Feasibility Analysis of Conditional DAG Tasks

Anonymous author
Anonymous affiliation

Anonymous author
Anonymous affiliation

Abstract

Feasibility analysis for Conditional DAG tasks (C-DAGs) upon multiprocessor platforms is shown to be complete for the complexity class PSPACE. It is shown that as a consequence integer linear programming solvers (ILP solvers) are likely to prove inadequate such analysis. A demarcation is identified between the feasibility-analysis problems on C-DAGs that are efficiently solvable using ILP solvers and those that are not, by characterizing a restricted class of C-DAGs for which feasibility analysis is shown to be efficiently solvable using ILP solvers.

2012 ACM Subject Classification Computer systems organization → Embedded and cyber-physical systems; Software and its engineering → Real-time schedulability

Keywords and phrases Multiprocessor feasibility analysis; Conditional Directed Acyclic Graphs; PSPACE-complete

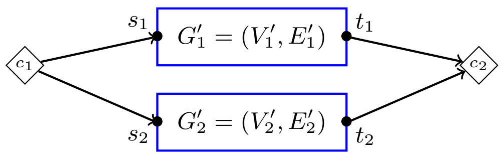
1 Introduction

This paper investigates the *feasibility analysis problem for C-DAG tasks*: the problem of determining whether a given real-time workload which is specified in the Conditional Directed Acyclic Graph task (C-DAG) model [**?**, **?**] and is to be implemented upon a particular multiprocessor platform, can be scheduled to always complete by a specified deadline. Since it follows from earlier results [**?**] that a simpler version of this problem, in which the workload is specified as a DAG (i.e., without any conditional nodes) is already NP-hard in the strong sense, we should not expect to obtain algorithms with polynomial or pseudo-polynomial running times that solve our problem exactly. Two approaches to such feasibility analysis problems (i.e., those that are provably NP-hard in the strong sense) have previously been investigated in the real-time literature: (i) design approximation algorithms that run in polynomial or pseudo-polynomial time; or (ii) derive exact algorithms that (necessarily, assuming $P \neq NP$) run in exponential time. The latter approach is often based upon transforming the feasibility analysis problem into an integer linear program (ILP), and leveraging the tremendous recent improvements that have been obtained in the performance of ILP solvers to achieve running times that are acceptable in practice for reasonably large problem instances.[1] In this paper we prove that an approach based on transformation to ILPs is unlikely to be applicable to the general C-DAG feasibility-analysis problem – to our knowledge, this is amongst the first feasibility-analysis problems for which such a negative result regarding the use of ILPs has been obtained in the real-time literature. We also identify an important restricted case for which ILP-solvers can in fact prove helpful: this special case essentially limits the number of conditional constructs that may be present.

Our Contributions. Two major technical results are proved in this paper:

1. the C-DAG feasibility analysis problem is PSPACE complete; and
2. it is in NP if the number of conditional constructs is *a priori* bounded by a constant.

[1] We point out that determining whether an ILP has a solution is known [**?**] to be NP-complete in the strong sense; hence the overall worst-case run-time complexity of this approach remains exponential.



Vertices s_1 and t_1 (vertices s_2 and t_2, resp.) are the sole source vertex and sink vertex of G'_1 (of G'_2, resp.).

Figure 1 A canonical conditional construct

While at first glance these may appear to be highly theoretical results that are a poor fit for ECRTS, we will establish that they do in fact have major implications to real-time systems design and implementation. We will show that it follows from our first result that it is highly unlikely we will be able to solve general C-DAG feasibility analysis problems in polynomial time even when calls to an ILP solver are **?**for free**?** (and hence, regardless of how good your ILP solver may be). The second result clearly shows that the root cause of this is the presence of the conditional constructs, and thereby demarcates the boundary between feasibility-analysis problems that are efficiently transformable to ILPs and those that are not. We also offer evidence that the size of the ILP for solving an instance of this restricted case grows exponentially with the number of conditional constructs that are present. This in turn suggests a design guideline: conditional constructs be considered as a scarce **?**resource**?** to be used only when their increased expressiveness is essential, since their presence can slow down feasibility analysis exponentially.

Organization. The remainder of this manuscript is organized as follows. We describe the Conditional DAG model in Section 2, and briefly review some needed results from complexity theory in Section 3. Our main technical results are in Section 4 (the PSPACE completeness proof) and Section **??** (the more tractable special case). We conclude in Section **??** by listing some additional implications of our findings and placing these within the context of related research, and briefly list some interesting directions for future research.

2 The Conditional DAG (C-DAG) Model

Task models based upon Directed Acyclic Graphs (DAGs) seek to expose parallelism in real-time workloads: the *sporadic DAG model* [**?**] (see [**?**, Chapter 21] for a text-book description) is an early example. A task in this model is specified as a 3-tuple (G, D, T), where G is a directed acyclic graph (DAG), and D and T are positive integers representing the relative deadline and period parameters of the task respectively. The task repeatedly releases *dag-jobs*, each of which is a collection of sequential jobs. Successive dag-jobs are released a duration of at least T time units apart. The DAG G is specified as $G = (V, E)$, where V is a set of vertices and E a set of directed edges between these vertices. Each $v \in V$ represents a job, which corresponds to the execution of a sequential piece of code and is characterized by a worst-case execution time (WCET). The edges represent dependencies between the jobs: if $(v_1, v_2) \in E$ then job v_1 must complete execution before job v_2 can begin execution. A release of a dag-job of the task at time-instant t means that all $|V|$ jobs $v \in V$ are released at t. If a dag-job is released at time t then all $|V|$ jobs that were released at t must complete execution by time $t + D$.

Conditional DAG tasks. The Conditional DAG (C-DAG) task model was introduced [**?**, **?**] to model the execution of conditional (e.g., **if-then-else**) constructs in parallel real-time code. A C-DAG task, too, is specified as a 3-tuple (G, D, T), where $G = (V, E)$ is a DAG, and D and T are positive integers denoting the relative deadline and period parameters of the task. They differ from regular sporadic DAGs in that certain vertices $\in V$ are designated as *conditional vertices* that are defined in matched pairs, each such pair defining a *conditional construct*. Let (c_1, c_2) be such a pair in the DAG $G = (V, E)$ — see Figure 1. Informally speaking, vertex c_1 represents a point in the code where a conditional expression is evaluated and, depending upon the outcome of this evaluation, control will subsequently flow along one of two different possible branches.[2] It is required that these two different branches meet again at a common point in the code, represented by the vertex c_2. More formally,

1. There are two outgoing edges from c_1 in E (say, to the vertices s_1 and s_2), and two incoming edges to c_2 (say, from the vertices t_1 and t_2), in E — see Figure 1.
2. For each $\ell \in \{1, 2\}$, let $V'_\ell \subseteq V$ and $E'_\ell \subseteq E$ denote all the vertices and edges on paths reachable from s_ℓ that do not include vertex c_2. By definition, s_ℓ is the sole source vertex of the DAG $G'_\ell \overset{\text{def}}{=} (V'_\ell, E'_\ell)$. Vertex t_ℓ must be the sole sink vertex of G'_ℓ.
3. It must hold that $V'_1 \bigcap V'_2 = \emptyset$. Additionally for each $\ell \in \{1, 2\}$, with the exception of (c_1, s_ℓ) there should be no edges in E into vertices in V'_ℓ from vertices that are not in V'_ℓ.

Edges (v_1, v_2) between pairs of vertices neither of which are conditional nodes represent precedence constraints exactly as in traditional sporadic DAGs, while edges involving conditional nodes represent conditional execution of code. More specifically, let (c_1, c_2) denote a defined pair of conditional vertices that together define a conditional construct. The semantics of conditional DAG execution mandate that

- After the job c_1 completes execution, exactly one of its two successor jobs becomes eligible to execute; it is not known beforehand which successor job this may be.
- Job c_2 begins to execute upon the completion of exactly one of its two predecessor jobs.

In the remainder of this paper we make the *simplifying assumption that each of the conditional vertices c_1 and c_2 demarcating a conditional construct has zero execution time.*

The C-DAG feasibility analysis problem. We are interested, from a real-time systems perspective, in understanding how to implement specified collections of C-DAG tasks upon a shared multiprocessor platform in a correct and resource-efficient manner. The *federated scheduling* paradigm [**?**], in which each task is restricted to execute upon a specified subset of the processors (and each processor is assigned to no more than one task), is a widely-studied approach for implementing collections of tasks represented using DAG-based models upon multiprocessor platforms. It is readily seen that federated scheduling of constrained-deadline tasks — tasks (G, D, T) for which the deadline parameter D is no larger than the period T — reduces to the problem of scheduling a single C-DAG upon a dedicated set of processors within a duration not exceeding the relative deadline parameter. Hence the problem considered in this paper is this:

▶ **Definition 1** (The C-DAG feasibility analysis problem). GIVEN *a C-DAG G, a number $m \in \mathbb{N}$ of processors upon which G is to execute, and a relative deadline parameter D,* DETERMINE *whether it is* feasible *to schedule G on the m processors such that it always completes execution within an interval of duration D, regardless of which conditional constructs in G evaluate to true and which evaluate to false?* ◀

[2] The model is more generally defined [**?**, **?**] to allow for > 2 possible branches; however any task with more than 2 branches is easily transformed in polynomial time to one with always only two branches.

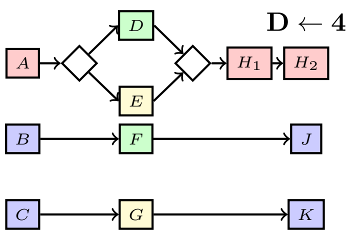
The problem definition above is incomplete: several variants can be defined based upon restrictions that are placed on how jobs may execute. For instance permitting or prohibiting preemption results in different variants. Variants may be also be defined based upon which processors each job is allowed to execute on:

global: any job may execute upon any processor, and the decision as to which processor a job executes upon may be made at run-time. When preemption is permitted, a preempted job may resume execution upon a different processor.

partitioned: each job may execute upon only one processor, and the determination as to which processor a job executes upon is made prior to run-time.

restricted (or typed [**?**]): each job is pre-assigned to a particular processor. I.e., a mapping from vertices to processors is provided as part of the problem specifications.

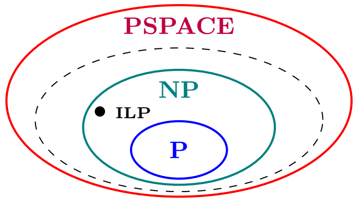
Why this is a difficult problem. It has been widely recognized [**?**, **?**, **?**, **?**] that *combinatorial explosion* is a major reason why C-DAG feasibility analysis is such a difficult problem: exponentially many different combinations of outcomes are possible of the evaluation of the conditional constructs in a single task, each of which may require a very different collection of jobs to be scheduled for execution. There is, however, an additional aspect to the difficulty of this problem that has received somewhat less attention: its inherently *on line* nature. Consider the following simple illustrative example for a typed C-DAG (i.e., where vertices are pre-assigned to individual processors):



Example: *Each vertex has WCET equal to one (except the conditional vertices – recall they have WCET zero). Processor assignments are color-coded: A, H_1, & H_2 share a processor, as do B, C, J, & K; D & F; and E & G. If the conditional construct executes D, then C should execute during* $[0, 1]$ *—otherwise the ?blue? processor will idle over* $[2, 3]$. *Else (i.e., the conditional construct executes E), B should execute during* $[0, 1]$.

There are two possible outcomes of the sole conditional construct, and it may be verified that upon either outcome the set of vertices that must be executed is individually schedulable. However, which of vertices B or C, both assigned to the same processor, should execute over time-interval $[0, 1]$ necessarily differs in these two schedules and hence depends upon the outcome of the conditional construct's evaluation. But the conditional construct is only executed *after* time-instant 1, and hence this information is revealed too late. Thus this C-DAG is *in*feasible despite the sets of vertices needing to be executed upon either outcome being feasible.

Summarizing Prior Complexity Results. Ullman showed [**?**] that it is NP-complete in the strong sense to determine whether a given DAG can be scheduled to meet a specified deadline under global or partitioned scheduling upon an identical multiprocessor platform, regardless of whether preemption is permitted or forbidden. Jansen subsequently showed [**?**] that feasibility analysis of DAGs is NP-hard in the strong sense for restricted/ typed C-DAGs (where each job is pre-assigned to a particular processor), again under both preemptive and non-preemptive scheduling. Since these basic problems are already NP-hard in the strong sense, so are the corresponding problems for the more general C-DAG model. (It is easily seen that all these problems are also in NP for (regular) DAGs; one of the contributions of this paper is to prove that such is not the case with C-DAGs.)



The innermost (blue) solid line represents the problems in P, the intermediate (teal) one includes problems that are in NP, and the outermost (red) one further includes problems that are in PSPACE. The dotted black line depicts the class P^{NP}.
As shown in the Venn diagram, the problem of solving ILPs is in NP but not in P (assuming P≠NP).

Figure 2 Venn diagram depicting the relationship between some complexity classes

3 Computational Complexity: Some Background

We now provide a brief introduction to concepts of computational complexity theory that are used in this manuscript.[3] We will make reference to the following four complexity classes:

1. P is the set of problems that can be *solved* by algorithms with running time polynomial in the size of their inputs.
2. NP is the set of problems that can be *verified* by algorithms with running time polynomial in the size of their inputs.
3. P^{NP} is the set of problems that can be solved in polynomial time by an algorithm that has access to an *oracle* for some NP-complete problem, where an oracle can be thought of as a ?black box? that is able to solve a specific decision problem in a single step.
4. PSPACE is the set of problems that can be solved by algorithms using an amount of *space* (memory) that is polynomial in the size of their inputs. Since this complexity class has not previously been widely used in real-time scheduling theory, we discus it a bit more below, and provide some intuition of its relationship to C-DAG feasibility analysis.

It is widely believed, although not proved, that $\left(\mathrm{P} \subsetneq \mathrm{NP} \subsetneq \mathrm{P}^{\mathrm{NP}} \subsetneq \textsc{pspace}\right)$ – see Figure 2.

PSPACE The class PSPACE can be thought of as representing the existence of a winning strategy for a particular player in bounded-length perfect-information games that can be played in polynomial time. I.e., consider a two-player game where players alternate making moves for a total of n moves. Given moves $m_1, \ldots, m_n$ by the players, let $M(m_1, \ldots, m_n) = 1$ if and only if player 1 has won the game. Then player 1 has a winning strategy in the game if and only if there exists a move m_1 that player 1 can make such that for every possible response m_2 of player 2 there is a move m_3 for player 1, . . . and so on. Formalizations of many popular two-player games, including checkers, generalized geography, and Sokoban, have been proven to be PSPACE-complete [**?**].

We can cast C-DAG feasibility in this two-player game framework. Given a C-DAG and a deadline D, then the first move of player 1 (the scheduler) is to decide the set of jobs to be scheduled until the first branch is executed; then player 2 (the environment) decides the outcome of the branch. The game continues until the scheduling is completed and the first player wins the game if and only if its strategy is able to complete the schedule in D time units for all outcomes of branches (i.e. all decisions of the second player).

[3] In order to keep things simple the presentation in this section is intentionally informal and not always precise: for instance, while most of the concepts discussed below differ in their applicability to *decision problems* – those for which there is a ?YES/ NO? answer – and *optimization problems*, we do not make this distinction here but treat both decision and optimization problems in similar fashion.

ILP solvers. Determining whether an integer linear program (ILP) has a solution or not is known to be NP-complete in the strong sense [?]. Assuming P $\neq$ NP, this implies that ILP solvers with polynomial or pseudo-polynomial running time cannot be developed. Despite this inherent intractability, however, the optimization community has devoted immense effort to devise very efficient implementations of ILP solvers, and highly optimized libraries with such efficient implementations are widely available today in both open-source and commercial offerings. Modern ILP solvers, executing upon powerful computing clusters, are commonly capable of solving ILPs with tens of thousands of variables and constraints.

4 C-DAG feasibility analysis is PSPACE-complete

One of our main results is a negative one: that the C-DAG feasibility analysis problem (Definition 1) is PSPACE-hard for all the variants — preemptive and non-preemptive; global and partitioned and restricted (or typed) — described in Section 2. As stated in Section 3 above, a PSPACE complete problem is highly unlikely to be in NP or P^{NP}; hence we cannot solve it in polynomial time by making additional calls to an ILP-solver, even if each such call took $\Theta(1)$ (i.e., constant) time. In the remainder of this section we will prove this intractability result for the variant[4] of the C-DAG feasibility analysis problem where preemption is permitted and migration is restricted (i.e., each job is pre-assigned to a particular processor):

▶ **Theorem 1.** *The C-DAG feasibility problem when each job is pre-assigned to a particular processor is* PSPACE *complete.*

It is trivial to show that this problem is $\in$ PSPACE — an algorithm that repeatedly simulates the scheduling of the C-DAG under all possible combinations of outcomes of the conditional constructs would require polynomial space. The rest of this section is devoted to proving that this problem is also PSPACE-hard.

(This proof is rather detailed and technical: the reader may wish to skim it, or skip it entirely, on a first reading. However, we do recommend that it eventually be read and understood, since it contains some novel techniques and interesting ideas that are likely to prove useful in further research. We consider these techniques and ideas an important part of the contribution of this paper.)

PSPACE-hardness for our C-DAG feasibility analysis problem is proved by deriving a polynomial-time reduction to the C-DAG feasibility analysis problem from the following problem, which has previously [?, ?] been shown to be PSPACE complete:

▶ **Definition 2** (The Quantified Boolean Formula Problem (QBF))**.**

INSTANCE. A boolean formula in the following form:

$$\exists x_1 \, \forall y_1 \, \exists x_2 \, \forall y_2 \; \ldots \exists x_n \, \forall y_n \bigwedge_{j=1}^{m} \left(\ell_{j,1} \vee \ell_{j,2} \vee \ell_{j,3}\right) \tag{1}$$

where each x_i and each y_i is a boolean variable, and each $\ell_{j,k}$ is one of the x_i or y_i Boolean variables or its negation.

[4] We have also proved this result for the variant that allows for global preemptive scheduling. We are choosing to present the variant with pre-assigned processors for pedagogical reasons: the main ideas in the proof of the hardness of the global preemptive case are also revealed in this proof while a lot of grungy details that are not particularly novel but must be addressed for the global preemptive version are not needed here.

QUESTION. Does this formula evaluate to **true**?[5] ◀

We will describe a polynomial-time algorithm that accepts as input a Boolean formula of the form given in Expression 1 above, and outputs a C-DAG, an assignment of jobs of the C-DAG to processors, and a deadline $D \stackrel{\text{def}}{=} 2n + 3$, such that the C-DAG can complete execution by the deadline if and only if Expression 1 is **true**. Since QBF is known to be PSPACE-complete, this polynomial-time reduction from QBF to C-DAG feasibility analysis suffices to show that C-DAG feasibility analysis is PSPACE hard. We start with a high-level overview of our polynomial-time reduction.

- We will define three kinds of **?**gadgets**?** – subgraphs that have each been designed to achieve some particular purpose – in Sections **??**, **??**, and **??**. The first kind of gadget we will define is used to represent the clauses in Expression 1; the second, the existentially quantified (i.e., x_i) variables and the third, the universally quantified (i.e., y_i) variables. Our C-DAG will be the union of m gadgets of the first kind, n gadgets of the second kind, and n gadgets of the third kind.
- For each boolean variable x_i (y_i, respectively), our C-DAG will have two jobs labeled X_i and $\neg X_i$ (Y_i and $\neg Y_i$, respectively). We will state that job X_i ***?corresponds to?*** literal x_i and job $\neg X_i$ corresponds to the literal $\neg x_i$ (analogously, that Y_i corresponds to y_i and $\neg Y_i$ corresponds to $\neg y_i$).
  We will see, in Sections **??** and **??**, that we construct the gadgets for the x_i's and the y_i's in a manner that enforces the constraint that at most one of each pair of jobs X_i and $\neg X_i$ (Y_i and $\neg Y_i$, respectively) can execute to completion by time-instant $2n$ in any schedule. We can think of all these jobs that complete execution by time-instant $2n$ as defining a truth assignment to the $2n$ variables $\{x_1, x_2, \ldots, x_n\} \cup \{y_1, y_2, \ldots, y_n\}$: boolean variable x_i is assigned **true** if job X_i is executed and **false** if $\neg X_i$ is executed, and analogously for the y_i variables.[6] Furthermore, we will see in Sections **??** and **??** that such a truth assignment happens in a manner that is consistent with the order and interpretation of the quantifiers upon the boolean variables.
- We will show, in Section **??** below, that the gadget representing each clause will complete by the deadline if and only if at least one of the literals in the clause evaluates to **true** in the truth assignment defined as above. Therefore, the gadgets representing all the clauses can complete by the deadline if and only if the truth assignment defined above is a satisfying one for all the clauses.

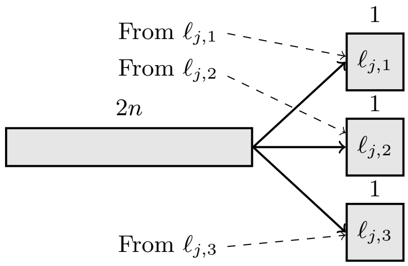
We detail the construction of the three kinds of gadgets in Sections **??**–**??**; in Section **??** we show that the C-DAG thus obtained is feasible if and only if Expression 1 is true, and hence this is indeed a polynomial-time reduction from QBF to C-DAG feasibility analysis.

4.1 Gadget for representing the clause $(\ell_{j,1} \vee \ell_{j,2} \vee \ell_{j,3})$

For the j'th clause $(\ell_{j,1} \vee \ell_{j,2} \vee \ell_{j,3})$, we have four jobs with precedence constraints as depicted in Figure **??**, all of which are assigned to a single dedicated processor. The WCET of each job is written above the job in Figure **??**. We will say that each of the three unit-sized jobs ***?represents?*** one of the three literals in the clause. Observe that the sum of the WCETs of the four jobs is $2n + 1 + 1 + 1 = \mathbf{2n + 3}$, which equals the deadline D; since all

[5] I.e., can x_1 be assigned some value such that regardless of whether y_1 is assigned **true** or **false**, x_2 be assigned some value such that. . . , such that each of the m conjuncts has some literal assigned **true**?

[6] If neither X_i nor $\neg X_i$ (neither Y_i nor $\neg Y_i$, respectively) are executed for any i, the truth assignment will be a partial one.



These four jobs are all assigned to the same processor; no other jobs are assigned to this processor. The WCET of each job is written above the job (i.e., the job with no predecessors has WCET $= 2n$ and the other three jobs each have WCET $= 1$). Each of the unit-sized jobs represents a literal of the clause; the dotted lines represent edges from the jobs that correspond to the literals (the notions of *representation* and *correspondence* are explained in Section 4).

Figure 3 **?**Gadget**?** representing the j'th clause.

these jobs are assigned to the same processor the processor must therefore never idle over $[0, D]$ in schedules that meet the deadline. This enforces the following schedule for these jobs:

1. the job with WCET $2n$ must execute over the interval $[0, 2n]$, and
2. at least one of the three unit-sized jobs, each of which has one additional input edge from the job corresponding to the literal that it represents, must become eligible to execute at time-instant $2n$.

Equivalently, in order for the part of the C-DAG we are constructing that is represented by this gadget to complete by the deadline, it is necessary that the truth assignment defined by the X_i, the $\neg X_i$, the Y_i and the $\neg Y_i$ jobs that completed execution by time-instant $2n$ have at least one of the literals $\ell_{j,1}$, $\ell_{j,2}$, and $\ell_{j,3}$ assigned the value true. I.e., this truth assignment must be a satisfying one for the clause $(\ell_{j,1} \vee \ell_{j,2} \vee \ell_{j,3})$.

Hence all m gadgets of the form depicted in Figure **??**, constructed for all m clauses in Expression 1, can complete by the deadline if and only if the truth assignment defined by the X_i, the $\neg X_i$, the Y_i and the $\neg Y_i$ jobs that completed execution by time-instant $2n$ is a satisfying one for each of the clauses in the QBF given in Expression 1. This is formally stated in Fact **??**:

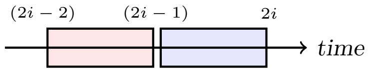
▶ **Fact 1.** A schedule can complete the jobs representing (as depicted in Figure **??**) all m clauses by the deadline $D = 2n + 3$ if and only if the truth assignment, defined by the jobs in $\bigcup_{1 \le i \le n} \{X_i, \neg X_i, Y_i, \neg Y_i\}$ that have executed to completion by time-instant $2n$ in the schedule, is a satisfying assignment for all the clauses. ◀

Requirements of the remaining gadgets. The remainder of the C-DAG —i.e., the gadgets for the x_i and the y_i boolean variables— must ensure that this truth assignment that is defined by the X_i, the $\neg X_i$, the Y_i and the $\neg Y_i$ jobs that completed execution by time-instant $2n$ is an accurate reflection of the alternating quantifiers in Expression 1:

$$\exists x_1 \; \forall y_1 \; \exists x_2 \; \forall y_2 \; \ldots \exists x_n \; \forall y_n$$

This desired alternation of quantifiers is achieved by ensuring the C-DAG is constructed to enforce the requirement that for each i, $1 \le i \le n$,

1. Prior to time-instant $2n$, in any correct schedule the scheduler can execute the pair of jobs X_i and $\neg X_i$, both of which are assigned to the same processor, only over the interval $[2i - 2, 2i - 1]$ — see Figure **??**. Therefore, it can choose to execute only one of this pair of jobs to completion prior to time-instant $2n$. (We will also see that it can execute the other job in the pair over $[2n, 2n + 1]$; hence both complete by time-instant $2n + 1$.)
2. Prior to time-instant $2n$, in any correct schedule the scheduler can execute only one of the pair of jobs Y_i and $\neg Y_i$, over the interval $[2i - 1, 2i]$ — see Figure **??**. *The decision as to*



The scheduler may choose to execute one of $\{X_i, \neg X_i\}$ over the interval $[2i-2, 2i-1]$. Run-time evaluation of conditional constructs enables only one of $\{Y_i, \neg Y_i\}$ to execute over interval $[2i-1, 2i]$.

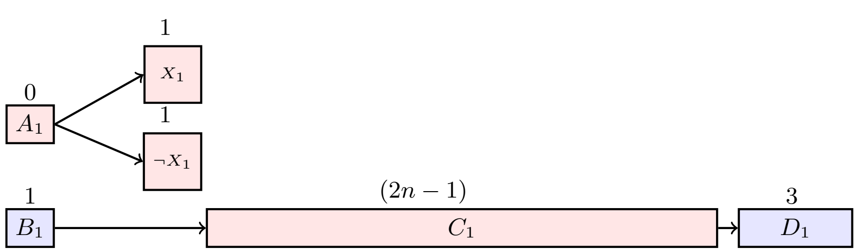
Figure 4 Illustrating the schedule over $[2i-2, 2i]$ for each i.

which job in the pair is able to execute over $[2i-1, 2i]$ *is not made by the scheduler, but is determined during run-time based on whether certain conditional constructs evaluate to true or false.* (We will also see that the scheduler can execute the other job in the pair over the time-interval $[2n, 2n+1]$; hence both the jobs complete by time-instant $2n+1$.) The existential quantification ($\exists$) of the x_i variables is reflected by the fact that the scheduler gets to decide whether to execute X_i or $\neg X_i$ over the interval $[2i-2, 2i-1]$, while the universal quantification ($\forall$) of the y_i variables is reflected by the fact that the environment (i.e., run-time conditions) determines which of Y_i or $\neg Y_i$ to execute, and the scheduler must make subsequent scheduling decisions for both outcomes (i.e., regardless of whether Y_i or $\neg Y_i$ is the job that was selected) by the environment. Notice that the order of the quantifiers is also maintained: the scheduler must decide to execute one of $\{X_i, \neg X_i\}$ *before* one of $\{Y_i, \neg Y_i\}$ is scheduled. And *after* one of $\{Y_i, \neg Y_i\}$ is chosen for execution by the environment, the scheduler must decide to schedule one of $\{X_{i+1}, \neg X_{i+1}\}$, and so on. In this manner the truth assignment to the variables $\{x_1, x_2, \ldots, x_n\} \cup \{y_1, y_2, \ldots, y_n\}$ that is defined by the schedule based on the jobs that complete execution by time-instant $2n$ reflects the order and alternation of the quantifiers in Expression 1.

It remains to describe how these restrictions on the execution of the $X_i, \neg X_i, Y_i$, and $\neg Y_i$ jobs in a manner that reflects the order and nature of the quantifiers is enforced — this we do in describing our other two kinds of gadgets. As stated previously, we will have one gadget for each x_i variable and one for each y_i variable; each gadget is defined on a unique set of jobs that are assigned to a unique set of processors. Our C-DAG is the union of all $2n$ of these gadgets and the m subgraphs of the form of Figure **??** (one per clause).

4.2 Gadget for enforcing the desired execution of X_i and $\neg X_i$

We first discuss the instantiation of this gadget for $(i \leftarrow 1)$, before subsequently describing the general case. The four jobs labeled A_1, B_1, C_1 and D_1 depicted below serve to ensure that prior to time-instant $2n$ the scheduler can execute the two jobs labeled X_1, $\neg X_1$ only over the time-interval $[0, 1]$ in any correct schedule.



The jobs $A_1, X_1, \neg X_1$, and C_1 are all assigned to one processor, while B_1 and D_1 are both assigned to another processor; furthermore, these two processors are used for no other purpose. (In these diagrams vertex colors encode their processor assignments.) Since the chain of jobs $B_1 \to C_1 \to D_1$ has cumulative WCET $1 + (2n-1) + 3 = \boldsymbol{2n+3}$ which is

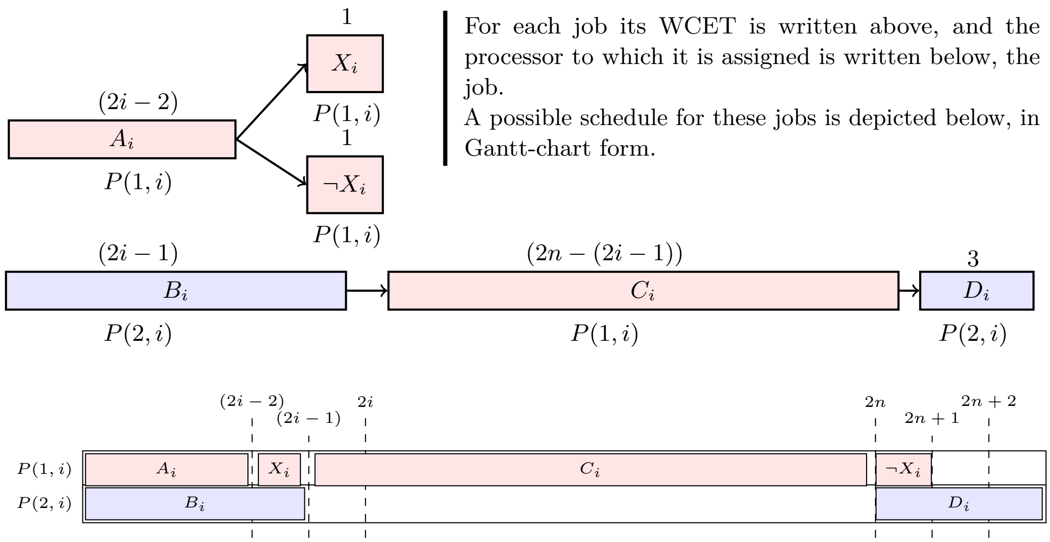


Figure 5 Gadget for X_i, comprising the four jobs A_i–D_i plus the two jobs X_i and $\neg X_i$, and the four edges shown, assigned to the two processors $P(1,i)$ and $P(2,i)$.

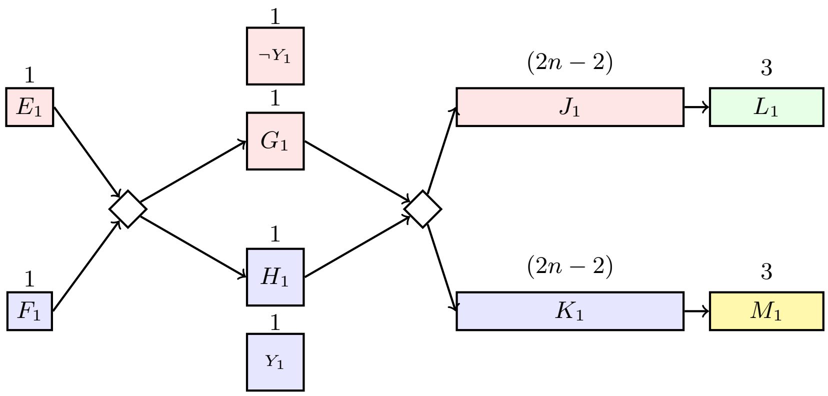
equal to the deadline D, these three jobs must execute without interruption. Hence in any correct schedule the processor shared by jobs $A_1, X_1, \neg X_1$, and C_1 is only available to jobs X_1 and $\neg X_1$ during the interval $[0,1]$, and after time $2n$. Thus at most one of these jobs may complete execution prior to time-instant $2n$, and this job must do so by executing over the interval $[0,1]$. (We point out that the other one may execute over the time-interval $[2n, 2n+1]$ and thereby complete by time-instant $2n+1$.)

The gadget depicted in Figure **??** generalizes the one described above for all i, $1 \leq i \leq n$. In this figure the two processors upon which the jobs are to execute are named as $P(1,i)$ and $P(2,i)$ – the processor to which each job is assigned is written below the job. A correct schedule for the jobs upon these two processors is depicted as a Gantt chart below the gadget.

4.3 Gadget for enforcing the desired execution of Y_i and $\neg Y_i$

As with the x_i's above, we first discuss the instantiation of this gadget for $(i \leftarrow 1)$; we will subsequently generalize to arbitrary i. The eight jobs labeled E_1, F_1, G_1,H_1, J_1, K_1, L_1, and M_1 along with one conditional construct,[7] and ten edges as depicted below, together ensure that at most one of the two jobs labeled Y_1, $\neg Y_1$, execute over the time-interval $[1,2]$ in any correct schedule while the other must execute after time-instant $2n$; furthermore, which of Y_1, $\neg T_1$ executes over $[1,2]$ is determined not by the scheduler but by which branch of the conditional construct ends up being executed during run-time.

[7] Recall that in this paper we are assuming that the two nodes demarcating the start and the end of a conditional construct each have WCET zero.



These ten jobs (E_1–M_1, plus the jobs Y_1 and $\neg Y_1$) are assigned to four processors in the following manner; no other jobs are assigned to any of these four processors[8];

- Jobs $E_1, G_1, \neg Y_1$ and J_1 are assigned to one processor.
- Jobs F_1, H_1, Y_1 and K_1 are assigned to a second processor;
- Job L_1 is assigned to a third processor; and job M_1 to a fourth processor.

Let us first suppose that during some execution of this C-DAG the conditional construct takes the upper branch (i.e., causes job G_1 to execute).

- Since the WCETs of the chain of jobs $E_1 \to G_1 \to J_1 \to L_1$ sum to the deadline $3n + 3$, this chain of jobs must execute without interruption in any correct schedule. This in turn implies that job $\neg Y_1$, which is assigned to the same processor as jobs E_1, G_1, and J_1, cannot execute prior to time-instant $2n$. (It may execute over the interval $[2n, 2n + 1]$ since there are no other jobs assigned to its processor.)
- In order for the chain $E_1 \to G_1 \to J_1 \to L_1$ to be able to execute without interruption, job F_1 must execute over the time-interval $[0, 1]$. Furthermore, the chain of jobs $K_1 \to M_1$ is only eligible to execute after the conditional construct completes: this happens when job G_1 completes (at time-instant 2). Note that jobs K_1 and L_1 must now execute without interruption over the interval $[2, 2n+3]$ in order to meet the deadline $D = 2n+3$. Therefore, the processor shared by jobs F_1, H_1 (which does not need to execute when the conditional construct takes the upper branch), Y_1, and K_1, is only free over the interval $[1, 2]$ prior to time-instant $2n$; this implies that the job Y_1 must execute over the interval $[1, 2]$ if it is to complete prior to time-instant $2n$.

When the conditional construct takes the lower branch and causes H_1 to execute, the situation mirrors the one above: job $\neg Y_1$ may execute over the interval $[1, 2]$ but job Y_1 may only execute after time-instant $2n$. Summarizing, we conclude that *one of the two jobs $Y_1, \neg Y_1$ may execute over the interval $[1, 2]$ and the other may execute over $[2n, 2n + 1]$; the determination as to which does which is made during run-time based on whether the conditional construct evaluates to true or false.*

The gadget depicted in Figure **??** generalizes the one described above for all i, $1 \leq i \leq n$. In this figure the four processors upon which the jobs are to execute are named as $P(3, i), P(4, i), P(5, i)$ and $P(6, i)$; as in Figure **??**, the processor to which each job is assigned is again written below the job. A correct schedule for the jobs upon these four processors is depicted as a Gantt chart below the gadget.

---

[8] As in Figure **??**, processor assignments are color-coded in this diagram. (Note that a fresh set of processors is used for each gadget and hence these colors do not ?carry over? from Figure **??**.)

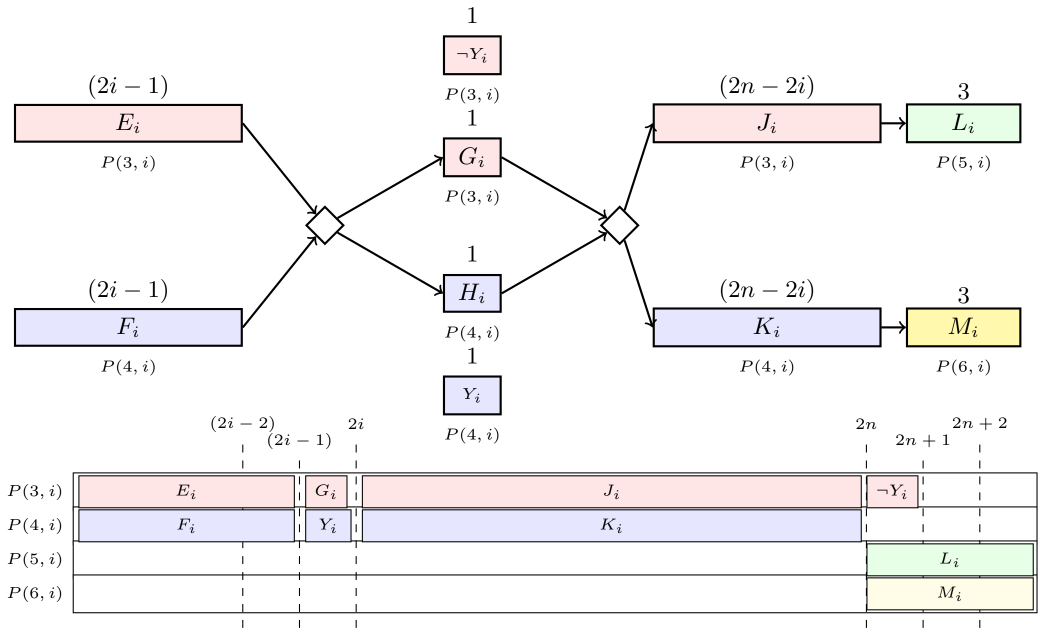


Figure 6 Gadget for Y_i (discussed in Section **??**).

The restrictions upon the execution of the jobs $X_i, \neg X_i, Y_i$, and $\neg Y_i$ that are enforced by the gadgets of Figure **??** and Figure **??** are stated in Facts **??** and **??** below (also see Figure **??**):

▶ **Fact 2.** For each i, $1 \leq i \leq n$, a scheduler may complete at most one of the two jobs $\{X_i, \neg X_i\}$ by time-instant $2n$ in any correct schedule. The choice as to which of these two jobs (if any) to complete by time-instant $2n$ must be made by the scheduler *after* it has already been decided which of the jobs $\left(\bigcup_{1\leq j<i}\{X_j, \neg X_j, Y_j, \neg Y_j\}\right)$ will complete by time instant $2n$.

▶ **Fact 3.** For each i, $1 \leq i \leq n$, a scheduler may complete at most one of the two jobs $\{Y_i, \neg Y_i\}$ by time-instant $2n$. The determination as to which of these two jobs (if either) to complete by time-instant $2n$ is made based on the outcome of the execution of a conditional construct during run-time, *after* it has already been decided which of $\left(\bigcup_{1\leq j<i}\{X_j, \neg X_j, Y_j, \neg Y_j\} \bigcup \{X_i, \neg X_i\}\right)$ will complete by time instant $2n$. ◀

4.4 Putting the pieces together

Consider now the truth assignment to the $2n$ variables $\{x_1, x_2, \ldots, x_n\} \cup \{y_1, y_2, \ldots, y_n\}$ that is defined by the schedule over $[0, 2n]$ in the following manner: for each i, $1 \leq i \leq n$, boolean variable x_i is assigned true if job X_i is executed and false if $\neg X_i$ is executed, and boolean variable y_i is assigned true if job Y_i is executed and false if $\neg Y_i$ is executed. By Fact **??**, a value is assigned by the scheduler to x_i in this assignment *after* values have been determined for x_j and y_j variables for all $j < i$, while by Fact **??** the value of y_i that is determined by the execution of conditional constructs at run-time happens *after* values have been determined for x_j and y_j variables for all $j < i$, as well as after the value of x_i has been assigned by the scheduler. Fact **??** follows.

▶ **Fact 4.** The truth assignment to the x_i and y_i variables defined by the execution of $X_i, \neg X_i, Y_i$, and $\neg Y_i$ jobs over $[0, 2n]$ is done in a manner that is compliant with the order of alternation of quantifiers in Expression 1. ◀

Summarizing the reduction. We have seen that the DAG we construct for a given quantified boolean formula

$$\exists x_1\ \forall y_1\ \exists x_2\ \forall y_2\ \ldots \exists x_n\ \forall y_n \bigwedge_{j=1}^{m} (\ell_{j,1} \vee \ell_{j,2} \vee \ell_{j,3})$$

comprises

1. For each of the m clauses, a sub-graph with four vertices and six edges as depicted in Figure **??** that is to execute upon a single processor;
2. For each of the n x_i variables, a sub-graph with the six vertices and four edges as depicted in Figure **??** that is to execute upon two processors; and
3. For each of the n y_i variables, a sub-graph with the ten vertices, one conditional construct, and ten edges as depicted in Figure **??** that is to execute upon four processors.

It is easily seen that the reduction from quantified boolean formula to DAG is a polynomial-time one: the resulting DAG has $(4m+16n)$ vertices, n conditional constructs, and $(6m+14n)$ edges, and is to be scheduled upon $(m + 6n)$ processors, and that it can be obtained in polynomial time from the quantified boolean formula.

▶ **Lemma 1.** *If Expression 1 is true, then the C-DAG constructed above can be scheduled to always complete by its deadline.*

Proof. Suppose that Expression 1 is true. This implies that variable x_1 can be assigned a value such that for every assignment of value to y_1 the formula

$$\exists x_2 \forall y_2 \exists x_3 \ldots \bigwedge_{j=2}^{m} (\ell_{j,1} \vee \ell_{j,2} \vee \ell_{j,3})$$

is true. If the assigned value to x_1 is true (false) then the scheduler completes job X_1 $(\neg X_1)$ by time $2n$; then, when the outcome of the first conditional construct is known, the job from amongst $\{Y_1, \neg Y_1\}$ that can be completed by time $2n$ is scheduled. By Fact **??** this decision is made before the scheduler gets to decide which job of the jobs amongst $\{X_2, \neg X_2\}$ will complete by time $2n$.

By repeated applications of Facts **??** and **??**, we can ensure that the jobs amongst the X_i, $\neg X_i$, Y_i, and $\neg Y_i$ jobs that execute over the interval $[0, 2n]$ mimic each truth assignment to the boolean variables $\{x_1, x_2, ..., x_n\} \cup \{y_1, y_2, ..., y_n\}$ that are made in a manner consistent with the alternation of quantifiers in Expression 1. It follows from Fact **??** that the gadget representing each clause (these are the gadgets depicted in Figure **??**) will complete by the deadline for each such truth assignment.

◀

▶ **Lemma 2.** *If the C-DAG constructed above can be scheduled to always complete by its deadline, then Expression 1 is true.*

Proof. Suppose that the C-DAG that we have constructed can be scheduled to always complete by its deadline, for all possible evaluations of the n conditional constructs (recall that one conditional constructs is present in each of the gadgets described in Section **??**, and these are the only conditional constructs) in it.

Consider the schedule for any one of the 2^n different possible combinations of outcomes for the execution of these n conditional constructs. Fact **??** ensures that the truth assignment defined by the jobs in $\bigcup_{1\leq i\leq n}\{X_i, \neg X_i, Y_i, \neg Y_i\}$ that have executed to completion by time-instant $2n$ in this schedule is a satisfying assignment for all the clauses in Expression 1; by Fact **??**, this truth assignment is compliant with the order of alternation of quantifiers in Expression 1.

Our premise is that the C-DAG completes by its deadline for each of the 2^n different possible combinations of outcomes for the execution of the conditional constructs. It follows that each clause in Expression 1 evaluates to **true** in the corresponding truth assignments defined by the jobs in $\bigcup_{1\leq i\leq n}\{X_i, \neg X_i, Y_i, \neg Y_i\}$ that have executed to completion by time-instant $2n$. Finally, it follows from Fact **??** that these 2^n different possible combinations of outcomes of the execution of the conditional constructs represent all possible interpretations of the universal quantifications of the y_i variables. The lemma follows. ◀

Lemmas **??** and **??** together establish that the C-DAG feasibility problem is PSPACE-hard when each job is pre-assigned to a particular processor. We have already seen that this problem is in PSPACE; this therefore completes the proof of Theorem 1.

5 A More Tractable Special Case

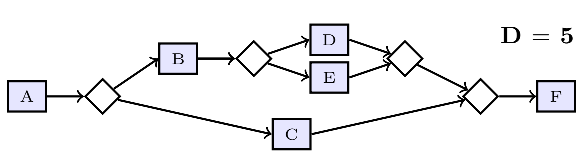
Theorem 1 above tells us that we are unlikely to be able to efficiently (i.e., in polynomial time) reduce the problem of determining whether a C-DAG is feasible to the problem of solving one, or even polynomially many, ILPs. In this section we will show that for C-DAGs satisfying the additional restriction that *the number of conditional constructs is bounded by a constant*, the feasibility-analysis problem can indeed be polynomial-time reduced to a single ILP. Our method of showing this is indirect, and based upon the following reasoning.

- As mentioned in Section 3, it is NP-complete to determine whether an ILP has a solution [**?**]. It follows from definition that a consequence of a problem being NP-complete is that all other problems in NP can be reduced to it in polynomial time.
- Hence in order to show that feasibility analysis for C-DAGs in which the number of conditional constructs is bounded by some constant can be reduced to an ILP in polynomial time, it suffices to show that this feasibility analysis problem is in NP.

Below we will show that this problem is indeed in NP. We do so by appealing to the definition of the complexity class NP: as stated in Section 3, a problem is defined to be in NP if a claimed solution to any problem instance can be *verified* by an algorithm with running time polynomial in the size of the instance. Hence we will describe a *verification algorithm* [**?**, page 1063] that accepts as input a C-DAG and a **?**certificate**?** claiming to show that the C-DAG is feasible, and verifies, in time polynomial in the representation of the C-DAG, whether the certificate does indeed show feasibility.[9]

The certificate for a C-DAG instance with k conditional constructs will be an explicit enumeration of the at most 2^k individual schedules, one each for the vertices that must be

[9] We acknowledge that the following description of this verification algorithm is at a high level and somewhat **?**hand-wavy**?**; however we believe it is adequate for conveying the main ideas as to what information is contained in the certificate, and how the verifier checks this information.



1.	A	◇	B	◇	D	F
2.	A	◇	B	◇	E	F
3.	A	◇	C	F		

Figure 7 A C-DAG instance with two conditional constructs. Each vertex has WCET=1, and all are assigned to the same processor. Its **?**certificate**?** of feasibility is shown on the right: it comprises three schedules, all of which are identical until the first conditional construct is executed (depicted as a ◇). The top two schedules, which correspond to the upper branch being taken, are further identical until the second conditional construct is executed.

executed upon each possible combination of outcomes of the execution of the k conditional expressions. The number of schedules in the certificate may be fewer than 2^k since not all outcomes may be possible – e.g., the C-DAG depicted in Figure **??** has two conditional constructs but only 3 possible outcomes. A certificate with the three schedules is provided in Figure **??** for when this C-DAG is to be implemented on a single processor.

Given such a certificate, the *verification algorithm* verifies that

1. Each schedule in the certificate is indeed a feasible schedule for the vertices that must be executed upon some possible outcome of the execution of the conditional constructs.
2. The sets of vertices that must be executed upon all possible outcomes have schedules in the certificate.
3. The schedules in the certificate are *consistent* in the following sense:
   - They are all identical (i.e., schedule the same jobs at the same instants) until the end of the first execution of a conditional expression (the diamond-shaped node marking the beginning of a conditional construct).
   - After then the set of schedules is partitioned into two subsets, one representing each of the two possible outcomes of the execution of that conditional expression.
   - Each of these two subsets must satisfy the two properties above: all schedules in the subset are identical up to the next execution of a conditional expression, and split into two sets representing the schedules for the two different outcomes thereafter.
   - This repeats until each set contains a single schedule.

This establishes that C-DAG feasibility analysis is in NP, and can therefore be reduced in polynomial time to the NP-complete problem ILP. We are currently working on developing such a polynomial-time algorithm: although the main ideas are fairly straightforward – in essence, use integer decision variables to specify the different schedules in the certificate and write constraints to enforce the requirements listed above as being checked by the verification algorithm, there are a lot of rather tedious details that must be enumerated.

The number of variables and the number of constraints in the ILP depend upon the number of schedules in the certificate. Notice the relationship between the number of conditional constructs k and the number of schedules in the certificate (at most 2^k) — this suggests that ILPs with fewer conditional constructs are likely to be representable using smaller ILPs.

6 Context and Conclusions

Real-time scheduling theory has begun considering the use of ILP solvers to obtain efficient algorithms for solving feasibility analysis problems. Several schedulability analysis problems have recently been solved by representing them as ILPs (e.g.,[?, ?]); here we have shown that an important problem *cannot* be solved efficiently in this manner. We also note some additional implications of our main technical results.

1. Observe that the workload model for heterogeneous multiprocessor platforms is unchanged from the one for identical multiprocessors for typed systems (those in which all vertices are pre-assigned to individual processors). Therefore *our results for typed systems also hold for heterogeneous multiprocessors.* Many are also applicable to the recently proposed more general Heterogeneous Parallel Conditional (HPC) DAG model [?].
2. Most solvers that are used in system design (including SAT solvers, many SMT [?] solvers, etc.) actually solve problems that are in NP.[10] Hence our main negative conclusion holds for all these solvers as well: they're unlikely to be helpful for C-DAG feasibility analysis.
3. In this work we have required that problems be reducible to ILPs in polynomial time in order to be considered tractable. As an alternative, we could have instead required that there be a polynomial-*sized* ILP representation. However, this alternative definition is unsatisfactory: one could conceivably determine feasibility for any instance of a problem via exhaustive enumeration by taking inordinate amounts of time, and then represent its feasibility as a simple ILP of just one or two variables and constraints which has a solution if and only if the instance is feasible. Hence, one could argue that just about any feasibility-analysis problem can be represented by a small ILP: the true measure of tractability is how rapidly such an ILP can be obtained.

Some Related Work. ILP solvers have previously been used in real-time system design and analysis — see, e.g., [?, ?]. But in the real-time scheduling theory community, where the focus has primarily been on obtaining efficient algorithms with polynomial or pseudo-polynomial running times, ILP-based techniques have traditionally not found much favor for obvious reasons. The recent dramatic improvements in performance of modern solvers mentioned in Section 3 is starting to change this, and the real-time scheduling theory community has begun to investigate the use of ILP-based methods [?, ?, ?, ?].

Future work. We have established a conceptual and technical framework for both showing problems to not be efficiently solvable using ILP solvers, and for identifying restricted versions that are so solvable. We plan to apply our framework to better demarcate the boundary between what if efficiently solvable and what is not with ILP solvers, as well as extend the framework to answer additional questions of interest. For a start, we plan to investigate notions of *approximability* — we could, e.g., seek sufficient ILP-based feasibility-analysis algorithms of the following kind: *given an instance generate, in polynomial time, an ILP such that (i) if it is feasible, then the instance is feasible upon unit-speed processors; and (ii) if it is infeasible, then the instance is not feasible on speed-s processors (for some $s \leq 1$).*

With regards to C-DAG feasibility, we have identified one specific structural property —restrict the number of conditional constructs— that enables efficient solution via ILP's. The reason such instances are efficiently solved is that certificates attesting to their feasibility contain relatively few schedules. We are currently identifying other such structural properties of C-DAGs that also possess this property (of having ?small? certificates of feasibility).

[10] One important reason for this is that the results returned by such solvers can be *verified* efficiently, in polynomial time. Solutions obtained by using solvers that solve problems not in NP must either be accepted ?on faith?, or inordinate amounts of time are required to validate their correctness.

References

1 Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 825–885. IOS Press, 2009. URL: `https://doi.org/10.3233/978-1-58603-929-5-825`, `doi:10.3233/978-1-58603-929-5-825`.

2 Sanjoy Baruah. Scheduling DAGs when processor assignments are specified. In *Proceedings of the Twenty-Fifth International Conference on Real-Time and Network Systems*, RTNS '20, New York, NY, USA, 2020. ACM.

3 Sanjoy Baruah, Marko Bertogna, and Giorgio Buttazzo. *Multiprocessor Scheduling for Real-Time Systems*. Springer Publishing Company, Incorporated, 2015.

4 Sanjoy Baruah, Vincenzo Bonifaci, Renato Bruni, and Alberto Marchetti-Spaccamela. Ilp-based approaches to partitioning recurrent workloads upon heterogeneous multiprocessors. In *Proceedings of the 2016 28th EuroMicro Conference on Real-Time Systems*, ECRTS '16, Toulouse (France), 2016. IEEE Computer Society Press.

5 Sanjoy Baruah, Vincenzo Bonifaci, and Alberto Marchetti-Spaccamela. The global EDF scheduling of systems of conditional sporadic DAG tasks. In *Proceedings of the 2014 26th Euromicro Conference on Real-Time Systems*, ECRTS '15, pages 222–231, Lund (Sweden), 2015. IEEE Computer Society Press.

6 Sanjoy Baruah, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, Leem Stougie, and Andreas Wiese. A generalized parallel task model for recurrent real-time processes. In *Proceedings of the IEEE Real-Time Systems Symposium*, RTSS 2012, pages 63–72, San Juan, Puerto Rico, 2012.

7 Sanjoy K. Baruah, Vincenzo Bonifaci, Renato Bruni, and Alberto Marchetti-Spaccamela. ILP models for the allocation of recurrent workloads upon heterogeneous multiprocessors. *Journal of Scheduling*, Dec 2018. URL: `https://doi.org/10.1007/s10951-018-0593-x`, `doi:10.1007/s10951-018-0593-x`.

8 Slim Ben-Amor. *Multicore Scheduling of Dependent Tasks with Probabilistic Execution Times*. PhD thesis, Sorbonne Université, 2021.

9 T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, third edition, 2009.

10 Jose Fonseca, Vincent Nelis, Gurulingesh Raravi, and Luis Miguel Pinho. A Multi-DAG model for real-time parallel applications with conditional execution. In *Proceedings of the ACM/SIGAPP Symposium on Applied Computing (SAC)*, Salamanca, Spain, April 2015. ACM Press.

11 Robert A. Hearn and Erik D. Demaine. *Games, puzzles and computation*. A K Peters, 2009.

12 Klaus Jansen. Analysis of scheduling problems with typed task systems. *Discrete Applied Mathematics*, 52(3):223 – 232, 1994.

13 R. Karp. Reducibility among combinatorial problems. In R. Miller and J. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, New York, 1972.

14 Jing Li, Abusayeed Saifullah, Kunal Agrawal, Christopher Gill, and Chenyang Lu. Analysis of federated and global scheduling for parallel real-time tasks. In *Proceedings of the 2012 26th Euromicro Conference on Real-Time Systems*, ECRTS '14, Madrid (Spain), 2014. IEEE Computer Society Press.

15 Y. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16(12):1477–1487, 1997. `doi:10.1109/43.664229`.

16 Alessandra Melani, Marko Bertogna, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, and Giorgio Buttazzo. Response-time analysis of conditional DAG tasks in multiprocessor systems. In *Proceedings of the 2014 26th Euromicro Conference on Real-Time Systems*, ECRTS '15, pages 222–231, Lund (Sweden), 2015. IEEE Computer Society Press.

17 L. Stockmeyer. The polynomial-time hierarchy. *Theoretical Computer Science*, 3:1–22, 1976.

18 J. Ullman. NP-complete scheduling problems. *Journal of Computer and System Sciences*, 10(3):384 – 393, 1975.

19 Reinhard Wilhelm. Why AI + ILP Is Good for WCET, but MC Is Not, Nor ILP Alone. pages 309–322, 01 2004. doi:10.1007/978-3-540-24622-0_25.

20 C. Wrathall. Complete sets and the polynomial-time hierarchy. *Theoretical Computer Science*, 3:23–33, 1976.

21 Houssam-Eddine Zahaf, Nicola Capodieci, Roberto Cavicchioli, Marko Bertogna, and Giuseppe Lipari. The HPC-DAG task model for heterogeneous real-time systems. *IEEE Transactions on Computers*, pages 1–1, 2020.