CISTER
Research Centre in
Real-Time & Embedded
Computing Systems

Conference Paper

Improved Response Time Analysis of Sporadic DAG Tasks for Global FP Scheduling

José Fonseca
Geoffrey Nelissen
Vincent Nélis

CISTER-TR-170901

2017/10/04

Improved Response Time Analysis of Sporadic DAG Tasks for Global FP Scheduling

José Fonseca, Geoffrey Nelissen, Vincent Nélis

*CISTER Research Centre

Polytechnic Institute of Porto (ISEP-IPP)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8321159

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

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

Abstract

One of the major sources of pessimism in the response time analysis of globally schedule d real-time tasks is the computation of the 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 fie d-priority scheduling (G-FP) of sporadic real-time tasks, each one mo dele d by a directed acyclic graph (DAG) of parallel subtasks. We present a response time analysis technique based on the concept of problem window. We propose two novel techniques to derive tight upper-bounds on the workload produced by the carry-in and carry-out jobs of the interfering tasks, by taking into account the precedence constraints between their subtasks. We show that with these new upper-bounds, the proposed schedulability test does not only theoretically dominate state-of-the-arttechniques but also offers significant improvements on the schedulability of DAG tasks for randomly generate d task sets.

Improved Response Time Analysis of Sporadic DAG Tasks for Global FP Scheduling

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

ABSTRACT

One of the major sources of pessimism in the response time analysis of globally scheduled real-time tasks is the computation of the 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 scheduling (G-FP) of sporadic real-time tasks, each one modeled by a directed acyclic graph (DAG) of parallel subtasks. We present a response time analysis (RTA) technique based on the concept of problem window. 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, by taking into account the precedence constraints between their subtasks. We show that with these new upper-bounds, the proposed schedulability test does not only theoretically dominate state-of-the-art techniques but also offers significant improvements on the schedulability of DAG tasks for randomly generated task sets.

1 INTRODUCTION

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 [2–8, 12–16]. In this paper, we study the sporadic DAG task model introduced in [4] 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. [15] represents the first attempt at analyzing the schedulability of a set of general sporadic DAG tasks with a G-FP scheduling policy. Their RTA is based on the concept of problem window developed originally by Baker in [1]. 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 [15] 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, such abstraction leads to a significant pessimism in the schedulability analysis. Motivated by this observation, 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.

Contributions. We present two novel techniques that exploit the internal structure of the DAGs to upper-bound the worst-case interference between tasks. We then use these new upper-bounds to refine traditional schedulability analysis techniques and we show that the resulting schedulability test not only dominates the state-of-the-art analysis [15] but it is also robust to systems with increased number of cores.

Related work. The real-time community has been devoting significant attention to the problem of scheduling parallel tasks on multiprocessor platforms. Parallel task models have been proposed to cope with the different forms of task parallelism introduced by widely used parallel programming models. Among the models that impose more restrictions on the parallel features, we highlight both the fork-join model [12] and the synchronous parallel model [6, 14, 16]. For the DAG model, most of the work published in the past few years target G-EDF [2, 4, 5, 13], partitioned scheduling [8] and federated scheduling strategies [7]. Recently, the DAG model has been extended to support conditional constructs, allowing a parallel task to experience different flows of execution [3, 15].

2 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. We consider that τ is comprised of constrained-deadline tasks, i.e., $D_i \leq T_i, \forall i$.

Each job of 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

RTNS '17, October 4–6, 2017, Grenoble, France

ACM ISBN 978-1-4503-5286-4/17/10…$15.00
https://doi.org/10.1145/3139258.3139288

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 $\forall v_j \in \lambda \setminus \{v_{n_i}\}$, $(v_j, v_{j+1}) \in E_i$.*

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 (for instance, $m \geq n_i$), 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. Since the length of every critical path is L_i, ℓ time units after its release, a job of τ_i must still execute during at least $\max\{0,\ L_i - \ell\}$ time units to complete. Hence, at most $W_i - \max\{0,\ L_i - \ell\}$ units executed in the interval of length ℓ. □

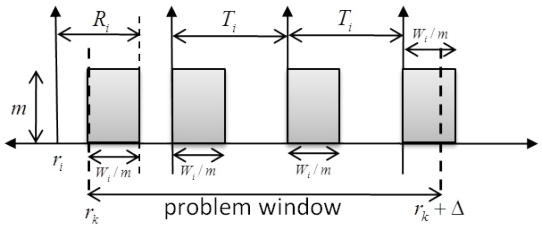


Figure 1: Worst-case interfering workload of a HP task τ_i.

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

3 BACKGROUND

In this section, we summarize the RTA introduced by Melani et al. [15] as it sets the foundations for the schedulability analysis proposed in the upcoming sections. Although their work uses a more general task model, known as "conditional DAG tasks", empirical evaluation in [15] shows that it is also state-of-the-art for the non-conditional DAG task model used in this paper.

A key challenge in the RTA of globally scheduled multiprocessor systems is to compute an upper-bound on the interference between tasks. That is, when analyzing the response time of a task τ_k, the analysis must find an upper-bound on the delays that may be incurred by every subtask *in the critical paths of* τ_k. These delays are the intervals of time during which those subtasks are ready but cannot execute because all the cores are busy executing tasks of higher-priority than τ_k (inter-task interference), or subtasks from τ_k that are *not* part of its critical paths (self interference).

Regarding the self-interference, in a constrained-deadline system two jobs of a same task τ_k cannot interfere with each other, because one must finish before the next one is released. Therefore, the self interfering workload denoted $I_{k,k}$, is independent of the response time of τ_k. Furthermore, because of the absence of priorities at the subtask-level, every subtask *that is not part of a critical path of* τ_k may potentially contribute to the overall response time of τ_k and thus to its self interfering workload $I_{k,k}$. In other words, an upper-bound on the self interfering workload $I_{k,k}$ of τ_k is given by

$$I_{k,k} \leq W_k - L_k \tag{1}$$

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, the computation of the maximum cumulative execution time of all the subtasks of a higher-priority task τ_i that are released within these Δ time units divides the window in three consecutive portions: the carry-in, the body, and the carry-out (see Fig. 1). The first portion (i.e., the carry-in) starts at the release time r_k of the analyzed task τ_k. It assumes that one job of τ_i has been released before the beginning of the window but has not yet completed its execution (its "carry-in" job). The carry-in portion finishes at the release of the next job of τ_i. Then, all the subsequent job releases that are fully contained in the window constitute the body portion (they are called the body jobs of τ_i). Finally, the carry-out portion is defined as the remaining subinterval of time at the end of the window, and its start is coincident with the release of the last job of τ_i that still executes within the time window (its "carry-out" job).

In [15], the authors formulated a generic upper-bound on the interfering workload of a task τ_i on a task τ_k. This upper-bound considers a time window of length $\Delta = R_k$ (the response time of τ_k) and works for any work-conserving algorithm. It is given by $\mathcal{W}_i(R_k) = \left\lfloor \frac{R_k+R_i-W_i/m}{T_i} \right\rfloor W_i + \min\left(W_i, m((R_k + R_i - W_i/m) \bmod T_i)\right)$. This upper-bound 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 produced 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, (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 the fixed-point iteration of the following expression [15]:

$$R_k = L_k + \frac{1}{m}(W_k - L_k) + \frac{1}{m} \sum_{\forall i<k} \mathcal{W}_i(R_k) \tag{2}$$

4 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 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 which 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.

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 [15], 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. 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 $r_i \leq r_k$ finishes its execution at time $r_i + R_i$ and executes as much

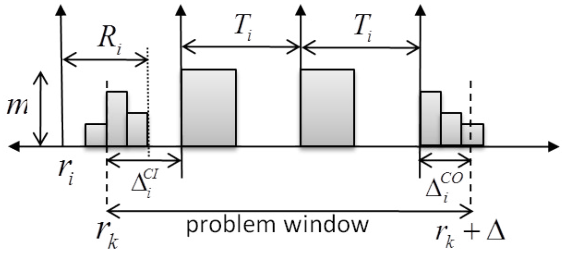


Figure 2: Worst-case scenario for the interfering workload computation of τ_i on τ_k.

workload as possible as late as possible (to maximize its workload contribution to the 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 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 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. We seek to derive (i) an upper-bound on the workload done by the carry-in job as a function of Δ_i^{CI}, ii) an upper-bound on the workload done by the 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*.

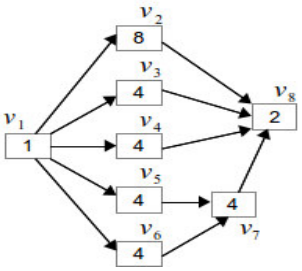
Definition 5 (Workload Distribution). *For a given task τ_i and a given schedule S of τ_i's subtasks, the workload distribution $\mathcal{WD}_i^S = [B_1, \ldots, B_\ell]$ describes S as a sequence of consecutive blocks. Each block $B_b \in \mathcal{WD}_i^S$ is a tuple (w_b, h_b) with the interpretation 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 it is required for S to be a valid schedule of G_i. Also, according to Def 5, every interfering job of a task τ_i is modeled in [15] with a workload distribution $\mathcal{WD}_i^S$ that comprises only one block $B_1 = (\frac{W_i}{m}, m)$.

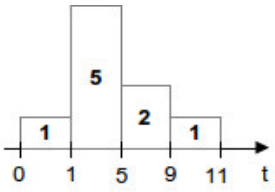
5 CARRY-IN

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. Remember that the carry-in job is the first job of τ_i that enters the window of interest such that its release time r_i is earlier than r_k and its deadline falls within $[r_k, r_k + \Delta)$. 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.



(a) DAG example. Numbers in nodes are their WCET.



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

Figure 3: Example for the carry-in interfering workload.

5.1 Upper-bounding the carry-in workload

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. 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:

$$f_j = \begin{cases} C_j & \text{if } v_j \text{ is the source} \\ C_j + \max\limits_{v_h \in pred(v_j)} (f_h) & \text{otherwise} \end{cases} \quad (3)$$

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. (3), 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

$$AS(t) = \sum_{v_j \in V_i} actv(v_j, t) \quad (4)$$

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

$$actv(v_j, t) = \begin{cases} 1 & \text{if } t \in \left[f_j - C_j, f_j\right) \\ 0 & \text{otherwise} \end{cases} \quad (5)$$

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

Based on both the workload distribution $\mathcal{WD}_i^{UCI}$ and the WCRT R_i estimated by Eq. (2), we compute an upper-bound on the interfering workload produced by τ_i's carry-in job 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). Therefore, the carry-in job of τ_i is released at $r_k + \Delta_i^{CI} - T_i$ and completes at most at $r_k + \Delta_i^{CI} - T_i + R_i$.

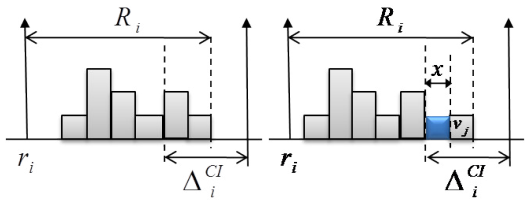


Figure 4: Interference (blue block) on $\mathcal{WD}_i^{UCI}$ critical path.

Next, we compute the cumulative workload found in the last $\Delta_i^{CI} - (T_i - R_i)$ time units of the workload distribution $\mathcal{WD}_i^{UCI}$. 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 $\Delta_i^{CI} - (T_i - R_i)$. Therefore, under the schedule UCI, the maximum workload produced by τ_i's carry-in job is upper-bounded by the function[1]:

$$CI_i(\mathcal{WD}_i^{UCI}, \Delta_i^{CI}) = \sum_{b=1}^{|\mathcal{WD}_i^{UCI}|} h_b \times \left[\Delta_i^{CI} - T_i + R_i - \sum_{p=b+1}^{|\mathcal{WD}_i^{UCI}|} w_p\right]_0^{w_b} \quad (6)$$

Eq. (6) 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 1. *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. (6) 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. (6) would return 0 since $\Delta_i^{CI} - (T_i - R_i)$ is then smaller than 0.*

We now prove that the workload imposed 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 $(r_k + \Delta_i^{CI} - T_i + R_i)$. Recall that R_i is computed by Eq. (2).

The workload distribution $\mathcal{WD}_i^{UCI}$ 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 as one block just before its completion time at $(r_k + \Delta_i^{CI} - T_i + R_i)$. We prove in Lemmas 2 to 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 execute earlier. In the latter case, it may cause those descendants to finish before (instead of within) the carry-in window and thus reduce the total interfering workload. □

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

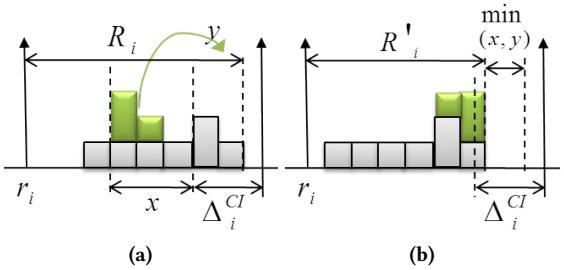


Figure 5: y units of workload (green blocks) of $\mathcal{WD}_i^{UCI}$ are moved in the carry-in window.

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 τ_i's critical path 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 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 all 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.*** *(2). 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.*

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 needs to show that the claim is still true even when the interference exerted on τ_i does not interfere with its critical paths but may delay the execution of other subtasks of τ_i.

The proof is by contradiction. Assume that there is a schedule of τ_i such that y extra units of workload enter the carry-in window $[r_k, r_k + \Delta_i^{CI})$ comparatively to $\mathcal{WD}_i^{UCI}$ by delaying subtasks of τ_i. By the above discussion, those 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 x be the maximum distance between the execution instant in $\mathcal{WD}_i^{UCI}$ of any of the delayed subtasks, and the beginning of that carry-in window (see Fig. 5a). 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 since by assumption the schedule length is not increased. Therefore, we have at least

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

workload that does 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. (2), and Eq. (2) 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

$$\begin{aligned} R_i' &\leq R_i - \left(\frac{(m-1) \times x + y}{m}\right) = R_i - \left(\frac{m \times y}{m} + \frac{(m-1) \times (x-y)}{m}\right) \\ &\leq R_i - y - \frac{(m-1) \times (x-y)}{m} \end{aligned} \tag{7}$$

We analyse two cases:

- If $y \leq x$, then the last term in (7) is negative 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 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 (7) is positive 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.

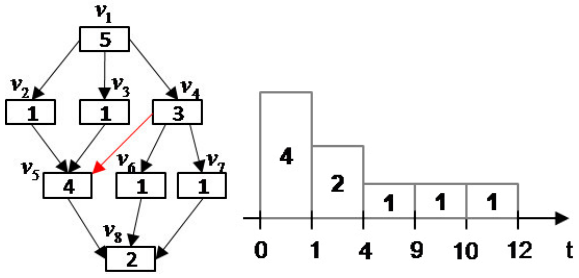
The two cases above prove the claim. □

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 to 4. □

5.2 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 Equation 6, Theorem 2 below shall present an improved upper-bound that is simply the minimum between that given by Equation 6 and that presented in Lemma 5.



(a) DAG example. (b) Workload distribution $\mathcal{WD}_i^{UCO}$.

Figure 6: Running example for the carry-out workload.

Lemma 5. *An 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} 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. □

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. □

6 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 that enters the window of interest such that its release time is earlier than $r_k + \Delta$ and its deadline is after $r_k + \Delta$. 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 dependencies may be immediately resolved and the degree of parallelism in the DAG is potentially increased, leading to more workload at the beginning of the carry-out job.

Example 2. *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 workload.*

Hence, 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).

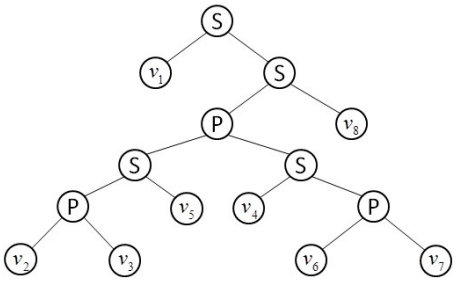


Figure 7: Decomposition tree of the NFJ-DAG in Fig. 6a.

6.1 DAG's maximum parallelism

Computing the maximum degree of parallelism in an arbitrary graph consists in identifying the largest set of subtasks that can execute concurrently. The complexity of computing the maximum parallelism of a DAG is therefore equivalent to the problem known as the *maximum independent set* problem in graph theory [9]. For DAGs in particular, this problem is known to be NP-hard in the general case [9]. Hence, we restrict our attention to a well-structured (less general) type of DAG, which we call "nested fork-join DAG" (NFJ-DAG). We define a NFJ-DAG[2] recursively as follows.

Definition 6 (Nested Fork-Join DAG). *A DAG comprised of two nodes connected by a single edge is NFJ. 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. 6. However, if the edge (v_4, v_5) is removed, then the DAG becomes NFJ.

Many efficient algorithms exist in the literature to identify if a DAG is NFJ [11, 17]. 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 [10]. 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's 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. 6, a DAG is NFJ if and only if it respects the following property.

[2] In graph theory, it is known as *two terminal series parallel digraph* [11].

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

Proof. The property directly follows from Def. 6, 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. 6, 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 its in-degree become equal to 1.
(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 3. *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.*

By Def. 6, 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. in [17] can be used to efficiently build the decomposition tree of any NFJ-DAG. Fig. 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 Equation (8) 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, respectively.

$$par(T_i^U) = \begin{cases} par(T_i^L) \cup par(T_i^R) & \text{if } U \text{ is a P-node} \\ par(T_i^L) & \text{if } U \text{ is a S-node and} \\ & |par(T_i^L)| \geq |par(T_i^R)| \\ par(T_i^R) & \text{if } U \text{ is a S-node and} \\ & |par(T_i^R)| > |par(T_i^L)| \\ \{U\} & \text{otherwise} \end{cases} \quad (8)$$

Eq. (8) 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. (8) 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 $par(.)$ for G_i^{NFJ}'s decomposition tree.

6.2 Upper-bounding the carry-out workload

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 list 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 elements in the list. 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.

Example 4. *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*

Algorithm 1: Constructing $\mathcal{WD}_i^{UCO}$ of 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** $\mathcal{WD}_i^{UCO}$;

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.*

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, that is, the distance between the last release of τ_i in the problem window and the end of the problem window of the analyzed task τ_k (see Fig. 2):

$$\Delta_i^{CO} = (r_k + \Delta) - \left(r_i + \left\lfloor \frac{(r_k - r_i) + \Delta}{T_i} \right\rfloor \times T_i \right)$$

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:

$$CO_i(\mathcal{WD}_i^{UCO}, \Delta_i^{CO}) = \sum_{b=1}^{|\mathcal{WD}_i^{UCO}|} h_b \times \left[\Delta_i^{CO} - \sum_{p=1}^{b-1} w_p \right]_0^{w_b} \quad (9)$$

Example 5. *If $\Delta_i^{CO} = 3$ and $\mathcal{WD}_i^{UCO}$ is given by the workload distribution presented in Fig. 6b, then Eq.* (9) *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 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. Due to space limitation, we provide a proof sketch. 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 Alg. 1, and because Alg. 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. □

6.3 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 when any of the removed edges 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).

As stated by Cor. 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 Cor. 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 Lem. 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 $CO_i(\mathcal{WD}_i^{UCO}, \Delta_i^{CO})$ (Th. 3) and $W_i - \max\{0, L_i - \Delta_i^{CO}\}$ (Lem. 6) are also upper-bounds on W_i^{CO}, so is the minimum between the three values. □

7 SCHEDULABILITY ANALYSIS

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.

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

$$\mathcal{W}_i(\Delta) = \mathcal{W}_i^C(\Delta_i^C) + \max\left\{0, \left\lfloor \frac{\Delta - \Delta_i^C}{T_i} \right\rfloor\right\} \times W_i \quad (10)$$

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. (10), 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.

$$\Delta_i^C = \Delta - \max\left\{0, \left\lfloor \frac{\Delta - L_i}{T_i} \right\rfloor\right\} \times T_i \quad (11)$$

Δ_i^C is given by aligning the problem window with the end of the carry-out job of τ_i (which is no less than L_i time units long according to Cor. 1) and removing all the body jobs of τ_i from the problem window Δ. 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.* (11).

Proof. Note that $L_i \leq \Delta_i^C < L_i + T_i$ when computed with Eq. (11) (assuming that $\Delta \geq L_i$). 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 $\Delta_i^C < L_i + T_i$, $L_i \leq T_i$ and each body job execute 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 Equation (11), the computed value of Δ_i^C is Δ minus an integer multiple of T_i and thus, when injecting Equation (11) into Equation (10), 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., $\Delta_i^C \geq L_i$), 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 [14]. 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 $\mathcal{WD}_i^{UCI}$; or (ii) the problem window ends at the completion of a block in $\mathcal{WD}_i^{UCO}$. It was proven in [14], 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. (10) in Eq. (2), a schedulability condition for task τ_k is stated in the next theorem.

Algorithm 2: Computing $\mathcal{W}_i^C$.

Input : Δ_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.

$\mathcal{W}_i^C \leftarrow CO_i(\mathcal{WD}_i^{UCO}, \Delta_i^C)$;
$x1 \leftarrow T_i - R_i$;
foreach $(w_b, h_b) \in \mathcal{WD}_i^{UCI}$ *in reverse order* **do**
  $x1 \leftarrow x1 + w_b$;
  $x2 \leftarrow \Delta_i^C - x1$;
  $\mathcal{W}_i^C \leftarrow max\{\mathcal{W}_i^C, CI_i(\mathcal{WD}_i^{UCI}, x1) + CO_i(\mathcal{WD}_i^{UCO}, x2)$;
end
$\mathcal{W}_i^C \leftarrow max\{\mathcal{W}_i^C, CI_i(\mathcal{WD}_i^{UCI}, \Delta_i^C)$;
$x2 \leftarrow 0$;
foreach $(w_b, h_b) \in \mathcal{WD}_i^{RCO}$ **do**
  $x2 \leftarrow x2 + w_b$;
  $x1 \leftarrow \Delta_i^C - x2$;
  $\mathcal{W}_i^C \leftarrow max\{\mathcal{W}_i^C, CI_i(\mathcal{WD}_i^{UCI}, x1) + CO_i(\mathcal{WD}_i^{UCO}, x2)$;
end
return $\mathcal{W}_i^C$;

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.

8 EXPERIMENTAL EVALUATION

The analysis presented in this paper has been implemented within the MATLAB framework released by the authors of [15]. We follow the same technique in [11] and [15] to generate random task sets composed of DAG tasks. Each DAG in the 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 as a maximum *depth* that limits the number of nested forks. Additionally, the number of parallel branches leaving from a fork node is uniformly chosen within $[2, n_{par}]$. Finally, a general DAG is obtained by randomly adding directed edges between arbitrary 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}. 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 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 $[\beta, W_i/M_i]$. The relative deadline D_i is set to its period T_i. When the number of tasks is not specified, we keep generating and adding new tasks to the task set until the target total utilization U_{tot} is met. U_{tot} is achieved by adjusting the period of the last task added to the system. Priorities are assigned following the Rate Monotonic policy.

We compare our response time analysis (referred to as IRTA-FP) to the test described in [15] (referred to as Mel-DAG). 500 task sets

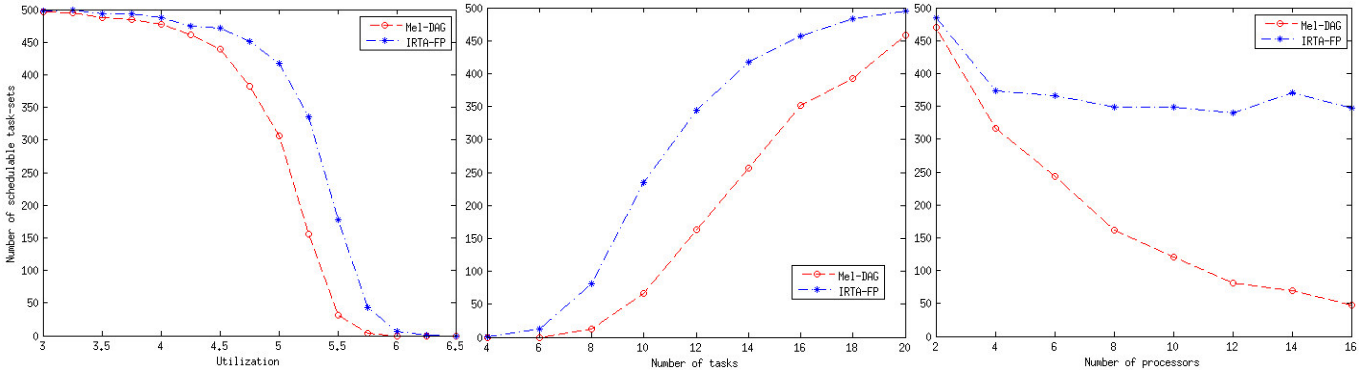


Figure 8: Number of schedulable task sets when varying U_{tot}, n and m.

were generated for each tested configuration. In all experiments reported herein, we have set $p_{par} = 0.8$, $p_{term} = 0.2$, $depth = 2$, $n_{par} = 5$, $p_{add} = 0.2$ and $\beta = 0.035 \times m$. These settings lead to a rich variety of internal DAG structures: we observed 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.

In the first set of experiments, the system utilization U_{tot} was varied in $[1, m]$ by steps of 0.25. The left-most plot of Fig. 8 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.

The center plot of Fig. 8 reports the schedulability as a function of the number of tasks n. n was varied from 4 to 20 by step of 2, while m was set to 8 and the platform utilization to 70%. We used UUnifast to derived individual task utilizations (and consequently their period) for a fixed value of n. IRTA-FP outperforms Mel-DAG for any value of n, although both tests converge to full schedulability for larger n. Intuitively, it is easier to schedule many light tasks than few heavy tasks.

The right-most plot of Fig. 8 illustrates how IRTA-FP performs when m varies according to the sequence $[2, 4, 6, 8, 10, 12, 14, 16]$, with $U_{tot} = 0.7m$ and $n = 1.5m$. 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 jobs: 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.

9 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, 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.

ACKNOWLEDGMENTS

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

[1] T. P. Baker. Multiprocessor EDF and deadline monotonic schedulability analysis. In *RTSS'03*. 120–129.
[2] Sanjoy Baruah. Improved multiprocessor global schedulability analysis of sporadic DAG task systems. In *ECRTS'14*. 97–105.
[3] Sanjoy Baruah, Vincenzo Bonifaci, and Alberto Marchetti-Spaccamela. The Global EDF Scheduling of Systems of Conditional Sporadic DAG Tasks. In *ECRTS'15*. 10.
[4] Sanjoy K. Baruah, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, Leen Stougie, and Andreas Wiese. A Generalized Parallel Task Model for Recurrent Real-time Processes. In *RTSS'12*. 63–72.
[5] V. Bonifaci, A Marchetti-Spaccamela, S. Stiller, and A Wiese. Feasibility Analysis in the Sporadic DAG Task Model. In *ECRTS'13*.
[6] Hoon Sung Chwa, Jinkyu Lee, Kieu-My Phan, A Easwaran, and Insik Shin. Global EDF Schedulability Analysis for Synchronous Parallel Tasks on Multicore Platforms. In *ECRTS'13*. 25–34.
[7] J. Li et al. Analysis of Federated and Global Scheduling for Parallel real-Time tasks. In *ECRTS'14*. 85–96.
[8] J. Fonseca, G. Nelissen, V. Nelis, and L. M. Pinho. Response time analysis of sporadic DAG tasks under partitioned scheduling. In *SIES'16*.
[9] Michael R. Garey and David S. Johnson. 1990. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co.
[10] Arturo González-Escribano, Arjan J. C. Van Gemund, and Valentín Cardeñoso Payo. Mapping Unstructured Applications into Nested Parallelism. In *VECPAR'02*. 407–420.
[11] Xin He and Yaacov Yesha. 1987. Parallel recognition and decomposition of two terminal series parallel graphs. *Information and Computation* 75, 1 (1987), 15–38.
[12] Karthik Lakshmanan, Shinpei Kato, and Ragunathan Rajkumar. Scheduling Parallel Real-Time Tasks on Multi-core Processors. In *RTSS'10*. 259–268.
[13] Jing Li, Kunal Agrawal, Chenyang Lu, and Christopher D. Gill. Analysis of Global EDF for Parallel Tasks. In *ECRTS'13*. 3–13.
[14] Cláudio Maia, Marko Bertogna, Luís Nogueira, and Luís Miguel Pinho. Response-Time Analysis of Synchronous Parallel Tasks in Multiprocessor Systems. In *RTNS'14*. 3–12.
[15] Alessandra Melani, Marko Bertogna, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, and Giorgio C. Buttazzo. Response-Time Analysis of Conditional DAG Tasks in Multiprocessor Systems. In *ECRTS'15*. 211–221.
[16] Abusayeed Saifullah, Kunal Agrawal, Chenyang Lu, and Christopher Gill. Multi-core Real-Time Scheduling for Generalized Parallel Task Models. In *RTSS'11*. 217–226.
[17] Jacobo Valdes, Robert E. Tarjan, and Eugene L. Lawler. The Recognition of Series Parallel Digraphs. In *STOC'79*. 1–12.