Partitioning-Based Scheduling of OpenMP Task Systems With Tied Tasks

Yang Wang, Xu Jiang, Nan Guan, Zhishan Guo, Xue Liu, and Wang Yi, *Fellow, IEEE*

Abstract—OpenMP is a popular programming framework in both general and high-performance computing and has recently drawn much interest in embedded and real-time computing. Although the execution semantics of OpenMP are similar to the DAG task model, the constraints posed by the OpenMP specification make them significantly more challenging to analyze. A `tied` task is an important feature in OpenMP that must execute on the same thread throughout its entire life cycle. A previous work [1] succeeded in analyzing the real-time scheduling of `tied` tasks by modifying the Task Scheduling Constraints (TSCs) in OpenMP specification. In this article, we also study the real-time scheduling of OpenMP task systems with `tied` tasks but without changing the original TSCs. In particular, we propose a partitioning-based algorithm, P-EDF-omp, by which the `tied` constraint can be automatically guaranteed as long as an OpenMP task system can be successfully partitioned to a multiprocessor platform. Furthermore, we conduct comprehensive experiments with both synthetic workloads and established OpenMP benchmarks to show that our approach consistently outperforms the work in [1]—even without modifying the TSCs.

Index Terms—Multicore, parallel tasks, real-time scheduling, partitioning, OpenMP, `tied` tasks

1 Introduction

Real-time systems are shifting from single-core to multicore processors to meet the rapidly increasing requirements of high performance and low power consumption. Software must be parallelized to fully utilize the computation power of multicore processors. OpenMP [2], the de facto parallel programming framework for shared memory architectures in both general and high-performance computing domains, is gaining increasing attention for use in embedded platforms [3], [4], [5], [6], [7], [8], [9], [10].

Using Directed Acyclic Graphs (DAG) to model parallel workloads is a common way in real-time analysis. OpenMP has supported explicit tasks since version 3.0, and its execution semantics are quite similar to the DAG model, which has motivated much theoretical work on the real-time scheduling and analysis of DAG task models [11], [12], [13], [14], [15].

- *Yang Wang and Xu Jiang are with the Northeastern University, Shenyang 110819, China. E-mail: {wy09e15, jiangxu617}@163.com.*
- *Nan Guan is with the Hong Kong Polytechnic University, Hong Kong. E-mail: nan.guan@polyu.edu.hk.*
- *Zhishan Guo is with the University of Central Florida, Orlando, FL 32816 USA. E-mail: zsguo@ucf.edu.*
- *Xue Liu is with the McGill University, Montreal, QC H3A 0G4, Canada. E-mail: xueliu@cs.mcgill.ca.*
- *Wang Yi is with the Northeastern University, Shenyang 110819, China, and also with the Uppsala University, 752 36 Uppsala, Sweden. E-mail: yi@it.uu.se.*

Manuscript received 30 Oct. 2019; revised 3 Dec. 2020; accepted 28 Dec. 2020. Date of publication 31 Dec. 2020; date of current version 28 Jan. 2021. (Corresponding author: Xu Jiang.)
Recommended for acceptance by M. Becchi.
Digital Object Identifier no. 10.1109/TPDS.2020.3048373

A `tied` task is an important feature in OpenMP task systems. In OpenMP, the tasks are `tied` by default, unless an `untied` keyword is explicitly placed. `Tied` task forces a task to execute on the same thread throughout its entire life cycle without migrating to another thread. In particular, if the execution of a `tied` task is interrupted, it must be resumed on the same thread later. In addition, the OpenMP specification poses special constraints on the execution of `tied` tasks, called Task Scheduling Constraints (TSCs), which also need to be taken into account while scheduling `tied` tasks. Therefore, the existing results using DAG models cannot be directly applied to OpenMP task systems with `tied` tasks because the DAG models cannot fully capture the constraints of `tied` tasks posed by the OpenMP specification.

Despite the constraints on `tied` tasks, `tied` tasks enjoy the following benefits [4], [5], [16] because they preclude migrations among threads: (1) a `tied` task simplifies the implementation of the scheduling algorithm and reduces context switching costs; (2) in many cases, a `tied` task can help reduce the difficulty of avoiding deadlocks in the presence of critical sections; (3) a `tied` task can help make library functions thread-safe. Meanwhile, situations still exist when developers must use `tied` tasks instead of `untied` tasks. OpenMP has always been thread-centric before OpenMP 3.0. Threads provide a very useful abstraction of processors, and developers have capitalized on this capability. Threadprivate storage, threadspecific features and thread-local storage provided by the native threading package or the linker are all useful for making library functions thread-safe. However, employing threadprivate variables or anything dependent on thread ID is strongly discouraged in `untied` tasks [16]. In contrast, it is easier and more predictable to use this information in `tied` tasks. Therefore, `tied` tasks are essential in OpenMP programming.

There is not much work focusing on analyzing the real-time scheduling of OpenMP task systems with `tied` tasks. Sun *et al.* proposed the first guaranteed response time bound for the OpenMP task system with `tied` tasks in [1], under a scheduling algorithm called BFS*. BFS* *modifies* the original TSCs posed in the OpenMP specification to mitigate the `tied` task scheduling problem. However, how to use the *original TSCs* to schedule and analyze OpenMP task systems with `tied` tasks, which can provide hard real-time guarantees, is vastly open.

To address the above problem, we propose an effective algorithm called P-EDF-omp, which is a partitioning-based multiprocessor scheduling algorithm for OpenMP-DAGs. We first decompose the OpenMP-DAG into subtasks corresponding to the vertices using the existing decomposition strategy from [17]. These subtasks have their own release times and deadlines. Then at design time, the Subtask Assignment Procedure (SAP) in P-EDF-omp partitions every subtask to a dedicated processor. Next at runtime, each processor uses the non-preemptive earliest-deadline-first algorithm (EDF_{np}) to schedule the subtasks that have been assigned to it. In this study, we prove that all the subtasks can automatically meet their deadlines when scheduled by EDF_{np} on their dedicated processors, if they were successfully assigned to the processors by the SAP in P-EDF-omp. Thus, the SAP can be used as an off-line schedulability-test for OpenMP-DAGs with `tied` tasks.

We conduct experiments under both synthetic workloads and established OpenMP benchmarks to evaluate the performance of our schedulability-test. The experimental results show that P-EDF-omp outperforms the BFS* algorithm proposed in [1] under different parameter configurations in terms of the acceptance ratio.

2 Related Work

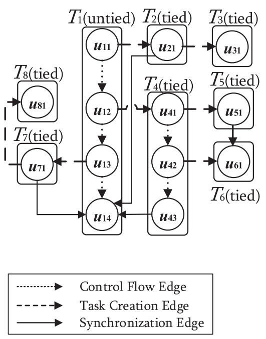
OpenMP4 [2], the de-facto standard for shared memory parallel programming in high-performance computing (HPC), has recently gained much attention in the embedded and real-time domains [1], [3], [4], [5], [7], [18], [19], [20], [21] due to its capability to define explicit subtasks and the data dependencies existing among them. This capability allows very sophisticated types of fine-grained and irregular parallelism to be expressed. Moreover, OpenMP is supported in the newest multicore embedded architectures and has become a firm candidate for developing future real-time embedded systems.

The authors of [22] conducted an evaluation of different scheduling policies using their run-time system Nanos++ [23] and analyzed the differences existing between `tied` and `untied` tasks from an average performance point of view. In addition, the average-case performance analysis of OpenMP applications is discussed in [24], [25], [26].

The first attempt to apply OpenMP4 was introduced in [4], where the authors studied how to construct an OpenMP task graph that contains sufficient information for real-time DAG scheduling models to be applied. Then, timing guarantees can be derived from the task graph with considering the tasking semantics of OpenMP4. Serrano *et al.* [5] provided the first response time bound analysis for the OpenMP DAG task model with `untied` tasks and

```
1.  #pragma omp parallel num_threads (m)
2. {   #pragma omp single
3.    {   #pragma omp task untied// T1
4.       {   block11;   // u11
5.          #pragma omp task// T2
6.          {  block21; // u21
7.             #pragma omp task// T3
8.             {   block31;  }// u31
9.          }
10.         block12;   // u12
11.         #pragma omp task // T4
12.         {   block41; // u41
13.            #pragma omp task depend(out:x)// T5
14.            {   block51; }//u51
15.            block42; //u42
16.            #pragma omp task depend(in:x) // T6
17.            {   block61;    }//u61
18.            block43;  //u43
19.         }
20.         block13;   //u13
21.         #pragma omp task// T7
22.         {   block71; //u71
23.            #pragma omp task// T8
24.            {   block81;  }// u81
25.         }
26.         #pragma omp taskwait
27.         block14;  }//u14
28.   }}
```

(a) OpenMP program.



(b) OpenMP-DAG.

Fig. 1. An example of OpenMP program and OpenMP-DAG.

pointed out that when `tied` tasks exist, the OpenMP task system would have an unacceptably pessimistic response time bound.

Moreover, Serrano *et al.* [19] investigated the scheduling of OpenMP tasks with limited preemptions. Sun *et al.* [20] considered the conditional branches in OpenMP programs and proposed a linear-time algorithm for computing the response time bound. Serrano *et al.* [7] analyzed the response time for an OpenMP task system supporting heterogeneous multicores. However, none of these works have considered `tied` tasks. Sun *et al.* proposed the first guaranteed response time bound for the OpenMP task system with `tied` tasks in [1]. However, they modified the original Task Scheduling Constraints (TSCs) posed by the OpenMP specification, while in this paper we use the original TSCs without modifying them.

3 Overview of OpenMP Programs

With OpenMP, one can design parallel tasks that are either implicit tasks (e.g., omp loop) or explicit tasks (omp task). In this paper, we consider only OpenMP 3.0 or higher versions, which support the task directive.[1]

3.1 OpenMP Threads

An OpenMP program starts with a `parallel` directive (e.g., Line 1 in Fig. 1a), which constructs an associated `parallel` region that includes all the codes enclosed in a pair of brackets following the `parallel` directive (e.g., Lines 2–27 in Fig. 1a). The `parallel` directive creates a team of m OpenMP threads (m is specified with the `num_threads` clause). In OpenMP, the execution entity for executing tasks is called a thread (which equates to a thread in the underlying OS). Similar to previous works [4], [5], each thread is assumed to exclusively execute on a dedicated processor (i.e., the *OMP_PROC_BIND* [2] variable is set to be "true"). In

1. For simplicity, we focus only on explicit OpenMP tasks, which are annotated by the task directive. The implicit OpenMP tasks related to work-sharing directives are out of the scope of this paper.

2. The OMP_PROC_BIND is an OpenMP environment variable: if it is set to be true, OpenMP threads do not move among processors; otherwise, OpenMP threads may move among processors.

the rest of this paper, the concept of *"processor"* is equivalent to the thread executing on it and we use these two terms interchangeably. The general case in which a processor is bound to multiple threads is out of the scope of this paper.

3.2 OpenMP Tasks

A `parallel` region can consist of a set of independent parallel units, called `OpenMP tasks`. In this paper, the term "`task`" refers to an OpenMP task. A task is created when a `task` directive is encountered (e.g., T_1, Line 3 in Fig. 1a). All the codes enclosed in the brackets following the `task` directive (e.g., the codes in Lines 4, 10, 20 and 27 belong to task T_1) form the *body* of the task.

If a task T_i is enclosed in the *body* of another task T_j, T_i is a *child* of T_j and T_j is the *parent* of T_i. Two tasks that share the same parent are *siblings*. Moreover, a task T_i is a *descendant* of T_j, if T_i is a child (or the child of child—with arbitrary levels of recursion) of T_j. In this case, T_j is an *ancestor* of T_i. For example, in Fig. 1b, task T_2 is a child of task T_1, and tasks T_5 and T_6 are siblings because they share the common parent T_4. Task T_5 is a descendant of T_1, and T_1 is an ancestor of T_5.

3.3 Task Synchronization

The two most widely used synchronization mechanisms in OpenMP are taskwait directives (e.g., Line 26 in Fig. 1a) and depend clauses (e.g., Lines 13 and 16 in Fig. 1a) as described below.

- `taskwait.` A task can synchronize with its children via `taskwait` directives. A `taskwait` directive blocks the parent task until all of its children (but not other descendants beyond children) created prior to the `taskwait` directive have completed. For example, in Fig. 1b, tasks T_2, T_4 and T_7 synchronize with T_1 through a taskwait directive: T_1 cannot complete the execution of u_{14} until tasks T_2, T_4 and T_7 complete.
- `depend.` Depend clauses impose an order between two sibling tasks. If a task has an `in` dependence on a variable, it cannot start execution until all its previously created sibling tasks with an `out` or `inout` dependences on the same variable complete. In Fig. 1b, T_5 and T_6 synchronize with each other through a depend clause, and T_6 must wait for T_5 to complete.

3.4 Runtime Constraints

The scheduling process for OpenMP tasks assigns tasks (or task vertices) onto threads, ensuring that the following OpenMP scheduling constraints are satisfied.

Task Scheduling Points (TSP). In OpenMP, a TSP is a point in a program at which execution can be interrupted and scheduling may be triggered. A TSP occurs upon task creation and completion, and at synchronization points such as `taskwait` directives.[3] TSPs divide a program into several parts (e.g., `block11` in Fig. 1a), and a TSP exists between any two adjacent parts, implying that the execution of each vertex should not be interrupted (i.e., the execution of each part is non-preemptive).

Tied Tasks. In OpenMP, a task can be either `tied` or `untied`. When a `tied` task starts execution on a thread, it will subsequently only execute on this thread throughout its entire life cycle. Specifically, if the execution of a `tied` task is interrupted, this task must later resume on the same thread. In contrast, an `untied` task can be executed on different threads. Thus, when the execution of an `untied` task is interrupted, this task can later be resumed by any thread. By default, OpenMP tasks are `tied`, unless explicitly specified as `untied`.

Task Scheduling Constraint (TSC). OpenMP enforces the task scheduling constraint [27]: *"Scheduling of new `tied` tasks is constrained by the set of task regions that are currently `tied` to the thread, and that are not suspended in a barrier region. If this set is empty, any new `tied` task may be scheduled. Otherwise, a new `tied` task may be scheduled only if it is a descendent task of every task in the set.*[4]*"* Details about this constraint will be introduced in Section 4.2.

4 Modeling

4.1 OpenMP Task Model

We consider an OpenMP task system Θ, which can be represented as a DAG $G = (V, E)$, where V represents the set of vertices, and E represents the set of edges. Θ consists of n OpenMP tasks $\{T_1, T_2, \dots, T_n\}$, and each task is either `tied` or `untied`. Specifically, we use Ψ to denote the task set that consists of all the `tied` tasks in Θ. A task T_h consists of a set of vertices $\{u_{h1}, u_{h2}, \dots, u_{hn_h}\}$,[5] and a vertex u_{hx} in V corresponds to the x-th vertex of task T_h and is associated with a worst-case execution time $c(u_{hx})$. Each OpenMP task contains a unique entry vertex and a unique exit vertex. The task system Θ is released recurrently with a period P and has an implicit deadline, i.e., $D = P$. The total worst-case execution time of all vertices of a task system Θ is denoted by $C = \sum_{u \in V} c(u)$. The utilization U of a task system Θ is defined as $U = C/P$. In this paper, we only consider task systems with $U > 1$.

Edge (u_{hx}, u_{jz}) in E denotes the precedence constraint between vertices u_{hx} and u_{jz} such that u_{jz} can execute only after u_{hx} completes. In this case, u_{hx} is called a *predecessor* of u_{jz} and u_{jz} is a *successor* of u_{hx}. u is *eligible* to be executed when all its predecessors have completed. And T_h is *eligible* if u_{h1} is *eligible*. Moreover, we call T_h an *active* task if T_h is *eligible* but has not completed the execution of all the vertices it contains.

Definition 1 (Descendant Task Set). $\Psi_{des}(T_h)$ *denotes the set of OpenMP tasks that are descendant tasks of* T_h.

Definition 2 (Preassigned Vertices). ω_{T_h} *denotes the set of every successor vertex of* u_{h1} *in `tied` task* T_h, *i.e.,* $\omega_{T_h} = \{u_{h2}, u_{h3}, \dots, u_{hn_h}\}$

3. Additional TSPs are implied by various constructs *barrier, target, taskyield, taskgroup*; however, for simplicity, we do not consider these constructs or the if/final clauses of the task directive in this paper.

4. We do not consider the barrier construct in this paper and do not include the barrier condition. In addition, we do not consider TSC4 in OpenMP4 because we do not consider if/final clauses of task directives.

5. For simplicity, we assume that OpenMP-DAGs have no conditional branches to focus on the main point of this paper: how to schedule `tied` tasks.

L denotes the sum of $c(u)$ of each vertex u on the longest chain (also called the critical path) of task system Θ, i.e., the execution time of the task system Θ to be exclusively executed on an infinite number of processors. L can be computed in linear time with respect to the size of the DAG [28]. The laxity of Θ is $(P-L)$. We use Υ to denote the elasticity of Θ, which is defined as $\Upsilon = L/P$. Apparently, when a task set Θ is schedulable on a multicore platform composed of m identical processors, the following conditions must hold:

$$L \leq P \quad and \quad U \leq m.$$

There are three types of edges in a graph, i.e., $E = E_1 \cup E_2 \cup E_3$, as detailed below.

Control Flow Edges (E_1), denoted by dotted-line arrows in Fig. 1b, model the control flow dependencies within a task.

Task Creation Edges (E_2) are denoted by dashed-line arrows in Fig. 1b. A parent task points its child tasks via this type of edges.

Synchronization Edges (E_3) are denoted by solid-line arrows in Fig. 1b. There are two synchronization edge subtypes that correspond to the `taskwait` directives and `depend` clauses.

Notice 1. *The P-EDF-omp algorithm in this paper treats all edges in the same way. In other words, from the viewpoint of our scheduling algorithm, these three types of edges are all equivalent in the OpenMP-DAG. Hence in the rest of this paper, the figures use only a single type of edge (solid-line arrows).*

Fig. 1b shows the OpenMP-DAG which corresponds to the OpenMP program in Fig. 1a.

4.2 OpenMP Runtime Model

Given a task graph $G = (V, E)$ and a team of processors $S = \{s_1, \ldots, s_m\}$ (recall that the concept of *"processor"* is equivalent to the thread executing on it), a schedule is to assign tasks to processors such that each vertex of V can be executed until completion.

Definition 3 (Tied Task Set). $\Gamma_k(t)$ *denotes the set of* active *tasks that have been tied to processor* s_k *before time* t.

As introduced in Section 3.4, according to the OpenMP specification, the OpenMP task scheduling must fulfill several constraints at runtime. We concentrate on the following three constraints in this paper. The formal statements for these constraints are as follows.

- *Task Scheduling Points (TSP):* For $\forall u_{hx} \in V$, once the execution starts, it cannot be interrupted until completion (but task execution may be preempted or suspended at vertex boundaries).
- *Tied Tasks:* For $\forall T_h \in \Psi$, if the entry vertex u_{h1} in T_h starts execution on processor s_k, then u_{h1} itself as well as $\forall u_{hi} \in \omega_{T_h}$ must be executed on s_k throughout their entire life cycle; they cannot migrate to another processor.
- *Task Scheduling Constraint (TSC):* At time t, the entry vertex u_{h1} of a `tied` task T_h can be executed by processor s_k if for $\forall T_j \in \Gamma_k(t)$, $T_h \in \Psi_{des}(T_j)$. TSC enforces that a new `tied` task T_h can be executed by s_k at time t only if T_h is a descendant of all the *active* tasks tied to s_k before t. Specifically, T_h can be executed by s_k if $\Gamma_k(t) = \emptyset$.

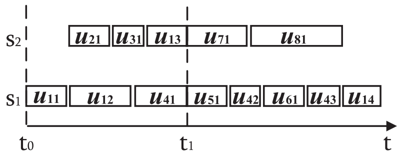


Fig. 2. An example schedule of OpenMP-DAG in Fig. 1b.

Example 1. An example schedule of OpenMP-DAG in Fig. 1b satisfying the TSC is shown in Fig. 2. At time t_1, the currently `tied` task set of s_1 is $\Gamma_1(t_1) = \{T_4\}$, and the tasks T_5 and T_7 are both eligible at time t_1. Under TSC, T_5 can be executed by s_1 but T_7 cannot because T_5 is a child of T_4 but T_7 is not.

Table 1 summarizes the notations used in this paper.

5 PARTITION

In this section, we present the P-EDF-omp algorithm, which is a partitioning-based multiprocessor scheduling algorithm for scheduling the sporadic subtask set decomposed from the OpenMP-DAG with `tied` tasks on identical multiprocessor platforms. The main idea of P-EDF-omp is based on the FBB-FFD algorithm in [11]. The FBB-FFD algorithm is a simple partitioning algorithm, which is a variant of a bin-packing heuristic known as first-fit-decreasing.

In Section 5.1, we clarify the basic procedure of our approach. In Section 5.2, we briefly introduce the decomposition strategy in [17], which we use to transfer the OpenMP-DAG into a sporadic sequential subtask set. Then, in Section 5.3, we define P-EDF-omp and state how the Subtask Assignment Procedure works.

5.1 Overview of Our Algorithm

Based on the constraint for `tied` tasks that they cannot migrate to another processor throughout their entire life cycle, we chose partitioned scheduling. In partitioned scheduling, the vertices are not allowed to migrate among processors once partitioned to one processor. In order to get the subtasks corresponding to the vertices for partitioning, we need to decompose the OpenMP-DAG first. After we determine how to partition the subtasks to processors, the processors schedule their "local" subtasks at runtime, and the problem becomes that of real-time scheduling on a single core. Due to the optimality of EDF for uniprocessors [29], [30] and the constraint that the vertex execution cannot be interrupted, we chose the non-preemptive earliest-deadline-first algorithm (EDF_{np}) as the runtime scheduler. Later in Section 6, we prove that if P-EDF-omp can successfully partition all the subtasks to the processors in the off-line phase, every subtask can automatically meet its deadline with EDF_{np} as the runtime scheduler on the corresponding processor. Hence, the goal of the proposed algorithm is to find a feasible way to partition the subtasks (corresponding to the vertices in OpenMP-DAG), which we obtain from decomposition. The algorithm can be divided into two phases: the off-line phase and the on-line phase.

- Phase 1: off-line phase

  This part consists of two steps:
  - Decomposition. In this step, we use the decomposition strategy in [17] to decompose the

TABLE 1
Notations Adopted in This Paper

Notations	Descriptions
Θ	an OpenMP task system
n	number of OpenMP tasks in Θ
T_h	an OpenMP task
n_h	number of vertices in T_h
G	the workload structure of Θ
V	the set of vertices in G
E	the set of edges in G
N	the number of vertices/subtasks in G
$c(u_{hx})$	worst-case execution time (WCET) of a vertex u_{hx}
C	total WCET of all vertices of Θ
L	the longest length among all path of Θ
D	the deadline of Θ
P	the period of Θ
U	the utilization of Θ
Υ	the elasticity of Θ
Ψ	the set of `tied` tasks in Θ
$\Psi_{des}(T_h)$	set of all descendant tasks of T_h
ω_{T_h}	the set of successors of u_{h1} in `tied` task T_h
S	the team of the processors
s_k	the k-th processor
Π_k	the set of subtasks already assigned to s_k
n_{s_k}	number of subtasks in Π_k
Dm_{t_0,t_d}	the processor demand in time interval $[t_0, t_d]$
$\Gamma_k(t)$	*active* tasks that were tied to s_k before time t
Θ_{decomp}	the resulting sequential subtask from decomposition
s^i	a segment in decomposition
τ_i	a resulting sequential subtask from decomposition
e_i	WCET of $\tau_i's$ corresponding vertex ($\tau_i's$ execution requirement)
Δ_i	the lifetime window of τ_i
d_i	length of lifetime window Δ_i
η_i^{st}/η_i^{sp}	starting and stopping time of Δ_i
iv_x	a time interval
d_{iv_x}	the length of time interval iv_x
iv_x^{st}/iv_x^{sp}	starting and stopping time instant of iv_x
$K(\tau_i, iv_x)$	The Interval Load of τ_i in time interval iv_x
$\Delta_{un}(\tau_i, \tau_j)$	Union Time Interval of lifetime windows of τ_i and τ_j
$d_{un}(\tau_i, \tau_j)$	length of the Union Time Interval
$\Delta_{un}^{st}(\tau_i, \tau_j)$	starting time instant of $\Delta_{un}(\tau_i, \tau_j)$
$\Delta_{un}^{sp}(\tau_i, \tau_j)$	stopping time instant of $\Delta_{un}(\tau_i, \tau_j)$
Λ_h	the lifetime window of T_h
ρ_h^{st}/ρ_h^{sp}	starting and stopping time instant of Λ_h
$\Gamma_k^o(t)$	`tied` task T_h assigned to s_k with $\rho_h^{st} \leq t < \rho_h^{sp}$

OpenMP-DAG (which contains `tied` tasks) into a sequential subtask set Θ_{decomp}. Each subtask corresponds to a vertex in Θ_{decomp} and has its own execution requirement (which equals the WCET of the vertex), starting and ending times of its own lifetime window.

- Partition. After obtaining the resulting sequential sporadic subtask set, we use the Subtask Assignment Procedure (SAP) to assign the subtasks to "available" processors following the runtime constraints introduced in Section 3.4, using a "first-fit" heuristic.

- Phase 2: on-line phase

  In this phase, each processor uses the non-preemptive earliest-deadline-first algorithm (EDF_{np}) as the runtime scheduler to schedule the subtasks assigned to it.

Discussion About the On-Line Phase. Although the majority of the P-EDF-omp algorithm completes during the off-line phase, we still choose on-line scheduling with EDF_{np} rather than choosing off-line scheduling for the following reasons. If the execution times of the subtasks are all constants, we could indeed create a scheduling table at design time and implement off-line scheduling. However, we can only obtain the worst-case execution time rather than the real execution time of each subtask at design time, and the worst-case execution time (WCET) may occur only in various extreme cases; the real execution time is typically less than the WCET. Thus, if we were to adopt off-line scheduling, the subtasks would have to wait even after they complete execution at runtime; otherwise, we could not guarantee that the rules in the table would be satisfied. Consequently, adopting off-line scheduling reduces processor utilization, especially when other jobs (not the ones from the OpenMP-DAG) are waiting to be scheduled in the system. However, on-line scheduling does not suffer from this problem. In general, on-line scheduling is more flexible, and it utilizes processor resources better, which makes it more popular than off-line scheduling in embedded and real-time domains. Hence, we choose to use on-line scheduling rather than off-line scheduling in P-EDF-omp.

5.2 Decomposition Strategy

In this section, we first briefly introduce the decomposition strategy in [17].

The target of OpenMP-DAG decomposition is to assign an artificial release time and deadline to each vertex, such that the dependencies among different vertices can be automatically guaranteed as long as each vertex respects its own release time and deadline constraints. Upon decomposition, an implicit deadline OpenMP task system is decomposed into a set of constrained-deadline (i.e., deadline is no greater than period) sequential subtasks, where each subtask corresponds to a vertex in the OpenMP-DAG. Moreover, this decomposition process ensures that the OpenMP-DAG will be schedulable if all its subtasks are schedulable.

We use the example in Fig. 3 to illustrate the decomposition procedure, which consists of three main steps.

5.2.1 *Segmentation*

In this step, we divide the time window between two successive releases of Θ (of length P) into several segments $\{s^1, s^2, \ldots, s^p\}$ and assign the workload of each vertex (which equals the WCET of the vertex) to these segments. A vertex may be split into several parts and assigned to different segments. We first construct a timing diagram for Θ that defines the earliest ready time of each vertex u, denoted by $rdy(u)$, and the latest finish time of u, denoted by $fsh(u)$,[6] assuming that Θ executes exclusively on a sufficient number of processors and that the entire Θ workload must be completed within L.

The segmentation algorithm consists of three steps:

- Step 1: Assign each vertex that only covers a single segment. All the segments are then classified into two types (*light* and *heavy* segments) according to the

6. If vertex u has no outgoing edges, $fsh(u) = L$.

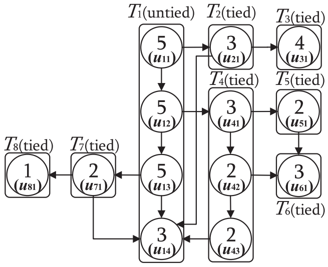


Fig. 3. OpenMP-DAG in Fig. 1b with random $c(u)$ (P=36).

ratio of the total amount of the workload of every vertex (or a part of a vertex) assigned to the segment and the segment length. For each segment, if this ratio is no greater than C/L, the segment is a light segment; otherwise, it is a heavy segment.

- Step 2: Assign the remaining vertices to *light* segments insofar as possible, without turning any *light* segment into a *heavy* segment.
- Step 3: Assign the remaining vertices, if any, to the *heavy* segments arbitrarily.

Definition 4 (Lifetime window of a vertex u_i).
Regardless of whether a vertex u_i has been split into several parts, the time interval between the starting time of segment s^p and the ending time of segment s^q is called the lifetime window *of u_i, where s^p is the segment whose starting time equals $rdy(u_i)$ in the segmentation and s^q is the segment whose ending time equals $fsh(u_i)$ in the segmentation.*

We use Δ_i to denote the lifetime window *of u_i, and η_i^{st} and η_i^{sp} to denote the times at which Δ_i starts and ends, respectively, i.e., η_i^{st} equals the starting time of s^p and η_i^{sp} equals the ending time of s^q.*

Clearly, in the segmentation, the starting and ending times of each vertex $u's$ original *lifetime window* are $rdy(u)$ and $fsh(u)$, respectively. Later in the laxity distribution, the *lifetime window* of u_i will change as the starting time of s^p and the ending time of s^q change. See [17] for more details in segmentation algorithm.

Example 2. After segmentation, the OpenMP-DAG in Fig. 3 is transformed into a timing diagram with the workload assigned as shown in Fig. 4. The *lifetime window* of u_{11} (which has not been split) is [0,5], while the *lifetime window* of $u_{12}'s$ (which has been split) is [5,10].

5.2.2 Laxity Distribution

In this step, the laxity $(P - L)$ of this OpenMP-DAG will be distributed into each segment created previously in the

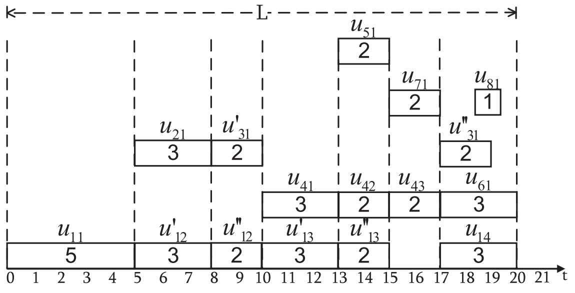


Fig. 4. A possible segmentation of OpenMP-DAG in Fig. 3.

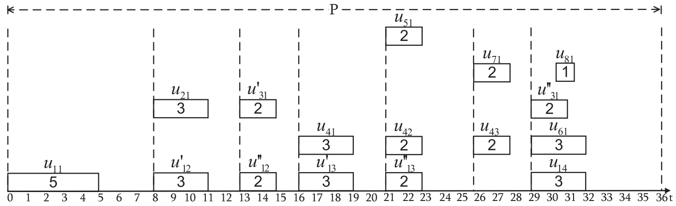


Fig. 5. Laxity distribution of example in Fig. 3 (before reassembling).

segmentation step, based on a value called the *structure characteristic value* for *light* and *heavy* segments. This means that the length of every segment will be "stretched" to be longer. Correspondingly, the *lifetime window* of each vertex changes when the starting times of the segments change. After the laxity distribution, each segment has new starting and ending times. Correspondingly, the vertices contained in the segments will have their own release times and relative deadlines. Thus, an OpenMP-DAG is transformed into a set of independent sequential sporadic subtasks. All the dependency constraints of Θ can be preserved if each vertex u_i executes in its *lifetime window* $[\eta_i^{st}, \eta_i^{sp}]$. See [17] for more details regarding the laxity distribution rules.

Example 3. After laxity distribution, the OpenMP-DAG in Fig. 3 is transformed into a timing diagram with the workload assigned as shown in Fig. 5. The *lifetime window* of u_{11} becomes [0,8] and the *lifetime window* of u_{21} becomes [8,16].

5.2.3 Vertex Reassembling

A vertex may be split into several parts during the segmentation step such that each vertex part may be assigned to a different segment. In this step, we reassemble the different parts of a vertex, adjust the time constraints obtained from the laxity distribution step accordingly, and then use them to obtain the vertex's (the subtask's) release time and deadline. Note that in this step, we simply reassemble all the parts belonging to the same vertex; we do not change the *lifetime window* obtained from the previous step.

The vertex reassembling forces the sequential subtasks that belong to the same vertex to be assigned to one processor by the P-EDF-omp algorithm (which will be discussed later). The reassembling operation will not affect the off-line partitioning. Hence, we choose to implement this step during decomposition to simplify the Subtask Assignment Procedure, which will be shown next.

5.2.4 Resulting Sequential Sporadic Subtask Set

After decomposition, we obtain a resulting sequential sporadic subtask set $\Theta_{decomp} = \{\tau_1, \tau_2, \ldots, \tau_N\}$, where each sequential subtask τ_i corresponds to a vertex in the original OpenMP-DAG (after vertex reassembling). We use N to denote the total number of sequential subtasks as well as the number of nodes in OpenMP-DAG. Each subtask τ_i can be represented as a triple $\langle e_i, \eta_i^{st}, \eta_i^{sp} \rangle$, where e_i represents the execution requirement of τ_i (which equals the WCET of corresponding vertex), and η_i^{st} and η_i^{sp} represent the starting and ending times of Δ_i (the *lifetime window* of τ_i), respectively. More specifically, η_i^{st} and η_i^{sp} are the artificial release time and deadline (relative to the release time of the OpenMP-DAG) of each subtask after vertex reassembling.

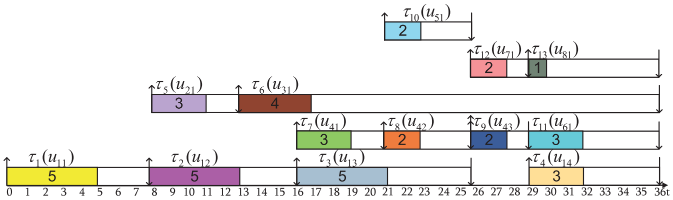


Fig. 6. Decomposition result of example in Fig. 3 (after reassembling).

Notice 2. *In this paper, we consider η_1^{st} (the starting time of the first released vertex's* lifetime window*) to be time 0, i.e., $\eta_1^{st} = 0$. Therefore, the time points considered in this paper, such as η_i^{st} and η_i^{sp}, are all absolute times.*

This resulting sequential sporadic subtask set is the object partitioned by the SAP in P-EDF-omp at design time.

Example 4. After decomposition, the OpenMP-DAG in Fig. 3 is transformed into a resulting sequential subtask set, as shown in Fig. 6. The lifetime window of u_{11} is [0,8] and the lifetime window of u_{21} is [8,16], both of which are the same as the results after laxity distribution.

5.3 The P-EDF-omp algorithm

In this section, we present P-EDF-omp, a partitioning-based multiprocessor scheduling algorithm for OpenMP-DAGs.

The most important part of P-EDF-omp is the Subtask Assignment Procedure (SAP), which is used to partition the subtasks to processors satisfying the *Partitioning Conditions* (which will be introduced later) at design time.

To better understand how P-EDF-omp works, we first introduce some auxiliary concepts. Recall that we assume $\eta_1^{st} = 0$, and the starting and ending times of the *lifetime window* of subtask τ_i are both absolute times.

We use the concept of Interval Load (denoted by K) to denote the processor demand of each subtask τ_i during the considered time interval.

Definition 5 (The Interval Load). *For subtask τ_i, the Interval Load $K(\tau_i, iv_x)$ in time interval $iv_x \rightarrow [iv_x^{st}, iv_x^{sp}]$ is defined as follows:*

$$K(\tau_i, iv_x) = \begin{cases} e_i, & \eta_i^{st} \geq iv_x^{st} \ \&\& \ \eta_i^{sp} \leq iv_x^{sp} \\ 0, & \text{otherwise} \end{cases}. \tag{1}$$

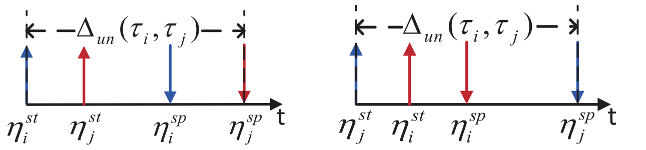
Definition 6 (Overlapping Subtasks). *If the* lifetime windows *of $\tau_i(\Delta_i \rightarrow [\eta_i^{st}, \eta_i^{sp}])$ and $\tau_j(\Delta_j \rightarrow [\eta_j^{st}, \eta_j^{sp}])$ satisfy the following:*

1) $\eta_i^{st} \leq \eta_j^{st} < \eta_i^{sp} \ \&\& \ \eta_j^{sp} > \eta_i^{sp}$ *or*
2) $\eta_j^{st} \leq \eta_i^{st} < \eta_j^{sp}$

then we say that τ_j is an Overlapping Subtask of τ_i, which means that the lifetime windows *of these two subtasks either partially or entirely overlap.*

Definition 7 (Union Time Interval). *For two Overlapping Subtasks τ_i and τ_j, we use $\Delta_{un}(\tau_i, \tau_j)$ to denote the* `Union Time Interval`*, which is the union of* lifetime windows *of τ_i and τ_j, and use $d_{un}(\tau_i, \tau_j)$ to denote the length of the* `Union Time Interval` *$\Delta_{un}(\tau_i, \tau_j)$.*

Example 5 illustrates the `Union Time Interval` $\Delta_{un}(\tau_i, \tau_j)$ for the Overlapping Subtasks τ_i and τ_j.



(a) Partially overlap. (b) Entirely overlap.

Fig. 7. Example of Union Time Interval $\Delta_{un}(\tau_i, \tau_j)$.

Example 5. As shown in Fig. 7a, suppose there are two Overlapping Subtasks τ_i and τ_j; in this case, the `Union Time Interval` of τ_i and τ_j is $\Delta_{un}(\tau_i, \tau_j) \rightarrow [\eta_i^{st}, \eta_j^{sp}]$ (i.e., $\Delta_{un}^{st}(\tau_i, \tau_j) = \eta_i^{st}, \Delta_{un}^{sp}(\tau_i, \tau_j) = \eta_j^{sp}$) and the length of $\Delta_{un}(\tau_i, \tau_j)$ is $d_{un}(\tau_i, \tau_j) = \eta_j^{sp} - \eta_i^{st}$. For cases in which the *lifetime windows* of two subtasks overlap entirely, as shown in Fig. 7b, the `Union Time Interval` of these two *lifetime windows* is simply the larger *lifetime window*, i.e., in this case, $\Delta_{un}(\tau_i, \tau_j) \rightarrow [\eta_j^{st}, \eta_j^{sp}]$, where $\Delta_{un}^{st}(\tau_i, \tau_j) = \eta_j^{st}$, $\Delta_{un}^{sp}(\tau_i, \tau_j) = \eta_j^{sp}$ and $d_{un}(\tau_i, \tau_j) = \eta_j^{sp} - \eta_j^{st}$.

Similar to the *lifetime window* of subtask τ_i, we define the *lifetime window* of an OpenMP task T_h as follows: (Suppose T_h consists of the subtask set $\{\tau_{h1}, \tau_{h2}, \ldots, \tau_{hn_h}\}$)

Definition 8 (Lifetime window of OpenMP Task). *The* lifetime window *of T_h (denoted by Λ_h) is the time interval from the starting time of the* lifetime window *of the entry vertex in T_h to the ending time of the exit vertex's* lifetime window*, i.e., $\Lambda_h \rightarrow [\eta_{h1}^{st}, \eta_{hn_h}^{sp}]$. Specifically, we use ρ_h^{st} and ρ_h^{sp} to denote the starting and ending times of Λ_h, i.e., $\rho_h^{st} = \eta_{h1}^{st}$, $\rho_h^{sp} = \eta_{hn_h}^{sp}$.*

In the following, we introduce the *Partitioning Conditions* used in our work and how we employ them to determine whether a processor is "available" for a subtask.

Partitioning Conditions. The *Partitioning Conditions* cover three possible conditions (the subtask to be assigned is $\tau_i(\langle e_i \eta_i^{st}, \eta_i^{sp} \rangle), \tau_i \in T_h$).

Condition (1): (Basic Condition.)

In the *lifetime window* of τ_i ($\Delta_i \rightarrow [\eta_i^{st}, \eta_i^{sp}]$)

$$d_i - \sum_{\tau_j \in \Pi_k, j \neq i} K(\tau_j, \Delta_i) \geq e_i, \tag{2}$$

where d_i is the length of the *lifetime window* of τ_i , i.e., $d_i = \eta_i^{sp} - \eta_i^{st}$. Π_k denotes the set of subtasks already assigned to processor s_k, and $\Pi_k = \emptyset$ at time 0.

Condition (2): (Additional Condition.)

If an Overlapping Subtask τ_j of τ_i exists,[7] then in their Union Time Interval $\Delta_{un}(\tau_i, \tau_j)$

$$d_{un}(\tau_i, \tau_j) - \sum_{\tau_h \in \Pi_k, h \neq i} K(\tau_h, \Delta_{un}(\tau_i, \tau_j)) \geq e_i. \tag{3}$$

Condition (3): (TSC Condition.)

$$\Gamma_k^o(\eta_i^{st}) = \emptyset \quad || \quad \forall T_q \in \Gamma_k^o(\eta_i^{st}),\ T_h \in \Psi_{des}(T_q), \tag{4}$$

where $\Gamma_k^o(t)$ denotes the set of `tied` tasks that have been assigned to processor s_k, whose *lifetime window* starts before

7. We illustrate how this condition works in the situation where more than one Overlapping Subtask of τ_i in Notice 3 exists, and the details are provided in Algorithm 1.

the current time and ends after the current time (i.e., for $\forall T_q \in \Gamma_k^o(t),\ \rho_q^{st} \le t < \rho_q^{sp}$). In particular, $\Gamma_k^o(0) = \emptyset$.

The *Partitioning Conditions* consist of three parts. We use the following example to explain the main idea behind these parts. Suppose we want to partition the subtask τ_i to processor s_k.

(1) *Condition (1)*, the basic condition, simply ensures that s_k can satisfy the processor demand of τ_i (which is its WCET) in its *lifetime window*.

(2) *Condition (2)*, the additional condition, addresses the following situation. If the subtask τ_j, which corresponds to u_{h1} in the `tied` task T_h, has been assigned to s_k, all the subtasks corresponding to $\forall u \in \omega_{T_h}$ should all be partitioned to s_k. Therefore, although their *lifetime windows* may not have started yet, we can already determine which processor they should be assigned to, and the processor resource should be seen as "pre-reserved" for these vertices. Hence, by the time we partition τ_i, some subtasks whose *lifetime windows* start after τ_i may already exist on the processor. Therefore, if a subtask τ_j exists such that τ_j is an Overlapping Subtask of τ_i, we need to ensure that s_k can accommodate both subtasks during their Union Time Interval. Thus, we design *Condition (2)*.[8]

(3) *Condition (3)* ensures that the TSC in the OpenMP specification can be satisfied.

Evidently, we do not need to check all three *Partitioning Conditions* while partitioning every subtask in Θ_{decomp}. For example, if u_{h1} in `tied` task T_h has been assigned to s_k, there is no need to check whether *Condition (3)* can be satisfied for $\forall u \in \omega_{T_h}$. Next, we will clarify how to use the *Partitioning Conditions* to determine whether a processor is "available" for different types of subtasks.

Availability. Suppose the subtask to be assigned is τ_i ($\tau_i \in T_h$). The process of using the *Partitioning Conditions* to determine whether a processor s_k is *"available"* for τ_i can be divided into the following cases for different types of τ_i.

- Case 1: T_h is an `untied` task,
  s_k is *"available"* for τ_i if and only if *Condition (1)* and *Condition (2)* in the *Partitioning Conditions* are satisfied.
- Case 2: T_h is `tied` and τ_i corresponds to $u_{h1} \in T_h$,
  s_k is *"available"* for τ_i if and only if all three conditions in *Partitioning Conditions* are satisfied for τ_i itself, and *Condition (1)* and *Condition (2)* can be satisfied for every subtask corresponding to $u \in \omega_{T_h}$ at the same time.
- Case 3: T_h is `tied` and τ_i corresponds to $u \in \omega_{T_h}$,
  the *"available"* processor for this type of subtask τ_i will always be the one to which u_{h1} has been assigned.

The subtasks are divided into three types.

(1) If τ_i corresponds to a vertex in an `untied` task, we only need to consider whether a processor can satisfy the processor demand (its WCET) in its *lifetime window* using *Partitioning Conditions (1) and (2)*. This guarantees that if we can find an "available" processor for τ_i, τ_i can meet its deadline when scheduled by EDF_{np} at runtime (which will be proved later in Section 6).

(2) If τ_i corresponds to u_{h1} in `tied` task T_h, we first need to determine whether the TSC in OpenMP can be satisfied using *Partitioning Condition (3)*. Moreover, since every subtask corresponding to $u \in T_h$ has to be assigned to this processor, we need to use *Partitioning Conditions (1) and (2)* to check whether the processor can satisfy their processor demand in their *lifetime windows* for all these subtasks (as we do for the vertex in an `untied` task) rather than checking only whether the processor can satisfy the processor demand of τ_i itself. Otherwise, the processor resources may be insufficient for the complete `tied` task, and some successor subtasks of τ_i may miss their deadlines. We can only partition τ_i to s_k if we can ensure that sufficient processor resources exist for all vertices in the same task.

Algorithm 1. Function is_available(s_k, τ_i) $(\tau_i \in T_h$, $\Delta_i \rightarrow [\eta_i^{st}, \eta_i^{sp}])$

f_available(s_k, τ_i)=0;
if $d_i - \sum_{\tau_j \in \Pi_k, j \neq i} K(\tau_j, \Delta_i) \ge e_i$ **then**
  f_available(s_k, τ_i)=1;
  for tp=1:1:$|Q_k|$ **do**
    if $((\eta_{Q_k(tp)}^{st} < \eta_i^{st}\ \&\&\ \eta_{Q_k(tp)}^{sp} > \eta_i^{st})\ \ ||$ $(\eta_i^{st} \le \eta_{Q_k(tp)}^{st} < \eta_i^{sp}\ \&\&\ \eta_{Q_k(tp)}^{sp} > \eta_i^{sp}))$ **then**
      f_available(s_k, τ_i)=-1;
      $\Delta_{un}^{st} = \min\{\eta_{Q_k(tp)}^{st}, \eta_i^{st}\}$;
      $\Delta_{un}^{sp} = \max\{\eta_{Q_k(tp)}^{sp}, \eta_i^{sp}\}$;
      if $(\Delta_{un}^{sp} - \Delta_{un}^{st} - \sum_{\tau_j \in \Pi_k, j \neq i} K(\tau_j, \Delta_{un}) \ge e_i$ && check_union$(\tau_i, s_k, \Delta_{un}^{st}, \Delta_{un}^{sp})$ == 1) **then**
        f_available(s_k, τ_i)=1;
      else
        f_available(s_k, τ_i)=0;
        break;
      end if
    end if
  end for
end if
returnf_available(s_k, τ_i);

(3) If τ_i corresponds to $u \in \omega_{T_h}$ in `tied` task T_h, τ_i can be directly assigned to the processor to which u_{h1} is assigned. This processor has already been labeled as "available" for τ_i while assigning u_{h1}; thus, it does not need to be checked again.

Accordingly, the Subtask Assignment Procedure (SAP) will try to assign all subtasks in Θ_{decomp} to the "available" processors, using a "first-fit" heuristic. In other words, for $\forall \tau_i \in \Theta_{decomp}$, the SAP will scan the processors in a canonical order (e.g., from processor 1 to m) and assign τ_i to the first "available" processor. These subtasks are partitioned in non-decreasing order based on the starting times of their *lifetime windows*; i.e., if there are two subtasks τ_i and τ_j, and $\eta_i^{st} < \eta_j^{st}$, the SAP will try to assign τ_i before τ_j. If the SAP fails to find a processor s_k for τ_i then this OpenMP-DAG is not schedulable by P-EDF-omp.

After τ_i has been assigned to s_k, s_k subsequently uses EDF_{np} to schedule its local subtasks at runtime.

The pseudo-code of our "is_available[9]" algorithm, which will be called by the SAP, is shown in Algorithm 1. It is used

8. The details of how Overlapping Subtasks of τ_i are detected will be presented later in Algorithm 1.

9. Every processor s_k maintains a data structure Q_k, where each element in Q_k is a triple $\langle e_i, \eta_i^{st}, \eta_i^{sp} \rangle$ that stores the execution requirements, and the starting and ending times of the *lifetime window* of τ_i, which has been assigned to s_k.

to check whether the processor demand of τ_i can be satisfied by s_k, using *Partitioning Conditions (1) and (2)*.

Algorithm 2. Function check_union($\tau_i, s_k, iv^{st}, iv^{sp}$)

f_union=1;
for tp=1:1:$|Q_k|$ **do**
  if ($\eta^{st}_{Q_k(tp)} < iv^{st}$ && $\eta^{sp}_{Q_k(tp)} > iv^{st}$) || ($iv^{st} \leq \eta^{st}_{Q_k(tp)} < iv^{sp}$ && $\eta^{sp}_{Q_k(tp)} > iv^{sp}$) **then**
    f_union=-1;
    $iv^{st} = \min\{\eta^{st}_{Q_k(tp)}, iv^{st}\}$;
    $iv^{sp} = \max\{\eta^{sp}_{Q_k(tp)}, iv^{sp}\}$;
    if ($iv^{sp} - iv^{st} - \sum_{\tau_j \in \Pi_k, j \neq i} K(\tau_j, \Delta'_{un}) \geq e_i$ && check_union($\tau_i, s_k, iv^{st}, iv^{sp}$) == 1) **then**
      f_union=1;
    else
      f_union=0;
      break;
    end if
  end if
end for
returnf_union;

Notice 3. *As stated in Algorithm 1, if there exists more than one subtask whose* lifetime window *partially overlaps with the* lifetime window *of τ_i (we use $\mathcal{S}$ to denote the set of these subtasks), not only will we check whether Condition (2) in* Partitioning Conditions *can be fulfilled for $\forall \tau_j \in \mathcal{S}$ with τ_i but also whether Condition (2) in* Partitioning Conditions *can be fulfilled while considering all these subtasks jointly. For example, if both* lifetime windows *of τ_j and τ_h partially overlap with τ_i's lifetime window, when considering the partitioning for τ_i, we will check whether*

1) $d_{un}(\tau_i, \tau_j) - \sum_{\tau_p \in \Pi_k, p \neq i} K(\tau_p, \Delta_{un}(\tau_i, \tau_j)) \geq e_i$
2) $d_{un}(\tau_i, \tau_h) - \sum_{\tau_p \in \Pi_k, p \neq i} K(\tau_p, \Delta_{un}(\tau_i, \tau_h)) \geq e_i$
3) $d_{un}(\tau_i, \tau_j, \tau_h) - \sum_{\tau_p \in \Pi_k, p \neq i} K(\tau_p, \Delta_{un}(\tau_i, \tau_j, \tau_h)) \geq e_i$,

where, respectively

$$\Delta_{un}(\tau_i, \tau_j) \rightarrow [min(\eta^{st}_j, \eta^{st}_i), max(\eta^{sp}_j, \eta^{sp}_i)],$$

$$\Delta_{un}(\tau_i, \tau_h) \rightarrow [min(\eta^{st}_h, \eta^{st}), max(\eta^{sp}_h, \eta^{sp}_i)],$$

and

$$\Delta_{un}(\tau_i, \tau_j, \tau_h) \rightarrow [min(\eta^{st}_j, \eta^{st}_i, \eta^{st}_h), max(\eta^{sp}_j, \eta^{sp}_i, \eta^{sp}_h)].$$

Next, we present the pseudo-code of our “check_union” function, which is a recursive function called by Algorithm 1 and ensures that *Partitioning Condition (2)* can be fulfilled during partitioning. We use it to ensure that the processor demand in the `Union Time Interval` we considered will not exceed the length of the time interval in these two cases:

- Case 1: when more than one overlapping subtask of τ_i exists, as described in Notice 3
- Case 2: when there exists a subtask τ_h whose *lifetime window* does not overlap with the *lifetime window* of τ_i, but overlaps with the `Union Time Interval` of τ_j and τ_i. For example, as shown in Fig. 8, we not only need to check whether *Partitioning Condition (2)* can be fulfilled for $\Delta_{un}(\tau_i, \tau_j)$ but also whether *Partitioning Condition (2)* can be fulfilled for $\Delta_{un}(\tau_i, \tau_j, \tau_h)$, even though τ_h is not an overlapping subtask of τ_i.

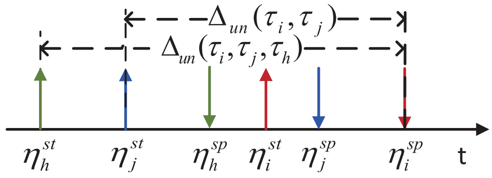


Fig. 8. Example of the usage of “check_union” function in Case 2.

Given Algorithms 1 and 2, we now show how the Subtask Assignment Procedure works in P-EDF-omp. The SAP assigns the subtasks to the processors based on the starting times of their *lifetime windows*. The pseudo-code of the SAP is shown in Algorithm 3. As stated earlier, it addresses the vertices for three different cases. Suppose the subtask that needs to be assigned is τ_i and $\tau_i \in T_h$.

Algorithm 3. Subtask Assignment Procedure $\tau_i\langle e_i, \eta^{st}_i, \eta^{sp}_i\rangle$ is to be Assigned and $\tau_i \in T_h$

if T_h is an `untied` task **then**
  for s_k=1:1:m **do**
    if is_available(s_k, τ_i)==1 **then**
      Assign τ_i to s_k and update s_k's data structure;
      F_available=1;
      break;
    else
      F_available=0;
    end if
  end for
else
  if τ_i is not the entry vertex in T_h **then**
    Assign τ_i to the processor τ_{h1} has been assigned to.
  else
    for k=1:1:m **do**
      if ((is_available(s_k, τ_i)) && ($\Gamma^o_k(\eta^{st}_i) = \emptyset$ || $\forall T_q \in \Gamma^o_k(\eta^{st}_i)$, $T_h \in \Psi_{des}(T_q)$)) **then**
        Get ω_{T_h};
        for tp=1:1:$|\omega_{T_h}|$ **do**
          if (is_available($s_k, \omega_{T_h}(tp)$)==0) **then**
            F_available=0;
            break;
          else
            F_available=1;
          end if
        end for
        if (F_available==1) **then**
          Assign τ_i to s_k, update $s_k{}'s$ data structure;
          break;
        end if
      end if
    end for
  end if
end if

Example 6. We illustrate how SAP partitions the different types of vertices using the subtask set in Example 4.

Suppose we want to schedule this subtask set on a platform containing four identical processors. We use the partition of τ_1, τ_7 and τ_8 (corresponding to u_{11}, u_{41}, and u_{42}, respectively) as representatives.

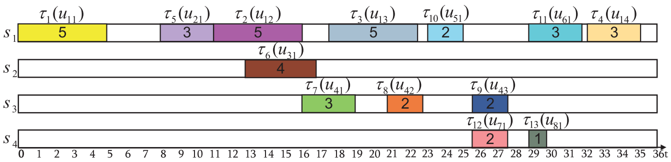


Fig. 9. Partition result of subtask set in Example 4.

- τ_1: *$\tau_1 \in T_1$ and T_1 is* `untied`*; therefore, we only need to check whether the processor can provide enough resources for τ_1 using* Partitioning Conditions (1) and (2). *In this case,* Partitioning Condition (1) *is satisfied and no Overlapping Subtasks exist for τ_1; consequently, s_1 is available for τ_1. As a result, τ_1 is assigned to s_1.*
- τ_7: *T_4 is* `tied` *and τ_7 corresponds to u_{41}; therefore, we first need to check whether Partitioning Conditions (1), (2) and (3) can all be satisfied. In this case, s_1 cannot fulfill* Condition (1) *for τ_7, so we continue to check s_2—*Condition (3) *cannot be fulfilled because T_3 has been assigned to this processor and $T_4 \notin \Psi_{des}(T_3)$. Hence, we move on to check s_3 and find that all three conditions can be satisfied. We also need to check whether* Partitioning Conditions (1) and (2) *can be fulfilled for u_{42} and u_{43}. In other words, s_3 can satisfy the processor demand for all subtasks in T_4 and the TSCs are satisfied. Consequently, τ_7 is assigned to s_3.*
- τ_8: *T_4 is* `tied` *and τ_8 corresponds to u_{42}; therefore, we can assign it directly to the processor to which τ_7 has been assigned (i.e., s_3).*

The partitioning process for other vertices functions the same way as described above. When the partitioning procedure is complete, the partitioned result for all the subtasks is illustrated in Fig. 9.

Discussion About OpenMP-Compliant Scheduling. Most existing OpenMP implementations support only two scheduling algorithms: Work First Scheduling (WFS) [31] and Breadth First Scheduling (BFS) [32]. WFS prefers to execute newly created tasks, while BFS tends to execute tasks that have been executed on the threads. The common feature of WFS and BFS is that they are both work-conserving with `untied` OpenMP task systems, where tasks can migrate among threads. For OpenMP task systems with `tied` tasks, BFS and WFS not only lose their work-conserving property but may also lead to extremely bad timing behaviors (in the worst-case, all parallel workloads will be executed on the same thread) [5].

In our paper, P-EDF-omp is not OpenMP-compliant from a scheduling view. Instead, P-EDF-omp is based on the decomposition of the OpenMP-DAG and the release times of the subtasks has been changed, which affects their runtime behavior. However, our approach is compliant with the special scheduling constraints in the OpenMP specification such as `tied` tasks, TSPs and TSC.

Therefore, this work has value, and we hope that it can provide some insights into what features could be included in the future OpenMP specification, because the OpenMP specification is a standard that keeps actively evolving rather than a static one. More specifically, P-EDF-omp is not supported by the current OpenMP specification for the following reasons. P-EDF-omp is a partitioning-based scheduling algorithm which needs to control the release time of each TSP and it also needs the deadline of each TSP for the *Partitioning Conditions*. Hence, to implement P-EDF-omp with OpenMP, we need to annotate each TSP and pass the timing parameters (release time and deadline) of each TSP to the OpenMP so that at runtime the each TSP can be scheduled by P-EDF-omp accordingly. However, the current OpenMP has neither the explicit annotations of TSPs nor the notion of recurrence [10]. Therefore, to be able to support partitioning-based scheduling algorithms like P-EDF-omp, the future OpenMP specification should be extended with *subtask construct* to explicit annotate each subtask corresponding to the TSPs, and corresponding clauses to receive the timing parameters of each subtask (as later introduced in Section 8.3).

On the other hand, although the proposed algorithm may reduce processor utilization, its advantage is that it can provide hard real-time guarantees for OpenMP applications at design time. Hence, although P-EDF-omp is not currently an OpenMP-compliant scheduling algorithm, we believe this work could help in applying OpenMP to embedded and real-time domains.

Discussion About Our Approach. In fact, the dynamic execution model is one of OpenMP's main strengths, and the actual workload structure can only be determined during runtime. Specifically, if the OpenMP program has conditional branches, we cannot know which branch will be taken at runtime, which will affect the OpenMP-DAG structures. This problem can be divided into two cases.

Case 1: the branches are contained in a single vertex. In this case, no matter which branch is taken in runtime, it can be bounded by the worst-case execution time such that our approach can address it.

Case 2: the branches cover multiple vertices. In this case, the number of spawned tasks may vary when different branches are taken.

However, our approach is based on decomposition; thus, we need to obtain the complete OpenMP-DAG beforehand. Consequently, the decomposition cannot be performed on-line or incrementally during execution. Therefore, our approach can neither address the dynamic execution model (e.g., Case 2) nor those OpenMP applications that are input-dependent whose task-graph and task-granularities vary. This is a serious limitation of our approach and occurs because up to now, we have focused solely on how to schedule `tied` tasks. The runtime schedulers that determine what occurs on-line, such as BFS, are suitable for the dynamic task model but cannot provide hard real-time guarantees for OpenMP applications as does our approach. Our approach would become much more valuable if it were able to address the dynamic execution model of OpenMP programs with `tied` tasks and test their schedulability—we hope to address this in future work.

6 SCHEDULABILITY

In this section, we prove that P-EDF-omp can work as an off-line schedulability-test in Theorem 2, using the SAP in Algorithm 3. Compared with the previous work [1], this schedulability-test can be used to determine whether an

OpenMP-DAG with `tied` tasks is schedulable during offline phase, without modifying the original TSCs posed in the OpenMP specification.

First, we prove that when using P-EDF-omp to schedule an OpenMP-DAG with `tied` tasks, the OpenMP scheduling constraints can be satisfied.

Lemma 1. *When scheduled by P-EDF-omp, the scheduling constraints imposed by* TSPs, Tied Tasks *and* TSC *in the OpenMP framework, as stated in Section 4.2, can all be satisfied.*

Proof. We prove the three constraints individually.

- As shown in Section 4.2, the existence of `TSPs` introduces constraints on the execution of the OpenMP task graph. In P-EDF-omp, we use non-preemptive EDF on every individual processor. Because the decomposition converts each node of a DAG into a traditional multiprocessor subtask, "non-preemptive" here means node-level non-preemption which corresponds to the definition of `TSP`. Therefore, this constraint can be satisfied.
- `Tied Task` is another important constraint brought by OpenMP. In P-EDF-omp, after we find the "available" processor for the entry vertex v_{h1} of a `tied` task T_h, the following vertices will be directly assigned to the same processor, which ensures that all the vertices of a task annotated as `tied` will be executed by the same processor. Hence, the scheduling constraint introduced by *Tied Tasks* is proved to be guaranteed.
- The constraint brought by `TSC` can clearly be guaranteed based on *Condition (3)* in the *Partitioning Conditions*. □

Next, we prove that P-EDF-omp can guarantee that the deadline of every subtask will be met if all subtasks can be successfully assigned to the "available" processors by SAP.

A sufficient schedulability-test is given in [33]. Based on Theorem 1 in [33] combined with our model, we obtain Corollary 1, which provides sufficient conditions for a subtask set to be scheduled successfully by EDF_{np} on a single processor.

Theorem 1. *[33] Let Θ_{decomp} be a set of sporadic tasks $\Theta_{decomp} = \{\tau_1, \tau_2, \ldots, \tau_{|\Theta_{decomp}|}\}$ where $\tau_i = (e_i, p_i)$ (e_i and p_i denote τ_i's WCET and period, respectively), sorted in non-decreasing order by their period p_i. If Θ_{decomp} satisfies the following conditions (1) and (2), EDF_{np} will schedule any concrete set of sporadic tasks generated from Θ_{decomp}*

$$(1) \quad \sum_{i=1}^{n} \frac{e_i}{p_i} \leq 1$$

$$(2) \quad \forall i, 1 < i \leq n; \forall d_{iv_x}, p_1 < d_{iv_x} < p_i,$$
$$d_{iv_x} \geq e_i + \sum_{j=1}^{i-1} \left\lfloor \frac{d_{iv_x} - 1}{p_j} \right\rfloor e_j$$

Corollary 1. *Let $\Pi_k = \{\tau_1, \tau_2, \ldots, \tau_{n_{s_k}}\}$ be the resulting concrete sporadic subtask set decomposed for OpenMP-DAG which are assigned to the same processor s_k, sorted in non-decreasing order by their deadlines, then if Π_k satisfies conditions (1) and (2), then the EDF_{np} scheduling algorithm will schedule Π_k successfully.*

$$(1) \quad \text{for } iv_x \to [0, \eta^{sp}_{n_{s_k}}] \text{ (whose length is } \eta^{sp}_{n_{s_k}}),$$
$$\frac{\sum_{i=1}^{n_{s_k}} K(\tau_i, iv_x)}{\eta^{sp}_{n_{s_k}}} \leq 1$$

$$(2) \quad \text{for } \forall i, \forall \text{ time interval } iv_x \text{ (whose length is } d_{iv_x}),$$
$$d_{iv_x} \geq e_i + \sum_{\tau_h \in \Pi_k, h \neq i} K(\tau_h, iv_x),$$
$$\text{where } iv_x^{st} \leq \eta_i^{st}, iv_x^{sp} \geq \eta_i^{sp}$$

Proof. We prove the contrapositive of the Corollary 1, i.e., Π_k satisfies the conditions (1) and (2) and yet there exists a subtask $\tau_h \in \Pi_k$ missing its deadline at some point in time when Π_k is scheduled by EDF_{np}.

In our model, for $\forall \tau_i \in \Theta_{decomp}$, the period of τ_i is the same as the period of the OpenMP-DAG. Therefore, in every OpenMP-DAG life cycle, one and only one subtask needs to be considered.[10] Hence, Condition (2) in our Corollary 1 is the same as Condition (2) in Theorem 1 in [33]; therefore, in the following, we focus only on whether Condition (1) in Theorem 1 in [33] can be replaced by Condition (1) in our Corollary 1.

Let t_d be the earliest point in time at which a deadline is missed. Π_k can be partitioned into three disjoint subsets.

S1 = the set of subtasks that have an invocation with a deadline at time t_d,

S2 = the set of subtasks that have an invocation occurring prior to time t_d and a deadline after t_d,

S3 = the set of subtasks not in S1 or S2.

Tasks in S3 either have a release time greater than t_d, or they have not been invoked immediately prior to t_d. As will shortly become apparent, to bound the processor demand prior to t_d, it is sufficient to concentrate on the tasks in S2. Let $b_1, b_2, \ldots, b_k$ be the invocation times immediately prior to t_d of the tasks in S2. There are two cases to consider.

Case 1: None of the invocations of tasks in S2 occurring at times $b_1, b_2, \ldots, b_k$ are scheduled prior to t_d.

Let t_0 be the end of the last period prior to t_d in which the processor was idle. If the processor has never been idle, let $t_0 = 0$. In the interval $iv_x \to [t_0, t_d]$, the processor demand is the total processing requirement of the tasks invoked at or after time t_0, with deadlines at or before time t_d (Dm_{t_0,t_d} is the processor demand in the interval $iv_x \to [t_0, t_d]$). Because no idle period exists in the interval $iv_x \to [t_0, t_d]$ and because a task misses a deadline at t_d, it follows that $Dm_{t_0,t_d} > t_d - t_0$. Therefore

$$t_d - t_0 < Dm_{t_0,t_d} = \sum_{i=1}^{n_{s_k}} K(\tau_i, iv_x).$$

10. EDF_{np} scheduling happens at runtime and it schedules the jobs generated by each subtask τ_i. However, because only one job exists for each subtask to be considered, we can directly use the subtask for simplicity.

Hence

$$1 < \frac{\sum_{i=1}^{n_{s_k}} K(\tau_i, iv_x)}{\eta_{n_{s_k}}^{sp}}.$$

This contradicts Condition (1) in Corollary 1. Therefore, Condition (1) has been proved.

Case 2: Some of the invocations of tasks in S2 occurring at times $b_1, b_2, \ldots, b_k$ are scheduled prior to t_d.

Case 2 establishes Condition (2) in Corollary 1. Because the proof of this case is almost the same as that in [33], we omit the proof of Case 2 here (see [33] for more details about this case). The main idea still involves proving by contradiction. □

Lemma 2. *After a subtask τ_i has successfully been assigned to a processor s_k by the Subtask Assignment Procedure in P-EDF-omp, it can be guaranteed to meet its deadline.*

Proof. Based on Corollary 1, if our *Partitioning Conditions* can ensure that Θ_{decomp} can satisfy the conditions in Corollary 1, after τ_i is successfully assigned to s_k, EDF_{np} can successfully schedule it; thus, Lemma 2 can be proved.

In the following, we prove that our *Partitioning Conditions* can guarantee that the conditions in Corollary 1 are fulfilled.

According to *Partitioning Conditions (1) and (2)*, for $\forall \tau_i \in \Pi_k$ (suppose τ_i is the subtask assigned to s_k with the latest η_i^{sp}). Then, we can conclude that the processor demand in the time interval $iv_x \rightarrow [0, \eta_i^{sp}]$ will never exceed the length of this time interval, which is η_i^{sp}, i.e.,

$$\sum_{\tau_j \in \Pi_k, j \neq i} K(\tau_j, iv_x) + e_i \leq \eta_i^{sp}.$$

Hence

$$\sum_{j=1}^{n_{s_k}} \frac{K(\tau_j, iv_x)}{\eta_i^{sp}} \leq 1 \Rightarrow \frac{\sum_{j=1}^{n_{s_k}} K(\tau_j, iv_x)}{\eta_i^{sp}} \leq 1.$$

Thus, Condition (1) in Corollary 1 can be fulfilled.

For Condition (2), the main idea is to guarantee that in any time interval, the processor demand will not exceed the length of the interval. We will prove this in three cases. The subtasks in Π_k, which are assigned to the same processor s_k, can be partitioned into three disjoint subsets (suppose τ_i is the current subtask to be scheduled):

S1=the set of subtasks whose *lifetime window* does not overlap with $\tau_i's$ lifetime window

S2=the set of subtasks whose *lifetime window* partially overlaps with $\tau_i's$ lifetime window

S3=the set of subtasks whose *lifetime window* entirely overlaps with $\tau_i's$ lifetime window

Case 1: subtasks in S1.

For $\tau_j \in S1$ (suppose the *lifetime window* of τ_j is the one closest to $\tau_i's$ lifetime window), in the interval $\Delta_j \rightarrow [\eta_j^{st}, \eta_j^{sp}]$, we have

$$d_j - \sum_{\tau_h \in \Pi_k, h \neq j} K(\tau_h, \Delta_j) \geq e_j.$$

For τ_i in the time interval $\Delta_i \rightarrow [\eta_i^{st}, \eta_i^{sp}]$, we have

$$d_i - \sum_{\tau_p \in \Pi_k, p \neq i} K(\tau_p, \Delta_i) \geq e_i.$$

Then, if we use $\epsilon \geq 0$ to denote the time distance between the *lifetime windows* of τ_i and τ_j, we obtain

$$\sum_{\tau_h \in \Pi_k} K(\tau_h, \Delta_j) + \sum_{\tau_p \in \Pi_k} K(\tau_p, \Delta_i) \leq d_j + d_i + \epsilon,$$

which reflects that the processor demand in the time interval $[\eta_j^{st}, \eta_i^{sp}]$ (supposing that the *lifetime window* of τ_j starts prior to $\tau_i's$ lifetime window) is no greater than the length of the time interval, i.e.,

$$d_{iv_x} = d_j + d_i + \epsilon \geq e_i + \sum_{\tau_h \in \Pi_k, h \neq i} K(\tau_h, \Delta_{un}(\tau_i, \tau_j)).$$

Therefore, in Case 1, Condition (2) in Corollary 1 can be fulfilled.

Case 2: subtasks in S2.

We can directly use our *Partitioning Condition (2)* to prove that Condition (2) in Corollary 1 can be fulfilled.

For $\forall \tau_j \in S2$, we have

$$d_{un}(\tau_i, \tau_j) - \sum_{\tau_h \in \Pi_k, h \neq i} K(\tau_h, \Delta_{un}(\tau_i, \tau_j)) \geq e_i,$$

where $d_{un}(\tau_i, \tau_j)$ is the length of the time interval $[min(\eta_j^{st}, \eta_i^{st}), max(\eta_j^{sp}, \eta_i^{sp})]$. In other words, our *Partitioning Condition (2)* ensures that in this case, the processor demand of the subtasks in this time interval will never be greater than the length of the interval, i.e.,

$$d_{iv_x} = d_{un}(\tau_i, \tau_j) \geq e_i + \sum_{\tau_h \in \Pi_k, h \neq i} K(\tau_h, \Delta_{un}(\tau_i, \tau_j)).$$

Therefore, in this case, Condition (2) in Corollary 1 is also guaranteed to be fulfilled.

Case 3: subtasks in S3.

The subtasks in S3 can be divided into two situations:

1) the *lifetime window* of τ_i contains the window of τ_j,
In this situation, we actually already consider the processor demand of τ_j in *Partitioning Condition (1)*—it is already included in $\sum_{\tau_j \in \Pi_k, j \neq i} K(\tau_j, \Delta_i)$. Consequently, Condition (2) in Corollary 1 will be fulfilled, i.e., $d_{iv_x} = d_i \geq e_i + \sum_{\tau_h \in \Pi_k, h \neq i} K(\tau_h, \Delta_i)$.[11]
2) the *lifetime window* of τ_j contains the window of τ_i,
This situation is addressed in *Partitioning Condition (2)*:

$$d_{un}(\tau_i, \tau_j) - \sum_{\tau_h \in \Pi_k, j \neq i} K(\tau_h, \Delta_{un}(\tau_i, \tau_j)) \geq e_i,$$

in which $d_{un}(\tau_i, \tau_j)$ denotes the length of the `Union Time Interval` $[\eta_j^{st}, \eta_j^{sp}]$, i.e.,

11. This situation appears mainly for the "preassigned" subtasks due to the existence of `tied` tasks—τ_j is not actually assigned to s_k yet, but the `tied` task to which it belongs has already been assigned to s_k; therefore, the processor resources should be "pre-reserved" for τ_j. Thus, despite the fact that the starting time of the *lifetime window* of τ_i occurs prior to $\tau_j's$, we still have to consider $\tau_j's$ processor demand.

$$d_{iv_x} = d_j \geq e_i + \sum_{\tau_h \in \Pi_k, h \neq i} K(\tau_h, \Delta_j).$$

Therefore, Condition (2) in Corollary 1 is proved to be fulfilled in this situation.

In conclusion, Lemma 2 has been proved. □

At this point, we have proven that we can use the SAP in Algorithm 3 to test the schedulability of OpenMP-DAG at design time.

Theorem 2. *An OpenMP-DAG is schedulable with the OpenMP scheduling constraints satisfied, if the SAP in Algorithm 3 can find "available" processors for* $\forall \tau_i \in \Theta_{decomp}$.

Proof. With Lemmas 1 and 2, Theorem 2 is proved. □

Similar to Theorem 1 in [33], our schedulability-test is a sufficient schedulability-test. P-EDF-omp is a partitioning-based scheduling algorithm, which is a transformation from the bin-packing problem. For classic bin packing problems, algorithms for finding an optimal solution to partitioning require exponential time [34]. For the partitioning of OpenMP-DAGs, we need to consider the OpenMP scheduling constraints, which raise more challenges to find optimal solutions. Hence in this paper, instead of trying to find optimal solutions, we study the sufficient conditions to schedule the OpenMP-DAGs with `tied` tasks in polynomial time.

7 Evaluation

In this section, we evaluate the performance of P-EDF-omp under both synthetic workloads and established OpenMP benchmarks. Specifically, we use the schedulability-test in Theorem 2 to test the schedulability of the OpenMP-DAG rather than executing the OpenMP-DAG on real hardware.

Since `tied` tasks raise significant challenges, the schedulability-tests of existing scheduling algorithms for the DAG model are not directly applicable to OpenMP-DAGs with `tied` tasks. Although several other papers exist that address the scheduling and analysis of OpenMP task systems, these papers either did not consider `tied` tasks [4], [7], [8], [9] or concluded that `tied` tasks would lead to unacceptably pessimistic response-time bounds due to their inherent complexity [5]. Hence, we compare the performance of our approach with the analysis methods in [1]: the bound in Equation (12) (denoted by BFS^*-1) and the bound in Equation (25) (denoted by BFS^*-2).

Moreover, we include a hypothesis schedulability test for OpenMP-DAGs without `tied` tasks in [5]: the response time bound under BFS in Equation (1) (denoted by untied). We use it to show the influence of `tied` tasks on the schedulability of OpenMP-DAGs. For these methods, the OpenMP-DAG is deemed to be schedulable if $R \leq D$. In addition, to evaluate the schedulers' impacts on the performance of the OpenMP application itself for synthetic workloads, we compare the average simulation execution times of OpenMP-DAGs under these three schedulers (BFS, BFS^* and P-EDF-omp). We normalized the three simulation execution times with respect to the response time bound for `untied` tasks under BFS in [5], which is $R_{untied} \leq L + (V - L)/m$.

The acceptance ratio is the ratio between the number of OpenMP-DAGs deemed to be schedulable and the total number of OpenMP-DAGs participating in the experiment (with a specific parameter configuration), without considering the overheads from context switching and migration.

7.1 Synthetic Workload

The task parameters are generated as follows:

Task Graph. The OpenMP-DAG $G = < V, E >$ is generated with $n = 50$ OpenMP tasks. The number of vertices contained in the OpenMP-DAG is randomly chosen in [200,400], and the worst-case execution time of each vertex is randomly chosen in $[300, 1500]$. Each task is set to be a `tied` task with a probability of p_{tied}. For every task T_i, we generate the synchronization edges as follows:

- If T_i has a child task, the last vertex v_{ix} of T_i is set to be the `taskwait` vertex with a probability of p_{tw} when at least one predecessor of v_{ix} (which belongs to T_i) has an outgoing task creation edge.
- The last vertex of T_i points to one of its siblings created after T_i by `depend` edge with a probability of p_{dep}.

Deadlines and Periods. The period is set as $P = D$.

- In Fig. 10d, the deadline D of Θ is set as $D = L/\Upsilon$.
- In other subgraphs in Fig. 10, the deadline D of Θ is generated in a similar way with that in [35]: after L is fixed, D is generated based on a ratio between L and D, which is randomly chosen in [0.2,0.5].

Number of Processors.

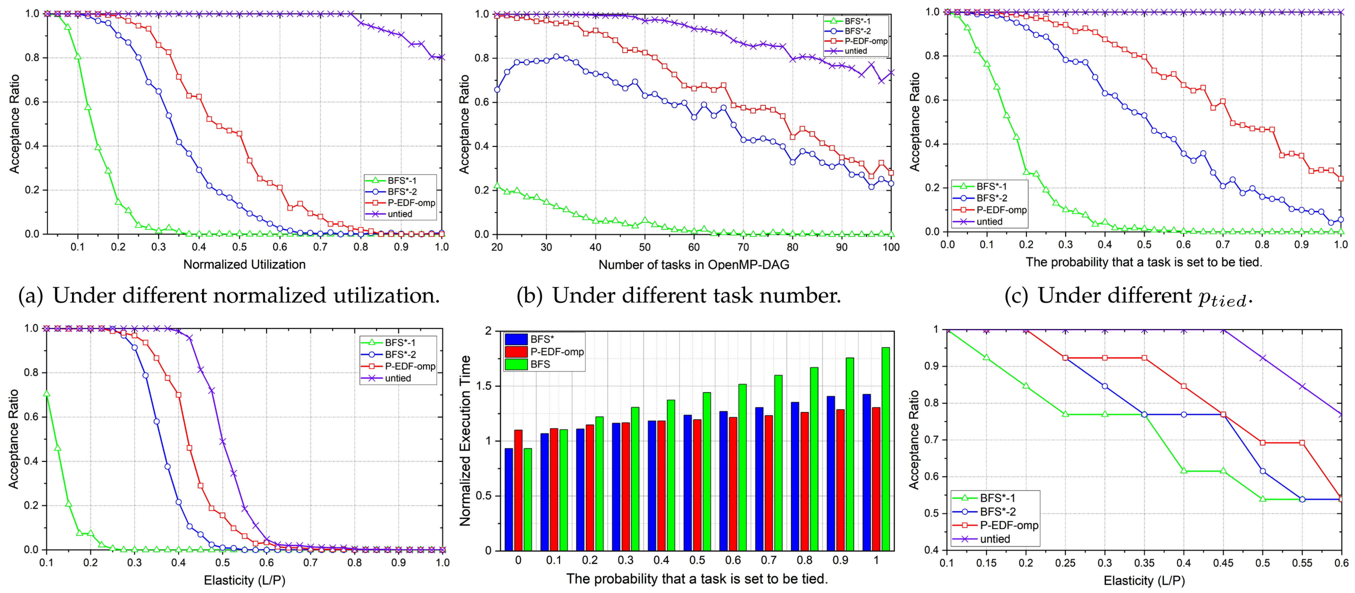
- In Fig. 10a, the normalized utilization U_{norm} ($U_{norm} = U/m$) of Θ is predefined. After generating the OpenMP-DAG, we can compute the utilization U of Θ; then, we set the number of processors according to the formula $m = \lceil \frac{U}{U_{norm}} \rceil$.
- For Figs. 10b and 10c, we set $m = 4$. For Figs. 10d and 10e, we set $m = 8$.

For each configuration (corresponding to one point on the X-axis), we generate 500 OpenMP-DAGs.

As shown in Figs. 10a, 10b, 10c, 10d, and 10e, we first set a basic configuration; then, in each group of experiments, we vary one parameter while keeping others unchanged. The basic configurations are: $n = 50$, $m = 4$, $p_{tied} = 0.5$, $p_{dep} = 0.5$ and $p_{tw} = 0.8$.

In Fig. 10a, we evaluate the acceptance ratios of all tests under different normalized utilization. In Fig. 10b, the OpenMP-DAG is generated with different numbers of OpenMP tasks. Fig. 10c evaluates the acceptance ratios under different p_{tied} values. Fig. 10d shows the acceptance ratios of all tests under different elasticities Υ. Finally, in Fig. 10e, we test the average normalized simulation execution times of the three schedulers under different p_{tied} values.

As the experiment results in Figs. 10a, 10b, 10c, and 10d show, in general, the schedulability of all tests decreases as the processor contentions among tasks become more significant. We can observe that the performance comparison among all three approaches under different parameter settings is relatively consistent: P-EDF-omp $>$ BFS^*-2 $>$ BFS^*-



(a) Under different normalized utilization. (b) Under different task number. (c) Under different p_{tied}.

(d) Under different elasticity Υ. (e) Simulations under different p_{tied}. (f) Under different Υ (benchmarks).

Fig. 10. Comparisons under different dimensions.

1. Among these schedulability-tests, BFS*-1 performs the worst because it simply counts the number of `tied` tasks that may be suspended at idle threads/processors and ignores the fact that only the executions of a subset of vertices in these tasks may influence the schedulability of the OpenMP-DAG. P-EDF-omp performs better than both BFS*-1 and BFS*-2 in most cases, because both BFS*-1 and BFS*-2 work only under modified TSC (which is called E-TSC in [1]) instead of the original TSC in OpenMP specification. Compared with the original TSC, E-TSC restricts not only the execution of vertices in `tied` tasks but also the execution of vertices in `untied` tasks (it does not allow `untied` vertices to start execution on an arbitrary processor) to make it possible to derive the response bounds for OpenMP-DAGs with `tied` tasks. However, it is evident that restricting the execution of vertices in `untied` tasks can negatively affect the schedulability of the OpenMP-DAG, while our schedulability-test works under the original TSC and does not suffer from this problem. Hence in most cases, P-EDF-omp performs better.

However, in Fig. 10d, the acceptance ratio of BFS*-2 is slightly higher than that of P-EDF-omp with $\Upsilon = 0.25$. This result occurs because in a configuration where the elasticity is small, the period and the deadline of the OpenMP-DAG will be quite large ($D = P$), making it more likely to satisfy $R \leq D$. In our approach, a lower Υ means a larger laxity, and the utilization will be relatively lower. Therefore, it is possible that there exist several OpenMP-DAGs that are deemed to be schedulable by BFS*-2 but unschedulable by our schedulability-test in Section 6. However, this kind of cases rarely occurs, and as shown in Fig. 10d, P-EDF-omp performs better than BFS*-2 in almost 99.5 percent cases.

From Fig. 10e, we can see that the average simulation execution times of the OpenMP applications increase with more `tied` tasks in the OpenMP-DAGs under all three schedulers. The increasing speed of the execution time is the fastest under BFS. This result occurs because when `tied` tasks exist in the OpenMP-DAG, BFS is no longer work-conserving, and in the extreme case, BFS may perform as poorly as WFS (i.e., BFS may also execute the parallel workloads sequentially, thus leading to poor timing behavior). This condition is the reason why we were motivated to design a scheduling algorithm for OpenMP-DAGs with `tied` tasks. When only `untied` tasks exist in the OpenMP-DAGs, the simulation execution time under P-EDF-omp is larger than those under the other two scheduling algorithms because during decomposition, we modify the release time and deadline of each vertex, which reduces processor utilization to some extent. In other words, if the SAP successfully assigns all the subtasks to processors, the execution time of the OpenMP-DAG ought to be relatively close to the period P. However, the increasing speed of the execution time under P-EDF-omp is considerably slower than those under the other two scheduling algorithms, and the normalized execution time under P-EDF-omp is smaller than those under the other two scheduling algorithms when more `tied` tasks exist in the OpenMP-DAGs.

7.2 Established Benchmarks

Here, we evaluate the three approaches with workloads generated according to established OpenMP benchmarks. We collected 11 OpenMP programs written in the C language from several benchmarks. Table 2 shows detailed information regarding the OpenMP-DAGs corresponding to these benchmarks. Columns 4–6 show the DAG feature of each application.[12]

For these established OpenMP benchmarks, the task parameters are collected and generated as follows:

Task Graph. For the OpenMP-DAG topologies of every OpenMP program, we insert codes into these programs to generate the vertices and edges corresponding to the TSPs. The OpenMP-DAG topologies of each OpenMP program are generated dynamically by running these OpenMP

12. There are 11 applications while we test two different input_sizes for *nqueens* and *sort* in the evaluation.

TABLE 2
Summary of the Established OpenMP Benchmarks

Applications	Source	adjusted inputs	Tasks	Vertices	Edges
botsspar	spec [36]	size 10×10	136	290	424
fft	bots [37]	size 150	81	227	304
fib		size 10	177	353	528
nqueens		size 5	221	485	704
		size 14 cut_off 8	(15,984,851)	(35,323,344)	(52,278,549)
sort		size 20000	66	130	193
		size 33,554,432	383,078	766,154	1,156,893
sparselu.for		size 10×10	137	301	435
nbody	dash [38]	bodies 1000 steps 5	151	320	469
pingpong	ompmpi [39]	data_size [1,2] target_time 100	204	408	604
overlap		data_size [1,2] target_time 100	204	408	604
taskbench	ompb [40]	outer-repetition 200 test-time 10 delay-time 10	109	216	311
strassen	kastors [41]	size 256	153	321	472

applications on real single-core hardware (Intel i7-4770 CPU running at 3.5 GHz with a 8,192 KB cache, and 4 GB RAM). Then, we measured the WCETs for the vertices by executing the programs on the same real hardware above. More specifically, we instrumented OpenMP programs with instructions to read the timer at the beginning and the end of each vertex. The execution time of the vertex is the difference between the start and end time stamps. We executed each program on a single core 100 times and recorded the maximum execution time for each vertex as its WCET. The measured WCET gives a rough idea of the workload of each vertex; however, it does not guarantee a safe WCET bound. In the OpenMP-DAG generation, for most of the benchmarks, we use the default input data that are included in the source code of each benchmark. Table 2 lists the input data that we adjusted to control the size of the OpenMP-DAG, where the size column denotes the input_size. Other parameters not presented are default values that we did not set manually when running the applications.

Deadlines and Periods. Deadline D of each OpenMP application is set as $D = L/\Upsilon$. The period is set as $P = D$.

Number of Processors. For each OpenMP application, we set the number of processors as $m = 20$.

For each configuration (corresponding to one point on the X-axis), we tested the schedulability of every OpenMP application and computed the acceptance ratio as

$$\left\{\frac{\text{the number of OpenMP applications that are schedulable}}{\text{the total number of OpenMP applications}}\right\},$$

where the total number of OpenMP applications is 13.

As shown in Fig. 10f, although our approach still performs better than BFS^*-1 and BFS^*-2, the improvement in acceptance ratio compared to BFS^*-2 is slightly reduced compared to the synthetic ones. The first reason is that the number of OpenMP-DAGs (13) we used for each configuration is considerably smaller than the number in synthetic ones, and not all of the OpenMP applications we collected contain `tied` tasks (such as *botsspar* in spec [36]). Our approach is superior mainly when scheduling OpenMP-DAGs with `tied` tasks. The second reason is that the response time bound in BFS^*-2 is highly dependent on the max number of `tied` tasks in the OpenMP task chain where the tasks are connected by taskwait edges. For these OpenMP applications, this number ranges from 0 to 9, which improves the acceptance ratio of BFS^*-2. Thus, when the elasticity is less than 0.25 (which means the period of each OpenMP-DAG is relatively large), BFS^*-2 performs as better as does our schedulability-test. However, as the elasticity increases, the performance of BFS^*-2 drops more quickly than that of our schedulability-test. The general trend shown in Fig. 10f is similar to the results of synthetic workload experiment: as the elasticity Υ increases, the acceptance ratio decreases. Among these 13 OpenMP applications, the lower the ratio between the number of `tied` tasks and the total number of tasks, the more easily the OpenMP-DAG can be scheduled successfully. For example, there are no `tied` tasks in *botsspar* in spec [36]; consequently, this application can always be successfully scheduled with all three algorithms on the considered platform, even when $\Upsilon = 0.6$. Some applications in bots [37] (from *fft* to *sort*) contain recursive functions and have a relatively large number of `taskwait` vertices, and these applications appear to be more difficult to schedule. We also test the schedulability of *nqueens* and *sort* with larger input sizes to test the performance of P-EDF-omp at large cases. We found that the WCET features and the structure features of the corresponding OpenMP-DAGs with larger input sizes are quite similar to the ones with small input sizes, and the schedulability of the OpenMP-DAGs appear not to be influenced much by the size of the inputs.

8 Discussion About the Implementation

P-EDF-omp is a partitioning-based algorithm that does not modify the TSCs in OpenMP specification. Ruffaldi *et al.* presented an OpenMP toolchain for multicore partitioning in [42] called "SOMA", which includes a runtime support that makes partitioning-based scheduling practicable for OpenMP programs. SOMA allows the developers to add specific information to each task, such as its deadline, activation time and period. Due to its similarity, P-EDF-omp can be implemented based on SOMA in [42][13] and the transformation tool in [43].

In this section, we present a possible solution about how to implement P-EDF-omp based on the current OpenMP specification. The basic implementation procedure includes: (1) OpenMP-DAG generation, (2) Schedule generation, (3) Code profiling, and (4) Runtime support.

8.1 OpenMP-DAG Generation

The first phase of the implementation is source code analysis, where the goal is to generate the OpenMP-DAG of the program and the annotated OpenMP code. Fig. 11a shows the architecture of the transformation tool.[14]

The parser outputs abstract syntax trees (AST) that store all the relevant information to create the corresponding graph structure. The AST is used (1) by the Drawer to construct the task graph of individual functions and (2) by the Call Analyzer

13. One important difference is that SOMA was designed for partitioning OpenMP tasks while we need to partition the subtasks.

14. The light blue boxes are the existing tools we utilized, while the dark blue boxes indicate functionalities we developed.

to generate call graphs. The integrator combines both of them to generate the task graph of the entire application.

Meanwhile, we introduce a new directive *#pragma omp subtask* to annotate every TSP in the source code. The syntax of the subtask construct is as follows:

#pragma omp subtask [clause ...] new-line
structured-block.

We annotate the source code with this directive; every subtask construct includes all the code segments corresponding to a vertex in the OpenMP-DAG, and we give each subtask construct a unique identifier for further analysis. Thus, in addition to the OpenMP-DAG, we obtain the annotated OpenMP code.

In addition to the task graph topology, we provide two types of reference weight values:

- *Static Analysis.* This type of reference value is obtained by using the static WCET analysis tool Chronos [44] to compute a safe WCET bound for the codes associated with each individual vertex.
- *Measurement.*We measure the execution time for the vertices by executing the programs on real hardware (as described in Section 7.2).

8.2 Schedule Generation

In this phase, we use our proposed algorithm to compute the schedule of the OpenMP-DAG and record relevant information, including the starting and ending times of each vertex's *lifetime window*, and the mapping of each vertex to the processors (vertices are identified by the identifiers obtained in the preceding phase).

8.3 Code Profiling

OpenMP lacks an important feature: the notion of recurrence [10]. To include the real-time features, we incorporate two clauses associated with the subtask construct: the *event* clause and the *deadline* clause. These two clauses were first proposed in [10]. The syntaxes of these two clauses are as follows: (These two clauses are compatible.)

$$\text{\#pragma omp subtask event } (event - expression)$$

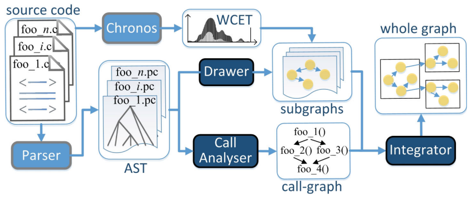
$$\text{\#pragma omp subtask deadline } (deadline - expression),$$

where the *event-expression* represents the exact moment in time at which the subtask release occurs and the *deadline-expression* is the expression that determines the time instant at which the subtask must finish. Only if *event-expression* evaluates to true is the associated subtask released. The expression shall evaluate to false after the subtask release.

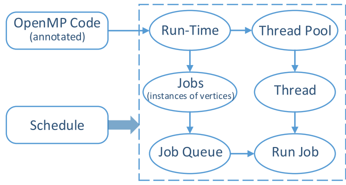
We set the *event-expression* and *deadline-expression* of every subtask construct based on the information obtained in the preceding step. Thus, we obtain the final annotated OpenMP code, which includes the real-time features we need.

8.4 Runtime Support

After obtaining the schedule and the final annotated OpenMP code, we can use them as the input to the runtime, as SOMA did in [42]. The aim of the runtime is to instantiate and manage the threads, and to control the execution of the vertices. In particular it must allocate each vertex on the correct thread



(a) Structure of the transformation tool.



(b) Structure of the Runtime support

Fig. 11. Stages of implementation.

and must guarantee the artificial release time of each vertex. The runtime does not require time-consuming computations; all its allocation decisions are made based on information written in the schedule. The runtime support spawns the threads and allocates jobs to them according to the schedule.

Specifically, when jobs arrive, the runtime uses the identifier to identify the corresponding subtask and allocate the job to the thread based on the mapping written in the schedule. These jobs will be stored in each thread's local pool. The thread uses EDF_{np} to schedule the jobs in its local pool. The *deadline-expression* allows the thread to identify jobs corresponding to the subtasks with the closest deadlines. Fig. 11b shows the structure of the runtime support.

9 Conclusion

OpenMP is a promising parallel programming framework in general-purpose and high-performance computing, and has garnered increasing attention in the embedded and real-time domains [1], [3], [4], [5], [6], [7], [8], [9], [10]. Previous work in [1] studied the timing analysis of OpenMP task systems with regard to response time bounds by modifying the original TSCs in the OpenMP specification. In this paper, we propose a new algorithm, P-EDF-omp, that can automatically guarantee satisfying the tied constraints as long as an OpenMP task system can be successfully partitioned to the multiprocessor platform. Experiments with both randomly generated task sets and established OpenMP benchmarks show that our approach consistently outperforms the work in [1]—even without modifying TSCs.

Acknowledgments

This work was sponsored in part by the NSFC Project under Grant 61772123. Xu Jiang is the co-first author.

References

[1] J. Sun, N. Guan, Y. Wang, Q. He, and W. Yi, "Real-time scheduling and analysis of OpenMP task systems with tied tasks," in *Proc. IEEE Real-Time Syst. Symp.*, 2017, pp. 92–103.
[2] OpenMP Architecture Review Board (ARB), "OpenMP application program interface V4.0," *OpenMP Forum*, Tech. Rep., 2013.
[3] R. E. Vargas, S. Royuela, M. A. Serrano, X. Martorell, and E. Quinones, "A lightweight OpenMP4 run-time for embedded systems," in *Proc. 21st Asia South Pacific Des. Autom. Conf.*, 2016, pp. 43–49.

[4] R. Vargas, E. Quinones, and A. Marongiu, “OpenMP and timing predictability: A possible union?,” in *Proc. Des. Autom. Test Eur. Conf. Exhib.*, 2015, pp. 617–620.

[5] M. A. Serrano, A. Melani, R. Vargas, and A. Marongiu, “Timing characterization of OpenMP4 tasking model,” in *Proc. Int. Conf. Compilers Archit. Synthesis Embedded Syst.*, 2015, pp. 157–166.

[6] D. Ferry, J. Li, M. Mahadevan, K. Agrawal, C. Gill, and C. Lu, “A real-time scheduling service for parallel tasks,” in *Proc. IEEE 19th Real-Time Embedded Technol. Appl. Symp.*, 2013, pp. 261–272.

[7] M. A. Serrano and E. Quiñones, “Response-time analysis of DAG tasks supporting heterogeneous computing,” in *Proc. 55th ACM/ESDA/IEEE Des. Autom. Conf.*, 2018, pp. 1–6.

[8] G. Tagliavini, D. Cesarini, and A. Marongiu, “Unleashing fine-grained parallelism on embedded many-core accelerators with lightweight OpenMP tasking,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 9, pp. 2150–2163, Sep. 2018.

[9] A. Marongiu and L. Benini, “An OpenMP compiler for efficient use of distributed scratchpad memory in MPSoCs,” *IEEE Trans. Comput.*, vol. 61, no. 2, pp. 222–236, Feb. 2012.

[10] M. A. Serrano, S. Royuela, and E. Quiñones, “Towards an OpenMP specification for critical real-time systems,” in *Proc. Int. Workshop OpenMP*, 2018, pp. 143–159.

[11] A. Saifullah, J. Li, K. Agrawal, C. Lu, and C. Gill, “Multi-core real-time scheduling for generalized parallel task models,” *Real-Time Syst.*, vol. 49, no. 4, pp. 404–435, 2013.

[12] J. Li, Z. Luo, D. Ferry, K. Agrawal, C. Gill, and C. Lu, “Global EDF scheduling for parallel real-time tasks,” *Real-Time Syst.*, vol. 51, no. 4, pp. 395–439, 2015.

[13] K. Lakshmanan, S. Kato, and R. Rajkumar, “Scheduling parallel real-time tasks on multi-core processors,” in *Proc. IEEE 31st Real-Time Syst. Symp.*, 2010, pp. 259–268.

[14] A. Saifullah, D. Ferry, J. Li, K. Agrawal, C. Lu, and C. D. Gill, “Parallel real-time scheduling of DAGs,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 12, pp. 3242–3252, Dec. 2014.

[15] S. Baruah, “Improved multiprocessor global schedulability analysis of sporadic DAG task systems,” in *Proc. 26th Euromicro Conf. Real-Time Syst.*, 2014, pp. 97–105.

[16] E. Ayguadé *et al.*, “The design of OpenMP tasks,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 20, no. 3, pp. 404–418, Mar. 2009.

[17] X. Jiang, N. Guan, X. Long, and H. Wan, “Decomposition-based real-time scheduling of parallel tasks on multi-cores platforms,” *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 10, pp. 2319–2332, Oct. 2020.

[18] P. Burgio, G. Tagliavini, A. Marongiu, and L. Benini, “Enabling fine-grained OpenMP tasking on tightly-coupled shared memory clusters,” in *Proc. Des. Autom. Test Eur. Conf. Exhib.*, 2013, pp. 1504–1509.

[19] M. A. Serrano, A. Melani, M. Bertogna, and E. Quiñones, “Response-time analysis of DAG tasks under fixed priority scheduling with limited preemptions,” in *Proc. Des. Autom. Test Eur. Conf. Exhib.*, 2016, pp. 1066–1071.

[20] J. Sun, N. Guan, J. Sun, and Y. Chi, “Calculating response-time bounds for OpenMP task systems with conditional branches,” in *Proc. IEEE Real-Time Embedded Technol. Appl. Symp.*, 2019, pp. 169–181.

[21] A. Melani, M. A. Serrano, M. Bertogna, I. Cerutti, E. Quinones, and G. Buttazzo, “A static scheduling approach to enable safety-critical OpenMP applications,” in *Proc. 22nd Asia South Pacific Des. Autom. Conf.*, 2017, pp. 659–665.

[22] A. Duran, J. Corbalán, and E. Ayguadé, “Evaluation of OpenMP task scheduling strategies,” in *Proc. Int. Workshop OpenMP*, 2008, pp. 100–110.

[23] X. Teruel, X. Martorell, A. Duran, R. Ferrer, and E. Ayguadé, “Support for OpenMP tasks in Nanos v4,” in *Proc. Conf. Center Adv. Stud. Collaborative Res.*, 2007, pp. 256–259.

[24] X. Liu, J. Mellor-Crummey, and M. Fagan, “A new approach for performance analysis of OpenMP programs,” in *Proc. 27th Int. ACM Conf. Int. Conf. Supercomput.*, 2013, pp. 69–80.

[25] K. A. Huck, A. D. Malony, S. Shende, and D. W. Jacobsen, “Integrated measurement for cross-PlaNorm OpenMP performance analysis,” in *Proc. Int. Workshop OpenMP*, 2014, pp. 146–160.

[26] R. Dietrich, F. Schmitt, A. Grund, and D. Schmidl, “Performance measurement for the OpenMP 4.0 offloading model,” in *Proc. Eur. Conf. Parallel Process.*, 2014, pp. 291–301.

[27] OpenMP Architecture Review Board (ARB), “OpenMP application program interface V4.5,” *OpenMP Forum*, Tech. Rep., 2015.

[28] P. Voudouris, P. Stenström, and R. Pathan, “Timing-anomaly free dynamic scheduling of task-based parallel applications,” in *Proc. Real-Time Syst. Symp.*, 2017, pp. 365–376.

[29] M. Dertouzos, “Control robotics: The procedural control of physical processeds,” in *Proc. IFIP Congr.*, 1974, pp. 807–813.

[30] C. L. Liu and J. W. Layland, “Scheduling algorithms for multiprogramming in a hard-real-time environment,” *J. ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[31] M. Frigo, C. E. Leiserson, and K. H. Randall, “The implementation of the Cilk-5 multithreaded language,” in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 1998, pp. 212–223.

[32] G. J. Narlikar, “Scheduling threads for low space requirement and good locality,” *Theory Comput. Syst.*, vol. 35, no. 2, pp. 151–187, 2002.

[33] K. Jeffay, D. F. Stanat, and C. U. Martel, “On non-preemptive scheduling of periodic and sporadic tasks,” in *Proc. 12th Real-Time Syst. Symp.*, 1991, pp. 129–139.

[34] D. S. Johnson, “Near-optimal bin packing algorithms,” PhD dissertation, Dept. Math., Massachusetts Inst. Technol., Cambridge, MA, USA, 1973.

[35] S. Dinh, J. Li, K. Agrawal, C. Gill, and C. Lu, “Blocking analysis for spin locks in real-time parallel tasks,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 4, pp. 789–802, Apr. 2018.

[36] M. S. Müller *et al.*, “SPEC OMP2012 — An application benchmark suite for parallel systems using OpenMP,” in *Proc. Int. Workshop OpenMP*, 2012, pp. 223–236.

[37] A. D. González, X. Teruel, R. Ferrer, X. M. Bofill, and E. Ayguadé Parra, “Barcelona OpenMP tasks suite: A set of benchmarks targeting the exploitation of task parallelism in OpenMP,” in *Proc. 38th Int. Conf. Parallel Process.*, 2009, pp. 124–131.

[38] 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. Front.*, 2014, Art. no. 4.

[39] 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.

[40] J. M. Bull, F. Reid, and N. McDonnell, “A microbenchmark suite for OpenMP tasks,” in *Proc. Int. Workshop OpenMP*, 2012, pp. 271–274.

[41] P. Virouleau *et al.*, “Evaluation of OpenMP dependent tasks with the KASTORS benchmark suite,” in *Proc. Int. Workshop OpenMP*, 2014, pp. 16–29.

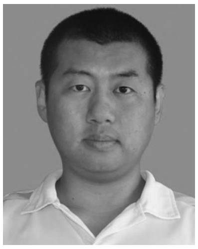
[42] E. Ruffaldi, F. Brizzi, G. Dabisias, and G. Buttazzo, “SOMA: An OpenMP toolchain for multicore partitioning,” in *Proc. 31st Annu. ACM Symp. Appl. Comput.*, 2016, pp. 1231–1237.

[43] Y. Wang *et al.*, “Benchmarking OpenMP programs for real-time scheduling,” in *Proc. IEEE Int. Conf. Embedded Real-Time Comput. Syst. Appl.*, 2017, pp. 1–10.

[44] X. Li, Y. Liang, T. Mitra, and A. Roychoudhury, “Chronos: A timing analyzer for embedded software,” *Sci. Comput. Program.*, vol. 69, no. 1, pp. 56–67, 2007.

Yang Wang received the BS degree in computer science and technology and the MS degrees in computer system architecture, both from Northeastern University, Shenyang, China, in 2013 and 2015, respectively. She is currently working toward the PhD degree with the School of Computer Science and Engineering, Northeastern University, Shenyang, China. Her research interests include real-time embedded systems, parallel tasks, and real-time scheduling.

Xu Jiang received the BS degree in computer science from Northwestern Polytechnical University, Xi’an, 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, Beijing, China, in 2012, and the PhD degree from Beihang University, Beijing, China, in 2018. He is currently working with the School of Computer Science and Engineering, Northeastern University. 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, Shenyang, China, in 2003 and 2006, respectively, and the PhD degree from Uppsala University, Uppsala, 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, Best Paper Award of IEEE Real-time Systems Symposium (RTSS), in 2009, Best Paper Award of Conference on Design Automation and Test in Europe (DATE), in 2013.

Zhishan Guo received the BE degree in computer science and technology from Tsinghua University, Beijing, China, in 2009, the MPhil degree in mechanical and automation engineering from the Chinese University of Hong Kong, Hong Kong, in 2011, and the PhD degree in computer science from the University of North Carolina at Chapel Hill, Chapel Hill, North Carolina, in 2016. He is an assistant professor with the Department of Electrical and Computer Engineering, University of Central Florida, Orlando, Florida, and an assistant professor with the Department of Computer Science, Missouri University of Science and Technology, Rolla, Missouri. His current research interests include real-time scheduling, cyber-physical systems, and neural networks and their applications.

Xue Liu received the PhD (Hons.) degree in computer science from the University of Illinois at Urbana-Champaign, Champaign, Illinois. He was the Samuel R. Thompson chaired associate professor with the University of Nebraska-Lincoln and a visiting faculty with HP Labs, Palo Alto, California. He is currently a William Dawson scholar and professor with the School of Computer Science, McGill University. He has published more than 200 research papers in major peer-reviewed international journals and conference proceedings in these areas and received several best paper awards. His research interests include cyber-physical systems, IoT, machine learning, big data and applications, green IT, sustainability, and smart energy systems, computer systems and networking.

Wang Yi (Fellow, IEEE) received the PhD degree in computer science from the Chalmers University of Technology, Gothenburg, Sweden, in 1991. He is a chair professor with Uppsala University. His interests include models, algorithms, and software tools for building and analyzing computer systems in a systematic manner to ensure predictable behaviors. He was awarded with the CAV 2013 Award for contributions to model checking of real-time systems, in particular the development of UPPAAL, the foremost tool suite for automated analysis and verification of real-time systems. For contributions to real-time systems, he received best paper awards of RTSS 2015, ECRTS 2015, DATE 2013, and RTSS 2009, Outstanding Paper Award of ECRTS 2012 and Best Tool Paper Award of ETAPS 2002. He is on the steering committee of ESWEEK, annual joint event for major conferences in embedded systems areas. He is also on the steering committees of ACM EMSOFT (co-chair), ACM LCTES, and FORMATS. He serves frequently on Technical Program Committees for a large number of conferences, and was the TPC chair of TACAS 2001, FORMATS 2005, EMSOFT 2006, HSCC 2011, LCTES 2012 and track/topic chair for RTSS 2008, and DATE 2012–2014. He is a member of Academy of Europe (Section of Informatics).