Response Time Analysis for Prioritized DAG Task with Mutually Exclusive Vertices

Ran Bi[1], Qingqiang He[2], Jinghao Sun[1*], Zhenyu Sun[1], Zhishan Guo[3], Nan Guan[4], Guozhen Tan[1]

[1]Dalian University of Technology, China; [2]Hong Kong Polytechnic University, Hong Kong;
[3]North Carolina State University, U.S; [4]City University of Hong Kong, Hong Kong

Abstract—**Directed acyclic graph (DAG) becomes a popular model for modern real-time embedded software. It is really a challenge to bound the worst-case response time (WCRT) of DAG task. Parallelism, dependencies and mutual exclusion become three of the most critical properties of real-time parallel tasks. Recent work applied prioritizing techniques to reduce DAG task's WCRT bound, which has well studied the first two properties, i.e., parallelism and dependencies, but leaves the mutually exclusive property as an open problem. This paper focuses on all the three properties of real-time parallel software, and investigates how to estimate the WCRT of the DAG task model with mutually exclusive vertices and under prioritized list scheduling algorithms. We derive a reasonable WCRT bound for such a complicated DAG task, and prove that the corresponding WCRT bound computation problem is strongly NP-hard. It means that there are no pseudo-polynomial time algorithms to compute the WCRT bound. For the prioritized DAG with a constant number of mutual exclusive vertices, we develop a dynamic programming algorithm that is able to estimate the WCRT bound within pseudo-polynomial time. Experiments are conducted to evaluate the performance of our analysis method implemented with different priority assignment policies against the state-of-the-art.**

I. Introduction

Directed acyclic graph (DAG) can fully explore fine-grained parallelism of emerging complex applications, which has risen in popularity over the last decade [1]–[8]. Nowadays, modeling a system as an event- or time-triggered DAG task is a common setup in many modern domains, such as automotive, robotics, and industrial automation. For example, in self-driving area, a complete automotive task chain from perception to control is converted to a DAG task [2]. Moreover, researchers use DAG task to model the execution sequences of multiple deep neural networks (DNNs) across computation nodes during the perception process [1]. These domains often require to execute a DAG task upon an embedded multi-core platform and under real-time scenarios. DAG parallelism techniques in existing parallel programming frameworks (e.g., OpenMP [9], Threading Building Blocks (TBB) [10] and Cilk family [11]) are commonly developed for general purpose high-performance computing (HPC) domain, and it is really challenging to adopt them to real-time embedded applications.

*The corresponding author: Dr. Jinghao Sun, jhsun@dlut.edu.cn

†This work is partially supported by the National Key Research and Development Program of China (No. 2021ZD0112400), the National Natural Science Foundation of China (No. 61972076, U1808206), Hong Kong Research Grant Council (GRF 11208522 and GRF 15206221).

One of the most concerned problems is how to derive a safe upper bound for the response time of a DAG task. Graham [12] developed the first response time analysis for the DAG task based on the notion of *critical* path. An important property of the critical path is that maximizing its execution length leads to the worst-case response time (WCRT) of the corresponding DAG task. This implies a concise WCRT bound (called Graham's bound) which is composed of two parts: One is the length of the longest path in the DAG task, and the other one records the workload that may interfere with the execution of vertices in the longest path. However, the main drawback is that all vertices (except the ones in the longest path) are considered to disrupt the execution of the longest path, making Graham's bound overly pessimistic.

Recent work [13]–[15] attempted to utilize real-time mechanisms (e.g., prioritizing and preemptive techniques) to schedule the DAG and to guarantee a much smaller WCRT bound. He et al. [13] proposed the *prioritized* DAG task model by assigning different priorities to vertices of the DAG task. When scheduling the prioritized DAG task, the execution order of vertices is determined by vertex priorities. A vertex of the critical path can only incur delay from the concurrent vertices with higher priorities, and the interference vertices of the critical path are significantly reduced. It eventually derives a more reasonable WCRT bound (called He's bound in the rest of this paper), which dominates Graham's bound [12].

Unfortunately, these existing theoretical results cannot provide a safe WCRT bound when applied on some practical embedded systems. The reason is that the DAG task model which is originally proposed in HPC domains cannot fully capture embedded system's behaviors. Embedded systems usually have limited resources due to the requirement of low power consumption. Under limited resource constraints, more parallel vertices in a DAG task have to share the same resource (e.g., PCI bus, network on-chip, and related hardware interfaces), and are enforced to execute in a mutually exclusive way. Blocking time caused by mutual exclusion, which is not mainly concerned in HPC domain, becomes an important factor that may dramatically worsen DAG task's WCRT and cannot be ignored in real-time embedded systems. DAG task models can only express the common features (e.g., parallelism and dependencies) of general purpose HPC applications, but fail to reveal the feature of mutually exclusive (ME) vertices. It is still an open problem how to guarantee the

WCRT bound for the prioritized DAG task with ME vertices.

In this paper, we focus on the DAG task model with ME vertices (abbreviated as the ME-DAG task model for the sake of convenience). We use a DAG as the basic graph, and extend it into a mixed graph (i.e., ME-DAG) by adding undirected edges to express the ME relations between vertices. We derive a WCRT bound for the prioritized ME-DAG task, which can be calculated by finding the longest simple path in the mixed graph where the weight of each vertex v is determined by not only v's execution time but also the interference vertices that may block v's execution. We prove that this WCRT bound computation problem is strongly NP-hard even if there is only a single type of priority assigned to vertices.

We propose a dynamic programming algorithm to efficiently estimate our proposed WCRT bound. The main idea of our algorithm is to solve the longest path (that corresponds to the WCRT bound) by merging promising sub-paths. Instead of enumerating paths, we abstract paths into concise data structures (called tuples), and the path merging process is equivalently implemented by recursively computing tuples, i.e., new tuples are computed by using the tuples that have been computed beforehand. The major challenge we face is that a vertex may be repeatedly traveled in a path since a mixed graph contains cycles, and moreover, interference vertices of a path may be overly estimated. By bringing more information about ME vertices in the tuple and by carefully designing the way two paths are merged, the repeatedly traveled vertices are totally forbidden and the overestimation of interference vertices is drastically relieved. Our algorithm has a pseudo-polynomial time complexity if there are a constant number of ME vertices in the DAG task model. In the evaluation work, we conduct comprehensive experiments under different settings to compare the performance of our analysis method with the state-of-the-art DAG analysis technique which can also deal with ME vertices but simply assigns all vertices to the same priority. We implement several priority assignment strategies that are commonly used in the existing research work, and show how significantly these priority assignment strategies can reduce the WCRT bound of ME-DAG tasks.

II. Related Work

There is plentiful literature on scheduling algorithms and analysis techniques for multiple recurrent DAG tasks upon multi-core processors [16]–[30], which can be classified into three different paradigms: global scheduling [16]–[22], decomposition-based scheduling [23]–[25], and federated scheduling [26]–[30]. All these work concerns how to reduce the worst-case response time (WCRT) for multi-DAG task, which is mainly determined by two critical issues: inter-task interference [19], [22] and intra-task interference [13]–[15], [20], [31]. The WCRT of a single DAG is used to bound the intra-task interference, which is the focus of this paper.

Graham [12] proposed the first WCRT bound for the single DAG task. Many researchers apply Graham bound to more practical applications [32]–[43]. Recent work found that the WCRT bound becomes much smaller than Graham bound if assigning each vertex to a different priority and determining the execution order of vertices based on vertices priorities. [13] proposed the first WCRT bound for prioritized DAG tasks, and they developed a polynomial-time algorithm to compute the WCRT bound. However, their method is restricted to the special case that the vertex priority must comply with the topological order of DAG. [15] relaxed the constraint in [13], and proposed a polynomial-time WCRT bound computation method to handle any arbitrary priority assignment. [14] explored parallelism and dependencies in DAG structure, and gave a priority assignment policy and response time bounds based on concurrent provider and consumer model. All these existing works do not consider mutually exclusive vertices.

Recently there has been some work considering mutually exclusive (ME) vertices in DAG tasks. All the studies applied the real-time locking protocol to deal with ME issues. There are two major lock types: spin locks and suspension-based semaphores. [44] firstly analyzed the blocking time of non-preemptive spin-lock under federated scheduling, which was later improved by [45]. [46] extended spin locking protocol to the DAG task with OpenMP semantics. [47] and [48] implemented spin-based analysis for DAG tasks under a finer-grained resource model. In [49], a suspension-based protocol called Limited Pending Protocol (LPP) and associated blocking analysis were proposed for DAG tasks. Suspension-based locking protocols OMLP [50] and OMIP [51] for clustered scheduling were extended to DAG tasks by [52] and [53]. The blocking analysis in each existing work is designed for a specified locking protocol, and may be inefficient or even fail to bound the blocking time when some details of the protocol are changed. Moreover, all the existing analysis work is based on the classic Graham bound, which is quite pessimistic and needs to be improved by using prioritizing techniques. The present paper is not oriented toward a detailed locking protocol, but aims to study a general problem and reveal the inherent complexity brought by mutually exclusive vertices. To the best of our knowledge, this paper is the first work that considers both vertex-level priorities and mutually exclusive vertices in a DAG, which can provide a new possibility for improving the performance of real-time locking protocols in DAG tasks by using prioritizing techniques.

III. System Model

In Section III-A, we propose a formal model for the DAG task with prioritized and mutually exclusive vertices. In Section III-B, we introduce how to schedule such a task upon multi-core platform by prioritized list scheduling algorithms.

A. Task Model

We formulate a parallel real-time task τ as a prioritized graph-based task model $\tau = (D, E)$, where D represents the prioritized DAG structure of τ, and E defines mutually exclusive relations between vertices.

§ Prioritized DAG structure

The DAG D is further defined as $D = (V, A)$, where V is the set of n vertices, and A is the set of m arcs. Each vertex

v_i of V represents a continuous piece of executing code. We use $c(v_i)$ to denote the worst-case execution time (WCET) of v_i, and use an integer $p(v_i)$ to denote the priority of v_i. The larger $p(v_i)$ is, the lower priority it represents. The arc $a = (v_i, v_j)$ of A represents the precedence relation between vertices v_i and v_j, indicating that v_j cannot start its execution before v_i is completed. In this case, v_i is the *predecessor* of v_j, and v_j is the *successor* of v_i. We use $\text{pred}(v_i)$ and $\text{succ}(v_i)$ to denote the set of predecessors and successors of v_i, respectively. Moreover, we call vertex v_i as the *ancestor* of vertex v_j if v_i is (a predecessor of) v_j's predecessor, and in this case, v_j is the *descendant* of v_i. We use $\text{anc}(v_i)$ and $\text{des}(v_i)$ to denote the set of ancestors and descendants of v_i, respectively. A vertex is called the source vertex (the sink vertex) of G if it does not have predecessors (successors). Without loss of generality, we assume that there is a single source vertex v_{src} and a single sink vertex v_{snk} in G. If G has multiple source/sink vertices, we add a dummy source/sink vertex with zero WCET to comply with our assumption.

Example 1. *An example of DAG D that contains 10 vertices and 12 arcs is given in Fig. 1. The source vertex and sink vertex of D are v_1 and v_{10}, respectively. The WCET is labeled inside the vertex. Vertex v_6 has two predecessors v_3 and v_5, and two successors v_7 and v_8. Moreover, the set of ancestors of v_6 is $anc(v_6) = \{v_1, v_2, v_3, v_5\}$. The set of descendants of v_6 is $des(v_6) = \{v_7, v_8, v_{10}\}$. We define 3 types $\{0, 1, 2\}$ of priorities, which are labeled in red beside vertices. Moreover, the blue edges represent mutually exclusive (ME) edges, which will be introduced later in this section.*

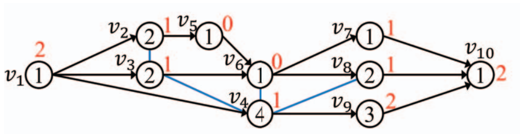

Fig. 1: An example prioritized DAG with ME vertices.

§ Mutually Exclusive Vertices

We use an undirected edge $[v_i, v_j]$ to connect two vertices v_i and v_j, indicating that v_i and v_j are enforced to execute in a mutually exclusive (ME) way. We call $[v_i, v_j]$ as the ME edge, and we use E to denote the set of all ME edges. Vertices v_i and v_j are *neighbors* if there is an edge $[v_i, v_j] \in E$. We use $\text{neb}(v_i)$ to denote the set of neighbors of v_i. We call a vertex v_i as the ME vertex if v_i is associated with ME edges, and we use V_{ME} to store all ME vertices, i.e., $V_{\text{ME}} \subset V$. As shown in Fig. 1, there are four ME edges $[v_2, v_3], [v_3, v_4], [v_4, v_6]$, and $[v_4, v_8]$. ME vertices include v_2, v_3, v_4, v_6, and v_8. For the sake of convenience, we use $G = (D, E)$ to denote the mixed graph that contains the DAG D and the ME edges of E. We call D as the *basic graph* of G, and call G as the *DAG with ME vertices*, abbreviated as *ME-DAG* for short.

B. Scheduling Model

We execute the prioritized ME-DAG G on a multi-core platform $\mathcal{C} = \{c_1, \cdots, c_m\}$ that consists of m identical cores. At any time t, a vertex of G can only execute on one core, and meanwhile, a core cannot execute two vertices simultaneously. A vertex v_i is *eligible* to execute if all its predecessors are finished and there are no unfinished neighbors of v_i. Here we say a vertex is *unfinished* if it has already started the execution but has not finished yet. Once a core becomes idle, it always selects an eligible vertex to execute. The execution order of vertices is determined according to the vertex's priority. The execution of a vertex is preemptive. More specifically, the execution of a vertex v_i may be interrupted by an eligible vertex v_j if v_j has a higher priority. Given two vertices with the same priority, the vertex that becomes eligible earlier should execute first. When two vertices with the same priority become eligible at the same time, we arbitrarily choose one of them to execute first. At any time t, we always choose at most m eligible vertices with the highest priorities for execution. Moreover, vertex migration is allowed, i.e., if vertex v_i starts its execution on a core c_k and is interrupted before its completion, v_i can resume its execution on another core c_l.

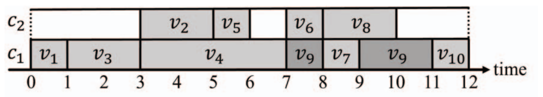

Fig. 2: An example schedule of the ME-DAG G in Fig. 1.

Example 2. *Fig. 2 gives a possible schedule of the ME-DAG G in Fig. 1 upon a dual-core platform. Here we assume that all the vertices execute at their worst-case execution time. The ME-DAG G starts and finishes its execution at time 0 and 12, respectively. Vertex v_2 cannot execute when v_3 is executing, though the predecessor v_1 of v_2 has finished. This is because v_3 is v_2's neighbor that starts before v_2, and v_2 cannot be eligible to execute unless v_1 and v_3 both finish, making core c_2 idle during the execution of v_3. Vertex preemption occurs at time 8, i.e., v_9 is interrupted by vertex v_7 which has a higher priority and becomes eligible at that time.*

IV. Response Time Analysis

The *response time* $R(\tau)$ of a parallel real-time task τ (as defined in the above section) is the length of the time interval that starts with the execution of the source vertex v_{src} and ends at the completion of the sink vertex v_{snk}. In the following, we derive upper bounds of the response time $R(\tau)$ for prioritized ME-DAG task. Before going into details, we first introduce some useful notations below.

A. Preliminaries

We call a vertex sequence $\pi = (v_1, \cdots, v_k)$ as a *path* of the mixed graph G if v_i is either the predecessor of v_{i+1} or the neighbor of v_{i+1} for each $i = 1, \cdots, k-1$. A path π is *simple* if it does not travel a vertex twice. Moreover, a path π is *feasible* if it is simple and it does not travel a vertex v_i before traveling v_i's ancestors. A path π is a *complete path* if it is feasible and it starts with the source vertex and ends at the sink vertex of G. For example, as shown in Fig.1, path $\pi_1 = (v_3, v_6, v_4, v_3)$ is not simple as it travels v_3 twice. Path $\pi_2 = (v_2, v_5, v_6, v_4, v_3)$ is simple but not feasible, as it first travels v_6 , and then travels v_6's ancestor v_3. Path $\pi_3 = (v_2, v_5, v_6, v_4, v_9)$ is a feasible path. Path $\pi_4 = (v_1, v_2, v_5, v_6, v_4, v_9, v_{10})$ is a complete path. We only focus on feasible paths without special mention in the rest of this paper.

For any path π and for any ME vertex $v_i \in V_{\text{ME}}$, we define a function $\alpha_\pi(v_i)$ to indicate whether v_i or the ancestor of v_i is traveled in π, i.e.,

$$\alpha_\pi(v_i) = \begin{cases} 1 & (\text{anc}(v_i) \cup \{v_i\}) \cap \pi \neq \emptyset \\ 0 & \text{else} \end{cases} \quad (1)$$

Similarly, we define a function $\beta_\pi(v_i)$ to indicate whether v_i or the descendant of v_i is traveled in π, i.e.,

$$\beta_\pi(v_i) = \begin{cases} 1 & (\text{des}(v_i) \cup \{v_i\}) \cap \pi \neq \emptyset \\ 0 & \text{else} \end{cases} \quad (2)$$

For any path π and for any vertex $v_i \in \pi$, we say that a vertex v_j is the *pioneer* of v_i if π travels v_j before v_i. Otherwise, v_j is the *follower* of v_i. We let $\text{pin}_\pi(v_i)$ and $\text{flw}_\pi(v_i)$ be the sets of v_i's pioneers and v_i's followers, respectively. For any vertex $v_i \in \pi$, we separately denote the set of (ancestors of) v_i's pioneers and the set of (descendants of) v_i's followers as follows.

$$\text{acp}_\pi(v_i) = \bigcup_{v_j \in \text{pin}_\pi(v_i)} (\text{anc}(v_j) \cup \{v_j\}) \quad (3)$$

$$\text{dsf}_\pi(v_i) = \bigcup_{v_j \in \text{flw}_\pi(v_i)} (\text{des}(v_j) \cup \{v_j\}) \quad (4)$$

For two ending vertices u and v of a path $\pi = (u, \cdots, v)$, we provide the lower bound of $\text{dsf}_\pi(u)$ and $\text{acp}_\pi(v)$ by using indicator functions as follows and as shown by Lem. 1.

$$\overline{\text{dsf}}_\pi(u) = \bigcup_{v_i \in V_{\text{ME}} \wedge \alpha_\pi(v_i)=1} \text{des}(v_i) \quad (5)$$

$$\overline{\text{acp}}_\pi(v) = \bigcup_{v_i \in V_{\text{ME}} \wedge \beta_\pi(v_i)=1} \text{anc}(v_i) \quad (6)$$

Lemma 1. *$\overline{dsf}_\pi(u) \subseteq dsf_\pi(u)$ and $\overline{acp}_\pi(v) \subseteq acp_\pi(v)$ for any path $\pi = (u, \cdots, v)$.*

Proof. For any ME vertex $v_i \in V_{\text{ME}}$, $\beta_\pi(v_i) = 1$ indicates that a vertex $v_x \in \{v_i\} \cup \text{des}(v_i)$ is traveled in π according to (2). We know that $\text{anc}(v_i) \subseteq \text{anc}(v_x)$, and by (3) and (6), we have $\overline{\text{acp}}_\pi(v) \subseteq \text{acp}_\pi(v)$. With the similar reason, and from (1), (4) and (5), we can prove $\overline{\text{dsf}}_\pi(u) \subseteq \text{dsf}_\pi(u)$. □

Example 3. *We consider the path $\pi = (v_5, v_6, v_4, v_9)$ in Fig. 1,and we only take the ending points v_5 and v_9 of π as examples. The follower set of v_5 is $flw_\pi(v_5) = \{v_6, v_4, v_9\}$, and the pioneer set of v_9 is $pin_\pi(v_9) = \{v_5, v_6, v_4\}$. According to (4) and (3), we have $dsf_\pi(v_5) = \{v_4, v_6, v_7, v_8, v_9, v_{10}\}$ and $acp_\pi(v_9) = \{v_1, v_2, v_3, v_4, v_5, v_6\}$. The indicator functions of π are given as follows.*

	v_2	v_3	v_4	v_6	v_8
$\alpha_\pi(v_i)$	0	0	1	1	1
$\beta_\pi(v_i)$	1	1	1	1	0

According to (5) and (6), we get $\overline{dsf}_\pi(v_5) = \{v_7, v_8, v_9, v_{10}\}$ and $\overline{acp}_\pi(v_9) = \{v_1, v_2, v_3, v_5\}$. We have $\overline{dsf}_\pi(v_5) \subseteq dsf_\pi(v_5)$ and $\overline{acp}_\pi(v_9) \subseteq acp_\pi(v_9)$, which are consistent with Lem. 1.

For any vertex v_i, we say that another vertex v_j is *parallel* with v_i if v_j is neither an ancestor nor a descendant nor a neighbor of v_i. The set of parallel vertices of v_i is denoted as

$$\text{par}(v_i) = \{v_j | v_j \notin \{v_i\} \cup \text{anc}(v_i) \cup \text{des}(v_i) \cup \text{neb}(v_i)\} \quad (7)$$

The *interference set* of v_i is defined as

$$\text{ins}(v_i) = \{v_j | v_j \in \text{par}(v_i) \wedge p(v_j) \leq p(v_i)\} \quad (8)$$

Taking v_3 in Fig.1 for example, the parallel vertices of v_3 is included in $\text{par}(v_3) = \{v_5, v_9\}$, and the interference set of v_3 is $\text{ins}(v_3) = \{v_5\}$. Compared with $\text{par}(v_3)$, the interference set $\text{ins}(v_3)$ excludes vertex v_9 since $p(v_9) > p(v_3)$.

For any path $\pi = (u, \cdots, v)$, we let $\overline{\pi}$ be the set of the *intermediate vertices* in π, i.e., $\overline{\pi} = \pi - \{u, v\}$. For any intermediate vertex $v_i \in \overline{\pi}$, we define the *quasi-interference set* of v_i as follows.

$$\overline{\text{ins}}_\pi(v_i) = \text{ins}(v_i) - (\text{acp}_\pi(v_i) \cup \text{dsf}_\pi(v_i)) \quad (9)$$

and the quasi interference set of $\overline{\pi}$ is defined as

$$I(\overline{\pi}) = \bigcup_{v_i \in \overline{\pi}} \overline{\text{ins}}_\pi(v_i) - (\text{ins}(u) \cup \text{ins}(v)) \quad (10)$$

Based on the above notations, the *interference set* of π is defined as follows.

$$I(\pi) = I(\overline{\pi}) \cup \text{ins}(u) \cup \text{ins}(v) \quad (11)$$

For any symbol X, we let $X_{\text{ME}}(\cdot)$ be the subset of $X(\cdot)$ that only contains ME vertices, and let $X_{\text{NM}}(\cdot)$ be the subset of $X(\cdot)$ that only contains non-ME vertices, i.e.,

$$X_{\text{ME}}(\cdot) = X(\cdot) \cap V_{\text{ME}} \text{ and } X_{\text{NM}}(\cdot) = X(\cdot) - V_{\text{ME}} \quad (12)$$

Obviously, $X(\cdot) = X_{\text{ME}}(\cdot) \cup X_{\text{NM}}(\cdot)$ and $X_{\text{ME}}(\cdot) \cap X_{\text{NM}}(\cdot) = \emptyset$. With these notations, and according to (11), the interference set of $\pi = (u, \cdots, v)$ can be written as follows.

$$I(\pi) = \text{ins}(u) \cup \text{ins}(v) \cup I_{\text{ME}}(\pi) \cup I_{\text{NM}}(\overline{\pi}) \quad (13)$$

The last item is disjoint with the first three items of (13).

For any path π, we let $\text{len}(\pi) = \sum_{i=1}^{k} c(v_i)$ be the *length* of path π, and we let $\text{vol}(I(\pi)) = \sum_{v_i \in I(\pi))} c(v_i)$ be the *volume* of the interference set of π. The *weight* of π is defined as

$$\varpi(\pi) = \text{len}(\pi) + \frac{\text{vol}(I(\pi))}{m} \quad (14)$$

Example 4. *We consider the path $\pi = (v_3, v_4, v_6, v_8)$ in Fig. 1. The set of π's intermediate vertices is $\overline{\pi} = \{v_4, v_6\}$. In the following, we solve $\overline{ins}_\pi(v_6)$ and $\overline{ins}_\pi(v_4)$, respectively. On the one hand, $\overline{ins}_\pi(v_6) = \emptyset$ since $ins(v_6) = \emptyset$. On the other hand, the pioneers and followers of v_4 are included in $pin_\pi(v_4) = \{v_3\}$ and $flw_\pi(v_4) = \{v_6, v_8\}$, respectively. According to (3) and (4), we have $acp_\pi(v_4) = \{v_1, v_3\}$ and $dsf_\pi(v_4) = \{v_6, v_7, v_8, v_{10}\}$. Since $ins(v_4) = \{v_2, v_5, v_7\}$, we get $\overline{ins}_\pi(v_4) = ins(v_4) - (acp_\pi(v_4) \cup dsf_\pi(v_4)) = \{v_2, v_5\}$. Moreover, since $ins(v_3) \cup ins(v_8) = \{v_5, v_7\}$, we have $I(\overline{\pi}) = \overline{ins}_\pi(v_4) - (ins(v_3) \cup ins(v_8)) = \{v_2\}$, and $I(\pi) = \{v_2, v_5, v_7\}$.*

Given a schedule, and for each vertex v_i, we use $s(v_i)$ and $f(v_i)$ to denote the starting time and finishing time of v_i, respectively. Denote by pren(v_i) the set of v_i's neighbors v_j that have started the execution beforehand, i.e., $s(v_j) < s(v_i)$. We define pre(v_i) = pren$(v_i)\cup$pred(v_i), recalling that pred(v_i) is the set of v_i's predecessors. Based on the notations above, we define the critical path as follows.

Definition 1 (Critical Path). *A complete path $\pi = (v_1, \cdots, v_k)$ is critical if it satisfies the following conditions.*

$$v_i = \arg \max_{v_j \in pre(v_{i+1})} f(v_j), \quad \forall i = 1, \cdots, k-1 \tag{15}$$

Formula (15) indicates that for any two adjacent vertices v_i and v_{i+1} of a critical path π ($i = 1, \cdots, k-1$), v_i has the maximum finishing time among all vertices in pre(v_{i+1}). For example, path $\pi = (v_1, v_3, v_4, v_9, v_{10})$ is the critical path of G in Fig. 1 for a given schedule as shown in Fig. 2. For the sake of convenience, we summarize the main notations used in this paper as listed in Table I.

TABLE I: Notations adopted in this paper

Notation	Description
τ	a parallel real-time task
G	the graph structure of τ
V_{ME}	The set of ME vertices in G
$R(\tau)$	The response time of the task τ
v_i	a vertex of G
$p(v_i)$	The priority of v_i
$s(v_i)$	The starting time of v_i
$f(v_i)$	The finishing time of v_i
pred(v_i)	The set of predecessors of v_i
pren(v_i)	The set of v_i's neighbors starting executions before v_i
pre(v_i)	pre(v_i) = pren$(v_i) \cup$ pred(v_i)
anc(v_i)	The set of ancestors of v_i
des(v_i)	The set of descendants of v_i
par(v_i)	The set of parallel vertices of v_i, defined in (7)
ins(v_i)	The interference set of v_i, defined in (8)
π	a path of G, i.e., $\pi = (u, \cdots, v)$
$\overline{\pi}$	The set of π's intermediate vertices, i.e., $\overline{\pi} = \pi - \{u, v\}$
len(π)	the length of π
$\varpi(\pi)$	The weight of π, defined in (14)
$I(\pi)$	The interference set of π, defined in (11)
$I(\overline{\pi})$	The quasi interference set of $\overline{\pi}$ as defined in (10)
$\alpha_\pi(v_i)$	Indicator function of an ME vertex v_i as defined in (1)
$\beta_\pi(v_i)$	Indicator function of an ME vertex v_i as defined in (2)
pin$_\pi(v_i)$	The set of v_i's pioneers
flw$_\pi(v_i)$	The set of v_i's followers
acp$_\pi(v_i)$	The set of (ancestors of) v_i's pioneers, defined in (3)
dsf$_\pi(v_i)$	The set of (descendants of) v_i's followers, defined in (4)
$\overline{\text{acp}}_\pi(v)$	The subset of acp$_\pi(v)$ as defined in (6)
$\overline{\text{dsf}}_\pi(u)$	The subset of dsf$_\pi(u)$ as defined in (5)
$\overline{\text{ins}}_\pi(v_i)$	The quasi interference set of $v_i \in \overline{\pi}$, defined in (9)

B. WCRT Bound

The response time $R(\tau)$ of τ equals the finishing time of v_{snk} minus the starting time of v_{src}, i.e.,

$$R(\tau) = f(v_{snk}) - s(v_{src}) \tag{16}$$

The time interval $T = [s(v_{src}), f(v_{snk})]$ can be divided into two disjoint parts: $T = T_{\text{C}} \cup T_{\text{NC}}$, where T_{C} contains the time points at which the critical path π is executed, and $T_{\text{NC}} = T - T_{\text{C}}$. We have

$$R(\tau) = |T_{\text{C}}| + |T_{\text{NC}}| \tag{17}$$

On the one hand, $|T_{\text{C}}|$ is bounded by the length of the critical path π. On the other hand, the bound of $|T_{\text{NC}}|$ can be derived by the following lemmas.

Lemma 2. *All cores are busy when no vertex of the critical path executes.*

Proof. Suppose not. We assume that there is a time point t, at which no vertex of the critical path π executes, and there is an idle core c_k. We let

$$t_1 = \max\{t' | t' \in T_{\text{C}} \wedge t' < t\} \tag{18}$$
$$t_2 = \min\{t' | t' \in T_{\text{C}} \wedge t' > t\} \tag{19}$$

and we let v_i and v_j be the vertices of π that execute at time points t_1 and t_2, respectively. We show that no vertex of pre(v_j) executes during the interval (t_1, t_2). Otherwise, we assume that there is a time point t_3 such that $t_3 > t_1$ and $t_3 < t_2$, and meanwhile, a vertex v_l of pre(v_j) executes at time t_3. According to (15), v_l should be in the critical path. There are two cases.

- $t_3 < t$, and in this case, we have $t_1 \geq t_3$ according to (18). It contradicts the assumption that $t_3 > t_1$.
- $t_3 > t$, and in this case, we have $t_2 \leq t_3$ according to (19). It contradicts the assumption that $t_3 < t_2$.

By now, we have proved that no vertex of pre(v_j) executes at time t. It indicates that vertex v_j can execute on core c_k at time t, which leads to a contradiction with the assumption that core c_k is idle at time t. □

Lemma 3. *The critical path π can only be blocked by the vertices in the interference set $I(\pi)$ of π.*

Proof. As we know that only intermediate vertices of the critical path π can be blocked, and we prove this theorem by showing that each intermediate vertex v_i of π can only be blocked by the vertices of $\overline{\text{ins}}_\pi(v_i)$. Suppose not. There is a vertex $v_j \notin \overline{\text{ins}}_\pi(v_i)$, and we assume that v_j blocks the execution of v_i. According to (9), there are two possible cases:

- If $v_j \notin$ ins(v_i), according to (8), v_j and v_i must sequentially execute, or v_j and v_i can execute in parallel, but v_j has a lower priority than v_i. In each of these sub-cases, v_j does not block the execution of v_i.
- Otherwise, $v_j \in$ ins$(v_i) \cap$ (acp$_\pi(v_i) \cup$ dsf$_\pi(v_i)$). Without loss of generality, we assume that $v_j \in$ acp$_\pi(v_i)$, and according to (3), v_j must be completed before v_i becomes eligible to execute. Therefore, v_j cannot block the execution of v_i. With the similar reason, we can prove that v_j cannot block the execution of v_i if $v_j \in$ dsf$_\pi(v_i)$.

This completes the proof. □

According to Lem. 2, all cores are busy at any time $t \in T_{\text{NC}}$. According to Lem. 3, only the vertices in $I(\pi)$ can execute during the time interval T_{NC}. Therefore, we have $|T_{\text{NC}}| \leq \frac{\text{vol}(I(\pi))}{m}$. Moreover, since we have proved that $|T_{\text{C}}| \leq len(\pi)$, and according to (17), the response time $R(\tau)$ is bounded by

$$R(\tau) \leq \varpi(\pi) \tag{20}$$

From Def. 1, the critical path π is one of the complete paths, and according to (20), we derive the upper bound of $R(\tau)$ as shown in the following theorem.

Theorem 1. *The response time of a prioritized ME-DAG G is bounded by*

$$R(\tau) \leq \max_{\pi \in \Pi_{\mathrm{G}}} \{\varpi(\pi)\} \tag{21}$$

where Π_{G} is the set of complete paths of G.

The adopted worst-case-response-time analysis is tight only when all ME vertices are sequentially executed and there is no overlap between the executions of ME and normal vertices.

V. Complexity of WCRT Bound Computation

We show that computing the WCRT bound in (21) is strongly NP-hard by building a Turing reduction from the Hamiltonian path (HAM-PATH) problem as described below.

Definition 2 (HAM-PATH.)**.** *Given an undirected graph $G' = (V', E')$, the HAM-PATH problem is to determine whether there is a path that visits each vertex of G' exactly once.*

Proposition 1 ([54])**.** *HAM-PATH is strongly NP-hard.*

Given any instance of HAM-PATH G' with n vertices, we construct the corresponding parallel real-time task $\tau = (D, E, P)$ as follows. The basic DAG graph D has a source vertex v_{src}, a sink vertex v_{snk} and n *normal* vertices. For each normal vertex v_i, there are two arcs (v_{src}, v_i) and (v_i, v_{snk}). No arc is between two normal vertices. Each vertex u_i and edge (u_i, u_j) of G' correspond to a normal vertex v_i of D and an edge (v_i, v_j) of E, respectively. Each vertex $v_i \in V$ has a unit WCET, i.e., $c(v_i) = 1$. Moreover, each vertex of V has the same priority. We schedule τ on m cores, where $m \geq 2$. The reduction runs in $O(n)$, which is linear in the length of the input. We illustrate the construction with Example 5.

Example 5. *The instance of HAM-PATH G' is given in Fig. 3(a). There is a Hamiltonian path $\pi_h = (u_1, u_2, u_3, u_4)$. The parallel real-time task $\tau = (D, E, P)$ built from G' is shown in Fig. 3(b). We construct a complete path $\pi = (v_{src}, v_1, v_2, v_3, v_4, v_{snk})$, which corresponds to π_h in G'. The WCRT bound in (21) equals to $len(\pi) = 4 + 2 = 6$.*

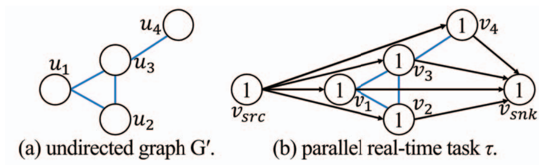

Fig. 3: An example for the construction of τ from a given graph G'.

The following two lemmas close the Turing reduction.

Lemma 4. *If there is a Hamiltonian path in G', the WCRT bound of τ in (21) is $n + 2$.*

Proof. Suppose there is a Hamiltonian path π' in G', i.e., $\pi' = (u_{[1]}, \cdots, u_{[n]})$, where $u_{[i]}$ is the i-th vertex traveled in π'. We construct a complete path $\pi = (v_{src}, v_{[1]}, \cdots, v_{[n]}, v_{snk})$. The WCRT bound of τ in (21) equals to $len(\pi) = n + 2$. □

Lemma 5. *If the WCRT bound of τ in (21) is $n + 2$, there is a Hamiltonian path in G'.*

Proof. The proof is by contradiction. We suppose that there is no Hamiltonian path in G'. It indicates that every complete path π of Π_{G} has a length less than $n + 2$, i.e.,

$$\mathrm{len}(\pi) < n + 2 \tag{22}$$

Since each vertex has the same priority, the interfering set $I(\pi)$ is $V - \{v_i | v_i \in \pi\}$. Moreover, since each vertex has a unit WCET, we have $\mathrm{vol}(I(\pi)) = |V| - \mathrm{len}(\pi)$. Therefore, the WCRT bound of τ in (21) is calculated as

$$\begin{aligned} R(\tau) &\leq \max_{\pi \in \Pi_{\mathrm{G}}} \{\mathrm{len}(\pi) + \frac{|V| - \mathrm{len}(\pi)}{m}\} \\ &= \max_{\pi \in \Pi_{\mathrm{G}}} \{\frac{n + 2 + (m-1)\mathrm{len}(\pi)}{m}\} \quad (\because |V| = n + 2) \\ &< n + 2 \quad (\because (22) \text{ and } m \geq 2) \end{aligned}$$

which leads to a contradiction. □

Theorem 2. *It is strongly NP-hard to compute the WCRT bound in (21) even if there is only one priority in P.*

Proof. The WCRT bound computation problem belongs to NP class since the bound in (21) is computed in polynomial time for a given critical path π. The theorem can be proved by the combination of Lem. 4 and Lem. 5. □

From the proof of Thm. 2, the inherent complexity of the WCRT bound computation problems highly depends on the number of ME vertices. In the following, we develop an efficient method for WCRT bound computation by using the dynamic programming technique, which is pseudo-polynomial time when there are a constant number of ME vertices.

VI. Algorithm for Computing WCRT Bound

Computing the WCRT bound in (21) is equivalent to find a complete path π^* with the maximum weight, i.e., $\varpi(\pi^*) \geq \varpi(\pi), \forall \pi \in \Pi_{\mathrm{G}}$. In the following, we develop a dynamic programming (DP) algorithm to solve such an optimal complete path π^*. Before going into details, we first sketch the main idea of our method, and introduce two main challenges our method faces, as shown in Sec. VI-A. Then we give some key theoretical results for solving the challenges, as shown in Sec. VI-B and VI-C.

A. Main Idea and Challenges

We iteratively merge sub-paths into a longer path, until the target complete path is constructed. The main difficulty is that there are exponential number of paths in Π_{G}, and we have to construct all of them to determine the one with the maximum weight in the worst case. Instead of directly enumerating paths, we abstract the path into a concise data structure, called *tuple*, and merging paths is mimicked by tuple computation process, i.e., new tuples (corresponding to longer paths) are computed by using the tuples (corresponding to sub-paths) that have been computed beforehand. When using tuples to store very limited information of paths, there are two major challenges:

- **Challenge 1:** ***How to guarantee path's feasibility?*** A path may be infeasible even if it is merged by two feasible paths. As shown in Fig. 1, the path $\pi = (v_5, v_6, v_4, v_3, v_2)$ is merged by two feasible paths $\pi_1 = (v_5, v_6, v_4)$ and $\pi_2 = (v_4, v_3, v_2)$, but it is infeasible. As our method only considers feasible paths, the information stored in tuples must be sufficient to forbid merging infeasible paths.
- **Challenge 2:** ***How to reduce overly estimated interference vertices when calculating path's weight?*** Once two paths π_1 and π_2 are merged into a path π, we aim to estimate the volume of $I(\pi)$ by using the interference sets $I(\pi_1)$ and $I(\pi_2)$. Intuitively, we can bound $vol(I(\pi))$ by $vol(I(\pi_1)) + vol(I(\pi_2)$, but this bound is too pessimistic. The reason is twofold: (1) The interference vertex sets $I(\pi_1)$ and $I(\pi_2)$ may share the same vertices. (2) The interference vertex of $I(\pi_1)$ and $I(\pi_2)$ may not be the interference vertex of $I(\pi)$. For example, in Fig. 1, v_4 and v_2 are interference vertices of $\pi_1 = (v_1, v_2, v_3)$ and $\pi_2 = (v_3, v_4, v_6, v_8)$, respectively. However, these two vertices should be removed from the interference set of path π that is merged by π_1 and π_2. It is really difficult to relieve the overestimation of interference vertices if a tuple does not know what vertices its corresponding path travels.

B. Solving Challenge 1: Key Clues

To solve **Challenge 1**, we derive the necessary and sufficient condition for constructing feasible paths during the path combination process, as shown in Thm. 3.

Theorem 3. *For any two feasible paths $\pi_1 = (u, \cdots, w)$ and $\pi_2 = (w, \cdots, v)$, the path π merged by π_1 and π_2 is feasible if and only if $\beta_{\pi_1}(v_i) = 0 \vee \alpha_{\pi_2}(v_i) = 0$, $\forall v_i \in V_{\text{ME}} - \{w\}$.*

Proof. **Necessity**. We assume that the predefined condition is violated, i.e., there is an ME vertex $v_i \in V_{\text{ME}} - \{w\}$ such that $\beta_{\pi_1}(v_i) = 1 \wedge \alpha_{\pi_2}(v_i) = 1$. In this case, we show that the merged path π is infeasible, and there are two possible cases.

- v_i is traveled in π, and without loss of generality, we assume that v_i is traveled in π_1. In this case, v_i or the ancestor of v_i is traveled in π_2 as $\alpha_{\pi_2}(v_i) = 1$. It indicates that π is infeasible.
- v_i is not traveled in π. In this case, from the assumption, the descendant v_y of v_i is traveled in π_1, and the ancestor v_x of v_i is traveled in π_2. We can conclude that v_x is the ancestor of v_y. As π travels v_y before traveling the ancestor v_x of v_y, the merged path π is infeasible.

In sum, we prove that the predefined condition must hold if the merged path π is feasible.

Sufficiency. When the predefined condition holds, we aim to prove that the merged path π is feasible. We suppose that the merged path π is infeasible, i.e., there is a vertex v_j such that π first travels v_j, and then π travels a vertex $v_i \in \{v_j\} \cup \text{anc}(v_j)$. As π_1 and π_2 both are feasible paths, each of them cannot travel both of v_j and v_i. It indicates two facts: 1) Neither v_j nor v_i equals to w. 2) The vertex v_j is traveled in π_1 and v_i is traveled in π_2. We consider the following two cases.

- If at least one of the vertices v_i and v_j is an ME vertex, i.e., $\{v_i, v_j\} \cap V_{\text{ME}} \neq \emptyset$. Without loss of generality, we assume that v_i is an ME vertex. Since the ME vertex v_i is traveled in π_2, and according to (1), we have $\alpha_{\pi_2}(v_i) = 1$. Moreover, since the descendant v_j of v_i is traveled in π_1, and according to (2), we have $\beta_{\pi_1}(v_i) = 1$. It indicates that the predefined condition is violated, which contradicts the assumption.
- Otherwise, neither of the vertices v_i and v_j belongs to the ME vertex set, i.e., $\{v_i, v_j\} \cap V_{\text{ME}} = \emptyset$. We let π' be the sub-path of π that starts with v_j and ends at v_i. The path π' must travel at least one ME edge (as well as at least one ME vertex). Suppose not, i.e., π' only contains directed arcs. It indicates that v_j is the ancestor of v_i, which leads to a contradiction. There are two sub-cases.
  - There are ME vertices in $\pi_2 \cap \pi'$. We let v_x be the last ME vertex traveled in π', i.e., for any ME vertex $v_y \in \pi'$, π' travels v_y before traveling v_x. It indicates that v_i is the descendant of the ME vertex v_x, and obviously, v_j is the descendant of v_x, i.e., $\beta_{\pi_1}(v_x) = 1$ according to (2). Moreover, since v_x is the last ME vertex of π' and $\pi_2 \cap \pi' \cap V_{\text{ME}} \neq \emptyset$, v_x must be in π_2, and according to (1), $\alpha_{\pi_2}(v_x) = 1$.
  - There are ME vertices in $\pi_1 \cap \pi'$. We let v_x be the first ME vertex traveled in π', and obviously, $v_x \in \pi_1$, i.e., $\beta_{\pi_1}(v_x) = 1$ according to (2). Moreover, v_j is the ancestor of v_x, and thus, v_i is the ancestor of v_x, i.e., $\alpha_{\pi_2}(v_x) = 1$ according to (1).

  Both of the above sub-cases lead to a violation of the predefined condition.

In sum, we prove that the predefined condition sufficiently indicates the feasibility of the merged path π. □

Thm. 3 indicates that it is not necessary to know all vertices of a path for preventing infeasible paths, and instead, it is necessary and sufficient to forbid infeasible path constructions by only storing the information of two indicator functions $\alpha_\pi : V_{\text{ME}} \rightarrow \{0, 1\}$ and $\beta_\pi : V_{\text{ME}} \rightarrow \{0, 1\}$ for a path π.

C. Solving Challenge 2: Key Clues

To solve **Challenge 2**, we derive a reasonable estimation of the interference set $I(\pi)$ of a path π, and show how to calculate the estimation's volume by only using limited information of π's sub-paths $\pi_1 = (u, \cdots, w)$ and $\pi_2 = (w, \cdots, v)$. A trivial bound of $I(\pi)$ is $I(\pi_1) \cup I(\pi_2)$, and this bound may contain some overly estimated interference vertices as shown by the example in the description of **Challenge 2**. In the following, we propose a much finer bound of $I(\pi)$. According to (13), the union of interference sets $I(\pi_1) \cup I(\pi_2)$ can be divided into the following three subsets.

$$I(\pi_1) \cup I(\pi_2) = I_0 \cup I_{\text{NM}}(\overline{\pi_1}) \cup I_{\text{NM}}(\overline{\pi_2}) \tag{23}$$

where $I_0 = I_{\text{ME}}(\pi_1) \cup I_{\text{ME}}(\pi_2) \cup \text{ins}(u) \cup \widetilde{\text{ins}(w)} \cup \text{ins}(v)$. As shown in (24) to (26), we use $\tilde{I}_1$, $\tilde{I}_2$ and $\widetilde{\text{ins}}_\pi(w)$ to denote the subsets of $I_{\text{ME}}(\pi_1)$, $I_{\text{ME}}(\pi_2)$ and $\text{ins}(w)$, respectively.

$$\tilde{I}_1 = I_{\text{ME}}(\pi_1) - \overline{\text{dsf}}_{\pi_2}(w) \quad (24)$$

$$\tilde{I}_2 = I_{\text{ME}}(\pi_2) - \overline{\text{acp}}_{\pi_1}(w) \quad (25)$$

$$\widetilde{\text{ins}}_{\pi}(w) = \text{ins}(w) - (\overline{\text{acp}}_{\pi_1}(w) \cup \overline{\text{dsf}}_{\pi_2}(w)) \quad (26)$$

With these notations, we derive the subset $\tilde{I}_0$ of I_0 as follows.

$$\tilde{I}_0 = \tilde{I}_1 \cup \tilde{I}_2 \cup \text{ins}(u) \cup \widetilde{\text{ins}}_{\pi}(w) \cup \text{ins}(v) \quad (27)$$

According to (23) and (27), we eventually derive a subset of $I(\pi_1) \cup I(\pi_2)$ as follows.

$$\tilde{I}_0 \cup I_{\text{NM}}(\overline{\pi_1}) \cup I_{\text{NM}}(\overline{\pi_2}) \subseteq I(\pi_1) \cup I(\pi_2) \quad (28)$$

The following lemma shows that the LHS of (28) derives a safe estimation of $I(\pi)$.

Lemma 6. *For any path $\pi = (u, \cdots, v)$ and for any paths $\pi_1 = (u, \cdots, w)$ and $\pi_2 = (w, \cdots, v)$ such that the path π merged by π_1 and π_2, the interference set of π is bounded by*

$$I(\pi) \subseteq \tilde{I}_0 \cup I_{\text{NM}}(\overline{\pi_1}) \cup I_{\text{NM}}(\overline{\pi_2}) \quad (29)$$

The proof of Lem. 6 is mainly based on (13), and is given in Appendix A. In the following, we use the RHS of (29) to estimate the volume of $I(\pi)$. Before going into details, we first introduce the notation of *regular paths* below.

Definition 3 (Regular path)**.** *A path $\pi = (u, \cdots, v)$ is left (and right) regular if $p(v_i) \leq p(u)$ (and $p(v_i) \leq p(v)$), $\forall v_i \in \pi$. We let a path be regular if it only contains two vertices.*

As shown in Fig. 1, path $\pi_1 = (v_2, v_5, v_6)$ is left regular; and path $\pi_2 = (v_5, v_6, v_7)$ is right regular. The regular path defined in Def. 3 is used for path merging, which has some elegant properties as described below.

Lemma 7. *For any left regular path $\pi = (w, v)$ and for any vertex $v_i \in par(w)$, $v_i \in ins(w)$ if there is an intermediate vertex $v_j \in \overline{\pi}$ such that $v_i \in \overline{ins}_{\pi}(v_j)$.*

Proof. Since $v_i \in \overline{\text{ins}}_{\pi}(v_j)$, and by (9), we have $v_i \in \text{ins}(v_j)$. According to (8), $p(v_i) \leq p(v_j)$. Moreover, since π is left regular, we have $p(v_j) \leq p(w)$. Therefore, $p(v_i) \leq p(w)$, and since $v_i \in \text{par}(w)$, we have $v_i \in \text{ins}(w)$ according to (8). □

Symmetrically, we can conclude the following proposition.

Proposition 2. *For any right regular path $\pi = (u, w)$ and for any vertex $v_i \in par(w)$, $v_i \in ins(w)$ if there is an intermediate vertex $v_j \in \overline{\pi}$ such that $v_i \in \overline{ins}_{\pi}(v_j)$.*

Based on Lem. 7 and Pro. 2, we derive the following lemma to show that the vertices that interfere with intermediate vertices of two regular paths can be safely isolated.

Lemma 8. *For any paths $\pi_1 = (u, \cdots, w)$ and $\pi_2 = (w, \cdots, v)$, $I_{\text{NM}}(\overline{\pi_1}) \cap I_{\text{NM}}(\overline{\pi_2}) = \emptyset$ if π_1 and π_2 are right and left regular, respectively.*

Proof. Suppose not. There is a non-ME vertex $v_i \in I_{\text{NM}}(\overline{\pi_1}) \cap I(\overline{\pi_2})$. Since $v_i \in I_{\text{NM}}(\overline{\pi_2})$ and by (10), there is a vertex $v_j \in \overline{\pi_2}$ such that $v_i \in \overline{\text{ins}}_{\pi_2}(v_j)$. We know that $v_i \in \text{par}(w)$. Otherwise, we consider the following two cases.

- v_i is an ancestor of w. For any vertex $v_j \in \overline{\pi_2}$, since $w \in \text{pin}_{\pi_2}(v_j)$ and by (3), we have $v_i \in \text{acp}_{\pi_2}(v_j)$. By (9), we have $v_i \notin \overline{\text{ins}}_{\pi_2}(v_j)$, and by (10), $v_i \notin I_{\text{NM}}(\overline{\pi_2})$.
- v_i is a descendant of w. For any vertex $v_j \in \overline{\pi_1}$, since $w \in \text{flw}_{\pi_1}(v_j)$ and by (4), we have $v_i \in \text{dsf}_{\pi_1}(v_j)$. By (9), we have $v_i \notin \overline{\text{ins}}_{\pi_1}(v_j)$, and by (10), $v_i \notin I_{\text{NM}}(\overline{\pi_1})$.

Both cases contradict the assumption. By now we have proved that $v_i \in \overline{\text{ins}}_{\pi_2}(v_j) \cap \text{par}(w)$. Since π_2 is left regular, we have $v_i \in \text{ins}(w)$ by Lem. 7. By (10), we know that $I(\overline{\pi_x}) \cap \text{ins}(w) = \emptyset$ $(x = 1, 2)$, and thus, $v_i \notin I_{\text{NM}}(\overline{\pi_1}) \cap I_{\text{NM}}(\overline{\pi_2})$, which contradicts the assumption. □

Based on Lem. 6 and 8, we derive the following theorem to safely estimate the volume of $I(\pi)$ by using limited information of its (regular) sub-paths.

Theorem 4. *For any path $\pi = (u, \cdots, v)$ merged by paths $\pi_1 = (u, \cdots, w)$ and $\pi_2 = (w, \cdots, v)$, the volume of $I(\pi)$ is bounded by*

$$vol(I(\pi)) \leq vol(\tilde{I}_0) + vol(I_{\text{NM}}(\overline{\pi_1})) + vol(I_{\text{NM}}(\overline{\pi_2})) \quad (30)$$

if π_1 and π_2 are right regular and left regular, respectively.

Proof. Based on (10) and (12), $I_{\text{NM}}(\overline{\pi_1}) \cup I_{\text{NM}}(\overline{\pi_2})$ is disjoint with $\tilde{I}_0$. Since π_1 is right regular and π_2 is left regular, and based on Lem. 8, $I_{\text{NM}}(\overline{\pi_1})$ and $I_{\text{NM}}(\overline{\pi_2})$ are disjoint with each other. Therefore, we can derive (30) by using (29). □

We can compute the first item of (30) by knowing the interference ME vertices of paths π_1 and π_2 and the ending points u, w and v of π_1 and π_2. The left problem is how to compute the second item and third item of (30), which is solved in the following lemma.

Lemma 9. *For any path $\pi = (u, \cdots, v)$, $vol(I_{\text{NM}}(\overline{\pi})) = vol(I(\pi)) - vol(res)$, where $res = ins(u) \cup ins(v) \cup I_{\text{ME}}(\pi)$.*

Proof. This can be derived from (13) and since the last item of (13) is disjoint with the other three items of (13). □

According to Thm. 4 and Lem. 9, the volume of path's interference set can be bounded by the following theorem.

Theorem 5. *Given a path $\pi = (u, \cdots, v)$ merged by the right regular path $\pi_1 = (u, \cdots, w)$ and the left regular path $\pi_2 = (w, \cdots, v)$, the volume of $I(\pi)$ can be bounded by*

$$vol(\tilde{I}_0) + vol(I(\pi_1)) + vol(I(\pi_2)) - vol(res_1) - vol(res_2) \quad (31)$$

where

$$res_1 = ins(u) \cup ins(w) \cup I_{\text{ME}}(\pi_1) \quad (32)$$

$$res_2 = ins(w) \cup ins(v) \cup I_{\text{ME}}(\pi_2) \quad (33)$$

Proof. According to Lem. 9, we have

$$\text{vol}(I_{\text{NM}}(\overline{\pi_1})) = \text{vol}(I(\pi_1)) - \text{vol}(\text{res}_1) \quad (34)$$

$$\text{vol}(I_{\text{NM}}(\overline{\pi_2})) = \text{vol}(I(\pi_2)) - \text{vol}(\text{res}_2) \quad (35)$$

By substituting (34) and (35) into (30), we derive (31). □

To compute (31), it is necessary to know the ending points of π_1 and π_2, the interference ME vertices of π_1 and π_2, as well as the indicator functions α_{π_1} and β_{π_2}. Thm. 5 provides an iterative method to compute the volume of a path π's interference set by using the information of π's sub-paths π_1 and π_2 that have already been computed beforehand. The following theorem implies that such an iterative method can be applied on any paths.

Theorem 6. *Any path with at least three vertices is merged by a right regular path and a left regular path.*

Proof. We consider a path $\pi = (u, \cdots, v)$ that contains at least three vertices, and let w be the vertex with the lowest priority among all vertices in $\overline{\pi}$. The path π is divided into two sub-paths $\pi_1 = (u, \cdots, w)$ and $\pi_2 = (w, \cdots, v)$, i.e., the path π is merged by π_1 and π_2. According to Def. 3, it is trivial to show that a path is regular if it contains only two vertices. Without loss of generality, we assume that paths π_1 and π_2 both contain at least three vertices. For each vertex $v_i \in \overline{\pi_1}$, since $p(v_i) \leq p(w)$, we know that π_1 is *right* regular. Moreover, for each vertex $v_i \in \overline{\pi_2}$, since $p(v_i) \leq p(w)$, we know that π_2 is *left* regular. This completes the proof. □

D. Dynamic Programming Algorithm

In this section, we design a dynamic program (DP) to estimate the WCRT bound of an ME-DAG G based on the idea of path merging as described in Sec. VI-A. A potential alternative approach is to enumerate exponential number of paths and select the one with the maximum weight to estimate the WCRT bound. Comparatively, our method is in pseudo-polynomial time if the number of ME vertices is constant. We do not explicitly merge concrete paths, but use *tuples* to abstract paths. As shown in Sec. VI-B and Sec. VI-C, the tuple not only needs to be simple, but also is required to store sufficient information to guarantee the path's feasibility and to relieve the overestimation of interference vertices. The formal definition of path's tuple is given as follows.

Definition 4 (Tuple)**.** *For any path $\pi = (u, \cdots, v)$, its tuple is defined as $\kappa = (u, v, \chi, \alpha, \beta, \psi, \varpi)$, where u and v are ending points of π; $\chi \subseteq \{\mathtt{L}, \mathtt{R}\}$ indicates whether π is left (and right) regular, i.e., if π is right regular, $\mathtt{R} \in \chi$; if π is left regular, $\mathtt{L} \in \chi$; $\alpha = \alpha_\pi$ and $\beta = \beta_\pi$ are indicator functions; $\psi = I_{\mathrm{ME}}(\pi)$ contains ME vertices of $I(\pi)$; ϖ is the bound of weight $\varpi(\pi) = len(\pi) + \frac{vol(I(\pi))}{m}$.*

Lemma 10. *There are at most $W|V|^2 2^{3|V_{\mathrm{ME}}|+2}$ tuples for an ME-DAG G, where W is the total WCET of all vertices in G.*

Proof. There are at most $|V|^2$ vertex pairs in G. There are four possible χ sets as χ is the subset of $\{\mathtt{L}, \mathtt{R}\}$. The functions α (and β) map each ME vertex to a Boolean value, and thus, there are at most $2^{|V_{\mathrm{ME}}|}$ possible indicator functions. There are at most $2^{|V_{\mathrm{ME}}|}$ ψ sets as $\psi \subseteq V_{\mathrm{ME}}$. The value of ϖ is bounded by W. In sum, there are at most $W|V|^2 2^{3|V_{\mathrm{ME}}|+2}$ tuples. □

We define tuple operations to mimic the path merging process. As any arc (and edge) can be seen as a path with length 1, we initially construct the tuple for each arc (and edge) as shown in Def. 5. Moreover, merging two paths can be mimicked by tuple computation as defined in Def. 6.

Definition 5 (Tuple Construction)**.** *For any arc (or edge) with ending points u and v, we let $\pi = (u, v)$ and construct the tuple $\kappa = \digamma(u, v)$ by the function $\digamma$ as follows.*

$$\digamma(u, v) = (u, v, \{\mathtt{L}, \mathtt{R}\}, \alpha_\pi, \beta_\pi, \emptyset, \varpi) \tag{36}$$

where $\varpi = c(u) + c(v) + \frac{vol(ins(u) \cup ins(v))}{m}$.

The computation of ϖ above is based on the fact that the length of path $\pi = (u, v)$ is $\mathrm{len}(\pi) = c(u) + c(v)$, and the interference set of π is $\mathrm{ins}(u) \cup \mathrm{ins}(v)$ since $\overline{\pi} = \emptyset$ and according to (11).

Definition 6 (Tuple Computation)**.** *Given tuples $\kappa_1 = (u, w, \chi_1, \alpha_1, \beta_1, \psi_1, \varpi_1)$ and $\kappa_2 = (w, v, \chi_2, \alpha_2, \beta_2, \psi_2, \varpi_2)$, we compute the tuple $\kappa = \kappa_1 * \kappa_2$ by the operation $*$ below.*

$$\kappa_1 * \kappa_2 = (u, v, \chi, \alpha, \beta, \psi, \zeta, \mu) \tag{37}$$

where

$$\chi = \begin{cases} \emptyset & p(u) < p(w) \wedge p(v) < p(w) \\ \{\mathtt{L}\} & p(u) \geq p(w) \wedge p(v) < p(w) \\ \{\mathtt{R}\} & p(u) < p(w) \wedge p(v) \geq p(w) \\ \{\mathtt{L}, \mathtt{R}\} & p(u) \geq p(w) \wedge p(v) \geq p(w) \end{cases} \tag{38}$$

$$\alpha(v_i) = \alpha_1(v_i) \vee \alpha_2(v_i), \quad \forall v_i \in V_{\mathrm{ME}} \tag{39}$$

$$\beta(v_i) = \beta_1(v_i) \vee \beta_2(v_i), \quad \forall v_i \in V_{\mathrm{ME}} \tag{40}$$

$$\psi = \tilde{\psi}_1 \cup \tilde{\psi}_1 \cup \widetilde{ins}_{\mathrm{ME}}(w) \tag{41}$$

$$\varpi = \varpi_1 + \varpi_2 - c(w) + \frac{vol(\tilde{I}_0) - vol(res_1) - vol(res_2)}{m} \tag{42}$$

The parameters used above are computed as follows.

$$\tilde{\psi}_1 = \psi_1 - \overline{dsf}_2(w) \tag{43}$$

$$\tilde{\psi}_2 = \psi_2 - \overline{acp}_1(w) \tag{44}$$

$$\widetilde{ins}(w) = ins(w) - (\overline{dsf}_2(w) \cup \overline{acp}_1(w)) \tag{45}$$

$$\tilde{I}_0 = \tilde{\psi}_1 \cup \tilde{\psi}_2 \cup ins(u) \cup \widetilde{ins}(w) \cup ins(v) \tag{46}$$

$$res_1 = ins(u) \cup ins(w) \cup \psi_1 \tag{47}$$

$$res_2 = ins(w) \cup ins(v) \cup \psi_2 \tag{48}$$

As shown in (38), we construct χ according to Def. 3 and by assuming that κ_1 and κ_2 correspond to a right regular path and a left regular path, respectively. According to (1) and (2), we derive the indicator functions α and β in (39) and (40). The computation of ψ in (41) is based on the fact that w becomes an intermediate vertex after the path merging, and moreover, some vertices should be removed from the original sets ψ_1 and ψ_2 according to (53) and Lem. 11. As shown in (43) to (45), the parameters $\tilde{\psi}_1$, $\tilde{\psi}_2$ and $\widetilde{ins}(w)$ coincide with (24), (25) and (26), respectively. We compute ϖ in (42) according to Thm. 5 and based on the fact that w is duplicated when two paths are merged at w (and thus, $c(w)$ must be removed from the path's length). Parameters $\tilde{I}_0$, res_1 and res_2 as shown in (46) to (48) coincide with (27), (32) and (33), respectively.

From Thm. 3 and Thm. 6, we know that not all paths need to be merged, and accordingly, the computation of Def. 6 can only be applied for some tuples that satisfy the condition defined as follows.

Definition 7. *For any tuples $\kappa_1 = (u, w_1, \chi_1, \alpha_1, \beta_1, \psi_1, \varpi_1)$ and $\kappa_2 = (w_2, v, \chi_2, \alpha_2, \beta_2, \psi_2, \varpi_2)$, we let con$(\kappa_1, \kappa_2) = 1$ if the following condition holds.*

$$\begin{aligned} & w_1 = w_2 \wedge \mathrm{R} \in \chi_1 \wedge \mathrm{L} \in \chi_2 \wedge \\ & \beta_1(v_i) = 0 \vee \alpha_2(v_i) = 0, \forall v_i \in V_{\mathrm{ME}} - \{w\} \end{aligned} \tag{49}$$

The condition in (49) indicates that the paths corresponding to κ_1 and κ_2 are right regular and left regular, respectively, and moreover, these two paths can be merged into a feasible path according to Thm. 3. Based on the above notations, the pseudo-code of our DP algorithm is given as follows.

Algorithm 1: The WCRT bound computing algorithm.

```
1  let $\varkappa = \emptyset$ and $\varkappa_{\text{new}} = \emptyset$;
2  for each arc $(u, v)$ of $A$ do
3      add $\kappa = F(u, v)$ into $\varkappa$ and $\varkappa_{\text{new}}$;
4  for each edge $(u, v)$ of $E$ do
5      add $\kappa = F(u, v)$ and $\kappa' = F(v, u)$ into $\varkappa$ and $\varkappa_{\text{new}}$;
6  while $\varkappa_{new} \neq \emptyset$ do
7      for any $\kappa_1 \in \varkappa_{new}$ do
8          for any $\kappa_2 \in \varkappa$ do
9              if con$(\kappa_1, \kappa_2) = 1$ then
10                 add $\kappa = \kappa_1 * \kappa_2$ into $\varkappa$ and $\varkappa_{\text{new}}$;
11             if con$(\kappa_2, \kappa_1) = 1$ then
12                 add $\kappa = \kappa_2 * \kappa_1$ into $\varkappa$ and $\varkappa_{\text{new}}$;
13         remove $\kappa_1$ from $\varkappa_{\text{new}}$;
14 return $R = \max\{\varpi | \kappa = (v_{src}, v_{snk}, \chi, \alpha, \beta, \psi, \varpi) \in \varkappa\}$
```

As shown in Line 1, we use $\varkappa$ to store the tuples enumerated during the computing process, and use $\varkappa_{\text{new}}$ to store the tuples newly derived at each iteration. Initially, we construct the tuple for each arc (as well as each edge) according to Def. 5 (see in Lines 2 to 5). Once some tuple κ_1 is added into $\varkappa_{\text{new}}$ at the last iteration, we find an existing tuple κ_2 in $\varkappa$ and compute the new tuple κ by using κ_1 and κ_2 if κ_1 and κ_2 (or κ_2 and κ_1) satisfy the condition (49) as shown in Lines 7 to 12. When we obtain all possible tuples that can be derived from κ_1, we delete κ_1 from $\varkappa_{\text{new}}$ as shown in Line 13. This process repeats until no tuple is newly added, and then we calculate the WCRT bound for each tuple that corresponds to a complete path according to (21), and return the maximum WCRT bound as shown in Line 14.

Theorem 7. *Alg. 1 is pseudo-polynomial time when given a constant number of ME vertices in an ME-DAG.*

Proof. This theorem can be derived by Lem. 10 and according to Def. 5 and 6 such that the tuple construction and tuple computation applied in Alg. 1 are $O(1)$, respectively. □

The correctness of Alg. 1 is shown in Thm. 8.

Theorem 8. *Alg. 1 derives a safe estimation of the WCRT bound in (21).*

Proof. For each tuple κ with the weight ϖ, we let π be the path corresponding to κ, and we have $\varpi \geq \varpi(\pi)$ according to Def. 4 and Thm. 5. According to Thm. 1, Alg. 1 can return a safe estimation of the WCRT bound of the ME-DAG G if $\varkappa$ contains the tuples of all complete paths in Π_{G}. Therefore, this theorem is proved by showing that each complete path of Π_{G} corresponds to a tuple generated by Alg. 1. Suppose not, i.e., there is a complete path π such that the tuple of π is not generated until Alg. 1 terminates. We assume that π has K vertices, and we prove that Alg. 1 generates a corresponding tuple for each feasible path with K vertices (which certainly include π). we prove this by induction as follows.

According to Lines 2 to 5, we generate tuples for each arc (and each edge). It indicates that Alg. 1 generates the tuples of all paths with 2 vertices, as we know that each feasible path with 2 vertices is an arc (or an edge) of G.

We assume that the tuples of all feasible paths with no more than k vertices are generated by Alg. 1. In the following, we aim to prove that the tuple of each feasible path π' with $k+1$ vertices is also generated by Alg. 1.

According to Thm. 6, π' can be divided into a right regular path π_1 and a left regular path π_2, i.e., the path π is merged by π_1 and π_2. From the hypothesis, Alg. 1 has generated the tuples κ_1 and κ_2 which separately correspond to π_1 and π_2. Without loss of generality, we assume that κ_2 is removed from $\varkappa_{\text{new}}$ earlier than κ_1. We consider the iteration at which we select κ_1 from $\varkappa_{\text{new}}$ and use κ_1 to generate new tuples. In this case, κ_2 must be removed at the previous iteration, and only is contained in $\varkappa$. According to Line 8, we can select κ_2 from $\varkappa$. Since π_1 and π_2 are right regular and left regular, respectively, and moreover, since π_1 and π_2 are feasible as π is feasible, we know that the condition (49) holds, i.e., con$(\kappa_1, \kappa_2) = 1$. According to Lines 9 and 10, $\kappa = \kappa_1 * \kappa_2$ is generated, and we know that κ corresponds to the path π' merged by π_1 and π_2. Therefore, the tuple of π' is generated by Alg. 1.

By now, we prove that the tuple of each feasible path with K vertices is generated by Alg. 1, which contradicts the assumption that the tuple corresponding to the complete path π with K vertices cannot be generated by Alg. 1. □

VII. Evaluations

In this section, we evaluate our method with task graph models derived from randomly generated task graphs and realistic OpenMP benchmark applications. We implement the dynamic programming algorithm proposed in Sec. VI-D by Python 3.7. The code runs on a PC with Intel Core i5-9400F CPU at 2.9GHz with 16G RAM.

We demonstrate our method improves the WCRT bound by conducting the comparison with the state-of-the-art: the WCRT bound R_0 for the DAG task with spin locks as established in Thm. 1 of [45], i.e.,

$$R_0 = \frac{\mathrm{vol}(D) + (m-1)(\mathrm{len}(D) + \sum_{l_q \in \Theta} N_q L_q)}{m} \tag{50}$$

where Θ is the set of resources. For any resource l_q, N_q represents the times that vertices of D access to l_q, and L_q is the worst-case execution time of the vertex that accesses the resource l_q. An ME edge indicates that two vertices of the ME edge access a shared resource in sequence. Thus, $\sum N_q L_q$ equals the total degrees of all ME vertices (by counting the ME edges). To fit the work in [45], we adapt our ME-DAG task by letting each pair of ME vertices connected by an ME edge share a resource. We evaluate the performance of our method in terms of the gap defined as follows.

$$\text{gap} = \frac{R_0 - R_{\text{dp}}}{R_0} \tag{51}$$

where R_{dp} is the WCRT bound computed by our DP method.

We implement several priority assignment policies that are commonly used in the existing work, and evaluate which policy performs better. The priority assignment policies exhibited in our experiments include: NVP (No Vertex Priority is defined for the ME-DAG), SCF (Smallest Co-levels First) [55], HLF (Highest Level First) [55], MISF (Most Immediate Successors First) [56], CPF (Critical Path First) [14] and LVLF (Longest Vertex Length First) [15]. We present our task experiments and results in terms of the gap defined in (51) and computation time with synthetic tasks in Sec. VII-A and realistic tasks derived from OpenMP programs in Sec. VII-B.

A. Randomly Generated Tasks

We randomly generate a DAG by the TGFF tool [57], a DAG generator developed to facilitate standardized random benchmarks for scheduling research. The graph generation algorithm of TGFF tool enables us to generate the DAG with n vertices, where the "in" and "out" degrees of each vertex are bounded by δ^- and δ^+, respectively. The WCET of each vertex is randomly set in the range of $[1, 100]$. We randomly select n_{ME} ME vertices from the DAG, and for any two ME vertices, we randomly add an undirected edge to connect them with probability p_{ME}.

We conduct experiments with different combinations of parameters as shown 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 our method significantly reduces the WCRT bound, i.e., the gap achieves more than 40% on average with different priority assignments. Our algorithm can analyze the ME-DAG with 60 vertices and with no more than 12 ME vertices within 20 seconds on average.

Fig. 4(a) shows that the gap increases when the number m of cores increases. According to (14) and (20), a larger m means a smaller proportion of the interference volume in the WCRT bound. From Fig. 4(a), we know that our method performs better when the longest path in the ME-DAG plays a major role in the WCRT bound computation. Fig. 4(b) shows that the gap decreases when the scale of the DAG grows. The reason may be as follows. As the number of ME vertices is fixed, the ratio (denoted as γ_{ME}) of ME vertices to total vertices becomes smaller when the number n of vertices increases. In this case, the impact of ME vertices on WCRT bounds is weaker, and thus, the room left for our method to improve is limited. Fig. 4(c) shows that the gap increases significantly, when the ratio γ_{ME} of ME vertices grows. As we know that both of the WCRT bounds R_0 and R_{dp} increase when γ_{ME}

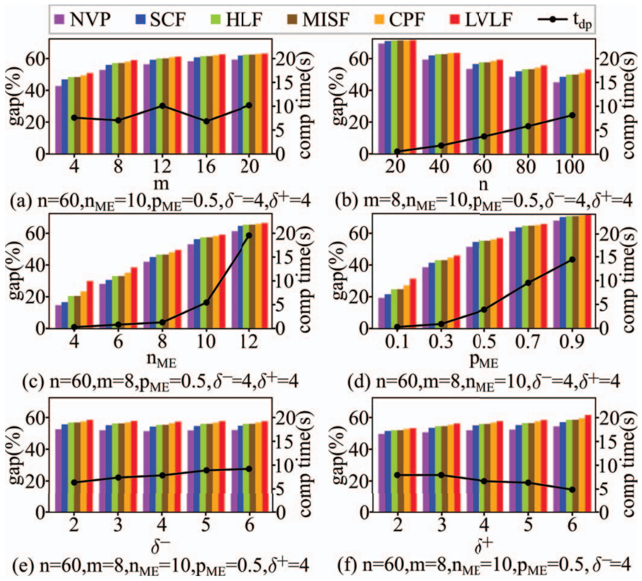

Fig. 4: Evaluation results for different configurations.

increases. The experimental result indicates that our method is more tolerant of the increase of ME vertices. Fig. 4(d) shows that the gap increases sharply, when the probability p_{ME} increases. There are more ME edges when p_{ME} becomes large. The experimental result indicates that R_0 is more sensitive to the number of ME edges. Fig. 4(e) and (f) show that the gap increases when the out-degree δ^+ increases and when the in-degree δ^- decreases. A larger δ^+ and a larger δ^- separately indicate a higher parallelism and a longer longest path of the ME-DAG. The experimental results imply that the path length usually dominates the volume of the interference set when computing the WCRT bound.

The computation time t_{dp} of our DP method linearly grows with the linear increase of the number n of vertices or the probability p_{ME} of generating ME as shown in Fig. 4(b) and (d). Fig. 4(c) shows that t_{dp} grows exponentially when the number of ME vertices increases. Besides, t_{dp} changes slightly when the number m of cores, in-degree δ^- and out-degree δ^+ increase as shown in Fig. 4(a), (e) and (f). All of these cases coincide with the complexity analysis of our algorithm.

B. Realistic OpenMP Programs

In this section, we evaluate our method with task graphs generated according to realistic OpenMP programs. OpenMP supports task parallelization since version 3.0 [58], which can be modeled as DAGs [59]. We collect 3 OpenMP programs (see Table II) that use C language and contain "`#pragma omp critical`" clauses from the BOTS benchmark suite [60], and transform them into directed graphs by ompTG tool [61]. The first five columns of Table II show the detailed information of the benchmark programs, where LoC is the number of lines in program; N_{fu} is the number of functions; N_{lp} is the number of `loop` structures; N_{if} is the number of `if-else` structures; N_{cr} is the number of `critical` clauses. The last two columns of Table II give the information of each program's directed graph model[1], where N_v is the number of vertices and N_c is the number of critical vertices.

TABLE II: Summary of BOTS programs

#name	program					task graph	
	LoC	N_{fu}	N_{lp}	N_{if}	N_{cr}	N_v	N_c
concom	35	9	2	3	1	24	1
knapsack	166	9	0	17	1	17	1
floorplan	269	13	8	22	1	24	1

We note that the selected programs have `loop` and `if-else` structures, and thus, their corresponding directed graphs contain cycles and branches, which cannot be directly handled by our method. We transform the directed (cyclic) graphs into DAGs by unrolling loops. We use N_{lb} to denote the loop bound, indicating that each loop structure cannot be traveled more than N_{lb} times. Moreover, when encountering an `if-else` structure, we randomly travel one of its branches. For any two ME vertices (that correspond to `critical` clauses), we add an ME edge between them if they are parallel with each other, i.e., there is no path traveling both of them. Evaluation results of OpenMP programs are shown in Fig. 5.

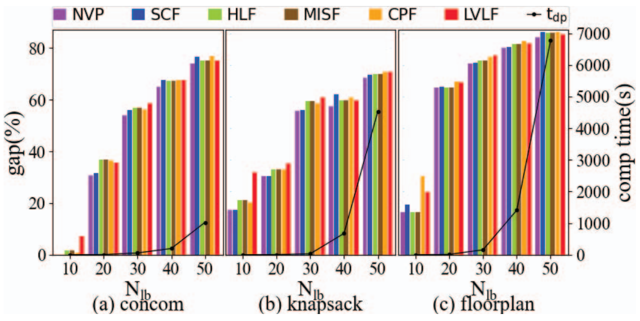

Fig. 5: Evaluation results of OpenMP programs.

Fig. 5 shows that our method significantly reduces the WCRT bound, i.e., the gap achieves more than 40% on average in different scales of task graphs. Moreover, the gap becomes larger when the loop bound N_{lb} increases. The gap achieves more than 70% on average among three programs, when the loop bound N_{lb} is 50. Our method can analyze all these three programs within a reasonable time (e.g., no more than 110 minutes). The analysis time for "concom" program is much less than that of "knapsack" and "floorplan" programs. The reason may be that the scale of "concom" program is far less than the scales of both "knapsack" and "floorplan" programs as shown in Table II. The computation time is proportional to the increment of loop bound N_{lb}. Especially, the computation time sharply increases when loop bound N_{lb} is larger than 40. When N_{lb} is 50, the computation time for analyzing "concom" program is less than 20 minutes, and the computation time for analyzing "knapsack" program (where the task graph has 68 vertices and 19 ME vertices) and floorplan program (where the task graph has 202 vertices and 25 ME vertices) are more than 60 minutes and 100 minutes, respectively. The analysis time for both "knapsack" and "floorplan" programs are within 30 minutes, when N_{lb} is 40. When N_{lb} is 10, the analysis time of "knapsack" (where the task graph has 49 vertices and 4 ME vertices) and "floorplan" (where the task graph has 42 vertices and 3 ME vertices) is bounded by a few seconds.

[1]For the sake of simplicity, we shrink the vertices connected by a dedicated path (without any branches) into one vertex.

VIII. Conclusion

In this paper, we propose the first WCRT bound for the prioritized ME-DAG task. We prove that computing the WCRT bound is strongly NP-hard, and we propose an efficient DP method for computing the WCRT bound. Experimental work shows that our method significantly reduces the WCRT bound compared with the state-of-the-art method, indicating that the prioritizing technique is a promising approach to guarantee the real-time property of the ME-DAG task. In the future, we will investigate whether there are more efficient priority assignment policies that can further improve the WCRT bound of ME-DAG task.

Appendix A: Proof of Lem. 6

In order to prove Lem. 6, we first define a subset of $\tilde{I}_0$ as $\widehat{I}_0$, i.e,. $\widehat{I}_0 \subseteq \tilde{I}_0$. It is efficient to show the correctness of (29) if the following formula holds.

$$I(\pi) \subseteq \widehat{I}_0 \cup I_{\text{NM}}(\overline{\pi_1}) \cup I_{\text{NM}}(\overline{\pi_2}) \tag{52}$$

More specifically, the subset $\widehat{I}_0$ of $\tilde{I}_0$ is denoted as follows.

$$\widehat{I}_0 = I_1 \cup I_2 \cup \text{ins}(u) \cup \overline{\text{ins}}_\pi(w) \cup \text{ins}(v) \tag{53}$$

where $I_1 = I_{\text{ME}}(\pi_1) - \text{dsf}_{\pi_2}(w)$ and $I_2 = I_{\text{ME}}(\pi_2) - \text{acp}_{\pi_1}(w)$. Based on Lem. 1, we have $\overline{\text{dsf}}_{\pi_2}(w) \subseteq \text{dsf}_{\pi_2}(w)$ for the starting point w of π_2. According to the definitions of $\tilde{I}_1$ in (24) and I_1 above, we have $I_1 \subseteq \tilde{I}_1$. Symmetrically, from (25) and the definition of I_2, we have $I_2 \subseteq \tilde{I}_2$. With the similar reasons, and from (9) and (26), we have $\overline{\text{ins}}_\pi(w) \subseteq \widetilde{\text{ins}}_\pi(w)$. According to (27) and (53), we have $\widehat{I}_0 \subseteq \tilde{I}_0$.

In the following, we prove the correctness of (52) by providing the safe estimations of $I_{\text{ME}}(\pi)$ and $I_{\text{NM}}(\overline{\pi})$ as shown in Lem. 11 and Lem. 12, respectively.

Lemma 11. $I_{\text{ME}}(\pi) \subseteq I_1 \cup I_2$.

Proof. By (11) and (12), we have

$$I_{\text{ME}}(\pi) = \bigcup_{v_i \in \pi} \overline{\text{ins}}_{\pi,\text{ME}}(v_i) \tag{54}$$

For each $v_i \in \pi_1$, by (3) and (4), we know that $\text{acp}_{\pi_1}(v_i) = \text{acp}_\pi(v_i)$ and $\text{dsf}_{\pi_1}(v_i) \cup \text{dsf}_{\pi_2}(w) \subseteq \text{dsf}_\pi(v_i)$, and by (9),

$$\overline{\text{ins}}_\pi(v_i) \subseteq \text{ins}(v_i) - (\text{acp}_{\pi_1}(v_i) \cup \text{dsf}_{\pi_1}(v_i)) - \text{dsf}_{\pi_2}(w) \tag{55}$$

and according to (9) and (12), we have

$$\overline{\text{ins}}_{\pi,\text{ME}}(v_i) \subseteq \overline{\text{ins}}_{\pi_1,\text{ME}}(v_i) - \text{dsf}_{\pi_2}(w) \tag{56}$$

With the similar reason above, we symmetrically derive the following result for each vertex $v_i \in \pi_2$.

$$\overline{\text{ins}}_{\pi,\text{ME}}(v_i) \subseteq \overline{\text{ins}}_{\pi_2,\text{ME}}(v_i) - \text{acp}_{\pi_1}(w) \tag{57}$$

By combining (56) and (57) into (54), this lemma holds. □

Lemma 12. $I_{\text{NM}}(\overline{\pi}) \subseteq I_{\text{NM}}(\overline{\pi_1}) \cup I_{\text{NM}}(\overline{\pi_2}) \cup \overline{ins}_\pi(w)$.

Proof. This lemma trivially holds since $\overline{\text{ins}}_\pi(v_i) \subseteq \overline{\text{ins}}_{\pi_1}(v_i)$, $\forall v_i \in \pi_1$ and $\overline{\text{ins}}_\pi(v_j) \subseteq \overline{\text{ins}}_{\pi_2}(v_j)$, $\forall v_j \in \pi_2$. □

By substituting formulas of Lem. 11 and Lem. 12 into (13), we eventually derive (52).

References

[1] Y. Xiang and H. Kim, "Pipelined data-parallel CPU/GPU scheduling for multi-DNN real-time inference," in *2019 IEEE Real-Time Systems Symposium (RTSS)*. Los Alamitos, CA, USA: IEEE Computer Society, dec 2019, pp. 392–405. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/RTSS46320.2019.00042

[2] M. Verucchi et al., "Latency-aware generation of single-rate DAGs from multi-rate task sets," in *RTAS*, 2020.

[3] M. Becker, D. Dasari, S. Mubeen, M. Behnam, and T. Nolte, "Synthesizing job-level dependencies for automotive multi-rate effect chains," in *2016 IEEE 22nd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. IEEE, 2016, pp. 159–169.

[4] Y. Saito, F. Sato, T. Azumi, S. Kato, and N. Nishio, "Rosch: real-time scheduling framework for ROS," in *2018 IEEE 24th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. IEEE, 2018, pp. 52–58.

[5] Y. Suzuki, T. Azumi, S. Kato *et al.*, "Hlbs: Heterogeneous laxity-based scheduling algorithm for DAG-based real-time computing," in *2016 IEEE 4th International Conference on Cyber-Physical Systems, Networks, and Applications (CPSNA)*. IEEE, 2016, pp. 83–88.

[6] J. Forget, F. Boniol, E. Grolleau, D. Lesens, and C. Pagetti, "Scheduling dependent periodic tasks without synchronization mechanisms," in *2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2010, pp. 301–310.

[7] S. E. Saidi, N. Pernet, and Y. Sorel, "Automatic parallelization of multi-rate FMI-based co-simulation on multi-core," in *TMS/DEVS 2017-Symposium on Theory of Modeling and Simulation*. ACM, 2017, pp. Article–No.

[8] A. Vincentelli, P. Giusto, C. Pinello, W. Zheng, and M. Natale, "Optimizing end-to-end latencies by adaptation of the activation events in distributed automotive systems," in *13th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS'07)*. IEEE, 2007, pp. 293–302.

[9] A. OpenMP, "OpenMP application program interface version 4.0," 2013.

[10] I. O. R. Libraryt, "https://www.openmprtl.org/," 2018.

[11] CilkPlus, "https://software.intel.com/en-us/intel-cilk-plus-support."

[12] R. Graham, "Bounds on multiprocessing timing anomalies," *SIAP*, 1969.

[13] Q. He et al., "Intra-task priority assignment in real-time scheduling of DAG tasks on multi-cores," *TPDS*, 2019.

[14] S. Zhao et al., "DAG scheduling and analysis on multiprocessor systems: Exploitation of parallelism and dependency," in *RTSS*, 2020.

[15] Q. He et al., "Response time bounds for DAG tasks with arbitrary intra-task priority assignment," in *ECRTS*, 2021.

[16] V. Bonifaci et al., "Feasibility analysis in the sporadic DAG task model," in *ECRTS*, 2013.

[17] J. Li et al., "Outstanding paper award: Analysis of global EDF for parallel tasks," in *ECRTS*, 2013.

[18] S. Baruah, "Improved multiprocessor global schedulability analysis of sporadic DAG task systems," in *ECRTS*, 2014.

[19] A. Melani et al., "Response-time analysis of conditional DAG tasks in multiprocessor systems," in *ECRTS*, 2015.

[20] J. Fonseca et al., "Improved response time analysis of sporadic DAG tasks for global FP scheduling," in *RTNS*, 2017.

[21] M. Han et al., "Bounding carry-in interference for synchronous parallel tasks under global fixed-priority scheduling," *JSA*.

[22] J. Fonseca et al., "Schedulability analysis of DAG tasks with arbitrary deadlines under global fixed-priority scheduling," *RTS*, 2019.

[23] J. Xu et al., "On the decomposition-based global EDF scheduling of parallel real-time tasks," in *RTSS*, 2016.

[24] M. Qamhieh et al., "Schedulability analysis for directed acyclic graphs on multiprocessor systems at a subtask level," in *AEiC*, 2014.

[25] A. Saifullah et al., "Parallel real-time scheduling of DAGs," *TPDS*, 2014.

[26] S. Baruah, "The federated scheduling of constrained-deadline sporadic DAG task systems," in *DATE*, 2015.

[27] J. Li et al., "Analysis of federated and global scheduling for parallel real-time tasks," in *ECRTS*, 2014.

[28] S. Baruah, "Federated scheduling of sporadic DAG task systems," in *IPDPS*, 2015.

[29] J. Li et al., "Mixed-criticality federated scheduling for parallel real-time tasks," *RTS*, 2017.

[30] S. Baruah, "The federated scheduling of systems of conditional sporadic DAG tasks," in *EMSOFT*, 2015.

[31] P. Chen et al., "Timing-anomaly free dynamic scheduling of conditional DAG tasks on multi-core systems," *TECS*, 2019.

[32] R. Vargas et al., "OpenMP and timing predictability: a possible union?" *DATE*, 2015.

[33] M. A. Serrano et al., "Timing characterization of OpenMP4 tasking model," *CASES*, 2015.

[34] J. Sun et al., "Real-time scheduling and analysis of OpenMP task systems with tied tasks," *RTSS*, 2017.

[35] ——, "Real-time scheduling and analysis of synchronous OpenMP task systems with tied tasks," *DAC*, 2019.

[36] ——, "Calculating response-time bounds for OpenMP task systems with conditional branches," *RTAS*, 2019.

[37] ——, "Real-time scheduling and analysis of OpenMP DAG tasks supporting nested parallelism," *TC*, 2020.

[38] J. Sun, N. Guan, J. Sun, X. Zhang, Y. Chi, and F. Li, "Algorithms for computing the WCRT bound of OpenMP task systems with conditional branches," *IEEE Transactions on Computers*, vol. 70, no. 1, pp. 57–71, 2020.

[39] J. Sun et al., "Calculating worst-case response time bounds for OpenMP programs with loop structures," in *RTSS*, 2021.

[40] M. A. Serrano et al., "Response-time analysis of DAG tasks supporting heterogeneous computing," *DAC*, 2018.

[41] M. Han et al., "Response time bounds for typed DAG parallel tasks on heterogeneous multi-cores," *TPDS*, 2019.

[42] S. Chang, J. Sun, Z. Liu, X. Zhao, and Q. Deng, "Response time analysis of parallel tasks on accelerator-based heterogeneous platforms," *Journal of Systems Architecture*, vol. 126, p. 102484, 2022.

[43] S. Chang, J. Sun, Z. Hao, Q. Deng, and N. Guan, "Computing exact WCRT for typed DAG tasks on heterogeneous multi-core processors," *Journal of Systems Architecture*, p. 102385, 2022.

[44] S. Dinh, J. Li, K. Agrawal, C. Gill, and C. Lu, "Blocking analysis for spin locks in real-time parallel tasks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 4, pp. 789–802, 2017.

[45] X. Jiang, N. Guan, H. Du, W. Liu, and W. Yi, "On the analysis of parallel real-time tasks with spin locks," *IEEE Transactions on Computers*, vol. 70, no. 2, pp. 199–211, 2020.

[46] H. Du, X. Jiang, T. Yang, M. Lv, and W. Yi, "Real-time scheduling and analysis of OpenMP programs with spin locks," in *2020 IEEE 26th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 2020, pp. 99–108.

[47] Z. Chen, H. Lei, M. Yang, Y. Liao, and L. Qiao, "A finer-grained blocking analysis for parallel real-time tasks with spin-locks," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2021, pp. 1177–1182.

[48] ——, "A hierarchical hybrid locking protocol for parallel real-time tasks," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 20, no. 5s, pp. 1–22, 2021.

[49] X. Jiang, N. Guan, W. Liu, and M. Yang, "Scheduling and analysis of parallel real-time tasks with semaphores," in *Proceedings of the 56th Annual Design Automation Conference 2019*, 2019, pp. 1–6.

[50] B. B. Brandenburg and J. H. Anderson, "Optimality results for multiprocessor real-time locking," in *2010 31st IEEE Real-Time Systems Symposium*. IEEE, 2010, pp. 49–60.

[51] B. B. Brandenburg, "A fully preemptive multiprocessor semaphore protocol for latency-sensitive real-time applications," in *2013 25th Euromicro Conference on Real-Time Systems*. IEEE, 2013, pp. 292–302.

[52] X. Jiang, N. Guan, Y. Tang, W. Liu, and H. Duan, "Suspension-based locking protocols for parallel real-time tasks," in *2019 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2019, pp. 274–286.

[53] Y. Wang, X. Jiang, N. Guan, Y. Tang, and W. Liu, "Locking protocols for parallel real-time tasks with semaphores under federated scheduling," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2021.

[54] R. Karp, "Reducibility among combinatorial problems," in *Complexity of computer computations*. Springer, 1972, pp. 85–103.

[55] Y. Kwok and I. Ahmad, "Static scheduling algorithms for allocating directed task graphs to multiprocessors," *ACM Comput. Surveys (CSUR)*, vol. 31, no. 4, pp. 406–471, 1999.

[56] H. Kasahara and S. Narita, "Practical multiprocessor scheduling algorithms for efficient parallel processing," *IEEE Transactions on computers*, vol. 33, no. 11, pp. 1023–1029, 1984.

[57] K. Vallerio, "Task graphs for free (TGFF v3.0)," *Official version released in April*, vol. 15, 2008.

[58] O. Board, “OpenMP application program interface version 3.0,” in *The OpenMP Forum, Tech. Rep*, 2008.
[59] Y. W. et al, “Benchmarking OpenMP programs for real-time scheduling,” in *RTCSA*, 2017.
[60] A. Duran, X. Teruel, R. Ferrer, and et al., “Barcelona OpenMP tasks suite: A set of benchmarks targeting the exploitation of task parallelism in OpenMP,” in *2009 international conference on parallel processing*. IEEE, 2009, pp. 124–131.
[61] J. Sun, T. Jin, Y. Xue, and et al., “ompTG: From OpenMP programs to task graphs,” *Journal of Systems Architecture*, vol. 126, p. 102470, 2022.