On the Volume Calculation for Conditional DAG Tasks: Hardness and Algorithms*

Jinghao Sun[1], Yaoyao Chi[1], Tianfei Xu[1], Lei Cao[1], Nan Guan [2,†] , Zhishan Guo[3], and Wang Yi[4]
1. Northeastern University, China; 2. The Hong Kong Polytechnic University, China
3. University of Central Florida, USA; 4. Uppsala University, Sweden

***Abstract*—The hardness of analyzing conditional directed acyclic graph (DAG) tasks remains unknown so far. For example, previous researches asserted that the conditional DAG's volume can be solved in polynomial time. However, these researches all assume *well-nested* structures that are recursively composed by single-source-single-sink parallel and conditional components. For conditional DAGs in general that do not comply with this assumption, the hardness and algorithms of volume computation are still open. In this paper, we construct counterexamples to show that previous work cannot provide a safe upper bound of the conditional DAG's volume in general. Moreover, we prove that the volume computation problem for conditional DAGs is strongly $\mathcal{NP}$-hard. Finally, we propose an exact algorithm for computing the conditional DAG's volume. Experiments show that our method can significantly improve the accuracy of the conditional DAG's volume estimation.**

***Index Terms*—DAG, Conditional branches, Volume, $\mathcal{NP}$-hard**

I. Introduction

Nowadays, multicores are becoming mainstream hardware platforms for embedded and real-time systems. To fully utilize the processing capacity of multicores, programs are parallelized. Directed acyclic graph (DAG), a natural model to formulate parallel programs, recently has gained a lot of attention in real-time communities, and motivates much theoretical work on real-time scheduling and analysis of DAG models [1]–[10].

However, the standard DAG model cannot fully capture the characteristics of parallel programs. One important difference is that the program code has not only intra-task parallelism, but also conditional structures (such as `if-else` statements). Inspired by this, the conditional DAG modeling both intra-task parallelism and conditional branches has been proposed and analyzed [11]–[15].

Conditional DAGs are more difficult to analyze, i.e., traditional problems on non-conditional DAGs are polynomial-time solvable, but many of them become $\mathcal{NP}$-hard on conditional DAGs. Nevertheless, many researchers still devote to propose polynomial-time algorithms for conditional DAGs, even though the problem on conditional DAGs is inherently $\mathcal{NP}$-hard. Consider the task volume computation[1] problem as an example, in non-conditional DAGs, since every vertex must be executed exactly one time, the volume of DAG equals the summation of all vertices' execution time. However, when `if-else` structures are brought into DAGs, the number of possible execution flows on the conditional DAG is exponential. The volume of conditional DAGs, which is the maximum total execution time among all possible execution flows, is more complicated to be solved. Although Baruah [13] and Melani et.al [15] propose polynomial-time algorithms for computing the conditional DAG's volume, their DAG task models assume *well-nested* structures recursively composed by single-source-single-sink parallel and conditional components. For the *non-well-nested* conditional DAGs that do not comply with this assumption, the hardness of computing volume is still open.

In this paper, we investigate the hardness of the conditional DAG's volume computation. First, we construct counterexamples to show that the algorithm in previous work cannot exactly derive the volume of the non-well-nested conditional DAGs in general, and even cannot provide a safe upper bound (of its volume). Then we formally prove that the volume computation problem for conditional DAGs in general is strongly $\mathcal{NP}$-hard, indicating that there is no (pseudo)-polynomial time algorithm for calculating (precisely) a conditional DAG's volume. Finally, we propose an exact algorithm for the conditional DAG's volume computation. Although the algorithm in general runs in exponential time, we show that under some special cases, the time complexity can be polynomial. Experimental work shows that our algorithm dramatically improves the accuracy of the conditional DAG's volume estimation.

The rest of this paper is organized as follows. Sec. II presents related work. Sec. III formally defines conditional DAG models and relevant notations. Sec. IV reveals the drawbacks of existing work. Sec. V analyzes the complexity of conditional DAG's volume computation. Sec. VI proposes the exact algorithm, and Sec. VII reports our experimental results. Sec. VIII concludes this paper.

II. Related Work

Conditional DAG models are investigated in [11]–[15]. To compute the conditional DAG's volume, [12], [13] transform conditional DAGs to equivalent non-conditional DAGs, and then the volume computation method designed for non-conditional DAGs can be applied. [14], [15] develop a dynamic program to compute the conditional DAG's volume directly.

*This work is supported by NSFC (61972076, 61772123, 61532007, 61602104), GRF (15213818, 15204917) and NSF(CNS-1850851).

†Corresponding Author:Nan Guan,nan.guan@polyu.edu.hk

[1] The volume of DAG is a very important parameter for the DAG task's response time analysis. Generally speaking, computing volume is the first step for analyzing DAG task's response time.

978-3-9819263-4-7/DATE20/©2020 EDAA

Although the previous methods have polynomial time complexity, they are all restricted to well-nested DAGs. In [12], [13], the transformation is applied from innermost conditional components to outermost components, ensuring that the transformations for conditional components are independent with each other. The dynamic programming in [14], [15] always chooses the branch with the maximum volume from each conditional component. It may bring inaccuracy if non-well-nested DAGs are considered (See in Sec. IV for details). Therefore, these existing techniques cannot deal with non-well-nested DAGs. Sun et.al [16] solves the response time of non-well-nested DAGs, but their method is restricted to OpenMP programs.

III. System Model

In this section, we formally define the conditional DAG model and its execution semantics. We also introduce the relevant notations of the conditional DAG's volume computation.

A. Conditional DAG Model

We define the conditional DAG as $G = (V, E)$, where V is the set of vertices, and E is the set of edges. Each vertex v_i of V is associated with the worst-case execution time (WCET) $c(v_i)$. Each edge (v_i, v_j) of E represents the dependency between vertices v_i and v_j, indicating that v_i must complete execution before vertex v_j can begin execution. A vertex v_i is the *predecessor* of vertex v_j if there is an edge from v_i to v_j, and in this case, vertex v_j is called the *successor* of v_i. Moreover, a vertex v_j is the *descendant* of v_i if v_j is a successor of v_i or a successor of the descendant of v_i. For each vertex v_i, we use $\text{Pred}(v_i)$ to denote the set of v_i's predecessors, and use $\text{Succ}(v_i)$ to denote the set of v_i's successors, and use $\text{Desc}(v_i)$ to denote the set of v_i's descendants. A vertex v_i is called the *source* vertex of G if it has no predecessor. A vertex v_i is called the *sink* vertex of G if it has no successor. Without loss of generality, we assume that each conditional DAG has exactly one source vertex v_{src} and one sink vertex v_{snk}[2].

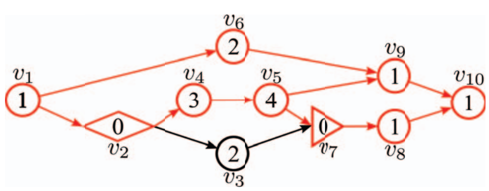

Fig. 1. An example of the conditional DAG.

Conditional DAGs distinguish two types of vertices: (1) **regular vertices**, represented as circles, formulate the sequential chunk of execution (or "sub-task"); (2) **conditional vertices**, coming in pairs and denoted by diamonds and triangles, represent the *entry* and the *exit* of a conditional component respectively. A vertex v_l belongs to a *conditional component* $\mathcal{P}$ if v_l is in a path from $\mathcal{P}$'s entry vertex to $\mathcal{P}$'s exit vertex. Fig. 1 shows an example conditional DAG, where vertices v_2 and v_7 are the entry and the exit of a conditional component including three regular vertices v_3, v_4 and v_5.

[2]If this is not the case, a dummy source/sink vertex with zero WCET is added to G with arcs to/from all the source/sink vertices.

A conditional DAG G is *well nested* if for any conditional component $\mathcal{P}$ of G, there is no edge from the regular vertex of $\mathcal{P}$ to the vertex outside $\mathcal{P}$. The DAG in Fig. 1 is *non-well nested* since there is an edge from v_5 (inside a conditional component) to v_9 (outside the conditional component).

B. Execution Semantics

The execution of conditional DAG G starts with the source vertex v_{src} and ends at the sink vertex v_{snk}. During the execution, at any time once a vertex v_i is completed,

a1. If v_i is the entry vertex of a condition component, exactly one of its successors should be executed. For example, in Fig. 1, once the conditional entry vertex v_2 is executed, either v_3 or v_4 is executed.

a2. Otherwise, all of v_i's successors should be executed. For example, in Fig. 1, once vertex v_1 is executed, then vertices v_2 and v_6 are both executed.

When encountering a vertex v_i that should be executed,

b1. If v_i is the exit vertex of a conditional component, v_i is *eligible* to be executed once one of its predecessors is completed. For example, in Fig. 1, once one of the vertices v_3 and v_5 is completed, then the vertex v_7 is executed.

b2. Otherwise, v_i is *eligible* to be executed only when all of its predecessors are completed. For example, in Fig. 1, once vertices v_8 and v_9 are completed, then vertex v_{10} is executed.

Definition 1. *An execution flow $\mathcal{E}$ of conditional DAG G is the subgraph of G that contains all the vertices executed in an execution of G satisfying (a1),(a2), (b1) and (b2) above.*

An execution flow is marked in red in Fig. 1.

Definition 2. *The volume $vol(G)$ of G is the maximum total execution time (workload) among all the execution flows of G, i.e.,*

$$vol(G) = \max_{\mathcal{E} \in G} \sum_{v_i \in \mathcal{E}} c(v_i) \tag{1}$$

where $\mathcal{E}$ is the execution flow of G.

IV. Problems in Existing Work

A straightforward way to solve the volume $vol(G)$ of G is to enumerate all possible execution flows of G, and then choose the one maximizing the volume. However, this is computationally intractable since the number of possible execution flows is exponential of `if-else` component numbers. Instead of explicit enumeration of execution flows, previous work (e.g., [15]) proposes a dynamic program (DP) to solve the conditional DAG's volume in polynomial time[3]. But we

[3]In the literature, [12], [13] also propose a polynomial-time method to solve the conditional DAG's volume, which first transforms the conditional DAG into an equivalent non-conditional DAG, and then compute the non-conditional DAG's volume. The transformation method is rather complicate, and moreover it results in the same value of conditional DAG's volume as the DP method does. Therefore, we only discuss the DP method in this paper.

observe that the DP algorithm may not lead to the correct volume $vol(G)$ of the conditional DAG G discussed in Sec. III. In the following, we first briefly introduce the DP algorithm in [15], and then we construct a counterexample to reveal drawbacks of the DP algorithm.

Revisit of Melani's DP Algorithm [15]. The pseudo-code of the DP algorithm is given in Alg. 1, which has the complexity in time quadratic in the conditional DAG's size.

Algorithm 1: Volume computation in [15].

1 $\sigma \leftarrow$ TopologicalOrder(G)
2 $S(v_{snk}) \leftarrow \{v_{snk}\}$
3 **for** $v_i \in \sigma$ *from sink to source* **do**
4   **if** *Succ*$(v_i) \neq \emptyset$ **then**
5   **if** v_i *is the entry of a conditional component* **then**
6       $v^* \leftarrow \text{argmax}_{v_j \in \text{Succ}(v_i)} C(S(v_j))$
7       $S(v_i) \leftarrow \{v_i\} \cup S(v^*)$
8   **else**
9       $S(v_i) \leftarrow \{v_i\} \cup \bigcup_{v_j \in \text{Succ}(v_i)} S(v_j)$
10 **return** $C(S(v_{src}))$

The algorithm exploits the (reverse) topological order $\sigma =$ TopologicalOrder(G) of the conditional DAG G (Line 1), and computes for each vertex the accumulated workload corresponding to the portion of the graph already examined. More precisely, for each vertex v_i, Alg. 1 uses $S(v_i)$ to denote the set of v_i and v_i's descendants that achieves the maximum workload of the subgraph $D(v_i)$ that contains v_i and v_i's descendants (as well as their associated edges). Moreover, Alg. 1 uses $C(S(v_i))$ to denote the total WCET of the vertices in $S(v_i)$. For each vertex v_i under analysis, Alg. 1 distinguishes the following two cases:

- If v_i is the entry vertex of a conditional component, we select the successor v^* of v_i, i.e., $v^* \in \text{Succ}(v_i)$, such that v^* achieves the maximum accumulated workload among v_i's all successors (Line 6), and then we merge $S(v^*)$ and v_i into the set $S(v_i)$ of v_i (Line 7).
- Otherwise, the workload contributions of all successors of v_i must be merged into $S(v_i)$ (Line 9).

The following example shows that Alg. 1 cannot solve the conditional DAG's volume exactly, and moreover, it fails to bring a safe upper bound of the conditional DAG's volume.

Example 1. *The conditional DAG G in Fig. 2 consists of two conditional components. The first conditional component has an entry vertex v_3, an exit vertex v_{10} and regular vertices v_6 and v_7. The second conditional component has an entry vertex v_2, an exit vertex v_8 and regular vertices v_4 and v_5. It is noting that the vertices v_5 and v_6 from different conditional components point to the same regular vertex v_9 that is outside the conditional components. The execution time of v_9 is 15. The execution time of v_7 and v_4 is 10. Other vertices have unit execution time. According to Lines 5 to 7 of Alg. 1, for the entry vertex of a conditional component, the branch with the maximum volume is always selected for the further computation. Therefore, the vertex set $S(v_2)$ of the conditional entry vertex v_2 includes v_2, v_5, v_8, v_9 and v_{11}, and moreover, the vertex set $S(v_3)$ of the conditional entry vertex v_3 contains v_3, v_6, v_9, v_{10} and v_{11}. As shown in Line 9 of Alg. 1, the vertex sets $S(v_2)$ and $S(v_3)$ are further merged to compute the vertex set $S(v_1)$ of the vertex v_1, i.e., $S(v_1) = \{v_1\} \cup S(v_2) \cup S(v_3) = \{v_1, v_2, v_3, v_5.v_6, v_8, v_9, v_{10}, v_{11}\}$, and thus, the volume $vol(G)$ computed by Alg. 1 equals 23. Actually, the execution flow with the maximum volume is $\{v_1, v_2, v_3, v_4, v_7, v_8, v_{10}, v_{11}\}$, which has the volume 26. Clearly, the volume computed by Alg. 1 is much smaller than the actual one.*

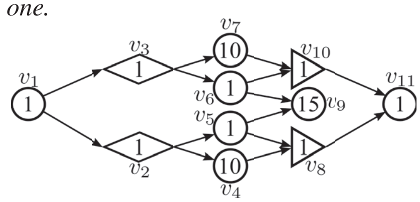

Fig. 2. The counterexample of Alg. 1.

The example above shows that the DP algorithm of [15] cannot correctly estimate the conditional DAG's volume, and even cannot exhibit a safe bound for the volume. Actually, Alg. 1 is restricted to well-nested DAGs. For non-well-nested conditional DAGs, we prove that the volume computation problem is strongly $\mathcal{NP}$-hard as shown in the next section.

V. Strong $\mathcal{NP}$-Hardness

In order to show $\mathcal{NP}$-hardness in the strong sense of the volume computation problem for conditional DAG models, we provide a reduction from the classical 3SAT problem [17] described as follows.

INSTANCE: Given a boolean expression $\mathscr{E}$ in conjunctive normal form (CNF) that is the conjunction of clauses, each of which is the disjunction of three distinct literals.

QUESTION: Is $\mathscr{E}$ satisfiable?

Proposition 1 ([17]). *3SAT is strongly $\mathcal{NP}$-Complete.*

The reduction from 3SAT to the volume computation for a conditional DAG G is as follows. Given an instance of 3SAT $\mathscr{E}$ that is the conjunction of n clauses, i.e., $\mathscr{E} = C_1 \wedge C_2 \wedge \cdots \wedge C_n$. Each clause C_i of $\mathscr{E}$ is the disjunction of three distinct literals, i.e., $C_i = c_{i1} \vee c_{i2} \vee c_{i3}$, where each literal c_{ik} is either a variable or the negation of a variable in the set X of m variables $\{x_1, \cdots, x_m\}$ taking values in the boolean set $\{0,1\}$. We construct the conditional DAG G (as shown in Fig. 3) with the following properties.

1) A witness of the truth of the condition $vol(G) \geq n$ will give a satisfying assignment for $\mathscr{E}$, and vice versa.
2) The number of vertices of G and all involved values need to be polynomially bounded in the size of $\mathscr{E}$.

The first requirement above is sufficient to assert that the conditional DAG's volume computation is $\mathcal{NP}$-hard. The second requirement above is necessary to establish $\mathcal{NP}$-hardness in the *strong* sense.

As shown in Fig. 3, the conditional DAG G constructed in the reduction contains the follow two parts.

The first part forks m conditional components $\mathscr{C} = \{\mathcal{C}_1, \cdots, \mathcal{C}_m\}$ from the source vertex v_{src}, and then joins all the components of $\mathscr{C}$ into the vertex v_{mid}. Each component $\mathcal{C}_i$ of $\mathscr{C}$ has two branches B_i and $\bar{B}_i$, which respectively correspond to the variable x_i of X and the negation $\bar{x}_i$ of x_i.

The second part forks n conditional components $\mathscr{C}' = \{\mathcal{C}'_1, \cdots, \mathcal{C}'_n\}$ from vertex v_{mid}, and then joins the components of $\mathscr{C}'$ into the sink vertex v_{snk}. Each component $\mathcal{C}'_i$ of $\mathscr{C}'$ has three branches B_{i1}, B_{i2} and B_{i3}, which respectively correspond to literals c_{i1}, c_{i2} and c_{i3} of C_i.

For each variable x_i of X, for each clause C_j of $\mathscr{E}$ and for each literal c_{jk} of C_j, if $c_{jk} = x_i$, we add an edge from B_i to B_{jk}. If $c_{jk} = \bar{x}_i$, we add an edge from $\bar{B}_i$ to B_{jk}. These edges are marked red in Fig. 3.

For each component $\mathcal{C}'_i$ of $\mathscr{C}$, we set its conditional exit vertex with unit execution time. The other vertices of G all have zero execution time. (See in Fig. 3)

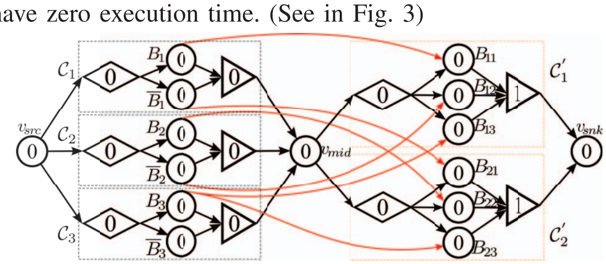

Fig. 3. An example of the conditional DAG constructed in the reduction which corresponds to the 3SAT instance such as $\mathscr{E} = (x_1 \vee \overline{x}_2 \vee x_3) \wedge (\overline{x}_1 \vee x_2 \vee x_3)$.

We now check the two properties from above. In the following we first prove that the second property above is satisfied as shown in Lem. 1.

Lemma 1. *The number of vertices of G and all involved values are polynomially bounded in the size of $\mathscr{E}$.*

Proof. There are $4m + 5n + 3$ vertices and no more than $(4m + 5n + 3)^2$ edges in G. Moreover, each vertex has 0/1 execution time. $\square$

In the following we prove the first property's satisfaction. First we show that each assignment of X corresponds to an execution flow $\mathcal{E}$ of G:

i. For each $\mathcal{C}_i$ of $\mathscr{C}$, if $x_i = 1$, the branch B_i of $\mathcal{C}_i$ is visited in $\mathcal{E}$. Otherwise, the branch $\bar{B}_i$ of $\mathcal{C}_i$ is visited.
ii. For each $\mathcal{C}'_j$ of $\mathscr{C}'$, without loss of generality, let $c_{jk} = x_i$ $(1 \le k \le 3)$, which indicates that there is an edge from B_i outside $\mathcal{C}'_j$ to the branch B_{jk} inside $\mathcal{C}'_j$. According to Semantic (b2), the branch B_{jk} of $\mathcal{C}'_j$ is visited in $\mathcal{E}$ only if B_i is visited.

Lemma 2. *$C_j = 1$ if and only if the branch of $\mathcal{C}'_j$ is visited.*

Proof. Without loss of generality, we assume that the literal c_{jk} of C_j is associated with the variable x_i of X, i.e., $c_{jk} = x_i$. In this case, there is an edge from B_i to B_{jk} in G.

Sufficiency. If a branch B_{jk} of $\mathcal{C}'_j$ is visited, according to Semantic (b2), the predecessor B_i of B_{jk} must be visited. According to (i) above, we know that $x_i = 1$. Since $c_{jk} = x_i$, we have $C_j = \bigvee_{k=1}^{3} c_{jk}$ is true, i.e., $C_j = 1$.

Necessity. If $C_j = 1$, and without loss of generality, we assume that $c_{jk} = 1$. Since $c_{jk} = x_i$, we have $x_i = 1$. According to (i) above, we know that B_i must be visited in $\mathcal{E}$. Moreover, according to (ii) above, we know that the branch B_{jk} of $\mathcal{C}'_j$ is visited in $\mathcal{E}$. This completes the proof. $\square$

Based on Lem. 2, we show the first property's satisfaction in the following lemma.

Lemma 3. *$vol(G) \ge n$ if and only if $\mathscr{E}$ is satisfiable.*

Proof. Sufficiency. If there is an assignment of X such that $\mathscr{E}$ is satisfied, i.e., each clause C_j of $\mathscr{E}$ is true. According to Lem. 2, there is an execution flow $\mathcal{E}$ such that at least one branch of every component $\mathcal{C}'_j$ of $\mathscr{C}'$ is successfully visited. From Semantic (b1), the exit vertex (with unit execution time) of each component $\mathcal{C}'_j$ of $\mathscr{C}'$ must be visited (exactly once) in $\mathcal{E}$. Thus, the volume of $\mathcal{E}$ is n.

Necessity. If $\mathscr{E}$ is unsatisfiable, i.e., given any assignment of X, at least one clause C_j of $\mathscr{E}$ is false. According to Lem. 2, the execution flow $\mathcal{E}$ cannot visit the branch of $\mathcal{C}'_j$, and from Semantic (b1), the exit vertex of $\mathcal{C}'_j$ cannot be visited in $\mathcal{E}$. Therefore, the volume of $\mathcal{E}$ must be less than n. $\square$

Finally, we summarize our main result in the following theorem.

Theorem 1. *The volume computation problem for conditional DAGs is $\mathcal{NP}$-hard in the strong sense.*

Proof. The theorem is correct since the two properties from above for a proper reduction are satisfied by Lem. 1 and 3. $\square$

VI. Exact Algorithm

For the general conditional DAG G, we propose an exact algorithm for its volume computation. Similar to Alg. 1, we also use a simple dynamic program exploring the topological order of G. The main difference is that when storing a vertex v_i, we check whether the execution of v_i relies on the execution of vertices in the portion of the graph that has not been examined. The pseudo-code is shown in Alg. 2.

In Alg. 2, for each vertex v_i, we use $\Omega(v_i)$ to collect the vertex set S that contains v_i and v_i's descendants, and moreover, all the vertices of S must be in the same execution flow, i.e., there is an execution flow $\mathcal{E}$ on G such that $S \subseteq \mathcal{E}$. First, at Line 1, the topological sorting of the vertices of G is computed and stored in the permutation σ. Then, the permutation σ is scanned in the reverse order, i.e., from the sink vertex v_{snk} to the source vertex v_{src} of G (Line 2). For each vertex v_i, its associated $\Omega(v_i)$ is initialized at Line 3. There are two possibilities. If v_i has no successors (e.g., v_i is the sink vertex), the vertex set containing the single vertex v_i is added into $\Omega(v_i)$ (Lines 18 to 19). Otherwise, v_i has (multiple) successors ($\mathrm{Succ}(v_i) \neq \emptyset$ at Line 4), the computation of $\Omega(v_i)$ considers the following two cases.

Case 1. If v_i is the entry vertex of a conditional component, for any successor v_j of v_i, and for any vertex set S' of $\Omega(v_j) \cup \{\emptyset\}$, we construct the vertex set S by using S' as shown in Lines 8 to 10, and then add S into $\Omega(v_i)$. At Line 8, the operation "$\uplus$" is denoted as follows. For any two vertex sets S_1 and S_2, $S_1 \uplus S_2 = S_1 \cup S_2$ if there is an execution flow $\mathcal{E}$ on G such that $S_1 \cup S_2 \subseteq \mathcal{E}$, and otherwise, $S_1 \uplus S_2 = \emptyset$.

Case 2. Otherwise, v_i is not the entry vertex of a conditional component. In this case, we construct the product $\Pi(\mathrm{Succ}(v_i))$ as follows. Without loss of generality, let $\mathrm{Succ}(v_i) = \{v_{j_1}, \cdots, v_{j_k}\}$, where v_{j_l} is the l-th successor of v_i ($1 \leq l \leq k$). The product $\Pi(\mathrm{Succ}(v_i)) = \Omega(v_{j_1}) \times \Omega(v_{j_2}) \times \cdots \times \Omega(v_{j_k})$. For each $P = (S_{j_1}, S_{j_2}, \cdots, S_{j_k}) \in \Pi(\mathrm{Succ}(v_i))$ with $S_{j_l} \in \Omega(v_{j_l}) \cup \{\emptyset\}$, we compute $\cup P$ as $S_{j_1} \cup S_{j_2} \cup \cdots \cup S_{j_k}$. At Lines 14 to 16, we construct the vertex set S by using $\cup P$, and then add S into $\Omega(v_i)$.

Algorithm 2: Exact volume computation.

1 $\sigma \leftarrow \mathrm{TopologicalOrder}(G)$
2 **for** $v_i \in \sigma$ *from sink to source* **do**
3 $\quad \Omega(v_i) = \emptyset$
4 $\quad$ **if** *Succ*$(v_i) \neq \emptyset$ **then**
5 $\quad\quad$ **if** v_i *is the entry of a conditional component* **then**
6 $\quad\quad\quad$ **for** *any* $v_j \in$ *Succ*(v_i) **do**
7 $\quad\quad\quad\quad$ **for** *any* $S' \in \Omega(v_j) \cup \{\emptyset\}$ **do**
8 $\quad\quad\quad\quad\quad S \leftarrow S' \uplus \{v_i\}$
9 $\quad\quad\quad\quad\quad$ **if** $S = \emptyset$ **then**
10 $\quad\quad\quad\quad\quad\quad S \leftarrow \{v_i\}$
11 $\quad\quad\quad\quad\quad \Omega(v_i) \leftarrow \Omega(v_i) \cup \{S\}$
12 $\quad\quad$ **else**
13 $\quad\quad\quad$ **for** *any* $P \in \Pi($*Succ*$(v_i))$ **do**
14 $\quad\quad\quad\quad S \leftarrow (\cup P) \uplus \{v_i\}$
15 $\quad\quad\quad\quad$ **if** $S = \emptyset$ **then**
16 $\quad\quad\quad\quad\quad S \leftarrow \{v_i\}$
17 $\quad\quad\quad\quad \Omega(v_i) \leftarrow \Omega(v_i) \cup \{S\}$
18 $\quad$ **else**
19 $\quad\quad \Omega(v_i) \leftarrow \Omega(v_i) \cup \{\{v_i\}\}$
20 $\quad$ compress $\Omega(v_i)$ by Alg. 3
21 **return** $\max\{C(S) | S \in \Omega(v_{src})\}$

Collection Compression. For each vertex v_i, its collection $\Omega(v_i)$ is compressed by Alg. 3 to remove the vertex set that does not contribute to the maximum volume.

Algorithm 3: Compressing the collection $\Omega(v_i)$.

1 **for** *any vertex set* $S' \in \Omega(v_i)$ **do**
2 $\quad$ **if** *there is a vertex set* $S \in \Omega(v_i) - \{S'\}$ *such that* $A(S) \subseteq A(S')$ *and* $C(S) \geq C(S')$ **then**
3 $\quad\quad$ remove S' from $\Omega(v_i)$

Alg. 3 removes S' from $\Omega(v_i)$ if the condition of the `if` clause at Line 2 holds. At Line 2, for any $S \in \Omega(v_i)$, $A(S)$ is denoted as follows.

$$A(S) = \bigcup_{v_j \in S - \{v_i\}} \mathrm{Pred}(v_j) - D(v_i) \tag{2}$$

where $D(v_i)$ is the subgraph of G that contains v_i and v_i's descendants, i.e., $D(v_i) = \mathrm{Desc}(v_i) \cup \{v_i\}$. Intuitively, $A(S)$ stores the vertices that are outside S but can affect the execution of S. More precisely, S is successfully executed only if all the vertices in $A(S)$ are executed. The parameter $C(S)$ at Line 2 denotes the total execution time (workload) of the vertices in S. The `if` condition at Line 2 indicates that compared with S, the vertex set S' achieves smaller workload and is affected by more vertices in the portion of the graph that has not been examined. Therefore, S is the candidate to contribute to the maximum volume rather than S', i.e., S' should be removed.

By applying Alg. 2 on the conditional DAG G in Fig. 2, more than one vertex set is stored in the collection $\Omega(v_2)$ of the conditional entry vertex v_2, e.g., $S_1 = \{v_2, v_5, v_8, v_9, v_{11}\}$ and $S_2 = \{v_2, v_4, v_8, v_{11}\}$. By (2), we have $A(S_1) = \{v_6, v_{10}\}$ and $A(S_2) = \{v_{10}\}$. Since $A(S_2) \subseteq A(S_1)$ and $C(S_2) < C(S_1)$, i.e., the condition of `if` at Line 2 of Alg. 3 is violated, neither S_1 nor S_2 is removed from $\Omega(v_2)$ after compression. With similar reasons, the collection $\Omega(v_3)$ of v_3 contains more than one vertex set after compression, e.g., $S_3 = \{v_3, v_7, v_{10}, v_{11}\}$ and $S_4 = \{v_3, v_6, v_9, v_{10}, v_{11}\}$. To compute the collection $\Omega(v_1)$ of v_1, we construct the product $\Pi(\mathrm{Succ}(v_1))$ as $\Omega(v_2) \times \Omega(v_3)$, and according to Lines 13 to 17 of Alg. 2, we obtain $\Omega(v_1)$ including $S_5 = \{v_1, v_2, v_3, v_4, v_7, v_8, v_{10}, v_{11}\}$ and $S_6 = \{v_1, v_2, v_3, v_5, v_6, v_8, v_9, v_{10}, v_{11}\}$. Since $A(S_5) = A(S_6) = \emptyset$ and $C(S_5) > C(S_6)$, i.e., the condition of `if` at Line 2 of Alg. 3 holds, the collection $\Omega(v_1)$ is further compressed as $\Omega(v_1) = \{S_5\}$. According to Line 21 of Alg. 2, the volume $vol(G)$ is eventually computed as 26. Clearly, our algorithms can exactly compute the conditional DAG's volume.

Complexity. Lem. 4 shows the complexity of Alg. 2. Before going into details, we first introduce an useful notation below.

Definition 3 (maximum predecessor cut)**.** *For any vertex v_i of G, its predecessor cut (PC) is the set of vertices below.*

$$Cut(v_i) = \bigcup_{v_j \in Desc(v_i)} Pred(v_j) - D(v_i) \tag{3}$$

The maximum PC of G is $Cut(G) = \arg\max_{v_i \in G} |Cut(v_i)|$.

For example, in Fig. 1, the PC of v_5 is $\mathrm{Cut}(v_5) = \{v_3, v_6\}$.

Lemma 4. *The runtime of Alg. 2 is polynomially bounded by the vertex number n if the maximum predecessor cut $Cut(G)$ of G has a constant cardinality $K = |Cut(G)|$.*

Proof. In Alg. 2, we store a collection $\Omega(v_i)$ of vertex sets for each vertex of G. According to Line 20 of Alg. 2, the collection $\Omega(v_i)$ is compressed before used for further computation. After compression, any set S of Ω corresponds to a unique $A(S)$ according to Alg. 3. Moreover, since $A(S)$ is a subset of $\mathrm{Cut}(v_i)$, the cardinality of $\Omega(v_i)$ is bounded by the number of subsets of $\mathrm{Cut}(v_i)$, i.e., $|\Omega(v_i)| \leq 2^{|\mathrm{Cut}(v_i)|}$. Therefore, the total number of stored vertex sets is bounded by $\sum_{v_i \in G} 2^{|\mathrm{Cut}(v_i)|} \leq n2^K$, where $K = |\mathrm{Cut}(G)|$.

Furthermore, as shown in Lines 8 to 11 and Lines 14 to 17 of Alg. 2, each stored set S is computed within polynomial time. This completes the proof. □

VII. Evaluation

This section evaluates our algorithm using randomly generated task graphs. For each task instance, we compare the volume V_1 computed by Melani's algorithm (Alg. 1) and the volume V_2 computed by our algorithm (Alg. 2). Moreover, we also show the computation time of Alg. 2, where the algorithm is coded in python 3.7, and the code runs on a PC with Intel core i5-6300U CPU at 2.4GHz with 8G RAM.

We randomly generate conditional DAGs using TGFF tool [18], a DAG generator developed to facilitate standardized random benchmarks for scheduling research. More precisely, we construct the DAG G by using the series-parallel algorithm of TGFF described as follows. We generate a root vertex that is connected to a set of chains of vertices[4]. The number of chains and the length of each chain are in the ranges $[2, 5]$ and $[1, 3]$ respectively. The chains that connect to the same root will rejoin at an extra (end) vertex by the probability p_{rjn}. A root vertex connected to a set of chains along with the corresponding end vertex (if it is generated) is called a series-parallel unit (SPU) [18].

The algorithm is performed recursively: we first generate one SPU, and assume that G contains at most n vertices. If the graph has less than the required number n of vertices, one vertex inside the SPU is designated as the root of a new SPU. This process is repeated until the graph has the requested number n of vertices. After the DAG G generation, we randomly set a SPU of G to be *conditional* by the probability p_{cnd}. Moreover, to make the DAG G more flexible, for any vertex v_i that is inside a conditional SPU and v_i's descendant v_j that is outside the conditional SPU, we randomly generate the edge from v_i to v_j by the probability p_{jmp}. The execution time of each vertex is in the range $[10, 100]$.

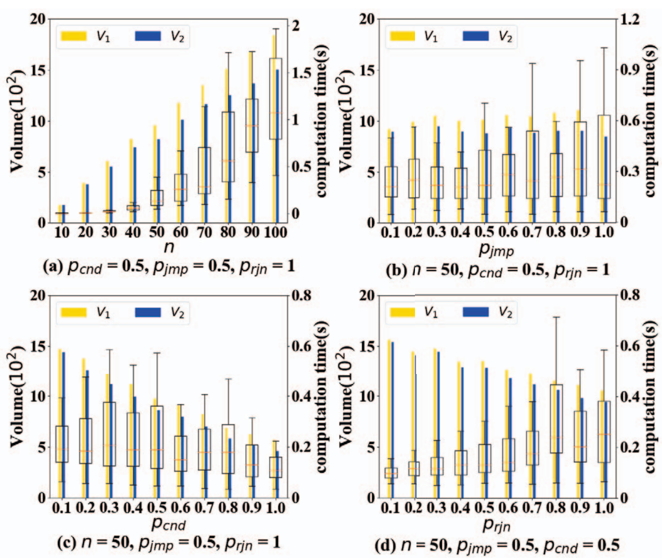

Fig. 4. Evaluation results for different configurations.

We conduct experiments with different combinations of parameters in Fig. 4. The values of configurations are written in the figure caption. For each data point, 1000 random experiments have been run. We observe that the volumes V_1 and V_2 have the same trend under different configurations. V_2 is smaller than V_1 since Alg. 2 eliminates the infeasible execution flows which are incorrectly accepted by Alg. 1.

[4]A chain of vertices is one or more vertices that are linked together in series to form a chain.

Alg. 1 can solve each instance within 1s, while the computation time of Alg. 2 is much larger, i.e., The average computation time of Alg. 2 is 27.7s. In some special cases, computing the volume of a DAG with 60 vertices and 150 edges may cost more than 2000s. Nearly 92% instances in our experiments can be solved by Alg. 2 within 2s, and we only exhibit the computation time of these instances by using the box plot[5] of Fig. 4.

In Fig. 4(a), the gap between volumes V_1 and V_2 (the volume gap for short) becomes larger as the number of vertices increases, and so does the computation time of Alg. 2. In Fig. 4(b), the volume gap and the computation time both become larger as the probability p_{jmp} increases. In Fig. 4(c), the volume gap becomes larger, and the computation time increases and then slightly decreases when the probability p_{cnd} increases. In Fig. 4(d), the volume gap and the computation time both become larger as the probability p_{rjn} increases.

VIII. Conclusion

In the real-time community, many DAG models have been proposed, but few of them consider conditional branches and analyze the non-well-nested conditionals. In this paper, we investigate the non-well-nested conditional DAGs, and prove that the volume computation for conditional DAGs is strongly $\mathcal{NP}$-hard. Moreover, we propose an exact algorithm for computing the conditional DAG's volume. Experiments show the effectiveness of our method.

References

[1] A Saifullah et al. Multi-core real-time scheduling for generalized parallel task models. *RTS*, 2013.
[2] J Li et al. Global EDF scheduling for parallel real-time tasks. *RTS*, 2015.
[3] D Ferry et al. A real-time scheduling service for parallel tasks. In *RTAS*, 2013.
[4] A Saifullah et al. Parallel real-time scheduling of DAGs. *IEEE Trans on PDS*, 2014.
[5] S Baruah. Improved multiprocessor global schedulability analysis of sporadic DAG task systems. In *ECRTS*, 2014.
[6] M Qamhieh et al. Global EDF scheduling of directed acyclic graphs on multiprocessor systems. In *RTNS*, 2013.
[7] M Qamhieh et al. A stretching algorithm for parallel real-time DAG tasks on multiprocessor systems. In *RTNS*, 2014.
[8] M Serrano et al. Timing characterization of OpenMP4 tasking model. In *CASES*, 2015.
[9] R Vargas et al. OpenMP and timing predictability: a possible union? In *DATE*, 2015.
[10] J Sun et al. Real-time scheduling and analysis of OpenMP task systems with tied tasks. In *RTSS*, 2017.
[11] J Fonseca et al. A multi-DAG model for real-time parallel applications with conditional execution. In *SAC*, 2015.
[12] S Baruah et al. The global EDF scheduling of systems of conditional sporadic DAG tasks. In *ECRTS*, 2015.
[13] S Baruah. The federated scheduling of systems of conditional sporadic DAG tasks. In *Emsoft*, 2015.
[14] A Melaniet et al. Response-time analysis of conditional DAG tasks in multiprocessor systems. In *ECRTS*, 2015.
[15] A Melani et al. Schedulability analysis of conditional parallel task graphs in multicore systems. *IEEE Trans on Computers*, 2017.
[16] J Sun et al. Calculating response-time bounds for openmp task systems with conditional branches. In *RTAS*, 2019.
[17] S Cook. The complexity of theorem-proving procedures. In *SOTC*, 1971.
[18] R Dick et al. TGFF task graphs for free. In *IWHC*, 1998.

[5]In a box plot, the top and the bottom of the box respectively represent the first and third quartiles of data. The middle line of the box represents the median of data. The whiskers extended from the box show the range of data.