On the Complexity of Conditional DAG Scheduling in Multiprocessor Systems

Alberto Marchetti-Spaccamela*, Nicole Megow†, Jens Schlöter†, Martin Skutella‡, Leen Stougie§
*Sapienza University of Rome, alberto.marchetti@dis.uniroma1.it, †University of Bremen, {nmegow,jschloet}@uni-bremen.de,
‡Technical University of Berlin, martin.skutella@tu-berlin.de, §CWI Amsterdam, leen.stougie@cwi.nl

***Abstract*—As parallel processing became ubiquitous in modern computing systems, parallel task models have been proposed to describe the structure of parallel applications. The workflow scheduling problem has been studied extensively over past years, focusing on multiprocessor systems and distributed environments (e.g. grids, clusters). In workflow scheduling, applications are modeled as directed acyclic graphs (DAGs). DAGs have also been introduced in the real-time scheduling community to model the execution of multi-threaded programs on a multi-core architecture. The DAG model assumes, in most cases, a fixed DAG structure capturing only straight-line code. Only recently, more general models have been proposed. In particular, the conditional DAG model allows the presence of control structures such as conditional (if-then-else) constructs. While first algorithmic results have been presented for the conditional DAG model, the complexity of schedulability analysis remains wide open.**

We perform a thorough analysis on the worst-case makespan (latest completion time) of a conditional DAG task under list scheduling (a.k.a. fixed-priority scheduling). We show several hardness results concerning the complexity of the optimization problem on multiple processors, even if the conditional DAG has a well-nested structure. For general conditional DAG tasks, the problem is intractable even on a single processor. Complementing these negative results, we show that certain practice-relevant DAG structures are very well tractable.

***Index Terms*—parallel processing, makespan, conditional DAG, complexity**

I. Introduction

As parallel processing became ubiquitous in modern computing systems, parallel task models have been proposed to describe the structure of parallel applications.

A popular representation is the DAG (directed acyclic graph) model; in this model a task is represented by a DAG $G = (V, E)$ 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 sub-task (or job), and it is characterized by an execution time. 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.

The DAG model has been extensively used to represent cooperative tasks (workflows) which typically require more computing power beyond single machine capability: scientific workflows, multi-tier web service workflows, and big data processing workflows such as Map Reduce from Google and Dryad from Microsoft. A lot of research effort has been done to include in the model specific aspects of the computer platform (multiprocessor systems or distributed environments like clusters and grids), and on other specific aspects of the considered workflow (e.g. resource provisioning and mapping, communications costs etc.). We refer to the survey papers [1], [2] and references therein for a thorough presentation.

The DAG model has also been used in the real-time systems community to model the execution of multi-threaded recurrent tasks to be executed on a multi-core architecture [3], [4].

It is well known that many variants of DAG scheduling are NP-complete even in simple cases. In fact, Ullman [5] showed that it is NP-complete to decide whether the makespan (latest completion time among all jobs) for scheduling a DAG is within a certain deadline, even if *i)* all vertices have unitary execution times using an arbitrary number of processors, and *ii)* all vertices have execution time equal to one or two using only two processors.

A scheduling paradigm which is widely used in practice is called *list scheduling*. The basic idea is to assign priorities to jobs and obtain a list of jobs by sorting them according to their priorities; during execution whenever a processor is idle the available job with highest priority is selected for processing. Graham [6] proved the following performance guarantee: for any priority order of jobs, list scheduling produces a schedule on m processors that has a makespan no greater than $(2 - 1/m)$ times the minimum possible makespan. Most run-time scheduling algorithms that are used for scheduling DAGs use some variant of list scheduling.

While the DAG model captures the intra-parallelism of tasks, it does not capture the typical conditional nature of control flow instructions, such as `if-then-else` statements. The presence of such conditional constructs within the code modeled by the task may mean that different activations of the task cause different parts of the code to be executed. The *conditional DAG* model generalizes the DAG model allowing additional *conditional* nodes [7], [8]. We note that a workflow can also have conditional branches as in BPEL [9]. A formal definition follows in Section II.

In this paper we study the problem of computing the worst-case makespan for a conditional DAG under list scheduling with an arbitrary but fixed priority order (a.k.a. FP-scheduling)

Partially funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) Projects ME 3825/1 and 146371743 - TRR 89 Invasive Computing.

from a complexity theoretic view point. This is a fundamental question, e.g., when performing schedulability analysis.

Our results

We show that it is coNP-complete to decide whether FP-scheduling a given conditional DAG task on multiple processors can be done within a given deadline in each of the (possibly exponentially many) realizations of conditional vertices. This is true even if conditions are independent (see Section 2 for a formal definition). The coNP-completeness holds in both, a non-preemptive (run-to-completion) and preemptive migratory setting. For these results we give a general reduction framework based on a non-obvious relation between the problem of computing the *maximum* makespan for a conditional DAG and *minimizing* the maximum completion time (makespan) for a DAG. The framework allows us to derive several refined complexity results for special graph classes by using (known) NP-hardness results for classical makespan minimization.

It is known that the worst-case makespan for FP list scheduling a conditional DAG task with independent constructs on a single processor can be computed in polynomial time [8]. We show that it is crucial for this result that different conditional constructs are independent of each other, meaning that they are nested and there is no dependency between them such as a shared node. In particular, we show in Section III-D that computing the worst-case makespan for FP-scheduling a general conditional DAG task with shared nodes is coNP-complete even on a single processor.

If the conditional DAG (without shared nodes) has in each realization bounded width (precise definitions follow), then we can compute the worst-case makespan for non-preemptive FP-scheduling in pseudo-polynomial time via a dynamic program. We show also that this algorithm can be turned into an efficient algorithm by loosing only marginally in the performance if a certain monotonicity property holds for the job completion times under FP-scheduling. We present a fully polynomial-time approximation scheme (FPTAS) in this case. We prove that it does hold, e.g., for a bounded number of chains. This monotonicity result might be of independent interest to the scheduling community as has been motivated e.g. in [10]. For general DAGs the monotonicity property does not hold as a classical example known as Graham anomaly [6] shows.

Other related work.

None of the (earlier proposed) parallel-task models (fork/join, synchronous parallel, DAG) captures control flow information such as conditional executions. Some alternatives to the conditional DAG model have been considered. Fonseca et al. [11] propose the *multi-DAG* that represents a task as a collection of DAGs, each describing a conditional execution. Whenever a task is executed, exactly the sub-tasks of one of its DAGs need to be processed. The main issue with this model is the possibly exponential number of control flows.

Chakraborty et al. [12], [13] consider a more restricted variant of the conditional DAG model, which models tasks as a two-terminals DAG and for each node exactly one successor needs to be executed. Additionally each edge characterizes a delay for the start time of the successor. The authors provide complexity results and exact and approximative schedulability analysis for preemptive and non-preemptive scheduling.

Erlebach et al. [14] consider makespan minimization for *AND/OR-Networks* that allow constraints to specify that a job can be executed if at least one predecessor has been completed.

Federated scheduling is a scheduling policy that has been proposed for scheduling a set of recurrent tasks modeled by DAGs in a multiprocessor system; each task has a release time and a deadline. In this model, we assign high-demand conditional DAGs (with high density) to a number of processors that are completely dedicated whereas all remaining low-demand tasks are assigned to a pool of shared processors.

Baruah [15] considered federated scheduling for conditional recurring DAG tasks assuming constrained deadlines. Since each high-demand task is executed independent of the other tasks the main challenge is to assign each such task to a minimum number of dedicated processors, such that it can be completed within its deadline for each possible realization. In the case of constrained deadlines our results directly imply complexity bounds on the problem of minimizing the number of processors necessary to schedule high-demand DAG tasks.

Overview. In Section II we define the task model and notation. In Section III we give our hardness results. For conditional DAG tasks with bounded width, we give in Section IV a pseudo-polynomial algorithm. In Section V we show how to turn this algorithm into a fully polynomial approximation scheme (FPTAS) if a certain monotonicity property holds.

II. System Model and Definitions

The Conditional DAG Model

Let τ be a conditional parallel task (*cp-task*) that executes on m identical processors. The cp-task τ is characterized by a conditional directed acyclic graph $G = (V, E, C)$ where V is a set of nodes, $E \subseteq V \times V$ is a set of directed edges (arcs) and $C \subseteq V \times V$ is a set of distinguished node pairs, the *conditional pairs*. Each node $j \in V$ represents a sequential computation unit (sub-task, job) with an individual execution time p_j. Slightly abusing notation we refer to jobs and nodes equivalently. The arcs describe the dependencies between sub-tasks as follows: if $(v_1, v_2) \in E$, then v_2 can only start processing if v_1 has completed. Job v_1 is called a predecessor job of v_2, and job v_2 is called a successor job of v_1. A distinguished pair $(c_1, c_2) \in C$ of nodes is a conditional pair which denotes the beginning and ending of a conditional construct such as *if-then-else* statements. In sub-task c_1, there is a conditional expression being evaluated and, depending on the outcome, exactly one out of many possible subsequent successors must be chosen. In our figures, the conditional nodes are depicted by a square whereas all other nodes are circles; see Figure 1.

Following the definition given in [7], [8] we define a conditional DAG formally as follows.

```
if c^1 then
  if c^2 then
    basic block b_1;
  else
    basic block b_2;
  end if
else
  basic block b_3;
end if
```

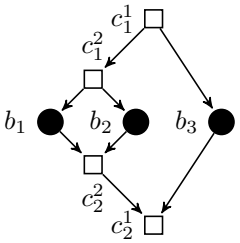

Fig. 1. Example source code and its representation as a conditional DAG. Each b_i represents a sequence of statements and each c^i a boolean expression.

Definition 1. A *conditional DAG* $G = (V, E, C)$ is a DAG (V, E) and a set of *conditional pairs* $C \subseteq V \times V$ such that the following holds for each $(c_1, c_2) \in C$:

1) There are multiple outgoing edges from c_1 in E. Suppose that there are exactly k outgoing edges from c_1 to vertices $s_1, s_2, \ldots, s_k$ for some $k > 1$. We call k the *branching factor* of (c_1, c_2). Then there are exactly k incoming edges into c_2 in E, from the vertices $t_1, t_2, \ldots, t_k$.
2) For each $l \in \{1, \ldots, k\}$ let P_l be the set of all paths from s_l to t_l in G. We define $G_l = (V_l, E_l)$ as the union of all paths from s_l to t_l, i.e., $V_l = \bigcup_{p \in P_l} V(p)$ and $E_l = \bigcup_{p \in P_l} E(p)$, where $V(p)$ and $E(p)$ denote the sets of vertices and edges on path p. We refer to each G_l with $l \in \{1, \ldots, k\}$ as a *conditional branch* of (c_1, c_2).
3) It must hold that $V_l \cap V_{l'} = \emptyset$ for all l, l' with $l \neq l'$. Additionally, with the exception of (c_1, s_l) and (t_l, c_2) there should be no edges in E into vertices in V_l from nodes not in V_l or vice versa for each $l \in \{1, 2, \ldots, k\}$. That is, $E \cap ((V \setminus V_l) \times V_l) = \{(c_1, s_l)\}$ and $E \cap (V_l \times (V \setminus V_l)) = \{(t_l, c_2)\}$ must hold for all l.

For each pair $(c_1, c_2) \in C$ we call c_1 and c_2 *conditional vertices* and refer to the subgraph of G beginning at c_1 and ending at c_2 as *conditional construct* in G. Notice that in the above definition, 3) explicitly rules out any interaction between a node within a conditional branch and any other node outside this particular branch. The restriction to such well-nested structures is very natural when modeling the execution flow of a structured programming language [8]. We refer to a *conditional DAG with shared nodes* when relaxing restriction 3) and allowing interaction between different conditional branches. We consider such a generalized model only in Subsection III-D for demonstrating a drastic increase in complexity.

When executing a conditional DAG $G = (V, E, C)$ at most one conditional branch per conditional pair is executed. For a $c = (c_1, c_2) \in C$ no branch is executed if and only if the conditional construct of c is nested into a branch that is not executed. Thus, a job j is executed if one of the following conditions holds:

(i) node j is not part of any conditional branch, i.e, $j \notin V_l$ for each branch G_l of any conditional pair (c_1, c_2), or
(ii) the innermost branch G_l with $j \in V_l$ is being executed.

Let $J \subseteq V$ be a set of jobs obtained by *fully executing* the jobs of the conditional DAG $G = (V, E, C)$ taking into account the outcome of the conditional nodes. Let $G_J = (V_J, E_J)$ with $V_J = J$ denote the subgraph of G induced by J. We call G_J a *realization* of G, and say a vertex $j \in V$ is *active* for J, if $j \in V_J$ holds. Let $\mathcal{J}$ denote the collection of all job sets J for which there is a realization with $V_J = J$.

Fixed-Priority List Scheduling

Let τ be a conditional DAG task with $G = (V, E, C)$ and with execution times p_j, for each $j \in V$, to be executed on m parallel identical processors. Let $\prec$ be a given *fixed-priority order (FP-order)* over V.

A *non-preemptive* fixed-priority list schedule (FP-schedule) is constructed as follows. At any point in time, when a processor is idle, the job with the highest priority according to $\prec$ among the available jobs starts execution and runs until completion. A job is *available* if all its predecessors have been completed. To avoid ambiguities for jobs j with $p_j = 0$, we say the successors of such jobs are available if all predecessors j with $p_j = 0$ have been started and all predecessors j with $p_j > 0$ have been completed.

If we allow *preemption* (and migration), then at the arrival of a job of a higher priority, any executing lower-priority job is preempted. A preempted job may resume processing at any later point in time and on any processor at no extra cost. We assume that any overhead is covered in p_j.

For each $J \in \mathcal{J}$ let S_J denote the FP-schedule induced by $\prec$ for the realization G_J. Let C_J denote the latest completion time of any job in G_J in S_J. This is the *makespan* for realization G_J of the cp-task τ. We may assume that there is just a single cp-task in our non-periodic task setting since several cp-tasks can be merged into one by adding nodes with zero execution times.

Then, $M(G, \prec) = \max_{J \in \mathcal{J}} C_J$ is the *worst-case makespan* of τ for list scheduling according to the FP-order $\prec$.

Definition 2 (Problem CDAG-MAX)**.** Given a cp-task with a conditional DAG G, execution times p_j, a number m of parallel identical processors and an FP-order $\prec$, the *worst-case makespan problem* (CDAG-MAX) is to compute $M(G, \prec)$.

Slightly abusing notation, we use CDAG-MAX also to refer to the following *decision variant of* this problem in the complexity analysis: for a given CDAG-MAX instance and a parameter D decide whether $M(G, \prec) \leq D$.

We observe that $M(G, \prec)$ can be approximated within a factor 2 in polynomial time. To see that, consider the well-known Graham bounds [6] on the makespan for any FP-schedule (mentioned also in [8], [15]):

$$M(G, \prec) \geq L_{\max} \quad \text{and} \quad M(G, \prec) \geq V_{\max}/m,$$

where $L_{\max}$ denotes the length of the longest chain in the conditional DAG and $V_{\max}$ is the maximum total volume of execution time that has to be executed in a realization of the cp-task. Both lower bounds can be computed in polynomial time; how to compute $V_{\max}$ is shown in [8], [15]. Further, it holds that

$$M(G, \prec) \leq L_{\max} + V_{\max}/m.$$

Lemma 1. CDAG-MAX *can be approximated within a factor* 2 *in polynomial time, i.e., we can efficiently compute the value* $\text{apx} = L_{\max} + V_{\max}/m$ *that satisfies*

$$M(G, \prec) \leq \text{apx} \leq 2M(G, \prec).$$

III. Complexity

In this section we show several NP-hardness and inapproximability results regarding non-preemptive and preemptive FP-scheduling of conditional DAG tasks. Firstly, we establish a reduction framework via a makespan maximization problem and prove that for non-preemptive FP-scheduling CDAG-MAX is strongly coNP-hard and that approximating CDAG-MAX within a factor of $\frac{7}{5}$ is NP-hard. The framework can also be used to derive further hardness results for various special graph classes. We furthermore investigate the problem for preemptive FP-scheduling and show that deciding CDAG-MAX remains strongly coNP-hard and approximating CDAG-MAX within a factor of $\frac{6}{5}$ is still NP-hard. Finally, we consider the conditional DAG model with shared nodes and show that CDAG-MAX is much harder in this case: it is coNP-hard already on a single processor. This is in contrast to CDAG-MAX for the conditional DAG model as studied in this paper, which is solvable in polynomial time on a single machine [8].

As we can show that the complement of CDAG-MAX is in NP by using realizations G_J with $C_J > D$ for an input parameter D as certificates, all our coNP-hardness results imply the coNP-completeness of the corresponding problems.

A. A Reduction Framework

We first introduce an approximation preserving polynomial time reduction from an auxiliary problem, the *list scheduling makespan maximization problem* (LS-MAX). This reduction then gives us a framework to deduce hardness and inapproximability results for CDAG-MAX; by showing the NP-hardness of solving or approximating LS-MAX, we prove corresponding hardness results for CDAG-MAX by exploiting the reduction.

Definition 3 (Problem LS-MAX)**.** We are given a precedence constraint DAG $G = (V, E)$, jobs with execution times p_j for each $j \in V$, m identical parallel processors and a deadline D. The task is to decide whether $\overline{C}_{\max} > D$, where $\overline{C}_{\max}$ is the maximum makespan that can be achieved by any list scheduling order (i.e., any FP-order).

Theorem 1. *There is an approximation preserving polynomial time reduction from* LS-MAX *to* CDAG-MAX.

Proof. Consider an LS-MAX instance with DAG $G = (V, E)$, jobs $V = \{1, \ldots, n\}$, execution times p_j, for $j \in V$, and m processors. We construct an instance of CDAG-MAX on $m' = m$ processors with a conditional DAG $G' = (V', E', C')$, execution times p'_j and an FP-order $\prec$ as follows:

1) For each job $j \in V$ add
   a) n job copies $v_j^1, \ldots, v_j^n$, each with execution time p_j,
   b) a conditional pair (c_1^j, c_2^j) with execution times zero that uses each v_j^l as a conditional branch; see Fig. 2*a*.
2) For each $(i, j) \in E$ introduce an edge from c_2^i to c_1^j.
3) Fix the priority order $\prec$ such that
   a) $k < k'$ implies $v_j^k \prec v_i^{k'}$ for each $i, j \in V$.
   b) $j \prec j'$ holds for all $j, j' \in V'$ with $p'_{j'} > 0 = p'_j$.

See Figure 2*b* for an illustration of the construction which can be done in polynomial time. Let $\overline{C}_{\max}$ be the maximum makespan of the given LS-MAX instance. We now show that $\overline{C}_{\max} = M(G', \prec)$.

To do so, consider an arbitrary realization G'_J. Figure 2*c* illustrates a realization for the example in Fig. 2*b*. Observe that in G'_J exactly one job copy v_j^l is active for each job $j \in V$. Let v_j^l and $v_{j'}^{l'}$ be active job copies in G'_J, then, by construction, there is a path $P = (v_j^l, c_2^j, c_1^{j'}, v_{j'}^{l'})$ in G'_J if and only if $(j, j') \in E$. Note that all vertices on P, apart from the endpoints, have an execution time of zero and that, according to the given order $\prec$, all vertices with execution time zero are processed before all other vertices. Therefore, the only function of P when scheduling G'_J is, that it formulates a precedence constraint between v_j^l and $v_{j'}^{l'}$. Thus, all precedence constraints between the original jobs in G are also present between the corresponding active job copies in G'_J.

In addition to the active job copies and the paths that connect them, G'_J only contains a unique predecessor respectively successor with an execution time of zero for job copies v_j^l such that j has no predecessor respectively successor in G. As those jobs have an execution time of zero, they do not affect the schedule of G'_J given $\prec$.

Furthermore, each job copy v_j^l has the same execution time as the original job j, and G as well as G'_J are scheduled on the same number of processors.

By definition of CDAG-MAX, the active job copies in any realization G'_J are scheduled using FP-scheduling with order $\prec$ to achieve a makespan of C_J. In LS-MAX, any list scheduling order can be chosen. In specific, for each realization G'_J there is an order $\prec_{LS}$ that orders the jobs in G the same way as $\prec$ schedules the active job copies in G'_J. I.e., there is an order $\prec_{LS}$, such that $v_j^l \prec v_{j'}^{l'}$ holds for the active job copies v_j^l and $v_{j'}^{l'}$ of original jobs $j, j' \in V$ if and only if $j \prec_{LS} j'$. Thus, $\prec_{LS}$ achieves a makespan of C_J. As this holds for any realization, $M(G', \prec) \leq \overline{C}_{\max}$ follows.

Consider an arbitrary list scheduling order $\prec_{LS}$ on the jobs in G that achieves a makespan of C. We show that there is a realization G'_J such that $\prec$ orders the active job copies in G'_J exactly as $\prec_{LS}$ orders the original jobs in G. For each job $j \in V$ let q_j denote the position of j in $\prec_{LS}$, i.e., j has the q_j highest priority in V. Then, there is a realization G'_J such that $v_j^{q_j}$ is the sole active job copy of j in G'_J for each $j \in V$. Let j and j' with $j \neq j'$ be two arbitrary jobs in V with $j \prec_{LS} j'$, then $q_j < q_{j'}$ holds by definition and $v_j^{q_j}$ and $v_{j'}^{q_{j'}}$ are the active job copies in G'_J. By construction of $\prec$, $q_j < q_{j'}$ implies $v_j^{q_j} \prec v_{j'}^{q_{j'}}$. Thus, $\prec$ orders the active job copies in G'_J exactly as $\prec_{LS}$ orders the original jobs in G and achieves a makespan of C. As this holds for each list scheduling order $\prec_{LS}$, $\overline{C}_{\max} \leq M(G', \prec)$ and thus $\overline{C}_{\max} = M(G', \prec)$ follows. □

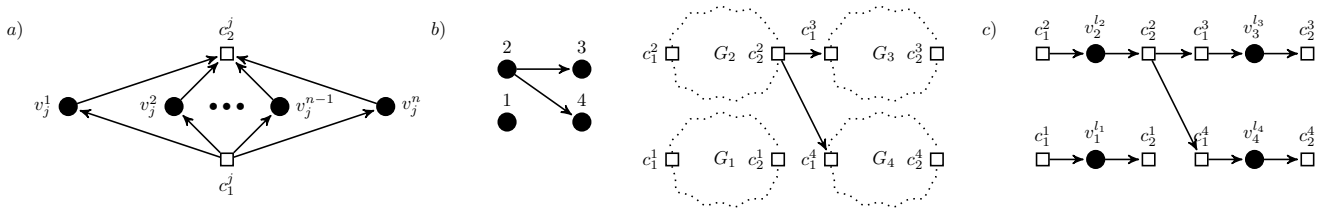

Fig. 2. Examples for the reduction of Theorem 1: a) Conditional pair as in the reduction, b) Precedence constraint graph of an LS-MAX instance and the constructed conditional DAG where each G_i is an abstraction of a conditional pair as illustrated in a), and c) Realization G'_J of the conditional DAG in b).

Observe that the reduction from LS-MAX to CDAG-MAX in some way preserves the structure of the input precedence constraint graph G. Let $G' = (V', E', C')$ be the constructed conditional DAG. Consider the graph $G_{C'}$ that uses each conditional pair $c_i = (c_1^i, c_2^i) \in C'$ as a vertex and has edges between c_i and c_j if and only if $(c_2^i, c_1^j) \in E'$. We can observe that G and $G_{C'}$ are isomorphic by definition of the reduction. As each realization G'_J of G' contains a simple path instead of each conditional construct, we can observe the following.

Lemma 2. *Let $G = (V, E)$ be the precedence constraint graph of an* LS-MAX *instance I and let $G' = (V', E', C')$ be the conditional DAG that is constructed by the reduction of Theorem 1 given I. Then, the following statements hold.*

1) *If G is a tree, then each realization G'_J of G' is a tree.*
2) *If G is a set of k chains, then each realization G'_J of G is a set of k chains.*

This lemma is useful to show the coNP-hardness of CDAG-MAX for special graph classes. If we show that LS-MAX is NP-hard for precedence constraints graphs that form a tree or are constant number of chains, Lemma 2 and the reduction of Theorem 1 imply the coNP-hardness of the corresponding CDAG-MAX variant.

B. Hardness and Inapproximability Results

We now exploit the previously introduced reduction framework to show hardness and inapproximability results for CDAG-MAX. First, we show the strong NP-hardness of LS-MAX that, in combination with the reduction of Theorem 1, implies the strong coNP-hardness of CDAG-MAX. Then, we consider special cases of LS-MAX and CDAG-MAX and use the reduction framework to derive further hardness results. Finally, we show that it is NP-hard to approximate LS-MAX within a factor of $\frac{7}{5}$, which implies the same result for CDAG-MAX.

Theorem 2. LS-MAX *is strongly NP-hard, even if there are no precedence constraints.*

Proof. We show the NP-hardness of LS-MAX by reducing from a classical load balancing problem, typically denoted by $P||C_{\max}$: given a set of jobs $J = \{1, \ldots, n\}$ with execution times p_j for $j \in J$, and a number of processors m, one asks for an FP-schedule of *minimum* makespan. Deciding if an FP-schedule with a makespan of $\sum_{j=1}^{n} \frac{p_j}{m}$ exists, is known to be strongly NP-hard [16]. This holds true even if $p_j \leq \sum_{j=1}^{n} \frac{p_j}{m}$ for all $j \in J$.

Given an instance I of $P||C_{\max}$with $p_j \leq \sum_{j=1}^{n} \frac{p_j}{m}$ for all $j \in J$, we construct the following LS-MAX instance I'. Let $J \cup \{n+1\}$ be the set of jobs, $p_{n+1} = \sum_{j=1}^{n} \frac{p_j}{m}$ be the execution time of the new job, and let m be the number of processors. We construct a precedence constraint DAG $G = (V, E)$ with $V = J \cup \{n+1\}$ and $E = \emptyset$.

We show that the maximum makespan of I' is $\overline{C}_{\max}(I') = 2 \cdot \sum_{j=1}^{n} \frac{p_j}{m}$ if and only if the minimum makespan of I is $C_{\max}(I) = \sum_{j=1}^{n} \frac{p_j}{m}$.

By the classical list scheduling arguments of Graham [6], any FP-schedule achieves a makespan on I' of at most $\sum_{j \neq l} \frac{p_j}{m} + p_l$, where p_l is the job that determines the makespan. (This is because job l is placed on the least loaded processor which has a load of at most the average load.) This upper bound is tight if each processor has a load of $\sum_{j \neq l} \frac{p_j}{m}$ in the moment when l is assigned. The term $\sum_{j \neq l} \frac{p_j}{m} + p_l$ is maximal for the constructed instance if $n+1$ is the job that determines the makespan since it is the largest job. Thus, the following term is an upper bound on the maximum possible makespan for the constructed instance I':

$$\sum_{j \neq n+1} \frac{p_j}{m} + p_{n+1} = \sum_{j=1}^{n} \frac{p_j}{m} + p_{n+1} = 2 \cdot \sum_{j=1}^{n} \frac{p_j}{m}.$$

As argued above, this upper bound can be reached iff jobs $1, \ldots, n$ can be scheduled such that each processor has load $\sum_{j=1}^{n} \frac{p_j}{m}$. This is possible iff $C_{\max}(I) = \sum_{j=1}^{n} \frac{p_j}{m}$, which is a lower bound on the makespan of the input instance. Thus, $\overline{C}_{\max}(I') = 2 \cdot \left(\sum_{j=1}^{n} \frac{p_j}{m} \right)$ iff $C_{\max}(I) = \sum_{j=1}^{n} \frac{p_j}{m}$. □

The strong NP-hardness of LS-MAX and the reduction of Theorem 1 imply the following theorem.

Theorem 3. CDAG-MAX *is strongly coNP-complete.*

By similar arguments as in the proof of Theorem 2 and a particular hardness result for makespan minimization in [17], we can give further refined complexity results for particular graphs classes. The full proof is omitted.

Theorem 4. LS-MAX *is (a)* strongly NP-hard *even if the precedence constraint graph is a tree and (b)* weakly NP-hard *even if the precedence constraint graph consists of four chains to be processed on two processors.*

The theorem, the reduction of Theorem 1 and Lemma 2 then directly imply the following hardness results for CDAG-MAX. Part (c) of the theorem can be shown by using the existing reduction, exploiting the hardness of LS-MAX without precedence constraints and adding dummy terminals.

Theorem 5. CDAG-MAX *is (a)* strongly coNP-complete *even if each realization G_J of the conditional DAG G is a tree, (b)* weakly coNP-complete *even if each realization G_J of the conditional DAG G consists of four chains to be processed on two processors and (c)* strongly coNP-complete *even if the conditional DAG is a two-terminals series-parallel graph.*

Finally we give an inapproximability result for LS-MAX and consequently CDAG-MAX. Our reduction from CLIQUE is inspired by [18]; their reduction for makespan minimization with precedence constraints gives a hardness of approximation bound of $4/3$ assuming unit execution times. We obtain a slightly better bound using non-unit execution times.

Theorem 6. *Approximating* LS-MAX *with a ratio better than* $7/5$ *is NP-hard.*

Proof. Given a (undirected) graph $G = (V, E)$ and an integer $k \geq 3$ we ask whether G contains a clique of k vertices. Let $h = k(k-1)/2$ be the number of edges of a k-clique. We construct an LS-MAX instance I on $m = |V| + |E|$ processors and a DAG H with nodes partitioned in six sets A, B, C, V', E', X as follows:

1) $V' = \{v_j \mid j \in V\}$ and $E' = \{e' \mid e \in E\}$.
2) For all $e = \{i, j\} \in E$, add arcs $(v_i, e'), (v_j, e')$ to H.
3) Define $|A| = m-k$, $|B| = |E|+k-h$ and $|C| = |V|+h$.
4) Introduce edges from each vertex in A to each vertex in B and from each vertex in B to each vertex in C.
5) X contains a single node x.
6) Each node represents a job; all jobs in A, B, C, E', V' have unitary execution times; job x requires 4 time units.

To show the inapproximability result, we separately proof the following two claims that imply the theorem:

1) If G has a k-clique, then there exists an FP-schedule of I with a makespan of 7.
2) If there is no k-clique in G, then all FP-schedules of I have a makespan of at most 5.

Proof of Claim 1): Consider the following order of nodes: first nodes in A, then the nodes of V' that correspond to vertices of a k-clique of G, then the remaining nodes of V' followed by nodes in B, E', C and finally $X = \{x\}$. The schedule is as follows:

- time $t = 0$: all nodes of A and the k nodes of V' corresponding to vertices of the k-clique are processed.
- $t = 1$: the still unprocessed nodes in V' and all nodes in B are executed on $(|V|-k)+(|E|+k-h) = m-h$ processors. Therefore, we can additionally process h nodes in E' that are already available; the nodes corresponding to the edges of the k-clique of G.
- $t = 2$: we execute the remaining nodes in E' and all nodes in C using $(|E|-h)+(|V|+h) = m$ processors.
- $t = 3$: we start execution of job x that finishes at time 7.

Proof of Claim 2): We first show that if there is no k-clique in G, node x starts execution at time $t \leq 1$ for any FP-order. In fact, if x does not start execution at $t = 0$, then at time 0 we execute jobs of V' and A. If all jobs in A are executed at $t = 0$, then we can process only k jobs in V'. Since G has no k-clique, it follows that at $t = 1$ we can execute jobs in B and at most $h-1$ jobs in E' whose predecessors have been processed in the previous step. Therefore job x is processed at $t = 1$ independently of its position in the ordering.

If x does not start execution at $t = 0$ and not all jobs in A are executed at time 0, then we observe that at least $m - |V|$ jobs of A have been processed. Therefore at time $t = 1$ there are at most $|V| - k$ unprocessed jobs in A and possibly all jobs in E' can be processed; therefore there are at least $m - (|V| - k) - |E| = k$ free processors and job x is processed independently of its position in the ordering at time $t = 1$. We conclude that job x is completed by time $t = 5$.

We now show that all other jobs are completed by time $t = 5$ for all FP-orders. To prove this, we first show that at $t = 2$ all jobs in A and V' are completed for all FP-orders.

If x and all jobs in A are started at $t = 0$, then at time 1 there are at most $|V| - k + 1$ unscheduled jobs in V' and at most $(k-1)(k-2)/2 = h-k+1$ jobs of E' can be processed. This leaves enough free processors to schedule all remaining jobs in V' and all jobs in B, as $(h-k+1)+(|V|-k+1)+(|E|+k-h) = |V|+|E|-k+2 \leq |V|+|E|-1 = m-1$ holds (using $k \geq 3$) and only one processor is busy processing x. It follows that all jobs in V', A and B are finished at $t = 2$.

If at time $t = 0$ job x is not started while all jobs in A are executed, then there are at most $|V| - k$ unscheduled jobs in V' at time 1. Since G has no k-clique, there are at most $h-1$ jobs in E' that can be processed. Even if all these jobs are processed alongside job x, we are left with $(m-1) - (h-1) = m-h$ free processors to process jobs in B and remaining jobs in V'. These sets consist of a total of at most $(|E|+k-h)+(|V|-k) = m-h$ jobs. Therefore we conclude that all jobs in V', A and B are completed at time 2.

If not all jobs in A are executed at $t = 0$, we observe that at time 0 we execute at least $m-1$ jobs in $(V' \bigcup A)$. Thus, at time 1 there are at most $|V|-k+1$ unprocessed jobs in V' and A. It follows that at time 1 we can execute jobs in V', A and a subset of jobs in E' that has size at most $|E|$. Therefore, using $m-1$ processors, we can schedule $(m-1) - |E| = |V| - 1$ jobs in $(V' \bigcup A)$. We conclude that all jobs in A and V' are completed at time $t = 2$ independently of the job ordering.

We now show that at time 5 all jobs in E', B and C are completed. First, assume that at $t = 3$ all nodes in B are processed. It follows that at $t = 3$ all predecessors of remaining unscheduled jobs in E' and C are completed and that therefore the schedule of jobs in C and E' is completed by time $t = 5$ for any FP-order. If at time $t = 3$ not all jobs in B are processed, we can schedule jobs in E' and B but no jobs in C; it follows that at time $t = 4$ we complete the execution of jobs in E' and B. Therefore at time $t = 5$ we complete the schedule by completing all jobs in C. □

Theorem 6 and the reduction of Theorem 1 imply the following result.

Theorem 7. *Approximating* CDAG-MAX *with a ratio better than* $7/5$ *is NP-hard.*

C. Preemptive Scheduling

Theorem 2 states that LS-MAX is strongly NP-hard even if there are no precedence constraints. We reduce from LS-MAX without precedence constraints to the preemptive variant of CDAG-MAX in which each realization G'_J is scheduled using preemptive FP-scheduling based on $\prec$. We follow the reduction of Theorem 1. In the case without precedence constraints, no edge will be introduced in Step 2) of the reduction. Thus, the constructed conditional DAG G' has no path between any pair of job copies v_j^l and $v_{j'}^{l'}$ with $v_j^l \neq v_{j'}^{l'}$. As there are no paths between job copies and all other jobs have an execution time of zero by definition, it follows that all active job copies are available at time zero in any realization G'_J. Consider the preemptive variant of CDAG-MAX. The FP-schedule of G'_J will never use preemption, because no high priority job will ever become available and interrupt a low priority job. Thus, the preemptive and non-preemptive schedules are equivalent and we conclude the following result.

Theorem 8. CDAG-MAX *is strongly coNP-complete even under preemptive FP-scheduling.*

By using a similar proof to the one of Theorem 6, we can give the following inapproximability result for LS-MAX with unit-size jobs FP-scheduling.

Theorem 9. *Approximating* LS-MAX *with a ratio less than* $6/5$ *is NP-hard even with unit-size jobs.*

The following theorem is implied by Theorem 9 and the reduction of Theorem 1. Note that the reduction introduces conditional nodes with execution times zero. The result for such unit- and zero-size jobs holds also for preemptive CDAG-MAX as preemption will not occur.

Theorem 10. *Approximating* CDAG-MAX *with a ratio better than* $6/5$ *is NP-hard, even (a) when* $p_j = 0$ *for conditional jobs and* $p_j = 1$ *otherwise, and (b) under preemptive FP-scheduling.*

D. Generalized Conditional DAG with Shared Nodes

In this section, we consider a more general variant of conditional DAGs allowing less nested structures and demonstrate a substantial increase in the complexity. While CDAG-MAX for conditional DAGs without shared nodes can be solved in polynomial time on a single processor [8], we show that CDAG-MAX for conditional DAGs with shared nodes is strongly NP-hard even on a single processor.

A *conditional DAG with shared nodes* $G = (V, E, C)$ is defined analogous to Definition 1 with an adjusted third requirement for each $(c_1, c_2) \in C$ that allows edges from conditional branches G_l to conditional branches $G_{l'}$ of pairs $(c'_1, c'_2) \in C$ with $(c_1, c_2) \neq (c'_1, c'_2)$.

When executing a conditional DAG with shared nodes, the execution of branches is defined as before and we say that a job $j \in V$ is executed if one of the following conditions holds:

- j is not part of any conditional branch. That is, $j \notin V_l$ for each branch G_l of any conditional pair (c_1, c_2),
- *at least one* of the innermost conditional branches G_l with $j \in V_l$ is being executed.

Realizations, FP-schedules and makespans are then defined as for conditional DAGs without shared nodes. Observe that the makespan C_J of a realization G_J on a single processor is just the sum of the execution times of all active jobs in G_J. Therefore, the used FP-order has no influence on the makespan and we do not consider it subsequently.

We show a reduction from the strongly NP-hard problem 1in3-SAT [19]. In 1in3-SAT, there is given a set of propositional logic 3-Clauses $\mathfrak{C} = \{\mathfrak{C}_1, \ldots, \mathfrak{C}_n\}$ and a set of variables $L = \{\lambda_1, \ldots, \lambda_n\}$ such that each clause contains only positive literals and each variable occurs in exactly three clauses. The question is whether there is a satisfying variable assignment that satisfies exactly one λ_{ij} for each $\mathfrak{C}_i = \{\lambda_{i1}, \lambda_{i2}, \lambda_{i3}\}$.

Theorem 11. CDAG-MAX *for conditional DAGs with shared nodes is strongly coNP-complete even on a single processor.*

Proof. Let $(\mathfrak{C}, L)$ be a given 1in3-SAT instance. We construct a conditional DAG $G = (V, E, C)$ with execution times p_j for all $j \in V$ and $m = 1$ as follows (see also Figure 3):

1) For each clause $\mathfrak{C}_j \in \mathfrak{C}$, add a node γ_j with an execution time of one.
2) For each $\lambda_i \in L$, add two nodes c_1^i and c_2^i with execution times of zero that form a conditional pair $c_i = (c_1^i, c_2^i)$.
   a) For each branch $l \in \{1, 2\}$ of c_i, add a source s_l^i and sink t_l^i with execution times of zero.
   b) For each node γ_j with $\lambda_i \in \mathfrak{C}_j$, add edges from s_1^i to γ_j and from γ_j to t_1^i.
   c) Add vertices π_i and ρ_i with execution times of one and edges from s_2^i to π_i and ρ_i and from π_i and ρ_i to t_2^i.

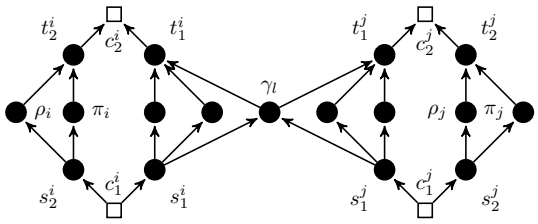

Fig. 3. Reduction of Theorem 11 for variables λ_i, λ_j that share a clause $\mathfrak{C}_l$.

Obviously the reduction can be done in polynomial time. To prove correctness and completeness, we show the following two statements:

1) If the given 1in3-SAT instance has a feasible solution, $M(G, \prec) \geq \frac{7n}{3}$ holds.
2) If the given 1in3-SAT instance does not have a feasible solution, $M(G, \prec) < \frac{7n}{3}$ holds.

For each variable assignment $\alpha : L \to \{0, 1\}$ we can construct a realization G_J such that for each $c_i = (c_1^i, c_2^i) \in C$

holds that $G_1^i \subseteq G_J$ if $\alpha(\lambda_i) = 1$ and $G_2^i \subseteq G_J$ if $\alpha(\lambda_i) = 0$, where G_1^i and G_2^i are the two conditional branches of c_i. In this way, a unique realization is constructed for each different α. As additionally the number of possible variable assignments and realizations is equal at 2^n, we can conclude that there is an one-to-one correspondence between variable assignments and realizations of the constructed conditional DAG.

To prove the first statement, assume that a 1in3-SAT-instance with a feasible solution is given. Then, a satisfying variable assignment $\alpha : L \to \{0, 1\}$ exists such that α satisfies each clause by exactly one literal. Let G_J be the realization constructed for α as described above. We show that $C_J \geq \frac{7n}{3}$ holds for realization G_J. Then, $M(G, \prec) \geq \frac{7n}{3}$ follows.

For each $\lambda_i \in L$ the branch G_1^i is part of G_J if and only if $\alpha(\lambda_i) = 1$ by definition of G_J. In the resulting schedule, each γ_j must be executed as α satisfies all clauses and thus for each clause $\mathfrak{C}_j$ there is a literal $\lambda_i \in \mathfrak{C}_j$ with $\alpha(\lambda_i) = 1$ and thus $G_1^i \subseteq G_J$. Therefore $\gamma_j \in V_1^i$ is active and executed in G_J. This contributes n time units to the makespan C_J.

Because α satisfies each clause by exactly one literal and each variable occurs only as a positive literal in exactly three clauses, it follows by pigeon hole principle that at least $\frac{2n}{3}$ variables $\lambda_i \in L$ exist with $\alpha(\lambda_i) = 0$ and thus $G_2^i \subseteq G_J$. Therefore, the vertices π_i and ρ_i are active for each such λ_i. This contributes $\frac{4n}{3}$ time units to C_J. Adding up both parts leads to an makespan of at least $\frac{4n}{3} + n = \frac{7n}{3}$.

To prove the second statement, assume that the formula is not satisfiable and consider an arbitrary variable assignment α. Let k be the number of variables λ_i with $\alpha(\lambda_i) = 0$ and let G_J be the realization corresponding to α as described above.

If $k < \frac{2n}{3}$, then at most n nodes γ_j are active for G_J. For k variables λ_i, it holds that π_i and ρ_i are active for G_J. Therefore the makespan of G_J is at most $n + 2k < n + \frac{4n}{3} = \frac{7n}{3}$.

If $k > \frac{2n}{3}$, then at most $3 \cdot (n - k)$ nodes γ_j are active for G_J as each variable occurs in at most 3 clauses. Additionally, for k variables λ_i it again holds that π_i and ρ_i are active for G_J. Therefore the makespan of G_J is at most $2k+3(n-k) < 2\frac{2n}{3} + 3(n - \frac{2n}{3}) = \frac{7n}{3}$.

To finish the proof, consider $k = \frac{2n}{3}$. As α does not satisfy the given formula by assumption, it follows that of the $n - k$ positive assigned variables at least two must occur in the same clause. This means that at least one node γ_j is not active for G_J. Therefore the makespan G_J is strictly less than $n+2k = n + \frac{4n}{3} = \frac{7n}{3}$.

It follows that $C_J < \frac{7n}{3}$ holds for each realization G_J that corresponds to an assignment α. Because each realization corresponds to an assignment α, $C_J < \frac{7n}{3}$ holds for each realization G_J and therefore $M(G, \prec) < \frac{7n}{3}$ follows. □

IV. Pseudo-polynomial Time Algorithm for Bounded Width

In this section, we consider conditional DAGs G with the property that each realization G_J represents a partial order of width bounded by a constant k. The *width* of a partial order is the maximum number of pairwise incomparable tasks, that is, the maximum antichain. Slightly abusing notation, we say that the underlying graph has width bounded by k.

Theorem 12. CDAG-MAX *can be solved exactly in pseudo-polynomial time if each realization G_J of the given conditional DAG G has width at most k.*

Assume w.l.o.g. that G has a single source; otherwise we simply add a dummy terminal with execution time zero.

We present a dynamic program (DP) for solving CDAG-MAX. Each state of the DP describes a partial schedule in terms of an *ideal* as defined in [20]. An ideal I of a realization G_J is a subset of V_J such that a job in I implies all of its predecessors to be contained in I as well. We say that I is an ideal of G if I is an ideal of some realization G_J. Every partial schedule for a set of jobs in G_J must contain all jobs in the corresponding ideal of G to ensure that precedence constraints and conditions are respected. Our DP establishes the reachability in a graph of ideals, where an ideal $\bar{I}$ is reachable from I if a feasible subschedule for I can be extended to a feasible schedule for $\bar{I}$ by adding tasks in $\bar{I} \setminus I$ while respecting the FP-order.

An ideal I can be represented in terms of its *front tasks* $I' \subseteq I$, which are all jobs $j \in I$ without successors in I. According to *Dilworths Decomposition Theorem* [21] an ideal I of a graph G with a width bounded by k can have at most k front tasks. Thus, the number of different ideals is bounded by n^k. A *state* of our DP is a tuple (I, P) with

- $I \subseteq V$ is the set of front tasks of an ideal for some realization G_J of G such that there is a point in time t in the FP-schedule S_J where all jobs in I are either being processed or available to being processed.
- For each $j \in I$, $P_j \in \mathbb{N} \cup \{-\}$ either denotes the remaining execution time necessary to complete j or indicates that j has not been started yet ($P_j = -$).

We define a weighted, directed and acyclic state graph $H = (U, F, w)$ with one source and one sink such that U contains the states of the DP and F contains all feasible state transitions. We define H such that the length of the longest source-sink-path in H corresponds to the worst-case makespan of G. To construct H, consider the *initial state* $u_0 = (\{s\}, -)$ where s is the source of G. The state u_0 is part of H and we inductively define the rest of H.

Consider a state $u = (I, P) \in U$. We define B as the set of jobs that will be started next according to the FP-order. Let m_u denote the number of jobs that are being processed in state u (jobs $j \in I$ with $P_j \neq -$). Then $m' = m - m_u$ is the number of free processors in u and B is the set of the (up-to) m' jobs $j \in I$ with the highest priority and $P_j = -$. If B contains jobs with execution times zero, define B to only contain such jobs. This differentiation is necessary, as the start of jobs with execution times zero might cause other jobs to become available and thus change the set of the m' available jobs with the highest priority. Then, $p_r = \min(\{p_j \mid j \in B\} \cup \{P_j \mid j \in I \wedge P_j \neq -\})$ is the time that passes until the next job finishes and $C = \{j \in B \mid p_j = p_r\} \cup \{j \in I \mid P_j = p_r\}$ contains the jobs that finish next.

We define all states $u' = (I', P')$ and transitions $f = (u, u')$ with $w(f) = p_r$ to be part of H if the following holds

- $I' = (I \setminus C) \cup S$ contains all jobs $j \in I$ that have not been completed ($j \notin C$) and the set of jobs S that become available. Therefore S contains exactly one successor for each $j \in C$ that is the start of a conditional pair and all successors of other jobs in C for which all predecessors have been finished.
- P'_j remains the same for all jobs that were not being processed in u and have not been started ($P'_j = -$ for all $j \in I \setminus B$ with $P_j = -$). The jobs that become available are not being processed, $P'_j = -$ for all $j \in S$. For all jobs that have been processed or started in u but have not finished, the remaining execution time is decreased by p_r. That is, $P'_j = P_j - p_r$ for all $j \in I$ with $P_j > p_r$ and $P'_j = p_j - p_r$ for all $j \in B$ with $p_j > p_r$.

Note that a state u can have multiple successors in H, as multiple job sets S can become available due to conditional constructs. Additionally observe that we can decide whether a predecessor j' of a completed job $j \in C$ has been finished. A predecessor j' has finished if either $j' \in C$ holds or if neither j' nor predecessors of j' are elements of $I \setminus C$.

In summary, the state graph H contains the start state u_0, all states and transitions that can be reached from u_0, and the state $d = (\emptyset, -)$ as the sink of H. A straightforward induction on the construction implies the following.

Lemma 3. *There is a realization G_J with a makespan of C_J if and only if there is a u_0-d-path in H with weight C_J.*

By Lemma 3 we can compute the worst-case makespan $M(G, \prec)$ for a given cp-task as follows: we construct the state graph H and find a longest path from the start state u_0 to the end state d. The corresponding worst-case realization G_J and the corresponding FP-schedule can be found by backtracking.

Observe that each state is of polynomial size and the successor states of a given state u can be computed in polynomial time. It remains to show that the number of states in H is pseudo-polynomial in the input size.

As argued before, width k of G implies that there are $\mathcal{O}(n^k)$ different sets for I. Moreover, for each I there are at most $p_{\max}$ possible elements for P, where $p_{\max} = \max_{j \in V} p_j$. Thus, there are $\mathcal{O}(n^k \cdot p_{\max}^k)$ states, which is pseudo-polynomial as k is constant. Thus, a dynamic program can compute the longest u_0-d-path for H in $\mathcal{O}(n^{2k} \cdot p_{\max}^{2k})$ and, by Lemma 3, solves CDAG-MAX exactly. This proves Theorem 12.

V. FPTAS under Monotonicity

We present a *fully polynomial-time approximation scheme (FPTAS)* for CDAG-MAX for a certain class of *monotone* conditional DAGs, which we define below.

Definition 4. A family of algorithms $\{A_\varepsilon\}$ is called FPTAS if, for every input I and every $\varepsilon > 0$, algorithm A_ε finds a solution of value within a factor of $1 + \varepsilon$ (resp. $1 - \varepsilon$) of the optimal solution for I and the running time of A_ε is polynomial in the encoding of I and $1/\varepsilon$.

Roughly speaking, a scheduling algorithm is *monotone* if increasing the execution times does not decrease the makespan.

Definition 5. A scheduling algorithm is *monotone*, if for any pair of scheduling instances I and I', that differ only in one job j with $p_j < p'_j$, the respective makespans C_I and C'_I for each instance satisfy the following:

(a) $C_I \leq C_{I'}$, and
(b) $C_{I'} \leq C_I + \delta$ with $\delta = p'_i - p_i$.

In general, FP-scheduling for conditional DAGs is not monotone, even if each realization is of bounded width, see the Graham anomalies [6].

We consider graphs G that have in each realization G_J bounded width and FP-schedules on G_J that are *monotone*. Conditional DAGs that consist of a constant number of chains in each realization belong to this class. The proof is omitted.

Theorem 13. *Let I be a scheduling instance such that the precedence constraint graph G is a set of disjoint chains, then each FP-schedule of I is monotone.*

We now design an FPTAS for CDAG-MAX. Let $I = (G, p, m, \prec)$ be an instance with $G = (V, E, C)$ and $|V| = n$. For fixed $\varepsilon > 0$, set $\mu = \varepsilon \cdot p_{\max}/n$ where $p_{\max} = \max_{j \in V} p_j$ is the maximum execution time in I. We define algorithm A_ε:

1) Let $\hat{I} = (G, \hat{p}, m, \prec)$ be a rounded CDAG-MAX instance with $\hat{p}_j = \lceil \frac{p_j}{\mu} \rceil$ for each $j \in V$.
2) Solve the rounded instance $\hat{I}$ using the dynamic program (DP) from Section IV. Let $G_{\hat{J}}$ be the realization that corresponds to the longest u_0-d-path determined by DP.
3) Return $\mu \cdot \hat{C}_{\hat{J}}$ where $\hat{C}_{\hat{J}}$ is the completion time of $G_{\hat{J}}$ on the rounded instance $\hat{I}$.

This definition follows the rather standard method of rounding the values to reduce the state space and thus the running time of the pseudo-polynomial time DP. It remains to prove that the solution quality does not deteriorate too much.

Theorem 14. *Algorithm $\{A_\varepsilon\}$ is an FPTAS for* CDAG-MAX *for conditional DAGs such that each realization has a width bounded by a constant and FP-schedules are monotone.*

Proof. We first show that the runtime of each A_ε is polynomial in the input size and $\frac{1}{\varepsilon}$. The runtime of A_ε is dominated by the time that the DP takes for solving the rounded instance $\hat{I}$, which is $\mathcal{O}(n^{2k} \cdot \hat{p}_{\max}^{2k})$. By definition of $\hat{I}$, we have:

$$\hat{p}_{\max} = \lceil p_{\max}/\mu \rceil = \lceil p_{\max} \cdot n/(\varepsilon p_{\max}) \rceil \leq \lceil n/\varepsilon \rceil .$$

Thus, the overall runtime is $\mathcal{O}(n^{4k}/\epsilon^{2k})$, which is polynomial in the input size and $\frac{1}{\varepsilon}$.

It remains to show that algorithm A_ε computes for any given instance I a solution of value $A_\varepsilon(I)$ that satisfies

$$M(G, \prec) \leq A_\varepsilon(I) \leq (1 + \varepsilon) \cdot M(G, \prec).$$

Consider an instance I, $\varepsilon > 0$ and A_ε. Let J^* be a realization with $C_{J^*} = M(G, \prec)$ on instance I and let $\hat{J}$ be the realization computed by A_ε, i.e., by the DP on the rounded instance $\hat{I}$ in Step 2). For a realization $J \in \mathcal{J}$ let

C_J and $\hat{C}_J$ denote the makespans of J in instance I and its rounded variant $\hat{I}$, respectively. The value for CDAG-MAX computed by A_ε is $A_\varepsilon(I) = \mu \cdot \hat{C}_{\hat{J}}$.

First, we observe that for every realization $J \in \mathcal{J}$ holds:

$$C_J \leq \mu \cdot \hat{C}_J. \tag{1}$$

This observation is crucial and relies on the monotonicity property. In general, Inequality (1) may not be true for FP-scheduling [6]. To see (1), consider an instance I' obtained from I by scaling all execution times down by the factor μ. For any realization G_J, the makespan C_J of the original instance I and the makespan C'_J for the scaled instance I' satisfy $C_J = \mu \cdot C'_J$. As G_J is monotone by assumption and as execution times in $\hat{I}$ equal those in I' rounded up to the next integer values, the makespan of the FP-schedule of $\hat{I}$ satisfies $C'_J \leq \hat{C}_J$ (cf. Definition 5(a)), which implies (1).

Algorithm A_ε computes in Step 2) an exact solution for the rounded instance $\hat{I}$, i.e., $\hat{C}_{\hat{J}}$ is the worst-case makespan for $\hat{I}$ which is not less than the makespan $\hat{C}_{J^*}$ for J^*. Using Inequality (1) we conclude

$$M(G, \prec) = C_{J^*} \leq \mu \cdot \hat{C}_{J^*} \leq \mu \cdot \hat{C}_{\hat{J}} = A_\varepsilon(I)$$

It remains to show that $A_\varepsilon(I) \leq (1+\varepsilon) \cdot M(G, \prec)$. Similar to the arguments above, we can show the following inequality by repeatedly applying monotonicity property (b) in Definition 5. For every realization $J \in \mathcal{J}$ holds:

$$\hat{C}_J < C_J/\mu + n. \tag{2}$$

Combining (2) with the definition of μ, we conclude

$$\begin{aligned} A_\varepsilon(I) = \mu \cdot \hat{C}_{\hat{J}} &< C_{\hat{J}} + n \cdot \mu = C_{\hat{J}} + \varepsilon \cdot p_{\max} \\ &\leq (1+\varepsilon) \cdot M(G, \prec), \end{aligned}$$

which completes the proof. □

We observe that by symmetry we have an FTPAS $\{A'_\varepsilon\}$ that returns the solution $A'_\varepsilon(I) = \left\lceil \frac{1}{1+\varepsilon} \cdot A_\varepsilon(I) \right\rceil$ that satisfies

$$(1-\varepsilon) \cdot M(G, \prec) \leq A'_\varepsilon(I) \leq M(G, \prec).$$

As mentioned above, we can show that conditional DAGs that consist of a constant number of chains in each realization satisfy the monotonicity property of Theorem 13. With Theorem 14 this implies the existence of an FPTAS which is best possible given the weak coNP-hardness (Theorem 5).

Corollary 1. *Algorithm $\{A_\varepsilon\}$ is an FPTAS for* CDAG-MAX *for conditional DAGs G such that each realization of G is a constant number of disjoint chains.*

VI. Conclusion

In this work, we resolve the complexity status of the CDAG-MAX problem in conditional DAGs. We obtained refined results depending on the graph structure. While we were able to show that CDAG-MAX is (weakly) coNP-hard for $m = 2$ processors if each realization of the given conditional DAG consists of $k = 4$ chains, the complexity for $m = 2$ and $k = 3$ remains open. As LS-MAX for three chains on two processors is in P, we cannot hope to settle this open question by using our reduction framework.

Our reduction from LS-MAX to CDAG-MAX relies on the ability to arbitrarily assign priorities. In particular, the reduction does not work if we require all jobs of the same conditional construct to have the same priority. We leave the complexity of this special case open. It seems to be an interesting and relevant case as it is a common approach in practice to assign priorities on a thread level in multi-threading scenarios.

References

[1] F. Wu, Q. Wu, and Y. Tan, "Workflow scheduling in cloud: a survey," *The Journal of Supercomputing*, vol. 71, no. 9, pp. 3373–3418, 2015.

[2] S. Singh and I. Chana, "A survey on resource scheduling in cloud computing: Issues and challenges," *J. Grid Comput.*, vol. 14, no. 2, pp. 217–264, 2016.

[3] S. K. Baruah, V. Bonifaci, A. Marchetti-Spaccamela, L. Stougie, and A. Wiese, "A generalized parallel task model for recurrent real-time processes," in *RTSS*. IEEE Computer Society, 2012, pp. 63–72.

[4] V. Bonifaci, A. Wiese, S. K. Baruah, A. Marchetti-Spaccamela, S. Stiller, and L. Stougie, "A generalized parallel task model for recurrent real-time processes," *TOPC*, vol. 6, no. 1, pp. 3:1–3:40, 2019.

[5] J. D. Ullman, "NP-complete scheduling problems," *J. Comput. Syst. Sci.*, vol. 10, no. 3, pp. 384–393, 1975.

[6] R. L. Graham, "Bounds on multiprocessing timing anomalies," *SIAM Journal on Applied Mathematics*, vol. 17, no. 2, pp. 416–429, 1969.

[7] S. Baruah, V. Bonifaci, and A. Marchetti-Spaccamela, "The global EDF scheduling of systems of conditional sporadic DAG tasks," in *ECRTS*, 2015, pp. 222–231.

[8] A. Melani, M. Bertogna, V. Bonifaci, A. Marchetti-Spaccamela, and G. C. Buttazzo, "Response-time analysis of conditional dag tasks in multiprocessor systems," in *ECRTS*, 2015, pp. 211–221.

[9] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana, "Business process execution language for web services," 2013.

[10] J. Chen, G. von der Brüggen, J. Shi, and N. Ueter, "Dependency graph approach for multiprocessor real-time synchronization," in *RTSS*, 2018, pp. 434–446.

[11] J. C. Fonseca, V. Nélis, G. Raravi, and L. M. Pinho, "A multi-dag model for real-time parallel applications with conditional execution," in *SAC*. ACM, 2015, pp. 1925–1932.

[12] S. Chakraborty, T. Erlebach, and L. Thiele, "On the complexity of scheduling conditional real-time code," in *Algorithms and Data Structures*, F. Dehne, J.-R. Sack, and R. Tamassia, Eds., 2001, pp. 38–49.

[13] S. Chakraborty, T. Erlebach, S. Kunzli, and L. Thiele, "Schedulability of event-driven code blocks in real-time embedded systems," in *DAC*, 2002, pp. 616–621.

[14] T. Erlebach, V. Kääb, and R. H. Möhring, "Scheduling and/or-networks on identical parallel machines," in *Approximation and Online Algorithms*, R. Solis-Oba and K. Jansen, Eds. Springer Berlin Heidelberg, 2004, pp. 123–136.

[15] S. Baruah, "The federated scheduling of systems of conditional sporadic DAG tasks," in *EMSOFT*. IEEE, 2015, pp. 1–10.

[16] M. Garey and D. Johnson, "Strong NP-completeness results: motivation, examples, and implications," *J. Assoc. Comput. Mach.*, vol. 25, no. 3, pp. 499–508, 1978.

[17] A. Agnetis, M. Flamini, G. Nicosia, and A. Pacifici, "Scheduling three chains on two parallel machines," *European Journal of Operational Research*, vol. 202, no. 3, pp. 669–674, 2010.

[18] J. Lenstra and A. Rinnooy Kan, "Complexity of scheduling under precedence constraints," *Oper. Res.*, vol. 26, no. 1, pp. 22–35, 1978.

[19] T. J. Schaefer, "The complexity of satisfiability problems," in *STOC*. ACM, 1978, pp. 216–226.

[20] R. H. Möhring, "Computationally tractable classes of ordered sets," in *Algorithms and Order*, I. Rival, Ed. Springer, 1989, pp. 105–193.

[21] R. P. Dilworth, "A decomposition theorem for partially ordered sets," *Annals of Mathematics*, vol. 51, no. 1, pp. 161–166, 1950.