job J_i.

A. Additional terminology and assumptions

Whenever an edge exists between v_i^a and v_i^b, directed towards v_i^b, v_i^a is said to be a direct (or immediate) predecessor of v_i^b, whereas v_i^b is said to be a direct (or immediate) successor of v_i^a. The set of direct predecessors of v_i^a is denoted by $\text{ipred}(v_i^a)$, while the set of direct successors of v_i^a is denoted by $\text{isucc}(v_i^a)$. A node with no incoming edge is referred to as a *source* node, while a node with no outgoing edge is referred to as a *sink* node. The set of sink nodes in a DAG G_i is denoted by $\text{sink}(G_i)$.

A *path* is defined as an ordered sequence of subtasks where a directed edge exists between any two adjacent subtasks in the sequence, each subtask in the sequence is an immediate predecessor of the following subtask, and the sequence starts from a source node and ends on a sink node. Given a path λ, $V(\lambda)$ represents the set of nodes belonging to the path, while $E(\lambda)$ represents the set of edges traversed by the path. The set of all paths in a DAG G_i is denoted by $\text{path}(G_i)$.

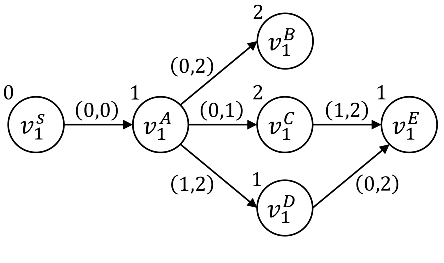

Figure 1. DAG G_1 of example task τ_1.

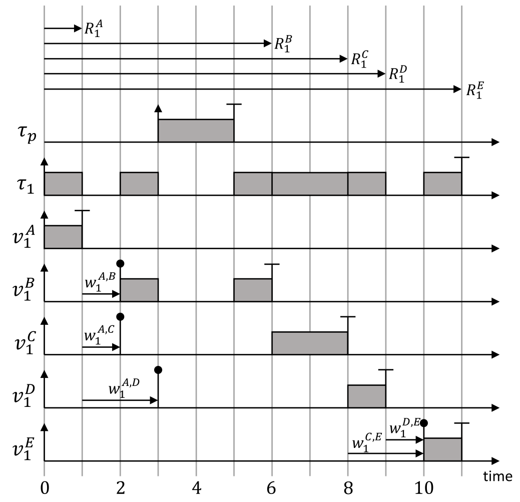

Figure 2. Example schedule of task τ_1 with interference from task τ_p.

Given this definition, a subtask v_i^a is said to be a predecessor of v_i^b whenever there exists a path in the graph G_i that includes both v_i^a and v_i^b, with v_i^a appearing before v_i^b in the sequence. In the same situation, v_i^b is said to be a successor of v_i^a. Note that the set of successors also results from the transitive application of the definition of immediate successors; and similarly for the set of predecessors. In addition, two subtasks are said to be independent if none is a predecessor or successor of the other, either directly or transitively. The model assumes that each task τ_i includes a single source node, denoted by v_i^s, with $C_i^s = 0$, which corresponds to the release of a job of τ_i. Hence, any edge $e_i^{s,a}$ connecting the source node v_i^s to another node v_i^a implies that v_i^a can become ready after a variable delay in $[\underline{W}_i^{s,a}, W_i^{s,a}]$ relative to the task release. Note that the introduction of this assumption is without loss of generality, as any subtask v_i^a that must become ready as soon as the task is released can be modeled with an edge connecting v_i^s to v_i^a labeled with the pair $(0, 0)$.

Figure 1 depicts the structure of an example EDD task τ_1.

In the figure, the value on each node corresponds to the WCET of the corresponding subtask, while the values on the edges represent the minimum and maximum waiting times between the two subtasks connected by the edge. Since v_i^s is the only source node in the task, the corresponding subtask has no incoming precedence constraints and, therefore, becomes ready for execution as soon as the job J_i is released.

B. Example schedule

An example schedule of task τ_1 illustrated in Figure 1 with interference from a classical sequential higher-priority task τ_p with WCET $C_p = 2$ is provided in Figure 2. The figure highlights the time intervals corresponding to the values of the response times R_1^a for all $v_1^a \in V_1$ and the actual waiting times $w_1^{a,b} \in [\underline{W}_1^{a,b}, W_1^{a,b}]$ experienced in the example schedule for each $e_1^{a,b} \in E_1$. Note that subtask v_1^s is omitted from the figure, as it corresponds to the release event of τ_1 (i.e., it is released at time 0 and immediately terminates its execution). In the schedule, task τ_1 is *suspended* within the time intervals $(1, 2]$ and $(9, 10]$, as none of the subtasks of τ_1 is ready for execution at those intervals.

III. Applications

The EDD task model allows explicitly modeling the delays that result from the time spent waiting for events that occur after a bounded time from the task release or the completion of a subtask. This peculiarity makes this model more expressive than other models available in the literature and suitable to analyze a set of complex execution behaviors (discussed in the following) with a fine level of detail.

It is important not to confuse the EDD task model with other graph-based models proposed in previous work. For instance, it largely differs from the digraph real-time (DRT) task model [2] due to a number of reasons. Indeed, differently from EDD tasks, **(i)** DRT tasks are described by a directed graph G, which is typically cyclic, whose nodes denote possible job instances for the task, and whose paths denote possible job sequences; **(ii)** DRT tasks have edges in G that model mutually-exclusive evolutions of job sequences; **(iii)** they have edges in G labeled with the minimum inter-arrival time between the corresponding jobs (i.e., they are not released with a common period); and **(iv)** they specify an individual deadline for each job instance. EDD tasks are also different from parallel DAG tasks, whose precedence constraints are not characterized by a variable delay with which they can be satisfied. They are also more generic than self-suspending tasks since self-suspending tasks are not characterized by a DAG but only by a single sequence of execution segments.

Relevant applications of the EDD task model are discussed next.

Modeling asynchronous hardware acceleration. A common programming scheme when dealing with hardware accelerators consists in asynchronously offloading a heavy computation to an accelerator, then continuing executing on a processor, and eventually waiting by suspending the execution on the processor as long as the accelerated task is not completed. To name a relevant example, this is the case of the CUDA Runtime API, where the user can launch a number of operations (such as memory copies and kernel executions) to be served by a set of GPUs in an asynchronous fashion, and then wait for their completion with the synchronization functions offered by the API, such as `cudaDeviceSynchronize()` and `cudaStreamSynchronize()`, which implement the waiting by suspending the calling process. A task that is running on a processor and makes use of this API can make progress in its execution from the time an operation is offloaded to the GPUs to the time in which a synchronization function is called. A minimal example code that uses asynchronous execution with the CUDA Runtime API is reported in Listing 1.

```
TASK(example)
{
  << ... >>

  // Asynchronously launch the kernel on GPU
  my_gpu_kernel <<<blockCount, threadCount>>>();

  // Continue to execute on the CPU
  my_cpu_intensive_func();

  // Wait for the completion of the GPU kernel
  cudaDeviceSynchronize();

  << ... >>
}
```

Listing 1. Example task that uses asynchronous GPU acceleration.

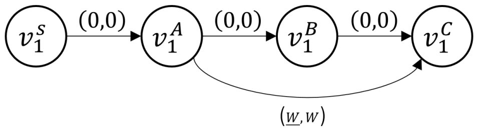

Figure 3. EDD task to model the example task of Listing 1.

In this example, a task first executes some code on its processor, then offloads a computation to a GPU, subsequently executes some other code on the processor, and finally waits for the completion of the computation offloaded to the GPU before proceeding with the completion of its execution on the processor. This example code can be modeled with the EDD task illustrated in Figure 3, where $\underline{W}$ and W denote the minimum and the maximum time the GPU can take to execute the offloaded computation, respectively. Clearly, in the presence of tasks that launch multiple GPU operations and/or offload computations to multiple accelerators (such as the case of multi-GPU systems or hybrid platforms with different kinds of accelerators), more complex execution behaviors can occur, highlighting the need for a DAG-based model as offered by EDD tasks. For instance, branches in the DAG may allow modeling the case in which a callback is executed when an accelerated computation completes.

Note that a similar programming scheme also applies to the case of I/O devices with direct memory access (DMA), where the programmer can asynchronously instruct memory copies to/from the device and later wait for their completion after executing some other code.

Modeling partitioned parallel tasks. EDD tasks are also particularly suited to model partitioned parallel real-time tasks. Given the high relevance of parallel tasks, this case is formally addressed in Section VI, while in the following it is presented by means of an example. Consider Figure 4(a), which depicts the structure of a sample sporadic parallel DAG task τ^P in terms of its DAG G^{P}. Task τ^{P} is executed under partitioned fixed-priority scheduling on a two-processor platform, i.e., each subtask (corresponding to a node in G^P) is statically allocated to one of the two processors. In the figure, the white nodes represent subtasks assigned to processor P_1, while the grey nodes represent subtasks assigned to processor P_2.

For the purpose of real-time analysis, the parallel task τ^{P} can be modeled as a set of two synchronously-released EDD tasks, one for each processor. The first EDD task, τ_1, is provided in Figure 4(b), and is related to the subtasks assigned to processor P_1, while Figure 4(c) provides the second EDD task, τ_2, related to the subtasks assigned to processor P_2. In the parallel DAG task model, each edge denotes a precedence constraint that is satisfied when the node from which the edge starts has completed its execution. Consequently, every two subtasks in τ^{P} connected by an edge and allocated to the same processor can be modeled in an EDD task by the same subtasks connected by an edge labeled with $(0, 0)$. Conversely, each edge connecting two subtasks in τ^{P} that are allocated to different processors is modeled by an edge in the EDD task corresponding to the processor to which the destination node is allocated. For example, the edge connecting v_1^C (allocated to P_2) and v_1^E (allocated to P_1) in τ^{P} can be modeled by an edge in τ_1 that connects the source node of τ_1 to v_1^E. This edge in τ_1 can then be labeled by $(0, R_1^C)$, where R_1^C is the worst-case response time of v_1^C. This transformation is safe because the corresponding precedence constraint in τ_1 has to be satisfied when v_1^C completes, which, by definition of response time, occurs no later than R_1^C time units from the release of τ_1, because the two EDD tasks are synchronously released. In the general case in which a subtask in τ^{P} has multiple incoming edges of this kind, a single edge labeled with $(0, R^*)$ can be placed in the EDD task, where R^* is the maximum among the worst-case response times of the predecessors allocated to remote processors, e.g., as it is the case for v_1^H in Figure 4(c).

Generalization of other task models. It is worth observing that the EDD task model generalizes other task models such as sequential tasks with release jitter [3], transactional tasks [4], and segmented self-suspending tasks [5].

A sequential task τ^{J} with WCET C and release jitter J behaves as a subtask in the EDD task model that can become ready for execution with a variable delay in $[0, J]$, relative to the release of τ^{J}. Hence, it can be modeled by an EDD task τ_i with the same period and deadline as τ^{J}, and a DAG with two nodes v_i^S and v_i^1, where v_i^S is the source node and v_i^1 is a subtask with WCET C. The source node is then connected to v_i^1 by an edge labeled by $(0, J)$.

A transactional task τ^{T} comprises a set of subtasks to be sequentially executed, where each subtask $\tau_{i,j}^T$ is characterized by a WCET $C_{i,j}^T$, an offset $\Phi_{i,j}^T$, and a jitter $J_{i,j}^T$. The behavior of a task τ^{T} can be modeled by means of an EDD task τ_i with the same period as τ^{T} by defining, for each subtask $\tau_{i,j}^T$ of τ^{T}, (i) a node v_i^j with WCET equal to $C_{i,j}^T$, and (ii) an edge connecting v_i^S to v_i^j, labeled with $(\Phi_{i,j}^T, \Phi_{i,j}^T + J_{i,j}^T)$.

Segmented self-suspending (SS) tasks alternate k execution regions to $k-1$ self-suspension regions. Formally, they are defined by a sequence $(C^1, S^1, C^2, S^2, \ldots, C^k)$, where C^j denotes the WCET of the j-th execution region and S^j denotes the duration of the j-th self-suspension region. Such tasks can be modeled with an EDD task τ_i by defining (i) for each execution region with index j one subtask in τ_i with WCET C^j; (ii) for each pair of execution regions with index j and $j+1$, respectively, with $j < k$, an edge in τ_i labeled with $(0, S^j)$ connecting the two corresponding subtasks; and (iii) one edge labeled with $(0, 0)$ connecting the source node of τ_i to the subtask corresponding to the first execution region, as the first execution region of the SS task becomes ready as soon as the task is released.

IV. Preliminaries and baseline analysis

This section presents preliminary results and a baseline analysis for computing an upper bound on the worst-case response time (WCRT) of each task in τ. In the following, let τ_i be the EDD task under analysis, i.e., the one for which a bound on the WCRT is to be derived.

A. Upper bounds on the worst-case response times of each subtask

An upper bound on the WCRT of a set of EDD tasks can be obtained by converting each EDD task into a *dynamic self-suspending* (DSS) task. To make the paper self-contained, the DSS task model and its analysis are reviewed next.

1) Dynamic self-suspending task model and analysis: In the dynamic self-suspending task model, each task τ_i is characterized by a tuple $(C_i, S_i, D_i, T_i, \pi_i)$, where C_i corresponds to the cumulative worst-case execution time of τ_i, S_i is the maximum cumulative self-suspension time for task τ_i, D_i represents the relative deadline of each job of τ_i, T_i represents the minimum inter-arrival time of jobs of τ_i, and π_i is the fixed priority assigned to the task. The utilization of a self-suspending task τ_i is defined as $U_i = C_i/T_i$. In the following, assume that $D_i \leq T_i$ holds for each self-suspending task τ_i (constrained deadlines), and that tasks are scheduled on a single-processor platform according to a fixed-priority scheduling policy, where each task is assigned a unique priority level such that task τ_i has a higher priority than task τ_j if $i < j$.

Under this model, assuming that $R_i \leq D_i$ holds for each τ_i such that $1 \leq i \leq k-1$, an upper bound on the worst-case response time R_k of a self-suspending task τ_k is given by the minimum positive value for t such that

$$C_k + S_k + \sum_{i=1}^{k-1} \left\lceil \frac{t + \sum_{j=i}^{k-1} S_j \cdot x_j + (1 - x_i)\,(R_i - C_i)}{T_i} \right\rceil \cdot C_i \leq t, \tag{1}$$

where $x_i = 1$ if $U_i(R_i - C_i) > S_i \times \sum_{\ell=1}^{i} U_\ell$, and $x_i = 0$ otherwise [6].

Note that the above bound is safe if the DSS task τ_k releases a single job, or if the obtained bound on R_k is not larger than the task deadline, and the deadline is constrained (i.e., $D_k \leq T_k$).

The time complexity of this approach is pseudo-polynomial in the representation of the task set [6].

2) Transformation of EDD tasks into DSS tasks: To analyze EDD tasks, we propose to reuse the analysis for DSS tasks presented in the previous section. To achieve this, Theorem 1 below explains how we can safely transform a set of EDD

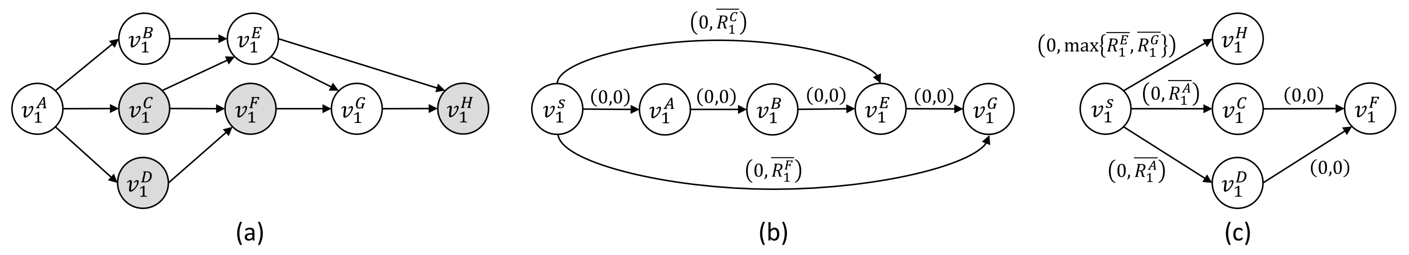

Figure 4. (a) DAG G^{P} of example sporadic parallel DAG task τ^{P}. (b) Projection of task τ^{P} on processor P_1. (c) Projection of task τ^{P} on processor P_2.

tasks into a set of equivalent DSS tasks for which Equation (1) provides an upper bound on their WCRTs.

Theorem 1. *An EDD task $\tau_i = (G_i, T_i, D_i, \pi_i)$ can be safely modeled by a dynamic self-suspending task $\tau_i' = (C_i', S_i', D_i, T_i, \pi_i)$, where*

$$C_i' = \sum_{v_i^a \in V_i} C_i^a, \tag{2}$$

and

$$S_i' = \max_{v_i^a \in \mathrm{sink}(G_i)} \{S_i^a\}, \tag{3}$$

where, for each subtask $v_i^a \in V_i$,[1]

$$S_i^a = \overset{0}{\max_{v_i^b \in \mathrm{ipred}(v_i^a)}} \left\{ S_i^b + W_i^{b,a} \right\}.$$

Proof: By definition, an EDD task τ_i, and thus the equivalent dynamic self-suspending task, cannot execute for more than $\sum_{v_i^a \in V_i} C_i^a$ time units. Hence, Equation (2) holds. Then, consider an arbitrary job J_i of τ_i and let s_i denote the actual cumulative amount of time J_i can be suspended for (i.e., pending without any ready subtask). To prove the theorem, it remains to show that the maximum amount of time an EDD task τ_i can be suspended for is bounded by Equation (3), that is, $s_i \leq S_i'$.

For each subtask $v_i^a \in V_i$, let r_i^a and f_i^a denote the time at which v_i^a becomes ready for execution and the time at which it completes its execution, respectively. Also, let s_i^a represent the actual amount of time J_i can be suspended for between its release time and the ready time r_i^a of v_i^a. Clearly, it must be that $s_i = \max_{v_i^a \in V_i} \{s_i^a\}$, as no suspensions can occur after the latest time at which a subtask $v_i^a \in V_i$ becomes ready. In addition, it can be seen that $s_i = \max_{v_i^a \in V_i} \{s_i^a\} = \max_{v_i^a \in \mathrm{sink}(G_i)} \{s_i^a\}$. As a result, it must be proven that

$$s_i = \max_{v_i^a \in \mathrm{sink}(G_i)} \{s_i^a\} \leq S_i = \max_{v_i^a \in \mathrm{sink}(G_i)} \{S_i^a\}.$$

Showing that $s_i^a \leq S_i^a$ for each $v_i^a \in \mathrm{sink}(V_i)$ proves that $s_i \leq S_i$. Equivalently, showing that $s_i^a \leq S_i^a$ for each $v_i^a \in V_i$ proves that $s_i \leq S_i$. Hence, it is sufficient to show that the following holds for each $v_i^a \in V_i$:

$$s_i^a \leq S_i^a = \overset{0}{\max_{v_i^b \in \mathrm{ipred}(v_i^a)}} \left\{ S_i^b + W_i^{b,a} \right\}.$$

The expression for S_i^a is a recursive function; therefore, we proceed by structural induction on the set of immediate predecessors.

Base case. The base case of the recursion S_i^a is obtained when v_i^a has no immediate predecessors. The proof for the base case consists in showing that $s_i^a \leq S_i^a$ whenever v_i^a has no immediate predecessors, i.e., $\mathrm{ipred}(v_i^a) = \emptyset$. The maximum operator on the empty set has a value of 0; hence, it must be shown that $s_i^a \leq S_i^a = 0$. According to the system model, a node v_i^a that has no immediate predecessors becomes ready for execution as soon as J_i is released. As a result, no suspension can occur before v_i^a becomes ready, which implies $s_i^a = 0$.

Inductive step. The inductive case of the recursion S_i^a is obtained when v_i^a has at least one immediate predecessor. The inductive hypothesis is that $s_i^b \leq S_i^b$ for each v_i^b in $\mathrm{ipred}(v_i^a)$, and, given the hypothesis, the proof consists in showing that $s_i^a \leq S_i^a = \max_{v_i^b \in \mathrm{ipred}(v_i^a)} \left\{ S_i^b + W_i^{b,a} \right\}$.

In order to prove this statement, we proceed by contradiction. Assume that $s_i^a > S_i^a$. The inductive hypothesis implies that

$$S_i^a = \max_{v_i^b \in \mathrm{ipred}(v_i^a)} \left\{ S_i^b + W_i^{b,a} \right\} \geq \max_{v_i^b \in \mathrm{ipred}(v_i^a)} \left\{ s_i^b + W_i^{b,a} \right\}.$$

It follows that

$$s_i^a > S_i^a \geq \max_{v_i^b \in \mathrm{ipred}(v_i^a)} \left\{ s_i^b + W_i^{b,a} \right\}.$$

This inequality holds if and only if $s_i^a - s_i^b > W_i^{b,a}$ for each $v_i^b \in \mathrm{ipred}(v_i^a)$. This means that the fraction of actual cumulative suspension time that takes place in the interval $(r_i^b, r_i^a]$ is strictly greater than the waiting time $W_i^{b,a}$ for each $v_i^b \in \mathrm{ipred}(v_i^a)$.

Let v_i^c be the last node in $\mathrm{ipred}(v_i^a)$ whose precedence constraint towards v_i^a is satisfied; that is, let $v_i^c = \mathrm{argmax}_{v_i^b \in \mathrm{ipred}(v_i^a)} \left\{ f_i^b + W_i^{b,a} \right\}$. Given this subtask v_i^c, it holds that $s_i^a - s_i^c > W_i^{c,a}$.

In the following, the cumulative suspension time in the time interval $I = (r_i^c, r_i^a]$ is analyzed. The interval I can be partitioned into two subintervals, corresponding to the time before (I_1) and after (I_2) the completion of v_i^c; that is, $I_1 = (r_i^c, f_i^c]$, and $I_2 = (f_i^c, r_i^a]$. In the interval I_1, J_i cannot be suspended as at least one subtask (v_i^c itself) is ready for execution for the whole duration of the time interval. Therefore, the suspension time in the interval $I = (r_i^c, r_i^a]$ must all happen in interval I_2, i.e., after the completion of v_i^c, which implies that J_i must be suspended for $s_i^a - s_i^c$ time units during I_2. Since v_i^c is the last node for which the precedence constraint towards v_i^a is satisfied, the length of I_2 cannot be

[1]$\max^0_{x \in \mathcal{S}}\{f(x)\} = \max\{0, \max_{x \in \mathcal{S}}\{f(x)\}\}$, where $f(\cdot)$ is a function on x.

larger than $W_i^{c,a}$. This implies $s_i^a - s_i^c \leq W_i^{c,a}$, thus yielding a contradiction that proves the induction step.

The theorem follows since both the base case and the induction step of the structural induction are proven. ∎

Note that the value of S_i' in Equation (3) corresponds to the maximum worst-case cumulative delay encountered over any path of G_i, i.e., $S_i' = \max_{\lambda \in \text{path}(G_i)} \left\{ \sum_{e_i^{a,b} \in E(\lambda)} W_i^{a,b} \right\}$. Obtaining this value is equivalent to computing the length of the longest path in a weighted DAG. Thus, the time complexity of converting an EDD task into a DSS task is $\mathcal{O}(|V_i| + |E_i|)$ (i.e., linear in the size of the DAG G_i).

3) Response-time upper bounds: The above theorem allows to analyze a set of EDD tasks as a set of DSS tasks using Equation (1).

Furthermore, a corollary can also be devised to individually bound the response time of each subtask of an EDD task.

Corollary 1. *If the EDD task $\tau_i \in \tau$ is schedulable, then the worst-case response time of a node $v_i^a \in V_i$ of $\tau_i = (G_i, T_i, D_i, \pi_i)$ is bounded by the worst-case response time $\overline{R_i^a}$ of a dynamic self-suspending task $\tau_i' = (C_i', S_i', D_i, T_i, \pi_i)$, where*

$$C_i' = \sum_{v_i^b \in V_i \setminus \text{succ}(v_i^a)} C_i^b, \qquad (4)$$

and $S_i' = S_i^a$, with S_i^a defined as in Theorem 1, when all higher priority tasks in τ are transformed to DSS tasks using Theorem 1.

Proof: Note that the worst-case response time of a subtask cannot be affected by its successors: indeed, successors can execute (and hence interfere) only after the subtask of interest is completed. By leveraging this observation, it is possible to bound the response time of v_i^a by studying the EDD task, say, τ_i^*, resulting from τ_i after excluding from G_i all nodes in $\text{succ}(v_i^a)$. By Theorem 1, τ_i^* can be safely modeled by a DSS task with WCET given by Equation (4). Note also that v_i^a is a sink node of τ_i^*, and the cumulative time τ_i^* can be suspended for before the completion of v_i^a can be bounded with the same inductive argument used in the proof of Theorem 1. Hence the corollary. ∎

B. Bounding high-priority interference for each subtask

In the following, we show how to derive an upper bound on the higher-priority interference suffered by each subtask v_i^a of τ_i. This bound will provide a simple way to constrain the interference variables in the optimization problem in Section V.

First, let the set of higher-priority tasks of τ_i be defined as $\text{hp}(\tau_i)$. Then, following the results of [7], even in the presence of suspensions, any sequential computation with WCET C, released with a minimum inter-arrival time T, and with worst-case response time R cannot generate more interference than $\lceil (t + R)/T \rceil \cdot C$ in any time window of length t. Similarly to [5], the number of jobs of high-priority tasks in $\text{hp}(\tau_i)$ that can cause inter-task interference on v_i^a (within a job of τ_i) can be bounded by considering v_i^a as an independent task $\tau_i^{a\prime}$. Classical response time analysis can then be used to bound the worst-case response time of $\tau_i^{a\prime}$ by computing the smallest positive fixed point of the following recurrence: $t = C_i^a + \sum_{\tau_p \in \text{hp}(\tau_i)} \left\lceil \frac{t + R_p}{T_p} \right\rceil \cdot C_p$. Let $\hat{t}$ be the fixed point of the recurrence. Then, an upper bound $UI_{p,a}$ on the maximum interference a higher-priority task $\tau_p \in \text{hp}(\tau_i)$ can cause on v_i^a can be derived as $UI_{p,a} = \lceil (\hat{t} + R_p)/T_p \rceil \cdot C_p$.

V. MILP-BASED ANALYSIS

This section presents a *mixed-integer linear programming* (MILP) formulation to bound the WCRT of EDD tasks. The formulation uses the response time upper bounds from Section IV-A to constrain the search space and improve upon the analysis provided by Theorem 1.

A. Interference

In order to characterize the worst-case interference contributions, the following definitions are introduced.

Definition 1 (Inter-task interference)**.** *The inter-task interference $I_{p,a}$ imposed by a higher-priority task τ_p on a subtask v_i^a of task τ_i is the maximum cumulative time during which v_i^a is ready but not executing because τ_p is executing on the processor.*

Definition 2 (Self-interference)**.** *The self-interference $L_{b,a}$ imposed by a subtask v_i^b of EDD task τ_i on another subtask v_i^a of the same task is the maximum cumulative time during which v_i^a is ready but not executing because v_i^b is executing on the processor.*

In the example schedule of Figure 2, task τ_p causes inter-task interference on subtasks v_1^B, v_1^C, and v_1^D within the interval $(3, 5]$. In the same example, subtask v_1^B causes self-interference on subtask v_1^C within the intervals $(2, 3]$ and $(5, 6]$, while subtask v_1^C causes self-interference on subtask v_1^D within the interval $(6, 8]$.

Note that the definition of self-interference refers to a form of direct self-interference, as opposed to indirect (or transitive) self-interference, that is, it does not account for delays in the ready time incurred by v_i^a due to another subtask v_i^b that caused self-interference on one of the predecessors of v_i^a. In the analysis, delays accumulated by the predecessors are accounted for in the computation of the time at which subtask v_i^a becomes ready for execution.

B. Overview of the analysis approach

The problem of deriving an upper bound on the WCRT of task τ_i is formulated as a MILP problem. Assuming that upper bounds on the WCRT are available for the set of higher-priority tasks $\text{hp}(\tau_i)$, the MILP formulation for task τ_i models an arbitrary schedule of a single job of that task and the set of higher-priority tasks $\text{hp}(\tau_i)$. That schedule is characterized by a set of variables modeling the response times of the subtasks of τ_i, the actual waiting times between subtasks, and the inter-task and self-interference suffered by each subtask of τ_i.

The values of such variables for a specific schedule are unknown a priori. The role of the MILP solver is, given a set of constraints that bound the value of the variables, to associate values to each variable such that the WCRT of the task under analysis is maximized. Once an upper bound $\overline{R}_i$ on the WCRT is so derived for each task $\tau_i \in \tau$, the system is deemed schedulable if $\overline{R}_i \leq D_i$ holds for each $\tau_i \in \tau$. Note that unconstrained maximization of the objective function yields an infinite value for the response time, which would result in a schedulability test that is indeed safe but also useless. Constraints are thus introduced to exclude impossible scheduling configurations by bounding the values of the variables. In particular, each constraint in the MILP formulation encodes a specific property that is necessarily satisfied by all schedules that can be generated in the proposed system. As such, these properties correspond to necessary conditions for determining whether a certain configuration of the variables in the search space constitutes a valid schedule. This reduction in the size of the feasible region corresponds to a reduction in the range of potential values assumed by the objective function, and each added constraint contributes to a refinement of the upper bound on the resulting WCRT $\overline{R}_i$ identified by the MILP formulation, which will thus constitute a tighter bound.

This approach for the derivation of the MILP formulation brings a number of advantages in the resulting analysis. Each constraint encodes a specific necessary condition that can be inferred from some individual structural or behavioral property that governs a restricted aspect of the scheduling system, and can be independently verified in terms of theoretical soundness. This leads to a formulation that is overall more easily understandable than complex closed-form WCRT formulations, as the various subproblems are analyzed independently, and to an approach that is sound by construction.

C. MILP variables

The following variables are defined in the proposed MILP formulation, with reference to an arbitrary schedule σ_i that maximizes the response time of a job of τ_i under analysis:

- For each subtask v_i^a of τ_i, the response time $R_i^a \in [0, \overline{R_i^a}]$ in the schedule σ_i is encoded as a real variable.
- For each subtask v_i^a of τ_i, and for each higher-priority task $\tau_p \in \text{hp}(\tau_i)$, $I_{p,a} \in [0, UI_{p,a}]$ is a real variable that encodes the inter-task interference caused by jobs of τ_p on v_i^a in the schedule σ_i.
- For each two subtasks v_i^a and v_i^b of the same task τ_i, $SI_{b,a} \in \{0, 1\}$ is a binary integer variable. A variable $SI_{b,a}$ has a value of 1 if, in the schedule σ_i, v_i^b causes self-interference on v_i^a and has the largest response time among the subtasks causing self-interference on v_i^a; otherwise, $SI_{b,a} = 0$.
- For each edge $e_i^{a,b} \in E_i$, a real variable $w_i^{a,b} \in [\underline{W}_i^{a,b}, W_i^{a,b}]$ is used to encode the actual waiting time between the completion of v_i^a and the time at which v_i^b becomes ready for execution in the schedule σ_i.

Note that the upper bounds $\overline{R_i^a}$ and $UI_{p,a}$ derived in the previous section have been used to constrain the domain of variables R_i^a and $I_{p,a}$ (i.e., $0 \leq R_i^a \leq \overline{R_i^a}$ and $0 \leq I_{p,a} \leq UI_{p,a}$), hence limiting the search space to be explored by the MILP solver.

Example. With respect to the example schedule illustrated in Figure 2, the values of the inter-task interference variables are $I_{p,s} = 0$, $I_{p,A} = 0$, $I_{p,B} = 2$, $I_{p,C} = 0$, $I_{p,D} = 0$, and $I_{p,E} = 0$. In addition, in the same example schedule, subtask v_1^B causes self-interference on subtask v_1^C within the intervals $(2, 3]$ and $(5, 6]$, while subtask v_1^C causes self-interference on subtask v_1^D within the interval $(6, 8]$. Consequently, the values of the self-interference variables $SI_{b,a}$ are 0 for each $e_1^{b,a}$ in E_1 except for $SI_{B,C}$ and $SI_{C,D}$, which have a value of 1.

D. MILP formulation

The goal of the MILP formulation is to **maximize** the task's WCRT and thus to maximize following objective function:

$$R_i = \max_{v_i^a \in \text{sink}(G_i)} \{R_i^a\}.$$

The constraints that compose the MILP formulation are provided next. To simplify the presentation, some of the constraints are not directly reported in a linear form. The linearization of such constraints can be performed using standard techniques [8].

Before proceeding, for each subtask v_i^a of τ_i, it is convenient to introduce an auxiliary real variable r_i^a defined as

$$r_i^a = \max_{v_i^b \in \text{ipred}(v_i^a)} \left\{R_i^b + w_i^{b,a}\right\},$$

which denotes the time at which v_i^a becomes ready for execution. Indeed, by the definition of the EDD task model, a subtask v_i^a becomes ready when all its immediate predecessors $v_i^b \in \text{ipred}(v_i^a)$ are completed and the corresponding delays $w_i^{b,a}$ relative to the their completion are elapsed.

Constraint 1. $\forall v_i^a \in V_i, R_i^a \geq r_i^a$.

Proof: By definition, subtasks cannot complete before they are ready to execute. ■

Constraint 2. $\forall v_i^a \in V_i, \forall v_i^b \in V_i \text{ s.t. } SI_{b,a} = 1, R_i^a \geq R_i^b$.

Proof: By definition of self-interference, if v_i^b generates self-interference to v_i^a ($SI_{b,a} = 1$), then, since the subtasks of the same task are scheduled in FIFO order, v_i^a cannot complete its execution before v_i^b. ■

Constraint 3. $\forall v_i^a \in V_i$, if $\exists v_i^b \in V_i \text{ s.t. } SI_{b,a} = 1$, *then*

$$R_i^a \leq R_i^b + C_i^a + \sum_{\tau_p \in \text{hp}(\tau_i)} I_{p,a}.$$

Proof: If there exists a subtask v_i^b such that $SI_{b,a} = 1$, then, by definition of variables $SI_{b,a}$, it means that v_i^a suffers self-interference and v_i^b is the last subtask that self-interferes with v_i^a. Hence, after the completion of v_i^b, which occurs at its response time R_i^b, v_i^a can only be interfered by higher-priority tasks. By definition of variables $I_{p,a}$, each high-priority task τ_p can interfere with v_i^a for at most $I_{p,a}$ time units. Subtask v_i^a itself can also execute for at most its WCET C_i^a. Hence, it follows that the response time R_i^a of v_i^a must be no later than $C_i^a + \sum_{\tau_p \in \text{hp}(\tau_i)} I_{p,a}$ time units after v_i^b's completion. ■

Constraint 4. $\forall v_i^a \in V_i$, if $\nexists v_i^b \in V_i \text{ s.t. } SI_{b,a} = 1$, *then*

$$R_i^a \leq r_i^a + C_i^a + \sum_{\tau_p \in \text{hp}(\tau_i)} I_{p,a}.$$

Proof: If there is no subtask v_i^b such that $SI_{b,a} = 1$, then, by definition of variables $SI_{b,a}$, it means that v_i^a only suffers interference from higher-priority tasks after becoming ready at time r_i^a. Thus, for the same reasons discussed in the proof of Constraint 3, the response time R_i^a of v_i^a must be no later than $C_i^a + \sum_{\tau_p \in \mathrm{hp}(\tau_i)} I_{p,a}$ time units after it becomes ready. ■

Constraint 5.

$$\forall v_i^a \in V_i, \forall v_i^b \in \{\mathrm{pred}(v_i^a) \cup \mathrm{succ}(v_i^a) \cup \{v_i^a\}\}, SI_{b,a} = 0.$$

Proof: The predecessors of subtask v_i^a must be completed when v_i^a is ready, while the successors can only execute when v_i^a is already completed. Hence, none of such subtasks can interfere with v_i^a. Finally, v_i^a cannot self-interfere with itself. ■

Constraint 6. $\forall v_i^a \in V_i, \sum_{v_i^b \in V_i} SI_{b,a} \leq 1$.

Proof: By definition of variable $SI_{b,a}$, for each subtask v_i^a, at most one subtask v_i^b satisfies $SI_{b,a} = 1$. ■

Constraint 7. $\forall v_i^a \in V_i, \forall v_i^b \in V_i, v_i^a \neq v_i^b, SI_{a,b} \leq 1 - SI_{b,a}$.

Proof: If v_i^b self-interferes with v_i^a ($SI_{b,a} = 1$), then, being the subtasks of the same task scheduled in FIFO order, v_i^b is completed when v_i^a starts executing, which means that v_i^a cannot self-interfere with v_i^b; thus, $SI_{a,b} \leq 1 - 1 = 0$. Otherwise ($SI_{b,a} = 0$), v_i^a may self-interfere with v_i^b, therefore $SI_{a,b} \leq 1 - 0 = 1$. ■

Constraint 8. $\forall v_i^a \in V_i, \forall v_i^b \in V_i$, *s.t.* $r_i^a > r_i^b$, $SI_{a,b} = 0$.

Proof: Since the subtasks of the same task are scheduled in FIFO order, if v_i^a becomes ready after v_i^b, i.e., $r_i^a > r_i^b$, then v_i^a cannot self-interfere with v_i^b. ■

Constraint 9. $\forall v_i^a \in V_i, \forall v_i^b \in V_i$, *s.t.* $R_i^a < r_i^b$, $SI_{a,b} = SI_{b,a} = 0$.

Proof: If v_i^a completes before the time v_i^b becomes ready, then the execution of v_i^a and v_i^b does not overlap in time. Hence, v_i^a cannot self-interfere with v_i^b and vice versa. ■

Constraint 10. *Let* $p(v_i^a) = \{\mathrm{pred}(v_i^a) \cup \{v_i^a\}\}$. $\forall \tau_p \in \mathrm{hp}(\tau_i), \forall v_i^a \in V_i$,

$$\sum_{v_i^b \in p(v_i^a)} \left(I_{p,b} + \sum_{v_i^c \in V_i | SI_{c,b} = 1} I_{p,c} \right) \leq \left\lceil \frac{R_i^a + R_p}{T_p} \right\rceil \cdot C_p.$$

Proof: Note that all subtasks $v_i^b \in p(v_i^a)$ must complete their execution by the time v_i^a completes, i.e., by R_i^a. Also, by definition of self-interference and the fact that the subtasks of the same task are scheduled in FIFO order, all subtasks v_i^c that self-interfere with any of the subtasks $v_i^b \in p(v_i^a)$ (i.e., $SI_{c,b} = 1$) must also be completed by R_i^a. In any time interval of length R_i^a, each task $\tau_p \in \mathrm{hp}(\tau_i)$ can interfere with at most $\lceil (R_i^a + R_p)/T_p \rceil$ jobs [6]. Hence, by definition of variables $I_{p,c}$, the left-hand side of the above inequality is upper-bounded by $\lceil (R_i^a + R_p)/T_p \rceil \cdot C_p$. ■

Constraint 11. $\forall \tau_p \in \mathrm{hp}(\tau_i), \forall v_i^a \in V_i$,

$$I_{p,a} \leq \left\lceil \frac{R_i^a - r_i^a + R_p}{T_p} \right\rceil_0 \cdot C_p,$$

where $\lceil x \rceil_0 = \max\{0, \lceil x \rceil\}$.

Proof: A subtask can suffer inter-task interference only from the time it becomes ready, i.e., r_i^a, to the time it completes, i.e., R_i^a. In any time interval of length $R_i^a - r_i^a$, each task $\tau_p \in \mathrm{hp}(\tau_i)$ can interfere with at most $\lceil (R_i^a - r_i^a + R_p)/T_p \rceil$ jobs [6]. Thus, the constraint follows after recalling the definition of the $I_{p,a}$ variables and observing that the number of such jobs cannot be negative. ■

The number of variables and the number of constraints involved in the resulting nonlinear problem formulation for a task τ_i are both bounded by $\mathcal{O}(|V_i|^2 + |V_i|n + |E_i|)$. The formulation can be linearized with standard techniques, retaining polynomial size with respect to the number of tasks n and the size of the DAG G_i. Linearization of conditional constraints, maximum operators, and conditional sum operators can be performed by means of auxiliary indicator variables, additional constraints, and a large constant M to represent infinity, in what is commonly referred to as the Big M method. Similarly, the ceiling operator can be linearized by means of an appropriately constrained auxiliary integer variable.

VI. Analyzing partitioned parallel tasks

This section presents how to analyze a set of parallel real-time tasks, scheduled with partitioned fixed-priority scheduling, using the EDD task model. Formally, under the sporadic DAG task model, a parallel task τ_i^{P} is characterized by the tuple $(G_i^{\mathrm{P}}, T_i, D_i, \pi_i)$, where $G_i^{\mathrm{P}} = (V_i^{\mathrm{P}}, E_i^{\mathrm{P}})$ is a DAG that describes the structure of the task, T_i and D_i are the period and the relative deadline of the task, respectively, and π_i is the task priority. Each node $v_i^{\mathrm{P},j} \in V_i^{\mathrm{P}}$ denotes a sequential computation, i.e., a subtask, with WCET $C_i^{\mathrm{P},j}$, and is allocated to a fixed processor. The edges in E_i^{P} denote precedence constraints among subtasks, i.e., each edge connecting node $v_i^{\mathrm{P},a}$ to node $v_i^{\mathrm{P},b}$ specifies that subtask $v_i^{\mathrm{P},b}$ can start executing only after the completion of subtask $v_i^{\mathrm{P},a}$. All subtasks are released with the same period T_i, have the same priority π_i, and are subject to the same deadline D_i.

Model transformation. A parallel task of this kind can be modeled and analyzed by means of a set of synchronously-released EDD tasks, as described in the following. Given a parallel DAG task τ_i^{P} executing on a set of processors P, for each processor $P_k \in P$, an EDD task $\tau_i^k = \mathcal{P}_k(\tau_i)$ to be executed on P_k is defined. $\mathcal{P}_k(\tau_i)$ is called the *projection* of task τ_i^{P} on processor P_k. The computational structure of the EDD task τ_i^k is described by the DAG $G_i^k = (V_i^k, E_i^k)$ constructed as follows:

1) Add a source node v_i^s with $C_i^s = 0$ to V_i^k to model the task release.
2) For each subtask $v_i^{\mathrm{P},j} \in V_i^{\mathrm{P}}$ allocated to P_k, create a corresponding node v_i^j with $C_i^j = C_i^{\mathrm{P},j}$ and add it to V_i^k.
3) For each edge in E_i^{P} that connects two subtasks $v_i^{\mathrm{P},a}$ and $v_i^{\mathrm{P},b}$ allocated to P_k, create an edge in E_i^k labeled

with $(0,0)$ that connects the corresponding nodes v_i^a and v_i^b of the EDD task τ_i^k.

4) For each subtask $v_i^{\mathrm{P},j} \in V_i^{\mathrm{P}}$ assigned to processor P_k, create an edge in E_i^k that connects the source node v_i^s to the corresponding node v_i^j. If $v_i^{\mathrm{P},j}$ has at least one immediate predecessor assigned to a processor other than P_k, then label the edge with $(0, R^*)$, where R^* is the maximum response time of the predecessors of $v_i^{\mathrm{P},j}$ in V_i^{P} allocated to processors other than P_k. Otherwise, label the edge with $(0,0)$.

The resulting EDD tasks have the same period, deadline, and priority as τ_i^{P}. An example of such a transformation is illustrated in Figure 4 and discussed in Section III.

Analysis techniques. Given the above model transformation, a parallel task τ_i^{P} can be deemed schedulable if all the EDD tasks resulting from its projection on each processor $P_k \in P$ are schedulable. Note that, by Corollary 1, the response time of each node v_i^a of an EDD task can be bounded via a transformation to a DSS task whose suspension time depends only on the delays on the edges connecting either v_i^a or v_i^a's predecessors. This means that the EDD tasks corresponding to the projections can be constructed by simply exploring τ_i^{P} in topological order, starting from the source node.

The MILP formulation of Section V can be slightly modified to allow the *simultaneous* analysis of all the projections of parallel task τ_i^{P}. The approach consists in deriving a specialized MILP formulation for each parallel DAG task τ_i^{P} given its set of projections on each processor. Unfortunately, due to space limitations, it is not possible to report the detailed MILP formulation for analyzing parallel tasks. Instead, we discuss the small set of changes that must be made to the formulation presented in Section V.

The variables in the modified MILP formulation are defined as the union of the variables associated to all EDD tasks resulting from the projection $\mathcal{P}_k(\tau_i)$ on every processor $P_k \in P$. Note that, given the use of partitioned scheduling, only higher-priority EDD tasks allocated to the same processor can generate inter-task interference. Hence, for each each subtask v_i^a of a projection $\mathcal{P}_k(\tau_i)$, the corresponding variables $I_{p,a}$ must be defined only for higher-priority tasks running on P_k.

The objective function corresponds to the maximum response time of the projection of τ_i^{P} on each processor, that is,

$$R_i = \max_{P_k \in P} \max_{v_i^a \in \mathrm{sink}(\mathcal{P}_k(\tau_i))} \{R_i^a\}.$$

This corresponds to maximizing the response time of all the sink nodes of the original task τ_i^{P}, which corresponds to the response time of τ_i^{P}.

As in the MILP formulation of Section V, variables $w_i^{s,a}$ are constrained as $\underline{W}_i^{s,a} \leq w_i^{s,a} \leq W_i^{s,a}$, where $\underline{W}_i^{s,a} = 0$. However, in this case, $W_i^{s,a}$ is not a constant but a MILP real *variable* that is constrained to be equal to the maximum response time of the predecessors executing on other processors in the original task τ_i^{P}, that is, $\forall e_i^{s,a} \in E_i^k$:

$$W_i^{s,a} = \max_{v_i^{\mathrm{P},c} \in \mathrm{ipred}(\mathcal{X}_i^{-1}(v_i^a))} \left\{ R_i^b \mid v_i^b = \mathcal{X}_i(v_i^{\mathrm{P},c}) \notin V_i^k \right\},$$

where $\mathcal{X}_i(v_i^{\mathrm{P},c})$ and $\mathcal{X}_i^{-1}(v_i^a)$ are functions that return, respectively, the node v_i^b of the projected EDD task that corresponds to the node $v_i^{\mathrm{P},c}$ in the original DAG task τ_i^{P}, and the node $v_i^{\mathrm{P},c}$ of τ_i^{P} that corresponds to the node v_i^a in the projection.

When solving the MILP, variables $W_i^{s,a}$ will be adjusted by the solver according to the constraints on the projections in order to synthesize the scheduling configuration that yields the maximum overall response time of the parallel DAG task.

The resulting size of the MILP for a task τ_i is polynomial with respect to the number of tasks n, the number of processors $|P|$, and the size of the DAG G_i.

Note that the response time of the parallel DAG task τ_i^{P} constitutes an upper bound of the response time of the various projections $\mathcal{P}_k(\tau_i)$. Such an upper bound is needed when propagating the resulting values to the analysis of lower-priority tasks (see Constraints 10 and 11).

VII. Experimental results

This section presents the results of an experimental evaluation that compares the two analysis techniques proposed for EDD tasks (DSS-based from Section IV and MILP-based from Section V). Furthermore, the performance of the analysis of parallel tasks under partitioned scheduling by means of EDD tasks presented in Section VI is evaluated against federated scheduling.

A. Experiments on EDD tasks

Generation of EDD tasks. Synthetic workload for the experiments has been generated as follows. The number n of EDD tasks to be generated for each task set is fixed within the experiments. For each EDD task in the task set, the topology of the DAG is generated according to the technique proposed by Melani et al. [9]. In this approach, a series-parallel graph is first generated starting from two disconnected nodes by recursively expanding non-terminal nodes to either terminal nodes or parallel subgraphs until a maximum recursion depth is reached. This results in a series-parallel graph with multiple nested levels of parallel branches. In the generation procedure, the maximum recursion depth is modeled as a generation parameter n_{rec}, while the probability for a non-terminal node to expand to a parallel subgraph is another generation parameter p_{par}. The number of branches to which the non-terminal node is expanded to is selected from the discrete uniform distribution $[2, n_{par}]$. The resulting series-parallel graph is then transformed into a DAG by randomly adding edges between pairs of nodes with a probability p_{add}, provided that each added edge does not introduce cycles in the graph.

In the following, let $C_i = \sum_{v_i^j \in V_i} C_i^j$ and $U_i = C_i / T_i$ represent the cumulative WCET and the utilization factor of each EDD task τ_i, respectively. In order to avoid biasing effects in the workload generation, the UUniFast algorithm [10] is used to generate the utilization factor U_i for each EDD task τ_i, such that $\sum_{\tau_i \in \tau} U_i = U$, where U is the total system utilization. Once the DAG topology is obtained, the minimum inter-arrival time T_i of τ_i is selected from a discrete log-uniform distribution in the range $[T_{min}, T_{max}]$, and the deadline is set to $D_i = T_i$ (implicit deadlines). The cumulative WCET C_i is then set to $C_i = T_i \cdot U_i$, and the WCET C_i^j of each node

$v_i^j \in V_i$ is generated by applying the UUniFast algorithm to partition the available cumulative WCET C_i among the DAG nodes, such that $\sum_{v_i^j \in V_i} C_i^j = C_i$. In particular, UUniFast is used to uniformly select n_i real values $\hat{c}_i^j \in [0,1]$ with the constraint that $\sum_{v_i^j \in V_i} \hat{c}_i^j = 1$; then, the value of the WCET C_i^j for each node $v_i^j \in V_i$ is set to $C_i^j = C_i \cdot \hat{c}_i^j$.

The minimum waiting time $\underline{W}_i^{a,b}$ of each edge $e_i^{a,b} \in E_i$ is set to 0. In order to generate the maximum waiting time $W_i^{a,b}$ of each edge $e_i^{a,b} \in E_i$, the cumulative waiting time W_i to be distributed among the nodes $v_i^j \in V_i$ is set to $W_i = \beta \cdot (D_i - C_i)$, where $\beta \in [0,1]$ is a real value that is used to control the overall amount of suspensions incurred in the execution of the task and is referred to as *suspension factor*. The maximum waiting time $W_i^{a,b}$ of each edge $e_i^{a,b} \in E_i$ is then generated by applying the UUniFast algorithm to partition the overall waiting time W_i among the edges, such that $\sum_{e_i^{a,b} \in E_i} W_i^{a,b} = W_i$. In particular, UUniFast is used to uniformly select $|E_i|$ real values $\hat{w}_i^{a,b} \in [0,1]$ with the constraint that $\sum_{e_i^{a,b} \in E_i} \hat{w}_i^{a,b} = 1$; then, the value of the maximum waiting time $W_i^{a,b}$ of each edge $e_i^{a,b} \in E_i$ is set to $W_i^{a,b} = W_i \cdot \hat{w}_i^{a,b}$. Finally, the resulting EDD tasks are assigned priorities π_i using Rate Monotonic.

Schedulability results. The experiment evaluates the schedulability ratio obtained with the DSS-based analysis (Theorem 1) and the MILP-based analysis. In this evaluation, the total utilization U was varied from 0 to 1 with a step of 0.1. Figure 5 reports the results of a representative configuration where the generation parameters were set to $n = 6$, $n_{rec} = 2$, $p_{par} = 0.8$, $n_{par} = 2$, $p_{add} = 0.2$, $T_{min} = 100$, $T_{max} = 1000$, and $\beta = 0.8$. For each value of U, 100 synthetic task sets were tested with both analyses to derive the corresponding schedulability ratio. The plot shows that system schedulability decreases for both approaches with higher system utilization, as it is to be expected, and that the MILP-based approach retains a slight edge in schedulability performance with respect to the DSS-based analysis. The experiments presented in the next section show that the MILP-based analysis is instead much more effective when analyzing parallel tasks.

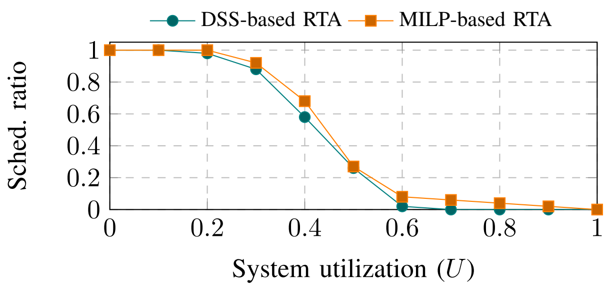

Figure 5. Schedulability ratio as a function of the system utilization U.

B. Experiments on parallel tasks

This section compares partitioned scheduling of parallel tasks, analyzed via EDD tasks, against federated scheduling [1]. Federated scheduling has been selected for comparison purposes because it is a popular solution, and because it is the closest to partitioned scheduling in terms of approach, runtime overhead, and implied predictability (given that only a subset of tasks is managed under global/clustered scheduling, while the remainder are handled with pure partitioned scheduling). Clearly, partitioned scheduling may have been compared to several other solutions (e.g., global scheduling). However, the resulting comparison would not be totally fair from a practical perspective, considering that more convoluted scheduling solutions tend to have several practical drawbacks when compared to a simple approach like partitioned scheduling. For instance, note that partitioned scheduling allows providing fine-grained control of memory contention [11]–[13] and tight blocking bounds in the presence of locking [14], as the computational activities (i.e., the subtasks) are all statically allocated to a single processor, and hence the contention generated by each processor can be more precisely bounded than with global or semi-partitioned scheduling.

Generation of parallel tasks. The task synthesis procedure for parallel tasks is derived from the generation algorithm employed for the synthesis of EDD tasks. For a given parallel task τ_i, the procedure leverages the same approach used in the generation of an EDD task τ_i^{EDD} for what concerns the generation of the DAG topology and of the task scheduling parameters T_i, D_i, and C_i^j, with a difference in how the utilization value U_i is generated. Specifically, the experiments on parallel tasks involve the generation of task sets with system utilization U larger than 1. In order to avoid biases in the generation of the utilization factors U_i for each parallel task τ_i, UUniFast is used to uniformly select n real values $\hat{U}_i \in [0,1]$ with the constraint that $\sum_{i=1}^{n} \hat{U}_i = 1$; then, the value of the utilization factor U_i for each task τ_i is set to $U_i = U \cdot \hat{u}_i$.

Since in this case the utilization U_i of a parallel task τ_i may be larger than 1, it is possible that the task generation procedure for τ_i results in an infeasible allocation of the WCET values C_i^j. To limit such occurrences, the generation procedure for a parallel task τ_i is repeated (up to 100000 times) in case either **(i)** $C_i^j > T_i$ for some node $v_i^j \in V_i$; or **(ii)** $\sum_{v_i^j \in V(\lambda)} C_i^j > T_i$ for some path $\lambda \in \text{path}(G_i)$. The value of an additional Boolean generation parameter b_{cond} determines whether the check on the second condition is effectively applied ($b_{cond} = true$) or not ($b_{cond} = false$).

Federated scheduling. The federated scheduling algorithm for parallel tasks by Li et al. [1] works by dividing the task set into two disjoint sets: the set of high-utilization (or heavy) tasks τ_{high}, which contains all tasks τ_i with $U_i \geq 1$, and the set of low-utilization (or light) tasks τ_{low}, which contains the remaining tasks, i.e., those with $U_i < 1$. Each heavy task is assigned $m_i = \lceil \frac{C_i - L_i}{D_i - L_i} \rceil$ dedicated processors, where L_i denotes the worst-case critical path length, which is computed as $L_i = \max_{\lambda \in \text{path}(G_i)} \left\{ \sum_{v_i^j \in V(\lambda)} C_i^j \right\}$. Once each heavy task is assigned its set of dedicated processors, the remaining processors are assigned to the light tasks. Each light task is treated as a sequential task by considering a topologically ordered sequential execution of its nodes and allocating them to the same processor. If a valid allocation of processors is determined, each heavy task can be scheduled on its assigned processors by a work-conserving global scheduler, while light tasks are scheduled on the remaining processors by a partitioned scheduler.

Partitioning algorithms. When testing the schedulability of parallel tasks under partitioned scheduling, the nodes must be partitioned in some way among the processors. This work considers three partitioning algorithms. **(WBF)** Tasks are sorted by decreasing utilization. Then, for each task, nodes are sorted by decreasing utilization and assigned to a processor using a standard partitioning heuristic chosen among Worst-Fit, Best-Fit, and First-Fit, where the fitting is determined by checking that the processor utilization does not exceed one. Each partitioning heuristic is tried in turn and the logic OR of their implied schedulability performance is taken. **(Pseudo-federated)** As under federated scheduling, tasks are divided into two disjoint sets: heavy tasks and light tasks, with processor allocation following the same rules in place for federated scheduling of parallel tasks [1]. The difference with respect to federated scheduling is that heavy tasks are scheduled on the assigned processors by a partitioned scheduler, as opposed to a global scheduler. The nodes of a heavy task τ_i are allocated on the m_i dedicated processors using WBF, while light tasks are allocated as sequential tasks (i.e., all their nodes are allocated to the same core) using WBF. **(Pseudo-federated++)** It works as pseudo-federated, with the following differences. Each heavy task τ_i is assigned a dedicated number of processors m_i, starting from $\lceil U_i \rceil$ and incremented by one until a feasible allocation of the nodes is found. As under pseudo-federated, the nodes of a heavy task τ_i are allocated on the m_i dedicated processors using WBF, while light tasks are allocated as sequential tasks using WBF. Heavy tasks are allocated first; then, if no feasible allocation is found for a light task τ_i, the nodes of τ_i are spread using WBF on all the available processors (including those dedicated to heavy tasks) instead of a single processor.

Schedulability results. We tested six different scheduling approaches: **(EDD-WFB)** the DSS-based analysis under partitioning with WFB; **(P-FED)** and **(P-FED-MILP)** the DSS-based and MILP-based analyses under partitioning with pseudo-federated, respectively; **(P-FED++)** and **(P-FED++-MILP)** the DSS-based and MILP-based analyses under partitioning with pseudo-federated++, respectively; and **(FED-WBF)** federated scheduling [1] using WBF to partition light tasks under Rate Monotonic scheduling. In these experiments, m is used to denote the number of processors. Figure 6 reports the results of this experiment in terms of schedulability analysis with respect to the system utilization U for various system configurations. For each value of U, 500 synthetic task sets were tested. For each configuration, the values of n_{rec}, n_{par}, and p_{add} were set to $n_{rec} = 2$, $n_{par} = 2$, and $p_{add} = 0.2$, respectively. The value of b_{cond} was set to $false$ for the configurations in Figures 6(a-f), and to $true$ for the configurations in Figures 6(g-i). In case $b_{cond} = true$, the ratio of task sets which satisfy both feasibility conditions in the generation is reported by the PAR-FEAS curve, which constitutes an upper bound on the obtainable schedulability performance. The other generation parameters (m, n, p_{par}, T_{min}, T_{max}) were varied among the experiments and their specific value is reported above each graph. In all the experiments, the total utilization U was varied from 0 to m with a step of 0.5.

The results for $m = 4$ (i.e., Figures 6(a-c)) show that partitioned scheduling of parallel tasks, when allocated according to pseudo-federated++ and analyzed through EDD tasks, outperforms federated scheduling by a significant margin. The gap between the approaches becomes larger when increasing the number of tasks from 6 to 8. These results are extremely relevant given the higher predictability of partitioned scheduling, especially when adopted in conjunction with techniques to control memory contention or locking protocols. The results also show that the performance of the pseudo-federated and federated scheduling techniques is comparable, given their similar approach for task allocation. The improvements provided by the MILP-based analysis are in this case larger than those observed for single-processor EDD tasks. Similar trends can be observed when $m = 8$ (i.e., Figures 6(d-f)). Finally, Figures 6(g-i) show how the schedulability performance varies with larger values of m and n when the additional check in the task synthesis procedure is active (i.e., $b_{cond} = true$). In this case, the increase in the ratio of feasible task sets results in a wider margin for improvement for the most performing approaches (P-FED++ and P-FED++-MILP), which can then benefit from a further increase in performance. In particular, the advantage of P-FED++-MILP over P-FED++ becomes very significant, reinforcing the relevance of the MILP-based analysis.

Table I reports statistics on the runtime of each approach. We report the minimum, maximum and average runtime for the experiment of Figure 6(a), for a machine equipped with an Intel Xeon E5-2640 v4 processor, with 10 multithreaded cores running at a 2.40 GHz base operating frequency, and 24 GB DRAM. These measurements show that the analyses proposed in this paper are suitable for offline system design and optimization workflows.

VIII. Related work

Various task models have been proposed in previous work to account for suspension-related delays. However, they either do not explicitly consider suspensions originated by waiting for events among portions of code, or are less expressive than the model proposed in this paper.

The most relevant related work to the present paper pertains to the literature on self-suspending tasks, which is excellently reviewed in the survey by Chen et al. [7]. The segmented self-suspending task model [5] is generalized by the EDD task model, while the dynamic self-suspending task model allows to safely analyze EDD tasks, as proven in Section IV. The hybrid self-suspending task model [15] extends the dynamic self-suspending task model by including a limit on the maximum number of suspensions. This model is still not capable of dealing with more complex execution and suspension behaviors that can be captured by EDD tasks thanks to their graph-based structure.

Previous work also studied the delays implied by hardware acceleration with self-suspending task models (either directly or indirectly through the analysis of locking protocols). Examples of this kind of works are those by Dong et al. [16], Patel et al. [17], and Elliot et al. [18] for GPUs, and by Biondi et al. [19] for FPGAs. However, such works considered synchronous acceleration requests, while the EDD task model is particularly effective in dealing with asynchronous acceleration, as explained in Section III.

In the context of distributed systems, the work by Gutiérrez et al. [20] shows how to transform complex task and message topologies (including modeling elements such as forks and joins) into a set of linear sequences. However, the framework

Table I. Analysis runtimes.

	EDD-WBF	P-FED	P-FED-MILP	P-FED++	P-FED++-MILP	FED-WBF
Minimum analysis runtime (ms)	1.56	0.08	0.08	0.63	0.63	0.06
Maximum analysis runtime (ms)	236.52	55.76	25025.80	124.86	70377.68	2.34
Average analysis runtime (ms)	27.18	2.41	200.67	7.30	231.79	1.06

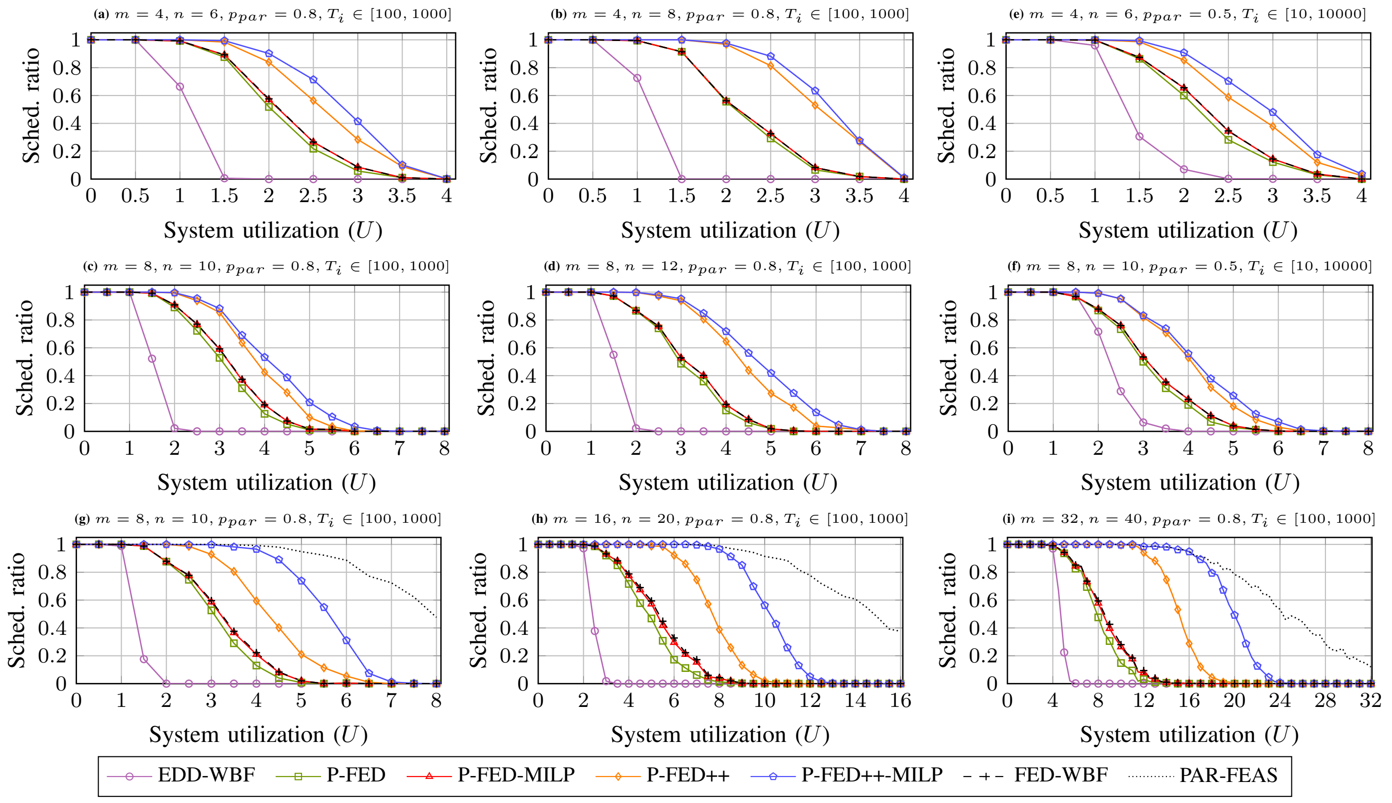

Figure 6. Schedulability ratio as a function of system utilization U for several system configurations. (a-f): $b_{cond} = false$; (g-i): $b_{cond} = true$.

does not include the possibility of explicitly modeling variable delays on the edges, and the available analysis techniques are considered pessimistic [20], [21]. Nonetheless, it might be possible to extend the framework to support the modeling elements required to represent EDD tasks. The MAST modeling and analysis toolset [22], which includes an implementation of the framework in [20], constitutes a possible option for modeling the behavior of EDD tasks through existing or additional modeling facilities.

Previous work also analyzed parallel tasks under partitioned scheduling by means of self-suspending tasks. Fonseca et al. [21] considered preemptive tasks and used segmented self-suspending tasks for analysis purposes. As EDD tasks generalize segmented self-suspending tasks, this work also generalizes the approach in [21]. Furthermore, the analysis method proposed in this work is much simpler than that of [21], which requires the execution of a complex recursive algorithm to perform the model transformation. Casini et al. [23] considered parallel tasks with non-preemptable nodes and used limited-preemptive DSS tasks for analysis purposes.

IX. Conclusion and future work

This paper presented the EDD task model. EDD tasks are meant to explicitly model complex computing workloads that incur delays due to waiting for events. We discussed how to use EDD tasks to model asynchronous hardware acceleration and the execution of parallel tasks under partitioned scheduling. Two analysis approaches for EDD tasks have been provided, one based on a model transformation to DSS tasks, and the other based on a MILP formulation. Then, we showed that parallel real-time tasks under partitioned scheduling can be modeled via a set of EDD tasks that capture the projections of the parallel task on each processor.

Experimental results showed that, for single-processor scheduling of EDD tasks, the DSS-based analysis provides performance close to that of the MILP-based analysis. Conversely, the MILP-based analysis is much more effective when EDD tasks are used to analyze parallel tasks. Partitioned scheduling of parallel tasks has been shown to outperform federated scheduling, with a significant gain in system utilization in all the tested configurations.

Considering the generality and the modeling power of EDD tasks, future work should evaluate the applicability of the model to other kinds of workloads and address its application under Earliest Deadline First (EDF) scheduling and in the presence of semi-partitioning of nodes on multicores. Future work should also further explore the case of partitioned parallel tasks analyzed by means of EDD tasks, possibly devising specialized partitioning algorithms based on structural properties of EDD tasks.

Acknowledgment

This work has been supported by the EU H2020 project AMPERE under the grant agreement no. 871669.

References

[1] J. Li, J.-J. Chen, K. Agrawal, C. Lu, C. Gill, and A. Saifullah, "Analysis of federated and global scheduling for parallel real-time tasks," in *Proceedings of the 26th Euromicro Conference on Real-Time Systems (ECRTS 2014)*. IEEE, 2014, pp. 85–96.

[2] M. Stigge, P. Ekberg, N. Guan, and W. Yi, "The digraph real-time task model," in *Proceedings of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2011)*. IEEE, 2011, pp. 71–80.

[3] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings, "Applying new scheduling theory to static priority pre-emptive scheduling," *Software Engineering*, vol. 8, no. 5, pp. 284–292, 1993.

[4] J.C. Palencia Gutiérrez and M. González Harbour, "Schedulability analysis for tasks with static and dynamic offsets," in *Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS 1998)*. IEEE, 1998, pp. 26–37.

[5] G. Nelissen, J. Fonseca, G. Raravi, and V. Nelis, "Timing analysis of fixed priority self-suspending sporadic tasks," in *Proceedings of the 27th Euromicro Conference on Real-Time Systems (ECRTS 2015)*. IEEE, 2015, pp. 80–89.

[6] J.-J. Chen, G. Nelissen, and W.-H. Huang, "A unifying response time analysis framework for dynamic self-suspending tasks," in *Proceedings of the 28th Euromicro Conference on Real-Time Systems (ECRTS 2016)*. IEEE, 2016, pp. 61–71.

[7] J.-J. Chen, G. Nelissen, W.-H. Huang, M. Yang, B. Brandenburg, K. Bletsas, C. Liu, P. Richard, F. Ridouard, N. Audsley, R. Rajkumar, and G. von der Brüggen, "Many suspensions, many problems: A review of self-suspending tasks in real-time systems," *Real-Time Systems*, vol. 55, no. 1, pp. 144–207, 2019.

[8] D. Bertsimas and J. N. Tsitsiklis, *Introduction to linear optimization*, ser. Athena Scientific series in optimization and neural computation. Athena Scientific, 1997, vol. 6.

[9] A. Melani, M. Bertogna, V. Bonifaci, A. Marchetti-Spaccamela, and G. C. Buttazzo, "Response-time analysis of conditional DAG tasks in multiprocessor systems," in *Proceedings of the 27th Euromicro Conference on Real-Time Systems (ECRTS 2015)*. IEEE, 2015, pp. 211–221.

[10] E. Bini and G. C. Buttazzo, "Measuring the performance of schedulability tests," *Real-Time Systems*, vol. 30, no. 1-2, pp. 129–154, 2005.

[11] R. I. Davis, S. Altmeyer, L. S. Indrusiak, C. Maiza, V. Nelis, and J. Reineke, "An extensible framework for multicore response time analysis," *Real-Time Systems*, vol. 54, no. 3, pp. 607–661, Jul 2018.

[12] D. Casini, A. Biondi, G. Nelissen, and G. Buttazzo, "A holistic memory contention analysis for parallel real-time tasks under partitioned scheduling," in *Proceedings of the 26th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2020)*. IEEE, 2020, pp. 239–252.

[13] M. Hassan and R. Pellizzoni, "Bounding DRAM interference in COTS heterogeneous MPSoCs for mixed criticality systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2323–2336, 2018.

[14] B. B. Brandenburg, "Multiprocessor real-time locking protocols: A systematic review," 2019, arXiv:1909.09600.

[15] G. von der Brüggen, W.-H. Huang, and J.-J. Chen, "Hybrid self-suspension models in real-time embedded systems," in *Proceedings of the 23rd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2017)*. IEEE, 2017, pp. 1–9.

[16] Z. Dong, C. Liu, S. Bateni, K.-H. Chen, J.-J. Chen, G. von der Brüggen, and J. Shi, "Shared-resource-centric limited preemptive scheduling: A comprehensive study of suspension-based partitioning approaches," in *Proceedings of the 24th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2018)*. IEEE, 2018, pp. 164–176.

[17] P. Patel, I. Baek, H. Kim, and R. R. Rajkumar, "Analytical enhancements and practical insights for MPCP with self-suspensions," in *Proceedings of the 24th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2018)*. IEEE, 2018, pp. 177–189.

[18] G. A. Elliott, B. C. Ward, and J. H. Anderson, "GPUSync: A framework for real-time gpu management," in *Proceedings of the 34th IEEE Real-Time Systems Symposium (RTSS 2013)*. IEEE, 2013, pp. 33–44.

[19] A. Biondi, A. Balsini, M. Pagani, E. Rossi, M. Marinoni, and G. Buttazzo, "A framework for supporting real-time applications on dynamic reconfigurable FPGAs," in *Proceedings of the 37th IEEE Real-Time Systems Symposium (RTSS 2016)*. IEEE, 2016, pp. 1–12.

[20] J.J. Gutiérrez García, J.C. Palencia Gutiérrez, and M. González Harbour, "Schedulability analysis of distributed hard real-time systems with multiple-event synchronization," in *Proceedings of the 12th Euromicro Conference on Real-Time Systems (ECRTS 2000)*. IEEE, 2000, pp. 15–24.

[21] J. Fonseca, G. Nelissen, V. Nelis, and L. M. Pinho, "Response time analysis of sporadic DAG tasks under partitioned scheduling," in *Proceedings of the 11th IEEE Symposium on Industrial Embedded Systems (SIES 2016)*. IEEE, 2016, pp. 1–10.

[22] M. González Harbour, J.J. Gutiérrez García, J.C. Palencia Gutiérrez, and J.M. Drake Moyano, "MAST: Modeling and analysis suite for real time applications," in *Proceedings of the 13th Euromicro Conference on Real-Time Systems (ECRTS 2001)*. IEEE, 2001, pp. 125–134.

[23] D. Casini, A. Biondi, G. Nelissen, and G. Buttazzo, "Partitioned fixed-priority scheduling of parallel tasks without preemptions," in *Proceedings of the 39th IEEE Real-Time Systems Symposium (RTSS 2018)*. IEEE, 2018, pp. 421–433.