Response Time Bounds for DAG Tasks with Arbitrary Intra-Task Priority Assignment

Qingqiang He
Department of Computing, The Hong Kong Polytechnic University, Hong Kong

Mingsong Lv[1]
Department of Computing, The Hong Kong Polytechnic University, Hong Kong

Nan Guan
Department of Computing, The Hong Kong Polytechnic University, Hong Kong

Abstract

Most parallel real-time applications can be modeled as directed acyclic graph (DAG) tasks. Intra-task priority assignment can reduce the nondeterminism of runtime behavior of DAG tasks, possibly resulting in a smaller worst-case response time. However, intra-task priority assignment incurs dependencies between different parts of the graph, making it a challenging problem to compute the response time bound. Existing work on intra-task task priority assignment for DAG tasks is subject to the constraint that priority assignment must comply with the topological order of the graph, so that the response time bound can be computed in polynomial time. In this paper, we relax this constraint and propose a new method to compute response time bound of DAG tasks with *arbitrary* priority assignment. With the benefit of our new method, we present a simple but effective priority assignment policy, leading to smaller response time bounds. Comprehensive evaluation with both single-DAG systems and multi-DAG systems demonstrates that our method outperforms the state-of-the-art method with a considerable margin.

2012 ACM Subject Classification Software and its engineering → Real-time schedulability

Keywords and phrases real-time systems, response time bound, DAG tasks, intra-task priority assignment

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2021.8

1 Introduction

Multi-cores are becoming the mainstream of real-time systems for performance and energy efficiency. To utilize the power of multi-cores, software must be parallelized. Many parallel real-time applications can be modeled as directed acyclic graph (DAG) tasks. The DAG task model has gained much attention in the past few years [6,22,23,25]. In real-time community, researchers studied how to derive safe upper bounds for the response time of DAG tasks, which is a crucial characteristic for schedulability test.

When scheduling a DAG task, the execution order of eligible vertices has a large impact on the system schedulability [17,26]. A recent work [17] proposed to assign different priorities to vertices of a DAG task (i.e., intra-task priority assignment) to control the execution order of eligible vertices of a DAG task, and developed efficient algorithms to calculate safe response time bound of the DAG task in polynomial time. However, the approach in [17] is subject to the constraint that the intra-task priority must comply with the topological order of the graph (i.e., a vertex's priority cannot be higher than any of its ancestors). In general, allowing priority orders not complying with the topological order can lead to smaller response time bounds. The target of this work is to get rid of this constraint and further improve the schedulability of DAG tasks. More precisely, we develop algorithms to compute response time bounds of DAG tasks with *arbitrary* intra-task priority assignment.

[1] corresponding author

The major challenge we face is how to deal with dependencies between different parts of the graph incurred by intra-task priority assignment when it does not comply with the topological order (see Example 4). We explore insights into these dependencies and the structure of the problem, and propose solving the problem by abstracting the graph in a context-free manner. An essential observation is that vertices with lower priorities can serve to isolate dependencies. With our computing method unleashing possibilities for arbitrary priority assignment, we devise a quite simple but effective priority assignment policy, leading to a much smaller bound. Experiments with both single-DAG systems and multi-DAG systems show that our method outperforms the previous method in [17] with a considerable margin.

2 Preliminary

2.1 Task Model

The parallel real-time task is modeled as a DAG $G = (V, E)$, where V is the set of vertices and $E \subseteq V \times V$ is the set of edges. Each vertex $v_i \in V$ represents a piece of sequential workload with worst-case execution time (WCET) $c(v_i)$ (for brevity, also denoted as c_i). An edge $(v_i, v_j) \in E$ represents the precedence relation between v_i and v_j, i.e., v_j can only start execution after vertex v_i finishes its execution. A vertex with no incoming (outgoing) edges is called a *source vertex* (*sink vertex*). Without loss of generality, we assume that G has exactly one source (denoted as v_{src}), and one sink (denoted as v_{sink}). In case G has multiple source/sink vertices, a dummy source/sink vertex with zero WCET can be added to comply with our assumption.

A *path* λ starting from vertex π_0 and ending at vertex π_k ($\neq \pi_0$) is a sequence of vertices $(\pi_0, \cdots, \pi_k)$ such that $\forall i \in [0, k)$, $(\pi_i, \pi_{i+1}) \in E$, where π_0, π_k are the *start vertex* and *end vertex* of path λ respectively. We also use λ to denote the set of vertices which are in the path λ. The length of a path λ is defined as $len(\lambda) = \sum_{\pi_i \in \lambda} c(\pi_i)$. A *complete path* is a path $(\pi_0, \cdots, \pi_k)$ such that $\pi_0 = v_{src}$ and $\pi_k = v_{sink}$, i.e., a complete path is a path starting from the single source vertex and ending at the single sink vertex.

If there is an edge $(u, v) \in E$, u is a *predecessor* of v, and v is a *successor* of u. If there is a path in G from u to v, u is an *ancestor* of v and v is a *descendant* of u. We use $pred(v)$, $succ(v)$, $ance(v)$ and $desc(v)$ to denote the set of predecessors, successors, ancestors and descendants of v, respectively.

For any vertex set $V' \subset V$, we define $vol(V') = \sum_{v_i \in V'} c_i$. The volume of a DAG G denoted as $vol(G)$ is defined as $vol(V)$, i.e., $\sum_{v_i \in V} c_i$, which is the total WCET of all vertices in G. The longest path is a complete path with largest $len(\lambda)$ in G. $len(G)$ is defined as the length of the longest path.

2.2 Scheduling Model

We consider vertices of DAG G scheduled on a multi-core platform with m identical cores. The approach is divided into two phases:

- **Analysis phase.** In this phase, first, priorities are assigned to vertices (different vertices with identical priority are allowed). Formally, we assign a priority $p(v_i)$ to each vertex v_i of the DAG, and say vertex v_i has a higher priority than vertex v_j, if $p(v_i) < p(v_j)$. Second, response time bound is computed (the focus of this paper), and schedulability test is applied (not the focus of this paper).

- **Scheduling phase.** The scheduling algorithm for one DAG task with priority assignment is *prioritized list scheduling* [17], which is work-conserving and preemptive and always chooses at most m highest-priority eligible vertices for execution.

2.3 Problem Formulation

We first state some notations to present the problem. The *parallel set* of a vertex $v \in V$ is defined as $para(v) = \{u \in V \setminus \{v\} | u \notin ance(v) \wedge u \notin desc(v)\}$.

▶ **Definition 1** (Interference Set [17]). *The interference set of a vertex $v \in V$ is defined as $I(v) = \{u \in V | u \in para(v) \wedge p(u) \leq p(v)\}$. The interference set of a path λ is defined as $I(\lambda) = \bigcup_{\pi_i \in \lambda} I(\pi_i)$.*

▶ **Definition 2.** *For a path λ, $R(\lambda)$ is defined in [17] as*

$$R(\lambda) = len(\lambda) + \frac{vol(I(\lambda))}{m}$$

$R(\lambda)$ can be think of as the response time bound of this path λ. A response time bound for a DAG task was derived in [17] as stated in the following. The response time R of a DAG G with priority assignment scheduled by prioritized list scheduling on a platform with m cores is bounded by:

$$R \leq \max_{\lambda \in \Pi(G)} \{R(\lambda)\} \tag{1}$$

where $\Pi(G)$ is the set of all complete paths of G.

It can be easily checked that this bound is timing-anomaly free or sustainable [8], thus providing a safe bound if some vertices execute for less than its WCET and self-sustainable [2], thus the bound not increasing if the number of cores increases. Actually, when some vertices execute for less than its WCET, for a path λ, its inference set $I(\lambda)$ being not change, $len(\lambda)$ and $vol(I(\lambda))$ do not increase. As a result, the computed R in Equation 1 does not increase.

For a DAG task, the bound defined above varies for different priority assignments. In the computation of Equation 1, first, for a path, its interference set is computed which includes vertices that may interfere with the execution of this path. Second, the response time bound of the graph can be computed by searching all paths in this graph.

Although the bound is clearly defined in Equation 1, the computation of it can be a challenging task. Since the number of paths can be exponential in the size of the DAG, it is impractical to enumerate all the paths to compute the bound. Moreover, since the interference sets of two vertices in a path may contain the same vertex (see Example 4), which means dependencies exist among different parts of the graph, or different subproblems of the whole problem, it can be challenging to find a clear abstraction to compute the bound. We call the computation of Equation 1 as graph interference problem formulated as follows:

Graph Interference Problem. *For a DAG G with priority assignment and the number of cores m, the objective of this problem is to compute the bound defined in Equation 1.*

2.4 An Illustrating Example

An example is given to explain concepts introduced in Section 2.

▶ **Example 3.** Figure 2b shows a DAG task with a priority assignment. The number inside the circles (representing vertices) is the WCET of vertices, and the red number besides vertices is its priority. v_0, v_5 are the single source vertex and the single sink vertex respectively (both are dummy vertices with zero WCET). The longest path is $\lambda_1 = (v_0, v_1, v_4, v_5)$, and path $\lambda_2 = (v_0, v_2, v_4, v_5)$ is a complete path. We can compute $vol(G) = 18$ and $len(G) = 9$. For vertex v_4, $pred(v_4) = \{v_1, v_2\}$, $succ(v_4) = \{v_5\}$, $ance(v_4) = \{v_0, v_1, v_2\}$, $desc(v_4) = \{v_5\}$. The priority of v_1 is higher than that of v_2 with $p(v_1) = 1$, $p(v_2) = 5$. $para(v_2) = \{v_1, v_3\}$, $I(v_2) = \{v_1, v_3\}$. $len(\lambda_2) = 4$, $I(\lambda_2) = \{v_1, v_3\}$.

Suppose that the number of cores is $m = 2$. $R(\lambda_2) = len(\lambda_2) + vol(I(\lambda_2))/m = 4 + 14/2 = 11$. The graph has three complete paths. We can compute the bound defined in Equation 1 being 11 by searching all three complete paths exhaustively, and path $\lambda_2 = (v_0, v_2, v_4, v_5)$ leads to this bound.

3 Motivation

3.1 Discussion on Existing Work

In [17], a dynamic programming algorithm was proposed to compute the bound defined in Equation 1, alongside its response time analysis. But its computing method assumes the intra-task priority assignment should comply with the topological order of the DAG. In the following, we briefly introduce the computing method of [17] given in Algorithm 1, and use an example to show that its method may not produce a correct bound for a DAG with a priority assignment not complying with the topological order.

Algorithm 1 Bound Computation in [17].

```
1 $\sigma \leftarrow TopologicalOrder(G)$
2 $\lambda_{v_{src}} \leftarrow \{v_{src}\}$
3 for $v_i \in \sigma$ from $v_{src}$ to $v_{sink}$ do
4     if $v_i \neq v_{src}$ then
5         $u^* \leftarrow \arg\max_{u \in pred(v_i)} \{len(\lambda_u) + c_i + \frac{vol(I(\lambda_u) \cup I(v_i))}{m}\}$
6         $\lambda_{v_i} \leftarrow \lambda_{u^*} \cup \{v_i\}$
7     end
8 end
9 return $R(\lambda_{v_{sink}})$
```

The Algorithm first computes a topological order of the graph G (Line 1), then searches through the topological order for a path with max $R(\lambda)$. In Line 5, whenever two paths join at a vertex v_i, it search through the predecessors of v_i to find a path with maximum $R(\lambda)$ in the subgraph consisting of ancestors of v_i. This path is stored in λ_{v_i}. In Line 9, the searching reaches v_{sink}, and $R(\lambda_{v_{sink}})$ is returned as the response time bound of the whole graph.

The following example shows that Algorithm 1 may not compute a correct bound defined in Equation 1.

▶ **Example 4.** Figure 1 presents a DAG with a priority assignment. Red numbers besides vertices are priorities, and the sets above vertices are interference sets. Suppose that the number of cores is $m = 2$. We can compute the bound being 8 by searching all three complete paths exhaustively, and path $(v_0, v_1, v_4, v_5, v_6)$ leads to this bound. However, according to Algorithm 1, the computed bound is 7.5 and path $(v_0, v_2, v_4, v_5, v_6)$ leads to this bound, which is wrong and not the bound in Equation 1.

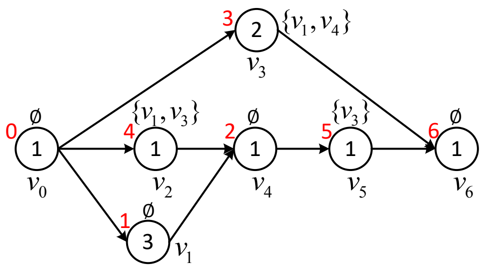
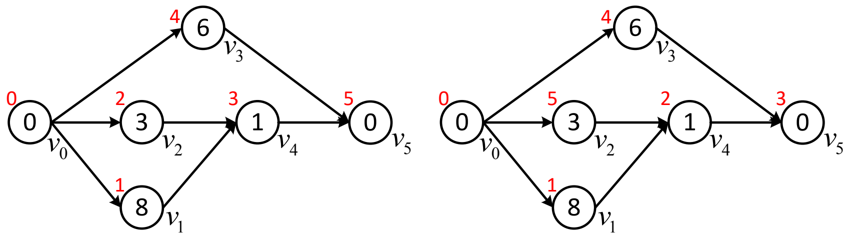


Figure 1 A counterexample of Algorithm 1.



(a) With topology constraint. **(b)** Without topology constraint.

Figure 2 A motivational example.

During Algorithm 1 for Example 4, when computing v_4 (v_4 as v_i in Line 5), it chooses path $\lambda_1 = (v_0, v_2, v_4)$ as λ_{v_4} because λ_1 suffers interference from v_3, thus λ_1 having a larger $R(\lambda)$ than that of path (v_0, v_1, v_4). When computing v_5 in Line 5, it chooses path (v_0, v_2, v_4, v_5) as λ_{v_5} because v_5 has only one predecessor v_4 and the path stored in λ_{v_4} is (v_0, v_2, v_4). However, this choice leads to a wrong result as shown in Example 4.

The reason why Algorithm 1 produces a wrong bound for the above example is that vertex v_3, being in the interference sets of both v_2 and v_5, incurs dependencies between subgraph $\{v_0, v_1, v_2, v_4\}$ and subgraph $\{v_4, v_5\}$. In fact, priority assignment, when it becomes arbitrary, may incur dependencies between different parts of the graph, which the method in [17] cannot resolve. Actually, the computing method in [17] is only valid when the priority assignment satisfies *topology constraint*, i.e., a vertex's priority is not higher than any of its ancestors.

3.2 Motivation of this Work

Two reasons motivate us to study the problem of computing response time bound for DAG tasks with arbitrary intra-task priority assignment.

First, there are situations where priorities are predetermined (e.g, by industry practitioners) before the schedulability analysis phase. For example, in OpenMP [1], practitioners can use the *priority* clause to specify a priority for a *task* construct. It is not necessary that these priority assignments satisfy topology constraint. In these cases, the computing method in [17] cannot be applied.

Second, for priority assignment determined during analysis phase as the scheduling model in Section 2.2 assumes, it is possible that much smaller response time bounds can be achieved by relaxing topology constraint and allowing arbitrary priority assignment. Example 5 is an illustration of this finding.

▶ **Example 5.** Figure 2 shows a DAG with two priority assignments (red numbers besides vertices are priorities). Figure 2a is the priority assignment according to [17] with topology constraint, while the priority assignment in Figure 2b is without this constraint (because v_2, as an ancestor of v_4, has a priority lower than that of v_4). Suppose that the number of cores is $m = 2$. In Figure 2a, we can compute the bound being 12 by searching all three complete paths exhaustively, and path $\lambda_1 = (v_0, v_3, v_5)$ leads to this bound. In Figure 2b, as shown in Example 3, the computed bound is 11, and path $\lambda_2 = (v_0, v_2, v_4, v_5)$ leads to it.

As shown in [17], assigning higher priorities to vertices in a longer path may lead to a smaller response time. However, in Figure 2a, under topology constraint without which the method in [17] is invalid, v_3 in a path with length 6 has a priority lower (i.e., $p(v_3) > p(v_2)$) than that of v_2 in a path with length 4, leading to a larger bound than that of Figure 2b.

With the two motivations, this paper focuses on solving the graph interference problem with arbitrary priority assignment, thus unleashing the possibilities for a better (possibly optimal) priority assignment policy without topology constraint.

4 Computing Response Time Bound

In this section, we solve the graph interference problem precisely through abstraction of the graph in a context-free manner, assuming priority assignment is arbitrary.

We first give an overview of our abstraction framework. A DAG with priority assignment is treated as a sentence of a formal language, and the graph interference problem is how to parse the graph to compute an abstraction of the graph (i.e., the bound in Equation 1) under a context-free grammar [18]. The abstraction of (part of) the graph is represented as tuple (Definition 6). Starting from edges represented as simple tuples, through tuples connecting into new tuples, the abstraction of the whole graph (i.e. the bound) is constructed gradually. On one hand, the context-free grammar is expressed by the connection principle (Definition 9), which functions to identify the context-free parts of the graph to isolate dependencies on other parts in the graph. On the other hand, why these parts of graph are context-free and can be computed independently without having to consider the other parts of the graph connecting to them is explained in Lemma 13, which states that these tuple connections only depend on a limited number of vertices, whose information is included in these tuples themselves, not the other parts of the graph. An illustrative example of the above concepts is shown in Figure 3 located after the computing algorithm (Algorithm 2).

Next, we will go into the details of our abstraction framework.

▶ **Definition 6** (Tuple). *For a path $\lambda = (\pi_0, \cdots, \pi_k)$, we define a tuple*

$$\langle \pi_0, \pi_k, R(\lambda) \rangle$$

where π_0, π_k are the start vertex, end vertex of path λ.

We say the tuple defined above corresponds to path λ, or path λ corresponds to this tuple. We also call π_0, π_k as the start vertex, end vertex of this tuple.

▶ **Definition 7** (Connection Vertex). *For a tuple $\langle u, v, R(\lambda) \rangle$, the connection vertex (denoted as $\kappa(u, v)$) of this tuple is defined as:*

$$\kappa(u,v) = \begin{cases} \perp & u = v_{src} \wedge v = v_{sink} \\ v & u = v_{src} \wedge v \neq v_{sink} \\ u & u \neq v_{src} \wedge v = v_{sink} \\ u & u \neq v_{src} \wedge v \neq v_{sink} \wedge p(u) \leq p(v) \\ v & u \neq v_{src} \wedge v \neq v_{sink} \wedge p(u) > p(v) \end{cases}$$

where $\perp$ means no connection vertex.

Although the equation for connection vertex seems complex, actually only two guidelines need to be kept in mind: (1) never choose a terminate vertex (i.e., v_{src}, v_{sink}); (2) always choose the vertex with a higher priority (i.e., $p(v)$ with a smaller value). According to Definition 7, except for tuples corresponding to a complete path, there is exactly one connection vertex in a tuple. For a tuple $\langle u, v, R(\lambda)\rangle$ with $u \neq v_{src} \vee v \neq v_{sink}$, we denote the priority of the connection vertex of this tuple as $p(\kappa(u, v))$(for brevity, also denoted as $p(u, v)$).

Two paths λ_0, λ_1 can be connected into a new path λ, if the end vertex of λ_0 is the same as the start vertex of λ_1, denoted as $\lambda = \lambda_0 \cup \lambda_1$. Similarly, two tuples can be connected into a new tuple, if the end vertex of the first tuple is the same as the start vertex of the second tuple.

▶ **Example 8.** For the graph in Figure 2b, path $\lambda_0 = (v_2, v_4)$ corresponds to a tuple $\alpha_0 = \langle v_2, v_4, R(\lambda_0)\rangle$, and path $\lambda_1 = (v_4, v_5)$ corresponds to a tuple $\alpha_1 = \langle v_4, v_5, R(\lambda_1)\rangle$. $\kappa(v_2, v_4) = v_4$, $\kappa(v_4, v_5) = v_4$. Since v_4 is the end vertex of α_0 and the start vertex of α_1, tuple α_0 and α_1 can be connected into a new tuple $\alpha = \langle v_2, v_5, R(\lambda)\rangle$, where $\lambda = \lambda_0 \cup \lambda_1 = (v_2, v_4, v_5)$.

With respect to tuple connection, we introduce the following connection principle.

▶ **Definition 9** (Connection Principle). *For two tuples $\langle u, v, R(\lambda_0)\rangle$, $\langle v, w, R(\lambda_1)\rangle$, if vertex v is the connection vertex of these two tuples, then they can be connected into a new tuple.*

We call these two tuples are connected under the connection principle denoted by

$$\langle u, v, R(\lambda_0)\rangle + \langle v, w, R(\lambda_1)\rangle \rightsquigarrow \langle u, w, R(\lambda)\rangle \qquad (2)$$

where $\lambda = \lambda_0 \cup \lambda_1$.

Note that since $R(\lambda_0)$ is just a value, the detailed information of a path is not stored in a tuple, which means $R(\lambda)$ cannot be computed by the above equation. Later in Lemma 14, the formula of computing the resulting tuple will be given.

For an edge $(u, v) \in E$, there is a path $\lambda = (u, v)$ and a tuple $\langle u, v, R(\lambda)\rangle$. A *simple tuple* is defined as a tuple $\langle u, v, R(\lambda)\rangle$, where λ is actually an edge.

▶ **Definition 10** (Tuple under Connection Principle). *The definition is given by the following two recursive rules:*

- *A simple tuple is under connection principle;*
- *A tuple, which is computed according to Equation 2 by tuples under connection principle, is also under connection principle.*

▶ **Example 11.** In Example 8, since α_0, α_1 are simple tuples, these two tuples are under connection principle. Since v_4 is the connection vertex of α_0, α_1, this tuple connection is also under connection principle, thus the resulting tuple α being under connection principle.

For the rest of this paper, unless explicitly specified, all tuples and tuple connections are under connection principle.

▶ **Lemma 12** (Connection Property). *For a tuple $\langle u, w, R(\lambda)\rangle$ with $u \neq v_{src} \vee w \neq v_{sink}$ under connection principle, the following holds:*

$$\forall \pi_i \in \lambda \setminus \{u, w\}, p(\pi_i) \leq p(u, w)$$

Proof. We prove it by induction.
Base case: For a simple tuple, since $\lambda \setminus \{u, v\} = \varnothing$, the lemma holds trivially.
Induction step: For a tuple $\alpha = \langle u, w, R(\lambda)\rangle$ that is not a simple tuple. Since α is under

connection principle, by Definition 10, there exist two tuples $\alpha_0 = \langle u, v, R(\lambda_0)\rangle$, $\alpha_1 = \langle v, w, R(\lambda_1)\rangle$, both being under connection principle, satisfying $\alpha_0 + \alpha_1 \rightsquigarrow \alpha$. Since α_0, α_1 are under connection principle, by induction hypothesis, both α_0 and α_1 satisfy connection property.

It is clear that $v \neq v_{src} \wedge v \neq v_{sink}$, and we already have $\kappa(u, v) = v$, $\kappa(v, w) = v$, which means $p(u, v) = p(v)$, $p(v, w) = p(v)$.

There are three cases: (1) $u = v_{src} \wedge w \neq v_{sink}$; (2) $u \neq v_{src} \wedge w = v_{sink}$; (3) $u \neq v_{src} \wedge w \neq v_{sink}$. For the first case, according to Definition 7, we have $\kappa(u, w) = w$, which means $p(u, w) = p(w)$. Since $\kappa(v, w) = v$, we have $p(v) \leq p(w)$, which means $p(v) \leq p(u, w)$. For the second case, according to similar reasons, we have $p(v) \leq p(u, w)$. For the third case, since $\kappa(u, v) = v$, $\kappa(v, w) = v$, according to Definition 7, we have $p(v) \leq p(u)$ and $p(v) \leq p(w)$, which means $p(v) \leq p(u, w)$. In summary, for the three cases, $p(v) \leq p(u, w)$.

Since α_0 satisfies connection property, we have

$$\forall \pi_i \in \lambda_0 \setminus \{u, v\}, p(\pi_i) \leq p(u, v)$$

Note that $p(u, v) = p(v)$ and $p(v) \leq p(u, w)$, we have

$$\forall \pi_i \in \lambda_0 \setminus \{u, v\}, p(\pi_i) \leq p(u, w)$$

For the same reason, we have

$$\forall \pi_i \in \lambda_1 \setminus \{v, w\}, p(\pi_i) \leq p(u, w)$$

Note that $\lambda = \lambda_0 \cup \lambda_1$, we have

$$\forall \pi_i \in \lambda \setminus \{u, w\}, p(\pi_i) \leq p(u, w)$$

which means that tuple α satisfies connection property. The lemma follows. ◀

Under the principle, a key observation for the connection of tuples is that the computation does not depend on the whole path necessarily, actually only depends on a limited number of vertices on this path, as stated in the following lemma.

▶ **Lemma 13.** *If two tuples $\langle u, v, R(\lambda_0)\rangle$, $\langle v, w, R(\lambda_1)\rangle$ under connection principle are connected into a new tuple $\langle u, w, R(\lambda)\rangle$ according to Equation 2, then*

$$I(\lambda_0) \cap I(\lambda_1) = I(v) \cup (I(u) \cap I(w)) \tag{3}$$

Proof. We use LHS and RHS to represent the left-hand side and right-hand side of Equation 3. Next, we shall prove the lemma by showing that both RHS⊆LHS and LHS⊆RHS hold.

(1) RHS⊆LHS. Since $v \in \lambda_0$ and $v \in \lambda_1$, we have

$$I(v) \subseteq I(\lambda_0) \cap I(\lambda_1)$$

Since $u \in \lambda_0$ and $w \in \lambda_1$, we have

$$I(u) \cap I(w) \subseteq I(\lambda_0) \cap I(\lambda_1)$$

In summary, we reach that

$$I(v) \cup (I(u) \cap I(w)) \subseteq I(\lambda_0) \cap I(\lambda_1)$$

(2) LHS⊆RHS. Obviously, $\kappa(u,v)=v$, $\kappa(v,w)=v$, $p(u,v)=p(v)$, $p(v,w)=p(v)$. According to Definition 1, $\forall x \in I(\lambda_0)\cap I(\lambda_1)$, $\exists \pi_i \in \lambda_0$ and $\exists \pi_j \in \lambda_1$, such that vertex $x \in I(\pi_i)$ and $x \in I(\pi_j)$, which means $p(x) \le p(\pi_i)$ and $p(x) \le p(\pi_j)$.
Next, we shall prove that $x \in para(v)$. If $x \in ance(v)$, since $v \in ance(\pi_j) \vee v = \pi_j$, then $x \in ance(\pi_j)$, which contradicts $x \in I(\pi_j)$. We have $x \notin ance(v)$. By similar reasons, $x \notin desc(v)$. In summary, $x \in para(v)$.
In the following, we prove by exhaustion. There are three cases:
(a) $\pi_i \neq u$. Since $\kappa(u,v)=v$, by Lemma 12, we have $p(\pi_i) \le p(v)$. Note that $p(x) \le p(\pi_i)$, so $p(x) \le p(v)$. Together with $x \in para(v)$, we have $x \in I(v)$, which means $x \in I(v) \cup (I(u)\cap I(w))$.
(b) $\pi_j \neq w$. For similar reasons to the first case, $x \in I(v)$, so $x \in I(v)\cup(I(u)\cap I(w))$.
(c) $\pi_i = u \wedge \pi_j = w$. Since $x \in I(\pi_i)$ and $x \in I(\pi_j)$, we have $x \in I(u)$ and $x \in I(w)$, which means $x \in I(u)\cap I(w)$. We reach that $x \in I(v)\cup(I(u)\cap I(w))$.
Summarizing these three cases, we have $\forall x \in I(\lambda_0)\cap I(\lambda_1)$, $x \in I(v)\cup(I(u)\cap I(w))$.

In conclusion, the lemma follows. ◀

▶ **Lemma 14.** *If two tuples $\langle u,v,R(\lambda_0)\rangle$, $\langle v,w,R(\lambda_1)\rangle$ under connection principle are connected into a new tuple $\langle u,w,R(\lambda)\rangle$ according to Equation 2, then*

$$R(\lambda) = R(\lambda_0) + R(\lambda_1) - c(v) - \frac{vol(I(v)\cup(I(u)\cap I(w)))}{m} \tag{4}$$

Proof. By Definition 2,

$$R(\lambda) = len(\lambda) + \frac{vol(I(\lambda))}{m}$$

Since $\lambda = \lambda_0 \cup \lambda_1$, we have $len(\lambda) = len(\lambda_0) + len(\lambda_1) - c(v)$ and $I(\lambda) = I(\lambda_0)\cup I(\lambda_1)$. Further,

$$R(\lambda) = R(\lambda_0) + R(\lambda_1) - c(v) - \frac{vol(I(\lambda_0)) + vol(I(\lambda_1)) - vol(I(\lambda))}{m}$$

Since $I(\lambda) = I(\lambda_0)\cup I(\lambda_1)$, we have

$$vol(I(\lambda_0)) + vol(I(\lambda_1)) - vol(I(\lambda)) = vol(I(\lambda_0)\cap I(\lambda_1))$$

By Lemma 13, we have

$$vol(I(\lambda_0)) + vol(I(\lambda_1)) - vol(I(\lambda)) = vol(I(v)\cup(I(u)\cap I(w)))$$

Together, we reach the conclusion. ◀

The meaning of the above lemma is twofold. First, it gives a formula to compute the connection of tuples iteratively. Second, it implies that for two paths with the same start and end vertex, the path with a smaller $R(\lambda)$ cannot result in a path with a larger $R(\lambda)$, as stated in Lemma 16 formally.

▶ **Definition 15** (Domination)**.** *Given two paths with the same start and end vertex, there are two tuples $\langle u,v,R(\lambda)\rangle$, $\langle u,v,R(\lambda')\rangle$. We say $\langle u,v,R(\lambda)\rangle$ dominates $\langle u,v,R(\lambda')\rangle$, denoted by*

$$\langle u,v,R(\lambda)\rangle \succcurlyeq \langle u,v,R(\lambda')\rangle$$

if and only if $R(\lambda) \ge R(\lambda')$.

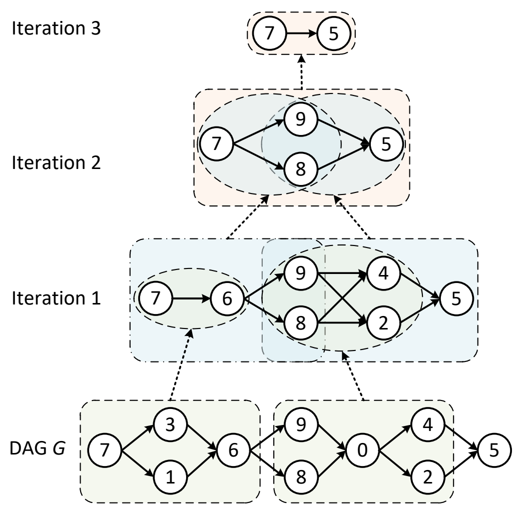


Figure 3 An example illustrating Algorithm 2.

▶ **Lemma 16.** *Under connection principle, given*

$$\langle u, v, R(\lambda_0)\rangle + \langle v, w, R(\lambda_1)\rangle \rightsquigarrow \langle u, w, R(\lambda)\rangle$$

and

$$\langle u, v, R(\lambda_0')\rangle + \langle v, w, R(\lambda_1)\rangle \rightsquigarrow \langle u, w, R(\lambda')\rangle$$

If

$$\langle u, v, R(\lambda_0)\rangle \succcurlyeq \langle u, v, R(\lambda_0')\rangle$$

then

$$\langle u, w, R(\lambda)\rangle \succcurlyeq \langle u, w, R(\lambda')\rangle$$

Proof. According to Definition 15, we have $R(\lambda_0) \geq R(\lambda_0')$

$$\begin{aligned}
&\Rightarrow R(\lambda_0) + R(\lambda_1) - c(v) - \frac{vol(I(v) \cup (I(u) \cap I(w)))}{m} \\
&\qquad \geq R(\lambda_0') + R(\lambda_1) - c(v) - \frac{vol(I(v) \cup (I(u) \cap I(w)))}{m} \\
&\Rightarrow R(\lambda) \geq R(\lambda') \\
&\Rightarrow \langle u, w, R(\lambda)\rangle \succcurlyeq \langle u, w, R(\lambda')\rangle
\end{aligned}$$

The conclusion is reached. ◀

The above lemma means when computing Equation 1, tuples with a smaller R can be discarded safely, since in future computation they cannot result in a tuple with a larger R. We summarize the above discussion into Algorithm 2 to compute the response time bound of a DAG as defined in Equation 1.

Figure 3 provides an illustrative example of Algorithm 2 to show the steps of abstraction of the graph. The graph is at the bottom of Figure 3 where the number inside each vertex is its priority (The WCET of each vertex is irrelevant to the example, and we omit such

Algorithm 2 Computing graph interference problem.

```
Input  : DAG $G = (V, E)$; every vertex $v_i \in V$ is with its WCET $c_i$ and its priority
         $p(v_i)$; the number of cores $m$
Output : the response time bound defined in Equation 1
1  $TS \leftarrow \{\langle u, v, R(\lambda)\rangle \mid \lambda = (u, v) \in E\}$
2  repeat
3      for each $(\alpha_i, \alpha_j) \in TS \times TS$ do
4          if $\alpha_i$, $\alpha_j$ can be connected by Definition 9 then
5              $\alpha \leftarrow \alpha_i + \alpha_j$ by Equation 2, 4
6              if $\exists \beta \in TS$ such that $\beta \succcurlyeq \alpha$ then
7                  continue
8              else if $\exists \beta \in TS$ such that $\alpha \succcurlyeq \beta$ then
9                  $TS \leftarrow (TS \setminus \{\beta\}) \cup \{\alpha\}$
10             else
11                 $TS \leftarrow TS \cup \{\alpha\}$
12             end
13         end
14     end
15 until nothing changes in TS
16 return $R$ such that $\langle v_{src}, v_{sink}, R\rangle \in TS$
```

information in the figure). In the example, since each vertex has a unique priority, we also use the priority to identify the vertex. After the first iteration of the loop in Line 2-15, with tuple connections under the connection principle, the original graph is transformed as illustrated in the figure. After three iterations of the loop, the tuple with start vertex and end vertex being v_{src} and v_{sink} respectively appears, which means the bound in Equation 1 is computed. It is easy to observe that all tuple connections in Figure 3 follow the connection principle in Definition 9. The context-free parts of the graph are indicated as colored rectangles in Figure 3. The relations between the context-free parts of the graph and their abstractions during each iteration are indicated as dashed arrows, which form an abstract syntax tree of the original graph.

Note that for clear and concise presentation of the principle behind Algorithm 2, the illustration of Figure 3 is not exactly the same as Algorithm 2. In Algorithm 2, during one iteration, plenty of tuples are connected, much of them being redundant and having been connected in the previous iterations. Since these redundant tuple connections are irrelevant to the correctness and theoretical computational complexity of Algorithm 2, we do not show these connections in Figure 3.

Next, we introduce the concept of *abstract path*, which is useful for proving the correctness of Algorithm 2.

▶ **Definition 17** (Abstract Path). *An abstract path $\lambda = (\pi_0, \cdots, \pi_k)$ $(k > 0)$ is a sequence of vertices such that $\forall i \in [0, k)$, there is a path λ_i with start and end vertex being π_i, π_{i+1} respectively. Further, for an abstract path $\lambda = (\pi_0, \cdots, \pi_k)$, we define $TS(\lambda) = \{\langle \pi_i, \pi_{i+1}, R(\lambda_i)\rangle \mid i \in [0, k)\}$ as the set of tuples corresponding to λ_i.*

Note that an abstract path is always an abstraction of a concrete path(i.e., $\cup_{i \in [0,k)} \lambda_i$ by using the above notation). To compute $TS(\lambda)$, the concrete path behind the abstract path λ should be given. Since $TS(\lambda)$ only serves as an intermediate concept when proving the correctness of Algorithm 2, for brevity, we will omit this concrete path. According to the definition of abstract path, a path is an abstract path, while an abstract path is not necessarily a path.

▶ **Lemma 18** (Connection Lemma). *For an arbitrary abstract path $\lambda = (\pi_0, \cdots, \pi_k)$ with $k > 1 \wedge \pi_0 = v_{src} \wedge \pi_k = v_{sink}$, $\exists \alpha, \beta \in TS(\lambda)$, such that tuple α, β can be connected under connection principle.*

Proof. Since $k > 1$, there are at least two tuples in $TS(\lambda)$. In the following, we prove this by contradiction. Assume that there are not two tuples which can be connected under connection principle, next we will consider the priority assignment along this abstract path starting from $\pi_0 = v_{src}$. There are four cases.

(1) $p(\pi_0) > p(\pi_1) \wedge p(\pi_1) \leq p(\pi_2)$. Since $\kappa(\pi_0, \pi_1) = \kappa(\pi_1, \pi_2) = \pi_1$, then $\langle \pi_0, \pi_1, R(\lambda_0) \rangle$, $\langle \pi_1, \pi_2, R(\lambda_1) \rangle$ can be connected under connection principle, which is a contradiction. Note that the reasoning in this case does not rely on $\pi_0 = v_{src}$, which means if π_0 is an arbitrary vertex in λ, the above reasoning is valid.

(2) $p(\pi_0) > p(\pi_1) \wedge p(\pi_1) > p(\pi_2)$. The reasoning in this case does not rely on $\pi_0 = v_{src}$ either.

(2a) If $\pi_2 = v_{sink}$, then $\kappa(\pi_0, \pi_1) = \kappa(\pi_1, \pi_2) = \pi_1$, which means a contradiction.

(2b) If $\pi_2 \neq v_{sink}$, consider $p(\pi_3)$. If $p(\pi_2) \leq p(\pi_3)$, we have the pattern $p(\pi_1) > p(\pi_2) \wedge p(\pi_2) \leq p(\pi_3)$. Since case (1) does not rely on $\pi_0 = v_{src}$, this pattern is the same as case (1) and finally leads to a contradiction. Actually in this case, to ensure two tuples cannot be connected under connection principle as indicated in the assumption, by the above reasoning, the priority of the next vertex π_{i+1} must be higher than that of the previous vertex π_i, formally $p(\pi_i) > p(\pi_{i+1})$. Considering all vertices along λ, finally we reach $\pi_k = v_{sink}$, and we have $p(\pi_{k-2}) > p(\pi_{k-1}) \wedge p(\pi_{k-1}) > p(v_{sink})$, which is the case in (2a) and finally leads to a contradiction.

(3) $p(\pi_0) \leq p(\pi_1) \wedge p(\pi_1) \leq p(\pi_2)$. Since $\pi_0 = v_{src}$, then $\kappa(\pi_0, \pi_1) = \kappa(\pi_1, \pi_2) = \pi_1$, which means a contradiction.

(4) $p(\pi_0) \leq p(\pi_1) \wedge p(\pi_1) > p(\pi_2)$.

(4a) If $\pi_2 = v_{sink}$, since $\pi_0 = v_{src}$, then $\kappa(\pi_0, \pi_1) = \kappa(\pi_1, \pi_2) = \pi_1$, which means a contradiction.

(4b) If $\pi_2 \neq v_{sink}$, consider $p(\pi_3)$. If $p(\pi_2) \leq p(\pi_3)$, we have $p(\pi_1) > p(\pi_2) \wedge p(\pi_2) \leq p(\pi_3)$, which is the case in (1) and finally leads to a contradiction. If $p(\pi_2) > p(\pi_3)$, we have $p(\pi_1) > p(\pi_2) \wedge p(\pi_2) > p(\pi_3)$, which is the case in (2) and finally leads to a contradiction.

In summary, all cases lead to a contradiction, therefore the initial assumption must be false. We reach the conclusion. ◀

For an abstract path $\lambda = (\pi_0, \cdots, \pi_i, \pi_{i+1}, \pi_{i+2}, \cdots, \pi_k)$, tuple $\alpha = \langle \pi_i, \pi_{i+1}, R(\lambda_i) \rangle$, $\beta = \langle \pi_{i+1}, \pi_{i+2}, R(\lambda_{i+1}) \rangle \in TS(\lambda)$, suppose α, β can be connected under connection principle, this connection will result in a new abstract path $\lambda' = (\pi_0, \cdots, \pi_i, \pi_{i+2}, \cdots, \pi_k)$ and a new $TS(\lambda')$ with $|TS(\lambda')| = |TS(\lambda)| - 1$.

▶ **Example 19.** For the graph in Figure 2b, for path $\lambda = (v_0, v_2, v_4, v_5)$ (which is also an abstract path by definition), $TS(\lambda) = \{\alpha_0 = (v_0, v_2, R_0), \alpha_1 = (v_2, v_4, R_1), \alpha_2 = (v_4, v_5, R_2)\}$. From Example 8, by Lemma 18, we know in $TS(\lambda)$, there exist α_1, α_2 that can be connected under connection principle. This tuple connection results in a new abstract path $\lambda' = \{v_0, v_2, v_5\}$ and $TS(\lambda') = \{(v_0, v_2, R'_0), (v_2, v_5, R'_1)\}$. It is obvious that $|TS(\lambda')| = |TS(\lambda)| - 1$.

Concerning the correctness and complexity of Algorithm 2, two aspects need to be considered. On one hand, according to connection principle in Definition 9, if two tuples are to be connected, first the end vertex of a tuple should be the start vertex of another tuple; second the connection vertex of these two tuples should be the same. Since we do not make any assumption about priority assignment, which is different from [17] where priority assignment should comply with topological order, is it possible that after the loop in Line 2-15 finishes, there is not a tuple $\langle u, v, R \rangle$ with $u = v_{src} \wedge v = v_{sink}$ in TS as required by Line 16? On the other hand, since the loop in Line 2-15 does not exit until nothing changes in TS, how can we guarantee that the loop will be completed within a reasonable number of iterations? By using Lemma 18, we address these concerns in the following theorem.

▶ **Theorem 20.** *The return value of Algorithm 2 equals the bound in Equation 1.*

Proof. We define $TS_{max} = \{\langle u, v, R \rangle \in TS | u = v_{src} \wedge v = v_{sink}\}$. The theorem is proved by the following three steps:

(1) In Line 16 of Algorithm 2, $|TS_{max}| = 1$.

a. $|TS_{max}| \geq 1$. For $\forall \lambda \in \Pi(G)$, if only two vertices in this path (i.e., $|\lambda| = 2$), then these two vertices must be v_{src} and v_{sink}, which means $\alpha = \langle v_{src}, v_{sink}, R(\lambda) \rangle$ is added to TS in Line 1. Although α might be removed from TS in Line 9, this only happens when α is dominated by a new tuple with start vertex and end vertex being v_{src}, v_{sink} respectively. If $|\lambda| > 2$, according to Lemma 18, after one iteration of the loop in Line 2-15, at least two tuples in $TS(\lambda)$ must be connected under connection principle, resulting in a new abstract path λ' with $|TS(\lambda')| \leq |TS(\lambda)| - 1$. This fact means after at most $|V|$ iterations of the loop in Line 2-15, a tuple with start vertex being v_{src} and end vertex being v_{sink} corresponding to λ (or a tuple dominating the tuple corresponding to λ) will be eventually computed. Since $\Pi(G) \neq \varnothing$, $|TS_{max}| \geq 1$.

b. $|TS_{max}| \leq 1$. According to Definition 15, It is sufficient to show that in any step of the algorithm, $\nexists \alpha, \beta \in TS$, such that $\alpha \succcurlyeq \beta \vee \beta \succcurlyeq \alpha$. First, obviously in Line 1, the statement is true. Second, during the loop in Line 2-15, according to the conditions in Line 6, 8, it is evident that Line 9, 11 will not lead to domination between tuples in TS.

We denote the only tuple in TS_{max} as α_{max}.

(2) There is a complete path $\lambda_{max} \in \Pi(G)$ corresponding to α_{max}. It is sufficient to show that $\forall \alpha \in TS$, there is a path corresponding to α. First, in Line 1, all simple tuples are added to TS. It is evident that a simple tuple corresponds to an edge, which is also a path. Second, in Line 5, according to Equation 2, for every tuple connection, two paths are connected into a new path corresponding to the newly computed tuple.

(3) There is no complete path $\lambda \in \Pi(G)$ such that $R(\lambda) > R(\lambda_{max})$. We prove this statement by showing that $\forall \lambda \in \Pi(G)$, $\langle v_{src}, v_{sink}, R(\lambda_{max}) \rangle \succcurlyeq \langle v_{src}, v_{sink}, R(\lambda) \rangle$. First, in Line 1, all simple tuples are added to TS. We have $\forall \lambda \in \Pi(G)$, $TS(\lambda) \subseteq TS$, which means all complete paths have a representation (i.e. $TS(\lambda)$) in TS. Second, it is obvious that all tuples in TS are under connection principle. According to Definition 15, together with the discussion in 1(a), $\forall \lambda \in \Pi(G)$, after at most $|V|$ iterations of the loop in Line 2-15, $\langle v_{src}, v_{sink}, R(\lambda) \rangle$ will either be computed in TS (in this case, $R(\lambda) = R(\lambda_{max})$), or be dominated (in this case, $R(\lambda) \leq R(\lambda_{max})$). We reach the conclusion.

Summarizing above three steps, the theorem is proved. ◀

▶ **Theorem 21.** *The time complexity of Algorithm 2 is polynomially bounded in $|V|$.*

In the above proof, from (1a), the number of iterations of the loop in Line 2-15 will not exceed $|V|$; from (1b), since no tuple domination in TS, the number of tuples in TS will not exceed $|V|^2$. In Line 3, there is $TS \times TS$. So the number of iterations of the loop in Line 3-14 will not exceed $|V|^4$. The time complexity of Algorithm 2 is $O(|V|^5)$. However, in Line 4, when tuple α_i is determined, the end vertex of α_i being determined, actually only $|V|$ number of α_j in TS need to be examined. Consequently, the algorithm can be easily implemented with time complexity being $O(|V|^4)$.

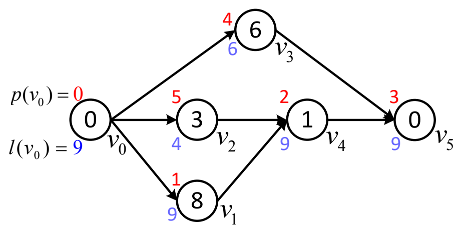


Figure 4 An example illustrating priority assignment.

5 Priority Assignment

To illustrate the performance of arbitrary priority assignment supported by our computing method in Section 4, we devise a priority assignment policy without topology constraint, leading to a much smaller bound as shown in Section 7. The guideline is to assign higher priorities to vertices in a longer path as indicated in Section 3.2. We first introduce a concept to identify vertices in a longer path.

▶ **Definition 22** (Vertex Length). *The vertex length of v (denoted as $l(v)$) is defined as*

$$l(v) = max\{len(\lambda) \mid \lambda \in \Pi(G) \wedge v \in \lambda\} \tag{5}$$

Intuitively, vertex length is the longest path length among all the paths which go through the vertex. Vertex length can be computed by a straightforward dynamic programming in polynomial time with respect to the size of the graph (Algorithm 3 in [17]). The fixed priorities of vertices are assigned based on vertex length as follows.

- A vertex with a larger length is assigned a higher priority, formally $p(v_i) < p(v_j)$ if $l(v_i) > l(v_j)$;
- If two vertices have the same length, the vertex with a smaller index in the graph is given a higher priority, formally $p(v_i) < p(v_j)$ if $l(v_i) = l(v_j) \wedge i < j$.

The vertex index does not necessarily follow topological order. Ties can be broken by any other rules. The second rule is introduced to make the priority assignment policy determinate and make the evaluation in Section 7 reproducible.

▶ **Example 23.** For the graph in Figure 2b, the length (the blue number below vertices) of each vertex is labelled in Figure 4, and a priority assignment (the red number above vertices) according to the proposed policy is also illustrated.

In Figure 4, the priority of v_4 is higher (smaller priority value means higher priority) than the priority of v_2 (note that v_2 is an ancestor of v_4), which does not comply with topology constraint (i.e., a vertex's priority is not higher than any of its ancestors). In consequence, the proposed priority policy is without topology constraint. As an illustrative specific example, the proposed priority assignment policy indeed relies on some topological characteristics (e.g., the vertex length in Definition 22). However, the claim and the main contribution of this paper is that our computing method is valid for arbitrary priority assignment, thus not limited to topology constraint required in [17].

We note that the proposed priority assignment policy does not strictly dominate the policy in [17], i.e., not always producing a bound smaller than the bound of the policy in [17]. However, the proposed policy is much simpler, and leads to a smaller bound in general cases (actually, only in very rare cases with a larger bound, see Section 7.1).

6 Extension to Multi-DAG Systems

The proposed method for computing response time bound for single DAG task can also be applied to multi-DAG sporadic systems. The approach of utilizing intra-task priority to improve the schedulability of multi-DAG systems was introduced in [17]. Although the intra-task priority studied in this work is without topology constraint, thus the computing method and priority assignment policy being completely different, the response time analysis behind the bound in Equation 1 is still the same. Therefore, the approach of [17] can be used directly to extend our method to multi-DAG systems. We briefly introduce it to help understanding the experiments in Section 7.2.

The scheduling algorithm for multi-DAG systems is *global prioritized list scheduling* [17], which has two levels: task level and vertex level. In task level, a priority policy, e.g., early deadline first (EDF) and rate monotonic (RM), is employed to determine the highest-priority ready DAG task; in vertex level, prioritized list scheduling is used. After priorities between tasks and further between vertices are determined, the scheduling behavior is unchanged, which is global, work-conserving, preemptive, and always chooses at most m (the core number) highest-priority eligible vertices for execution.

▶ **Theorem 24** ([17]). *For a multi-DAG sporadic system with constrained deadlines scheduled by global prioritized list scheduling on a platform with m cores, a bound R_j on the response time of a task τ_j can be derived by the fixed-point iteration of the following equation, starting with $R_j = len(G_j)$:*

$$R_j = \max_{\lambda \in \Pi(G_j)} \{len(\lambda) + \frac{vol(I(\lambda))}{m}\} + \frac{\sum_{i \neq j} I_j^i(R_j)}{m} \tag{6}$$

where $I_j^i(R_j)$ is the upper bound of the interference of task τ_i to τ_j during an interval of length R_j.

In Equation 6, $I_j^i(R_j)$ is related to task level priority policies and is computed by the method in [25]. The computation of this term for EDF and RM is detailed in Lemma V.2 and Lemma V.1 of [25], respectively. For details of Theorem 24, please refer to [17].

7 Performance Evaluation

In this section, the performance of the proposed method is evaluated. During the evaluation, we conduct experiments of both scheduling single-DAG systems and scheduling multi-DAG systems using randomly generated task graphs.

7.1 Evaluation of Single-DAG Systems

In this section, the following methods are compared:

- *B-OUR.* The bound in Equation 1 computed by our method with priority assignment policy introduced in Section 5.
- *B-ZHAO.* The bound proposed in [29] with explicit execution order developed alongside its analysis.
- *B-HE.* The bound proposed in [17] with its priority assignment policy.

These three bounds are normalized with respect to *B-HE* as the metric for comparison. So in the figures with respect to normalized bound, *B-HE* is always one. The bounds computed by other priority assignment algorithms [21], [19] are shown dominated by [17], thus not included in the evaluation.

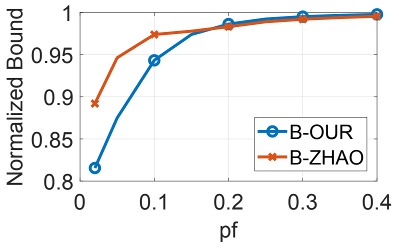


Figure 5 Normalized bound with different pf.

Table 1 The percentage of inferior cases comparing with *B-HE* and *B-ZHAO*.

m	*B-HE*	*B-ZHAO*
4-10	0.11%	77.69%
11-16	0%	17.38%
17-22	0%	1%
23-32	0.01%	1.44%

Task Generation. The DAG tasks are generated using the Erdos-Renyi method [10], where the number of vertices $|V|$ is randomly chosen in a specified interval. For each pair of vertices, the method generates a random value in [0, 1] and adds the edge to the graph if the generated value is less than a predefined *parallelism factor* pf. The larger pf, the more sequential the graph is. If the generated graph has multiple source/sink vertices, a dummy source/sink vertex with zero WCET is added to the graph.

For experiments, we first give a default parameter setting, then tune different parameters for evaluation. The default setting is as follows. The vertex number $|V|$ and the WCET of vertices c_i are randomly chosen in [50, 250] and [50, 100] respectively. For each configuration, we randomly generate 1000 DAG tasks to compute the average normalized bound.

Evaluation with Task Parallelism. We conduct experiments by changing parallelism factor pf with core number $m = 16$. The results are presented in Figure 5. Since the normalized bound is always smaller than one, our method consistently outperforms *B-HE* on average, especially when the DAG task has high parallelism (when pf is small), which is the typical case as benchmarks (such as [11,15]) and practical applications generally possess high parallelism. This is because paths of a DAG with higher parallelism suffer more interference. The priority assignment enabled by our computing method can balance such interference among different paths better, thus reducing the response time bound. From experimental results, compared with *B-HE*, the improvement of response time bound is up to 18.1%. For pf less than 0.2, the performance of our method is better than *B-ZHAO* on average, which further shows our method can balance interference among different paths effectively for DAG tasks with high parallelism. When pf becomes larger, the DAG being more sequential, both bounds becomes closer to *B-HE*, and finally approach to the length of the longest path in the graph. This observation is also obtained in [17,29].

In the following experiments, we randomly choose pf in [0.01, 0.1] to better represent real world applications which generally possess high parallelism as mentioned above.

Evaluation with Core Number. The objective of this experiment is to demonstrate how sensitive the evaluated method is to core number. Figure 6 shows that our method always produces smaller bounds than *B-HE* on average and outperforms *B-HE* by up to 13.3% with $m = 12$. With core number being smaller and larger, *B-OUR* is close to *B-HE*. This is because for core number being smaller, both bounds approach $vol(G)$; for core number being larger, both bounds approach $len(G)$.

When core number is small, *B-ZHAO* performs better than our method. This is because, for a small core number, the computing resource for non-critical vertices (vertices not in the longest path) becomes scarce, which results in non-critical vertices having a large impact on the response time bound. The method for *B-ZHAO* can delicately adjust the execution

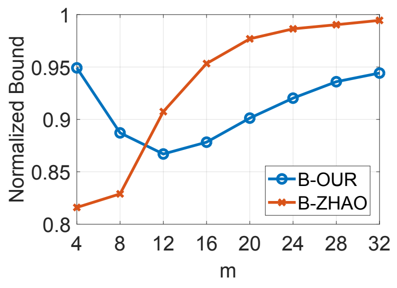


Figure 6 Normalized bound with different m.

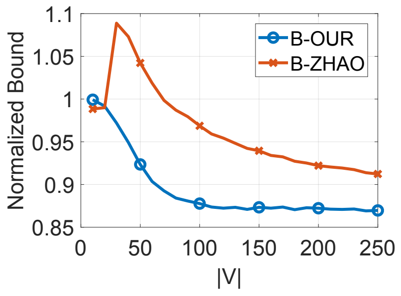


Figure 7 Normalized bound with different $|V|$.

order of non-critical vertices, which is utilized by its analysis method subsequently, finally leading to a smaller bound. With core number increasing, our method, being able to balance interference among different paths effectively, outperforms *B-ZHAO* (e.g., by up to 7.5% with $m = 20$). Real world applications, generally possessing high parallelism [17], may require executing on computing platforms with a larger core number to meet their deadlines. What's more, nowadays mainstream computing platforms generally have a large number of cores. These facts render our method more useful and effective in practice.

Table 1 reports the percentage of inferior cases (cases where the bound *B-OUR* is larger than *B-HE* or *B-ZHAO*) during experiments of Figure 6. For example, in Table 1, for m in [11, 16], no inferior case with respect to *B-HE* is observed and there are 17.38% cases where bounds computed by our method are larger than that of *B-ZHAO*. Table 1 is consistent with Figure 6: for small core numbers, *B-ZHAO* performs better; with core number increasing, our method becomes more effective. We observe that during experiments of Figure 6, the overall inferior cases are less than 0.04% with respect to *B-HE* and less than 24.38% with respect to *B-ZHAO*. This observation further demonstrates the effectiveness of the proposed method.

In the following experiments, we choose core number $m = 16$ as the representative to evaluate the performance.

Evaluation with Vertex Number. This experiment evaluates the sensitivity of the proposed analysis to the vertex number, and results are shown in Figure 7. For $|V| < 30$, the proposed method provides similar results to *B-HE*, which is because the parallelism of the DAG is relatively low when vertex number is small. As analysed above, our method is more effective when parallelism is relatively higher. With vertex number increasing, our method becomes more effective and outperforms *B-HE* more than 10% on average and up to 13.1% and $|V| = 240$. For almost all vertex numbers in this experiment, our method outperforms *B-ZHAO*. We also observe that for small vertex numbers, *B-ZHAO* may produce a bound larger than *B-HE*.

7.2 Evaluation of Multi-DAG Systems

This section evaluates the performance of our method for multi-DAG systems with constrained deadlines. All three methods in Section 7.1 extended their bounds for single DAG task to multi-DAG systems. For task level priority policy, *B-OUR* and *B-HE* can be applied to both dynamic priority (e.g., EDF) and static priority (e.g., RM), denoted as *EDF-OUR*, *RM-OUR*, *EDF-HE*, *RM-HE* respectively; *B-ZHAO* can only be applied to static priority, denoted as *RM-ZHAO*. These five methods are compared in this section.

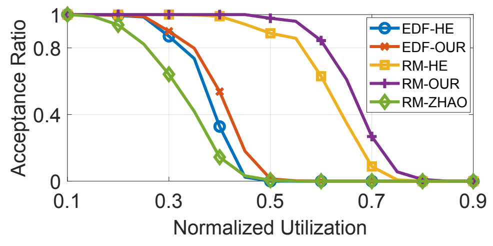


Figure 8 Acceptance ratio with different normalized utilization ($m = 16$).

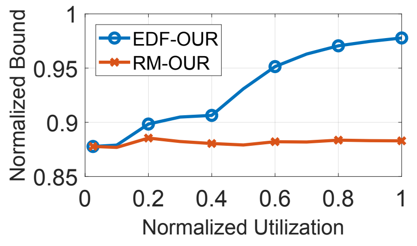


Figure 9 Normalized bound with different normalized utilization ($m = 16$).

Task Set Generation. DAG tasks are generated by the same method as Section 7.1 with pf, $|V|$ and c_i in [0.01, 0.1], [50, 250] and [50, 100] respectively. The period T (which is also the deadline in the experiment) of a DAG task is randomly chosen in $[L, 6L]$, where L is the length of the longest path of the task graph. To generate a task set with specific utilization, we randomly generate a DAG task and add it to the task set until the total utilization reaches the required value.

Evaluation Using Acceptance Ratio. We first test the schedulability of multi-DAG systems using acceptance ratio to evaluate our method. For configuration, we randomly generate 1000 task sets. From the results reported in Figure 8, the proposed method offers better schedulability than that of the state-of-the-art under all settings, especially when normalized utilization is in [0.4, 0.7]. Compared with methods in [17], the improvement of acceptance ratio for EDF and RM is up to 22.2% and 32.0% respectively. *RM-ZHAO* performs worse than *RM-HE*, the reason of which is explained in the following. First, the scheduling for task set in [29] is not work-conserving. Only when a DAG task finishes its execution completely, it can schedule another DAG task to execute. However, before a DAG task finishes its execution completely, some cores may be idle and available to execute tasks (this behavior is fundamental to its underlying response time analysis), which wastes a lot of computing resources. Second, the (α, β)-pair analysis for one DAG task proposed in [29] is not incorporated into its analysis for task set, which makes its performance even worse. The acceptance ratio for *RM-ZHAO* reported in our experiments is consistent with the results reported in [29].

Evaluation Using Normalized Bound. The normalized bound of a task set is the average value among normalized bounds of its tasks. Even if a task set is deemed to be unschedulable, we still try to iterate until all tasks reach a fixed point to compute the response time bound for all tasks. If a task set cannot reach a fixed point, it will be discarded. As reported in Figure 8, the performance of *RM-ZHAO* is relatively poor, which results in that the response time bound cannot be computed for lots of task sets (in our experiment, the fixed point iteration procedure for computing bounds cannot converge before the iterated bounds reach thirty times of its deadlines). Therefore, *RM-ZHAO* is not included in the result of this experiment. For each configuration, we have at least 1000 task sets to compute the average normalized bound. As shown in Figure 9, since the normalized bound is always smaller than 1, our method completely dominates *B-HE* for both EDF and RM, reducing the response time bound by up to 12.3% for EDF and up to 12.4% for RM. The results are consistent with the evaluation of single-DAG systems.

Summary. Experiments in this section show that our computing method and priority assignment can reduce response time bound, improve system schedulability compared to the state-of-the-art by a considerable margin. Specifically, compared to [17], our method reduces response time bound by more than 10% on average and improves schedulability up to 20%. The effectiveness of the method is also supported by the number of DAG tasks with a smaller bound than the state-of-the-art.

8 Related Work

He et al. [17] proposed a dynamic programming algorithm to compute response time bound for DAG tasks with intra-task priority assignment, alongside its response time analysis, which is the most relevant work to this paper. Their computing method assumed that priority assignment should comply with topology constraint. For a wide range of priority assignment without this constraint, their algorithm may produce a wrong bound.

The response time analysis for multi-DAG systems has been intensively studied in recent years, with different scheduling strategies including global scheduling [3, 7, 12, 13, 22, 25] and federated scheduling [4–6, 23, 24]. All the above works involve using the response time bound of a single DAG to bound the intra-task interference, which is the focus of this paper. Fonseca et al. [14] proposed a partitioned scheduling for sporadic DAG tasks.

For response time bound of a single DAG task, zhao et al. [29] explored parallelism and dependencies in DAG structure, and proposed a priority assignment policy and response time bounds based on its CPC (concurrent provider and consumer) model. Han et al. [16] studied typed DAG task for heterogeneous multi-core platform. Sun et al. [27] proposed a method to compute the exact worst case response time with exponential time complexity while being efficient for DAG tasks with small number of vertices. Chen et al. [9] proposed a bound for a DAG with conditional branches by simulating the DAG task with a predefined execution order. The bound in [9] was proved to be timing-anomaly free.

For intra-task priority assignment, in real-time community, Voudouris et al. [28] computed response time bound by simulating the timing-anomaly free scheduler. Pathan et al. [26] proposed a method to utilize intra-task priority assignment to improve resource utilization, and used the idea of ready time and response time of vertices to reduce intra-task interference. Besides research work from real-time community, there are plenty of techniques concerning scheduling task graphs on multiprocessor platform with intra-task priority assignment. Their objective is to reduce the response time on average, not the response time bound. Works [19, 21] considered priority assignment for static scheduling algorithms for DAGs. Kwok and Ahmad proposed a static scheduling algorithm for allocating task graphs to fully connected multiprocessor based on the critical path of task graphs [20].

9 Conclusion, Limitations and Future Work

Computing response time bound of DAG tasks is one of the most important problems in the real-time community. In this paper, we address a serious constraint of the previous result, and propose a method capable of computing response time bound for DAG tasks with arbitrary intra-task priority assignment. Experiments show that our method can greatly reduce the response time bound. In the future, we plan to formulate the graph interference problem into a formal language, clearly identify the context-free grammar inherently associated with this problem, and utilize the automata theory [18] to compute it within a constant number of iterations (the method of this paper computing within $|V|$ iterations) and further improve efficiency. Another direction is searching for an optimal priority assignment with respect to the bound in Equation 1.

References

1 Openmp-api-specification-5.0.pdf. `https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf`. (Accessed on 03/01/2021).

2 Theodore P Baker and Sanjoy K Baruah. Sustainable multiprocessor scheduling of sporadic task systems. In *2009 21st Euromicro Conference on Real-Time Systems*, pages 141–150. IEEE, 2009.

3 Sanjoy Baruah. Improved multiprocessor global schedulability analysis of sporadic dag task systems. In *2014 26th Euromicro conference on real-time systems*, pages 97–105. IEEE, 2014.

4 Sanjoy Baruah. The federated scheduling of constrained-deadline sporadic dag task systems. In *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1323–1328. IEEE, 2015.

5 Sanjoy Baruah. Federated scheduling of sporadic dag task systems. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 179–186. IEEE, 2015.

6 Sanjoy Baruah. The federated scheduling of systems of conditional sporadic dag tasks. In *Proceedings of the 12th International Conference on Embedded Software*, pages 1–10. IEEE Press, 2015.

7 Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, Sebastian Stiller, and Andreas Wiese. Feasibility analysis in the sporadic dag task model. In *2013 25th Euromicro conference on real-time systems*, pages 225–233. IEEE, 2013.

8 Alan Burns and Sanjoy Baruah. Sustainability in real-time scheduling. *Journal of Computing Science and Engineering*, 2(1):74–97, 2008.

9 Peng Chen, Weichen Liu, Xu Jiang, Qingqiang He, and Nan Guan. Timing-anomaly free dynamic scheduling of conditional dag tasks on multi-core systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 18(5s):1–19, 2019.

10 Daniel Cordeiro, Grégory Mounié, Swann Perarnau, Denis Trystram, Jean-Marc Vincent, and Frédéric Wagner. Random graph generation for scheduling simulations. In *Proceedings of the 3rd international ICST conference on simulation tools and techniques*, page 60. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2010.

11 Alejandro Duran, Xavier Teruel, Roger Ferrer, Xavier Martorell, and Eduard Ayguade. Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp. In *Parallel Processing, 2009. ICPP'09. International Conference on*, pages 124–131. IEEE, 2009.

12 José Fonseca, Geoffrey Nelissen, and Vincent Nélis. Improved response time analysis of sporadic dag tasks for global fp scheduling. In *Proceedings of the 25th international conference on real-time networks and systems*, pages 28–37, 2017.

13 José Fonseca, Geoffrey Nelissen, and Vincent Nélis. Schedulability analysis of dag tasks with arbitrary deadlines under global fixed-priority scheduling. *Real-Time Systems*, 55(2):387–432, 2019.

14 José Fonseca, Geoffrey Nelissen, Vincent Nelis, and Luís Miguel Pinho. Response time analysis of sporadic dag tasks under partitioned scheduling. In *2016 11th IEEE Symposium on Industrial Embedded Systems (SIES)*, pages 1–10. IEEE, 2016.

15 Vladimir Gajinov, Srđan Stipić, Igor Erić, Osman S Unsal, Eduard Ayguadé, and Adrián Cristal. Dash: a benchmark suite for hybrid dataflow and shared memory programming models: with comparative evaluation of three hybrid dataflow models. In *Proceedings of the 11th ACM conference on computing frontiers*, page 4. ACM, 2014.

16 Meiling Han, Nan Guan, Jinghao Sun, Qingqiang He, Qingxu Deng, and Weichen Liu. Response time bounds for typed dag parallel tasks on heterogeneous multi-cores. *IEEE Transactions on Parallel and Distributed Systems*, 30(11):2567–2581, 2019.

17 Qingqiang He, Xu Jiang, Nan Guan, and Zhishan Guo. Intra-task priority assignment in real-time scheduling of dag tasks on multi-cores. *IEEE Transactions on Parallel and Distributed Systems*, 30(10):2283–2295, 2019.

18 John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. Introduction to automata theory, languages, and computation. *Acm Sigact News*, 32(1):60–65, 2001.

19 H KASAHARA and S NARITA. Practical multiprocessor scheduling algorithms for efficient parallel processing. *IEEE transactions on computers*, 33(11):1023–1029, 1984.

20 Yu-Kwong Kwok and Ishfaq Ahmad. Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors. *IEEE transactions on parallel and distributed systems*, 7(5):506–521, 1996.

21 Yu-Kwong Kwok and Ishfaq Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys (CSUR)*, 31(4):406–471, 1999.

22 Jing Li, Kunal Agrawal, Chenyang Lu, and Christopher Gill. Outstanding paper award: Analysis of global edf for parallel tasks. In *2013 25th Euromicro Conference on Real-Time Systems*, pages 3–13. IEEE, 2013.

23 Jing Li, Jian Jia Chen, Kunal Agrawal, Chenyang Lu, Chris Gill, and Abusayeed Saifullah. Analysis of federated and global scheduling for parallel real-time tasks. In *2014 26th Euromicro Conference on Real-Time Systems*, pages 85–96. IEEE, 2014.

24 Jing Li, David Ferry, Shaurya Ahuja, Kunal Agrawal, Christopher Gill, and Chenyang Lu. Mixed-criticality federated scheduling for parallel real-time tasks. *Real-time systems*, 53(5):760–811, 2017.

25 Alessandra Melani, Marko Bertogna, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, and Giorgio C Buttazzo. Response-time analysis of conditional dag tasks in multiprocessor systems. In *2015 27th Euromicro Conference on Real-Time Systems*, pages 211–221. IEEE, 2015.

26 Risat Pathan, Petros Voudouris, and Per Stenström. Scheduling parallel real-time recurrent tasks on multicore platforms. *IEEE Transactions on Parallel and Distributed Systems*, 29(4):915–928, 2017.

27 Jinghao Sun, Feng Li, Nan Guan, Wentao Zhu, Minjie Xiang, Zhishan Guo, and Wang Yi. On computing exact wcrt for dag tasks. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2020.

28 Petros Voudouris, Per Stenström, and Risat Pathan. Timing-anomaly free dynamic scheduling of task-based parallel applications. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2017 IEEE*, pages 365–376. IEEE, 2017.

29 Shuai Zhao, Xiaotian Dai, Iain Bate, Alan Burns, and Wanli Chang. Dag scheduling and analysis on multiprocessor systems: Exploitation of parallelism and dependency. In *IEEE Real-Time Systems Symposium*. IEEE, 2020.