Response-Time Analysis of Conditional DAG Tasks in Multiprocessor Systems

Alessandra Melani*, Marko Bertogna†, Vincenzo Bonifaci‡, Alberto Marchetti-Spaccamela§, Giorgio C. Buttazzo*
*Scuola Superiore Sant'Anna, Pisa, Italy, E-mail: {alessandra.melani, g.buttazzo}@sssup.it
†Università di Modena e Reggio Emilia, Modena, Italy, E-mail: m.bertogna@unimore.it
‡Istituto di Analisi dei Sistemi ed Informatica, CNR, Roma, Italy, E-mail: vincenzo.bonifaci@iasi.cnr.it
§Università di Roma "La Sapienza", Roma, Italy, E-mail: alberto@dis.uniroma1.it

***Abstract*—Different task models have been proposed to represent the parallel structure of real-time tasks executing on many-core platforms: fork/join, synchronous parallel, DAG-based, etc. Despite different schedulability tests and resource augmentation bounds are available for these task systems, we experience difficulties in applying such results to real application scenarios, where the execution flow of parallel tasks is characterized by multiple (and nested) conditional structures. When a conditional branch drives the number and size of sub-jobs to spawn, it is hard to decide which execution path to select for modeling the worst-case scenario.**

To circumvent this problem, we integrate control flow information in the task model, considering conditional parallel tasks (cp-tasks) represented by DAGs composed of both precedence and conditional edges. For this task model, we identify meaningful parameters that characterize the schedulability of the system, and derive efficient algorithms to compute them. A response-time analysis based on these parameters is then presented for different scheduling policies. A set of simulations shows that the proposed approach allows efficiently checking the schedulability of the addressed systems, and that it significantly tightens the schedulability analysis of non-conditional (e.g., classic DAG) tasks over existing approaches.

I. Introduction

As a larger number of multi-/many-core systems is proposed by the main hardware producer to the embedded market [1], [3], [2], there is an increasing interest for applications requiring both high-performance and real-time requirements. The real-time community has been recently active in trying to update classic schedulability analysis to such relatively new platforms, providing new models and tests to guarantee the timing requirements of parallel task systems. Different real-time task models have been proposed to capture the most salient features of parallel applications: the fork/join model [16], the synchronous parallel task model [23], and the DAG-based task model [8]. Each of these models divides each task into multiple smaller computation units called sub-tasks, that are allowed to run simultaneously on different cores.

As noted in [15], the problem introduced by conditional statements does not arise when tasks have no parallelism, as in classic single-core applications. For these systems, it is sufficient to consider one single parameter for each task, i.e., the Worst-Case Execution Time (WCET), which corresponds to the longest chain of execution among all possible conditional paths. For this reason, conditional statements are not explicitly modeled in the schedulability analysis for single core applications, while they are much more detrimental to the schedulability analysis of parallel task models.

In fact, when checking the schedulability of real parallel applications (e.g., for image processing or autonomous-driving systems), many problems arise in mapping the actual task structures to the existing task models. This is mainly due to the existence of multiple conditional branches (such as if-then-else statements) and control flow instructions in correspondence with the creation of different parallel sub-tasks. For example, consider the case of a task that conditionally elaborates an image depending on the particular system state; either it may fork a large number of sub-tasks to process small clusters of pixels in a parallel way, or it may require to run a single sub-task to process a larger sampled set of pixels in a sequential way. Moreover, the level of detail at which each image region is processed may dynamically vary depending on runtime information. For example, when an object is detected in a particular region, the processing detail for the associated region may be progressively intensified, thus reducing the number of pixels to process.

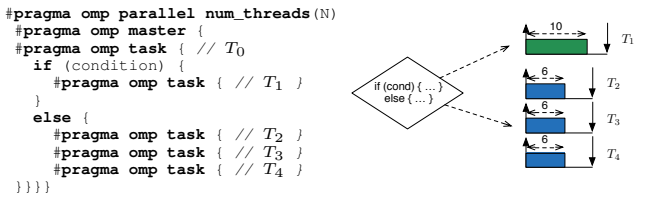

Fig. 1: A parallel program with conditional execution.

Therefore, the response-time of a task may significantly vary depending on the particular instance, as does its interfering contribution to other tasks. Identifying the scenario that mostly affects the schedulability of the system is then challenging.

Figure 1 represents a parallel task T_0 specified according to the OpenMP standard, with just one conditional statement at the beginning of execution. For simplicity, we consider only one task instance. Depending on the conditional clause, the task takes either the upper branch, creating a sequential sub-task T_1 of 10 time-units, or the lower branch, forking three sub-tasks T_2, T_3, T_4 of 6 time-units each. Which branch leads to the worst-case response-time depends on the number of cores and the interference from the other tasks. For example, with three or more cores and no interfering task, the upper branch leads to the largest response-time for every work-conserving scheduler[1] (i.e., 10 time-units instead of 6). With less cores, the largest response-time is given by the lower branch (i.e., 12 time-units with two cores, and 18 time-units with one core). If interfering tasks are added, the situation is even more challenging because the conclusions stated above may be reversed. For example, adding a sequential task of 6

[1]A scheduler is work-conserving if it never idles a core whenever there is pending workload to execute.

time-units, the worst-case response-time with three cores is now given by the lower branch (12 instead of 10).

Similarly, it is difficult to predict which branch imposes a larger interference on other tasks: depending on the characteristics of the other tasks, a higher interference may be produced by a set of parallel sub-tasks or by a longer sequential sub-task. Since applications are typically composed of several nested conditional statements, the problem of mapping parallel applications to task models that do not explicitly consider conditional statements is therefore not easy to solve.

A. Contributions and paper organization

We extend the parallel DAG model by integrating conditional constructs to provide a tighter analysis to parallel task systems. In the *conditional parallel* task (cp-task) model, each task is represented by a DAG containing both parallel and conditional nodes. In order to capture the structure of parallel applications, we provide a formal definition of cp-task by specifying the possible connections between the various (conditional and non-conditional) sections of the graph.

For this model, we derive efficient ways to compute an upper-bound on the response-time of each cp-task using different global scheduling algorithms, and show the effectiveness of our schedulability analysis by means of exhaustive simulations. Moreover, we show that the proposed response-time analysis can be efficiently applied to non-conditional task models as well (such as the DAG task model [8]). For these latter systems, our simulations show that a significantly higher number of schedulable task-sets is detected (at a considerably smaller time complexity) with respect to existing approaches.

The rest of this paper is organized as follows. In Sections II and III we review related work and introduce the proposed task model and the notation used throughout the paper. We then characterize the critical interference among tasks (Section IV), used to derive a response-time analysis (Section V) through the computation of certain task parameters of interest (Section VI). In Section VII we report the results of our simulations.

II. Related Work

There is a substantial amount of work in the real-time literature on parallel task models, but for the most part this has been limited to non-conditional execution modes. One of the first parallel task models to be proposed is the *fork-join* model by Lakshmanan, Kato and Rajkumar [16], [6], in which each task is represented as an alternating sequence of sequential and parallel segments, and every parallel segment has the same degree of parallelism (which is constrained to be less than or equal to the number of available processors). A natural extension, called the *synchronous parallel* model [23], [5], [20], [13], [19], allows consecutive parallel segments and an arbitrary degree of parallelism of every segment. Synchronization is still enforced at the boundary of each segment, in the sense that a sub-task in the new segment may start only after all sub-tasks in the previous segment have completed. A more flexible model of concurrency than either the fork-join or the synchronous parallel models is provided by the *DAG* model: a task is represented by a directed acyclic graph, where every node is a sequential sub-task, and arcs represent precedence constraints between the sub-tasks [8], [11], [17], [22], [7], [24], [18].

The *multi-DAG* model by Fonseca al. [15] is to our knowledge the first attempt at enriching a parallel task model with control-flow information. This model represents each parallel task as a collection of DAGs, each of which represents a different execution flow. A method is proposed to combine such flows into a single synchronous parallel task that preserves the execution requirements and the precedence constraints of all the execution flows that can possibly occur at runtime, thus reducing the schedulability problem to the simpler problem for synchronous parallel tasks. On the other hand, a disadvantage of the approach of [15] is that it is not scalable with respect to the number of sub-tasks, since the number of different flows through a DAG can be exponential in the number of nodes. Moreover, it adds pessimism in the task transformation process and requires server-based synchronization mechanisms that are difficult to implement.

III. System Model and Definitions

Let $\mathcal{T} = \{\tau_1, \ldots, \tau_n\}$ denote a set of n sporadic conditional parallel tasks (*cp-tasks*) that execute upon a platform consisting of m identical processors. Each cp-task τ_k releases a potentially infinite sequence of jobs. Each job of τ_k is separated from the next by at least T_k time-units, and has a constrained relative deadline $D_k \leq T_k$. Moreover, each cp-task τ_k is represented as a directed acyclic graph $G_k = (V_k, E_k)$, where $V_k = \{v_{k,1}, \ldots, v_{k,n_k}\}$ is a set of nodes (or vertices) and $E_k \subseteq V_k \times V_k$ is a set of directed arcs (or edges), see Figure 2. Each node $v_{k,j} \in V_k$ represents a sequential chunk of execution (or "sub-task") and is characterized by a worst-case execution time $C_{k,j}$. We assume preemption and migration overhead be integrated within WCET values. The arcs represent dependencies between sub-tasks, that is, if $(v_{k,1}, v_{k,2}) \in E_k$, then $v_{k,1}$ must complete before $v_{k,2}$ can begin execution. A node with no incoming arcs is referred to as a *source* of the DAG, while a node with no outgoing arcs is referred to as a *sink* of the DAG. Without loss of generality, we assume that each cp-task has exactly one source v_k^{source} and one sink node v_k^{sink}. If this is not the case, a dummy source/sink node with zero WCET can be added to the DAG, with arcs to/from all the source/sink nodes. We will omit using the subscript k for denoting parameters associated to the task τ_k whenever the reference to the task is clear in the discussion.

In our cp-task model, nodes can be of two types: a) *regular* nodes, which allow *all* successor nodes to be executed in parallel and are represented as rectangles in our figures; b) *conditional* nodes, which allow modeling the beginning and the end of a conditional (for example, if-then-else) construct. Conditional nodes come in start/end pairs and represent conditional branching by requiring execution of *exactly one* among the successors of the start node; in our figures, conditional nodes are represented by diamonds and circles, for the head and tail of the conditional pair, respectively.

To properly model the possible execution flows of a structured programming language, a further restriction is imposed to the possible connections within a conditional branch. That is, there cannot be any connection between a node belonging to a branch of a conditional statement and nodes outside that branch (including other branches of the same statement). More formally, we use the following definition.

Definition III.1. *Let (v_1, v_2) be a pair of conditional nodes in a DAG $G_k = (V_k, E_k)$. The pair (v_1, v_2) is a* conditional pair *if the following hold:*

1) *Suppose that there are exactly q outgoing arcs from v_1 to the nodes s_1, s_2, ..., s_q, for some $q > 1$. Then there are exactly q incoming arcs into v_2 in E_k, from some nodes t_1, t_2, ..., t_q.*
2) *For each $\ell \in \{1, 2, \ldots, q\}$, let $V'_\ell \subseteq V_k$ and $E'_\ell \subseteq E_k$ denote all the nodes and arcs on paths reachable from s_ℓ that do not include node v_2. By definition, s_ℓ is the sole source node of the DAG $G'_\ell := (V'_\ell, E'_\ell)$. It must hold that t_ℓ is the sole sink node of G'_ℓ.*
3) *It must hold that $V'_\ell \cap V'_j = \emptyset$ for all ℓ, j, $\ell \neq j$. Additionally, with the exception of (v_1, s_ℓ) there should be no arcs in E_k into nodes in V'_ℓ from nodes not in V'_ℓ, for each $\ell \in \{1, 2, \ldots, q\}$. That is, $E_k \cap ((V_k \setminus V'_\ell) \times V'_\ell) = \{(v_1, s_\ell)\}$ should hold for all ℓ.*

As in the classical DAG task model, two quantities of interest are the length and the volume of a task. We define a *chain* or *path* of a cp-task τ_k as a sequence of nodes $\lambda = (v_{k,a}, \ldots, v_{k,b})$ such that $(v_{k,j}, v_{k,j+1}) \in E_k, \forall j \in [a, b)$. The *length* of the chain, denoted by $\text{len}(\lambda)$, is the sum of the WCETs of all its nodes. That is, for a chain of task τ_k, the value $\sum_{j=a}^{b} C_{k,j}$. The *longest path* of a cp-task is any source-sink path of the task that achieves the longest length.

Definition III.2. *The* length *of a cp-task τ_k, denoted by L_k, is the length of any longest path of τ_k.*

Note that L_k also represents the minimum worst-case execution time of cp-task τ_k, that is, the time required to execute it when the number of processing units is sufficiently large (potentially infinite) to allow the task to always execute at maximum parallelism. A necessary condition for the feasibility of a cp-task τ_k is then $L_k \leq D_k$.

In the absence of conditional branches, the classical sporadic DAG task model defines the *volume* of the task as the worst-case execution time needed to complete it on a dedicated single-core platform [8], [11], [17], [24]. This quantity can be computed as the sum of the WCETs of all the sub-tasks, that is $\sum_{v_{k,j} \in V_k} C_{k,j}$. In the presence of conditional branches, assuming that all sub-tasks are always executed may be overly pessimistic. Hence, we generalize the concept of volume of a cp-task by introducing the notion of *worst-case workload*.

Definition III.3. *The* worst-case workload W_k *of a cp-task τ_k is the maximum time needed to execute an instance of τ_k on a dedicated single-core platform, where the maximum is taken among all possible choices of conditional branches.*

Section VI explains in detail how the worst-case workload of a task can be computed efficiently.

The *utilization* U_k of a cp-task τ_k is the ratio between its worst-case workload and period, that is, $U_k = W_k/T_k$. For the task-set $\mathcal{T}$, its *total utilization* is defined as $U_\mathcal{T} = \sum_{i=1}^{n} U_i$. A simple necessary condition for feasibility is $U_\mathcal{T} \leq m$.

Example. Figure 2 illustrates an example cp-task consisting of nine sub-tasks (nodes) $V = \{v_1, \ldots, v_9\}$ and twelve precedence constraints (arcs). The number inside each node represents its WCET. Two of the nodes, v_2 and v_6, form a conditional pair, meaning that only one sub-task between v_3 and v_4 will be executed (but never both), depending on a conditional clause. The (longest path) length of this cp-task is $L = 8$, and is given by the chain $(v_1, v_2, v_4, v_6, v_7, v_9)$. Its volume is 14 units, while its worst-case workload must take into account that either v_3 or v_4 are executed at every task instance. Since v_4 corresponds to the branch with the largest workload, we obtain $W = 11$.

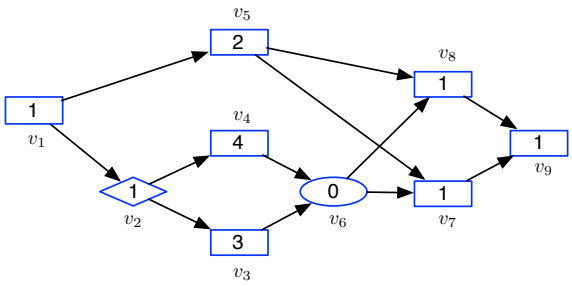

Fig. 2: An example cp-task. Each vertex is labeled with the WCET of the corresponding sub-task.

TABLE I: Notation

$\mathcal{T}$	set of cp-tasks	n	number of tasks in $\mathcal{T}$
τ_k	k-th task of $\mathcal{T}$	D_k	relative deadline of τ_k
T_k	period of τ_k	G_k	DAG associated to τ_k
V_k	node set of G_k	E_k	arc set of G_k
$v_{k,j}$	jth sub-task of τ_k	$C_{k,j}$	WCET of $v_{k,j}$
L_k	length of τ_k's longest chain	W_k	worst-case workload of τ_k
U_k	utilization of τ_k		

To further clarify the restrictions imposed to the graph structure, note that there cannot be any arc connecting v_4 to v_5, because this would violate the correctness of conditional constructs and the semantics of the precedence relation. In fact, v_5 could start executing only when v_4 has terminated; however, if the branch corresponding to v_3 is taken, v_5 should then wait for the completion of a sub-task that will never be executed. Analogously, a connection from v_5 to to v_4 would not make sense in case the branch corresponding to v_3 is taken. Indeed, v_4 belongs to one of the branches induced by the conditional pair (v_2, v_6), while v_5 does not; therefore, by Definition III.1, no arc is allowed to join v_4 to v_5 or vice versa. Similarly, no connection is possible between v_4 and v_3, as they belong to different branches of the same conditional statement.

Table I summarizes the notation used throughout the paper.

IV. Critical interference of cp-tasks

In this section, a schedulability analysis is presented for cp-tasks globally scheduled with any work-considering scheduler. The analysis is based on the notion of interference. In the existing literature for globally scheduled sequential task systems, the *interference* on a task τ_k is defined as the sum of all intervals in which τ_k is ready, but it cannot execute because all cores are busy executing other tasks. We modify this definition to adapt it to the parallel nature of cp-tasks, by introducing the concept of critical interference [13], [19].

Given a set of cp-tasks $\mathcal{T}$ and a (work-conserving) scheduling algorithm, we first define the notion of *critical chain*.

Definition IV.1. *The critical chain λ_k^* of a cp-task τ_k is the chain of nodes of τ_k that leads to its worst-case response-time R_k.*

In particular, the critical chain λ_k^* of a cp-task τ_k can be identified taking the sink vertex v_k^{sink} of the worst-case instance of τ_k (i.e., the job of τ_k that has largest response-time in the worst-case scenario), and recursively pre-pending the last to complete among the predecessor nodes (either conditional or not) until the source vertex $v_{k,1}$ has been included in the chain.

A *critical node* of task τ_k is a node that belongs to τ_k's critical chain. Since the response-time of a cp-task is given by the response-time of the sink vertex of the task, it follows that the sink node is always a critical node. For deriving the worst-case response-time of a task, it is then sufficient to characterize the maximum interference suffered by its critical chain.

Definition IV.2. *The* critical interference *I_k on task τ_k is defined as the maximum cumulative time in which any critical node of the worst-case instance of τ_k is ready but it cannot execute because all cores are busy.*

Lemma IV.1. *Given a set of cp-tasks $\mathcal{T}$ scheduled by any work-conserving algorithm on m identical processors, the worst-case response-time of each task τ_k is*

$$R_k = \text{len}(\lambda_k^*) + I_k. \tag{1}$$

Proof. Let r_k be the release time of the worst-case instance of τ_k. In the scheduling window $[r_k, r_k + R_k]$, the critical chain will require $\text{len}(\lambda_k^*)$ time-units to complete. By Definition IV.2, the interference from non-critical nodes of τ_k and from other cp-tasks $\tau_{i \neq k}$ is I_k. The lemma simply follows noting that the response-time of τ_k corresponds to the response-time of its sink node, which is the last node of the critical chain. □

The problem in using Equation (1) for schedulability analysis is that it is difficult to exactly bound the interference imposed on the considered task. An established solution is to express the total interfering workload as a function of individual contributions of the interfering tasks, and then upper-bound such contributions with the worst-case workload of each interfering task τ_i. In the following, we explain how such interfering contributions can be computed, and how they relate to each other to determine the total interfering workload.

Definition IV.3. *The* critical interference *$I_{i,k}$ imposed by task τ_i on task τ_k is defined as the cumulative workload executed by sub-tasks of τ_i while a critical node of the worst-case instance of τ_k is ready to execute but is not executing.*

Lemma IV.2. *For any work-conserving algorithm, the following relation holds:*

$$I_k = \frac{1}{m} \sum_{\tau_i \in \mathcal{T}} I_{i,k}. \tag{2}$$

Proof. By the work-conserving property of the scheduling algorithm, whenever a critical node of τ_k is interfered, all m cores are busy executing other sub-tasks. The total amount of workload executed by sub-tasks interfering with the critical chain of τ_k is then mI_k. Hence, $\sum_{\tau_i \in \mathcal{T}} I_{i,k}(L) = mI_k(L)$, and by reordering the terms, the lemma follows. □

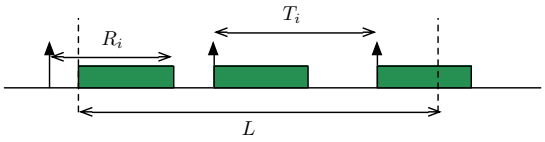

Fig. 3: Worst-case scenario to maximize the workload of an interfering task τ_i in the sequential case.

In the particular case when $i = k$, the critical interference $I_{k,k}$ may include the interfering contributions of non-critical sub-tasks of τ_k on itself, that is, the *self-interference*.

By combining Equations (1) and (2), we obtain that the response-time of a task τ_k is given by

$$R_k = \text{len}(\lambda_k^*) + \frac{1}{m} I_{k,k} + \frac{1}{m} \sum_{\tau_i \in \mathcal{T}, i \neq k} I_{i,k}. \tag{3}$$

V. Response-time analysis

We now show how to derive an upper-bound on the worst-case response-time of each cp-task. The first problem in using Equation (3) to derive a response-time analysis is related to the computation of the interfering contributions $I_{i,k}$. To sidestep this problem, we compute an upper-bound on the workload that an interfering task τ_i may produce within the scheduling window $[r_k, r_k + R_k]$ of the worst-case instance of a task τ_k, and use this value as an upper-bound on the interference $I_{i,k}$.

A. Inter-task interference

Following a typical approach adopted in the response-time analysis for globally scheduled systems [9], [19], we divide the contribution to the workload of an interfering task τ_i in a window of interest between carry-in, body, and carry-out jobs. The *carry-in* job is the first instance of τ_i that is part of the window of interest and has release time before and deadline within the window of interest. The *carry-out* job is the last instance of τ_i executing in the window of interest, having a deadline after the window of interest. All other instances of τ_i are named *body jobs*.

For sequential task-sets, an upper-bound on the workload of an interfering task τ_i within a window of length L is found in a situation where (i) the first job of τ_i starts executing as late as possible, with a starting time aligned with the beginning of the window of interest, and (ii) later jobs are executed as soon as possible [9]. Such a situation is depicted in Figure 3.

For cp-task systems, it is much more difficult to find a configuration that maximizes the carry-in and carry-out contributions. In fact:

1) Due to the precedence constraints and different degree of parallelism of the various execution paths of a cp-task, it may happen that a larger workload is executed within the window if the interfering task is shifted left, i.e., by decreasing the carry-in and increasing the carry-out contributions. This happens for example when the first part of the carry-in job has little parallelism, while the carry-out part at the end of the window contains multiple parallel sub-tasks.

2) A sustainable schedulability analysis [12] has to guarantee that all tasks meet their deadlines even when some of them execute less than the worst-case. For example, one of the

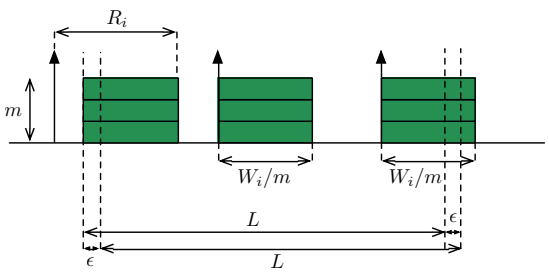

Fig. 4: Worst-case scenario to maximize the workload of an interfering cp-task τ_i.

sub-tasks of an execution path of a cp-task may execute for less than its WCET $C_{i,j}$. This may lead to larger interfering contributions within the window of interest, for example if a parallel section of a carry-out job is included in the window due to an earlier completion of a preceding sequential section. 3) The carry-in and carry-out contribution of a cp-task may correspond to different conditional paths of the same task, with different levels of parallelism.

To overcome the above problems, we consider a scenario in which each interfering job of task τ_i executes for its worst-case workload W_i. This quantity (which we show how to compute in Section VI) represents the maximum amount of workload that can be generated by a single instance of a cp-task. The next lemma provides a safe upper-bound on the workload of a task τ_i within a window of interest of length L.

Lemma V.1. *An upper-bound on the workload of an interfering task τ_i in a window of length L is given by*

$$\mathcal{W}_i(L) = \left\lfloor \frac{L + R_i - W_i/m}{T_i} \right\rfloor W_i + \\ + \min\left(W_i, m \cdot ((L + R_i - W_i/m) \mod T_i)\right).$$

Proof. Consider a situation in which all instances of τ_i execute for their worst-case workload W_i. The highest workload within a window of length L for such a task configuration is produced when the carry-in and carry-out contributions are evenly distributed among all cores, as shown in Figure 4. Note that distributing the carry-in or carry-out contributions on a lesser number of cores may not possibly increase the workload within the window. Moreover, other task configurations with a smaller workload for the carry-in or carry-out instance cannot lead to a higher workload in the window of interest: although a reduced carry-in workload may allow including a larger part of the carry-out (as in shifting right the window of interest by $\epsilon = \Delta W_i/m$ in the figure), the carry-out part that enters the window from the right cannot be larger than the carry-in reduction.

In the considered scenario, an upper-bound on the number of carry-in and body instances that may execute within the window is

$$\left\lfloor \frac{L + R_i - W_i/m}{T_i} \right\rfloor,$$

each one contributing for W_i. The portion of the carry-out job included in the window of interest is $(L + R_i - W_i/m) \mod T_i$. Since at most m cores may be occupied by the carry-out job within that interval, and the carry-out job cannot execute for more than W_i units, the lemma follows. □

When global EDF is used as a scheduling algorithm, it is possible to find an additional upper-bound on the interfering contribution of each task by noting that the deadline of the interfering jobs cannot be later than that of the interfered task.

Lemma V.2. *An upper-bound on the interfering workload of a task τ_i on a task τ_k with global EDF is given by*

$$\mathcal{I}_{i,k} = \left(\left\lfloor \frac{D_k - D_i}{T_i} \right\rfloor + 1\right) W_i + \\ + \min\left(W_i, m \cdot \max\left(0, D_k \bmod T_i - D_i + R_i\right)\right).$$

Proof. Consider a window $[r_k, r_k + D_k]$ of a task τ_k. The interfering contribution of a task τ_i is maximized when the deadline of the carry-out job is aligned with $r_k + D_k$. In such a situation, the number of body and carry-out jobs of τ_i is $\lfloor (D_k - D_i)/T_i \rfloor + 1$, each contributing for W_i. Concerning the carry-in job, its scheduling window overlaps with the considered window for $D_k \mod T_i$. Subtracting the slack $(D_i - R_i)$, we obtain that the carry-in job may execute for at most $(D_k \mod T_i) - (D_i - R_i)$ within the considered window, when this term is not negative. Since the carry-in job can occupy at most m cores, and its worst-case workload is bounded by W_i, the lemma follows. □

B. Intra-task interference

After deriving valid upper-bounds for the $I_{i,k}$ terms of Equation (3), we now consider the remaining terms of the response-time equation, which take into account the contribution of the considered task to its overall response-time. We hereafter derive an upper-bound[2] on the sum $Z_k \stackrel{\text{def}}{=} \text{len}(\lambda_k^*) + \frac{1}{m} I_{k,k}$.

Lemma V.3. *For a constrained deadline cp-task system scheduled with any work-conserving algorithm, the following relation holds for any task τ_k:*

$$Z_k = \text{len}(\lambda_k^*) + \frac{1}{m} I_{k,k} \leq L_k + \frac{1}{m}(W_k - L_k). \quad (4)$$

Proof. Since we are in a constrained deadline setting, a job will never be interfered by other jobs of the same task. Being W_k the maximum possible workload produced by a job of cp-task τ_k, the portion that may interfere with the critical chain λ_k^* is $W_k - \text{len}(\lambda_k^*)$. Then, $I_{k,k} \leq W_k - \text{len}(\lambda_k^*)$. Hence,

$$\text{len}(\lambda_k^*) + \frac{1}{m} I_{k,k} \leq \text{len}(\lambda_k^*) + \frac{1}{m}(W_k - \text{len}(\lambda_k^*)). \quad (5)$$

Since $\text{len}(\lambda_k^*) \leq L_k$ and $m \geq 1$, the lemma follows. □

C. Schedulability condition

Using the bounds derived in this section to replace the corresponding terms in Equation (3), the following theorem can be proved [9].

Theorem V.1. *Given a cp-task set globally scheduled on m cores, an upper-bound R_k^{ub} on the response-time of a task τ_k*

[2]Despite Z_k includes only the contribution of task τ_k, it would be wrong to identify this parameter with the makespan of τ_k, i.e., the worst-case response-time of τ_k when it has no interference from other tasks. See Appendix A.

can be derived by the fixed-point iteration of the following expression, starting with $R_k^{ub} = L_k$*:*

$$R_k^{ub} \leftarrow L_k + \frac{1}{m}(W_k - L_k) + \left\lfloor \frac{1}{m} \sum_{\forall i \neq k} \mathcal{X}_i^{ALG} \right\rfloor,$$

where, with global FP:

$$\mathcal{X}_i^{ALG} = \mathcal{X}_i^{FP} = \begin{cases} \mathcal{W}_i(R_k^{ub}), & \forall i < k \\ 0, & otherwise \end{cases};$$

with global EDF:

$$\mathcal{X}_i^{ALG} = \mathcal{X}_i^{EDF} = \min\left\{\mathcal{W}_i(R_k^{ub}), \mathcal{I}_{i,k}\right\};$$

and $\mathcal{X}_i^{ALG} = \mathcal{W}_i(R_k^{ub})$ *for any work-conserving scheduler.*

The schedulability of a cp-task system can then be simply checked using Theorem V.1 to compute an upper-bound on the response-time of each task. In the FP case, the bounds are updated in decreasing priority order, starting from the highest priority task. In this case, it is sufficient to apply Theorem V.1 only once for each task. Instead, in the EDF or general work-conserving cases, multiple rounds may be necessary. All bounds are initially set to $R_k^{ub} = L_k, \forall \tau_k \in \mathcal{T}$. Then, Theorem V.1 is used to compute a response-time bound for each task τ_k. The procedure continues until either (i) one of the response-time bounds exceeds the corresponding task deadline (returning a negative schedulability result), or (ii) no more update is possible (returning a schedulable condition).

VI. Computation of cp-task parameters

The response-time analysis of Theorem V.1 shows that the schedulability of a conditional DAG task system can be simply checked if, beside deadline and period, two characteristic parameters are known for each cp-task τ_k: the worst-case workload W_k and the length of the longest chain L_k. Then, an upper-bound on the response-time of each task can be easily computed with Theorem V.1 in pseudo-polynomial time.

The longest path of a cp-task can be computed exactly in the same way as for the longest path of a classical DAG task, since any conditional branch defines a set of possible paths in the graph. For this purpose, conditional nodes can be considered as if they were simply regular nodes. The computation can be implemented in time linear in the size of the DAG by standard techniques, see e.g. Bonifaci et al. [11] and references therein.

The computation of the worst-case workload of a cp-task is not as easy. We hereafter show an algorithm to compute W_k for each task τ_k.

A. Worst-case workload computation

We first compute a topological order of the DAG[3]. Then, exploiting the (reverse) topological order, a simple dynamic program can compute for each node the accumulated workload corresponding to the portion of the graph already examined. The algorithm must distinguish the case when the node under analysis is the head of a conditional pair or not. If this is the case, then the maximum accumulated workload among the successors is selected, otherwise the sum of the workload contributions of all successors is computed.

The pseudo-code for determining the worst-case workload of a cp-task is shown in Algorithm 1. This algorithm takes as input the graph representation of a cp-task G and outputs its worst-case workload W. In the algorithm, for any set of nodes S, its total WCET is denoted by $C(S)$. First, at line 2, a topological sorting of the vertices is computed and stored in the permutation σ. Then, the permutation σ is scanned in reverse order, that is, from the (unique) sink to the (unique) source of the DAG. At each iteration of the for loop at line 4, a node v_i is analyzed; a set variable $S(v_i)$ is used to store the set of nodes achieving the worst-case workload of the subgraph including v_i and all its descendants in the DAG. Since the sink node has no successors, $S(v^{\text{sink}})$ is initialized to $\{v^{\text{sink}}\}$ at line 3. Then, the function $\text{SUCC}(v_i)$ computes the set of successors of v_i. If that set is not empty, function $\text{IsBeginCond}(v_i)$ is invoked to determine whether v_i is the head node of a conditional pair. If so, the node v^* achieving the largest value of $C(S(v))$, among $v \in \text{SUCC}(v_i)$, is computed (line 7). The set $S(v^*)$ therefore achieves the maximum cumulative worst-case workload among the successors of v_i, and is then used to create $S(v_i)$ together with v_i. Instead, whenever v_i is not the head of a conditional pair, all its successors are executed at runtime. Therefore, the workload contributions of all its successors must be merged into $S(v_i)$ (line 10) together with v_i. The procedure returns the worst-case workload accumulated by the source vertex, that is $C(S(v^{\text{source}}))$.

Algorithm 1 Worst-Case Workload Computation

```
1:  procedure WCW(G)
2:      σ ← TopologicalOrder(G)
3:      S(v^sink) ← {v^sink}
4:      for v_i ∈ σ from sink to source do
5:          if SUCC(v_i) ≠ ∅ then
6:              if IsBeginCond(v_i) then
7:                  v* ← argmax_{v∈SUCC(v_i)} C(S(v))
8:                  S(v_i) ← {v_i} ∪ S(v*)
9:              else
10:                 S(v_i) ← {v_i} ∪ ⋃_{v∈SUCC(v_i)} S(v)
11:             end if
12:         end if
13:     end for
14:     return C(S(v^source))
15: end procedure
```

The complexity of the algorithm is quadratic in the size of the input DAG. Indeed, there are $O(|E|)$ set operations performed throughout the algorithm, and some operations on a set S (namely, the ones at line 7) also require computing $C(S)$, which has cost $O(|V|)$. So the time complexity is $O(|V||E|)$. To implement the set operations, set membership arrays are sufficient.

One may be tempted to simplify the procedure by avoiding the use of set operations, keeping track only of the cumulative worst-case workload at each node, and allowing a linear complexity in the DAG size. However, such an approach would lead to an overly pessimistic result. Consider a simple graph with a source node forking multiple parallel branches which

[3]A topological order is such that if there is an arc from u to v in the DAG, then u appears before v in the topological order. A topological order can be easily computed in time linear in the size of the DAG (see any basic algorithm textbook, such as [14]).

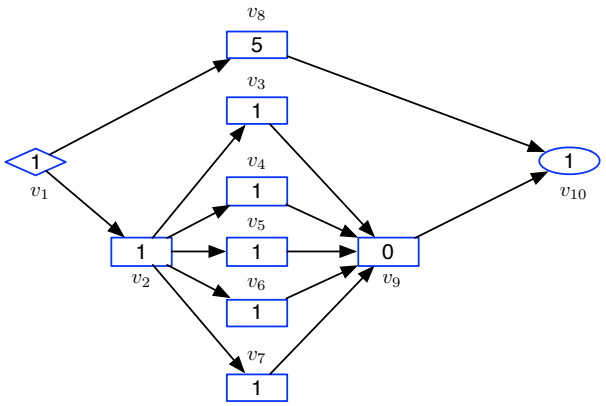

Fig. 5: Example of cp-task that shows the pessimism of the upper-bound given in Equation (4).

then converge on a common sink. The cumulative worst-case workload of each parallel path includes the contribution of the sink. If we simply sum such contributions to derive the cumulative worst-case workload of the source, the contribution of the sink would be counted multiple times. Set operations are therefore needed to avoid accounting multiple times each node contribution.

B. Improved upper-bound on intra-task interference

Once W_k and L_k are known for a task τ_k, Equation (4) gives an upper-bound on the term Z_k that accounts for the contribution of the considered task τ_k to its worst-case response-time. However, due to the presence of conditional branches, such an upper-bound might be pessimistic. As an example, consider the cp-task τ_k in Figure 5, which executes on a platform composed of $m = 2$ processors.

This cp-task has a longest path length of 7 time-units (given by the upper branch), and a worst-case workload $W_k = 8$ time-units (given by the lower branch). When $m = 2$, Equation (4) gives a bound on Z_k of 7.5. However, if the upper branch is taken after the completion of v_1, only the longest path of τ_k would be executed, yielding a value of $Z_k = 7$ time-units. Instead, if the lower branch is taken, only the corresponding portion of the graph would be executed, with an upper-bound of $Z_k \leq 4 + 4/2 = 6$ time-units. Hence, in both cases, the upper-bound computed by (4) would be pessimistic.

This is mainly due to the fact that Equation (4) considers the worst-case situation where, *simultaneously*, i) the critical path of G_k is executed; and ii) the total worst-case workload of τ_k is experienced. However, given the internal structure of the cp-task of Figure 5, this situation can never happen.

The example intuitively suggests that the bound in Equation (4) can be further improved by *jointly* computing the worst-case workload and the longest chain length for each portion of the cp-task, so that both values refer to the same conditional branch. Specifically, for a given chain λ of τ_k, let W_k^λ be the maximum workload attainable by τ_k *when the conditional choices are compatible with* λ. Then, arguing similarly as in Lemma V.3, we get:

Lemma VI.1.

$$Z_k \leq \text{len}(\lambda_k^*) + \frac{1}{m}(W_k^{\lambda_k^*} - \text{len}(\lambda_k^*))$$
$$\leq \max_\lambda \left(\text{len}(\lambda) + \frac{1}{m}(W_k^\lambda - \text{len}(\lambda)) \right)$$

where λ ranges over all source-sink paths of τ_k.

Algorithm 2 shows the pseudo-code computing an improved bound on Z_k, which takes into account the issue discussed above by computing jointly the worst-case workload and the contributions of different subgraphs of the task.

Algorithm 2 Z_k Bound Computation

```
1:  procedure ZBound(G, m)
2:      σ ← TopologicalOrder(G)
3:      S(v^sink) ← {v^sink}
4:      T(v^sink) ← {v^sink}
5:      f(v^sink) ← C^sink
6:      for v_i ∈ σ from sink to source do
7:          if succ(v_i) ≠ ∅ then
8:              if IsBeginCond(v_i) then
9:                  v* ← argmax_{v∈succ(v_i)} C(S(v))
10:                 S(v_i) ← {v_i} ∪ S(v*)
11:                 u* ← argmax_{u∈succ(v_i)} f(u)
12:                 T(v_i) ← {v_i} ∪ T(u*)
13:                 f(v_i) ← C_i + f(u*)
14:             else
15:                 S(v_i) ← {v_i} ∪ ⋃_{v∈succ(v_i)} S(v)
16:                 u* ← argmax_{u∈succ(v_i)} (f(u) +
17:                  +∑_{w∈succ(v_i),w≠u} C(S(w) \ T(u))/m)
18:                 T(v_i) ← {v_i} ∪ T(u*)
19:                 f(v_i) ← C_i + f(u*)+
20:                  +∑_{w∈succ(v_i),w≠u*} C(S(w) \ T(u*))/m
21:             end if
22:         end if
23:     end for
24:     return f(v^source)
25: end procedure
```

This algorithm takes as input a given task graph G and the number of processors composing the platform, and outputs an upper-bound on the task's Z_k value. As for Algorithm 1, a topological sorting of the nodes is required (line 2). Three variables for each node v_i are used by the algorithm to store intermediate results: $S(v_i)$, as in Algorithm 1, is a set representing the nodes that determine the largest partial workload from v_i till the end of the DAG; $f(v_i)$ stores the bound on the partial Z_k value from node v_i to the end of the DAG, including the full contribution of nodes belonging to the partial longest chain (stored in set $T(v_i)$) and the workload contribution over m cores due to other nodes of the same conditional instance. The computation of the values $S(v_i)$ (lines 3, 10, 15) is exactly as in Algorithm 1. In the following, we focus on the computation of $f(v_i)$ and $T(v_i)$.

Since the sink node has no successors, we initialize $T(v^{\text{sink}})$ to $\{v^{\text{sink}}\}$ and $f(v^{\text{sink}})$ to C^{sink}. The algorithm's main loop iterates over the nodes of G in reverse topological order (line 6). If the node under analysis has some successor, different

actions are taken depending on whether v_i is the head of a conditional pair or not. In the former case, we compute the successor u^* that maximizes the intermediate upper-bound on Z_k, and set $f(v_i)$ and $T(v_i)$ accordingly. If, instead, a parallel branch is departing from the current node v_i, the workload by all the successors will be transferred to v_i. Hence, the driving logic is to determine the successor u that yields the largest combined value of its partial Z_k bound ($f(u)$) plus the total self-interference from other nodes, which is bounded by

$$\sum_{w \in \text{SUCC}(v_i), w \neq u} C(S(w) \setminus T(u))/m.$$

Note that the set $T(u)$ is subtracted from the set to consider for the self-interfering contribution, because such nodes are already fully accounted for in the term $f(u)$.

As for the case of Algorithm 1, this procedure can be implemented efficiently, that is, to run in polynomial time in the size of the graph: the complexity of Algorithm 2 is $O(|V||E|\,\Delta)$, where Δ denotes the maximum out-degree of a node. In fact, similarly to the analysis of Algorithm 1, the complexity of the algorithm is $O(|V||E|)$, plus the cost of executing the instructions at lines $16-17$. The cost of performing such an instruction once is $O(|V|\delta(v_i)^2)$, where $\delta(v_i)$ is the out-degree of node v_i; since $\delta(v_i) \leq \Delta$, it follows that the total cost of the instructions at line $16-17$ is

$$O\left(|V| \sum_i \delta(v_i)^2\right) = O\left(|V|\Delta \sum_i \delta(v_i)\right) = O(|V||E|\,\Delta).$$

This cost dominates, in the worst case, the cost of other operations; hence, the complexity of Algorithm 2 is $O(|V||E|\,\Delta)$.

VII. Experimental Results

In this section, we validate the performance of the schedulability tests proposed in Section V in terms of number of schedulable task-sets against existing approaches, both for global FP and global EDF scheduling. In particular, we show that our response-time analysis not only outperforms the existing approaches for analyzing parallel tasks with conditional branches, but it is also significantly superior to the state-of-the-art techniques in the particular case with no conditional branches, i.e., for classic DAG tasks.

All the algorithms compared in our experiments have been implemented in MATLAB®. The code is fully available online [4].

A. Generation of cp-tasks

Concerning the simulation environment, we refer to [21] as a baseline to generate cp-tasks. In that work, series-parallel graphs are generated by recursively expanding *blocks* (i.e., non-terminal vertices) either to terminal vertices or to conditional subgraphs, until a maximum recursion depth is reached, hence permitting multiple nested levels of conditional branches. We extend the derivation rules given in [21] by considering that non-terminal vertices can be expanded to either terminal vertices, conditional subgraphs or parallel subgraphs. The probabilities that control such three events are p_{term}, p_{cond}, and p_{par}, respectively, subject to the relation $p_{term}+p_{cond}+p_{par}=1$. Moreover, we specify the maximum number of branches of parallel and conditional subgraphs (n_{par} and n_{cond}, respectively). Based on these values, whenever a non-terminal vertex is expanded to a parallel subgraph, the number of branches is uniformly selected in $[2, n_{par}]$. Analogously, whenever a conditional subgraph is generated, the number of its branches is uniformly selected in $[2, n_{cond}]$. By applying this methodology, a series-parallel graph can be obtained. However, in this work we address the analysis of a more general class of task graphs, i.e., DAGs that respect the structural restrictions mentioned in Section III, imposed by conditional pairs. For this reason, we randomly add edges between pairs of nodes with a certain probability p_{add}, provided that the structural restrictions of our cp-task model (see Definition III.1) are not violated.

In all our experiments we used a maximum recursion depth of 3 for each cp-task.

The generation of each cp-task τ_k is performed as follows:

- the WCET of each node $v_{k,j}$ is uniformly selected as a positive integer $C_{k,j}$ in the interval $[1, 100]$;
- then, L_k and W_k are computed;
- the period T_k is uniformly selected as an integer in the interval $[L_k, W_k/\beta]$, where $\beta \leq 1$ is used to control the minimum cp-task utilization. In particular, the utilization of each cp-task is uniformly distributed in the interval $[\beta, W_k/L_k]$, where the right endpoint of the interval (i.e., the maximum possible utilization) corresponds to the average degree of parallelism of the cp-task;
- the relative deadline D_k is an integer selected with uniform probability in the interval $[L_k, T_k]$.

Whenever a specific utilization is desired, we repeatedly add tasks until the cumulative utilization is achieved, increasing the period of the last task so that the total system utilization matches the desired one.

As the design of a sufficiently general setting for evaluating the performance of DAG-based tasks required a considerable effort, we created a repository [4] where any interested user can freely download our benchmark and use it to test the schedulability performance of conditional/parallel task systems.

B. Evaluation of cp-tasks

This first set of experiments aims at comparing our response-time analysis in a global fixed-priority setting (referred to as RTA-FP) against the only work in the literature that addresses the scheduling of DAG tasks with conditional branches, i.e., [15]. Since that work only proposes a transformation of the DAG task into a synchronous parallel task, without proposing any schedulability test, we adopt the test for synchronous parallel tasks that, to our knowledge, outperforms the others, i.e., the one proposed by Maia et al. [19]. We will refer to this schedulability test as COND-SP.

For this class of experiments, we set $p_{cond} = 0.4$, $p_{par} = 0.4$, $p_{term} = 0.2$, $p_{add} = 0.1$, $n_{cond} = 2$, $n_{par} = 6$, $\beta = 0.1$. When not explicitly specified, we assume a Deadline Monotonic (DM) priority ordering. For each experiment, we generated 500 task-sets for each value on the $x-$axis.

In the first set of experiments, we varied the total system utilization $U_{\mathcal{T}}$ in the range $[0, m]$. Figure 6 reports the number of schedulable task-sets obtained when $m = 4$, which is representative of the general behavior. As can be seen, RTA-FP clearly outperforms COND-SP for any value of $U_{\mathcal{T}}$.

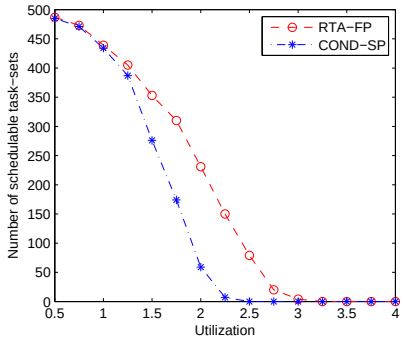

Fig. 6: Evaluation of RTA-FP as a function of $U_{\mathcal{T}}$, with $m = 4$.

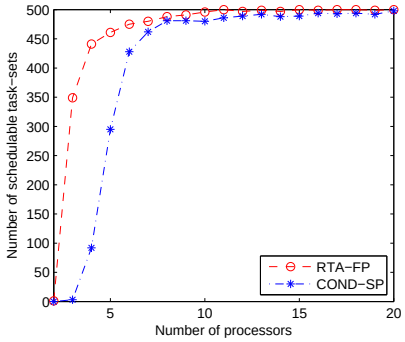

Fig. 7: Evaluation of RTA-FP as a function of m, with $U_{\mathcal{T}} = 2$.

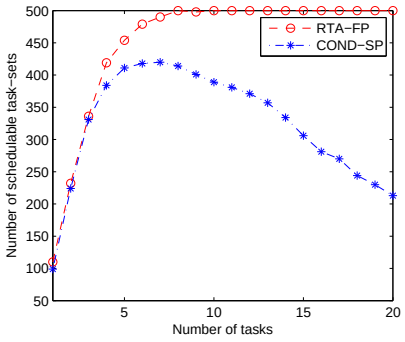

Fig. 8: Evaluation of RTA-FP as a function of n, with $m = 4$ and $U_{\mathcal{T}} = 2$.

In the second set of experiments, we varied the number of cores. Figure 7 illustrates the results for $U_{\mathcal{T}} = 2$. For a low number of processors, RTA-FP substantially outperforms COND-SP, while for higher values of m both tests are able to schedule nearly all the task-sets.

In the third set of experiments, we varied the number of tasks n in the range $[1, 20]$. Since n is now a fixed parameter in each experiment, we computed individual cp-task utilizations using UUnifast [10]. Figure 8 reports the results for $m = 4$ and $U_{\mathcal{T}} = 2$. While the two tests perform comparably for a small value of n, RTA-FP exhibits a substantial improvement over COND-SP when $n \geq 4$. In particular, RTA-FP achieves full schedulability for larger n, conforming to the intuition that scheduling a large number of light tasks is easier than scheduling fewer heavy tasks. Instead, COND-SP achieves its maximum for $n = 7$ and then degrades for higher values of n. This is due to the increased pessimism introduced by the transformation technique in [15] when the number of tasks is higher.

Other experiments when varying the task parallelism and the priority ordering are reported in Appendix B.

Another weakness of COND-SP is that it requires enumerating all the conditional flows of each cp-task, which are exponentially many in the nesting level of the conditional branches. Our approach instead relies on efficient algorithms to deal with conditional branches with pseudo-polynomial complexity. A straightforward consequence is that the running time of COND-SP is often quite prohibitive, while RTA-FP is in general very fast (i.e., in the order of milliseconds).

C. Evaluation of classical DAG tasks

In the following experiments, we considered non-conditional (i.e., classical) DAG tasks. This setting allowed us to evaluate the improvement of our response-time analysis for global EDF (referred to as RTA-EDF) over existing approaches for scheduling sporadic DAG tasks.

The random task generator described above can be used to generate classical DAG tasks by simply setting $p_{cond} = 0$ and requiring that $p_{term} + p_{par} = 1$. In particular, we set: $p_{par} = 0.8$, $p_{term} = 0.2$, $p_{add} = 0.1$, $n_{par} = 6$, $\beta = 0.1$.

We compared our RTA-EDF test against three schedulability tests for global EDF targeting systems of sporadic DAG tasks:

- the test by Baruah [7], which analytically dominates the one in [11];
- the test by Li et al. [17], based on capacity augmentation bound;
- the test by Qamhieh et al. [22] that takes into account the internal structure of the DAG.

Since the test in [17] targets implicit deadline DAG task systems, all the result are reported under that setting, even though the results were consistent also in the general case of constrained deadlines. We decided not to plot the results of the test in [22], because its performance was very poor in all the tested configurations. This is mainly because that work is more concerned with improving the minimum processor speed that guarantees schedulability under global EDF rather than effectively increasing the schedulability performance.

Figure 9 illustrates the number of schedulable task-sets with 8 processors when $U_{\mathcal{T}} \in [0, 8]$. While RTA-EDF is able to schedule nearly all task-sets until $U_{\mathcal{T}} = 2$, the performance of the other approaches degrades significantly at a much lower utilization.

Figure 10 illustrates how RTA-EDF performs when m is varied in the interval $[1, 30]$, with $U_{\mathcal{T}} = 2$. RTA-EDF significantly outperforms the other tests, requiring a much lower number of cores (around 5) to schedule most of the task-sets. The test in [7] typically requires twice that number of cores to schedule most task sets, and it cannot admit any task-set when $m < 7$. The test in [17] behaves even worse, since it cannot admit a large share of the generated task sets even with a very large number of cores. This result indeed reflects the analytical formulation of the test given in [17].

Figure 11 reports the results for $m = 8$ and $U_{\mathcal{T}} = 2$ when varying the number of tasks ($n \in [1, 20]$). Also in this case our approach substantially outperforms the others for any value of n. The test in [7] reaches almost a constant trend for high values of n. Instead, the one in [17] is favorably impacted by increasing n, since, by keeping the total utilization constant, the individual critical path lengths are reduced, which is beneficial for the outcome of the test.

This class of experiments clearly shows that our approach is able to significantly tighten the schedulability of non-conditional DAG task systems as well, widening the effectiveness of our schedulability analysis beyond conditional task structures.

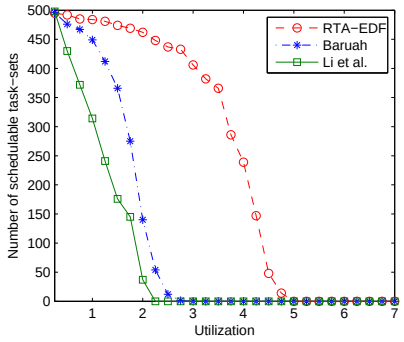

Fig. 9: Evaluation of RTA-EDF as a function of $U_{\mathcal{T}}$, with $m = 8$.

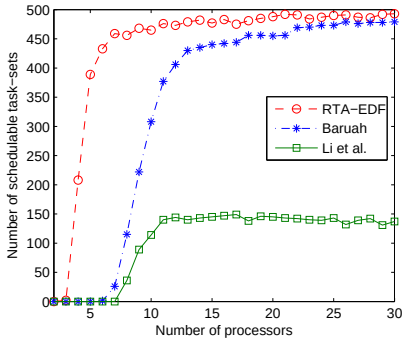

Fig. 10: Evaluation of RTA-EDF as a function of m, with $U_{\mathcal{T}} = 2$.

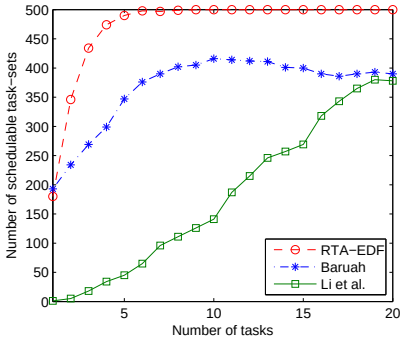

Fig. 11: Evaluation of RTA-EDF as a function of n, with $m = 8$ and $U_{\mathcal{T}} = 2$.

VIII. Conclusions

As multi-core systems become pervasive architectures, future programming practices are likely to be characterized by parallel constructs interleaved with conditional control flow executions. In this paper, we proposed the cp-task model as a generalization of the classic sporadic DAG task model, integrating conditional structures within the task model. This allows improving the information that can be used by the schedulability analysis to derive a tighter estimation of the interfering contributions, by discriminating their level of parallelism depending on the conditional path undertaken. We characterized the topological structure of a cp-task graph, specifying which connections are allowed between conditional and non-conditional nodes. Then, a schedulability analysis has been derived to compute a safe upper-bound on the response-time of each task in pseudo-polynomial time. Beside its reduced complexity, such an analysis has the advantage of requiring only two parameters to characterize the complex structure of the conditional graph of each task: the worst-case workload and the length of the longest path. Algorithms are proposed to derive these parameters from the DAG structure in polynomial time.

Experiments among randomly generated cp-task workloads clearly show that the proposed approach does not only improve over a previously proposed solution for conditional DAG tasks, but can also be used to significantly tighten the schedulability analysis of classic (non-conditional) sporadic DAG task systems.

Acknowledgement

This work has been supported in part by the European Community under the JUNIPER project (FP7-ICT-2011.4.4), grant agreement n. 318763, and the P-SOCRATES project (FP7/2007-2013), grant agreement n. 611016.

References

[1] *Kalray*. http://www.kalrayinc.com/.

[2] *Keystone*. http://www.keyelco.com/.

[3] *Parallela*. http://www.parallela.org/.

[4] *A MATLAB® implementation of schedulability tests for conditional and parallel tasks*. http://retis.sssup.it/∼al.melani/downloads/cptasks.zip, 2015.

[5] B. Andersson and D. de Niz. Analyzing global-edf for multiprocessor scheduling of parallel tasks. In *16th International Conference on Principles of Distributed Systems (OPODIS 2012), Rome, Italy, December 18-20, 2012*, pages 16–30, 2012.

[6] P. Axer, S. Quinton, M. Neukirchner, R. Ernst, B. Döbel, and H. Härtig. Response-time analysis of parallel fork-join workloads with real-time constraints. In *25th Euromicro Conference on Real-Time Systems (ECRTS 2013), Paris, France*, pages 215–224, July 9-12, 2013.

[7] S. Baruah. Improved multiprocessor global schedulability analysis of sporadic DAG task systems. In *26th Euromicro Conference on Real-Time Systems (ECRTS 2014)*, Madrid, Spain, July 8-11, 2014.

[8] S. Baruah, V. Bonifaci, A. Marchetti-Spaccamela, L. Stougie, and A. Wiese. A generalized parallel task model for recurrent real-time processes. In *33rd IEEE Real-Time Systems Symposium (RTSS 2012)*, San Juan, Puerto Rico, December 4-7, 2012.

[9] M. Bertogna and M. Cirinei. Response-time analysis for globally scheduled symmetric multiprocessor platforms. In *28th IEEE Real-Time Systems Symposium (RTSS 2007), Tucson, Arizona, USA*, December 3-6, 2007.

[10] E. Bini and G. C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1-2):129–154, 2005.

[11] V. Bonifaci, A. Marchetti-Spaccamela, S. Stiller, and A. Wiese. Feasibility analysis in the sporadic DAG model. In *25th Euromicro Conference on Real-Time Systems (ECRTS 2013)*, Paris, France, July 9-12, 2013.

[12] A. Burns and S. K. Baruah. Sustainability in real-time scheduling. *Journal of Computing Science and Engineering*, 2(1):74–97, 2008.

[13] H. S. Chwa, J. Lee, K.-M. Phan, A. Easwaran, and I. Shin. Global EDF schedulability analysis for synchronous parallel tasks on multicore platforms. In *25nd Euromicro Conference on Real-Time Systems (ECRTS 2013)*, Paris, France, July 9-12, 2013.

[14] S. Dasgupta, C. H. Papadimitriou, and U. Vazirani. *Algorithms*. McGraw-Hill, Inc., 2006.

[15] J. C. Fonseca, V. Nélis, G. Raravi, and L. M. Pinho. A multi-DAG model for real-time parallel applications with conditional execution. In *30th ACM/SIGAPP Symposium On Applied Computing (SAC 2015)*, Salamanca, Spain, April 13-17, 2015.

[16] K. Lakshmanan, S. Kato, and R. Rajkumar. Scheduling parallel real-time tasks on multi-core processors. In *31st IEEE Real-Time Systems Symposium*, San Diego, CA, November 30 - December 3, 2010. IEEE.

[17] J. Li, K. Agrawal, C. Lu, and C. Gill. Analysis of global EDF for parallel real-time tasks. In *25th Euromicro Conference on Real-Time Systems (ECRTS 2013)*, Paris, France, July 9-12, 2013.

[18] J. Li, J. Chen, K. Agrawal, C. Lu, C. D. Gill, and A. Saifullah. Analysis of federated and global scheduling for parallel real-time tasks. In *26th Euromicro Conference on Real-Time Systems (ECRTS 2014), Madrid, Spain*, July 8-11, 2014.

[19] C. Maia, M. Bertogna, L. Nogueira, and L. M. Pinho. Response-time analysis of synchronous parallel tasks in multiprocessor systems. In *22nd International Conference on Real-Time Networks and Systems (RTNS 2014)*, Versailles, France, October 8-10, 2014.

[20] G. Nelissen, V. Berten, J. Goossens, and D. Milojevic. Techniques optimizing the number of processors to schedule multi-threaded tasks. In *24th Euromicro Conference on Real-Time Systems (ECRTS 2012), Pisa, Italy*, July 11-13, 2012.

[21] B. Peng, N. Fisher, and M. Bertogna. Explicit preemption placement for real-time conditional code. In *26th Euromicro Conference on Real-Time Systems (ECRTS 2014)*, Madrid, Spain, July 8-11, 2014.

[22] M. Qamhieh, F. Fauberteau, L. George, and S. Midonnet. Global EDF scheduling of directed acyclic graphs on multiprocessor systems. In *21st International Conference on Real-Time Networks and Systems (RTNS 2013)*, Sophia Antipolis, France, October 16-18, 2013.

[23] A. Saifullah, K. Agrawal, C. Lu, and C. D. Gill. Multi-core real-time scheduling for generalized parallel task models. In *32nd IEEE Real-Time Systems Symposium (RTSS 2011)*, Vienna, Austria, November 29 - December 2, 2011.

[24] A. Saifullah, D. Ferry, J. Li, K. Agrawal, C. Lu, and C. D. Gill. Parallel real-time scheduling of DAGs. *IEEE Trans. Parallel Distrib. Syst.*, 25(12):3242–3252, 2014.

APPENDIX

A. Interference vs. makespan

To better clarify the theoretical insights behind the response-time analysis presented in this paper, we hereafter discuss the relation between the different terms of Equation (3).

$$R_k = \text{len}(\lambda_k^*) + \frac{1}{m} I_{k,k} + \frac{1}{m} \sum_{\tau_i \in \mathcal{T}, i \neq k} I_{i,k}.$$

While the meaning of the last term in the equation is clearly connected to the interference of the other cp-tasks τ_i on the considered task τ_k, one may think that the sum of the first two terms $(\text{len}(\lambda_k^*) + \frac{1}{m} I_{k,k})$ is equivalent to the worst-case response-time of τ_k when it executes in isolation on the multi-core system (i.e., the so-called makespan of τ_k). However, this is not true. Consider an example where a cp-task τ_k has only one if-then-else statement. When the "if" part is executed, the task executes one sub-task of length 10. Otherwise, the task executes two parallel sub-tasks of length 6 each. When τ_k executes in isolation on a two-core platform, the makespan is clearly given by the "if" branch, i.e., 10. When instead τ_k can be interfered by one job of a task τ_i which executes a single sub-task of length 6, the worst-case response time of τ_k is found when it executes the "else" branch, leading to a response time of 12. The share of the response time due to the term $\text{len}(\lambda_k^*) + \frac{1}{m} I_{k,k}$ in Equation (3) is $6 + (1/2) \cdot 6 = 9$, which is strictly smaller than the makespan. Note that $\text{len}(\lambda_k^*) + \frac{1}{m} I_{k,k}$ does not even represent a valid lower bound on the makespan. This can be seen by replacing the "if" branch in the above example with a shorter subtask of length 8, giving a makespan of 8. For this reason, one cannot replace the term $\text{len}(\lambda_k^*) + \frac{1}{m} I_{k,k}$ in Equation (4) with the makespan of τ_k.

The righthand side of Equation (4) $(L_k + \frac{1}{m}(W_k - L_k))$ and the refined bound computed by Algorithm 2 have been therefore introduced to upper-bound the term $\text{len}(\lambda_k^*) + \frac{1}{m} I_{k,k}$.

However, interestingly, these two quantities do also represent valid upper-bounds on the makespan of τ_k, so that they can be used to bound the response time of a cp-task executing in isolation. The proof is identical to the proofs of the presented bounds included in the paper, considering only the interference due to the task itself, i.e., the intra-task interference.

B. Further simulation results

Task parallelism: In this experiment we varied the degree of connectivity of the tasks, by acting on the probability p_{add} and keeping the other parameters (i.e., $U_{\mathcal{T}}$, n and m) constant. However, we do not recognize any particular trend, i.e., the schedulability ratio remains almost constant for all possible values of p_{add}, hence we do not report the corresponding plots. Additional experiments have been performed to vary the composition of the cp-tasks, by changing p_{cond} with respect to p_{par} while keeping their sum constant. Again, no particular trend has been identified. These results can be explained considering that the computation of the worst-case workload, which plays a fundamental role in the computation of the interference produced by each cp-task, is not very much influenced by its degree of parallelism (see Algorithm 1). Moreover, also the composition of tasks in terms of conditional/parallel branches does not significantly affect the schedulability performance, since the worst-case workload, which already abstracts from the different conditional flows, impacts the utilization of each task-set and its schedulability performance.

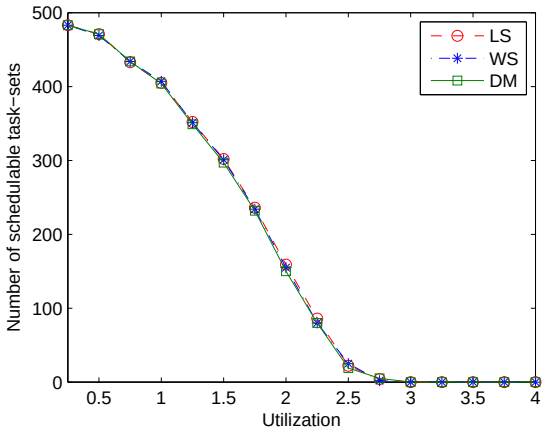

Fig. 12: Evaluation of RTA-FP as a function of $U_{\mathcal{T}}$ with different priority orderings and $m = 4$.

Priority ordering: The performance of our RTA-FP approach has been tested under three distinct priority orderings: i) *Deadline Monotonic* (DM); ii) *Worst-Case Workload Slack* (WS): the tasks are ordered by increasing slack between relative deadline and worst-case workload over m processors (i.e., $D_i - W_i/m$); iii) *Critical Path Length Slack* (LS): the tasks are ordered by increasing slack between relative deadline and critical path length (i.e., $D_i - L_i$). In this experiment, we generated more task-sets (i.e., 5000 for each value on the x-axis), since the running time of our RTA-FP is very fast (i.e., in the order of milliseconds). The results plotted in Figure 12 have been obtained by fixing $m = 4$, and varying the system utilization between 0 and 4. The figure shows that the three different orderings perform comparably, but LS is slightly superior to the other two. This result suggests to further analyze the problem of priority assignment in parallel task systems, where DM might not always be the best choice as it is in the sequential case. However, a thorough discussion on how to assign priorities under the cp-task model is out of the scope of this paper and is left to future investigation.