Counting Priority Inversions: Computing Maximum Additional Core Requests of DAG Tasks

Morteza Mohaqeqi, Gaoyang Dai, Wang Yi
Uppsala University, Sweden
{morteza.mohaqeqi|gaoyang.dai@it.uu.se|yi}@it.uu.se

***Abstract*—Many parallel real-time applications can be modeled as DAG tasks. Guaranteeing timing constraints of such applications executed on multicore systems is challenging, especially for the applications with non-preemptive execution blocks. The existing approach for timing analysis of such tasks with sporadic release relies on computing a bound on the interfering workload on a task, which depends on the number of *priority inversions* the task may experience. The number of priority inversions, in turn, is a function of the total number of additional cores a task instance may request after each node spawning.**

In this paper, we show that the previously proposed polynomial-time algorithm to compute the maximum number of additional core requests of a DAG is not correct, providing a counter example. We show that the problem is in fact NP-hard. We then present an ILP formulation as an exact solution to the problem. Our evaluations show that the problem can be solved in a few minutes even for DAGs with hundreds of nodes.

I. Introduction

Multicore processors are increasingly employed in real-time applications. The workload of many real-time software running on multicore chips can be naturally modeled as directed acyclic graphs (DAG), where nodes represent sequential computation units, also called subtasks, and edges represent functional dependencies between subtasks. OpenMP [6] programs, robotic software running on ROS [3], and automotive task chains [10] are instances of such applications.

In this paper, we consider a DAG task model, where nodes of a DAG execute non-preemptively, while a task may be preempted between the execution of any two nodes, i.e., at node boundaries. This preemption scheme provides a balanced trade-off: It avoids expensive preemption overheads of fully preemptive scheduling, while trying to rule out long blocking delays of fully non-preemptive execution.

Timing analysis of the considered task model is particularly challenging because a task can be interfered even by lower priority tasks due to non-preemptable execution of the nodes, causing *priority inversion*. It has been shown in [9] that the number of priority inversions a DAG task may experience, which is needed for response-time analysis, depends on the total number of additional cores the task may request after each node spawning.

Previously, a polynomial-time algorithm has been proposed to compute the maximum number of total additional cores a DAG task instance may request [9]. It turns out, however, that the algorithm does not necessarily compute the correct value, implying an unsafe response-time analysis.

Contributions. The key contributions of this paper are as follows. We prove the existing method for computing the additional core requests of a DAG task incorrect, giving a counter example. We then prove the problem NP-hard. We present an exact solution to the problem using an ILP formulation. While the solution is of a worst-case exponential complexity, it is solved in a few minutes using standard solvers even for DAG models with hundreds of nodes and edges.

Paper structure. We introduce the system model, and formally define the problem in Section II. A counter example for the existing algorithm is given in Section III. To simplify the subsequent discussions, Section IV shows that it suffices to focus only on the task instances where the finish of no two nodes coincide. Section V proves the problem hardness. Section VI presents an exact solution, which is evaluated using a parallel benchmark application in Section VII. The paper concludes in Section VIII.

II. Preliminaries

This section introduces the system model, reviews the notion of priority inversion, and formally defines the targeted problem.

A. DAG Tasks

We assume a set of real-time tasks running on a multicore processor. The structure of each task is described by a DAG $G = (V, E)$ where V denotes the set of nodes, representing subtasks, and $E \subseteq V \times V$ the set of edges, representing precedence constraints. For any edge $(u, v) \in E$, u is a *predecessor* of v, and v is a *successor* of u. The set of predecessors and successors of $v \in V$ is denoted by $Pred(v)$ and $Succ(v)$, respectively. A node v is *source* if $Pred(v) = \emptyset$. Each node represents a sequential execution unit, i.e., it can be running on at most one core at a time.

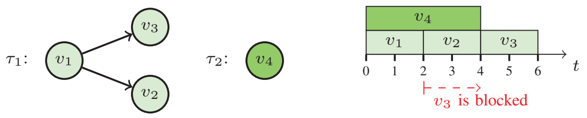
Each task releases a sequence of instances, called jobs. With the release of a job, the corresponding source nodes are released, i.e., they can be executed. A non-source node, on the other hand, is released when all of its predecessors are finished. The execution of each node is assumed to take a positive amount of time. We assume that at most one job of each task exists in the system at any time.

The tasks are scheduled under a priority-driven scheduler which dispatches the nodes according to the priority of the task they belong to. Our focus is on work-conserving scheduling, i.e., no core is left idle as long as there exist some pending released node. We assume non-preemptive execution of the nodes, i.e., when a node starts execution, it cannot be preempted until it is finished. This scheduling paradigm is sometimes referred to as *limited-preemptive* [9].

B. Priority Inversion

In the limited-preemptive execution, preemption of the nodes is not allowed. Therefore, a node that is released when all the cores are busy is blocked even if its priority is higher than the running nodes. The situation in which a higher-priority task (or subtask) is blocked by a lower-priority one is called *priority inversion* [1]. Priority inversion occurs upon the release of a DAG job if all the cores are busy and some of them are running lower-priority task(s). In addition, it may occur when a node is finished and spawns (releases) several new nodes, as in the following example.

Example 1. *Consider the two DAG tasks τ_1 and τ_2 in Fig. 1a, assuming τ_1 to be of a higher priority. Let each task release a job at time $t = 0$, and the execution time of all nodes be 2, except for v_4 which runs for 4 time units. Fig. 1b shows a sample schedule on two cores. At $t = 2$, v_3 is blocked by the lower-priority task τ_2, implying priority inversion.*



(a) Two DAGs (b) A sample schedule

Fig. 1: Priority inversion when new nodes are released.

In Example 1, priority inversion occurs when v_1 spawns two nodes, raising a request for one additional core (note that a core has been running v_1, which now can be used to run a newly released node; hence, the release of two nodes implies a request for *one additional* core). This paper targets computing the largest total number of additional cores a job of a given DAG task may request. This is important because it determines the number of priority inversions, needed for timing analysis of sporadic limited-preemptive DAG tasks [9][1].

C. Number of Additional Core Requests

To determine the number of additional cores a job requests, we need to inspect only the finishing times of the nodes as these are the only points where the job may request extra cores. Based on this observation, we formally define the desired parameter.

Assume a job J released by a DAG task $G = (V, E)$ executed on a multicore processor, where instances of other tasks are possibly running as well. Let f_v denote the finishing time of a node $v \in V$ of J. Further, let $F(t)$ and $R(t)$, respectively, denote the set of nodes finished and released at time t. Formally, $F(t) \overset{\text{def}}{=} \{v \in V | f_v = t\}$, and

$$R(t) \overset{\text{def}}{=} \bigcup_{v \in F(t)} \{u \in Succ(v) | \forall w \in Pred(u) : f_w \leq f_v\}.$$

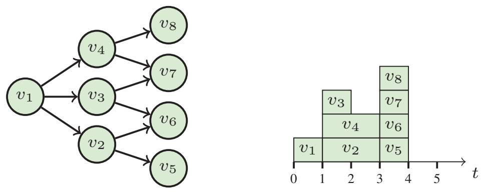
[1]See [9] for the exact relation between the number of priority inversions and additional core requests, and also for the corresponding timing analysis.

The number of additional cores requested by J at time t, denoted by $\delta(t)$, is defined as $\delta(t) \overset{\text{def}}{=} \max(0, |R(t)| - |F(t)|)$. Note that the max operator is used to exclude the negative values. The total number of additional core requests (ACR) of J is then computed as

$$\delta = \sum_{t \in \mathcal{T}} \delta(t), \tag{1}$$

where $\mathcal{T}$ denotes the set of time points at which at least one node of J is finished.

As an example, consider the DAG in Fig. 2a. A possible schedule of a job of this DAG on three cores is seen in Fig. 2b, where each node runs for one time unit except for v_2 and v_4 which run for two time units. The induced ACR is $\max(0, |R(1)| - |F(1)|) + \max(0, |R(3)| - |F(3)|) = 2 + 2 = 4$. (Note that $|R(t)| - |F(t)| \leq 0$ for $t \notin \{1, 3\}$.)



(a) An example DAG task (b) Schedule with $\delta = 4$

Fig. 2: Example of a DAG and additional cores requested.

It is important to note that the ACR of a DAG job depends on how it is scheduled (depending on the scheduler, other tasks in the system, etc.). For example, Fig. 3 shows another schedule for the DAG in Fig. 2 which implies an ACR of $\delta(1) + \delta(5) = 2 + 1 = 3$. (note that here $|R(1)| = 3$, $|F(1)| = 1$, $|R(5)| = 2$, and $|F(5)| = 1$.)

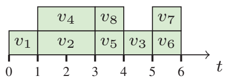


Fig. 3: A different schedule for the DAG in Fig 2a

The goal is to find the maximum ACR a DAG may cause in any possible schedule.

Problem 1 (Max ACR)**.** *Given a DAG task, the problem is to find the maximum total additional cores that a job of the task may request in any possible execution scenario.*

Note that the problem is defined independent of the number of cores and also other tasks; therefore, the search space includes all possible cases.

III. Counter Example

Previously, a polynomial-time algorithm has been proposed [9] as an exact solution to Problem 1 (shown here as Algorithm 1). We show that it does not always compute the true maximum value using a counter example.

Consider the DAG in Fig. 2a. The **for** loop in Line 3 in Algorithm 1 can iterate over the nodes in the following two different orders, revealing different results:

1) $[v_1, v_2, v_4, v_3, \ldots]$: $\delta = 2 + 1 + 1 = 4$
2) $[v_1, v_2, v_3, v_4, \ldots]$: $\delta = 2 + 1 = 3$

It is worth noting that the second result, i.e., 3, is not "safe" as the task can exhibit a larger ACR, as shown in Fig. 2b. Utilizing $\delta = 3$ leads to underestimation of the lower-priority interference and thereby an untrustable response-time analysis.

Algorithm 1 Computing Max. ACR of a DAG $G = (V, E)$ [9]

```
$\delta = 0$
$N = \{\}$
for each $v_i \in V$ do
    $cores = |Succ(v_i)| - 1$
    for each $v_j \in Succ(v_i)$ do
        if $v_j \in N$ then
            $cores = cores - 1$
        else
            if $Succ(v_i) \cap Pred(v_j) \neq \emptyset$ then
                $cores = cores - 1$
            end if
            $N = N \cup \{v_j\}$
        end if
    end for
    $\delta = \delta + max(0, cores)$
end for
```

In the following, we show that the problem is NP-hard (Section V), and thus, it is not subject to a polynomial-time solution unless P = NP. To simplify discussions, we assume that the finish of no two nodes of the job under consideration coincide. This does not lose generality, as shown in Section IV.

IV. Non-coinciding Finishing Times

According to (1), ACR of a job is a function of the finishing time of its nodes. We show that it suffices to consider only those scenarios where no two finishing times coincide.

Lemma 1 (Distinct finishing times)**.** *To find the maximum ACR of a DAG G, it is sufficient to consider only those schedules where the finish time of no two nodes of G coincide.*

Proof. Let S be a schedule where a job of G, say J, exhibits the maximum ACR of G, and some nodes of J finish synchronously. We show that there exists another schedule where no two nodes of J finish at the same time, and the number of ACR is not smaller than that in S.

Let t_1 be a time point at which more than one node finish and v be a node which finishes at t_1. We construct a schedule $\bar{S}$ as follows. The starting time of all the nodes that finish before t_1, as well as those that finish at t_1 except for v, remains unchanged in $\bar{S}$. The finish time of v and all nodes of J that finish after t_1 is delayed (e.g., by some higher priority tasks) with a constant $c > 0$. Let $F(t)$ and $R(t)$ respectively denote the set of nodes of J finished and released at time t in the schedule S. Also, let $\bar{F}(t)$ and $\bar{R}(t)$ denote the set of nodes of J finished and released at t in the schedule $\bar{S}$. Based on this, we have $\forall t < t_1 : F(t) = \bar{F}(t)$ and $R(t) = \bar{R}(t)$. In addition, $\forall t > t_1 : F(t) = \bar{F}(t + c)$ and $R(t) = \bar{R}(t + c)$. This means that the ACR of J remains unchanged in $\bar{S}$ for all $t < t_1$. Also, for any $t > t_1$, there is a counterpart $t + c$ in $\bar{S}$ with the same number of finished and released nodes, revealing the same number of ACR. To compare the number of ACR of the two schedules, therefore, we only need to compare the value of this parameter for time t_1 in S (denoted by $\delta(t_1)$) with the sum of the ACR of J in $\bar{S}$ at t_1 and $t_2 = t_1 + c$ (denoted by $\bar{\delta}(t_1)$ and $\bar{\delta}(t_2)$, respectively). For this, we first note that

$$R(t_1) = \bar{R}(t_1) \cup \bar{R}(t_2), \;\; F(t_1) = \bar{F}(t_1) \cup \bar{F}(t_2) \tag{2}$$

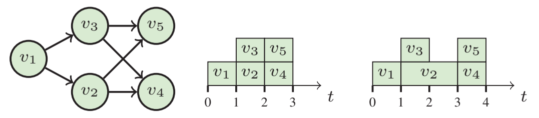
Based on this, one can write

$$\begin{aligned}\delta(t_1) &= \max(0, |R(t_1)| - |F(t_1)|) \\ &= \max(0, |\bar{R}(t_1) \cup \bar{R}(t_2)| - |\bar{F}(t_1) \cup \bar{F}(t_2)|) \\ &\overset{(*)}{=} \max(0, |\bar{R}(t_1)| + |\bar{R}(t_2)| - (|\bar{F}(t_1)| + |\bar{F}(t_2)|)) \\ &= \max(0, |\bar{R}(t_1)| - |\bar{F}(t_1)| + |\bar{R}(t_2)| - |\bar{F}(t_2)|) \\ &\overset{(**)}{\leq} \max(0, |\bar{R}(t_1)| - |\bar{F}(t_1)|) + \\ &\qquad \max(0, |\bar{R}(t_2)| - |\bar{F}(t_2)|) \\ &= \bar{\delta}(t_1) + \bar{\delta}(t_2)\end{aligned} \tag{3}$$

where $(*)$ holds since $\bar{R}(t_1) \cap \bar{R}(t_2) = \emptyset$ and $\bar{F}(t_1) \cap \bar{F}(t_2) = \emptyset$, and $(**)$ holds by the definition of the $\max$ operator for any value of the operands. Relation (3) shows that the ACR of J in $\bar{S}$ is no less than that in S. At the same time, the number of coinciding finishing times in $\bar{S}$ is decremented by one compared to S. This is because, in $\bar{S}$, the finish of v (whose finishing time was delayed) coincides with the finish of no other node of J.

By inductively applying the above procedure to separate coinciding finishing times, a schedule can be obtained where the finishing time of no two nodes of J coincide, and ACR is not smaller than that of the original schedule. □

As an illustration, consider the DAG in Fig. 4a. A sample schedule is seen in Fig. 4b where v_2 and v_3 finish at the same time, giving $\delta = 1$. Delaying the finish of v_2 and the subsequent nodes yields $\delta = 2$, as shown in Fig. 4c.



(a) An example DAG (b) $\delta = 1$ (c) $\delta = 2$

Fig. 4: Separating nodes' finishing time.

Based on Lemma 1, we henceforth restrict our attention to schedules with separate node finish times. Given that, the ACR of a job in a specified schedule can be computed by

$$\delta = \sum_{v \in V} \max(0, |rel(v)| - 1), \tag{4}$$

where $rel(v)$ denotes the set of nodes that are released once v completes, i.e., $rel(v) \equiv R(f_v)$ (recall that f_v denotes the finishing time of v). Further, the following remark follows.

Remark 1. *Given a DAG $G = (V, E)$, under any schedule with distinct finishing times, $\sum_{v \in V} |rel(v)|$ equals the number of non-source nodes.*

The remark can be verified by noting that: (1) $\sum_{v \in V} |rel(v)| = | \cup_{v \in V} rel(v)|$ since $rel(v_i) \cap rel(v_j) = \emptyset$

for any $v_i, v_j \in V$ due to non-coinciding finish times, and (2) $\cup_{v \in V} rel(v)$ encompasses all, and only, non-source nodes.

V. Problem Complexity

To establish the computational complexity of Problem 1, we present a polynomial-time reduction from the well-known Min Vertex Cover problem to the problem of computing maximum ACR (Max ACR). We first review the Min Vertex Cover problem in Section V-A, and then present the reduction in Section V-B.

A. Min Vertex Cover

Throughout this section, by graph we mean an undirected graph, unless otherwise specified. Consider a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with $\mathcal{V}$ denoting the set of vertices and $\mathcal{E}$ denoting the set of edges. Any set $\mathcal{V}_c \subseteq \mathcal{V}$ is a *vertex cover* for $\mathcal{G}$ if

$$\forall (u, v) \in \mathcal{E} : \{u, v\} \cap \mathcal{V}_c \neq \emptyset \tag{5}$$

In words, $\mathcal{V}_c$ is a vertex cover if each edge has at least one endpoint in it. The problem of Min Vertex Cover asks for the smallest vertex cover.

Definition 1 (Min Vertex Cover). *Given a graph $\mathcal{G}$, the problem is to find the size of the smallest vertex cover.*

The Min Vertex Cover problem is NP-hard [5].

B. Reduction

The idea is to transform any instance of Min Vertex Cover to an instance of Max ACR such that solving the latter gives a solution to the former.

Consider a problem instance of Min Vertex Cover with the graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ as input. We construct an instance of Max ACR with a DAG $G = (\mathcal{V} \cup V_{\mathcal{E}}, E)$, where $V_{\mathcal{E}}$ is composed of $|\mathcal{E}|$ elements: for each $(v_i, v_j) \in \mathcal{E}$, $V_{\mathcal{E}}$ has an element v_{ij}. (Therefore, G has $|\mathcal{V}| + |\mathcal{E}|$ vertices.) In turn, E is constructed such that for any $(v_i, v_j) \in \mathcal{E}$, E has two elements (v_i, v_{ij}) and (v_j, v_{ij}).

Example 2. *Fig. 5a depicts a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with $\mathcal{V} = \{v_1, v_2, v_3, v_4\}$ and $\mathcal{E} = \{(v_1, v_2), (v_1, v_3), (v_1, v_4)\}$. Fig. 5b shows the DAG obtained by the above-described transformation.*

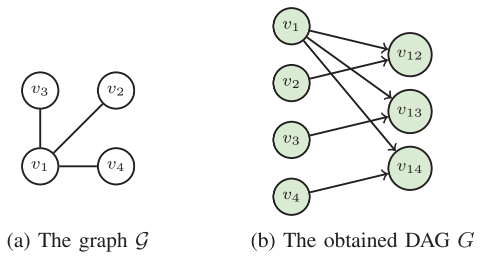


Fig. 5: Constructing a DAG G from a graph $\mathcal{G}$.

Remark 2. *Consider a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, and the DAG $G = (\mathcal{V} \cup V_{\mathcal{E}}, E)$ obtained by the described transformation. For any job of G, Remark 1 implies $\sum_{v \in \mathcal{V} \cup V_{\mathcal{E}}} |rel(v)| = |V_{\mathcal{E}}|$. This means, noting $|V_{\mathcal{E}}| = |\mathcal{E}|$, that*

$$\sum_{v \in \mathcal{V} \cup V_{\mathcal{E}}, |rel(v)| \geq 1} |rel(v)| = |\mathcal{E}| \tag{6}$$

We first present a property of any schedule that exhibits the maximum ACR.

Lemma 2. *Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, let $G = (\mathcal{V} \cup V_{\mathcal{E}}, E)$ be the corresponding DAG obtained from the transformation. Consider a job of G in an arbitrary schedule (with distinct finishing time of the nodes). The ACR of the job equals*

$$\delta = |\mathcal{E}| - |V^+|, \tag{7}$$

where

$$V^+ \overset{\text{def}}{=} \{v \in \mathcal{V} \cup V_{\mathcal{E}} : |rel(v)| \geq 1\} \tag{8}$$

Proof. From (4) we have

$$\begin{aligned} \delta &= \sum_{v \in \mathcal{V} \cup V_{\mathcal{E}}} \max(0, |rel(v)| - 1) \\ &= \sum_{v \in \mathcal{V} \cup V_{\mathcal{E}}, rel(v) \geq 1} (|rel(v)| - 1) \\ &= \left(\sum_{v \in \mathcal{V} \cup V_{\mathcal{E}}, rel(v) \geq 1} |rel(v)| \right) - |V^+| \\ &\overset{(*)}{=} |\mathcal{E}| - |V^+|, \end{aligned} \tag{9}$$

where $(*)$ follows from (6). □

As an example, consider the graph $\mathcal{G}$ in Example 2, and its corresponding DAG G shown in Fig. 5b. Fig. 6 shows a possible schedule of G, which gives $V^+ = \{v_1\}$. The number of ACR is then $|\mathcal{E}| - |V^+| = 3 - 1 = 2$.

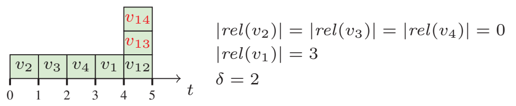


Fig. 6: A schedule for the DAG in Fig 5b

Lemma 3. *Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, let G and V^+ be as defined in Lemma 2. Then, V^+ is a vertex cover for $\mathcal{G}$.*

Proof. First, note that $V^+ \subseteq \mathcal{V}$ since $\forall v \in V_{\mathcal{E}} : |rel(v)| = 0$ meaning that $v \notin V^+$. Therefore, we just need to show that $\forall (v_i, v_j) \in \mathcal{E}$, either $v_i \in V^+$ or $v_j \in V^+$ (or both). Each vertex v_{ij} in $V_{\mathcal{E}}$ is certainly released by one of its predecessors, i.e., either $rel(v_i) \geq 1$ or $rel(v_j) \geq 1$. Thus, either $v_i \in V^+$ or $v_j \in V^+$. Equivalently, for each edge $(v_i, v_j) \in \mathcal{E}$, either $v_i \in V^+$ or $v_j \in V^+$ which means that all edges in $\mathcal{E}$ have at least one endpoint in V^+. □

As an example, consider the schedule in Fig. 6 for which $V^+ = \{v_1\}$. As seen from Fig. 5a, V^+ is a vertex cover for $\mathcal{G}$.

Lemma 4. *Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and its corresponding DAG G, if the maximum ACR of G is δ then there exists a vertex cover $\mathcal{V}_c$ for $\mathcal{G}$ such that $|\mathcal{V}_c| = |\mathcal{E}| - \delta$.*

Proof. Let V^+ be as defined in Lemma 2. From Lemma 3, V^+ is a vertex cover for $\mathcal{G}$, and from (7), $|V^+| = |\mathcal{E}| - \delta$. □

Next, we show that any solution to an instance of Min Vertex Cover can be linked to an ACR in the related DAG.

Lemma 5. *Assume a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, and the corresponding DAG $G = (\mathcal{V} \cup V_{\mathcal{E}}, E)$ obtained from our transformation. For any smallest vertex cover $\mathcal{V}_c$ of $\mathcal{G}$, there exists a schedule where G exhibits an ACR of $|\mathcal{E}| - |\mathcal{V}_c|$.*

Proof. Assume a schedule in which the nodes of a job of G finish in the following order: first, all nodes in $\mathcal{V} \setminus \mathcal{V}_c$,[2] then nodes in $\mathcal{V}_c$, and then, the rest (i.e., $V_{\mathcal{E}}$).

We first note that as $\mathcal{V}_c$ is a vertex cover, for any $(v_i, v_j) \in \mathcal{E}$ at least one endpoint is in $\mathcal{V}_c$. Equivalently, in G, each v_{ij} has at least one predecessor in $\mathcal{V}_c$. As nodes in $\mathcal{V}_c$ finish after nodes in $\mathcal{V} \setminus \mathcal{V}_c$, no node is released even when all nodes in $\mathcal{V} \setminus \mathcal{V}_c$ are finished, implying $\forall v \in \mathcal{V} \setminus \mathcal{V}_c : |rel(v)| = 0$. We next note that $\forall v \in V_{\mathcal{E}} : |rel(v)| = 0$ since nodes in $V_{\mathcal{E}}$ have no successor, and hence, release no node. In conclusion,

$$\forall v \notin \mathcal{V}_c : |rel(v)| = 0. \tag{10}$$

This implies $V^+ \subseteq \mathcal{V}_c$ where V^+ is as defined in (8) in Lemma 2. We claim $V^+ \equiv \mathcal{V}_c$. To prove this, we note that from Lemma 3, V^+ is a vertex cover for $\mathcal{G}$. Hence, if $V^+ \neq \mathcal{V}_c$, we will have $V^+ \subset \mathcal{V}_c$, implying $|V^+| < |\mathcal{V}_c|$, which contradicts the fact that $\mathcal{V}_c$ is a *minimum* vertex cover. Hence, according to Lemma 2, the ACR of the job is $|\mathcal{E}| - |V^+| = |\mathcal{E}| - |\mathcal{V}_c|$, concluding the proof. □

Putting the above results together, we show how Min Vertex Cover is reduced to Max ACR.

Lemma 6. *Assume an instance of the* Min Vertex Cover *problem with $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ as the input. Let G be the corresponding DAG obtained from our transformation. If the maximum ACR of G is δ, then, the size of the smallest vertex cover of $\mathcal{G}$ is $|\mathcal{E}| - \delta$.*

Proof. Let the maximum ACR of G be δ. We must show that

1) $\mathcal{G}$ has a vertex cover $\mathcal{V}_c$ for which $|\mathcal{V}_c| = |\mathcal{E}| - \delta$, and
2) $\mathcal{G}$ has no vertex cover smaller than $|\mathcal{E}| - \delta$.

1) This is already proven by Lemma 4.
2) We prove by contradiction. Assume it does not hold, i.e., there exists a minimum vertex cover $\mathcal{V}_c{}'$ for $\mathcal{G}$ such that

$$|\mathcal{V}_c{}'| < |\mathcal{E}| - \delta \tag{11}$$

From Lemma 5, this means that there is a schedule for a job of G with the ACR of $|\mathcal{E}| - |\mathcal{V}_c{}'|$, which, according to (11), is larger that δ. This means that δ is not the maximum ACR, which contradicts the initial assumption. □

Theorem 1 (Problem Complexity)**.** *Computing maximum ACR of a DAG is* NP*-hard.*

Proof. Based on Lemma 6, one can solve any instance of the Min Vertex Cover problem through transforming it to a Max ACR problem and solving the latter. This means that Max ACR is at least as *hard* as Min Vertex Cover (as our transformation is done in polynomial time). Therefore, as Min Vertex Cover is NP-hard, Max ACR is NP-hard. □

[2]"$\setminus$" denotes the *set subtraction* operator.

VI. An Exact Solution

We present an exact solution for computing the maximum ACR of a DAG $G = (V, E)$. For this, we formulate the problem as an Integer Linear Program (ILP). The formulation consists of two parts: (1) encoding the valid schedules, and (2) representing the objective.

A. Encoding Possible Schedules

As described earlier, the number of ACR depends on the nodes' finishing time. Therefore, we primarily consider $|V|$ variables $f_1, \ldots, f_{|V|}$ to denote the finishing times of the nodes. According to Lemma 1, we will restrict our focus to non-coinciding finishing times. Hence, we need to explore the cases where (1) no two finishing times coincide, and (2) the finishing times adhere the precedence constraints of the DAG.

First, we constrain the finishing time variables f_i as

$$\forall i, 1 \leq i \leq |V| : 0 \leq f_i \leq |V| - 1. \tag{c1}$$

The upper bound is needed for subsequent constraints, specified shortly. As only the relative value of finishing times does matter in computing ACR, putting an upper bound does not effectively confine the state space (as long as the variables can get $|V|$ distinct values).

To recognize the order in which the nodes are finished, we add the *ordering* constraints. Let M be a large constant (for our purpose, it suffices to set $M = |V|$). For $1 \leq i, j \leq |V|$, $i \neq j$, we add:

$$\begin{aligned} &f_j \geq f_i + 1 - M \cdot x_{ij} \\ &f_i \geq f_j + 1 - M \cdot (1 - x_{ij}) \\ &x_{ij} + x_{ji} = 1 \end{aligned} \tag{c2}$$

where x_{ij}s are binary variables which define the ordering: when $x_{ij} = 0$, the first constraint in (c2) forces $f_j > f_i$ while the second one becomes void. Likewise, $x_{ij} = 1$ forces $f_j < f_i$.

Next, we enforce the precedence constraints of the DAG by

$$\forall (v_i, v_j) \in E : x_{ij} = 0 \tag{c3}$$

Constraints (c1) to (c3) define a complete state space in the sense that they allow any ordering of finishing times that can be generated by G. Also, the constraints are sound since they disallow any invalid finishing time ordering. The reason is that the only constraints that G forces on the ordering are described by the edges, all of which are respected by the constraint (c3).

B. Formulating the Objective

To express the objective, we have to count the number of nodes that are released upon the finish of a node. For any $(v_i, v_j) \in E$, v_j is released upon the finish of v_i if all predecessors of v_j other than v_i have been finished before v_i, or equivalently, if $\sum_{v_k \in Pred(v_j) \setminus \{v_i\}} x_{ik} = |Pred(v_j)| - 1$. To detect this condition, we use a binary variable b_{ij} with the following constraint:

$$\begin{aligned}\sum_{v_k \in Pred(v_j)\setminus\{v_i\}} x_{ik} \geq |Pred(v_j)| - 1 - (1 - b_{ij}) \cdot M \\ |Pred(v_j)| - 2 \geq (\sum_{v_k \in Pred(v_j)} x_{ik}) - b_{ij} \cdot M\end{aligned} \quad \text{(c4)}$$

In words, $b_{ij} = 1$ if and only if v_j is released upon the finish of v_i. Therefore, the total number of nodes that are released upon v_i completion can be computed by $\sum_{v_j \in Succ(v_i)} b_{ij}$. The objective is to count ACR after a node v_i is finished, that is $\delta_i = \max\left(0, \sum_{v_j \in Succ(v_i)} b_{ij} - 1\right)$, which can be expressed by the following constraints:

$$\begin{aligned}&\delta_i \geq 0 \\ &\delta_i \geq (\sum_{v_j \in Succ(v_i)} b_{ij}) - 1 \\ &\delta_i \leq M \cdot b_i \\ &\delta_i \leq (\sum_{v_j \in Succ(v_i)} b_{ij}) - 1 + M \cdot (1 - b_i)\end{aligned} \quad \text{(c5)}$$

where b_i is a binary variable. The first two lines in (c5) guarantee $\delta_i \geq \max\left(0, \sum_{v_j \in Succ(v_i)} b_{ij} - 1\right)$, and the last two force $\delta_i \leq \max\left(0, \sum_{v_j \in Succ(v_i)} b_{ij} - 1\right)$.

Finally, we express the objective as: $maximize \sum_{1 \leq i \leq |V|} \delta_i$.

VII. Evaluation

This section explores the scalability of the proposed ILP solution with respect to the DAG size. We use the STR2RTS benchmark [7], which describes a number of digital signal processing applications in terms of DAGs. The benchmark programs are listed in Table I. The second and third columns provide the number of vertices and edges of the corresponding DAG, which determine the size of the extracted ILP formulation (i.e., number of variables and constraints). The fourth column shows the number of fork nodes in each DAG.

To solve the ILP formulation, we have employed the Gurobi optimizer v.9.1.2 [4] running on a quad-core 2.4 GHz processor with 8 GB of RAM. The runtime of the solver for each DAG is shown in the fifth column of Table I. As seen, the longest runtime belongs to the DES application, which consists of a DAG with more than 400 nodes. It is worth noting that the number of variables and constraints of the ILP for a DAG with $|V|$ nodes is $O(|V|^2)$. It is seen that despite the inherent hardness of the problem, solving the problem can be done efficiently in practice. This, to some extent, is related to the structure of the DAG in real applications, which usually do not entail an arbitrary complex structure, although being very large.

The last two columns of Table I respectively show the obtained number of ACR and a corresponding *upper bound*. The upper bound is obtained by simply computing $\sum_{v \in V} \max(0, |Succ(v) - 1|)$ for a DAG $G = (V, E)$. It is seen that in almost all cases the upper bound yields an exact solution (the exception is DCT2). The reason is that in most cases, all successors of a node v have only one predecessor,

Application	**#node**	**#edge**	**#fork**	**Runtime (s)**	δ	$\hat{\delta}$
802.11a	119	153	17	**5.8**	35	35
Audiobeam	20	33	1	**0.1**	14	14
BeamFormer	56	69	2	**0.4**	14	14
CFAR	4	3	0	**0.1**	0	0
Complex-FIR	3	2	0	**0.1**	0	0
DCT2	40	69	2	**0.4**	30	31
DES	423	598	80	**194.8**	176	176
FFT2	26	26	1	**0.4**	1	1
FFT4	42	51	10	**0.7**	10	10
Filterbank	53	59	1	**0.4**	7	7
FMRadio	43	53	7	**0.4**	11	11

TABLE I: Deriving maximum ACR for STR2RTS programs.

which is v itself. Hence, the finish of v releases all successor nodes, leading to an ACR of $|Succ(v) - 1|$.

VIII. Conclusions

We have studied the problem of computing maximum total number of additional cores a DAG job can request after each node spawning. We showed that the polynomial-time algorithm proposed in [9] for this problem is not correct, giving a counter example and proving the problem NP-hard. An ILP formulation was presented as an exact solution to the problem. Based on this correction, the timing analysis presented in [9] for sporadic DAG tasks will be updated, giving potentially larger response times. As a consequence, the corresponding schedulability results presented in [9], and also in later studied such as [2], can be degraded.

As a direction of future work, we note that although the problem is NP-hard, polynomial-time algorithms can be developed in certain cases. For instance, the algorithm presented in [9] is conjectured to be sound for DAGs where each node has at most one incoming edge, or for *synchronous* DAGs [8] which are composed of a sequence of sequential or parallel segments.

References

[1] Ö. Babaoğlu, K. Marzullo, and F. B. Schneider. A formalization of priority inversion. *Real-Time Systems*, 5(4):285–303, 1993.

[2] D. Casini, A. Biondi, G. Nelissen, and G. Buttazzo. Partitioned fixed-priority scheduling of parallel tasks without preemptions. In *Real-Time Systems Symposium*, pages 421–433, 2018.

[3] D. Casini, T. Blaß, I. Lütkebohle, and B. B. Brandenburg. Response-time analysis of ROS 2 processing chains under reservation-based scheduling. In *Euromicro Conference on Real-Time Systems*, 2019.

[4] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2021.

[5] R. M. Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. 1972.

[6] OpenMP Architecture Review Board. OpenMP application program interface version 5.1, 2020.

[7] B. Rouxel and I. Puaut. STR2RTS: Refactored streamit benchmarks into statically analyzable parallel benchmarks for wcet estimation & real-time scheduling. In *International Workshop on Worst-Case Execution Time Analysis (WCET)*, 2017.

[8] A. Saifullah, J. Li, K. Agrawal, C. Lu, and C. Gill. Multi-core real-time scheduling for generalized parallel task models. *Real-Time Systems*, 49(4):404–435, 2013.

[9] M. A. Serrano, A. Melani, S. Kehr, M. Bertogna, and E. Quiñones. An analysis of lazy and eager limited preemption approaches under DAG-based global fixed priority scheduling. In *International Symposium on Real-Time Distributed Computing*, pages 193–202, 2017.

[10] M. Verucchi, M. Theile, M. Caccamo, and M. Bertogna. Latency-aware generation of single-rate DAGs from multi-rate task sets. In *Real-Time and Embedded Technology and Applications Symposium*, pages 226–238, 2020.