Intra-Task Priority Assignment in Real-Time Scheduling of DAG Tasks on Multi-Cores

Qingqiang He, Xu Jiang, Nan Guan, and Zhishan Guo

Abstract—Real-time scheduling and analysis of parallel tasks modeled as directed acyclic graphs (DAG) have been intensively studied in recent years. However, no existing work has explored the execution order of eligible vertices *within* a DAG task. In this paper, we show that this intra-task vertex execution order has a large impact on system schedulability and propose to control the execution order by vertex-level priority assignment. We develop analysis techniques to bound the worst-case response time for the proposed scheduling strategy and design heuristics for proper priority assignment to improve system schedulability as much as possible. We further extend the proposed approach to the general setting of multiple recurrent DAG tasks. Experiments with both realistic parallel benchmark applications and randomly generated workload show that our method consistently outperforms state-of-the-art methods with different task graph structures and parameter configurations.

Index Terms—Intra-task priority assignment, response time analysis, parallel real-time tasks, multi-cores

1 Introduction

Multi-core platforms are more and more widely used in real-time systems, to meet their rapidly increasing requirements in performance and energy efficiency. To fully utilize the power of multi-cores, software must be properly parallelized. The migration from sequential software on single-core platforms to parallel software on multi-core platform poses many challenges to the real-time system design, which make it necessary for new scheduling algorithms and analysis techniques.

Scheduling algorithms on multi-core platforms can be divided into two categories: static and dynamic scheduling. In static scheduling, subtasks are statically assigned to cores during the design phase [1], which may fundamentally underutilize computing resources because the execution time of some subtasks may be less than its worst-case execution time (WCET). Dynamic scheduling can improve resource utilization. However, it may suffer from timing anomalies [2], [3], [4], in the sense that the response time may become longer if the execution time of some subtasks is shorter than its WCET. Due to timing anomalies, safe (yet usually pessimistic) response time bound must be provided for dynamic scheduling algorithms for real-time systems.

The classic response time bound developed by Graham [2] was widely used in literatures [5], [6], [7]. However, this bound assumes that the vertices which are not in the longest path cannot execute in parallel with the vertices in the longest path, making this bound overly pessimistic.

This paper aims at developing real-time scheduling algorithms and analysis techniques for parallel tasks characterized as directed acyclic graphs (DAG) running on a multi-core platform by utilizing intra-task vertex execution order. While scheduling a parallel DAG task, it is possible that at some time point many vertices of this task are eligible for execution, and the number of eligible vertices is more than the number of available cores. Existing scheduling algorithms, such as [5], [6], [7], [8], do not specify the execution order of these vertices in this situation (or assume a non-deterministic execution order). Scheduling algorithms, such as list scheduling [2], specify the scheduling order of these vertices, but do not utilize this order information in the analysis of its timing behavior. Some existing works [9], [10] considered priority assignment for *static* scheduling algorithms for DAGs. These priority assignment strategies are also applicable to *dynamic* scheduling algorithms which are the focus of this paper. However, their performance is not competitive with our proposed approach (as they are not designed for the purpose of optimizing the worst-case response time bound), as shown by the experiment results in Section 8.

In this paper, we show that this intra-task vertex execution order, if properly utilized, can greatly benefit the schedulability of the task. We propose to use intra-task vertex-level priority assignment to control their execution order. The technical contribution of this paper can be summarized as follows:

- By utilizing the priorities of vertices, we derive a new response time bound for a single DAG task, which dominates the state-of-the-art bound in [2].
- We propose an efficient algorithm with polynomial time complexity to compute the above-mentioned response time bound.

- *Q. He, X. Jiang, and N. Guan are with the Department of Computing, Hong Kong Polytechnic University, Hung Hom, Hong Kong. E-mail: {qianghe, nan.guan}@polyu.edu.hk, jiangxu617@163.com.*
- *Z. Guo is with the Department of Electrical and Computer Engineering, University of Central Florida, Orlando, FL 32816. E-mail: zsguo@ucf.edu.*

Manuscript received 21 July 2018; revised 24 Feb. 2019; accepted 31 Mar. 2019. Date of publication 11 Apr. 2019; date of current version 11 Sept. 2019. (Corresponding author: Xu Jiang.)
Recommended for acceptance by M. Guo.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2019.2910525

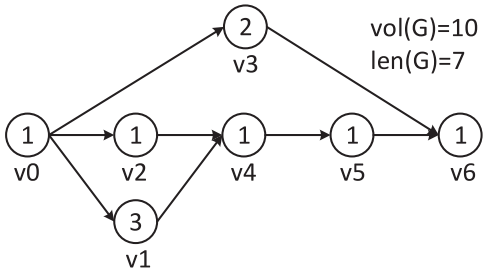


Fig. 1. A DAG task example.

- We propose a priority assignment algorithm to assign priorities to vertices, such that the response time bound is reduced as much as possible.
- We extend our result to the real-time scheduling of multiple recurrent DAG tasks, and give a new schedulability test, which dominates the state-of-the-art test [6] both theoretically and empirically.

We conduct simulation experiments with both realistic parallel benchmark applications and randomly generated workload. Experiment results show that our method consistently outperforms the state-of-the-art methods with different task graph structures and parameter configurations.

The rest of this paper is organized as follows. Section 2 defines the DAG model for parallel tasks and gives some definitions and prior results. Section 3 presents our motivation and the scheduling algorithm used in this paper. Section 4 presents our response time analysis framework for one parallel task. Section 5 introduces the dynamic programming algorithm to compute response time bound and proves its correctness. Section 6 presents the priority assignment algorithm. In Section 7, we extend our method to the scheduling of multiple tasks. Evaluation and experiments results are presented in Section 8. Section 9 discusses related work and Section 10 concludes this paper.

2 Preliminary

2.1 System Model

We consider a multi-core platform with M identical cores. The parallel real-time task is modeled as a directed acyclic graph (DAG) $G = (V, E)$, where V is the set of vertices and $E \subset V \times V$ is the set of directed edges of the graph. 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 completes.

A vertex with no incoming (outgoing) edges is called a *source* (*sink*). Without loss of generality, we assume that G has exactly one source (denoted as v_{src}) and one sink (denoted as v_{sink}). A DAG with multiple source (sink) vertices can be easily transferred to the required form by adding dummy vertices with zero WCET.

We distinguish a path and a complete path of a DAG. A *path* λ starting from vertex π_0 and ending at vertex π_k is a sequence of vertices $(\pi_0, \ldots, \pi_k)$ such that $\forall i \in [0, k)$, $(\pi_i, \pi_{i+1}) \in E$. 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 $len(\lambda)$, i.e., $\sum_{v_i \in \lambda} C_i$, which is the sum of the WCET of all vertices in this path. A *complete path* is a path $(\pi_0, \ldots, \pi_k)$ 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. We define a *longest path* to be a path with the longest length among all paths of the DAG. The length of the longest path of DAG G is denoted as $len(G)$.

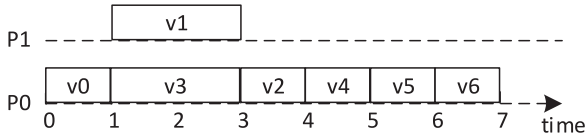


Fig. 2. An example illustrating execution sequence.

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 of the DAG task.

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. These sets can be computed in linear time in the size of DAG.

As long as the graph is a DAG, our model does not impose other restrictions on the dependencies (edges) between vertices. If there is no dependency between vertices in a task graph, this task has high parallelism. If there are dependencies between vertices, this task can still have high parallelism. Dependencies between vertices reduce parallelism, do not invalidate our results.

Fig. 1 shows a DAG task with 7 vertices. The number inside the circles (representing vertices) is the WCET of vertices. We can compute $vol(G) = 10$ and $len(G) = 7$. The longest path is $(v_0, v_1, v_4, v_5, v_6)$. v_0, v_6 are the single source vertex and the single sink vertex respectively.

2.2 Runtime Behavior

At runtime, vertices of G execute at certain time points on certain cores under the decision of a scheduling algorithm. An execution sequence of G describes at every time point t which vertex executes on which core.

With respect to an execution sequence, we say a vertex v is *eligible* at a certain time point if all its predecessors in the execution sequence have finished its execution and thus v can immediately execute if there are available cores.

In an execution sequence, the *start time* $s(v)$ and the *finish time* $f(v)$ of a vertex v are defined to be the time point when vertex v starts its execution on a certain core and the time point when vertex v finishes its execution respectively. Without loss of generality, we assume the source vertex of G starts execution at time 0.

We define the *response time* of G to be $f(v_{sink})$. For a path $\lambda = (\pi_0, \ldots, \pi_k)$, the response time of λ is defined to be $f(\pi_k)$. This paper focuses on deriving a safe upper bound of response time of G.

We use an example to illustrate concepts introduced here. Fig. 2 shows an execution sequence of the DAG in Fig. 1 (v_1's WCET is 3 but only executes for 2 in this execution sequence). At time point $t = 1$, vertices v_1, v_2, v_3 are all eligible. In this execution sequence, $s(v_1) = 1$, $f(v_1) = 3$, $s(v_2) = 3$, $f(v_2) = 4$, $f(v_{sink}) = f(v_6) = 7$, and the response time of the DAG is 7.

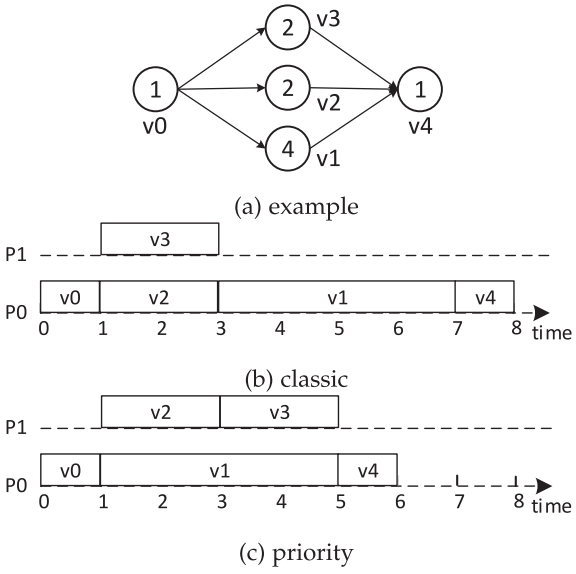


Fig. 3. A motivational example.

2.3 Work-Conserving Scheduling

Much previous work (such as [5], [6], [7], [8]) use work-conserving scheduling algorithms. In *work-conserving* scheduling, an eligible vertex must be executed if there are available cores. For example, *list scheduling* [2] is an instance of work-conserving scheduling. Our work is based on the classic result in [2], stated in the following theorem.

Theorem 1 (Classic Bound [2]). *The response time R of a DAG task G with a constrained deadline scheduled by a work-conserving algorithm on a platform with M cores can be bounded by*

$$R \leq len(G) + \frac{vol(G) - len(G)}{M}. \tag{1}$$

3 Method Overview

3.1 Motivational Example

At runtime, it is possible that at some time point the number of eligible vertices is larger than M (the number of cores). The work-conserving scheduling constraint introduced in Section 2.3 does not specify which vertices should be executed in this circumstance. Different instances of work-conserving scheduling may have different strategies to choose eligible vertices for execution. The response time bound in Theorem 1 is valid for all possible instances of work-conserving scheduling. Therefore, conceptually we can view the response time bound as derived for a work-conserving scheduling algorithm that *arbitrarily* chooses eligible vertices to execute on available cores at runtime.

In the following, we will use an example to show that by using a proper strategy to choose among eligible vertices for execution, the response time of the DAG task can be reduced compared with the arbitrary choice.

Suppose the DAG task G shown in Fig. 3a executes on $M = 2$ cores. Therefore, by Equation (1), we can compute its response time bound of an arbitrary work-conserving execution sequence

$$len(G) + \frac{vol(G) - len(G)}{M} = 6 + \frac{10 - 6}{2} = 8.$$

There is indeed a possible scheduling sequence that reaches this response time bound, as shown in Fig. 3b. In this scheduling sequence, each vertex executes to its WCET. At time 1, when v_0 is finished, v_1, v_2 and v_3 are all eligible for execution, but only two of them can start execution as $M = 2$. Suppose the scheduler selects to first execute v_2 and v_3, then v_1 can start execution at time 3 and the whole DAG finishes execution at time 8.

However, if the scheduler chooses to execute v_1 at time 1 (and the other core is used to execute one of v_2 and v_3), the response time of the DAG is 6, as shown in Fig. 3c. From this example, we can see that the choice of eligible vertices for execution affects the actual response time. Among the three paths from v_0 to v_4, the one via v_1 is the longest. Intuitively, one should prioritize vertices along the longest path for execution in order to get a smaller response time.

3.2 Priority-Based Scheduling

Inspired by the above example, we propose to assign priorities to the vertices and at runtime schedule the eligible vertices strictly according to their priorities. Given a total priority order of all vertices, at any time instant at runtime, the scheduler always chooses at most M highest-priority eligible vertices for execution.

Formally, we assign a priority $p(v_i)$ to each vertex v_i of the DAG. We say vertex v_i has higher priority than vertex v_j, if $p(v_i) < p(v_j)$.

We propose *prioritized list scheduling*, which satisfies the following properties:

- *Work-conserving*. Stated in Section 2.3.
- *Preemptive*. A higher-priority eligible vertex can preempt the execution of a lower-priority one. The preempted lower-priority vertex will resume execution later when there are available cores.

The result derived in the paper is valid for any specific scheduling algorithm satisfying these two properties.

4 Response Time Analysis

In this section, we introduce how to derive the response time bound of a DAG G scheduled by prioritized list scheduling, given an arbitrary order of vertex priorities. The only assumption we make here is that the priority order of vertices does not conflict with the topology order of the graph, i.e., a vertex's priority is not higher than any of its predecessors. There are exponentially many possible priority assignments complying with our assumption. If a priority assignment satisfies this assumption, we say priorities are assigned in *descending order* or a priority assignment with *descending order*. Later in Section 6 we will discuss how to assign priorities such that the response time bound of G can be reduced as much as possible.

Without loss of generality, we assume the whole DAG is released at time 0. Similar to [6], we define the concept of *critical path*.

Definition 1 (Critical Path). *A critical path $\lambda = (\pi_0, \ldots, \pi_k)$ of an execution sequence of a DAG task is a complete path satisfying the following property*

$$\forall \pi_i \in \lambda \setminus \{\pi_0\} : f(\pi_{i-1}) = \max_{u \in pred(\pi_i)} \{f(u)\}. \tag{2}$$

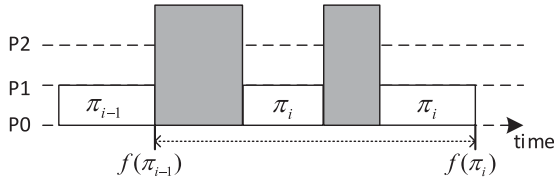


Fig. 4. An example illustrating interference.

Notice that the critical path is *not necessarily* the longest path of the DAG. The critical path depends on how the DAG is actually scheduled, i.e., the critical path of a DAG may be different in different execution sequences of the DAG. For example, in Fig. 2, the critical path of this execution sequence by our definition is $(v_0, v_2, v_4, v_5, v_6)$, while the longest path of the DAG is $(v_0, v_1, v_4, v_5, v_6)$.

Definition 2 (Interference). *For a critical path $\lambda = (\pi_0, \ldots, \pi_k)$, we say that a vertex $v \in V$ interferes with vertex $\pi_i \in \lambda$ if and only if vertex v executes in a time interval satisfying both of the following conditions:*

- *the time interval is in $[f(\pi_{i-1}), f(\pi_i))$, and*
- *π_i does not execute in this time interval.*

We say a vertex v *interferes* with critical path λ if and only if v interferes with a vertex $\pi_i \in \lambda$.

For example, in Fig. 4, suppose π_{i-1} and π_i are two vertices in the critical path. If vertex v has execution during the grey intervals as shown in Fig. 4, we say vertex v interferes with π_i (and thus interferes with this critical path).

The following lemma gives the *necessary* condition for a vertex v to interfere with a vertex π_i in a critical path.

Lemma 1. *Given a critical path $\lambda = (\pi_0, \ldots, \pi_i, \ldots, \pi_k)$, $v \in V \setminus \{\pi_i\}$. If v interferes with π_i, then the following three conditions must all be true*

- $v \notin ance(\pi_i)$
- $v \notin desc(\pi_i)$
- $p(v) \leq p(\pi_i)$

Proof. If v interferes with π_i, then v has execution in the interval $[f(\pi_{i-1}), f(\pi_i))$, which means $f(v) > f(\pi_{i-1})$. If $v \in ance(\pi_i)$, then π_{i-1} cannot satisfy Equation (2), which contradicts that π_{i-1} is in λ. We have $v \notin ance(\pi_i)$.

Again, if v interferes with π_i, then v has execution in the interval $[f(\pi_{i-1}), f(\pi_i))$, which means $s(v) < f(\pi_i)$. If $v \in desc(\pi_i)$, this fact contradicts the definition of DAG where a descendant cannot start execution before its ancestors. We have $v \notin desc(\pi_i)$.

Also, if v interferes with π_i, then v has execution in the interval during which vertex π_i does not execute. Note that according to the definition of critical path, during the interval $[f(\pi_{i-1}), f(\pi_i))$, all predecessors of π_i have completed their execution. The only reason why vertex π_i cannot execute is that all the cores are busy with vertices having priorities higher than or equal to $p(\pi_i)$. We have $p(v) \leq p(\pi_i)$. □

Definition 3 (Interference Set). *The interference set of a vertex $\pi_i \in V$ is defined as*

$$I(\pi_i) = \{v \in V \setminus \{\pi_i\} | v \notin ance(\pi_i) \wedge v \notin desc(\pi_i) \wedge p(v) \leq p(\pi_i)\}.$$

The interference set of a path λ is defined as

$$I(\lambda) = \bigcup_{\pi_i \in \lambda} I(\pi_i).$$

Lemma 2. *For a critical path λ, $\forall v \in V \setminus I(\lambda)$, v cannot interfere with λ.*

Proof. If v interferes with critical path λ, there exists a vertex $\pi_i \in \lambda$ such that vertex v interferes with vertex π_i. According to Lemma 1, we have $v \in I(\pi_i)$. Subsequently, $v \in I(\lambda)$. The contrapositive of the lemma follows and the lemma is true. □

Definition 4. *For a path λ, we define*

$$R(\lambda) = len(\lambda) + \frac{vol(I(\lambda))}{M}. \tag{3}$$

We can think of $R(\lambda)$ as the *response time bound* of path λ.

Theorem 2. *The response time R of a DAG task with a constrained deadline scheduled by prioritized list scheduling on a platform with M cores can be bounded by*

$$R \leq \max_{\lambda \in \Pi(G)} \{R(\lambda)\}, \tag{4}$$

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

Proof. We define $\Lambda(G)$ as the set of all critical paths of the DAG G. For a critical path λ, according to Lemma 2, the workload which can interfere with λ is bounded by $vol(I(\lambda))$. And according to Definitions 1 and 2, when vertices in a critical path cannot execute, all cores must be busy with vertices in $I(\lambda)$. So the response time of critical path λ is bounded by

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

And the response time of the DAG task is bounded by

$$\max_{\lambda \in \Lambda(G)} \{R(\lambda)\}.$$

It is hard to know which complete path is a critical path prior to the completion of execution of the DAG task, and since $\Lambda(G) \subset \Pi(G)$, the theorem follows. □

Our bound in Equation (4) dominates the classic bound in Equation (1), i.e., for any DAG task G

$$\max_{\lambda \in \Pi(G)} \{R(\lambda)\} \leq len(G) + \frac{vol(G) - len(G)}{M}.$$

In fact, let the complete path which gives our bound be λ^*. By Definition 3, $\forall \lambda \in \Pi(G)$, $\forall v \in V$, $v \in \lambda \Rightarrow v \notin I(\lambda)$. We have $v \in I(\lambda) \Rightarrow v \in V \setminus \lambda$, which means $vol(I(\lambda)) \leq vol(G) - len(\lambda)$. We have $\max_{\lambda \in \Pi(G)}\{R(\lambda)\} = R(\lambda^*) \leq len(\lambda^*) + \frac{vol(G)-len(\lambda^*)}{M} \leq len(G) + \frac{vol(G)-len(G)}{M}$

5 Computing Response Time Bound

In this section, we present how to calculate the response time bound given by Equation (4).

It is easy to see that in a DAG, the number of paths can be exponential in the size of the DAG. So it is impractical to enumerate all the paths to compute the response time bound. In this paper, we use dynamic programming to solve this problem.

First, we define some useful notations. For a path $\lambda = (\pi_0, \ldots, \pi_k)$, we use a tuple $\langle \pi_k, \lambda, R(\lambda)\rangle$ that corresponds to path λ.

For a tuple $\langle u, \lambda, R(\lambda)\rangle$, and an edge $(u, v) \in E$, a new tuple $\langle v, \lambda', R(\lambda')\rangle$ can be computed, where $\lambda' = \lambda \cup \{v\}$. We say that $\langle u, \lambda, R(\lambda)\rangle$ *generates* $\langle v, \lambda', R(\lambda')\rangle$, denoted by

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

The relation *generation* means if an edge between vertex u and vertex v exists, and u is the predecessor of v, a new tuple of v can be computed from the tuple of u.

Given two tuples $\langle v, \lambda_1, R(\lambda_1)\rangle$, $\langle v, \lambda_2, R(\lambda_2)\rangle$, we say $\langle v, \lambda_1, R(\lambda_1)\rangle$ *dominates* $\langle v, \lambda_2, R(\lambda_2)\rangle$, denoted by

$$\langle v, \lambda_1, R(\lambda_1)\rangle \succcurlyeq \langle v, \lambda_2, R(\lambda_2)\rangle,$$

if and only if $R(\lambda_1) \geq R(\lambda_2)$.

The relation *domination* means if two paths λ_1 and λ_2 end at the same vertex v, if the tuple of λ_1 dominates the tuple of λ_2, then the response time bound $R(\lambda_1)$ of λ_1 is larger than the response time bound $R(\lambda_2)$ of λ_2. Note that the domination between tuples of λ_1 and λ_2 requires that these two paths end at the same vertex.

It is obvious that for domination, transitivity holds, i.e., if

$$\langle v, \lambda_1, R(\lambda_1)\rangle \succcurlyeq \langle v, \lambda_2, R(\lambda_2)\rangle,$$

and

$$\langle v, \lambda_2, R(\lambda_2)\rangle \succcurlyeq \langle v, \lambda_3, R(\lambda_3)\rangle,$$

then

$$\langle v, \lambda_1, R(\lambda_1)\rangle \succcurlyeq \langle v, \lambda_3, R(\lambda_3)\rangle.$$

The following lemma gives an important property concerning generation and domination between tuples.

Lemma 3. *Given* $\langle u, \lambda_1, R(\lambda_1)\rangle \rightsquigarrow \langle v, \lambda_1', R(\lambda_1')\rangle$, $\langle u, \lambda_2, R(\lambda_2)\rangle \rightsquigarrow \langle v, \lambda_2', R(\lambda_2')\rangle$, *and* $\langle u, \lambda_1, R(\lambda_1)\rangle \succcurlyeq \langle u, \lambda_2, R(\lambda_2)\rangle$, *if*

$$I(\lambda_1) \cap I(v) = I(\lambda_2) \cap I(v),$$

then

$$\langle v, \lambda_1', R(\lambda_1')\rangle \succcurlyeq \langle v, \lambda_2', R(\lambda_2')\rangle.$$

Proof. We define $A_1 = I(\lambda_1) \cap I(v)$, and $A_2 = I(\lambda_2) \cap I(v)$

$$I(\lambda_1) \cap I(v) = I(\lambda_2) \cap I(v)$$

$$\Rightarrow I(v) \setminus A_2 \subset I(v) \setminus A_1.$$

We have

$$\frac{vol(I(v) \setminus A_2)}{M} \leq \frac{vol(I(v) \setminus A_1)}{M}.$$

Since $R(\lambda_2) \leq R(\lambda_1)$, we have

$$\Rightarrow len(\lambda_2) + \frac{vol(I(\lambda_2)) + vol(I(v) \setminus A_2)}{M} \leq len(\lambda_1) + \frac{vol(I(\lambda_1)) + vol(I(v) \setminus A_1)}{M}.$$

It is obvious that

$$vol(I(\lambda_2)) + vol(I(v) \setminus A_2) = vol(I(\lambda_2) \cup I(v))$$

$$vol(I(\lambda_1)) + vol(I(v) \setminus A_1) = vol(I(\lambda_1) \cup I(v)).$$

We have

$$len(\lambda_2) + \frac{vol(I(\lambda_2) \cup I(v))}{M} \leq len(\lambda_1) + \frac{vol(I(\lambda_1) \cup I(v))}{M}$$

$$\Rightarrow R(\lambda_2') \leq R(\lambda_1').$$

That is

$$\langle v, \lambda_1', R(\lambda_1')\rangle \succcurlyeq \langle v, \lambda_2', R(\lambda_2')\rangle.$$

We reach the conclusion. □

Lemma 3 means if (1) vertex v is a successor of u, and (2) path λ_1 and λ_2 end at the same vertex u, and (3) the response time bound of λ_1 is larger than the response time bound of λ_2, and (4) the interference set $I(\lambda_1)$ and $I(\lambda_2)$ have the same vertices with the interference set of vertex v, then the response time bound of λ_1' is larger than the response time bound of λ_2'.

Lemma 4. *For a priority assignment with descending order, given* $\langle u, \lambda_1, R(\lambda_1)\rangle \rightsquigarrow \langle v, \lambda_1', R(\lambda_1')\rangle$ *and* $\langle u, \lambda_2, R(\lambda_2)\rangle \rightsquigarrow \langle v, \lambda_2', R(\lambda_2')\rangle$, *if*

$$\langle u, \lambda_1, R(\lambda_1)\rangle \succcurlyeq \langle u, \lambda_2, R(\lambda_2)\rangle,$$

then

$$\langle v, \lambda_1', R(\lambda_1')\rangle \succcurlyeq \langle v, \lambda_2', R(\lambda_2')\rangle.$$

Proof. We use Lemma 3 to prove this lemma. For $\forall w \in I(\lambda_1) \cap I(v)$.

First, $u \in ance(v) \Rightarrow u \notin I(v)$, and since $w \in I(v)$, we have

$$w \neq u \Rightarrow w \in V \setminus \{u\}.$$

Second, $w \in I(\lambda_1) \Rightarrow w \notin desc(u)$, because if $w \in desc(u)$, for $\forall v_i \in \lambda_1$, $w \in desc(v_i) \Rightarrow w \notin I(v_i) \Rightarrow w \notin I(\lambda_1)$. This contradicts $w \in I(\lambda_1)$. So we have

$$w \notin desc(u).$$

Third, $w \in I(v) \Rightarrow w \notin ance(u)$, because if $w \in ance(u)$, since $u \in pred(v)$, we have $w \in ance(v) \Rightarrow w \notin I(v)$. This contradicts $w \in I(v)$. So we have

$$w \notin ance(u).$$

Finally, $w \in I(\lambda_1) \Rightarrow \exists \pi_i \in \lambda_1$ such that $w \in I(\pi_i) \Rightarrow p(w) \leq p(\pi_i)$. By descending order, in path λ_1, $p(\pi_i) \leq p(u)$. We have

$$p(w) \leq p(u).$$

In summary, we reach the conclusion that $w \in I(u)$.

Since $u \in \lambda_2$, we have

$$I(u) \subset I(\lambda_2) \Rightarrow v \in I(\lambda_2).$$

In conclusion, for $\forall w \in I(\lambda_1) \cap I(v)$

$$w \in I(\lambda_2) \Rightarrow I(\lambda_1) \cap I(v) \subset I(\lambda_2).$$

By similar reasons, we have

$$I(\lambda_2) \cap I(v) \subset I(\lambda_1),$$

which means

$$I(\lambda_1) \cap I(v) = I(\lambda_2) \cap I(v).$$

By Lemma 3, the lemma follows. □

Intuitively, Lemmas 3 and 4 state that with descending order, if two paths λ_1, λ_2 end at the same vertex u, $R(\lambda_1) \geq R(\lambda_2)$, and (u, v) is an edge, then $R(\lambda_1 \cup \{v\}) \geq R(\lambda_2 \cup \{v\})$.

Definition 5. $\forall v \in V$

$$\lambda_v = \begin{cases} \{v_{src}\} & v = v_{src} \\ \lambda_{u^*} \cup \{v\} & v \neq v_{src} \end{cases}, \tag{5}$$

where

$$u^* = \arg \max_{u \in pred(v)} \left\{ len(\lambda_u) + C(v) + \frac{vol(I(\lambda_u) \cup I(v))}{M} \right\}. \tag{6}$$

$\forall v \in V$, we define $G(v) = (V', E')$, where

$$V' = ance(v) \cup \{v\}$$

$$E' = \{(u_1, u_2) | (u_1, u_2) \in E \wedge u_1 \in V' \wedge u_2 \in V'\}.$$

By definition, $G(v)$ is a subgraph of G consisting of vertex v and its ancestors.

For brevity, let $\Pi(v) = \Pi(G(v))$. It is evident that $G(v_{sink}) = G$, $\Pi(v_{sink}) = \Pi(G)$.

The following lemma shows that λ_v can be used to compute the response time bound of G.

Lemma 5. *For a priority assignment with descending order,* $\forall v \in V$

$$R(\lambda_v) = \max_{\lambda \in \Pi(v)} \{R(\lambda)\}.$$

Proof. We prove it by induction. Let $\sigma = (\pi_0, \ldots, \pi_i, \ldots, \pi_n)$ be a topological order of G. It is obvious that $\pi_0 = v_{src}$, $\pi_n = v_{sink}$.

For $i = 0$, we have $\pi_i = v_{src}$, by Definition 5, $\lambda_v = (v_{src})$, $\Pi(v) = \{(v_{src})\}$. The lemma holds trivially.

For $i \neq 0$, suppose $\forall j < i$, the claim holds. Since σ is a topological order, $\forall u \in pred(\pi_i)$, the claim holds. Since $\lambda_{\pi_i} \in \Pi(\pi_i)$, $R(\lambda_{\pi_i}) \leq \max_{\lambda \in \Pi(\pi_i)}\{R(\lambda)\}$.

$\forall \lambda \in \Pi(\pi_i)$. Since λ is a complete path in $G(\pi_i)$, $\exists u \in pred(\pi_i)$ such that $\lambda = (v_{src}, \ldots, u, \pi_i)$. Let $\lambda' = (v_{src}, \ldots, u)$. Since $\lambda' \in \Pi(u)$, by inductive assumption, we have

$$R(\lambda') \leq R(\lambda_u).$$

So $\langle u, \lambda_u, R(\lambda_u)\rangle \succcurlyeq \langle u, \lambda', R(\lambda')\rangle$. Let $\lambda_1 = \lambda_u \cup \{\pi_i\}$. We have

$$\langle u, \lambda_u, R(\lambda_u)\rangle \rightsquigarrow \langle v, \lambda_1, R(\lambda_1)\rangle$$

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

By Lemma 4

$$\langle \pi_i, \lambda_1, R(\lambda_1)\rangle \succcurlyeq \langle \pi_i, \lambda, R(\lambda)\rangle.$$

By Equation (6), we have

$$R(\lambda_{\pi_i}) \geq R(\lambda_1),$$

so $\langle \pi_i, \lambda_{\pi_i}, R(\lambda_{\pi_i})\rangle \succcurlyeq \langle \pi_i, \lambda_1, R(\lambda_1)\rangle$.

We have

$$\langle \pi_i, \lambda_{\pi_i}, R(\lambda_{\pi_i})\rangle \succcurlyeq \langle \pi_i, \lambda, R(\lambda)\rangle,$$

which means

$$R(\lambda_{\pi_i}) \geq R(\lambda).$$

We have $\forall \lambda \in \Pi(\pi_i)$

$$R(\lambda_{\pi_i}) \geq R(\lambda),$$

which means

$$R(\lambda_{\pi_i}) \geq \max_{\lambda \in \Pi(\pi_i)} \{R(\lambda)\}.$$

Finally, we have $R(\lambda_{\pi_i}) = \max_{\lambda \in \Pi(\pi_i)}\{R(\lambda)\}$. The lemma follows. □

Theorem 3. *For a priority assignment with descending order*

$$R(\lambda_{v_{sink}}) = \max_{\lambda \in \Pi(G)} \{R(\lambda)\}.$$

Proof. By Lemma 5, the theorem follows. □

With Theorems 2 and 3, according to Definition 5, we give the following algorithm.

Algorithm 1. Computing Response Time Bound

```
1:  Input: DAG G = (V, E); every vertex v_i ∈ V is with its
    WCET C_i and its priority p(v_i); the number of cores M
2:  Output: the response time bound
3:  σ ← TOPOLOGICAL_ORDER(G)
4:  λ_{v_src} ← {v_src}
5:  for v_i ∈ σ from v_src to v_sink do
6:    if v_i ≠ v_src then
7:      u* ← arg max_{u∈pred(v_i)}{len(λ_u) + C_i + vol(I(λ_u)∪I(v_i))/M}
8:      λ_{v_i} ← λ_{u*} ∪ {v_i}
9:    end if
10: end for
11: return R(λ_{v_sink})
```

In Algorithm 1, we first compute a topological order of the DAG (line 3). Then, we start to compute the response

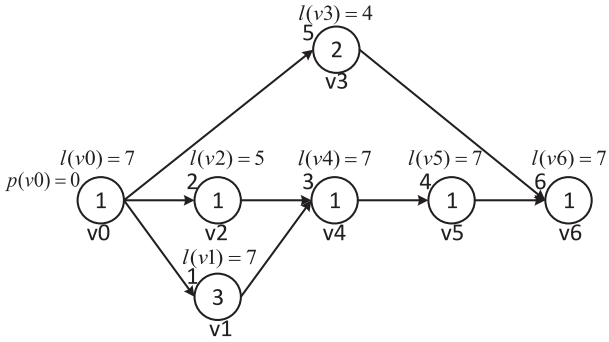


Fig. 5. An example illustrating priority assignment.

time bound of every vertex in the DAG. Due to topological order, at every vertex v_i, the response time bounds of its predecessors must have been computed. In line 7-8, we use Equation (5) to compute λ_{v_i}. The response time bound of the DAG is given by the response time bound of the sink vertex. The complexity of the algorithm is $O(|V| + |E|)$.

6 Priority Assignment

The policy of assigning priorities is of critical importance. Different policies can lead to very different response time bounds. In this section, we present our priority assignment algorithm.

Algorithm 2. Assigning Priorities

1: **Input**: DAG $G = (V, E)$; every vertex $v_i \in V$ is with its WCET C_i
2: **Output**: the priority $p(v_i)$ of every vertex $v_i \in V$
3: $COMPUTE\_LENGTH(G)$
4: $p \leftarrow 0$
5: $ASSIGN\_PRIORITY(G, p)$

Algorithm 3. Procedure $COMPUTE\_LENGTH([in]\ G)$

1: **Input**: DAG $G = (V, E)$; every vertex $v_i \in V$ is with its WCET C_i
2: **Output**: the $lf(v_i), lb(v_i), l(v_i)$ of every vertex $v_i \in V$
3: $\sigma \leftarrow TOPOLOGICAL\_ORDER(G)$
4: $lf(v_{src}) \leftarrow C_{src}$
5: **for** $v_i \in \sigma$ from v_{src} to v_{sink} **do**
6: **if** $v_i \neq v_{src}$ **then**
7: $lf(v_i) \leftarrow C_i + \max_{u \in pred(v_i)}\{lf(u)\}$
8: **end if**
9: **end for**
10: $lb(v_{sink}) \leftarrow C_{sink}$
11: **for** $v_i \in \sigma$ from v_{sink} to v_{src} **do**
12: **if** $v_i \neq v_{sink}$ **then**
13: $lb(v_i) \leftarrow C_i + \max_{u \in succ(v_i)}\{lb(u)\}$
14: **end if**
15: **end for**
16: **for** $v_i \in V$ **do**
17: $l(v_i) \leftarrow lf(v_i) + lb(v_i) - C_i$
18: **end for**

Algorithm 2 first computes some heuristic information in line 3 and, does priority assignment in line 5. Algorithm 3 runs a straight dynamic programming to compute the length of the longest path through each vertex. In line 4–9, we traverse the DAG forward in the topological order, computing the length of the path 1) starting from v_{src}; 2) ending at any vertex $v_i \in V$; 3) and with the maximum length. This length is stored in $lf(v_i)$. Similarly, in line 10–15, we traverse the DAG backward in the reverse topological order, computing the length of the path 1) starting from at any vertex $v_i \in V$; 2) ending at v_{sink}; 3) and with the maximum length. This length is stored in $lb(v_i)$. In line 16–18, for each vertex $v_i \in V$, we compute the length of the path 1) with v_i in this path; 2) with the maximum length. This length is stored in $l(v_i)$.

Algorithm 4. Procedure $ASSIGN\_PRIORITY([in]\ G, [inout]\ p)$

1: **Input**: DAG $G = (V, E)$; every vertex $v_i \in V$ is with its WCET C_i and $lf(v_i), lb(v_i), l(v_i)$ defined in Algorithm 3; p: the next available priority
2: **Output**: the priority $p(v_i)$ of every vertex $v_i \in V$
3: **while** $V \neq \varnothing$ **do**
4: $v \leftarrow$ vertex $v_i \in V$ which has no predecessor and is with maximum $l(v_i)$ (ties broken arbitrarily)
5: $p(v) \leftarrow p$; $p \leftarrow p + 1$; $A \leftarrow succ(v)$
6: $G \leftarrow$ the graph obtained by removing v and its related edges
7: **while** $A \neq \varnothing$ **do**
8: $v \leftarrow$ vertex $v_i \in A$ which is with maximum $l(v_i)$ (in case of ties, with maximum $lb(v_i)$, ties broken arbitrarily)
9: **if** $pred(v) \neq \varnothing$ **then**
10: $G' \leftarrow$ the graph composed of vertices in $ance(v)$ and their related edges
11: $ASSIGN\_PRIORITY(G', p)$
12: $G \leftarrow$ the graph obtained by removing vertices in $ance(v)$ and their related edges
13: **end if**
14: $p(v) \leftarrow p$; $p \leftarrow p + 1$; $A \leftarrow succ(v)$
15: $G \leftarrow$ the graph obtained by removing v and its related edges
16: **end while**
17: **end while**

Algorithm 4 assigns priorities to vertices recursively. The parameter p has the direction type of $inout$, which means the procedure both receives p from the caller procedure and returns p to the caller procedure. In Algorithm 4, first, priorities are assigned to vertices in the topological order, which means a vertex cannot be assigned a priority until all of its ancestors have been assigned priorities. Second, priorities are assigned to vertices according to $l(v_i)$ as computed in Algorithm 3, which means vertices with larger $l(v_i)$ can be assigned with higher priorities.

The complexity of Algorithm 2 is $O(|V| + |E|)$. We still use the example in Fig. 1 to illustrate the priority assignment algorithm, as shown in Fig. 5. In Fig. 5, the $l(v_i)$ as computed in Algorithm 2 is denoted beside the vertex, and priorities are also labeled beside the vertex.

The priority assignment policy shown in Algorithm 2 has the following properties:

- *Property 1.* Priorities are assigned in descending order.
- *Property 2.* $l(v_i)$ is the length of the longest path through each vertex v_i.

TABLE 1
Summary of OpenMP Benchmark Applications

Applications	Source	Vertices	Edges	Volume
alignment	bots[11]	400	399	469879
fft	bots	227	304	268
fib	bots	353	528	353
sort	bots	130	193	4369
lu_for	bots	280	279	95806
lu_single	bots	301	435	95238
strassen	bots	122	177	7890
botsspar	spec2012 [12]	290	424	381683
nbody	dash [13]	320	469	38113
overlap	openmpmpi [14]	408	604	431
pingpong	openmpmpi	408	604	424
taskbench	openmpbench [15]	216	311	1272

- *Property 3.* One of the longest paths has the highest priorities. Formally, there exists a path $\lambda \in \Pi(G)$ such that $(len(\lambda) = \max_{\lambda_i \in \Pi(G)}\{len(\lambda_i)\}) \wedge (I(\lambda) = \varnothing)$.

Property 1 holds, because priorities are assigned in the topological order and a vertex getting a priority at an earlier time always gets a higher priority (i.e., the numeric value of the priority is small).

Property 2 corresponds to our motivation shown in Section 3.1 and acts as a heuristic for our priority assignment. We always want to assign higher priorities to vertices in a longer complete path. The quantity $l(v_i)$, which means the length of the longest path through vertex v_i, is a suitable heuristic.

Property 3 holds, because when a vertex v is assigned a priority, the next vertex v_i to be selected to assign a priority is always chosen from its successors and with the maximum $l(v_i)$ (in Algorithm 4, line 8) unless there are predecessors of v_i which have not been assigned priorities. So there exists a longest path $\lambda \in \Pi(G)$ such that $\forall v_i \in \lambda$, $I(v_i) = \varnothing$. We have $I(\lambda) = \varnothing$. We mention that the heuristic of assigning higher priorities to vertices in a longer complete path is not formally defined, and it can be hard to achieve this heuristic perfectly. Our algorithm only guarantees that one of the longest paths has the highest priorities as stated in Property 3.

7 Extension to Multiple DAG Tasks

In this section, we apply our scheduling algorithm and analysis techniques to the global scheduling of multiple DAG tasks.

We first define some notations. We consider a task set τ of n tasks $\{\tau_1, \tau_2, \ldots, \tau_n\}$, scheduled on a multi-core platform of M identical cores. Each DAG task $\tau_i \in \tau$ is modeled as a tuple (G_i, D_i, T_i), where D_i is the deadline and T_i is the period. In the following, we only consider task set with constraint deadlines, i.e., $D_i \leq T_i$.

For any global work-conserving scheduler, a schedulability test is presented in [6]. We restate it in our notations as follows:

Theorem 4 ([6]). *For a DAG task set τ globally scheduled by any work-conserving scheduler on a platform with M cores, a bound R_k on the response time of a task τ_k can be derived by the fixed-point iteration of the following expression, starting with $R_k = len(G_k)$*

$$R_k = len(G_k) + \frac{vol(G_k) - len(G_k)}{M} + \frac{\sum_{\forall i \neq k} I_k^i(R_k)}{M}, \tag{7}$$

where $I_k^i(R_k)$ is the upper bound of the interference of task τ_i to τ_k during an interval of length R_k.

The result in [6] was developed for a more general DAG model with conditional branches, but can be directly applied to the DAG model of this paper, which is a special case of [6]. For details of Theorem 4, please refer to [6].

In the following, we extend our method to multiple DAG tasks. We first present the scheduling algorithm called *global prioritized list scheduling*. The scheduling algorithm is with two levels: task level and vertex level. In the task level, the scheduling algorithm is the same as [6], which can be any global work-conserving scheduler, such as EDF, RM. In the vertex level, the vertices inside a task are scheduled by prioritized list scheduling presented in this paper.

Theorem 5. *For a DAG task set τ scheduled by global prioritized list scheduling on a platform with M cores, a bound R_k on the response time of a task τ_k can be derived by the fixed-point iteration of the following expression, starting with $R_k = L_k$*

$$R_k = \max_{\lambda \in \Pi(G_k)} \left\{ len(\lambda) + \frac{vol(I(\lambda))}{M} \right\} + \frac{\sum_{\forall i \neq k} I_k^i(R_k)}{M}. \tag{8}$$

Theorem 5 is a straightforward extension of Theorems 2 and 4.

8 Performance Evaluation

In this section, we evaluate the performance of our proposed prioritized list scheduling algorithm and response time analysis technique. We both evaluate the performance of scheduling one task and scheduling multiple tasks. In our evaluation, we use both task graph models derived from realistic OpenMP benchmark applications and randomly generated task graphs.

8.1 Evaluation of Scheduling One Task

We first present the evaluation results with benchmark applications. The detailed information of benchmark applications used in our evaluation is provided in Table 1.

We transform these benchmark applications into task graphs by inserting instructions (stubs) into their source codes. These stubs serve for two purposes: generating task graphs and measuring the execution time of each vertex in graphs. The methodology to generate task graphs from the source codes is introduced in [16]. We run benchmark applications with stubs to measure the execution time of vertices on a machine with Intel i7-4770 CPU with 3.5 GHZ and 8,192 KB cache size, 4 GB RAM size. Although safe WCET of benchmark applications cannot be obtained by this method, these execution time values give a rough approximation of the workload of vertices. Note that our analysis is not directly conducted on benchmark applications, but on task graphs obtained by transforming these benchmarks. This is because this paper focuses on the timing behavior of parallel tasks, not their functionalities. Our schedulability

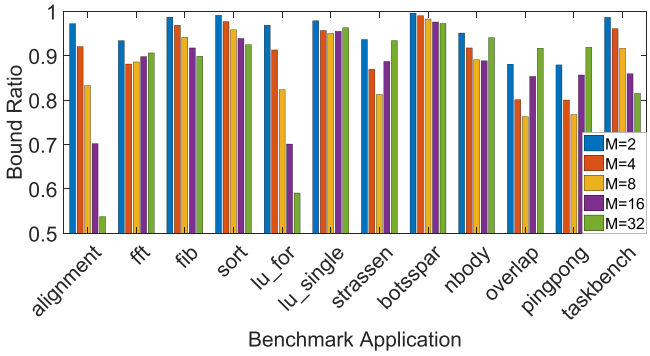


Fig. 6. Bound ratio with benchmark applications.

analysis is applied to the workload models of the applications (i.e., the task graphs) derived from their source codes, rather than to their source codes directly.

We first compare the response time bound (denoted as $R_{priority}$) computed by Equation (4) and the response time bound (denoted as $R_{classic}$) computed in Equation (1). We use a metric called *bound ratio* defined as $R_{priority}/R_{classic}$ to evaluate their performance. The result is presented in Fig. 6, where the horizontal axis is benchmarks listed in Table 1 and the vertical axis is the bound ratio. We conduct experiments with different core numbers and for each core number, we compute $R_{priority}$, $R_{classic}$ and bound ratio of each benchmark application. Experiments show that the response time bound of our scheduling algorithm is always smaller than the classic bound computed in Equation (1).

Second, we compare the required core number when changing the period of the benchmark applications. The result is presented in Fig. 7. The horizontal axis is the ratio T/L, where T is the period of benchmark applications and L is the length of the longest path of the DAG generated by benchmark applications. Here, we assume implicit deadline and use T as the deadline of the application. The vertical axis is *core number ratio*, which is defined as $M_{priority}/M_{classic}$. For our scheduling algorithm, we conduct a binary search to find the minimum core number (denoted as $M_{priority}$) such that the response time bound is less than the deadline. We compare it with the core number (denoted as $M_{classic}$) computed by the following equation, which is a reformulation of Equation (1).

$$M_{classic} \leq \left\lceil \frac{vol(G) - len(G)}{T - len(G)} \right\rceil. \tag{9}$$

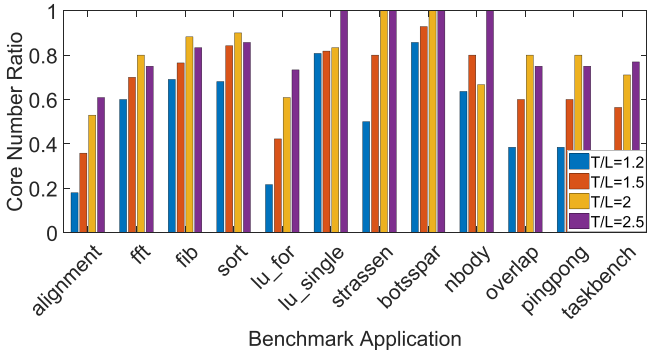


Fig. 7. Required core number with benchmark applications.

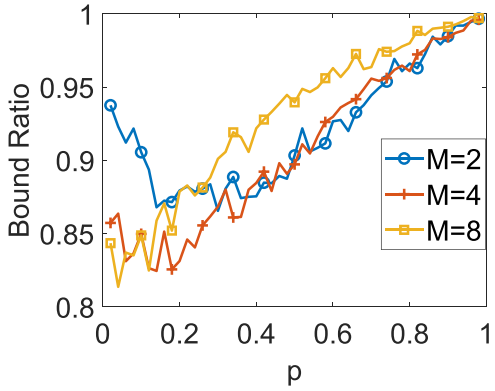


Fig. 8. Bound ratio with random tasks.

Fig. 7 shows that our method always outperforms the result in Equation (9) in terms of the required core number, especially in the case of tight deadlines. However, we also remark that when the deadline is much larger than the length of the longest path, the required core numbers of the two scheduling algorithms are almost the same.

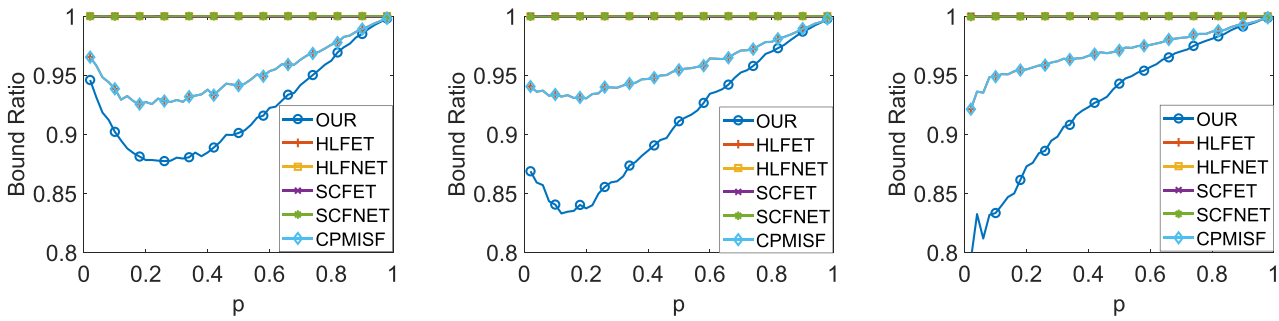
Since the benchmark applications only represent limited types of DAG, we also evaluate our scheduling algorithm by using randomly generated tasks. The task sets are generated using the Erdos-Renyi method $G(n, p)$ [17]. We still use the bound ratio to evaluate the performance.

We conduct experiments with different p in $G(n, p)$. The range of p is [0.02, 0.98]. For each p, we randomly choose n (the number of vertices in the DAG task) in [50, 250], and randomly choose C_i (the WCET of vertices in the DAG task) in [50, 100]. We compute $R_{priority}$ and $R_{classic}$ with core number 2, 4, 8. Under each configuration, we conduct 500 experiments to compute the average bound ratio. The results are presented in Fig. 8. In Fig. 8, since the bound ratio is always smaller than one, our method always outperforms the classic bound in Equation (1). The bound ratio increases as p increases and finally reaches 1 when p approaches 1. This is because the larger p, the more sequential are the generated task graphs (the more precedence constraints in the task graph). In the extreme case, there is a precedence constraint between any pair of vertices when $p = 1$, so there is no room to adjust the intra-task vertex priority order to improve the response time bound.

In summary, for the scheduling of one parallel task on a multi-core platform, our proposed scheduling algorithm consistently outperforms the classic result in Equation (1).

As we mentioned in Section 1, there are existing works on priority assignment for *static* scheduling of DAGs (while this paper focuses on *dynamic* scheduling of DAGs). These existing priority assignment strategies, although not designed for our target problem, are also applicable to our problem model. In the following, we compare the performance of priority assignment algorithm presented in Section 6 (denoted as OUR) with these existing priority assignment algorithms, including

- Highest Level First with Estimated Times (LFET) [9],
- Highest Levels First with No Estimated Times (HLFNET) [9],
- Smallest Co-levels First with Estimated Times (SCFET) [9],



(a) M=2 (b) M=4 (c) M=8

Fig. 9. Bound ratio with random tasks.

- Smallest Co-levels First with No Estimated Times (SCFNET) [9],
- Critical Path/Most Immediate Successors First (CPMISF) [10].

These priority assignment strategies all satisfy the property that the generated priority order among vertices does not conflict with their topology order, so Theorem 3 can be applied to compute the response time bounds. The settings in Fig. 9 are the same as settings in Fig. 8.

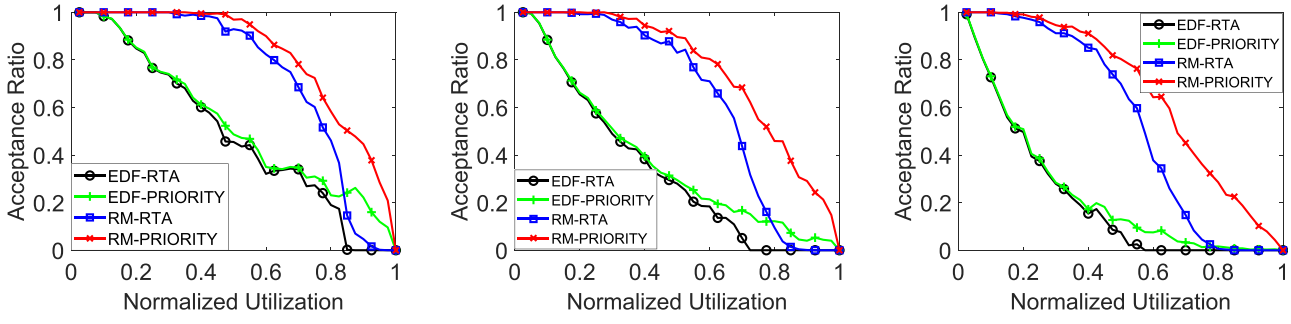
Fig. 9 shows the bound ratio (with respect to Graham's bound without exploring the intra-task priority order) between our priority assignment strategy and the compared existing priority assignment strategies. From the experiment results, we can see that our priority assignment strategy significantly outperforms all the others. In particular, some bound ratio of the existing priority assignment strategy is almost always 1 (i.e., they do not bring any benefit for reducing the response time bound). The bound ratio of all the priority assignment strategies (including ours) increases as p increases and finally converges to 1 when p is close to 1. As we mentioned above, this is because the larger p, the more sequential are the generated task graphs (the more precedence constraints in the task graph), and thus the less room to adjust the intra-task vertex priority order to improve the response time bound.

8.2 Evaluation of Scheduling Multiple Tasks

In this section, we evaluate the performance of scheduling multiple DAG tasks by using benchmark task set and randomly generated task set. We use the metric *acceptance ratio* to compare the results presented in Theorems 4 and 5 when applied to two global scheduling algorithms: EDF and RM. In detail, Theorem 4 with global EDF (denoted as EDF-RTA) and Theorem 5 (denoted as EDF-PRIORITY) with global EDF are compared; Theorem 4 with global RM (denoted as RM-RTA) and Theorem 5 with global RM (denoted as RM-PRIORITY) are compared.

The DAG of benchmark and the randomly generated DAG task are generated using the same method as Section 8.1. The period T (which is also the deadline) is randomly chosen from the interval $[L, 6L]$, where L is the length of the longest path. And for random task sets, the range of n and p in $G(n, p)$ is [50, 250] and [0.02, 0.2], respectively. The range of C_i (the WCET of vertices in DAG tasks) is [50, 100]. Benchmark applications that we use are listed in Table 1. To generate a benchmark task set with a specific utilization, we randomly choose benchmark applications in Table 1 until the total utilization reaches the required value. Similarly, to generate a random task set with a specific utilization, we randomly generate a DAG task and add it to the task set until the total utilization reaches the required value. For every parameter configuration, we generate 500 task sets to compute the average acceptance ratio. The results are presented in Figs. 10 and 11. Experiments show that with respect to acceptance ratio, our method is better than the method presented in [6], especially for benchmark task sets with high utilization. Note that there are differences between the results of benchmark and random task sets. This is because the benchmark applications usually have higher $vol(G)/len(G)$ than randomly generated tasks, which leads to a smaller number of tasks in benchmark task sets.

We also conduct experiments with different p in $G(n, p)$. The range of p is (0, 1). For each p, we set the core number to be 16, randomly choose n (the number of vertices in the DAG task) in [50, 250], randomly choose C_i (the WCET of



(a) M=8 (b) M=16 (c) M=32

Fig. 10. Acceptance ratio with benchmark task sets.

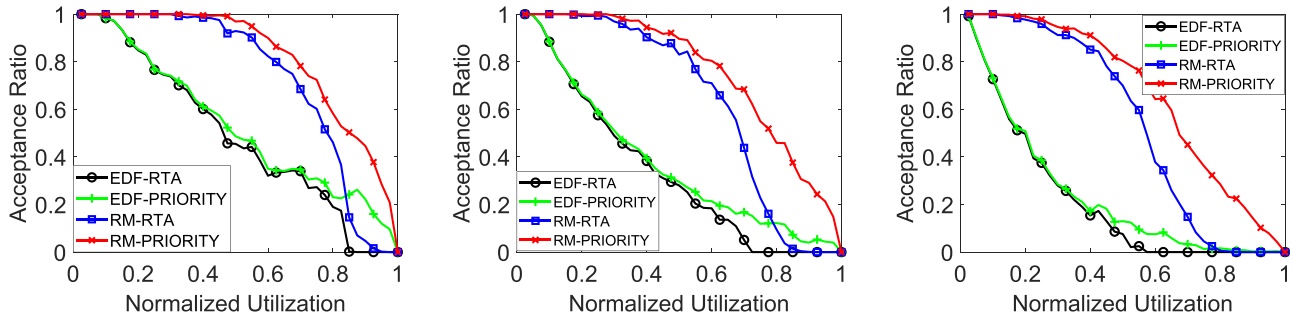


Fig. 11. Acceptance ratio with random task sets.

vertices in the DAG task) in [50, 100], and randomly choose normalized utilization in [0.1, 1] to generate task sets. Under each configuration, we generate 1,000 task sets to compute the average acceptance ratio. The results are presented in Fig. 12, which shows that with respect to acceptance ratio, our method outperforms the method presented in [6], especially for tasks with small p, which means these tasks have a higher degree of parallelism.

In the following, we conduct experiments to evaluate the effects of overhead concerning sorting vertices with different priorities. Theoretically, our method performs better than [6]. However, our method requires a priority queue to store the eligible vertices (while one can use FIFO queues if the intra-task priority order is not exploited). The operations with priority queues typically incur higher overhead than FIFO queues. Therefore, it is possible that the benefit of our method will fade in the presence of large extra runtime overhead incurred by the priority queues. In general, the larger extra overhead incurred by the priority queues (relative to the original execution time of the task graph), the less effective is our proposed method. In the following, we discuss the extra overhead incurred by the priority queues and conduct experiments with different degrees of extra overheads.

The worst-case overhead for operating the priority queues depends on the maximal number of eligible vertices at any time point (instead of the total number of vertices of the task graph). The maximal number of eligible vertices is bounded by the parallelism of the task graph, which is typically much smaller than the total number of task graphs. Therefore, we could argue that the extra overhead incurred by using the priority queues typically should be very small.

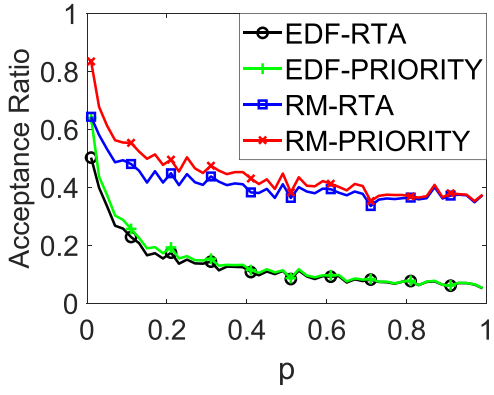


Fig. 12. Acceptance ratio with different p.

Nevertheless, the extra overhead incurred by the priority queues still negatively affect the performance of our proposed approach. In the following, we conduct experiments to quantitatively evaluate the gain and loss of our approach when the extra overhead is counted. The performance loss of our approach depends on the ratio between the runtime overhead and the original execution time of the task graph, instead of the absolute value of the runtime overhead. Therefore, we conduct experiments with changing values of this ratio, as shown in Fig. 13 (the x-axis is the ratio between the runtime overhead and maximal execution time among all vertices in the task graph). The experiment results show that our proposed approach is still rather effective when this ratio is below 10 percent.

9 Related Work

In this section, we review closely related work on real-time scheduling, concentrating primarily on parallel tasks.

For the response time bound of a parallel task, the classic bound in [2] is widely used in many literatures, including response time analysis of parallel task set, such as [6], [18] and federated scheduling, such as [5], [7], [19]. Besides the classic result, Ozaktas et al. [1] proposed techniques to compute an upper bound on the stall time due to synchronization. Voudouris et al. [20] proposed a timing-anomaly free scheduling algorithm, and by simulating this algorithm, a response time bound can be obtained. However, this algorithm requires too much modification to the existing systems, which is generally unacceptable. Besides, the response time bound is obtained by simulating the scheduling algorithm, which is not flexible and cannot be widely integrated into other analytical techniques, such as those in [6].

For response time analysis of parallel task set, Chwa et al. [21] proposed an RTA-based schedulability test for global EDF scheduling of synchronous tasks with constrained deadlines. Axer et al. [22] proposed an RTA-based schedulability analysis for fork-join tasks with arbitrary deadlines. Qamhieh et al. [23] proposed a schedulability test for global EDF scheduling of DAG task set with constrained deadlines. Maia et al. [24] proposed an RTA-based schedulability analysis for global fixed-priority scheduling of synchronous tasks with constrained deadlines. Melani et al. [6] proposed an RTA-based schedulability test for the general sporadic conditional DAG task set, and Fonseca et al. [18] improved the schedulability test in [6] by reducing the carry-in and carry-out interfering workload.

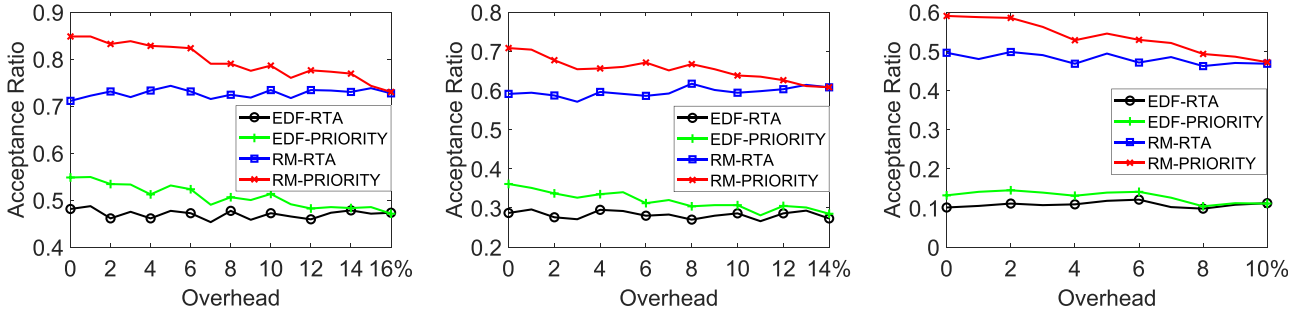


Fig. 13. Acceptance ratio with random task sets counting overheads.

Besides research work from real time community, there are plenty of techniques concerning scheduling task graphs on multiprocessor platform. Some existing work [9], [10] considered priority assignment for *static* scheduling algorithms for DAGs. These priority assignment algorithms can also be applied to *dynamic* scheduling algorithms which is the focus of this paper. However, as they are not designed for the purpose of optimizing the worst-case response time bound, their performance is not competitive with our proposed approach as shown by the experiment results in Section 8. Kwok and Ahmad proposed a static scheduling algorithm for allocating task graphs to fully connected multiprocessor based on the critical path of task graphs [25]. This model considers communication cost and supposes processor number is not given before scheduling, which is different from the model discussed in this paper. Sheikh and Ahmad proposed a task scheduling method for simultaneous optimization of performance, energy, and temperature [26], which did not involve techniques of statically assigning priorities to vertices of task graph.

Plenty of parallel benchmarks are published over years. Typical parallel applications, such as Laplace equation solver [27], fast Fourier transform (FFT) [28], LU-decomposition [29], are commonly used in evaluating parallel platforms. NAS Parallel Benchmarks [30] was developed for the performance evaluation of highly parallel supercomputers. Its Fortran-MPI version [31] and OpenMP version [32] are also widely used. SPEComp [33] targets mid-size parallel servers and includes a number of science, engineering and data processing applications. Barcelona OpenMP Task Suite (BOTS) [11] is a task-parallel benchmark suite with the purpose of testing different implementations of OpenMP tasks on multicore architectures. SPEC2012 [12] includes a set of scientific and engineering applications. The openmpmpi [14] benchmark is a set of microbenchmarks for mixed-mode programming. The openmpbench [15] contains a set of tests which measure the overhead of various OpenMP constructs. Dash [13] is a benchmark suite for hybrid dataflow and shared memory programming models. In this paper, we collect some of these benchmarks and transform them into task graphs to evaluate our schedulability method.

10 Conclusion

In this paper, by assigning priorities to vertices of the DAG, we derive a tighter response time bound of a parallel task. We propose a dynamic programming algorithm to compute this bound, and a priority assignment algorithm to assign priorities to vertices of the DAG. Besides, we extend our result to the scheduling of multiple DAG tasks. Experiments with realistic benchmark applications and randomly generated tasks show that our method consistently outperforms the state-of-the-art methods under different parameter configurations. In the future, we would like to investigate the relation between efficiently computing the response time bound derived in this paper and priority assignment policy to provide a tighter response time bound.

Acknowledgments

We thank the anonymous reviewers for their valuable suggestions to our work. This work is supported by the Research Grants Council of Hong Kong (GRF 15204917 and GRF 15213818), NSF grant CNS-1850851, NSFC grant 61532007 and 61672140 and the Ministry of Education Joint Foundation for Equipment Pre-Research under grant 6141A020333, and in the Fundamental Research Funds for the Central Universities under grant N172304025.

References

[1] H. Ozaktas, C. Rochange, and P. Sainrat, "Automatic WCET analysis of real-time parallel applications," in *Proc. 13th Int. Workshop Worst-Case Execution Time Anal.*, 2013, Art. no. 11.

[2] R. L. Graham, "Bounds on multiprocessing timing anomalies," *SIAM J. Appl. Math.*, vol. 17, no. 2, pp. 416–429, 1969.

[3] T. Lundqvist and P. Stenstrom, "Timing anomalies in dynamically scheduled microprocessors," in *Proc. 20th IEEE Real-Time Syst. Symp.*, 1999, pp. 12–21.

[4] J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, and B. Becker, "A definition and classification of timing anomalies," in *Proc. 6th Int. Workshop WCET Analysis*, 2006, pp. 23–28.

[5] J. Li, J.-J. Chen, K. Agrawal, C. Lu, C. Gill, and A. Saifullah, "Analysis of federated and global scheduling for parallel real-time tasks," in *2014 26th Euromicro Conf. on Real-Time Systems*, pp. 85–96, jul. 2014.

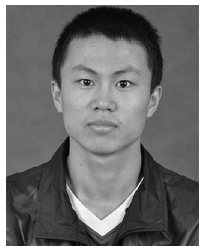
[6] A. Melani, M. Bertogna, V. Bonifaci, A. Marchetti-Spaccamela, and G. C. Buttazzo, "Response-time analysis of conditional DAG tasks in multiprocessor systems," in *Proc. 27th Euromicro Conf. Real-Time Syst.*, 2015, pp. 211–221.

[7] X. Jiang, N. Guan, X. Long, and W. Yi, "Semi-federated scheduling of parallel real-time tasks on multiprocessors," In *2017 IEEE Real-Time Systems Symposium (RTSS)*, pp. 80–91, Dec. 2017.

[8] S. Baruah, V. Bonifaci, and A. Marchetti-Spaccamela, "The global EDF scheduling of systems of conditional sporadic DAG tasks," in *Proc. 27th Euromicro Conf. Real-Time Syst.*, 2015, pp. 222–231.

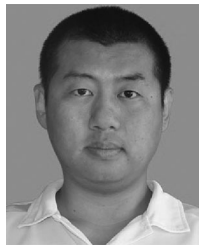
[9] Y.-K. Kwok and I. Ahmad, "Static scheduling algorithms for allocating directed task graphs to multiprocessors," *ACM Comput. Surveys*, vol. 31, no. 4, pp. 406–471, 1999.

[10] H. Kasahara and S. Narita, "Practical multiprocessor scheduling algorithms for efficient parallel processing," *IEEE Trans. Comput.*, vol. C-33, no. 11, pp. 1023–1029, Nov. 1984.
[11] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguade, "Barcelona OpenMP tasks suite: A set of benchmarks targeting the exploitation of task parallelism in OpenMP," in *Proc. Int. Conf. Parallel Process.*, 2009, pp. 124–131.
[12] M. S. Müller, J. Baron, W. C. Brantley, H. Feng, D. Hackenberg, R. Henschel, G. Jost, D. Molka, C. Parrott, J. Robichaux, et al., "SPEC OMP2012—An application benchmark suite for parallel systems using OpenMP," in *Proc. Int. Workshop OpenMP*, 2012, pp. 223–236.
[13] V. Gajinov, S. Stipić, I. Erić, O. S. Unsal, E. Ayguadé, and A. Cristal, "DaSH: A benchmark suite for hybrid dataflow and shared memory programming models: With comparative evaluation of three hybrid dataflow models," in *Proc. 11th ACM Conf. Comput. Frontiers*, 2014, Art. no. 4.
[14] J. M. Bull, J. P. Enright, and N. Ameer, "A microbenchmark suite for mixed-mode OpenMP/MPI," in *Proc. Int. Workshop OpenMP*, 2009, pp. 118–131.
[15] J. M. Bull, F. Reid, and N. McDonnell, "A microbenchmark suite for OpenMP tasks," in *Proc. Int. Workshop OpenMP*, 2012, pp. 271–274.
[16] Y. Wang, N. Guan, J. Sun, M. Lv, Q. He, T. He, and W. Yi, "Benchmarking OpenMP programs for real-time scheduling," in *Proc. IEEE 23rd Int. Conf. Embedded Real-Time Comput. Syst. Appl.*, 2017, pp. 1–10.
[17] D. Cordeiro, G. Mounié, S. Perarnau, D. Trystram, J.-M. Vincent, and F. Wagner, "Random graph generation for scheduling simulations," in *Proc. 3rd Int. ICST Conf. Simul. Tools Techn.*, 2010, Art. no. 60.
[18] J. Fonseca, G. Nelissen, and V. Nélis, "Improved response time analysis of sporadic DAG tasks for global FP scheduling," in *Proc. 25th Int. Conf. Real-Time Netw. Syst.*, 2017, pp. 28–37.
[19] S. Baruah, "The federated scheduling of systems of conditional sporadic DAG tasks," in *Proc. 12th Int. Conf. Embedded Softw.*, 2015, pp. 1–10.
[20] P. Voudouris, P. Stenström, and R. Pathan, "Timing-anomaly free dynamic scheduling of task-based parallel applications," in *Proc. IEEE Real-Time Embedded Technol. Appl. Symp.*, 2017, pp. 365–376.
[21] H. S. Chwa, J. Lee, K.-M. Phan, A. Easwaran, and I. Shin, "Global EDF schedulability analysis for synchronous parallel tasks on multicore platforms," in *Proc. 25th Euromicro Conf. Real-Time Syst.*, 2013, pp. 25–34.
[22] P. Axer, S. Quinton, M. Neukirchner, R. Ernst, B. Döbel, and H. Härtig, "Response-time analysis of parallel fork-join workloads with real-time constraints," in *Proc. 25th Euromicro Conf. Real-Time Syst.*, 2013, pp. 215–224.
[23] M. Qamhieh, F. Fauberteau, L. George, and S. Midonnet, "Global EDF scheduling of directed acyclic graphs on multiprocessor systems," in *Proc. 21st Int. Conf. Real-Time Netw. Syst.*, 2013, pp. 287–296.
[24] C. Maia, M. Bertogna, L. Nogueira, and L. M. Pinho, "Response-time analysis of synchronous parallel tasks in multiprocessor systems," in *Proc. 22nd Int. Conf. Real-Time Netw. Syst.*, 2014, Art. no. 3.
[25] Y.-K. Kwok and I. Ahmad, "Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 7, no. 5, pp. 506–521, May 1996.
[26] H. F. Sheikh and I. Ahmad, "Dynamic task graph scheduling on multicore processors for performance, energy, and temperature optimization," in *Proc. Int. Green Comput. Conf.*, 2013, pp. 1–6.
[27] M.-Y. Wu and D. D. Gajski, "Hypertool: A programming aid for message-passing systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 1, no. 3, pp. 330–343, Jul. 1990.
[28] V. A. Almeida, I. Vasconcelos, J. N. C. Arabe, and D. A. Menascé, "Using random task graphs to investigate the potential benefits of heterogeneity in parallel systems," in *Proc. ACM/IEEE Conf. Supercomput.*, 1992, pp. 683–691.
[29] R. E. Lord, J. Kowalik, and S. Kumar, "Solving linear algebraic equations on an MIMD computer," *J. ACM*, vol. 30, no. 1, pp. 103–117, 1983.
[30] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, et al., "The NAS parallel benchmarks summary and preliminary results," in *Proc. ACM/IEEE Conf. Supercomputing*, 1991, pp. 158–165.
[31] D. Bailey, T. Harris, W. Saphir, R. Van Der Wijngaart, A. Woo, and M. Yarrow, "The NAS parallel benchmarks 2.0," NASA Ames Res. Center, Mountain View, CA, USA, Tech. Rep. NAS-95–020, 1995.
[32] H.-Q. Jin, M. Frumkin, and J. Yan, "The OpenMP implementation of NAS parallel benchmarks and its performance," Technical Report NAS-99-011, NASA Advanced Supercomputing Division, Moffett Field, CA, USA, Oct. 1999.
[33] V. Aslot, M. Domeika, R. Eigenmann, G. Gaertner, W. B. Jones, and B. Parady, "SPEComp: A new benchmark suite for measuring parallel computer performance," in *Proc. Int. Workshop OpenMP Appl. Tools*, 2001, pp. 1–10.



Qingqiang He received the BS degree in computer science and technology from Northeastern University, China, in 2014, and the MS degree in computer software and theory from Northeastern University, China, in 2017. Now he is working toward the PhD degree at Hong Kong Polytechnic University. His research interests include embedded real-time system, real-time scheduling theory, and distributed ledger.

Xu Jiang received the BS degree in computer science from Northwestern Polytechnical University, China, in 2009, the MS degree in computer architecture from the Graduate School of the Second Research Institute of China Aerospace Science and Industry Corporation, China, in 2012, and the PhD degree in computer architecture from Beihang University, China, in 2018. His research interests include real-time systems, parallel and distributed systems, and embedded systems
.



Nan Guan received the BE and MS degrees from Northeastern University, China, in 2003 and 2006, respectively, and the PhD degree from Uppsala University, Sweden, in 2013. He is currently an assistant professor with the Department of Computing, Hong Kong Polytechnic University. Before joining PolyU in 2015, he worked as a faculty member with Northeastern University, China. His research interests include real-time embedded systems and cyber-physical systems. He received the EDAA Outstanding Dissertation Award in 2014, the Best Paper Award of IEEE Real-time Systems Symposium (RTSS) in 2009, the Best Paper Award of Conference on Design Automation and Test in Europe (DATE) in 2013.

Zhishan Guo received the PhD degree from the University of North Carolina at Chapel Hill, in 2016. He is an assistant professor with the Department of Electrical and Computer Engineering, University of Central Florida. His research and teaching interests include real-time scheduling, machine learning, and their applications in cyber-physical systems.