Real-Time Systems
https://doi.org/10.1007/s11241-022-09380-z

Feasibility analysis for HPC-DAG tasks

Sanjoy Baruah[1]

Accepted: 7 March 2022

Abstract
The HPC-DAG task model is a very general and feature-rich model that was developed for the purposes of representing real-time applications that are intended for implementation upon heterogeneous multiprocessor platforms. The computational complexity of determining feasibility for a task specified according to this model is considered, both for the general model and for some practically meaningful restricted variants.

Keywords Feasibility analysis · Heterogeneous Multiprocessors · Conditional directed acyclic graphs · Alternative nodes

1 Introduction

The *Heterogeneous Parallel Condition Directed Acyclic Graph* (HPC-DAG) task model was recently proposed by Houssam-Eddine et al. (2020) for the purposes of modeling the execution of complex real-time application systems that are to be implemented upon advanced commercial computing platforms featuring heterogeneous processing units and compute accelerators (e.g., different kinds of CPUs, GPUs, Deep Learning Accelerators, Programmable Vision Accelerators, etc.). The model, which we will describe in Sect. 3, is a DAG (directed acyclic graph) based one enhanced to include a number of interesting features that "allows the system designer to specify alternative implementations of a software component for different processing engines, as well as conditional branches to model if-then-else statements" Houssam-Eddine et al. (2020). That is, it allows for (i) the specification of *alternative implementations* of some desired functionality such that only one of the specified alternatives needs to be selected (prior to run-time) and implemented in order to achieve the desired functionality; and (ii) the modeling of *conditional*

Sanjoy Baruah
baruah@wustl.edu

[1] Washington University in Saint Louis, Campus Box 1045, Saint Louis, MO 63130, USA

*(*if-then-else*) statements* for which the conditional expressions determining which branch to take are evaluated during run-time.

Houssam-Eddine et al. argue pursuasively (2020) that the expressive power of the HPC-DAG model makes it an attractive choice for specifying application systems in a manner that facilitates the exploration of design alternatives: by specifying multiple alternative possible implementations of parts of a system, there is greater freedom to choose the most appropriate implementation of each part in order to meet desired system-wide properties. One particularly important desirable property for safety-critical real-time systems is *feasibility*: the ability to always meet all timing constraints during run-time. It has recently been shown (Baruah and Marchetti-Spaccamela 2021) that determining feasibility for tasks specified in the conditional DAG (C-DAG) model, of which the HPC-DAG model is a strict generalization, is already a PSPACE-complete problem; it therefore follows that feasibility analysis is PSPACE-hard for HPC-DAG tasks as well. In this paper we propose some practically meaningful restrictions on the semantics (in particular, the execution model) of HPC-DAG tasks that render the problem somewhat more tractable, and characterize the precise computational complexity of feasibility analysis for these restricted variants of the HPC-DAG model.

1.1 Organization

The remainder of this paper is organized as follows. In Sect. 2 we provide some background on computational complexity theory that is needed in order to follow along for the remainder of this paper. In Sect. 3 we describe the HPC-DAG model, provide some intuition of and insight into the various reasons why feasibility analysis is a particularly challenging problem for tasks specified in this model, and summarize the current state of our knowledge concerning the computational complexity of such feasibility analysis. In Sect. 4 we establish the computational complexity of feasibility analysis upon a restricted version of the HPC-DAG task model. In Sect. 5 we extend our techniques to the feasibility analysis problem for the general HPC-DAG task model but under a restricted execution model. We conclude in Sect. 6 by placing this work in context.

2 Some background from complexity theory

In this section we provide an informal review of some concepts from computational complexity theory Papadimitriou (1994); Arora and Barak (2009) that we will use in the remainder of this paper. The class $\underline{\text{P}}$ of problems that are known to be *solved* by algorithms with running time polynomial in the size of their inputs, and the class $\underline{\text{NP}}$ of problems for which claimed solutions can be *verified* by algorithms with running time polynomial in the size of their inputs, are (along with $\underline{\text{co}}\ \underline{\text{NP}}$, the class of problems whose complements are in NP) the foundational cornerstones of computational complexity theory. It is very widely believed—see Fig. 1—that P $\subsetneq$ NP (i.e., there are polynomial-time verifiable problems that cannot be solved in polynomial time);

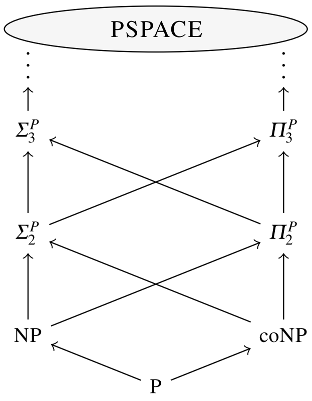

Fig. 1 Complexity classes: the polynomial hierarchy Stockmeyer (1976)

and that co NP $\neq$ NP (i.e., there are problems in NP that are not in coNP, and vice versa). The *polynomial-time hierarchy* Stockmeyer (1976) extends computational complexity theory beyond the classes P and NP by considering abstract computers equipped with an *oracle*: a "black box" that is able to solve a specific decision problem efficiently (for our purposes here, "efficiently" can be taken to mean "in polynomial time"). The complexity class $\underline{\Sigma_2^P}$ denotes the class of all problems that can be verified in polynomial time by an algorithm that is equipped with an oracle for solving some NP-complete problem. Similar to how coNP relates to NP, complexity class $\underline{\Pi_2^P}$ denotes the class of problems whose complement problems are in Σ_2^P. This idea is generalized for any $k \in \mathbb{N}$: Σ_k^P and Π_k^P are defined assuming access to an oracle that is complete for Σ_{k-1}^P. The relationship amongst these complexity classes is shown in Fig. 1 as a Hasse diagram depicting the subset relationship (i.e., an arrow $A \rightarrow B$ denotes that complexity class A is a subset of complexity class B: each problem that falls in complexity class A is also contained in complexity class B). It is widely believed that each such depicted containment is proper—i.e., each depicts the $\subsetneq$ relationship. The entire polynomial hierarchy is believed to be contained in the complexity class <u>PSPACE</u>, which 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.

3 The HPC-DAG task model

In this section we first provide a brief description of the HPC-DAG task model, emphasizing those aspects that are most relevant to deciding the computational complexity of feasibility analysis. Following this, in Sect. 3.1 we provide some intuition into the root causes of the challenges that arise in attempting to determine the feasibility of HPC-DAG tasks, and suggest some modifications to the model that render feaibility analysis somewhat more tractable in Sect. 3.2.

Workload models used in scheduling theory for representing recurrent real-time workloads that are to be implemented upon multiprocessor platforms should be capable of exposing the parallelism that may exist within these workloads. The *sporadic DAG model* (Baruah et al. 2012) (see Baruah et al. (2015, Chapter 21) for a text-book description) was proposed for this purpose. 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. (In this paper we restrict attention to tasks satisfying the additional constraint that $D \leq T$, i.e., to *constrained-deadline* tasks.) The task repeatedly releases *dag-jobs*, each of which is a collection of (sequential) jobs—releasing a dag-job corresponds to making all $|V|$ jobs available for execution (subject, of course, to their inter-job precedence constraints). 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 the execution of a sequential piece of code (a "job"), 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. If a dag-job is released at time-instant t then all $|V|$ jobs that were released at t must complete execution by time-instant $t + D$.

The **HPC-DAG** model (Houssam-Eddine et al. 2020) is obtained from the sporadic DAG model described above by incorporating three enhancements—*heterogeneous platforms*, *conditional execution*, and *implementation alternatives*. While the first two enhancements have previously been proposed the third is, to our knowledge, novel. We will discuss these three enhancements separately below.

3.1 Heterogeneity

The HPC-DAG model is intended for specifying application systems that are designed for execution upon multiprocessor platforms comprising different kinds of computing engines (henceforth, they will all be referred to as "*processors*" which may be different from each other—hence, heterogeneous). Recall that each vertex in G represents a sequential piece of code: it is reasonable to assume that this code is written to be implemented upon a particular kind of processor. The HPC-DAG model therefore associates a *tag* with each processor, and each vertex $v \in V$ is characterized by a tag identifying the kind of processor upon which it may execute.

3.2 Conditional execution

To enable the modeling of conditional branching, the HPC-DAG model incorporates the *conditional DAG (C-DAG)* task model that had been introduced (Baruah et al. 2015; Melani et al. 2015) as a generalization to DAG tasks. A conditional DAG task, too, is specified as a DAG $G = (V, E)$ plus a relative deadline D and a period T; it differs 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*. A conditional construct represents a point in the code where some conditional expression, whose outcome is not known beforehand and may differ upon different executions of the task, is executed and the subgraph that must be executed between this conditional vertex and its paired counterpart depends upon

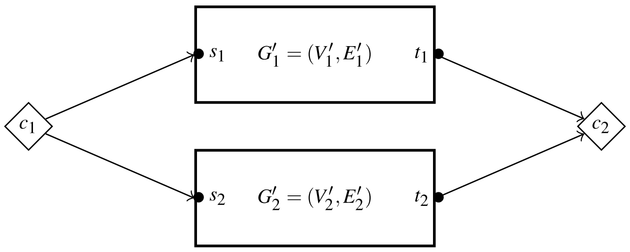

Fig. 2 A canonical conditional construct. Subgraphs $G'_1 = (V'_1, E'_1)$ and $G'_2 = (V'_2, E'_2)$ are disjoint, and vertices s_1 and t_1 (vertices s_2 and t_2, resp.) are the sole source vertex and sink vertex of the sub-graphs G'_1 (G'_2, resp.)

this outcome. Let (c_1, c_2) be such a pair in the DAG $G = (V, E)$ — see Fig. 2. Informally speaking, vertex c_1 can be thought of as representing a point in the code where a conditional expression is evaluated and, depending upon the outcome of this evaluation, control will subsequently flow along exactly one of several different possible paths in the code (in this paper we restrict attention to conditional constructs in which there are exactly two such outgoing edges, as in Fig. 2). It is required that all these different paths meet again at a common point in the code, represented by the vertex c_2. Edges (v_1, v_2) between pairs of vertices neither of which are conditional vertices represent precedence constraints exactly as in traditional sporadic DAGs, while edges involving conditional vertices 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 successor jobs becomes eligible to execute; it is not known beforehand which successor job may execute.
- Job c_2 begins to execute upon the completion of exactly one of its predecessor jobs.

There are additional syntactic rules concerning the structure of conditional DAG tasks; e.g., it is required that the two subgraphs reached by the outgoing edges from c_1 be disjoint from the remainder of the DAG (as stated in the caption of Fig. 2), that conditional constructs may be nested, etc., that we are not stating here because they are not needed in the remainder of this paper—please see Baruah et al. (2015); Melani et al. (2015) for a more complete description of the conditional sporadic DAG task model.

3.3 Implementation alternatives

The HPC-DAG task model introduces a further generalization to the sporadic DAG task model that, to our knowledge, has not been previously considered in the scheduling literature: alternative implementations for part of the task. This is achieved by designating certain matched pairs of vertices in V as *alternative vertices*. From a

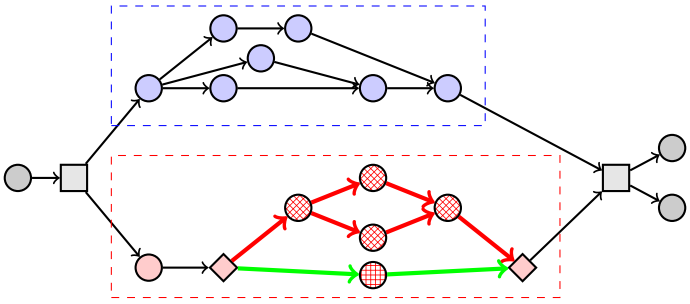

Fig. 3 Illustrating the HPC-DAG model (Houssam-Eddine et al. 2020): this task has one *conditional construct* and one pair of *alternative vertices*. (This task is discussed in Example 1.)

syntactic perspective their definition is essentially identical to the manner in which conditional vertices were defined. (For instance, the vertices c_1, c_2 in Fig. 2 could be considered to represent a matched pair of alternative vertices rather than a matched pair of conditional vertices.) However their interpretation is very different: they model alternative ways of implementing some functionality that must be achieved by some part of the task. Prior to run-time exactly one of the alternative implementations that are available between a matched pair of alternative vertices must be chosen. Once such a choice has been made for each pair of alternative vertices, we are left with a conditional DAG that is subsequently executed during run-time. Supposing vertices c_1 and c_2 in Fig. 2 were a matched pair of alternative vertices rather than of conditional vertices (as stated above, conditional vertices are syntactically identical to alternative vertices), the interpretation would be that either the subgraph G_1' or the subgraph G_2' needs to be selected, prior to run-time, as comprising a part of the conditional DAG that will be executed during run-time.

Example 1 Figure 3 illustrates some of the features of the HPC-DAG model. The circular vertices in this figure depict regular jobs—those representing the execution of sequential code. The square vertices (□) denote alternative choices—an implementor may choose to implement either the vertices contained in the upper dashed box or those contained in the lower dashed box. The diamond-shaped vertices (◇) denote a conditional construct. (The heterogeneity feature is not depicted in this example.)

If the system implementor chooses to implement the vertices contained in the lower dashed box then during each execution either the solitary vertex reached by the outgoing bottom arrow or the four vertices reached by the outgoing upper arrow, must be executed. The vertices that must be executed may differ upon different executions of the task. □

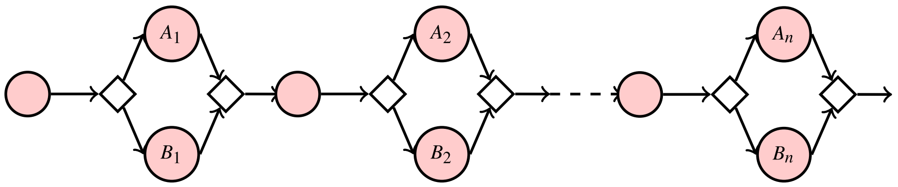

Fig. 4 Illustrating combinatorial explosion: this task may require the execution of any one of 2^n different collections of jobs

3.3.1 Some additional terminology

Houssam-Eddine et al. (2020) refer to a task specified in the HPC-DAG model as a "specification task", and the instantiation obtained by making a choice between each pair of alternative vertices as a "concrete task." That is, a *concrete task* is obtained from a *specification task* prior to run-time, by both (i) for each matched pair of alternative vertices, choosing an implementation from amongst the specified alternatives; and (ii) assigning each job to a particular processor of the kind specified by the tag characterizing the corresponding vertex.

3.4 Some challenges in scheduling HPC-DAG tasks

Given the specifications of a single HPC-DAG task and of the platform upon which this task is to be implemented, *feasibility analysis*, the problem we are considering in this paper, is to determine whether it is possible to choose from amongst the alternative implementations that are specified for the task such that the resulting conditional DAG task can be scheduled upon the specified platform in such a manner that the task deadline is always met. In this section we take a closer look at the HPC-DAG model in order to better understand the sources of difficulty in doing such feasibility analysis.

As stated above, selecting amongst implementation alternatives provided in an HPC-DAG task [a *specification task* in the terminology of Houssam-Eddine et al. (2020)] is done prior to run-time. However for conditional constructs [in the *concrete task* (Houssam-Eddine et al. 2020) that results from having made these selections] the choice as to which branch out of a conditional vertex to take is not made beforehand; rather, this decision is determined at run-time by the prevailing ambient state of the environment within which the system is executing. Since this state does not in general remain constant, the same conditional expression may evaluate differently during different executions of the task. This means that the same (concrete) task may require different subsets of its set of vertices to be executed during different executions of the task; it has been shown (Fonseca et al. 2015; Baruah et al. 2015; Melani et al. 2015) that this can lead to *combinatorial explosion*: exponentially many different combinations of outcomes are possible of the evaluation of the different conditional constructs in a single task, each of which may require a very different collection of jobs to be scheduled for execution. Consider, for instance, the conditional DAG task depicted in Fig. 4: it is evident that any of 2^n different collections of jobs may need to be executed during any particular execution of the task, depending upon which of the n conditional expressions evaluate to true and which to false.

This combinatorial explosion problem for conditional DAG tasks has previously been recognized (Fonseca et al. 2015; Baruah et al. 2015; Melani et al. 2015). However, there is an additional aspect to the difficulty of scheduling conditional DAG tasks that has received somewhat less attention: its inherently *on line* nature, arising from the fact that during run-time scheduling decisions for certain parts of the DAG task may need to be made without knowing how future conditional vertices will evaluate. Consider the following simple illustrative example adapted from Baruah and Marchetti-Spaccamela (2021).

Example 2 Figure 5 depicts a conditional DAG task that is the concrete task obtained from some HPC-DAG task (i.e., after the choices made available by all the alternative vertices have been made), for which a deadline of four is specified. In the figure the tag characterizing each vertex are listed above the vertices—is it is evident that there is only one possible job-processor assignment:

- the pair of jobs A and H must both execute upon processor P_1 ;
- the four jobs B, C, J, and K must all execute upon processor P_2;
- the pair of jobs D and F must both execute upon processor P_3; and
- the pair of jobs E and G must both execute upon processor P_4.

Job H has WCET 2 while the WCET for every other vertex is equal to one (except for the conditional vertices, which are assumed to have WCET zero).

Note that the chain of three jobs comprising A, one of D or E, and H has cumulative execution duration 4 and must hence execute uninterrupted throughout the interval [0, 4] if the deadline of four is to be met. Note also that the four jobs B, C, J, and K are all assigned to processor P_2; hence this processor must be kept busy throughout [0, 4] if the deadline of four is to be met. Let us now separately consider both possible outcomes of the execution of the conditional construct:

1. Suppose the execution of the conditional construct results in the upper branch being taken (i.e., D must be executed).
   - If job B had executed upon P_2 during [0, 1], then job C may execute upon it during [1, 2]. However, job F cannot execute upon its designated processor, P_3, during [1, 2] (since job D will be executing upon P_3 during this interval). This means that neither J nor K are ready to execute at time-instant 2 and hence processor P_2 must be idled during [2, 3]. This will result in a deadline miss.
   - If however job C were executed upon P_2 during [0, 1], then job B may execute upon it during [1, 2]. Additionally, job G may execute upon P_4 during [1, 2] (since the only other job assigned to P_4, job E, does not need to execute at all), and hence K is available to execute upon P_2 during [2, 3] (and J during [3, 4]).

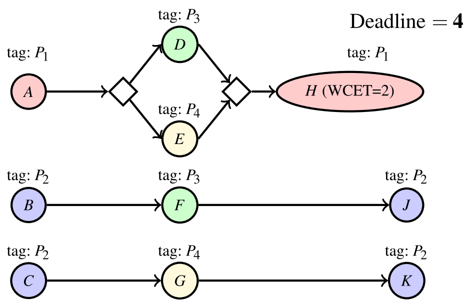

Fig. 5 A conditional DAG task that is the concrete task instantiation for some HPC-DAG specification task—see Example 2

Hence during the time-interval [0, 1] processor P_2 should be executing job C rather than job B in order that the deadline of four be met.

2. It may similarly be verified that if the execution of the conditional construct instead results in the lower branch being taken (i.e., E must be executed), then executing B during [0, 1] allows all deadlines to be met whereas instead executing C during this interval (i.e., during [0, 1]) results in a missed deadline.

We thus see that the "correct" job from amongst $\{B, C\}$ to execute upon processor P_2 during the time-interval [0, 1] is different for the two outcomes of the conditional expression. But the conditional expression is only evaluated *after* that interval has elapsed, and hence this information is revealed too late. □

Example 2 above illustrates the second challenge in scheduling conditional DAGs: since the collection of vertices (jobs) that need to be executed depends upon the outcome of the evaluation of the conditional expressions which only happens at run-time, this scheduling problem is an inherently *on-line* one.

3.5 A restricted execution model for conditional DAG tasks

We saw above (Sect. 3.1) that there are two factors, combinatorial explosion and the on-line manner in which information is revealed regarding which jobs need to be executed, that contribute to the challenge of determining feasibility for conditional DAG tasks. Considering both aspects together results in a highly intractable problem: it has recently been shown (Baruah and Marchetti-Spaccamela 2021) that feasibility analysis for conditional DAG tasks with vertices restricted to execute upon specified processors is PSPACE complete. Suppose now that a further restriction were placed upon the execution model for conditional DAG tasks, requiring that *the outcomes of the evaluation of all the conditional expressions in a task be known prior to beginning the execution of each instance (dag-job) of the task*. Under this restriction, the on-line nature of the problem is essentially defined away and only the combinatorial explosion problem remains: feasibility analysis reduces to determining whether all of the potentially exponentially many different collections of jobs that may need to be scheduled upon different executions of the task are all feasible or not.

From a pragmatic perspective, where might such a model be useful? Consider application systems that are designed to operate under a variety of different environmental conditions that can be described by assigning values to a number of orthogonal parameters, such that these environmental conditions change relatively slowly compared to the time-scale (the relative deadline and period) of the DAG task.[1] In such application systems *the conditional constructs serve as a compact representation of multiple possible different execution modes* for the system being modeled as a task, subject to the restriction that if a mode change is signaled whilst a dag-job is executing, the currently-executing dag-job executes under the prior mode and the new mode only applies to subsequent dag-jobs of the task.

As stated above, feasibility analysis for conditional DAGs has been shown (Baruah and Marchetti-Spaccamela 2021) to be PSPACE complete. Since the HPC-DAG model is even more general than the conditional DAG model (recall that the HPC-DAG model includes all features of the conditional DAG model plus the option of using alternative vertices to specify implementation alternatives for parts of the task), it immediately follows that feasibility analysis for HPC-DAG tasks is at least PSPACE hard; it can be shown with little additional effort that it in fact remains PSPACE complete. In the remainder of this paper we will establish that for the restricted execution model introduced above, which requires that the outcomes of evaluating all the conditional constructs be known prior to executing each dag-job, feasibility analysis falls quite some way down the polynomial hierarchy (Fig. 1), from PSPACE to Π_2^P for conditional DAG tasks with vertices restricted to execute upon specified processors, and to Σ_3^P for HPC-DAG tasks.

4 Feasibility analysis of conditional DAG tasks

In this section we will show that under the restricted execution model where the outcomes of the evaluation of all the conditional expressions in a task must be known prior to executing each instance of the task, the feasibility-analysis problem for conditional DAG tasks with vertices restricted to execute upon specified processors is Π_2^P-complete. [The contents of this section are a minor modification of material first presented in Baruah (2021).] We will explicitly show membership of this problem in Π_2^P, and establish its Π_2^P-hardness by presenting a polynomial-time reduction to this problem from the $\forall\,\exists$ 3SAT problem, which is defined in the following manner:

Definition 1 (The $\forall\,\exists$ **3**SAT Problem)

[1] A contrived and highly simplified example: consider a cyber-physical system designed to operate in wet or dry weather; in darkness or well-lit conditions; under strong winds or in mild ones; etc. While the space of possible operating conditions has cardinality $2 \times 2 \times \cdots \times 2 = 2^n$, it may be reasonable to assume that the conditions along each dimension—whether it is raining or not, the ambient light, the wind-speed, etc.—are all known at the start of each invocation of the task.

INSTANCE. A Boolean formula $\phi(\mathbf{x}, \mathbf{y})$ in 3CNF (Conjunctive Normal Form—i.e., as the conjunct—the 'and'—of clauses each of which comprises exactly 3 literals).
QUESTION. Is is true that $(\forall \mathbf{x})(\exists \mathbf{y})\phi(\mathbf{x}, \mathbf{y})$? □

It is known (Stockmeyer 1976; Wrathall 1976) that the $\forall\exists$ 3SAT problem is complete for complexity class Π_2^P.

Theorem 1 *Under the restricted execution model where the outcomes of the evaluation of all the conditional expressions in a task must be known prior to executing each instance of the task, it is Π_2^P-complete to determine whether a given conditional DAG in which each job is restricted to execute upon a specified processor, is guaranteed to always complete execution by a specified deadline.*

Proof We first show that this problem is in Π_2^P, by showing that the complementary problem: determining whether such a conditional DAG may miss a deadline, is in the class Σ_2^P. Consider a computer equipped with an oracle for determining whether a given "regular" (i.e., not conditional) DAG can be scheduled to completion within a specified deadline—this is known (Jansen 1994) to be an NP-complete problem. Such a computer can "guess", in polynomial time, which branch of each conditional construct is executed during an execution of the conditional DAG that results in a deadline miss, and verify its guess in polynomial time by querying its oracle as to whether the regular DAG resulting from taking exactly these guessed branches is schedulable within the specified deadline.

Having shown above that the problem is in Π_2^P, it remains to establish that it is Π_2^P-hard. We do this by reducing a given $\forall\exists$**3**SAT expression with $(n_x + n_y)$ boolean variables and m 3CNF clauses

$$\forall\left(x_1, x_2, \ldots, x_{n_x}\right)\exists\left(y_1, y_2, \ldots, y_{n_y}\right)\bigwedge_{k=1}^{m}\left(\ell_{k,1} \vee \ell_{k,2} \vee \ell_{k,3}\right) \tag{1}$$

where each $\ell_{k,j}$ is one of the x_i or y_j boolean variables or its negation, to a conditional DAG with

- $(7n_x + 2n_y + 3m)$ nodes, of which $2n_x$ are conditional nodes and the rest represent jobs;
- $(5n_x + 3m)$ edges;
- $(3n_x + n_y + m)$ processors; and
- deadline $D = 4$

that is feasible if and only if the $\forall\exists$**3**SAT expression evaluates to true. The reduction proceeds in the following manner.

For each boolean variable x_i. We define four jobs labeled X_i,$\neg X_i$, A_i, $\neg A_i$ with unit execution requirements, and a single job B_i with execution requirement 3. (We will say that the job X_i *corresponds to* the literal x_i, and the job $\neg X_i$ *corresponds to* the literal $\neg x_i$.)

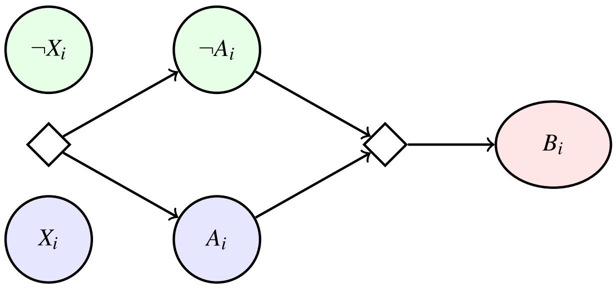

Fig. 6 The jobs constructed for each universally quantified boolean variable x_i

The edges connecting these vertices are as shown in Fig. 6: we have a conditional construct (start-node and associated end-node), with A_i on one branch and $\neg A_i$ on the other, and an edge from the end-node of the conditional construct to the node B_i.

Job B_i is assigned to processor $P_{1,i}$. Jobs X_i and A_i are both assigned to processor $P_{2,i}$. Jobs $\neg X_i$ and $\neg A_i$ are both assigned to processor $P_{3,i}$.

Since the deadline is at time-instant 4 and job B_i has an execution duration of 3, B_i must begin executing no later than time-instant 1 if it is to complete by the deadline. Hence, the conditional construct must complete execution no later than time-instant 1, implying that exactly one of the jobs $\{A_i, \neg A_i\}$ must execute during the time-interval [0, 1]. Since job A_i (job $\neg A_i$, respectively) is assigned to the same processor as job X_i (job $\neg X_i$, resp.), this in turn implies that

Fact 1 *For each $i, 1 \le i \le n_x$, at most one of the jobs $\{X_i, \neg X_i\}$ completes execution by time-instant 1 in any schedule in which the deadline is met.*

For each boolean variable y_j. We define two jobs labeled Y_j and $\neg Y_j$ with unit execution requirements, both assigned to the same processor $P_{4,j}$. Analogously to above, we will say that the job Y_j (job $\neg Y_j$, respectively) *corresponds to* the literal y_j (the literal $\neg y_j$, resp.).

Since both jobs Y_j and $\neg Y_j$ are assigned to the same processor, it follows that

Fact 2 *For each $j, 1 \le j \le n_y$, at most one of the jobs $\{Y_j, \neg Y_j\}$ completes execution by time-instant 1 in any schedule in which the deadline is met.*

Hence by time-instant 1 in any schedule in which the deadline is met, at most one of each pair of jobs $\{X_i, \neg X_i\}$, and at most one of each pair of jobs $\{Y_j, \neg Y_j\}$, could have completed execution. The literals to which the executed jobs correspond can be considered to comprise a truth assignment to the boolean variables $\left(\{x_i, x_2, \ldots, x_{n_x}\} \bigcup \{y_1, y_2, \ldots, y_{n_y}\}\right)$; this leads to the conclusion

Fact 3 *The jobs that have completed execution by time-instant* 1 *in any schedule in which the deadline is met are those corresponding to the literals in some (complete or incomplete) truth-assignment to the boolean variables of Expression* 1.

For each clause $\left(\ell_{k,1} \vee \ell_{k,2} \vee \ell_{k,3}\right)$. We will define three unit-sized jobs $C_{k,1}, C_{k,2}$, and $C_{k,3}$, all of which are assigned to the same processor $P_{5,k}$, and show that at least one of these jobs will be eligible to execute at time-instant 1 if and only if the truth-assignment of Fact 3 above causes the clause $\left(\ell_{k,1} \vee \ell_{k,2} \vee \ell_{k,3}\right)$ to evaluate to true; i.e., at least one of the three literals $\ell_{k,1}, \ell_{k,2}$, or $\ell_{k,3}$, is assigned the truth value $\boldsymbol{T}$ (for "TRUE"). We do so by having a single incoming edge into job $C_{k,1}$ from the job corresponding to the literal $\ell_{k,1}$, a single incoming edge into job $C_{k,2}$ from the job corresponding to the literal $\ell_{k,2}$, and a single incoming edge into job $C_{k,3}$ from the job corresponding to the literal $\ell_{k,3}$.

We point out that some boolean variable not being assigned a truth value (i.e., the truth assignment of Fact 3 not being a complete one) cannot cause some job to become eligible to execute, that would subsequently be rendered ineligible if the truth assignment were completed. That is,

Fact 4 *The number of* $C_{k,\ell}$ *jobs that become eligible to execute at time-instant* 1 *is maximized when the truth assignment of Fact* 3 *is a complete one.*

That concludes our description of the construction of our conditional DAG from Expression 1. We now prove that it can be scheduled to always complete by its deadline of 4 if and only if Expression 1 is valid.

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

Proof Suppose that Expression 1 is valid: for any assignment of truth values to the boolean variables $\left(x_1, x_2, \dots, x_{n_x}\right)$, there is an assignment of truth values to the boolean variables $\left(y_1, y_2, \dots, y_{n_y}\right)$ that causes each of the m clauses of Expression 1 to evaluate to true. We point out that

1. Each assignment of truth values to the boolean variables
$$\left(x_1, x_2, \dots, x_{n_x}\right)$$
can be emulated by executing the appropriate branch of the conditional construct that appears in our conditional DAG. For instance, suppose $x_i \leftarrow \boldsymbol{T}$; the execution of the conditional construct that causes the job $\neg A_i$ to execute would prevent job $\neg X_i$ from executing, but would permit job X_i to execute, by time-instant 1.
2. The assignment of truth values to the boolean variables

$$\left(y_1, y_2, \ldots, y_{n_y}\right)$$

that causes each of the m clauses of Expression 1 to evaluate to true can be emulated by executing the appropriate one of the two jobs that were generated for each y_j. Suppose, for instance, that $y_j \leftarrow \boldsymbol{F}$ in this assignment; this can be emulated by executing the job $\neg Y_j$ by time-instant 1.

3. Hence, each truth-assignment to the boolean variables that cause the m clauses of Expression 1 to evaluate to true can be emulated such that the jobs corresponding to the literals that are true in such a truth-assignment are executed by time-instant 1.
4. Consequently, at least one of the three jobs $C_{k,1}$, $C_{k,2}$, and $C_{k,3}$ corresponding to each clause is eligible to execute by time-instant 1, thereby allowing all three jobs to complete execution on their shared processor by the deadline at time-instant 4.

And this concludes the proof of Lemma 1. □

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

Proof Suppose that the conditional DAG we have constructed can be scheduled to always complete by its deadline.

1. The job B_i that was defined for each variable x_i has execution requirement 3, and so must begin execution no later than time-instant 1 in order to complete by the deadline. Hence the conditional construct defined for the variable x_i must complete execution by time-instant 1.
2. Since each of the n_x conditional constructs are independent of each other, the choice of which branches of each to execute, which in turn restricts which of the pair of jobs $X_i, \neg X_i$ may execute over the interval [0, 1] for each $i, 1 \le i \le n_x$, can force each of the 2^{n_x} possible truth-assignments to the variables $\left(x_1, x_2, \ldots, x_{n_x}\right)$ to be emulated by time-instant 1.
3. For each of these truth-assignments, it must be the case that at least one of the three jobs $C_{k,1}$, $C_{k,2}$, and $C_{k,3}$ corresponding to the k'th clause is eligible to execute by time-instant 1 (in order that all three of these jobs may complete on their common processor by time-instant 4), for each $k, 1 \le k \le m$.
4. Hence it must be possible to execute exactly one of the two jobs $Y_j, \neg Y_j$ upon their shared processor during the time-interval [0, 1] for each $j, 1 \le j \le n_y$, such that the assignment of truth values to the boolean variables $\left(y_1, y_2, \ldots, y_{n_y}\right)$, when combined with the choice of assignment to the boolean variables $\left(x_1, x_2, \ldots, x_{n_x}\right)$ implied by the conditional branches that were executed, causes each of the clauses to be satisfied.

This establishes that Expression 1 is true, and concludes the proof of Lemma 2. □

Taken together, Lemmas 1 and 2 lead us to conclude that under the restricted execution model where the outcomes of the evaluation of all the conditional expressions in a task must be known prior to executing each instance of the task, determining whether a conditional DAG with vertices restricted to execute upon specified processors can be scheduled to always meet its deadline is Π_2^P-hard. Having already shown that this problem in in Π_2^P, we have thus established the truth of Theorem 1.

□

5 Feasibility analysis of HPC-DAG tasks

We will now extend the results of Sect. 4, which were for the conditional DAG task model, to the more general HPC-DAG task model. In particular, we extend the proof of Theorem 1 above to show that under the restricted execution model where the outcomes of the evaluation of all the conditional expressions in a task must be known prior to executing each instance (dag-job) of the task, the feasibility-analysis problem for HPC-DAG tasks is Σ_3^P-complete. That is, the increased expressiveness of the HPC-DAG model over the conditional DAG model (by providing the added option of using alternative vertices to specify implementation alternatives for parts of the task) results in the computational complexity of feasibility analysis being one rung higher up the polynomial hierarchy of Fig. 1.

We will use the following problem analogously to the manner in which the $\forall\exists$ 3SAT problem (Definition 1) was used in Sect. 4:

Definition 2 (The $\exists\forall\exists$ **3**SAT Problem)

INSTANCE. A Boolean formula $\phi(\mathbf{z},\mathbf{x},\mathbf{y})$ in 3CNF (Conjunctive Normal Form—i.e., as the "and" of clauses each comprising exactly 3 literals)

QUESTION. Is is true that $(\exists\mathbf{z})(\forall\mathbf{x})(\exists\mathbf{y})\phi(\mathbf{z},\mathbf{x},\mathbf{y})$? □

It is known (Stockmeyer 1976; Wrathall 1976) that the $\exists\forall\exists$ **3**SAT problem is complete for complexity class Σ_3^P.

Theorem 2 *Under the restricted execution model where the outcomes of the evaluation of all the conditional expressions in a task must be known prior to executing each instance of the task, it is Σ_3^P-complete to determine whether a given HPC-DAG task is guaranteed to always complete execution by a specified deadline.*

Proof Sketch We first show that this problem is in Σ_3^P. Consider a computer equipped with an oracle for determining whether a given conditional DAG in which each job is restricted to execute upon a specified processor can be scheduled to completion within a specified deadline. Such a computer can "guess", in polynomial time, which implementation alternative from amongst the ones made available by each matched pair of alternative vertices should be chosen, and verify its guess in polynomial time by querying its oracle as to whether the resulting concrete task (which is a conditional DAG) is schedulable within the specified deadline. But as proved in Theorem 1 above, this oracle is solving a Π_2^P-complete problem; this serves to establish that under the restricted execution model where the outcomes of the evaluation of all

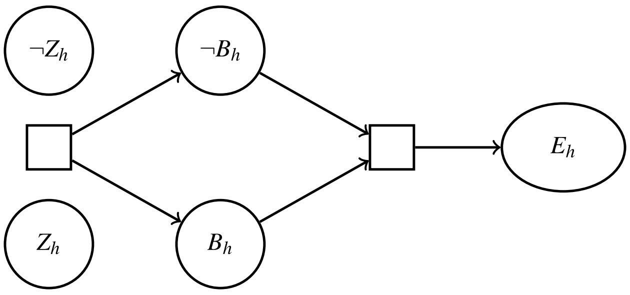

Fig. 7 The jobs constructed for (existentially quantified) boolean variable z_h

the conditional expressions in a task must be known prior to executing each instance of the task, feasibility analysis for HPC-DAG tasks is in Σ_3^P.

Next, we briefly outline how the Π_2^P-hardness reduction in the proof of Theorem 1 above may be extended to show that feasibility analysis for HPC-DAG tasks is Σ_3^P-hard. Analogously to that hardness proof, we will reduce an instance

$$\exists\left(z_1, z_2, \ldots, z_{n_z}\right)\forall\left(x_1, x_2, \ldots, x_{n_x}\right)\exists\left(y_1, y_2, \ldots, y_{n_y}\right)\bigwedge_{k=1}^{m}\left(\ell_{k,1} \vee \ell_{k,2} \vee \ell_{k,3}\right) \quad (2)$$

(where each $\ell_{k,j}$ is one of the x_i, y_i, or z_i boolean variables or its negation) of the $\exists\,\forall\,\exists\,$**3**SAT problem to an HPC-DAG task with deadline 4 that is feasible if and only if the $\exists\,\forall\,\exists\,$**3**SAT instance of Expression 2 evaluates to true. We do so in the following manner:

- The boolean variables $\left(x_1, x_2, \ldots, x_{n_x}\right)$ and $\left(y_1, y_2, \ldots, y_{n_y}\right)$ are dealt with in exactly the same manner as in the Π_2^P-hardness reduction in the proof of Theorem 1. (That is, exactly the same jobs, the same processors and processor assignments, and the same inter-job precedence constraints, are defined as in the proof of Theorem 1.)
- For each $z_h \in \left(z_1, z_2, \ldots, z_{n_z}\right)$, four additional jobs labeled B_h, $\neg B_h$, Z_h and $\neg Z_h$ each with unit execution requirement, and a single job E_h with execution requirement 3, are defined. (As before, we will say that the job Z_h *corresponds to* the literal z_h, and the job $\neg Z_h$ *corresponds to* the literal $\neg z_h$.) Jobs Z_h and B_h are assigned to a (new) processor $P_{6,h}$; jobs $\neg Z_h$ and $\neg B_h$ are assigned to another new processor $P_{7,h}$; and job E_h is assigned to yet another new processor $P_{8,h}$. The edges connecting these vertices are as shown in Fig. 7: a matching pair of alternative vertices (the square vertices in the figure) specify that exactly one of the jobs $\{B_h, \neg B_h\}$ needs to be selected for implementation, and the job E_h must be executed after the selected job.
- Recall the manner in which we had dealt with <u>clauses</u> in the proof of Theorem 1: for each clause $\left(\ell_{k,1} \vee \ell_{k,2} \vee \ell_{k,3}\right)$ we define three unit-sized jobs $C_{k,1}, C_{k,2}$, and

$C_{k,3}$, all assigned to the same processor $P_{5,k}$, and have an incoming edge into job $C_{k,1}$ from the job corresponding to the literal $\ell_{k,1}$, an incoming edge into job $C_{k,2}$ from the job corresponding to the literal $\ell_{k,2}$, and an incoming edge into job $C_{k,3}$ from the job corresponding to the literal $\ell_{k,3}$. This construction is repeated here in the current proof as well, the only additional detail being that since the literals in the clause may now include the variables $\left(z_1, z_2, \ldots, z_{n_z}\right)$ or their negations, the jobs corresponding to the literals may now include the Z_h and $\neg Z_h$ jobs constructed above.

According to the semantics of HPC-DAG tasks, one of $\{B_h, \neg B_h\}$ is selected prior to run-time; during run-time, this job must execute over time-slot [0, 1] in order that the deadline of 4 be met (since the selected job's execution must be followed by the execution of job E_h, which has WCET 3). Since jobs B_h and Z_h share a processor as do jobs $\neg B_h$ and $\neg Z_h$, if B_h ($\neg B_h$, respectively) is the selected job, then only job $\neg Z_h$ (only job Z_h, resp.) can complete by time-instant 1. Hence, *each assignment of truth values to the boolean variables* $\left(z_1, z_2, \ldots, z_{n_z}\right)$ *by time-instant* 1 *is equivalent to the selection of the appropriate alternatives* from amongst each pair $\{B_1, \neg B_1\}$, $\{B_2, \neg B_2\}$, ..., $\{B_{n_z}, \neg B_{n_z}\}$. With this observation is mind, it may readily be verified that the proofs of Lemmas 1 and 2 continue to hold (with Expression 1 replaced by Expression 2 in the statement of both lemmas); this completes the proof of Theorem 2. $\square$

6 Context and conclusions

The HPC-DAG task model (Houssam-Eddine et al. 2020) is a very expressive model that was custom-designed with the goals of facilitating the exploration of the wide space of design choices that become available when complex real-time applications are implemented upon sophisticated modern computing platforms, and enabling the tuning of system scheduling parameters in order to obtain feasible designs. It achieves these goals by enhancing the sporadic DAG model (Baruah et al. 2012) with several features: heterogeneity in the computing platform (via the specification and use of *tags* that identify and distinguish amongst different processors); conditional execution of code during run-time (by incorporating the previously-proposed C-DAG task model (Baruah et al. 2015; Melani et al. 2015)); and the introduction of alternative vertices to represent pre-run-time design choices. These innovative features do indeed result in a very powerful and expressive model; however, the cost of all this expressiveness is intractability—it follows from recent results (Baruah and Marchetti-Spaccamela 2021) that feasibility analysis for HPC-DAG tasks is PSPACE hard.

In this work, we proposed a restricted execution model for HPC-DAG tasks (Sect. 3.2): one which requires that the outcomes of the evaluations of all conditional expressions be known prior to the execution of each instance (dag-job) of a task. While this execution model is no longer able to represent the conditional execution of jobs based upon the evaluation of expressions at arbitrary points in time during run-time, it nevertheless has some expressive value in that it provides a compact representation of tasks modeling systems that are designed to operate under a variety of different “modes,” provided mode-changes are restricted to only being applicable to following dag-jobs (rather than the currently-executing one). Under such a restricted execution model, we showed that the feasibility analysis problem for C-DAG tasks is at the second level of the polynomial hierarchy (it is Π_2^P complete), while that for HPC-DAGs is at the third level of the polynomial hierarchy (Σ_3^P complete).

References

Arora S, Barak B (2009) Computational complexity—a modern approach. Cambridge University Press, Cambridge

Baruah S (2021) Feasibility analysis of conditional DAG tasks is co-NP$^{\mathrm{NP}}$-hard (why this matters). In: Proceedings of the twenty-ninth international conference on real-time and network systems, RTNS ’21, New York, NY, USA. ACM

Baruah S, Marchetti-Spaccamela A (2021) Feasibility analysis of conditional DAG tasks. In: Brandenburg BB (ed) 33rd Euromicro conference on real-time systems (ECRTS 2021), vol 196. Leibniz international proceedings in informatics (LIPIcs), Dagstuhl, Germany. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, pp 12:1–12:17

Baruah S, Bonifaci V, Marchetti-Spaccamela A, Stougie L, Wiese A (2012) A generalized parallel task model for recurrent real-time processes. In: Proceedings of the IEEE real-time systems symposium, RTSS 2012, San Juan, Puerto Rico, pp 63–72

Baruah S, Bertogna M, Buttazzo G (2015) Multiprocessor scheduling for real-time systems. Springer Publishing Company Incorporated, New York

Baruah S, Bonifaci V, Marchetti-Spaccamela A (2015) 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, Lund (Sweden). IEEE Computer Society Press, pp 222–231

Fonseca J, Nelis V, Raravi G, Pinho LM. 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

Houssam-Eddine Z, Capodieci N, Cavicchioli R, Bertogna M, Lipari G (2020) The HPC-DAG task model for heterogeneous real-time systems. IEEE Trans Comput 70:1747–1761

Klaus J (1994) Analysis of scheduling problems with typed task systems. Discret Appl Math 52(3):223–232

Melani A, Bertogna M, Bonifaci V, Marchetti-Spaccamela A, Buttazzo G (2015) Response-time analysis of conditional DAG tasks in multiprocessor systems. In: Proceedings of the 2014 26th Euromicro conference on real-time systems, ECRTS ’15, Lund (Sweden). IEEE Computer Society Press, pp 222–231

Papadimitriou CH (1994) Computational complexity. Addison-Wesley, Reading, MA

Stockmeyer L (1976) The polynomial-time hierarchy. Theoret Comput Sci 3:1–22

Wrathall C (1976) Complete sets and the polynomial-time hierarchy. Theoret Comput Sci 3:23–33

Sanjoy Baruah is the Hugo F. & Ina Champ Urbauer Professor of Computer Science & Engineering at Washington University in St. Louis. His research interests and activities are in real-time and safety-critical system design, scheduling theory, and resource allocation and sharing in distributed computing environments.