CISTER

Research Centre in
Real-Time & Embedded
Computing Systems

Journal Paper

Schedulability Analysis of DAG Tasks with Arbitrary Deadlines under Global Fixed-Priority Scheduling

José Fonseca

Geoffrey Nelissen*

Vincent Nélis

*CISTER Research Centre

CISTER-TR-190107

2019

Schedulability Analysis of DAG Tasks with Arbitrary Deadlines under Global Fixed-Priority Scheduling

José Fonseca, Geoffrey Nelissen*, Vincent Nélis

*CISTER Research Centre

Polytechnic Institute of Porto (ISEP P.Porto)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8321159

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

https://www.cister-labs.pt

Abstract

One of the major sources of pessimism in the response time analysis (RTA) of globally scheduled real-time tasks is the computation of an upper-bound on the inter-task interference. This problem is further exacerbated when intra-task parallelism is permitted because of the complex internal structure of parallel tasks. This paper considers the global fixed-priority (G-FP) scheduling of sporadic real-time tasks when each task is modeled by a directed acyclic graph (DAG) of concurrent subtasks. We present a RTA based on the concept of problem window, a technique that has been extensively used to study the schedulability of sequential task in multiprocessor systems. The problem window approach of RTA usually categorizes interfering jobs in three different groups: carry-in, carry-out and body jobs. In this paper, we propose two novel techniques to derive less pessimistic upper-bounds on the workload produced by the carry-in and carry-out jobs of the interfering tasks. Those new bounds take into account the precedence constraints between subtasks pertaining to the same DAG. We show that with this new characterization of the carry-in and carry-out workload, the proposed schedulability test offers significant improvements on the schedulability of DAG tasks for randomly generated task sets in comparison to state-of-the-art techniques. In fact, we show that, while the state-of-art analysis does not scale with an increasing number of processors when tasks have constrained deadlines, the results of our analysis are barely impacted by the processor count in both the constrained and the arbitrary deadline case.

Real-Time Systems
https://doi.org/10.1007/s11241-018-09325-5

Schedulability analysis of DAG tasks with arbitrary deadlines under global fixed-priority scheduling

José Fonseca[1] · **Geoffrey Nelissen**[1] · **Vincent Nélis**[1]

Abstract

One of the major sources of pessimism in the response time analysis (RTA) of globally scheduled real-time tasks is the computation of an upper-bound on the inter-task interference. This problem is further exacerbated when intra-task parallelism is permitted because of the complex internal structure of parallel tasks. This paper considers the global fixed-priority (G-FP) scheduling of sporadic real-time tasks when each task is modeled by a directed acyclic graph (DAG) of concurrent subtasks. We present a RTA based on the concept of problem window, a technique that has been extensively used to study the schedulability of sequential task in multiprocessor systems. The problem window approach of RTA usually categorizes interfering jobs in three different groups: carry-in, carry-out and body jobs. In this paper, we propose two novel techniques to derive less pessimistic upper-bounds on the workload produced by the carry-in and carry-out jobs of the interfering tasks. Those new bounds take into account the precedence constraints between subtasks pertaining to the same DAG. We show that with this new characterization of the carry-in and carry-out workload, the proposed schedulability test offers significant improvements on the schedulability of DAG tasks for randomly generated task sets in comparison to state-of-the-art techniques. In fact, we show that, while the state-of-art analysis does not scale with an increasing number of processors when tasks have constrained deadlines, the results of our analysis are barely impacted by the processor count in both the constrained and the arbitrary deadline case.

Keywords Parallel tasks · DAG scheduling · Response time analysis · Multiprocessor systems · Real-time systems

✉ José Fonseca
jcnfo@isep.ipp.pt

Geoffrey Nelissen
grrpn@isep.ipp.pt

Vincent Nélis
nelis@isep.ipp.pt

[1] CISTER/INESC-TEC, Instituto Superior de Engenharia do Porto, 4249-015 Porto, Portugal

Published online: 16 February 2019

1 Introduction

Few years ago, there was a neat and clear frontier separating the real-time embedded domain from the high-performance computing domain. Nowadays, many modern applications (e.g., intelligent transportation systems and autonomous driving) share requirements from both worlds: they are subject to strong timing constraints and have high computational demands. In order to cope with such orthogonal requirements, we have witnessed a strong push towards the adoption of parallel programming paradigms and multi-/many-core embedded architectures. Parallel programming models, such as OpenMP (Board 2013), enable both inter- and intra-task parallelism in the systems, thus offering opportunities for a more efficient exploitation of the immense processing power that is today at the industry's disposal.

For the real-time research community, the analysis of the worst-case timing behavior of parallel systems requires a detailed representation of the intrinsic parallelism within the application as well as a complete picture of the precedence constraints that it imposes on its parallel activities. These new challenges have been progressively tackled as shown by the different parallel task models and respective schedulability analysis recently proposed in the literature (Lakshmanan et al. 2010; Saifullah et al. 2011; Chwa et al. 2013; Maia et al. 2014; Baruah et al. 2012; Bonifaci et al. 2013; Li et al. 2013; Baruah 2014; Li et al. 2014; Fonseca et al. 2016; Melani et al. 2015; Baruah et al. 2015).

In this paper, we study the sporadic DAG task model introduced in Baruah et al. (2012) under global fixed-priority (G-FP) scheduling. In this model, each task is characterized by a directed acyclic graph (DAG). The nodes of the graph represent sequential computation units (e.g., openMP tasks) and the edges define precedence constraints between the execution of nodes. Nodes that are not directly or transitively connected with each other in the graph may execute in parallel, otherwise they must follow the sequential order given by the DAG structure.

A key challenge in the response time analysis (RTA) of globally scheduled multiprocessor task systems is to compute an upper-bound on the interference that tasks generate on each other. The complexity of computing such inter-task interference bound is exacerbated for parallel tasks, DAGs in particular, due to their complex and irregular internal structure. To the best of our knowledge, the work proposed by Melani et al. (2015) represents the first attempt at analyzing the schedulability of a set of sporadic DAG tasks with a general G-FP scheduling policy through a RTA approach. Their RTA is based on the concept of problem window developed originally by Baker (2003). This technique consists in estimating the maximum interfering workload produced by a higher priority task in a time interval of arbitrary length. While the work in Melani et al. (2015) indeed succeeded in upper-bounding the interfering workload generated by DAG tasks, it does so by considering that every job in the problem window is a compact block of execution which uniformly occupies all the available processors until its completion.

Since most DAGs exhibit different degrees of parallelism throughout their execution and do not necessarily require to constantly access all processors, such abstraction leads to a significant overestimation of the inter-task interference. This extra level of pessimism in the schedulability analysis is evident in the experimental results reported

Table 1 Performance of the schedulability test proposed by Melani et al. (2015)

Schedulability ratio (%)	94	63	49	32	24	16	14	10
Number of cores	2	4	6	8	10	12	14	16

in Table 1 (more details about the system configuration are deferred to Sect. 10). Table 1 shows the percentage of task sets that are deemed schedulable by the schedulability test proposed in Melani et al. (2015) when increasing the number of available cores but keeping the platform utilization fixed at 70% and the number of tasks proportional to the number of cores. The steady schedulability performance deterioration visible in Table 1 for the aforementioned test is counter-intuitive, as one would expect at least a constant schedulability ratio when the parallelism of the platform is increased and the average task utilization remains unchanged. Motivated by these observations, this paper proposes techniques to derive improved bounds on the inter-task interference by exploiting the knowledge of the precedence constraints in the internal structure of the DAGs. As reported in the experimental section of this paper, the proposed technique improves the ratio of task sets deemed schedulable and attenuate strongly the counter-intuitive deterioration of the analysis performance with the increasing number of cores.

1.1 Contributions and paper organization

In this paper, we study the schedulability of a set of sporadic DAG tasks under G-FP scheduling. We present two novel techniques that exploit the internal structure of the DAGs in order to derive improved upper-bounds on the worst-case workload that each higher priority task carries into the problem window of the analyzed task. We then identify the scenario that maximizes the combined interference contributions of both the carry-in and carry-out jobs, allowing us to use the new upper-bounds to refine traditional schedulability analysis methods. Experimental results show that the proposed schedulability test not only dominates the state-of-the-art analysis (Melani et al. 2015) but it is also robust to multiprocessor systems with larger number of cores. The analysis is derived for systems composed of both constrained and arbitrary deadline tasks. Substantial schedulability improvements are attained even for the general case.

The remainder of this paper is organized as follows. Next section provides a concise review of the related work. In Sect. 3 we formally define the sporadic DAG model. Sect. 4 describes briefly the RTA presented in Melani et al. (2015), while Sect. 5 introduces the proposed worst-case scenario for the interfering workload of the higher priority tasks. In Sects. 6 and 7 we present how to upper-bound the worst-case carry-in and carry-out workloads, which we then use to derive the schedulability analysis for constrained deadline tasks in Sect. 8. Section 9 extends the analysis of Sect. 8 to the more general case of systems composed of arbitrary deadline tasks. Finally, Sect. 10 reports our experimental results, right before we draw the conclusions in Sect. 11.

2 Related work

The real-time community has been devoting significant efforts to the problem of scheduling parallel tasks atop multiprocessor platforms. Parallel task models and respective schedulability tests have been proposed to cope with the different forms of task parallelism introduced by widely used parallel programming models. Imposing the most restrictions, the fork-join model (Lakshmanan et al. 2010) characterizes a task as an interleaved sequence of sequential and parallel segments, where the release of each segment is constrained by the completion of its predecessors. A common assumption is that every parallel segment contains the same number of subtasks, which cannot exceed the number of cores in the platform. The synchronous parallel model (Saifullah et al. 2011; Andersson and de Niz 2012; Nelissen et al. 2012; Chwa et al. 2013; Maia et al. 2014) extends the fork-join model by allowing consecutive parallel segments with an arbitrary number of subtasks. Nonetheless, synchronization is still enforced at every segment's boundary, meaning that a subtask cannot start executing until all the subtasks of the previous segment have completed.

A more flexible and general parallel structure is captured by the DAG model (Baruah et al. 2012) considered in this paper, where a task is instead represented by a directed acyclic graph. Nodes represent subtasks to be sequentially executed and edges define precedence constraints between nodes. According to this model, a subtask becomes ready for execution as soon as all its precedences constraints are satisfied, and unconnected subtasks may execute in parallel. Most existing work on the DAG model addresses global earliest deadline first (G-EDF) scheduling, with (Qamhieh et al. 2013; Saifullah et al. 2013, 2014) or without decomposition[1] (Baruah et al. 2012; Bonifaci et al. 2013; Li et al. 2013, 2014; Baruah 2014; Parri et al. 2015).

Researchers have also studied partitioned scheduling (Fonseca et al. 2016), where each subtask is statically assigned to a single processor and therefore cannot migrate. Yet multiple subtasks of the same DAG may still execute on different cores. On the other hand, federated scheduling (Li et al. 2014; Jiang et al. 2017) assigns each heavy task (i.e., a task with an execution workload larger than their deadline) to a set of dedicated processors, whereas light tasks (i.e., those that have a workload smaller than or equal to their deadline) are forced to execute sequentially on the remaining processors.

G-FP scheduling has been considered for DAG tasks with arbitrary deadlines, with Bonifaci et al. (2013) proving a resource augmentation bound of $3 - 1/m$ under a global deadline monotonic (G-DM) policy, whereas Parri et al. (2015) proposed a RTA for G-DM that accounts for the interference experienced by each subtask instead of each task. According to the authors (Parri et al. 2015), the analysis proposed by Parri et. al. is essentially tailored for arbitrary deadline tasks.

Recently, the DAG model has been extended to support conditional statements, allowing a parallel task to experience different flows of execution depending on input and state variables (Fonseca et al. 2015; Melani et al. 2015, 2017; Baruah et al. 2015). As a result, different instances of the same DAG may produce different parallel struc-

[1] The "decomposition" process consists in assigning independent release offsets and virtual deadlines to each subtask in a DAG. Different subtasks may then be scheduled as independent sequential tasks even if they belong to the same DAG.

tures during their execution. We particularly highlight the RTA presented in Melani et al. (2015, 2017) since it addresses G-FP scheduling as it is also the case in this paper. The RTA presented in Melani et al. (2015, 2017) is effective for both conditional and non-conditional DAG tasks. In this paper, we restrict ourselves to the non-conditional case.

3 Model

We consider a set of n sporadic real-time tasks $\tau = \{\tau_1, \ldots, \tau_n\}$ to be globally scheduled by a preemptive fixed-priority algorithm on a platform composed of m unit-speed processors. We assume that priorities are per-task and that task τ_i has higher priority than τ_k if $i < k$. Each task τ_i is characterized by a 3-tuple (G_i, D_i, T_i) with the following interpretation. Task τ_i is a recurrent process that releases a (potentially) infinite sequence of *jobs*, with the first job released at any time during the system execution and subsequent jobs released at least T_i time units apart. Every job released by τ_i has to complete its execution within D_i time units from its release. In this paper, we first consider the special case where τ is comprised of constrained deadline tasks, i.e., $D_i \leq T_i, \forall i$. Then, in Sect. 9, we consider the general case where tasks in τ may have arbitrary deadlines, i.e., smaller than, equal to or larger than their minimum inter-arrival time T_i.

Each job of task τ_i is modeled by a DAG $G_i = (V_i, E_i)$, where $V_i = \{v_{i,1}, \ldots, v_{i,n_i}\}$ is a set of n_i nodes and $E_i \subseteq (V_i \times V_i)$ is a set of directed edges connecting any two nodes. Each node $v_{i,j} \in V_i$ represents a computational unit (referred to as *subtask*) that must execute sequentially. A subtask $v_{i,j}$ has a worst-case execution time (WCET) denoted by $C_{i,j}$. Each directed edge $(v_{i,a}, v_{i,b}) \in E_i$ denotes a precedence constraint between the subtasks $v_{i,a}$ and $v_{i,b}$, meaning that subtask $v_{i,b}$ cannot execute before subtask $v_{i,a}$ has completed its execution. In this case, $v_{i,b}$ is called a *successor* of $v_{i,a}$, whereas $v_{i,a}$ is called a *predecessor* of $v_{i,b}$. A subtask is then said to be *ready* if and only if all of its predecessors have finished their execution. For simplicity, we will omit the subscript i when referring to the subtasks of task τ_i if there is no possible confusion. A subtask with no incoming (resp., outgoing) edges is referred to as a *source* (resp., a *sink*) of the DAG. Without loss of generality, we assume that each DAG has a single source v_1 and a single sink v_{n_i}. Note that any DAG with multiple sinks/sources complies with this requirement, simply by adding a dummy source/sink with zero WCET to the DAG, with edges from/to all the previous sources/sinks.

For each subtask $v_j \in V_i$, its set of direct predecessors is given by $pred(v_j)$, while $succ(v_j)$ returns its set of direct successors. Formally, $pred(v_j) = \{v_k \in V_i \mid (v_k, v_j) \in E_i\}$ and $succ(v_j) = \{v_k \in V_i \mid (v_j, v_k) \in E_i\}$. Furthermore, $ances(v_j)$ denotes the set of ancestors of v_j, defined as the set of subtasks that are *either directly or transitively* predecessors of v_j. Analogously, we denote by $desce(v_j)$ the descendants of v_j. Formally, $ances(v_j) = \{v_k \in V_i \mid v_k \in pred(v_j) \vee (\exists v_\ell, v_\ell \in pred(v_j) \wedge v_k \in ances(v_\ell))\}$ and $desce(v_j) = \{v_k \in V_i \mid v_k \in succ(v_j) \vee (\exists v_\ell, v_\ell \in succ(v_j) \wedge v_k \in desce(v_\ell))\}$. Any two subtasks that are not ancestors/descendants of each other are said to be *concurrent*. Concurrent subtasks may execute in parallel.

Definition 1 (*Path*) For a given task τ_i , a path $\lambda = (v_1, \ldots, v_{n_i})$ is a sequence of subtasks $v_j \in V_i$ such that v_1 is the source of G_i , v_{n_i} is the sink of G_i , and $\forall v_j \in \lambda \backslash \{v_{n_i}\}, (v_j, v_{j+1}) \in E_i$.

Informally, a path λ is a sequence of subtasks from the source to the sink in which there is a precedence constraint between any two adjacent subtasks in λ. Thus, there is no concurrency between the subtasks that belong to a same path. The length of a path λ, denoted $len(\lambda)$, is the sum of the WCET of all its subtasks, i.e., $len(\lambda) = \sum_{\forall v_j \in \lambda} C_j$.

Definition 2 (*Length of a task*) The length L_i of a task τ_i is the length of its longest path.

Definition 3 (*Critical path*) A path of τ_i that has a length L_i is a critical path of τ_i.

Note that when the number of cores m is greater than the maximum possible parallelism of τ_i, the length L_i represents the worst-case response time (WCRT) of τ_i in isolation (also known as the *makespan* of the graph). Therefore, an obvious necessary condition for the feasibility of τ_i is $L_i \leq D_i$.

Definition 4 (*Workload*) The workload W_i of a task τ_i is the sum of the WCET of all its subtasks, i.e. $W_i = \sum_{j=1}^{n_i} C_j$.

Finally, we prove the following property on τ_i's execution and its critical path.

Lemma 1 *At most* $W_i - \max\{0,\ L_i - \ell\}$ *units of workload can be executed by a job of* τ_i *in a window of length* ℓ.

Proof By Def. 1, all subtasks in a critical path have precedence constraints and must therefore execute sequentially. In the worst-case, a job of τ_i cannot finish its execution within a time window of length shorter than L_i independently of the number of cores, since the length of a critical path is L_i by Def. 3. Since each DAG has at least one critical path, ℓ time units after its release, a job of τ_i still has to execute for at least $\max(0,\ L_i - \ell)$ time units in order to meet the sequential execution requirements of its critical path entirely. Hence, at most $W_i - \max\{0,\ L_i - \ell\}$ units of workload are executed in the interval of length ℓ. □

Corollary 1 *No schedule of* G_i *whose length is shorter than* L_i *can accommodate* W_i *units of workload.*

Note that Lemma 1 is a very coarse and pessimistic bound on the amount of workload executed by a DAG task in an interval of length ℓ. Yet, that property will be useful to prove the correctness of the response time analysis proposed in this paper.

4 Background

In this section, we introduce the concept of interference for DAG tasks. We also summarize the RTA introduced by Melani et al. (2015) as it sets the foundations for

the schedulability analysis proposed in the upcoming sections. Although their work targets a more general task model, known as "conditional DAG model", empirical evaluation in Melani et al. (2015) shows that it is also state-of-the-art for the non-conditional DAG tasks considered in this paper.

A key challenge in the RTA of globally scheduled multiprocessor systems is the computation of the *interference* among tasks. For sequential tasks, the interference exerted on a task τ_k is defined as the cumulative length of all the time intervals in which τ_k is ready but cannot be scheduled on any processor due to the concurrent execution of m higher priority tasks. In order to adapt this definition to the parallel structure of DAG tasks, we introduce the notion of critical chain.

Definition 5 (*Critical chain*) The critical chain λ_k of a DAG task τ_k is the path of τ_k that leads to its worst-case response time R_k, with ties broken arbitrarily.

To determine the worst-case response time of τ_k, we then need to identify such critical chain and compute the maximum possible interference exerted on it. We start by characterizing the interference on a DAG task τ_k.

Definition 6 (*Interference*) The interference I_k on a DAG task τ_k is the cumulative length of all the time intervals in which at least one subtask that belongs to τ_k's critical chain is ready but cannot be scheduled on any processor because all m cores are busy.

Alternatively, the total interference can be expressed as a function of the worst-case interfering workload generated by each task in the system.

Definition 7 (*Interfering workload*) The interfering workload W_k^i imposed by a DAG task τ_i on a DAG task τ_k represents the total workload executed by subtasks of τ_i, while at least one subtask that belongs to τ_k's critical chain is ready but cannot be scheduled on any processor.

Definitions 6 and 7 also allow us to formulate a bound on the worst-case response time of τ_k:

$$R_k \leq len(\lambda_k) + I_k = len(\lambda_k) + \frac{1}{m} \sum_{\forall \tau_i \in \tau} W_k^i \tag{1}$$

Furthermore, under fixed-priority scheduling, a task τ_k cannot suffer interference from lower priority tasks. That is, $W_k^i = 0, \forall i > k$. However, when $i = k$, we have $W_k^i \geq 0$. That is because other subtasks of τ_k that *do not* belong to its critical chain may also delay the completion of τ_k itself. This phenomenon peculiar to parallel tasks is called *self-interference*.

Unfortunately, deriving concrete values for either the overall term I_k or the individual terms W_k^i is computational intractable for non-trivial task sets, otherwise a schedulability test would easily follow from Eq. (1). For this reason, an established workaround is to bound the total worst-case interfering workload by analyzing the maximum possible workload that can be produced by each interfering task during the worst-case instance of τ_k. In the following, we present the upper-bounds derived

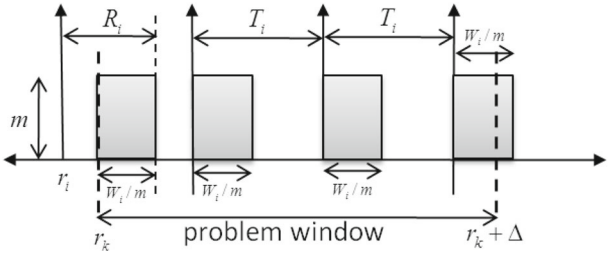


Fig. 1 Worst-case interfering workload of a higher priority task τ_i, as considered in Melani et al. (2015)

in Melani et al. (2015) for both the self-interference ($i = k$) and inter-task interference ($i < k$) components in the context of G-FP scheduling, as well as the resulting response time equation.

Regarding the self-interference, in a constrained deadline setting two jobs of a same task τ_k cannot interfere with each other. That is because one job must finish before the next one is released, otherwise τ_k would fail to meet its deadline and the system would immediately be deemed unschedulable. Therefore, the self-interfering workload is independent of the response time of τ_k. Furthermore, due to the absence of priorities at the subtask-level, every subtask *that is not part of* τ_k*'s critical chain* may potentially contribute to the overall response time of τ_k and thus to its self-interfering workload W_k^k.

Let M_k denote the contribution of DAG task τ_k to its own response time, i.e., $M_k \stackrel{\text{def}}{=} len(\lambda_k) + W_k^k/m$. It was proven in Melani et al. (2015) that, for *constrained deadline* tasks, an upper-bound on M_k is given by

$$M_k \leq L_k + \frac{1}{m}(W_k - L_k) \tag{2}$$

That is, the self-interfering workload is upper-bounded by $W_k^k \leq W_k - L_k$ (i.e., the remaining workload of τ_k after excluding the length of its critical path). Importantly, Eq. (2) not only provides a bound on the maximum makespan of τ_k (i.e., its WCRT in isolation) but also ensures that the critical chain λ_k can be safely replaced by a critical path of τ_k in the response time analysis, as long as such critical path is subject to at least the same amount of inter-task interference. Hence, we hereinafter restrict our attentions to a single critical path of τ_k, fixed arbitrarily.

Contrary to the self-interference, the amount of inter-task interfering workload depends on the length of the time interval that we consider. The longer the time interval, the more workload can be generated by the higher priority tasks and thus the larger is the inter-task interference on the analyzed task τ_k. For a time window of length Δ starting at τ_k's release, the contribution of a higher priority task τ_i to the inter-task interfering workload W_k^i is divided in three portions (see Fig. 1):

1. Carry-in: It accounts for the contribution of jobs of τ_i with release times before the beginning of the problem window (i.e., before τ_k's release at time r_k) and a deadline after the beginning of the problem window, i.e., after r_k. The *carry-in jobs* workload corresponds to the portion of those jobs execution that could not

finish prior to r_k. Note that for constrained deadline systems, if τ_i is schedulable, then τ_i has at most one carry-in job.

2. Body: It takes into account the contribution of all subsequent job releases of τ_i that are fully contained in the window. The workload of each of the *body jobs* to the interfering workload is upper-bounded to its complete execution time W_i.
3. Carry-out: In the related literature, it usually accounts for the contribution of a job of τ_i with release time within the problem window and deadline after the end of the window (i.e., after $r_k + \Delta$). Yet, in this paper we will slightly bend the definition and instead consider that a *carry-out job* is a job that is released within the problem window less than T_i time units before its end (i.e., the carry-out job of τ_i is released at time t such that $(r_k + \Delta - T_i) < t < (r_k + \Delta)$). Note that our definition is compliant with the state-of-the-art definition when tasks have implicit deadlines (i.e., $D_i = T_i$). The interfering workload of the *carry-out job* corresponds to the portion of its execution that actually overlaps with the time interval $[r_k, r_k + \Delta)$.

In Melani et al. (2015), the authors formulated a generic bound on the worst-case workload generated by an interfering task τ_i with constrained deadline within such window of length Δ. This upper-bound, which we state below, relates to the maximum interfering workload imposed by τ_i on task τ_k under analysis by fixing $\Delta = R_k$. Hence, $W_k^i \leq \mathcal{W}_i(R_k)$ where $\mathcal{W}_i(\Delta)$ is defined as follows:

$$\mathcal{W}_i(\Delta) \stackrel{\text{def}}{=} \left\lfloor \frac{\Delta + R_i - W_i/m}{T_i} \right\rfloor W_i + \min\left(W_i, m((\Delta + R_i - W_i/m) \bmod T_i)\right) \tag{3}$$

Notice that Eq. (3) ignores completely the structure of the DAG G_i of τ_i and corresponds to the scenario depicted in Fig. 1. The first term includes both the contributions from the carry-in and body jobs, whereas the second term represents the carry-out component. The interference imposed by τ_i on τ_k within the problem window is maximized when: (1) the carry-in job starts executing at the start of the time window and finishes by its WCRT R_i, (2) all subsequent jobs are released and executed as soon as possible and (3) every job of τ_i is assumed to execute on all the cores during W_i/m time units.

Putting all the pieces together, for a given task τ_k, the schedulability condition $R_k \leq D_k$ relies on a classic iterative RTA. Starting with $R_k = L_k$, an upper-bound on the response time of task τ_k under G-FP scheduling can be derived by a fixed-point iteration on the following expression:

$$R_k = L_k + \frac{1}{m}(W_k - L_k) + \frac{1}{m}\sum_{\forall i<k} \mathcal{W}_i(R_k) \tag{4}$$

5 Rationale

Looking at the RTA described in the previous section, it is obvious that one of the major sources of pessimism in the computation of the WCRT is the computation of

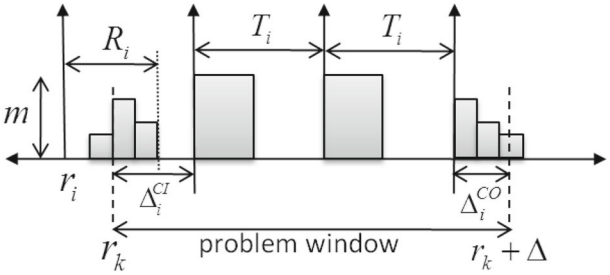


Fig. 2 Worst-case scenario that maximizes the interfering workload released by τ_i in the problem window of τ_k

the inter-task interference within the problem window. This is clear by examining the execution pattern assumed for every job of the tasks τ_i that interferes with the analyzed task τ_k (see Fig. 1). All these jobs are assumed to execute as a big compact block that uniformly occupies the m cores during W_i/m time units. Although this assumption provides a safe upper-bound on the interference that they cause, the upper-bound may be greatly improved by not overlooking the rich internal structure of their DAG. Both the precedence constraints and the number of subtasks in the DAG define the possible shapes that the execution of τ_i entails. In general, wider and uneven shapes limit the amount of workload that effectively enters the problem window. In fact, most DAGs do not exhibit a constant degree of parallelism equal to m throughout their entire execution (as it is assumed in the state-of-the-art analysis). Instead, the maximum workload they may execute in a given time interval is limited by their internal structure. This is illustrated in Fig. 2, where the maximum interfering workload imposed by the carry-in and carry-out jobs of a task τ_i is presented.

This observation is emphasized in the example below.

Example 1 Consider the execution of the task of Fig. 3a on $m = 5$ cores. The maximum parallelism attained by the DAG G_i is equal to 5, when subtasks $\{v2, v3, v4, v5, v6\}$ execute simultaneously. Such concurrent execution can only last for 4 time units. After that, the degree of parallelism drops to 2 as v_7 becomes ready but v_2 has not finished yet. We point out that different execution patterns are possible between the subtasks mentioned so far if we include, for example, interference from higher priority tasks. However, they cannot increase the amount of time during which G_i requires all the available cores. Moreover, both the source v_1 and the sink v_8 cannot execute concurrently with any other subtask of G_i. Therefore, the maximum workload that can be generated by G_i in a window of length 5 is at most 22. Yet, the state-of-the-art analysis presented in Sect. 4 assumes that 25 time units of interfering workload have been generated in a window of length 5.

In this paper, we use the internal structure of each DAG to derive more accurate upper-bounds on their contributions to the carry-in and carry-out interfering workload. Note that, according to this analysis method, the DAG's internal structure does not affect the contribution of the body jobs to the interfering workload since they are fully contained in the problem window. Thus, their exact execution pattern is irrelevant.

Similar to the work in Melani et al. (2015), our analysis of the inter-task interference is based on the notion of a problem window of length Δ. However, as illustrated in Fig. 2, we model more accurately the worst-case scenario by taking into account different execution patterns for the carry-in and carry-out jobs. Therefore, the workload produced by task τ_i is maximized in the problem window $[r_k, r_k + \Delta)$ of τ_k when: (i) every subtask of the body jobs of τ_i executes for its WCET; (ii) the carry-in job released at a time $r_i < r_k$ finishes its execution at time $r_i + R_i$ and executes as much workload as possible as late as possible (to maximize its workload in the problem window); (iii) all subsequent jobs are released T_i time units apart; and (iv) the carry-out job starts its execution as soon as it is released and executes as much workload as possible as early as possible (hence maximizing its workload in the problem window).

Our main problem to solve is the lack of a relative reference point between the release time of the carry-in job of τ_i and the problem window $[r_k, r_k + \Delta)$. More specifically, the value $(r_k - r_i)$ is unknown a priori because, as will be shown later in this paper, the worst-case schedules of the carry-in and carry-out jobs are incomparable. Let Δ_i^{CI} and Δ_i^{CO} denote the length of the carry-in portion and the length of the carry-out portion of τ_i's schedule, respectively. Formally, we have that[2] (see Fig. 2 for visual reference)

$$\Delta_i^{CI} \stackrel{\text{def}}{=} r_i + T_i - r_k \tag{5}$$

$$\Delta_i^{CO} \stackrel{\text{def}}{=} \max\{0,\ (r_k + \Delta) - (r_k + \Delta_i^{CI} + \left\lfloor \frac{\Delta - \Delta_i^{CI}}{T_i} \right\rfloor_0 \times T_i)\} \tag{6}$$

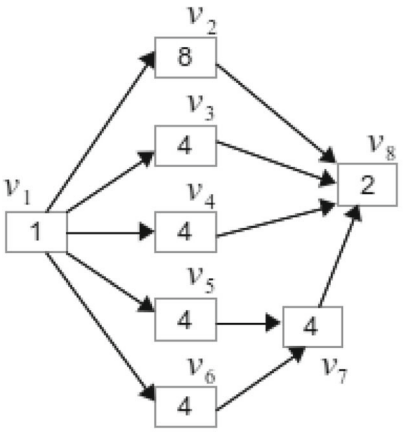
We seek to derive (i) an upper-bound on the interfering workload executed by τ_i's carry-in job as a function of Δ_i^{CI}, (ii) an upper-bound on the interfering workload executed by τ_i's carry-out job as a function of Δ_i^{CO}, and (iii) determine concrete values for Δ_i^{CI} and Δ_i^{CO} such that the interfering workload of τ_i on task τ_k cannot be larger under any possible execution scenario.

To characterize the execution pattern of a carry-in and carry-out job of τ_i, we introduce the notion of *workload distribution*.

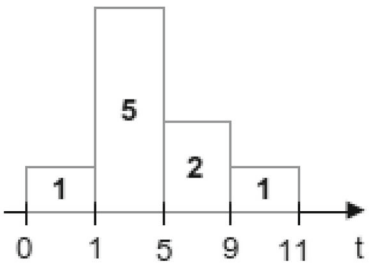
Definition 8 *(workload distribution)* For a given task τ_i and a given schedule S of τ_i's subtasks, the workload distribution $\mathcal{WD}_i^S = [B_1, \ldots, B_\ell]$ describes S as a sequence of consecutive blocks. Each block $B_b \in \mathcal{WD}_i^S$ is a tuple (w_b, h_b) with the interpretation that there are h_b subtasks (height) of G_i executing during w_b time units (width) in S, immediately after the completion of the subtasks that execute in the $(b-1)^{th}$ block.

Note that $\mathcal{WD}_i^S$ does not provide any information about the precedence constraints in the DAG G_i, neither is it required for S to be a valid schedule of G_i. Hence, according to Def. 8, every interfering job of a task τ_i is modeled in Melani et al. (2015) with a workload distribution $\mathcal{WD}_i^S$ that comprises only one block $B_1 = (\frac{W_i}{m}, m)$. In the next two sections, we will derive more accurate workload distributions in order to model

[2] The operator $\lfloor x \rfloor_0 \stackrel{\text{def}}{=} \max\{0,\ \lfloor x \rfloor\}$.



(a) Example of a DAG task. The WCET of each node is given by the number in that node.



(b) Workload distribution $\mathcal{WD}_i^{UCI}$. Numbers in blocks represent their height.

Fig. 3 Example for the carry-in interfering workload

the schedules of τ_i's carry-in and carry-out jobs that maximize their contribution to the interference suffered by a lower priority task τ_k.

6 Carry-in workload

This section presents the analysis to compute the carry-in workload of a higher priority task τ_i in the problem window $[r_k, r_k + \Delta)$ of τ_k. Recall that a carry-in job is a job of τ_i such that its release time r_i is earlier than r_k and its deadline falls after r_k. Therefore, to upper-bound the interfering workload generated by the carry-in job, we need to determine which subtasks of τ_i may execute within the carry-in window $[r_k, r_k + \Delta_i^{CI})$, either fully or partially. Intuitively, to maximize the interfering workload the carry-in job should execute as much workload as possible as late as possible.

For ease of understanding, we will use Fig. 3a as an example task throughout our discussion on the carry-in job.

6.1 Workload distribution of the carry-in job

When the degree of parallelism of the DAG G_i is not constrained by the number of cores (assuming $m = \infty$ for instance), the schedule of G_i that yields the maximum makespan is simply that in which every subtask executes for its WCET. Note that because there are always available cores, each subtask is scheduled as soon as it becomes ready. We call this particular schedule "unrestricted carry-in" (UCI). If f_j denotes the relative completion time of each subtask $v_j \in V_i$ in UCI, then it holds that:

$$f_j = \begin{cases} C_j & \text{if } v_j \text{ is the source} \\ C_j + \max\limits_{v_h \in pred(v_j)} (f_h) & \text{otherwise} \end{cases} \tag{7}$$

Note that the length (makespan) of UCI is given by the completion time f_{n_i} of the sink of G_i and according to Eq. (7), f_{n_i} is equal to the critical path length L_i.

Assuming that the source of τ_i starts executing at a relative time 0, the number of subtasks in UCI that execute at any time $t \in [0, L_i)$ can be computed by the function $AS(t)$ defined as

$$AS(t) = \sum_{v_j \in V_i} actv(v_j, t) \tag{8}$$

where $actv(v_j, t)$ is equal to 1 if v_j is executing at time t and 0 otherwise. That is,

$$actv(v_j, t) = \begin{cases} 1 & \text{if } t \in \left[f_j - C_j, f_j\right) \\ 0 & \text{otherwise} \end{cases} \tag{9}$$

Let F_i be the set of finishing times of the subtasks $v_j \in V_i$ (without duplicates) sorted in non-decreasing order. We build a workload distribution $\mathcal{WD}_i^{UCI}$ modeling the schedule UCI as follows:

- $\mathcal{WD}_i^{UCI}$ has as many blocks as there are elements in F_i;
- The b^{th} block of $\mathcal{WD}_i^{UCI}$ is represented by the tuple $(t_{b+1} - t_b, AS(t_b))$ such that t_b is the b^{th} time instant in the ordered set $\{0\} \cup F_i$.

Built that way, $\mathcal{WD}_i^{UCI}$ models the maximum parallelism of τ_i at any time t assuming that all subtasks execute for their WCET. An example of such workload distribution is depicted in Fig. 3b for the DAG presented in Fig. 3a.

6.2 Upper-bounding the carry-in workload

Based on both the workload distribution $\mathcal{WD}_i^{UCI}$ and the WCRT R_i estimated by Eq. (4), we compute an upper-bound on the interfering workload produced by one carry-in job of τ_i within its carry-in window $[r_k, r_k + \Delta_i^{CI})$. To do so, we push the workload distribution $\mathcal{WD}_i^{UCI}$ as much as possible "to the right". We first align the end of $\mathcal{WD}_i^{UCI}$ with the worst-case completion time of the carry-in job of τ_i. That is, we align the end of $\mathcal{WD}_i^{UCI}$ with the time-instant $r_k + \Delta_i^{CI} - (T_i - R_i)$ (see Fig. 2). This assumes that the carry-in job of τ_i is released at $r_k + \Delta_i^{CI} - T_i$ and completes at most at $r_k + \Delta_i^{CI} - T_i + R_i$.

Since the problem window starts at r_k and the carry-in job must complete by $r_k + \Delta_i^{CI} - (T_i - R_i)$, the part of the carry-in job that effectively interferes with τ_k is given by the subtasks of that job executed in the last $\Delta_i^{CI} - (T_i - R_i)$ time units of its schedule. Therefore, under the schedule UCI, the maxi-

mum interfering workload released by τ_i's carry-in job is upper-bounded by the function[3]:

$$CI_i(\mathcal{WD}_i^{UCI}, \Delta_i^{CI}) = \sum_{b=1}^{|\mathcal{WD}_i^{UCI}|} h_b \times \left[r_i + R_i - \sum_{p=b+1}^{|\mathcal{WD}_i^{UCI}|} w_p \right]_0^{w_b} \quad (10)$$

where $r_i \stackrel{\text{def}}{=} \Delta_i^{CI} - T_i$ is the latest time at which τ_i's carry-in job may be released (assuming that r_k happens at time 0).

Equation (10) returns 0 if Δ_i^{CI} is smaller than $(T_i - R_i)$ (i.e., if the carry-in job of τ_i completes before the beginning of the problem window). Otherwise, it sums the height h_b of the workload distribution $\mathcal{WD}_i^{UCI}$ in its last $\Delta_i^{CI} - (T_i - R_i)$ time units.

Example 2 If $\Delta_i^{CI} = 9$, $T_i = 20$, $R_i = 15$ and $\mathcal{WD}_i^{UCI}$ is given by the workload distribution presented in Fig. 3b, then Eq. (10) sums the height of the blocks in the last $\Delta_i^{CI} - (T_i - R_i) = 4$ time units of $\mathcal{WD}_i^{UCI}$. Hence, it gives us $CI_i(\mathcal{WD}_i^{UCI}, \Delta_i^{CI}) = 6$. If Δ_i^{CI} was equal to 4, then Eq. (10) would return 0 since $\Delta_i^{CI} - (T_i - R_i)$ is then smaller than 0.

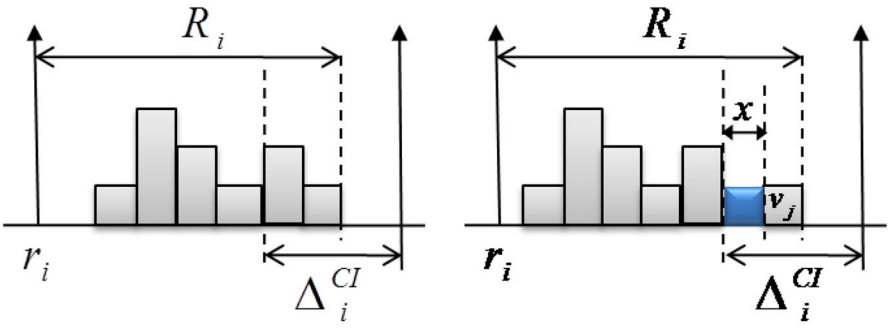
We now prove that the interfering workload executed by the carry-in job of τ_i is upper-bounded by the workload distribution $\mathcal{WD}_i^{UCI}$, when the end of $\mathcal{WD}_i^{UCI}$ is aligned with the time-instant $(r_k + \Delta_i^{CI} - T_i + R_i)$ where R_i is computed by Eq. (4).

The carry-in workload computed by Eq. (10) assumes that (i) all subtasks of τ_i execute for their WCET, (ii) the number of cores does not limit τ_i's parallelism and (iii) the carry-in job of τ_i executes following the workload distribution $\mathcal{WD}_i^{UCI}$ just before its completion time at $(r_k + \Delta_i^{CI} - T_i + R_i)$. We prove in Lemmas 2–4 that those three assumptions maximize the interfering workload of τ_i in the carry-in window.

Lemma 2 *The interfering workload generated by the carry-in job of a higher priority task τ_i is maximized when all its subtasks execute for their WCET.*

Proof If a subtask $v_j \in V_i$ executes for less than its WCET C_j, then either v_j contributes less to the interfering workload (assuming that v_j is executed within the carry-in window), or it may allow its successors (and subsequently its descendants) to be released earlier (note that the release times of subtasks that are not descendant of v_j are not impacted). In the latter case, it may cause those descendants to start executing before (instead of within) the carry-in window and thus reduce the total interfering workload they may generate. Similarly, descendants of v_j that were already starting before the beginning of the carry-in window, may complete before the start of the carry-in window, or earlier within the carry-in window. In both cases, the interfering workload in the carry-in window is reduced. □

[3] $[x]_z^y = max\{min\{x, y\}, z\}$, that is, y and z are an upper-bound and a lower-bound on the value of x, respectively.



(a) Before delaying the critical path. **(b)** After delaying the critical path.

Fig. 4 Interference (blue block) on $\mathcal{WD}_i^{UCI}$ critical path

Lemma 3 *Let R_i be an upper-bound on the worst-case response time of τ_i and let $\mathcal{WD}_i$ be any workload distribution* ***of length*** *L_i representing any possible schedule of τ_i. Assume that $\mathcal{WD}_i$ is aligned to the right with the time-instant $(r_k + \Delta_i^{CI} - T_i + R_i)$. The workload that can be generated by $\mathcal{WD}_i$ in the carry-in window cannot be increased by delaying subtasks in τ_i's critical path.*

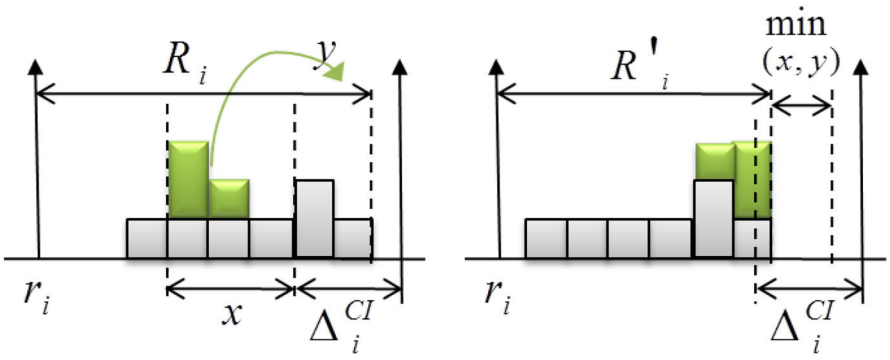
Proof Remember that the length of the workload distribution $\mathcal{WD}_i$ is L_i, i.e., the length of $\mathcal{WD}_i$ is equal to the length of the critical path of τ_i. Therefore, there must be a subtask of each critical path of τ_i[4] executing at any time instant between $(r_k + \Delta_i^{CI} - T_i + R_i - L_i)$ and $(r_k + \Delta_i^{CI} - T_i + R_i)$ (because $\mathcal{WD}_i$ is aligned to the right with $(r_k + \Delta_i^{CI} - T_i + R_i)$). This case is illustrated on Fig. 4a.

Now consider the case where $\mathcal{WD}_i$ is subject to self- and/or higher priority interference such that the execution of at least one subtask v_j of a critical path of τ_i is delayed by x time units.

Postponing the execution of v_j by x time units leads to move both the workload of v_j and its descendants x time units "to the right". Because v_j belongs to a critical path of τ_i, the length of τ_i's carry-in job schedule is increased by x (see Fig. 4b). However, because R_i is assumed to be an upper-bound on τ_i's worst-case response time, τ_i's carry-in job cannot complete later than $(r_k + \Delta_i^{CI} - T_i + R_i)$. Therefore, as visualized in Fig. 4b, it is not the subtask v_j or its descendants that are moved by x time units "to the right", but instead it is the workload executed by predecessors of v_j that is pushed by x time units to the left. Hence, the workload executed by τ_i in the carry-in window $[r_k, r_k + \Delta_i^{CI})$ can only decrease. □

Lemma 4 *Let R_i be the upper-bound on the worst-case response time of τ_i* ***computed by Eq.*** *(4). Aligning $\mathcal{WD}_i^{UCI}$ to the right with the time-instant $(r_k + \Delta_i^{CI} - T_i + R_i)$ gives an upper-bound on the maximum interfering workload that can be generated by τ_i in the carry-in window, independently of the interference imposed on τ_i.*

[4] A task τ_i may have more than one critical path.



(a) Before delaying subtasks not in the critical path.

(b) After delaying subtasks not in the critical path.

Fig. 5 y units of workload (green blocks) of $\mathcal{WD}_i^{UCI}$ are moved in the carry-in window

Proof Remember that the length of $\mathcal{WD}_i^{UCI}$ is L_i. Hence, Lemma 3 proved that the workload generated in the carry-in window cannot increase by interfering with the critical path of τ_i. Therefore, this proof must show that the claim is still true even when the interference exerted on τ_i does not interfere with its critical paths but delays the execution of other subtasks of τ_i.

The proof is by contradiction. Assume that there is a schedule of τ_i such that, by delaying subtasks of τ_i, y extra units of workload of τ_i enter the carry-in window $[r_k, r_k + \Delta_i^{CI})$ comparatively to the workload generated by $\mathcal{WD}_i^{UCI}$ (see Fig. 5a for an illustration of y extra units of workload, colored in green, moved in the carry-in window). By Lemma 3, the delayed subtasks do not belong to any critical path of τ_i and the length of τ_i's schedule is therefore not affected, i.e., it remains equal to L_i.

Let v_j be any of the delayed subtasks and let δ_j be the minimum time for which its execution has to be delayed, in comparison to the schedule based on $\mathcal{WD}_i^{UCI}$, so that v_j enters the carry-in window. Let x be the maximum δ_j over all the delayed subtasks, i.e., $x \stackrel{\text{def}}{=} \max_j\{\delta_j\}$ (see Fig. 5a for an illustration of x). That is, at least one subtask has been delayed by at least x time units to enter the carry-in window.

Since m subtasks are allowed to execute in parallel on m cores and the critical path of τ_i is not delayed, postponing a subtasks by x time units implies that at least $(m-1) \times x$ interfering workload executes in parallel with the critical path to prevent the delayed subtask to execute on any of the m cores. Additionally, note that the y units of shifted workload do not interfere with the critical path either, and hence execute in parallel with the critical path, since by assumption the schedule length is not increased. Therefore, we have at least

$$(m-1) \times x + y$$

units of workload that do not interfere with the critical path but execute in parallel with it instead.

Let R'_i be an upper-bound on *the actual* response time of τ_i's carry-in job under this modified schedule. Since R_i is computed with Eq. (4), and Eq. (4) assumes that

all higher priority jobs and all subtasks that do not belong to the critical path of τ_i interfere with it, R_i' must be smaller than R_i and we have

$$\begin{aligned} R_i' &\leq R_i - \left(\frac{(m-1)\times x + y}{m}\right) \\ &\leq R_i - \left(\frac{m \times y}{m} + \frac{(m-1)\times(x-y)}{m}\right) \\ &\leq R_i - y - \frac{(m-1)\times(x-y)}{m} \end{aligned} \tag{11}$$

We analyse two cases:

- If $y \leq x$, then the last term in (11) is positive and we have $R_i' \leq R_i - y$. Hence the response time of τ_i and thus the length of τ_i's schedule in the carry-in window has been reduced by at least y time units (see Fig. 5b). Since at least one subtask of each critical path of τ_i must execute at each of those time units (because the length of the schedule is L_i), the workload in the carry-in window has decreased by at least y time units. This is in contradiction with the assumption that the workload increased in the carry-in window.
- If $y > x$, then the last term of (11) is negative and we have $R_i' \leq R_i - y - (x-y) = R_i - x$. Hence, τ_i's response time has reduced by at least x time units. Therefore, the subtasks that were delayed by x time units could not enter the carry-in workload since the whole schedule of τ_i is pushed to the left by x time units too (see Fig. 5b). Therefore, it contradicts the assumption that extra workload of τ_i entered the carry-in window by delaying subtasks by x time units.

The two cases above prove the claim. □

Theorem 1 *The interfering workload W_i^{CI} generated by the carry-in job of a higher priority task τ_i in a window of length Δ_i^{CI} is upper-bounded by $CI_i(WD_i^{UCI}, \Delta_i^{CI})$.*

Proof The proof follows directly from Lemmas 2–4.

6.3 Improved carry-in workload

The lemma below presents another upper-bound on the maximum interfering workload that can be generated by a task τ_i in a carry-in window of length Δ_i^{CI}. Since this upper-bound cannot be compared with that given by Eq. (10), Theorem 2 below shall present an improved upper-bound that is simply the minimum between that given by Eq. (10) and that presented in Lemma 5.

The upper-bound on the carry-in workload of τ_i as computed in Eq. (10) may in some cases be pessimistic since the number of subtasks executing simultaneously in the workload distribution $\mathcal{WD}_i^{UCI}$ (i.e., the height of the blocks) may sometimes be greater than the number of cores m. Yet, we know for a fact that no more than m subtasks can run simultaneously on m cores. This leads to the following lemma.

Lemma 5 *An upper-bound on the maximum interfering workload that can be generated by a carry-in job of task τ_i in a carry-in window of length Δ_i^{CI} is given by* $\max\{0,\ \Delta_i^{CI} - (T_i - R_i)\} \times m$.

Proof Since τ_i cannot complete later than R_i, we know that τ_i does not execute during the last $(T_i - R_i)$ time units of the carry-in window (see Fig. 2). Therefore, τ_i executes during at most $\max\{0,\ \Delta_i^{CI} - (T_i - R_i)\}$ time units on m processors within the carry-in window of length Δ_i^{CI}, hence the claim. □

Combining Theorem 1 with Lemma 5, we derive an improved bound on the carry-in workload of an interfering task τ_i.

Theorem 2 *The interfering workload W_i^{CI} generated by the carry-in job of a higher priority task τ_i in a window of length Δ_i^{CI} is upper-bounded by* $\min\{CI_i(\mathcal{WD}_i^{UCI}, \Delta_i^{CI}),\ \max\{0,\ \Delta_i^{CI} - (T_i - R_i)\} \times m\}$.

Proof Follows from Theorem 1 and Lemma 5. □

7 Carry-out

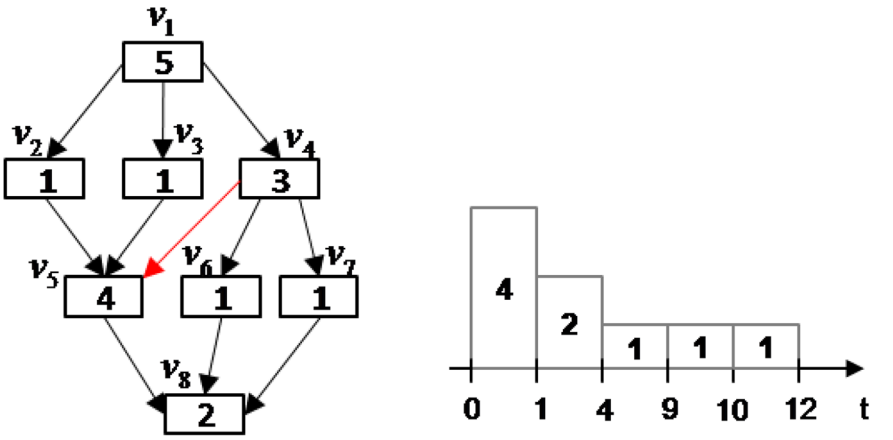
This section presents the analysis for computing an upper-bound on the carry-out part of the interfering workload of a higher priority task τ_i in the problem window $[r_k, r_k + \Delta)$ of a task τ_k. The carry-out job is the last job of τ_i released in the problem window, i.e., its release time is within the open interval $(r_k + \Delta - T_i,\ r_k + \Delta)$. Contrary to the carry-in job, the maximum interference generated by the carry-out job of τ_i is found when it starts executing as soon as it is released and at its highest possible concurrency level. That is, we are interested in pushing the workload of that job as much as possible "to the left" of the schedule. Also, contrary to the carry-in and the body jobs, finding an upper-bound on the interference generated by the carry-out job does not necessarily imply that its subtasks execute for their WCET. Indeed, unless the entire workload can contribute to the interference generated by τ_i, one must consider that any subtask may instead be instantly processed (i.e., its execution time is 0). With this assumption, some precedence constraints may be immediately resolved and the degree of parallelism in the DAG potentially increased, leading to more workload at the beginning of the carry-out window.

Example 3 Consider the DAG in Fig. 6a. If every subtask executes for its WCET then, initially, only one subtask is active (v_1) for 5 time units. On the other hand, if the subtasks v_1 and v_4 both execute for 0 time units, then the subtasks v_2, v_3, v_6 and v_7 are instantly ready and there are four subtasks active during the first time unit. Thus, if the carry-out window is only one time unit long, the latter case generates more interfering workload.

Therefore, we seek to derive a schedule that maximizes the cumulative parallelism throughout the execution of the job. We call this schedule "unrestricted carry-out" (UCO).

7.1 DAG's maximum parallelism

In order to maximize the workload produced by the carry-out job of τ_i within the problem window, we need to find an execution pattern such that the overall parallelism



(a) Example of a DAG task. **(b)** Workload distribution $\mathcal{WD}_i^{UCO}$.

Fig. 6 Running example for the carry-out workload

cannot be further increased. If the carry-out window is sufficiently short, then the maximum degree of parallelism of G_i maximizes the carry-out workload, as described in Example 3. Ideally, we would like to take the maximum parallelism of the DAG at each time instant as a solution to the problem of maximizing its cumulative parallelism within a time interval of arbitrary length. Unfortunately, this methodology cannot be applied to DAGs, since the scenario that maximizes the parallelism at a certain step may compromise the concurrency among subtasks later on. In fact, as shown in the example below, whether or not the DAG's maximum parallelism must be considered depends on the length of the carry-out window.

Example 4 Consider the DAG in Fig. 6a. The maximum parallelism is four, given by the subtasks v_2, v_3, v_6 and v_7 that can execute in parallel for at most 1 time unit. Note, however, that every schedule which maximizes the DAG's parallelism does not allow any of the remaining subtasks to execute in parallel — subtasks v_1, v_4, v_5 and v_8 have to execute sequentially due to their precedence constraints. Hence, if the maximum parallelism is reached, then the carry-out job cannot produce more than 5 units of workload within a window of length equal to 2. On the other hand, if subtask v_4 executes for 1 time unit, we can have three subtasks executing in parallel for 2 time units: first, subtasks v_2, v_3 and v_4 execute in parallel for 1 time unit, and then subtasks v_5, v_6 and v_7 also execute in parallel for 1 time unit. As a result, the latter schedule generates more interfering workload if the carry-out window is 2 time units long, but it produces at most 3 units of workload when the length of the window is reduced to 1.

The issue highlighted in Example 4 comes from the potentially very complex connection structures between subgraphs composing the DAG task. Maximizing the parallelism in one subgraph may constrain and hence reduce the achievable parallelism in another subgraph. We simplify the problem at hand by transforming the initial DAG that describes the task in a well-structured, less general, type of DAG, which we call

"nested fork-join DAG" (NFJ-DAG) (see below for an explanation on how the transformation is performed and why the transformation is safe). We define a NFJ-DAG[5] recursively as follows.

Definition 9 (*Nested Fork-Join DAG*) A DAG comprised of two nodes connected by a single edge is NFJ. Further, if G_1 and G_2 are two independent NFJ-DAGs, then the DAG obtained through either of the following operations is also NFJ:
(a) Series composition: merge the sink of G_1 with the source of G_2.
(b) Parallel composition: merge the source of G_1 with the source of G_2 and the sink of G_1 with the sink of G_2.

The series composition links two NFJ-DAGs one after another, whereas the parallel composition juxtaposes two NFJ-DAGs by merging their sources and sinks. For example, the DAG of Fig. 6a is not a NFJ-DAG because it cannot be constructed without violating the rules in Def. 9. However, if the edge (v_4, v_5) is removed, then the DAG becomes NFJ. It is clear from the definition of a NFJ-DAG that maximizing the parallelism of any of its subgraphs cannot limit the maximum parallelism achievable by other subgraphs composing the NFJ-DAG.

7.1.1 Transforming a DAG in NFJ-DAG

Many efficient algorithms exist in the literature to identify if a DAG is NFJ (Valdes et al. 1979; He and Yesha 1987). However, it is out of the scope of this paper to describe how those algorithms work. We assume here that one of those tests is performed on the graph G_i describing τ_i's structure. If it turns out that the original DAG G_i is not NFJ, a transformation is required. Traditionally, in graph theory, the transformation is performed by adding new edges between conflicting subtasks, so that the original precedences are preserved (González-Escribano et al. 2002). However, we are interested in removing edges so as to reduce the number of precedence constraints. This way, the set of all the valid schedules of τ_i (those that satisfy the precedence constraints of its original DAG G_i) is a subset of all the valid schedules of the resulting NFJ-DAG. That is because any schedule derived according to the DAG G_i will always respect all the precedence constraints of the NFJ-DAG. As a result, the maximum carry-out workload that can be generated by the NFJ-DAG is at least as large as the maximum interfering workload that can be generated by the initial DAG G_i.

Let us refer to a subtask v_j as a join-node if its "in-degree" is larger than one, i.e. $|pred(v_j)| > 1$. Similarly, we refer to a subtask v_j as a fork-node if its out-degree is larger than one, i.e. $|succ(v_j)| > 1$. According to Def. 9, a DAG (as defined in Sect. 3) is NFJ if and only if it respects the following property.

Property 1 Let $\mathcal{J}_i$ be the set of join-nodes in V_i and let $\mathcal{F}_i$ be the set of fork-nodes in V_i. DAG G_i is a NFJ-DAG iff $\forall v_j \in \mathcal{J}_i$, there exists a subgraph G' of G_i such that v_j is the sink of G', the source of G' is a fork-node $v_f \in \mathcal{F}_i$ and

$$\forall v_a \in G' \backslash \{v_f, v_j\}, \forall v_b \in \{succ(v_a) \cup pred(v_a)\},$$
$$v_b \in desce(v_f) \cup v_f \wedge v_b \in ances(v_j) \cup v_j.$$

[5] In graph theory, it is known as *two terminal series parallel digraph* (He and Yesha 1987).

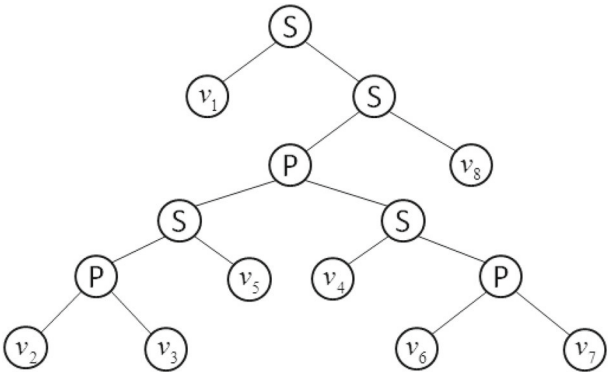


Fig. 7 Decomposition tree of the NFJ-DAG resulting from Fig. 6a

Proof The property directly follows from Def. 9, which enforces that any join-node is the result of a *parallel composition*. Hence, for every join-node v_j there must exist a fork-node v_f such that the subgraph G' that has v_f as a source and v_j as a sink is NFJ. Moreover, according to the construction rule defined in Def. 9, there cannot be any edge between a node $v_a \in G'$ and a node $v_b \notin G'$. Therefore, $\forall v_a \in G', \forall v_b \in \{succ(v_a) \cup pred(v_a)\}, v_b \in G'$, implying that $v_b \in desce(v_f) \cup v_f \wedge v_b \in ances(v_j) \cup v_j$. □

Using Property 1, a high-level algorithm for transforming a DAG G_i into a NFJ-DAG G_i^{NFJ}, can be defined as follows.

1. Select the unvisited join-node $v_j \in \mathcal{J}_i$ that is the closest to the source of G_i.
2. Find all the edges (v_c, v_j) in E_i for which there is no fork-node $v_f \in \mathcal{F}_i$ such that Prop. 1 is true. Call this set the set of conflicting edges E^C.
3. Remove as many edges in E^C as needed for join-node v_j to respect Prop. 1 or for its in-degree to become equal to 1 (in which case it is not a join node any more).
4. For each edge $(v_c, v_j) \in E^C$ that was removed, if $succ(v_c) = \emptyset$, add an edge (v_c, v_{n_i}) from node v_c to the sink of G_i.
5. Mark v_j as visited. Repeat until all join-nodes have been visited.

Example 5 The DAG of Fig. 6a has two join-nodes $\{v_5, v_8\}$. The above algorithm starts by analyzing join-node v_5. Since its ancestor v_4 has two direct successors $\{v_6, v_7\}$ which are not ancestors of v_5, (v_4, v_5) is a conflicting edge. Because there is no other conflicting edge with respect to join-node v_5, our only choice is to remove (v_4, v_5) from the DAG. In the next iteration, the DAG is already NFJ as join-node v_8 does not violate Property 1.

7.1.2 Maximum parallelism in a NFJ-DAG

By Def. 9, a NFJ-DAG can be reduced to a collection of basic DAGs by successively applying series and parallel binary decomposition rules. Therefore, a NFJ-DAG G_i^{NFJ} can be represented by a binary tree T_i, called *decomposition tree* (see Fig. 7 for an

example). Each external node (leaf) of the decomposition tree corresponds to a subtask $v_j \in V_i$, whereas each internal node represents the composition type (series or parallel) applied to its subtrees. That is, the children of a internal node are either smaller NFJ-DAGs or subtasks. A node depicting a parallel or series composition is labeled P or S, respectively. The algorithm proposed by Valdes et al. (1979) can be used to efficiently build the decomposition tree of any NFJ-DAG. Figure 7 shows the decomposition tree of the NFJ-DAG depicted in Fig. 6a (without the red edge).

The structure of the decomposition tree allows us to compute the sets of subtasks yielding the maximum parallelism of a NFJ-DAG G_i^{NFJ} in an efficient manner. The recursive function $par(T_i^U)$ defined below returns a set of subtasks in a decomposition tree T_i^U such that all subtasks in $par(T_i^U)$ can execute in parallel and the size of $par(T_i^U)$ is maximum. Note that, in Eq. (12) below, T_i^L and T_i^R denote the left and right subtrees of the binary tree T_i^U rooted in node U.

$$par(T_i^U) = \begin{cases} par(T_i^L) \cup par(T_i^R) & \text{if } U \text{ is a P-node} \\ par(T_i^L) & \text{if } U \text{ is a S-node and} \\ & |par(T_i^L)| \geq |par(T_i^R)| \\ par(T_i^R) & \text{if } U \text{ is a S-node and} \\ & |par(T_i^R)| > |par(T_i^L)| \\ \{U\} & \text{otherwise} \end{cases} \tag{12}$$

Eq. (12) works as follows. When node U denotes a parallel composition, the maximum parallelism corresponds to the sum of the maximum parallelism of its children. On the other hand, the maximum parallelism in a series composition is given by the maximum parallelism among its children. The recursion of Eq. (12) stops when U is a leaf of the decomposition tree and hence corresponds to a subtask in the associated NFJ graph. The set of subtasks in G_i^{NFJ} with maximum parallelism is obtained by calling $par(.)$ on G_i^{NFJ}'s decomposition tree root node.

7.2 Workload distribution of the carry-out job

As discussed earlier in this section, the carry-out job of an interfering task τ_i generates the maximum interfering workload when it starts executing as soon as it is released and at its highest possible concurrency level. Therefore, we use the $par(.)$ function defined above to build the workload distribution $\mathcal{WD}_i^{UCO}$ that characterizes the UCO schedule for the carry-out job of τ_i.

The workload distribution $\mathcal{WD}_i^{UCO}$ is constructed using Algorithm 1. In short, the algorithm identifies the maximum number of subtasks that can run in parallel at any point during the execution of the carry-out job as follows. It finds the largest set of subtasks which may execute in parallel according to the decomposition tree of G_i^{NFJ} (line 3). Then, it adds a new block (line 5) to the workload distribution $\mathcal{WD}_i^{UCO}$ with a width equal to the minimum WCET among those subtasks (line 4) and a height equal to the number of subtasks in the set. Finally, it proceeds by updating

the subtasks' execution times in the reduction tree, i.e., decreasing their execution time by the amount of time they executed in parallel (line 6). When a subtask reaches an execution time equal to 0 (it finishes), its corresponding leaf is removed from the decomposition tree (lines 7-8). Whenever a node of the decomposition tree has no children anymore, it is also removed from the tree. Algorithm 1 is called iteratively until all leaves have been removed.

Algorithm 1: Constructing $\mathcal{WD}_i^{UCO}$ from G_i^{NFJ}.

Input : G_i^{NFJ}, T_i^{NFJ} - A NFJ-DAG and its decomposition tree.
Output: $\mathcal{WD}_i^{UCO}$ - Workload distribution of the schedule UCO.

1 $\mathcal{WD}_i^{UCO} \leftarrow \emptyset$;
2 **while** $T_i^{NFJ} \neq \emptyset$ **do**
3 $\quad P \leftarrow par(T_i^{NFJ})$;
4 $\quad width \leftarrow min\{C_p \mid v_p \in P\}$;
5 $\quad \mathcal{WD}_i^{UCO} \leftarrow [\mathcal{WD}_i^{UCO},\ (width,\ |P|)]$;
6 $\quad \forall v_p \in P : C_p \leftarrow C_p - width$;
7 $\quad \forall v_j \in T_i^{NFJ}$ such that $C_j = 0$: remove v_j from T_i^{NFJ};
8 **end**
9 **return** $\mathcal{WD}_i^{UCO}$;

Example 6 The workload distribution $\mathcal{WD}_i^{UCO}$ for the DAG of Fig. 6a (without the red edge) is presented in Fig. 6b. It tells us that the NFJ-DAG in Fig. 6a can execute with a parallelism of 4 during 1 time unit. It can execute with a parallelism of 2 during 3 more time units and then it can finally execute with a parallelism of 1 during 8 additional time units.

7.3 Upper-bounding the carry-out workload

Similarly to what was presented for the carry-in workload, an upper-bound on the carry-out interfering workload generated by τ_i is calculated using the workload distribution $\mathcal{WD}_i^{UCO}$. Let Δ_i^{CO} denote the length of the carry-out window of τ_i (see Eq. (6)). The maximum workload executed by τ_i in any window of length Δ_i^{CO} is upper-bounded by the cumulative workload found in the first Δ_i^{CO} time units of the workload distribution $\mathcal{WD}_i^{UCO}$. Such cumulative workload is denoted by $CO_i(\mathcal{WD}_i^{UCO}, \Delta_i^{CO})$ and can be computed by the function:

$$CO_i(\mathcal{WD}_i^{UCO}, \Delta_i^{CO}) = \sum_{b=1}^{|\mathcal{WD}_i^{UCO}|} h_b \times \left[\Delta_i^{CO} - \sum_{p=1}^{b-1} w_p\right]_0^{w_b} \tag{13}$$

Example 7 If $\Delta_i^{CO} = 3$ and $\mathcal{WD}_i^{UCO}$ is given by the workload distribution presented in Fig. 6b, then Eq. (13) sums the height of the blocks in $\mathcal{WD}_i^{UCO}$ up to 3. That

is, $CO_i(\mathcal{WD}_i^{UCO}, \Delta_i^{CO}) = 8$. If Δ_i^{CO} was equal to 10, then $CO_i(\mathcal{WD}_i^{UCO}, \Delta_i^{CO})$ would be equal to 16.

We now prove that $CO_i(\mathcal{WD}_i^{UCO}, \Delta_i^{CO})$ is indeed an upper-bound on the carry-out interfering workload W_i^{CO}.

Theorem 3 *The interfering workload W_i^{CO} generated by the carry-out job of a higher priority task τ_i in a carry-out window of length Δ_i^{CO} is upper-bounded by $CO_i(\mathcal{WD}_i^{UCO}, \Delta_i^{CO})$.*

Proof We recall that τ_i's carry-out job generates the maximum interfering workload when it starts executing as soon as it is released and at its highest possible concurrency level.

First, we note that the NFJ-DAG G_i^{NFJ}, built from G_i by removing some of G_i's edges, has a concurrency level at least as high as G_i. Hence, the workload distribution $\mathcal{WD}_i^{UCO}$ constructed based on G_i^{NFJ} has at least as much workload than G_i in the carry-out window.

Since $\mathcal{WD}_i^{UCO}$ is constructed with Algorithm 1, and because Algorithm 1 computes the maximum parallelism of G_i^{NFJ} at each time t, the height of $\mathcal{WD}_i^{UCO}$ on its first Δ_i^{CO} time units maximizes the workload that τ_i can generate in the carry-out window.

Finally, because $CO_i(\mathcal{WD}_i^{UCO}, \Delta_i^{CO})$ provides the cumulative workload in $\mathcal{WD}_i^{UCO}$ over its first Δ_i^{CO} time units, $CO_i(\mathcal{WD}_i^{UCO}, \Delta_i^{CO})$ upper-bounds the interfering workload that can be generated by τ_i's carry-out job. □

7.4 Improved carry-out workload

Note that because the workload distribution $\mathcal{WD}_i^{UCO}$ is built based on the NFJ-DAG of τ_i and not on its DAG, the length of the schedule UCO may become shorter than L_i. That happens when any of the edges removed during the transformation belongs to the critical path of G_i. In fact, the length of $\mathcal{WD}_i^{UCO}$ matches the critical path length of G_i^{NFJ}, which may be shorter than the critical path of the initial DAG G_i (since edges may have been removed).

Example 8 The workload distribution $\mathcal{WD}_i^{UCO}$ presented in Fig. 6b has a length equal to 12, while the initial DAG (*with* the red edge) in Fig. 6a has a critical path composed of v_1, v_4, v_5 and v_8 of length $L_i = 14$.

As stated by Corollary 1, task τ_i cannot execute W_i time units in less than L_i time units. Therefore, we derive a new upper-bound on the interfering workload of τ_i's carry-out job, that respects Corollary 1.

Lemma 6 *The workload W_i^{CO} generated by the carry-out job of a higher priority task τ_i in a window of length Δ_i^{CO} is upper-bounded by $W_i - \max\{0, L_i - \Delta_i^{CO}\}$.*

Proof Directly follows from Lemma 1. □

Theorem 4 *The interfering workload W_i^{CO} generated by the carry-out job of a higher priority task τ_i in a window of length Δ_i^{CO} is upper-bounded by* $\min\left\{CO_i(\mathcal{WD}_i^{UCO}, \Delta_i^{CO}),\ \Delta_i^{CO} \times m,\ W_i - \max\{0, L_i - \Delta_i^{CO}\}\right\}$.

Proof Because at most m subtasks can execute simultaneously on m cores, $\Delta_i^{CO} \times m$ is an upper-bound on the workload that can execute in a window of length Δ_i^{CO}. Since $CO_i(\mathcal{WD}_i^{UCO}, \Delta_i^{CO})$ (Theorem 3) and $W_i - \max\{0, L_i - \Delta_i^{CO}\}$ (Lemma 6) are also upper-bounds on W_i^{CO}, so is the minimum between the three values. □

8 Schedulability analysis for constrained deadline tasks

In the previous two sections we have derived upper-bounds on the workload produced by the carry-in and carry-out jobs of τ_i as a function of Δ_i^{CI} and Δ_i^{CO}, respectively. Now we show how to balance Δ_i^{CI} and Δ_i^{CO} such that the interfering workload in the problem window of length Δ is maximized. In this section, we assume that all tasks have constrained deadlines (i.e., $D_i \leq T_i$). The case of arbitrary deadlines is considered in Sect. 9. If tasks have constrained deadlines, then at most one job of each higher priority task τ_i can be a carry-in job, i.e., at most one job of τ_i can be released before r_k and have a deadline after r_k. Similarly, at most one job of τ_i may be a carry-out job, i.e., there is at most one job of τ_i that can be the last job of τ_i released in the problem window.

The difficulty in computing the values Δ_i^{CI} and Δ_i^{CO} comes from the fact that the worst-case scenario for τ_k does not necessarily happen when the problem window is aligned with the start of the carry-in job or the end of the carry-out job (see Fig. 2). Furthermore, the positioning of the problem window of τ_k relatively to the release pattern of τ_i may have to vary according to the value of Δ in order to guarantee that the workload imposed by τ_i on τ_k is maximized.

Let Δ_i^C be the sum of the carry-in and the carry-out windows lengths, i.e, $\Delta_i^C = \Delta_i^{CI} + \Delta_i^{CO}$, and let $\mathcal{W}_i^C(\Delta_i^C)$ be the maximum workload produced by the carry-in and carry-out jobs of τ_i over Δ_i^C. An upper-bound on the total interfering workload generated by τ_i in a time interval of length Δ is therefore given by

$$\mathcal{W}_i(\Delta) = \mathcal{W}_i^C(\Delta_i^C) + \max\left\{0, \left\lfloor \frac{\Delta - \Delta_i^C}{T_i} \right\rfloor\right\} \times W_i \tag{14}$$

where the first term is the maximum workload produced by both the carry-in job and the carry-out job of τ_i and the second term is the maximum number of body jobs that can be released by τ_i within $(\Delta - \Delta_i^C)$, multiplied by their maximum workload. To use Eq. (14), we need to compute Δ_i^C and $\mathcal{W}_i^C(\Delta_i^C)$. The value of Δ_i^C can be computed as follows.[6]

$$\Delta_i^C = \Delta - \max\left\{0, \left\lfloor \frac{\Delta - B_i}{T_i} \right\rfloor\right\} \times T_i \tag{15}$$

[6] We note that Eq. (15) was incomplete in the original RTNS paper (Fonseca et al. 2017). We correct it here by replacing the term L_i by τ_i's best-case response time B_i. All the original experiments were performed again with the corrected equation and none was visibly impacted by the change made in Eq. (15).

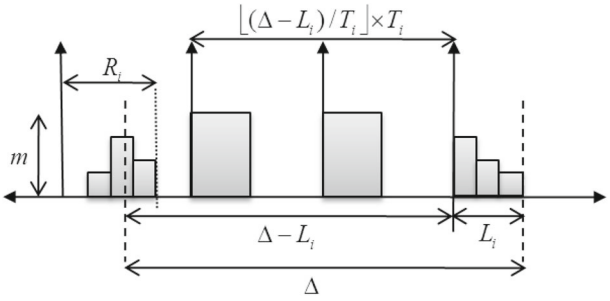


Fig. 8 Scenario that maximizes the number of body jobs released by τ_i over Δ

where B_i is the best-case response time (BCRT) of τ_i when it executes for its worst-case workload. It is given by

$$B_i = \max\left\{L_i, \frac{W_i}{m}\right\} \tag{16}$$

which was derived using Corollary 1 (i.e., the BCRT of τ_i cannot be smaller than L_i) and the fact that τ_i cannot execute on more than m processors at a time, hence B_i is lower-bounded by $\frac{W_i}{m}$.

The length Δ_i^C is thus obtained by aligning the problem window with the earliest completion time of the carry-out job of τ_i (which takes no less than B_i time units to execute) and removing all the body jobs of τ_i from the problem window of length Δ (see Fig. 8). This way, the number of full jobs of τ_i in the problem window is maximized, and so is its interference. Note that the fact that Δ_i^C is computed by aligning the problem window with the end of τ_i's carry-out job does not mean that τ_i's interference is maximized when Δ_i^{CO} contains the full carry-out job of τ_i. Instead, the window may be shifted left (yet without changing the number of body jobs) to include a larger portion of τ_i's carry-in job if it increases the total interfering workload generated by τ_i.

Lemma 7 *The interfering workload $\mathcal{W}_i(\Delta)$ generated by a higher priority task τ_i in a window of length Δ is maximized when Δ_i^C is computed by Eq.* (15).

Proof In this proof, we assume that $\Delta > B_i$ since otherwise $\Delta \leq T_i$ (i.e., assuming that τ_i is schedulable, its BCRT must be no larger than $D_i \leq T_i$) and there cannot be any body job released by τ_i. This would imply that Δ_i^C is by default equal to Δ, thereby proving the claim for that case.

Thus, if $\Delta > B_i$, we note that $B_i \leq \Delta_i^C < B_i + T_i$ when computed with Eq. (15). Two cases must be considered.

Case 1 If Δ_i^C is shortened then at most one more body job can be added to the problem window Δ (remember that $\Delta_i^C < B_i + T_i$ and $B_i \leq T_i$ and each body job executes in a window of length T_i). Therefore, the interfering workload generated by τ_i's body jobs increases by at most W_i (i.e., the workload of exactly one job). Moreover, because Δ_i^C is now T_i time units shorter, one less job can execute in Δ_i^C and the interfering

workload $\mathcal{W}_i^C(\Delta_i^C)$ generated by τ_i's carry-in and carry-out jobs must decrease by at least W_i time units too. Hence, in total, the interfering workload $\mathcal{W}_i(\Delta)$ does not increase.

Case 2 The length of Δ_i^C is increased. Using Eq. (15), the computed value of Δ_i^C is Δ minus an integer multiple of T_i and thus, when injecting Eq. (15) into Eq. (14), we get that $\left\lfloor \frac{\Delta-\Delta_i^C}{T_i} \right\rfloor = \frac{\Delta-\Delta_i^C}{T_i}$. By increasing Δ_i^C by a positive value ϵ, it thus holds that $\left\lfloor \frac{\Delta-(\Delta_i^C+\epsilon)}{T_i} \right\rfloor < \left\lfloor \frac{\Delta-\Delta_i^C}{T_i} \right\rfloor$ for $\epsilon > 0$. Therefore, at least one less body job can execute in the time window of length Δ and the interfering workload generated by τ_i's body jobs is decreased by at least W_i. Furthermore, since the carry-out job is already completely included in Δ_i^C (i.e., $\Delta_i^C \geq B_i$), in the best case increasing the length of Δ_i^C will allow us to fully integrate τ_i's carry-in job in $\mathcal{W}_i^C(\Delta_i^C)$. Hence, $\mathcal{W}_i^C(\Delta_i^C)$ may be increased by at most W_i time units (the workload of τ_i's carry-in job). Summing all the contributions to the interfering workload $\mathcal{W}_i(\Delta)$, we have that $\mathcal{W}_i(\Delta)$ does not increase. □

The problem of computing $\mathcal{W}_i^C(\Delta_i^C)$ can be formulated as the maximization of $CI_i(\mathcal{WD}_i^{UCI}, x1) + CO_i(\mathcal{WD}_i^{UCO}, x2)$ subject to $\Delta_i^C = x1 + x2$. The optimal solution of this optimization problem is an upper-bound on $\mathcal{W}_i^C(\Delta_i^C)$, whereas the final values of the decisions variables $x1$ and $x2$ correspond to Δ_i^{CI} and Δ_i^{CO}, respectively. We solve this problem by using Algorithm 2 that is based on a technique named "sliding window" introduced in Maia et al. (2014). It computes the maximum solution to the optimization problem defined above in linear time by checking all possible scenarios in which the problem window is aligned with any block of $\mathcal{WD}_i^{UCI}$ or $\mathcal{WD}_i^{UCO}$. Specifically, the scenarios tested can be divided into two groups: (i) the beginning of the problem window coincides with the start of a block in $\mathcal{WD}_i^{UCI}$ (lines 7 to 14); or (ii) the problem window ends at the completion of a block in $\mathcal{WD}_i^{UCO}$ (lines 15 to 22). Algorithm 2 also tries the configuration where the carry-out workload in the problem window is maximized (lines 1 to 3) and where the carry-in workload in is maximized (lines 4 to 6). It was proven in Maia et al. (2014), that the maximum interfering workload is obtained in one of those scenarios.

By replacing the terms $\mathcal{W}_i(R_k)$ $(1 \leq i < k)$ with Eq. (14) in Eq. (4), a schedulability condition for task τ_k is stated in the next theorem.

Theorem 5 *A task τ_k is schedulable under G-FP iff $R_k \leq D_k$, where R_k is the smallest $\Delta > 0$ to satisfy $\Delta = L_k + \frac{1}{m}(W_k - L_k) + \frac{1}{m}\sum_{\forall i<k} \mathcal{W}_i(\Delta)$.*

The task set is declared schedulable if all tasks are schedulable. This can be checked by applying Theorem 5 to each task $\tau_i \in \tau$, starting from the highest priority task (i.e., τ_1) and proceeding in decreasing order of priority.

Algorithm 2: Computing $\mathcal{W}_i^C$ for constrained deadline tasks.

Input : $\Delta_i^C, \mathcal{WD}_i^{UCI}, \mathcal{WD}_i^{UCO}$.
Output: $\mathcal{W}_i^C$ - Upper-bound on the workload of both the carry-in and carry-out jobs.

/* We maximize the carry-out workload inside the problem window */
$x2 \leftarrow \min\{\Delta_i^C,\ B_i\}$;
$x1 \leftarrow \Delta_i^C - x2$;
$\mathcal{W}_i^C \leftarrow CI_i(\mathcal{WD}_i^{UCI}, x1) + CO_i(\mathcal{WD}_i^{UCO}, x2)$;

/* We maximize the carry-in workload inside the problem window */
$x1 \leftarrow \min\{\Delta_i^C,\ B_i + (T_i - R_i)\}$;
$x2 \leftarrow \Delta_i^C - x1$;
$\mathcal{W}_i^C \leftarrow \max\{\mathcal{W}_i^C,\ CI_i(\mathcal{WD}_i^{UCI}, x1) + CO_i(\mathcal{WD}_i^{UCO}, x2)\}$;

/* We align the start of the problem window with the boundaries of every block in $\mathcal{WD}_i^{UCI}$ */
$x1 \leftarrow T_i - R_i$;
foreach $(w_b, h_b) \in \mathcal{WD}_i^{UCI}$ *in reverse order* **do**
  $x1 \leftarrow x1 + w_b$;
  $x2 \leftarrow \Delta_i^C - x1$;
  if $x2 \geq 0$ **then**
    $\mathcal{W}_i^C \leftarrow \max\{\mathcal{W}_i^C,\ CI_i(\mathcal{WD}_i^{UCI}, x1) + CO_i(\mathcal{WD}_i^{UCO}, x2)\}$;
  end
end

/* We align the end of the problem window with the boundaries of every block in $\mathcal{WD}_i^{UCO}$ */
$x2 \leftarrow 0$;
foreach $(w_b, h_b) \in \mathcal{WD}_i^{RCO}$ *in order of appearance* **do**
  $x2 \leftarrow x2 + w_b$;
  $x1 \leftarrow \Delta_i^C - x2$;
  if $x1 \geq 0$ **then**
    $\mathcal{W}_i^C \leftarrow \max\{\mathcal{W}_i^C,\ CI_i(\mathcal{WD}_i^{UCI}, x1) + CO_i(\mathcal{WD}_i^{UCO}, x2)\}$;
  end
end
return $\mathcal{W}_i^C$;

9 Schedulability analysis for arbitrary deadline tasks

In the previous section, we presented a RTA for the special case where all tasks have constrained deadlines. In this section, we treat the general case where tasks may have arbitrary deadlines.

The difficulty with arbitrary deadline tasks is twofold:

1. Let J_k be the job of τ_k for which we compute the WCRT and assume that J_k is released at time r_k. Since it may be that $D_k > T_k$, more than one job of τ_k may execute in the problem window $[r_k,\ r_k + \Delta)$. That is, jobs of τ_k released before

r_k (i.e., at time $t \leq r_k - T_k$) may not have completed their execution at r_k and yet τ_k may still be schedulable (i.e., it completes all jobs before their deadlines). Therefore, Eq. (4) that computes the WCRT of τ_k must be updated to integrate the residual workload of jobs of τ_k released before r_k but interfering with J_k's execution.

2. The second difficulty is that higher priority tasks may have more than one carry-in job. Specifically, if $D_i > T_i$, more than one job of τ_i may be released before r_k and have a deadline after r_k. This property, which is formally proven in Lemma 8 in Sect. 9.3, requires to derive a new bound on the carry-in wokload released by each higher priority task interfering with τ_k.

We address the first issue in Sect. 9.1 and the second in Sect. 9.3.

9.1 Response time analysis

In this section, we update Eq. (4) and derive a new bound on the WCRT of a task τ_k. We integrate the fact that, for arbitrary deadline tasks, a job $J_{k,l}$ of task τ_k may be released before the completion of its preceding job $J_{k,l-1}$. Indeed, let us assume that $J_{k,l-1}$ and $J_{k,l}$ were released at time $r_{k,l-1}$ and $r_{k,l}$, respectively. In the worst-case scenario we have that $r_{k,l} = r_{k,l-1} + T_k$ and $J_{k,l-1}$ may complete its execution at any time smaller than or equal to $(r_{k,l-1} + D_k)$. Therefore, if $D_k > T_k$, job $J_{k,l-1}$ may not have completed its execution when $J_{k,l}$ is released. In such situation, we assume that $J_{k,l}$ does not start executing before the completion of $J_{k,l-1}$.[7] Hence the earliest instant at which $J_{k,l}$ may start executing is not its release time $r_{k,l}$ anymore, but the maximum between its release time and the completion time of $J_{k,l-1}$.

We now consider the two cases mentioned above:

1. if job $J_{k,l}$ can start executing as soon as it is released (i.e., at $r_{k,l}$), then the previous job $J_{k,l-1}$ of τ_k has already completed by time $r_{k,l}$. In such case, the situation is identical, with respect to $J_{k,l}$, to the worst-case scenario considered for constrained deadline tasks. That is, there is no additional interference by previous jobs of τ_k and the WCRT of $J_{k,l}$ is therefore obtained using Eq. (4) and maximizing the higher priority task interference. This scenario is encountered for the first job released by τ_k. Let $X_{k,1}$ be the completion time of that job. Without any loss of generality we can assume that that job was released at time 0. Hence we have $r_{k,1} = 0$ and, using Eq. (4), $X_{k,1}$ is upper-bounded by the smallest positive solution to

$$X_{k,1} = L_k + \frac{1}{m}(W_k - L_k) + \frac{1}{m} \sum_{\forall i<k} \mathcal{W}_i(X_{k,1}) \tag{17}$$

2. if job $J_{k,l-1}$ is not yet completed when $J_{k,l}$ is released, then $J_{k,l}$ cannot start executing before the completion of $J_{k,l-1}$. Therefore, the worst-case scenario for $J_{k,l}$ happens when the overlap between the execution window of $J_{k,l-1}$ and the active window of $J_{k,l}$ is maximized. This happens when $J_{k,l-1}$ completes as late

[7] We enforce this execution behavior to avoid data inconsistencies between successive jobs of a same task. Indeed, a job may require the computation results of its preceding job to be able to proceed correctly.

as possible and $J_{k,l}$ is released as early as possible. Assume that $X_{k,l-1}$ and $X_{k,l}$ are the worst-case completion times of $J_{k,l-1}$ and $J_{k,l}$, respectively. The WCRT of $J_{k,l}$ is then given by

$$\begin{aligned} R_{k,l} &= X_{k,l} - r_{k,l} \\ &= X_{k,l} - (l-1) \times T_k \end{aligned} \tag{18}$$

where

$$X_{k,l} = X_{k,l-1} + L_k + \frac{1}{m}(W_k - L_k) + \frac{1}{m} \sum_{\forall i<k} \Big(\mathcal{W}_i(X_{k,l}) - \mathcal{W}_i(X_{k,l-1}) \Big) \tag{19}$$

Eq. (19) is composed of four terms detailed hereafter.

- $X_{k,l-1}$ is the worst-case completion time of the preceding job $J_{k,l-1}$, i.e., the earliest time at which $J_{k,l}$ may start executing;
- L_k is the minimum amount of time required by $J_{k,l}$ to complete its execution when it executes for its WCET and does not suffer any interference;
- $\frac{1}{m}(W_k - L_k)$ is an upper-bound on $J_{k,l}$'s self-interference (as proven in Melani et al. (2017));
- $\frac{1}{m} \sum_{\forall i<k} \big(\mathcal{W}_i(X_{k,l}) - \mathcal{W}_i(X_{k,l-1}) \big)$ is the maximum interfering workload that can be released by higher priority tasks in the problem window of length $X_{k,l}$ that has not yet been accounted for in the term $X_{k,l-1}$, i.e., the worst-case completion time of $J_{k,l-1}$.

The WCRT of a task τ_k is thus given by its job with the largest response time. Formally,

$$R_k = \max_{l>0} \left\{ X_{k,l} - (l-1) \times T_k \right\} \tag{20}$$

where $X_{k,l}$ is the worst-case completion time of the l^{th} job released by τ_k in the problem window. Combining Eqs. (17) and (19) we get that

$$X_{k,l} = l \times (L_k + \frac{1}{m}(W_k - L_k)) + \frac{1}{m} \sum_{\forall i<k} \mathcal{W}_i(X_{k,l}) \tag{21}$$

Note that we can stop iterating over l when

- we reach the first $l > 0$ such that $X_{k,l} \leq (l \times T_k)$, i.e., the first job of τ_k released in the problem window that completes before the release of the next job of τ_k;
- we reach the first $l > 0$ such that $X_{k,l} > (l-1) \times T_k + D_k$, i.e., at the first job of τ_k released in the problem window that has a response time larger than its deadline.

In the first case the task τ_k is schedulable while in the second it is not. One of these two termination conditions holds eventually in most cases. However, it cannot be guaranteed that Eq. (20) always terminates in the general case, as it has already been

shown for sequential tasks (Guan et al. 2009). Such rather special corners cases have not been detected at all during our experimental evaluation. Nonetheless, one can simply define a threshold for the values of l. Whenever the threshold is reached, the procedure terminates and the task τ_k is declared unschedulable. Note that this may decrease the effectiveness of the response time analysis.

The term $\mathcal{W}_i(X_{k,l})$ in Eq. (21) is computed using Eq. (14). Equation (14) uses an upper-bound $\mathcal{W}_i^C$ on the carry-in and carry-out workload that can be released by higher priority task τ_i. As discussed at the beginning of this section, each higher priority task may execute more than one carry-in job in the problem window and a new bound on $\mathcal{W}_i^C$ must be derived. We present this bound in the next subsections.

9.2 Carry-out workload

As defined in Sect. 4, a carry-out job is a job that is released in the problem window less than T_i time units before the end of that window. Hence the carry-out job of τ_i is the last job that can be released by τ_i in the problem window (remember that job releases are at least T_i time units apart). Therefore, each higher priority task τ_i can release at most one carry-out job, even when τ_i has an arbitrary deadline. It results that the upper-bound on the carry-out workload proven in Theorem 4 is still valid for arbitrary deadline tasks.

9.3 Carry-in workload

As mentioned in Sect. 4, a carry-in job is defined as a job released before the start of the problem window and with a deadline after the problem window start. When a higher priority task τ_i has a deadline smaller than or equal to its minimum inter-arrival time (i.e., $D_i \leq T_i$), at most one such carry-in job may exist. However, this result does not hold for tasks with arbitrary deadlines. Indeed, it may happen that $D_i > T_i$, in which case a job of τ_i may have its deadline after the release of one (or several) other job(s) of τ_i. Yet, the number of carry-in jobs may still be upper-bounded as proven in Lemma 8.

Lemma 8 *Each higher priority task τ_i with an arbitrary deadline has at most $\left\lceil \frac{D_i}{T_i} \right\rceil$ carry-in jobs.*

Proof Let J_i be the earliest carry-in job released by τ_i. Let r_i be its release time and d_i its absolute deadline. By definition of J_i, all jobs released before r_i are not carry-in jobs. Let $c = \left\lceil \frac{D_i}{T_i} \right\rceil$. Let J_{i+c} be any job of τ_i released at or later than $(r_i + c \times T_i)$. Then, J_{i+c} is released at or after d_i (because $d_i = r_i + D_i \leq r_i + \left\lceil \frac{D_i}{T_i} \right\rceil \times T_i = r_i + c \times T_i$). Since J_i is a carry-in job, d_i is necessarily after the problem window start. Hence any job J_{i+c} is released after the problem window start and is not a carry-in job. Since at most $c - 1$ jobs of τ_i can be released between r_i and $r_i + c \times T_i$, we conclude that there are at most $c - 1$ other jobs than J_i that may be carry-in jobs. This proves the claim. □

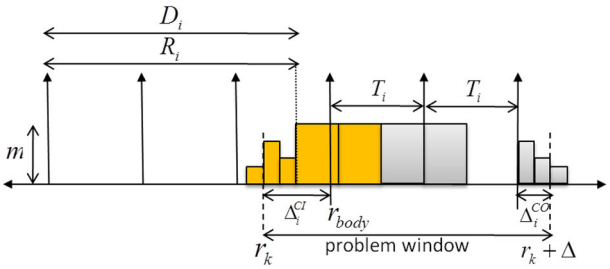


Fig. 9 Worst-case interfering workload released by τ_i in τ_k's problem window when $D_i > T_i$. Yellow jobs are carry-in jobs

Note that Lemma 8 covers the case of constrained deadline tasks too since $\left\lceil \frac{D_i}{T_i} \right\rceil = 1$ in that particular case.

Example 9 Consider the worst-case interfering scenario of task τ_i depicted in Fig. 9. We have that $D_i = 2.6 \times T_i$. Hence three jobs may be released by τ_i before r_k and have their deadline after r_k. Further, because in this example $R_i = D_i$, the three carry-in jobs (in yellow in the picture) execute at least partially in the problem window starting at time r_k.

Since there might be more than one carry-in job released by τ_i, we must update the definition of Δ_i^{CI} (Eq. (5)) and the upper-bound on the worst-case carry-in interfering workload (Eq. (10)).

As depicted in Fig. 2 for constrained deadline tasks and in Fig. 9 for arbitrary deadline tasks, we define the carry-in window of τ_i as the interval starting at the beginning of the problem window (i.e., at time r_k) and ending at the earliest release of a *body job* of τ_i. Therefore, if r_{body} is the release time of that job, we have that

$$\Delta_i^{CI} \stackrel{\text{def}}{=} r_{body} - r_k \tag{22}$$

By Lemma 8, we know that there are at most $\left\lceil \frac{D_i}{T_i} \right\rceil$ carry-in jobs released before r_{body}. Therefore, the j^{th} carry-in job of τ_i (with $1 \leq j \leq \left\lceil \frac{D_i}{T_i} \right\rceil$) cannot be released later than time

$$r_{i,j} \stackrel{\text{def}}{=} r_{body} - j \times T_i \tag{23}$$

$$= r_k + \Delta_i^{CI} - j \times T_i \tag{24}$$

Similar to the constrained deadline case, the carry-in workload generated by τ_i would be maximized if each carry-in job of τ_i is released as late as possible and executes as much workload as possible in the problem window. Now, let R_i be the upper-bound on the worst-case response time of τ_i computed with Eq. (21). Lemma 9 (see below) proves that aligning $\mathcal{WD}_i^{UCI}$ to the right with the time-instant $(r_{i,j} + R_i)$ and calculating the part of $\mathcal{WD}_i^{UCI}$'s workload released after r_k (using Eq. (10))

provides an upper-bound on the maximum interfering workload that can be generated by the j^{th} carry-in job of τ_i. Formally, we have that the interfering workload executed by the j^{th} carry-in job of τ_i in the problem window is upper-bounded by

$$CI_{i,j}(\mathcal{WD}_i^{UCI}, \Delta_i^{CI}) = \sum_{b=1}^{|\mathcal{WD}_i^{UCI}|} h_b \times \left\lceil \Delta_i^{CI} - j \times T_i + R_i - \sum_{p=b+1}^{|\mathcal{WD}_i^{UCI}|} w_p \right\rceil_0^{w_b} \tag{25}$$

This is stated in Lemma 9 below.

Lemma 9 *Let R_i be the upper-bound on the worst-case response time of τ_i **computed by Eq.** (21). Aligning $\mathcal{WD}_i^{UCI}$ to the right with the time-instant $(r_{i,j} + R_i)$ gives an upper-bound on the maximum interfering workload that can be generated by τ_i's carry-in job released at $r_{i,j}$ in the carry-in window, independently of the interference imposed on τ_i.*

Proof Since Eqs. (4) and (21) both compute the WCRT of a task based on the following algorithm (i) summing all the self-interfering workload and all the workload released by higher priority tasks in the problem window, (ii) dividing it by the number of cores m, and (iii) adding the result to τ_k's critical path length, the proof of this lemma is identical in every word to the proof of Lemma 4, replacing Eq. (4) with Eq. (21). □

Since there are up to $\left\lceil \frac{D_i}{T_i} \right\rceil$ carry-in jobs, we have that the maximum interfering carry-in workload generated by τ_i is given by the sum of the interfering workload generated by each of its carry-in jobs. That is,

$$\begin{aligned} &CI_i(\mathcal{WD}_i^{UCI}, \Delta_i^{CI}) \\ &= \sum_{j=1}^{\left\lceil \frac{D_i}{T_i} \right\rceil} \left(\sum_{b=1}^{|\mathcal{WD}_i^{UCI}|} h_b \times \left[\Delta_i^{CI} - j \times T_i + R_i - \sum_{p=b+1}^{|\mathcal{WD}_i^{UCI}|} w_p \right]_0^{w_b} \right) \end{aligned} \tag{26}$$

Note that the actual implementation of Eq. (26) can be drastically simplified using two simple mathematical facts on Eq. (26):

1. for each carry-in job j such that $(\Delta_i^{CI} - j \times T_i + R_i - L_i) \geq 0$, the contribution of the inner-sum to the carry-in workload will always be W_i;
2. for each carry-in job such that $(\Delta_i^{CI} - j \times T_i + R_i) \leq 0$, the contribution of the inner-sum to the carry-in workload will always be 0.

This means that there is at most one carry-in job and therefore only one j for which the summation on b needs to be done. For all the other $\left\lceil \frac{D_i}{T_i} \right\rceil - 1$ carry-in jobs, the interfering workload can readily be considered to be equal to W_i or 0 depending on whether $(\Delta_i^{CI} - j \times T_i + R_i - L_i) \geq 0$ or $(\Delta_i^{CI} - j \times T_i + R_i) \leq 0$, respectively.

Example 10 Consider the example in Fig. 10 where $D_i = 2.6 \times T_i$. As in Example 9, task τ_i releases three carry-in jobs. However, because the WCRT R_i of τ_i is smaller

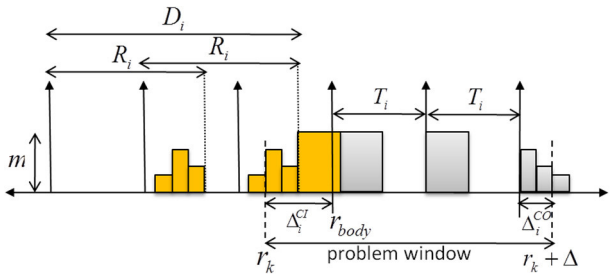


Fig. 10 Worst-case interfering workload released by τ_i in τ_k's problem window when $D_i > T_i$ but $R_i < D_i$. Yellow jobs are carry-in jobs

than D_i, the carry-in job released at $(r_{body} - 3 \times T_i)$ completes no later than time $(r_{body} - 3 \times T_i + R_i)$ which is before the start of the problem window (i.e., time r_k). Therefore, we have that $(\Delta_i^{CI} - 3 \times T_i + R_i) < 0$ and the contribution of that carry-in job to the interfering workload is 0. On the other hand, the carry-in job released at time $(r_{body} - T_i)$ respects the inequality $(\Delta_i^{CI} - T_i + R_i - L_i) \geq 0$ since it starts and complete after the beginning of the problem window. Therefore, its contribution to the interfering workload is equal to its total workload W_i. For the carry-in job released at time $(r_{body} - 2 \times T_i)$, none of the two conditions holds. Hence its execution overlaps with the beginning of the problem window and its contribution to the interfering workload is a portion of its workload distribution $\mathcal{WD}_i^{UCI}$.

Theorem 6 *The interfering workload W_i^{CI} generated by the carry-in jobs of a higher priority task τ_i in a window of length Δ_i^{CI} is upper-bounded by Eq. (26).*

Proof It directly follows from the combination of Lemmas 8 and 9. □

Similar to the constrained deadline case covered in Sect. 6.3, an improve bound on the carry-in workload can be derived using Lemma 10 proven below.

Lemma 10 *An upper-bound on the maximum interfering workload that can be generated by a carry-in job of task τ_i released at time $r_{i,j}$ in a carry-in window of length Δ_i^{CI} is given by* $\max\{0,\ \Delta_i^{CI} - j \times T_i + R_i\} \times m$.

Proof Since no job of τ_i can complete later than R_i time units after its release, we know that the carry-in job released at $r_{i,j}$ completes no later than $r_{i,j} + R_i = r_k + \Delta_i^{CI} - j \times T_i + R_i$ (using Eq. (24)). Therefore, the carry-in job executes during at most $\max\{0,\ \Delta_i^{CI} - j \times T_i + R_i\}$ time units on m processors within the carry-in window $[r_k,\ r_k + \Delta_i^{CI})$, hence the claim. □

Combining Theorem 6 with Lemma 10, we derive an improved bound on the carry-in workload of an interfering task τ_i with arbitrary deadline.

Theorem 7 *The interfering workload W_i^{CI} generated by the carry-in jobs of a higher priority task τ_i in a window of length Δ_i^{CI} is upper-bounded by*

$$\sum_{j=1}^{\left\lceil \frac{D_i}{T_i} \right\rceil} \min \left\{ \max\{0,\ \Delta_i^{CI} - j \times T_i + R_i\} \times m, \right.$$
$$\left. \sum_{b=1}^{|\mathcal{WD}_i^{UCI}|} h_b \times \left[\Delta_i^{CI} - j \times T_i + R_i - \sum_{p=b+1}^{|\mathcal{WD}_i^{UCI}|} w_p \right]_0^{w_b} \right\} \tag{27}$$

Proof Follows from Theorem 6 and Lemma 10. □

9.4 Upper-bounding the carry-in and carry-out Interference

In the previous subsections, we have upper-bounded the carry-in and carry-out interference that a higher priority task τ_i can generate in windows of length Δ_i^{CI} and Δ_i^{CO}, respectively. However, as already discussed in Sect. 8 for the constrained deadline case, the difficulty is to identify the lengths of Δ_i^{CI} and Δ_i^{CO} that maximize the total interference generated by τ_i. For constrained deadline tasks, this optimization problem was solved using Algorithm 2. In this section, we adapt Algorithm 2 to support systems composed of arbitrary deadline tasks. The result is presented in Algorithm 3.

Like for the constrained deadline case, Algorithm 3 uses the sliding window technique to maximize the interfering workload released by a task τ_i in the problem window. First, the distance Δ_i^C, which by definition is equal to $\Delta_i^{CI} + \Delta_i^{CO}$, is computed using Eq. (15) (note that the proof of Lemma 7 is still valid for arbitrary deadline tasks). Then, Algorithm 3 is called.

Algorithm 3 is identical to Algorithm 2 for lines 1 to 3 and lines 17 to 24, which are related to the carry-out workload. However, as it was to be expected, Algorithm 3 differs from Algorithm 2 for parts that are related to the carry-in workload (lines 4 to 16).

Algorithm 3 first tries to maximize the carry-out workload released by τ_i in the problem window (lines 1 to 3). To this end, it aligns the end of the problem window with the earliest time at which τ_i's carry-out job may complete (i.e., setting Δ_i^{CO} to B_i), or by setting Δ_i^{CO} to Δ_i^C if Δ_i^C is smaller than the BCRT of τ_i. Then, Algorithm 3 similarly tries to maximize the carry-in workload released by τ_i in the problem window (lines 4 to 6). This is achieved by aligning the beginning of the problem window with the latest time at which the earliest carry-in job of τ_i may start executing. Hence we set Δ_i^{CI} to $(B_i + \left\lceil \frac{D_i}{T_i} \right\rceil \times T_i - R_i)$, where $(\left\lceil \frac{D_i}{T_i} \right\rceil \times T_i - R_i)$ is the smallest possible distance between the completion of the earliest carry-in job of τ_i and the release of its first body job at r_{body}. The length $(B_i + \left\lceil \frac{D_i}{T_i} \right\rceil \times T_i - R_i)$ is thus the smallest possible distance between the time at which the earliest carry-in job of τ_i starts executing and r_{body}. Line 4 also ensures that Δ_i^{CI} cannot be larger than Δ_i^C.

Lines 6 to 16 iterate over the $\left\lceil \frac{D_i}{T_i} \right\rceil$ carry-in jobs released by τ_i. For each carry-in job it computes the latest time at which that job may complete (line 8) and then aligns the beginning of the problem window with the start of every block in the workload distribution $\mathcal{WD}_i^{UCI}$ of that job (lines 10 to 14).

Algorithm 3: Computing $\mathcal{W}_i^C$ for arbitrary deadline tasks.

Input : $\Delta_i^C, \mathcal{WD}_i^{UCI}, \mathcal{WD}_i^{UCO}$.
Output: $\mathcal{W}_i^C$ - Upper-bound on the workload of both the carry-in and carry-out jobs.

```
   /* We maximize the carry-out workload inside the problem window  */
1  x2 ← min{Δ_i^C, B_i};
2  x1 ← Δ_i^C − x2;
3  W_i^C ← CI_i(WD_i^UCI, x1) + CO_i(WD_i^UCO, x2);

   /* We maximize the carry-in workload inside the problem window   */
4  x1 ← min{Δ_i^C, B_i + ⌈D_i/T_i⌉ × T_i − R_i};
5  x2 ← Δ_i^C − x1;
6  W_i^C ← max{W_i^C, CI_i(WD_i^UCI, x1) + CO_i(WD_i^UCO, x2)};

   /* We align the start of the problem window with the boundaries of
      every block in WD_i^UCI for every carry-in job of τ_i          */
7  forall the j = 1 to ⌈D_i/T_i⌉ do
8      x1 ← j × T_i − R_i;
9      foreach (w_b, h_b) ∈ WD_i^UCI in reverse order do
10         x1 ← x1 + w_b;
11         x2 ← Δ_i^C − x1;
12         if x1 ≥ 0 and x2 ≥ 0 then
13             W_i^C ← max{W_i^C, CI_i(WD_i^UCI, x1) + CO_i(WD_i^UCO, x2)};
14         end
15     end
16 end

   /* We align the end of the problem window with the boundaries of
      every block in WD_i^UCO                                        */
17 x2 ← 0;
18 foreach (w_b, h_b) ∈ WD_i^RCO in order of appearance do
19     x2 ← x2 + w_b;
20     x1 ← Δ_i^C − x2;
21     if x1 ≥ 0 then
22         W_i^C ← max{W_i^C, CI_i(WD_i^UCI, x1) + CO_i(WD_i^UCO, x2)};
23     end
24 end
25 return W_i^C;
```

Lines 17 to 24 are identical to Algorithm 2 and align the end of the problem window with the end of every block in the workload distribution $\mathcal{WD}_i^{UCO}$ of τ_i's carry-out job.

The maximum interfering workload released by carry-in and carry-out jobs of τ_i is the maximum over the interfering workload computed for each of the scenarios described above (as already discussed in Maia et al. (2014)).

10 Experimental evaluation

The analysis presented in this paper has been implemented within the MATLAB framework released by the authors of Melani et al. (2015). We follow the same technique as in He and Yesha (1987) and Melani et al. (2015) to generate random task sets composed of DAG tasks.

Each DAG in the task set is initially a composition of two NFJ-DAGs connected in series. The NFJ-DAGs are constructed by recursively expanding their nodes. Each node has a probability p_{par} to fork and a probability p_{term} to join, where $p_{term} + p_{par} = 1$. Each parallel branch has a maximum *depth* that limits the number of nested forks. Additionally, the number of parallel branches leaving from a fork node is randomly chosen within a uniform distribution bounded by $[2, n_{par}]$. Finally, a general DAG is obtained by randomly adding directed edges between pairs of nodes, granted that such randomly-placed precedence constraints do not violate the "acyclic" semantics of the DAG. The probability of adding an edge between two nodes is given by p_{add}, with the restriction that any two nodes with a common fork-node as direct predecessor cannot be connected. This last restriction avoids generating degenerated DAGs that behave as sequential tasks.

Once the DAG G_i of a task τ_i is constructed, the task parameters are assigned as follows. The WCET C_j of a subtask $v_j \in V_i$ is uniformly chosen in the interval $[1, 100]$. The task length L_i, the workload W_i and the maximum makespan M_i (see Eq. 2) of τ_i are computed based on the internal structure of the DAG and the WCET of its nodes. The minimum inter-arrival time T_i is uniformly chosen in the interval $[M_i, W_i/\beta]$, where the parameter β is used to define the minimum utilization of all the tasks. Therefore, the task utilization becomes uniformly distributed over $[\beta, W_i/M_i]$. For all experiments that have a varying total utilization U_{tot} (i.e., Figs. 11, 16), we keep generating and adding new tasks to the task set until the target total utilization U_{tot} is met. U_{tot} is achieved exactly by adjusting the period of the last task added to the system. Otherwise, for all other experiments, we use UUnifast (Bini and Buttazzo 2005) to derive individual task utilizations (and consequently their period) for a fixed value of n. Priorities are assigned following the DM policy.

For each tested system configuration, we generated and assessed the schedulability of 500 task sets. Unless stated otherwise, in all experiments reported herein, we have set $p_{par} = 0.8$, $p_{term} = 0.2$, $depth = 2$, $n_{par} = 5$, $p_{add} = 0.2$, $\beta = 0.035 \times m$, $U_{tot} = 0.7m$, $n = 1.5m$ and $m = 8$. These settings lead to a rich variety of internal DAG structures, some of which resemble real-world applications as noted in Melani et al. (2017): we observed both heavy and unbalanced workloads with different degrees of parallelism and sequential segments in each task set. The maximum parallelism of a DAG (i.e., the number of subtasks that can execute in parallel) with such configuration is 25.

10.1 Evaluation for constrained deadlines

We compare our response time analysis for DAG tasks with constrained deadlines (referred to as IRTA-FP) to the schedulability analysis described in Melani et al.

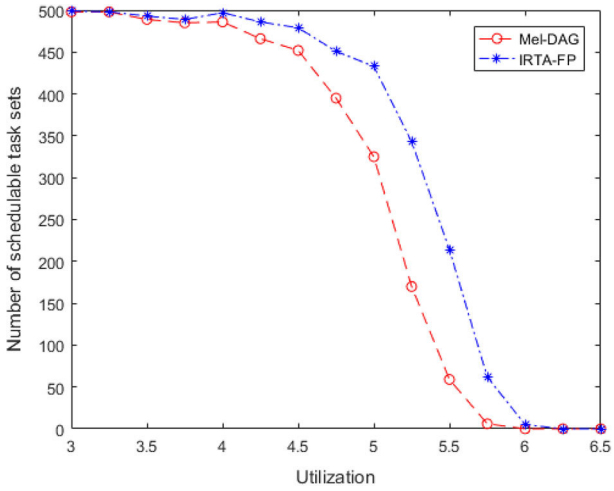


Fig. 11 IRTA-FP varying U_{tot}

(2015) (referred to as Mel-DAG) for G-FP scheduling. In an attempt to maximize the schedulability ratios of these tests, we restrict our attentions to the case where the relative deadline D_i is set equal to the period T_i. For insights concerning how RTA for G-FP scheduling fares against other scheduling algorithms and/or paradigms, the interested reader is referred to the experimental results reported in Melani et al. (2017), Jiang et al. (2017), and Pathan et al. (2018). Note that the different scheduling strategies are incomparable, since their performance varies significantly according to the application parameters.

In the first set of experiments, the system utilization U_{tot} was varied in $(0, m]$ by steps of 0.25. Figure 11 shows the number of schedulable task sets when $m = 8$. For both low and very high utilization (i.e., when all or none of the task sets are schedulable), IRTA-FP and Mel-DAG are indistinguishable. However, for $U_{tot} \in [4, 6]$, IRTA-FP performs substantially better. In particular, when $U_{tot} = 5.25$, IRTA-FP schedules 341 task sets against 156 for Mel-DAG. Instead, Fig. 12 reports the schedulability as a function of the number of tasks n, with n ranging from 4 to 20. The values of U_{tot} and m were kept constant and equal to $0.7m$ and 8, respectively. IRTA-FP outperforms Mel-DAG for any value of n with an average gain of approximately 20%, although both tests converge to full schedulability for larger n. Intuitively, it is easier to schedule many light tasks than few heavy tasks.

We then study the impact of the DAG structures on the outcome of the two schedulability tests. A trend similarly to that of Fig. 12 can be observed in Fig. 13, where we varied the maximum number of parallel branches n_{par} in the interval [2, 8]. Mel-DAG has clear limitations when the average parallelism of the DAGs is up to half of the platform's parallelism (i.e., $n_{par} \leq 4$) and only admits a large share of tasks sets for $n_{par} \geq 6$. On the other hand, IRTA-FP accepts at least 50% of the task sets for $n_{par} \geq 4$ even though the schedulability ratio reduces when the tasks become nearly sequential (i.e., n_{par} becomes close to 2). As expected, both approaches are comparable when the task parallelism is consistently greater than m. Figure 14 reports

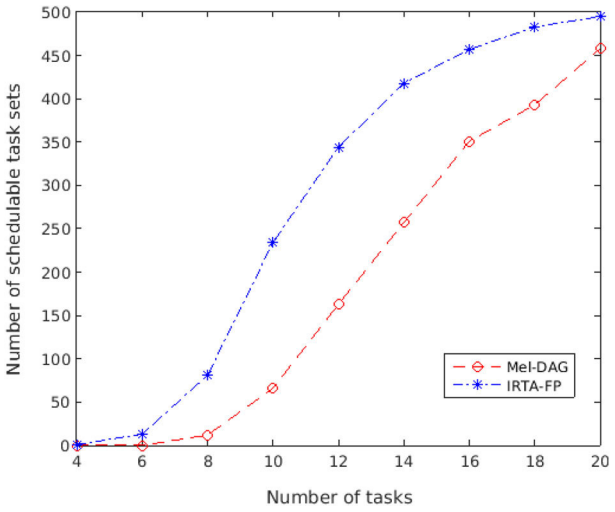


Fig. 12 IRTA-FP varying n

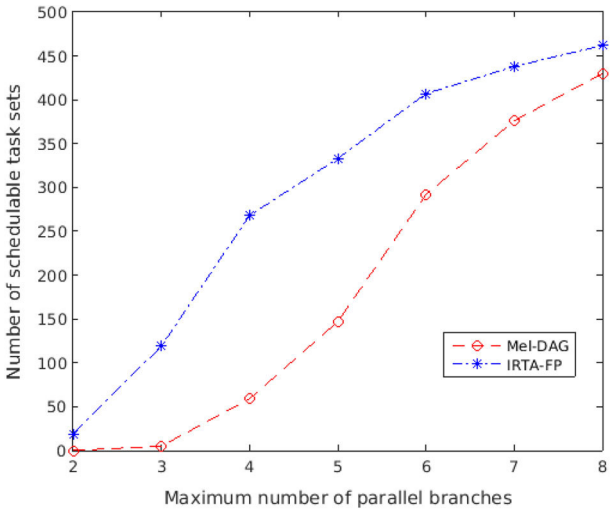


Fig. 13 IRTA-FP varying n_{par}

the results obtained for different types of DAGs, as the probability of adding edges p_{add} between two nodes is increased from 0 to 1 by steps of 0.1. To clarify, $p_{add} = 0$ corresponds to generating NFJ-DAGs, while $p_{add} = 1$ leads to synchronous parallel tasks. In between we have arbitrary DAGs. IRTA-FP attains a solid 40% schedulability improvement over Mel-DAG for any value of p_{add}. Interestingly, such gain is not maximized when IRTA-FP benefits from a more accurate characterization of the carry-out workload (i.e., in the case of NFJ-DAGs). This stresses the importance of exploring the precedence constraints within a DAG when deriving bounds on the interfering workload. Furthermore, we remark that IRTA-FP could achieve better results had we transformed the final DAGs into NFJ instead of considering the original ones.

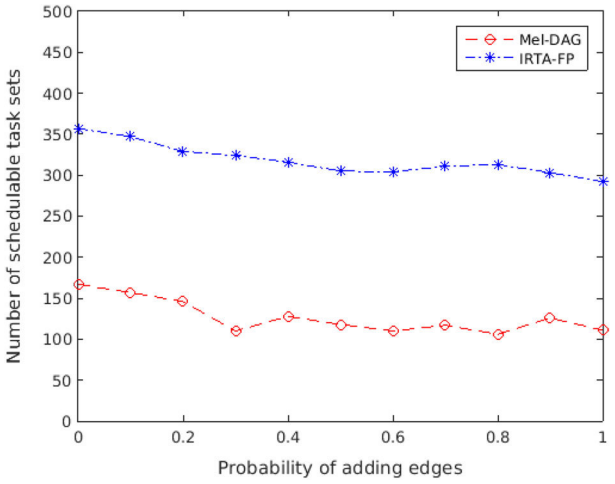


Fig. 14 IRTA-FP varying p_{add}

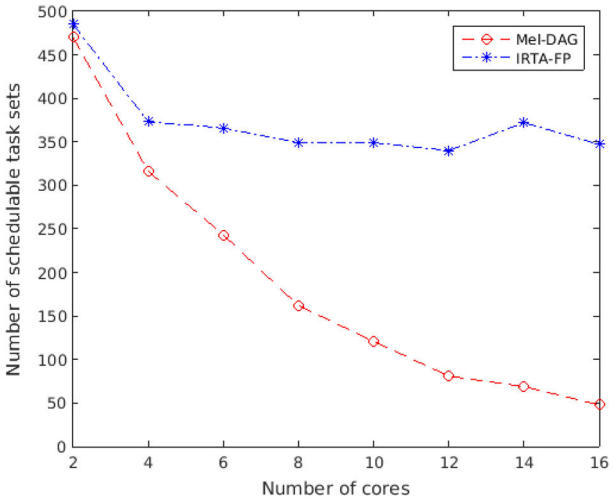


Fig. 15 IRTA-FP varying m

In conjunction with an average increase in the individual critical path lengths, this also justifies the slow degradation when increasing p_{add}.

In Fig. 15, we illustrate how IRTA-FP performs when m varies according to the sequence [2, 4, 6, 8, 10, 12, 14, 16], with U_{tot} and n scaling with m. Mel-DAG degrades for higher values of m, while IRTA-FP maintains a schedulabity ratio around 72%. Such improvement is due to the characterization of the carry-in and carry-out workload distribution. IRTA-FP exploits the internal structure of the DAGs to bound the parallelism of such jobs, hence limiting the number of cores on which they execute for larger m; whereas Mel-DAG assumes that all interfering jobs always use the m cores.

10.2 Evaluation for arbitrary deadlines

We now compare the performance of our response time analysis for DAG tasks with arbitrary deadlines (referred to as IRTA-FP2) to the schedulability test proposed by Parri et al. (2015) for G-DM scheduling (referred to as Parri(16)), which was shown to outperform the tests in Bonifaci et al. (2013), and hence, as far as we know, the only competitor to our test for arbitrary deadline DAG tasks. The number 16 added to Parri's test name denotes the maximum number of iterations allowed for the convergence of the outer loop in their RTA, which in most cases is sufficient to satisfy the convergence of the analysis, as suggested by the authors. Furthermore, since the analysis in Parri et al. (2015) assumes that multiple jobs of the same DAG tasks may execute in parallel (instead of a job becoming ready only after the previous one completes its execution, as we do), for the sake of fairness, we enforce that no task is assigned with a period smaller than its maximum makespan. That is, $T_i \geq M_i, \forall \tau_i \in \tau$. By default, the deadline D_i is uniformly selected in the interval $[T_i, \alpha_{max} T_i]$, with $\alpha_{max} = 3$ controlling the maximum ratio of D_i/T_i; meaning that $T_i \leq D_i \leq 3T_i$.

Figure 16 reports the number of schedulable tasks sets as a function of the total utilization U_{tot} for $m = 8$. While IRTA-FP2 has a breakdown utilization at $U_{tot} = 7$. For Parri(16) such breakdown happens 10% earlier. Notably, when $U_{tot} \in [5.25, 6.75]$, IRTA-FP2 greatly outperforms Parri(16), with a schedulability gain peaking at 75%. This suggests that the way we handle the multiple interfering jobs carried-in by the higher priority tasks largely compensates the handicap on the self-interference component due to the different runtime assumptions.

In order to study the effectiveness of both approaches for different values of D_i, we varied α_{max} in the range [1, 5]. The results are depicted in Fig. 17 for constant values of U_{tot}, n and m. In the case of implicit deadlines (i.e., $\alpha_{max} = 1 \implies D_i = T_i$), Parri(16) performs very poorly, confirming the author's observation that their analysis

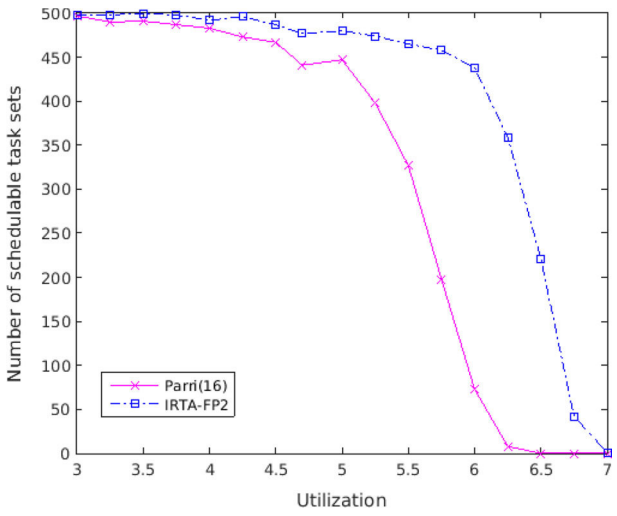


Fig. 16 IRTA-FP2 varying U_{tot}

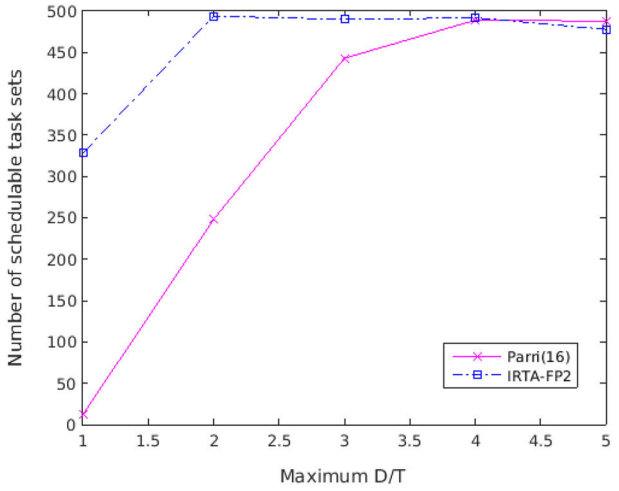


Fig. 17 IRTA-FP2 varying α_{max}

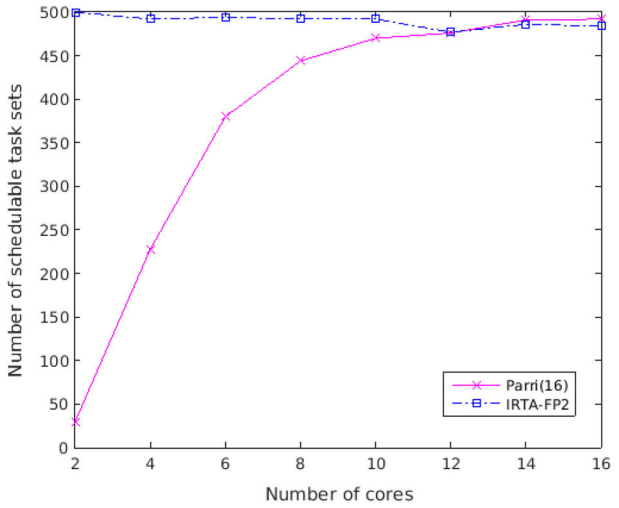


Fig. 18 IRTA-FP2 varying m

is specifically tailored for arbitrary deadlines and as such is overly pessimistic for more restrictive models. On the other hand, IRTA-FP2 is able to schedule 328 task sets as it was already witnessed in the constrained deadline case studied above. As α_{max} is increased, both tests rapidly achieve nearly full schedulablity. It is worth noting that larger values of D_i strongly benefit Parri(16) since they assume that several jobs of the same task can execute in parallel, whereas in IRTA-FP2 assumes that a job cannot start executing before its preceding job has been completed.

In Fig. 18, we show the schedulability results as a function of the number of cores m. Both tests are robust to platforms with increased parallelism, although IRTA-FP2 succeedes in scheduling most task sets for any value of m, Parri(16) requires $m \geq 12$

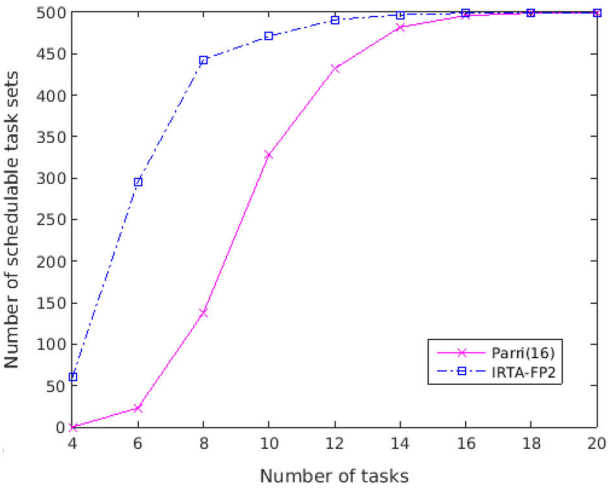


Fig. 19 IRTA-FP2 varying n

to perform similarly. Finally, Fig. 19 illustrates how IRTA-FP2 performs when the number of tasks n is varied according to the sequence [2, 4, 6, 8, 10, 12, 14, 16]. IRTA-FP2 substantially outperforms Parri(16) when $n < 14$, with an average schedulability improvement close to 35%. Nevertheless, both approaches are indistinguishable when the amount of tasks is at least twice the number of cores. From these last sets of experiments, we can conclude that the workload distributions derived to characterize the carry-in and carry-out jobs are also effective for the analysis of DAG tasks with arbitrary deadlines.

11 Conclusions

With the ubiquity of massively parallel architectures, it is expected that conventional real-time applications will increasingly exhibit general forms of parallelism. In this paper, we studied the sporadic DAG model under G-FP scheduling. Motivated by the fact that a poor characterization of the higher priority interfering workload leads to pessimistic analysis of parallel task systems, we presented new techniques to model the worst-case carry-in and carry-out workload. These techniques exploit both the internal structure and worst-case execution patterns of the DAGs. Following a sliding window strategy that leverages from such workload characterization, we then derived a schedulability analysis to compute an improved upper-bound on the WCRT of each DAG task. Experimental results not only attest the theoretical dominance of the proposed analysis over its state-of-the-art counterpart (in the constrained deadline case), but also showed that its effectiveness is independent of the number of cores and it substantially tightens the schedulability of DAG tasks on multiprocessor systems for both constrained and arbitrary deadline cases.

As future work, we plan to better characterize the self interfering workload as well as the interference generated by body jobs. We believe that most of the pessimism

remaining in the analysis is located in those two terms. Furthermore, we plan to perform an extensive comparison between global and partitioned scheduling. However, such comparison would require to first develop an efficient partitioning scheme for DAG tasks. Although analyses for partitioned DAGs exist (Fonseca et al. 2016), there is no algorithm for deciding which node of the DAG should be assigned to which core while maximizing the schedulability of the system.

Finally, similar to what was achieved by Melani et al. (2017), we are considering extending our work, and more particularly the workload distribution characterization presented in this paper, to G-EDF. We expect that the poor performance of G-EDF reported by the authors of Melani et al. (2017) may be attenuated when the carry-in and carry-out interfering workloads are modeled more accurately as it was done in this paper for G-FP scheduling.

Acknowledgements This work was partially supported by National Funds through FCT/ MCTES (Portuguese Foundation for Science and Technology) and co-financed by ERDF (European Regional Development Fund) under the PT2020 Partnership, within the CISTER Research Unit (CEC/04234).

References

Andersson B, de Niz D (2012) Analyzing global-edf for multiprocessor scheduling of parallel tasks. In: Principles of distributed systems, lecture notes in computer science, vol 7702, pp 16–30

Baker TP (2003) Multiprocessor edf and deadline monotonic schedulability analysis. In: RTSS'03, pp 120–129

Baruah S (2014) Improved multiprocessor global schedulability analysis of sporadic dag task systems. In: ECRTS'14, pp 97–105

Baruah S, Bonifaci V, Marchetti-Spaccamela A (2015) The global edf scheduling of systems of conditional sporadic dag tasks. In: ECRTS'15

Baruah SK, Bonifaci V, Marchetti-Spaccamela A, Stougie L, Wiese A (2012) A generalized parallel task model for recurrent real-time processes. In: RTSS'12, pp 63–72

Bini E, Buttazzo GC (2005) Measuring the performance of schedulability tests. Real-Time Syst 30(1):129–154

Board OAR (2013) OpenMP application program interface version 4.0 http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf

Bonifaci V, Marchetti-Spaccamela A, Stiller S, Wiese A (2013) Feasibility analysis in the sporadic dag task model. In: ECRTS'13

Chwa HS, Lee J, Phan KM, Easwaran A, Shin I (2013) Global edf schedulability analysis for synchronous parallel tasks on multicore platforms. In: ECRTS'13, pp 25–34

Fonseca J, Nélis V, Raravi G, Pinho LM (2015) A multi-dag model for real-time parallel applications with conditional execution. In: SAC'15

Fonseca J, Nelissen G, Nélis V (2017) Improved response time analysis of sporadic dag tasks for global fp scheduling. In: Proceedings of the 25th international conference on real-time networks and systems, pp 28–37. ACM

Fonseca J, Nelissen G, Nelis V, Pinho LM (2016) Response time analysis of sporadic dag tasks under partitioned scheduling. In: SIES'16

González-Escribano A, Van Gemund AJC, Cardeñoso Payo V (2002) Mapping unstructured applications into nested parallelism. In: VECPAR'02, pp. 407–420

Guan N, Stigge M, Yi W, Yu G (2009) New response time bounds for fixed priority multiprocessor scheduling. In: 30th IEEE real-time systems symposium, pp 387–397

He X, Yesha Y (1987) Parallel recognition and decomposition of two terminal series parallel graphs. Inf. Comput. 75(1):15–38

Jiang X, Guan N, Long X, Yi W (2017) Semi-federated scheduling of parallel real-time tasks on multiprocessors. In: RTSS'17

Lakshmanan, K., Kato, S., Rajkumar, R (2010) Scheduling parallel real-time tasks on multi-core processors. In: RTSS'10, pp 259–268
Li J, Agrawal K, Lu C, Gill CD (2013) Analysis of global EDF for parallel tasks. In: ECRTS'13, pp 3–13
Li J, Chen J, Agrawal K, Lu C, Gill CD (2014) Analysis of federated and global scheduling for parallel real-time tasks. In: ECRTS'14, pp. 85–96
Maia C, Bertogna M, Nogueira L, Pinho LM (2014) Response-time analysis of synchronous parallel tasks in multiprocessor systems. In: RTNS'14, pp 3–12
Melani A, Bertogna M, Bonifaci V, Marchetti-Spaccamela A, Buttazzo GC (2015) Response-time analysis of conditional dag tasks in multiprocessor systems. In: ECRTS'15, pp 211–221
Melani A, Bertogna M, Bonifaci V, Marchetti-Spaccamela A, Buttazzo GC (2017) Response-time analysis of conditional dag tasks in multiprocessor systems. IEEE Trans Comput 66(2):339–353
Nelissen G, Berten V, Goossens J, Milojevic D (2012) Techniques optimizing the number of processors to schedule multi-threaded tasks. In: ECRTS, pp 321–330
Parri A, Biondi A, Marinoni M (2015) Response time analysis for g-edf and g-dm scheduling of sporadic dag-tasks with arbitrary deadline. In: RTNS'15, pp 205–214
Pathan R, Voudouris P, Stenstrm P (2018) Scheduling parallel real-time recurrent tasks on multicore platforms. IEEE Trans Parallel Distrib Syst 29(4):915–928
Qamhieh M, Fauberteau F, George L, Midonnet S (2013) Global edf scheduling of directed acyclic graphs on multiprocessor systems. In: RTNS, pp 287–296
Saifullah A, Agrawal K, Lu C, Gill C (2011) Multi-core real-time scheduling for generalized parallel task models. In: RTSS'11, pp 217–226
Saifullah A, Ferry D, Li J, Agrawal K, Lu C, Gill C (2014) Parallel real-time scheduling of dags. IEEE Trans Parallel Distrib Syst 25(12):3242–3252
Saifullah A, Li J, Agrawal K, Lu C, Gill C (2013) Multi-core real-time scheduling for generalized parallel task models. Real-Time Syst 49(4):404–435
Valdes J, Tarjan RE, Lawler EL (1979) The recognition of series parallel digraphs. In: STOC'79, pp 1–12

José Fonseca holds a BSc (2010) and a MSc (2012) both in Computer Engineering from the School of Engineering of the Polytechnic Institute of Porto (ISEP). Currently, he is pursuing a PhD in Electrical and Computer Engineering at the Faculty of Engineering of the University of Porto (FEUP). Since he joined CISTER Research Unit in February 2012, his main research interests include real-time operating systems, parallel programming models, real-time scheduling theory and multi-/many-core platforms.

Geoffrey Nelissen is a research scientist at CISTER (Research Centre in Real-Time and Embedded Computing Systems), a research centre co-hosted by the Faculty of Engineering of the University of Porto (FEUP) and the School of Engineering (ISEP) of the Polytechnic Institute of Porto. Prior to joining CISTER, he studied in Brussels at the Université Libre de Bruxelles (ULB), where he earned his PhD in January 2013 and his master degree in electrical engineering in 2008. His research activities are mostly related to the modelling and analysis of real-time and safety critical embedded systems. His research interests span all theoretical and practical aspects of real-time embedded systems design with a particular emphasis on the analysis and configuration of real-time parallel applications on multicore and distributed platforms.

Vincent Nélis received his PhD degree in 2010, at the age of 25, at the Computer Science Department of the Université Libre de Bruxelles, Belgium. Since then, he has graduated 2 PhD students as main supervisor (both received the highest distinction for their thesis) and one PhD student as co-supervisor. He is currently the main supervisor of a third PhD student and co-supervises a second PhD student. He has published more than 40 papers with about 50 different co-authors in international journals, conferences, and workshops. He received 7 awards, contributed to 9 R&D projects, led a Work Package in a European FP7 STREP project, chaired 3 international workshops, and he has been member of the program committee of more than 30 international journals, conferences and workshops. Throughout his short career he has given countless presentations and attended numerous meetings, from simple collaborations with academic peers to project meetings, technical and review meetings. Currently, his main research interest is in developing methods and tools to derive all sorts of timing estimates for applications running on multicore platforms.