Global EDF Schedulability Analysis for Parallel Tasks on Multi-Core Platforms

Hoon Sung Chwa, *Member, IEEE*, Jinkyu Lee, *Member, IEEE*, Jiyeon Lee, *Student Member, IEEE*, Kiew-My Phan, *Student Member, IEEE*, Arvind Easwaran, and Insik Shin, *Member, IEEE*

Abstract—With the widespread adoption of multi-core architectures, it is becoming more important to develop software in ways that takes advantage of such parallel architectures. This particularly entails a shift in programming paradigms towards fine-grained, thread-parallel computing. Many parallel programming models have been introduced for targeting such intra-task thread-level parallelism. However, most successful results on traditional multi-core real-time scheduling are focused on sequential programming models. For example, thread-level parallelism is not properly captured into the concept of *interference*, which is key to many schedulability analysis techniques. Thereby, most interference-based analysis techniques are not directly applicable to parallel programming models. Motivated by this, we extend the notion of interference to capture thread-level parallelism more accurately. We then leverage the proposed notion of parallelism-aware interference to derive efficient EDF schedulability tests that are directly applicable to parallel task models, including DAG models, on multi-core platforms, without knowing an optimal schedule. Our evaluation results indicate that the proposed analysis significantly advances the state-of-the-art in global EDF schedulability analysis for parallel tasks. In particular, we identify that our proposed schedulability tests are adaptive to different degrees of thread-level parallelism and scalable to the number of processors, resulting in substantial improvement of schedulability for parallel tasks on multi-core platforms.

Index Terms—Real-time scheduling, parallel task, global EDF, interference

1 Introduction

With the advance of semiconductor technology, multi-/many-core architectures are widely used to better manage trade-offs between performance, power efficiency, and reliability in deep submicron technology. As the size of a CMOS transistor continuously shrinks, more cores are getting integrated on a single die. For example, Intel introduced the Intel Xeon Phi coprocessor with around 60 cores [1], and Cavium designed ARM-based processors that scale up to 48 cores [2]. Given the increasing emphasis on multi-/many-core chip design, software parallelism is likely to be one of the greatest constraints on computer performance. This inherently entails a shift in programming paradigms towards fine-grained thread-parallel computing, rather than relatively coarse-grained application-level parallelism.

A popular technique to achieve fine-grained, thread-level parallelism operates on the principle of *divide-and-conquer*. It breaks down a larger task into many smaller subtasks, runs those subtasks in parallel, and synchronizes them to merge the results once each subtask completes computation. Many parallel programming models have been proposed to support such a principle for parallel computation, including OpenMP [3], Cilk/Cilk++ [4], Intel Threading Building Blocks [5], Wool [6], and Chapel [7]. Those parallel programming models share a common scenario that some subtasks (threads) in a program (task) can run in parallel to produce partial results individually and certain threads should synchronize to integrate the partial results[1]. Many parallel task models have been considered to capture those two important aspects: *thread-level parallelism* and *synchronization*. One popular parallel task model is the synchronous parallel task model, where a task consists of a sequence of parallel regions, called *segments*, and each segment includes one or more threads. All the threads belonging to the same segment are released at the same time and at most m threads are able to run simultaneously, where m is the number of available cores. Two consecutive segments are subject to synchronization (precedence constraint); all the threads belonging to one segment must complete their own execution in order to move onward to the next segment. Recently, a growing research attention has been given to a more general parallel task model, the DAG (Directed Acyclic Graph) model, where a node represents a thread and an edge describes a precedence dependence between two threads. In the DAG model, the granularity of precedence constraint becomes smaller to the level of thread to offer a more

- H.S. Chwa, J. Lee, and I. Shin are with the School of Computing, KAIST, Daejeon 34141, South Korea. E-mail: {chwahs, jy.lee}@cps.kaist.ac.kr, insik.shin@cs.kaist.ac.kr.
- J. Lee is with the Department of Computer Science and Engineering, Sungkyunkwan University (SKKU), Seoul 110-745, South Korea. E-mail: jinkyu.lee@skku.edu.
- K. Phan is with PRECISE center, University of Pennsylvania, Philadelphia, PA 19104. E-mail: phankieumy@gmail.com.
- A. Easwaran is with the School of Computer Engineering, Nanyang Technological University, Singapore 639798. E-mail: arvinde@ntu.edu.sg.

Manuscript received 7 July 2015; revised 13 Sept. 2016; accepted 25 Sept. 2016. Date of publication 30 Sept. 2016; date of current version 12 Apr. 2017. Recommended for acceptance by R. Vuduc.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2016.2614669

1. In terminology of some programming models, such as OpenMP, Intel Threading Building Blocks, and Chapel, the term "task" is used to represent a unit of parallel execution referred to as a "thread" in this paper.

TABLE 1
Schedulability Analysis Methods for Parallel Tasks

	Comparative	Independent
Fork-join task model	[21]†	
Synchronous parallel task model	[22] [24] [26]	[23] [25] [27]†
DAG task model	[28] [30] [29]	[28] [29] [31] [32] **This paper**

†*Indicates a Case Where the Proposed Analysis Method is Applicable Only for Partitioned Scheduling*

flexible way of describing thread-level parallelism and synchronization than the synchronous parallel task model.

A shift from uni-core to multi-core processors allows *inter-task parallelism*, where several tasks can execute simultaneously on multi-core processors. In addition, a shift from single-thread to multi-thread tasks allows *intra-task parallelism*, where even a single task can have multiple threads running simultaneously to take full advantage of multi-core processing. Despite the growing importance of intra-task parallelism, most real-time multi-core scheduling studies are focused on inter-task parallelism of sequential tasks, and relatively much less attention has been paid to understanding intra-task parallelism towards the schedulability analysis of parallel tasks. For example, a large number of studies extensively investigated the schedulability analysis of sequential tasks under EDF (earliest-deadline-first) and fixed-priority global scheduling [8], producing many influential results. Such results include the concept of problem window and interference [9], [10], interference bounding techniques [11], [12], response time computation methods [10], [13], and optimal priority assignment [14]. Many other scheduling algorithms have been proposed for sequential models in order to take advantage of multi-core more effectively, including optimal algorithms such as pfair [15], DP-Fair [16], RUN [17], and U-EDF [18]. In addition, some approaches [19], [20] have been also proposed for scheduling tasks with pipeline precedence constraints in distributed real-time systems. However, the insights behind those successful results are not directly applicable to parallel tasks, due to the unique characteristics of thread-level parallelism.

For example, the notion of interference has been well defined in the sequential task case and serves as the basis for many schedulability analysis methods [9], [10], [11]. However, the current notion of interference does not capture thread-level parallelism because it assumes that each task has only a single thread to run at any time instant. Hence, those analysis methods are not directly applicable or easily extensible to parallel tasks.

Recently, a few schedulability analysis methods have been proposed for parallel task systems (see Table 1). Those methods can broadly fall into two types: *comparative* and *independent*. The analysis methods in the literature can be considered as *comparative*, if they derive resource augmentation bounds; the resource augmentation bound indicates how well a scheduler performs relatively to an optimal scheduler. Thus, such resource augmentation bounds serve as good measures to assess the performance of a scheduler and compare different schedulers. However, they can be hardly used to determine the schedulability of a set of given parallel tasks when no optimal schedule is known. In particular, [33] proved that it is impossible to find an optimal online multi-core scheduler for sporadic task systems. On the other hand, analysis methods can be said to be *independent*, if they can serve as schedulability tests directly even without having to find any optimal schedule. Motivated by this, the goal of this paper is to develop an efficient, independent schedulability test that can determine the schedulability of parallel tasks, including DAG tasks, directly in connection to no optimal schedule.

Contribution. The main contributions of this paper can be summarized as follows:

- We identify a chain of threads, called a *critical thread*, in a DAG task that makes the most significant impact to a deadline miss, if exists. We then derive interference-based schedulability analysis of global EDF scheduling for sporadic DAG task systems with novel notions of *critical interference* and *p-depth critical interference* in order to capture thread-level parallelism accurately (see Section 4).
- We develop a polynomial-time workload-based schedulability test. We identify the worst-case interference scenario for DAG tasks considering the structure of a DAG in detail and derive tight upper bounds on interference based on workload (see Section 5).
- We also develop a pseudo-polynomial-time slack-based iterative schedulability test to reduce pessimism effectively in bounding interference (see Section 6).
- We present simulation results, showing that the proposed schedulability analysis methods significantly outperforms the state-of-the-art methods available for DAG tasks even if no optimal schedule is known (see Section 7).

In our earlier work [23], we presented interference-based schedulability tests for a synchronous parallel task set under global EDF scheduling. In this paper, we extend this initial study towards a more expressive parallel task model, DAG task model, with further improvements. In the synchronous parallel task model, each task consists of a sequence of segments with synchronization points at the end of each segment. The DAG task model refines the granularity of synchronization from segment-level to thread-level, allowing to describe more flexible way of thread-level parallelism and synchronization. In order to generalize the schedulability analysis techniques in [23] for the DAG task model, this paper has the following new technical contributions in addition to re-organizing and re-writing the entire paper for better and/or concise presentation:

- The notions of critical thread, critical interference, and p-depth critical interference are extended accordingly with the DAG task model. Building upon those new notions, we extend the workload-based schedulability test in [23] towards the DAG task model, including a new way of calculating the worst-case workload for DAG tasks.

- We additionally propose an improved schedulability test by the use of slack values.
- We include more evaluation results, including a comparison with other related works for DAG tasks.

2 Related Work

Many scheduling approaches have been introduced for intra-task thread-level parallelism in the hard real-time context. The schedulability tests developed in the literature can be largely categorized into two types: (1) *comparative* ones that determine the schedulability of a scheduler relatively to an ideal scheduler, and (2) *independent* ones that determine the schedulability of a scheduler independently in connection to no other scheduler. For example, according to [28], resource and capacity augmentation bounds[2] serve as comparative and independent tests, respectively. Table 1 summarizes the scheduling approaches in the literature by schedulability test types and parallel task models.

Fork-Join Task Model. One of the widely used parallel task models is the *fork-join* model [21]. A fork-join task consists of an alternate sequence of sequential and parallel regions, called *segments*, and all the threads within each segment should synchronize in order to proceed to the next segment. Under the assumption that each parallel segment can have at most as many threads as the number of processors, [21] presented a resource augmentation bound of 3.42 for partitioned DM (deadline-monotonic) scheduling of periodic fork-join tasks with implicit deadlines.

Synchronous Parallel Task Model. Relaxing the restriction that sequential and parallel segments alternate, several studies have considered a more general synchronous parallel task model that allows each segment to have any arbitrary number of threads. [22] presented a resource augmentation bound of 4 for global EDF scheduling and 5 for partitioned DM scheduling of periodic tasks with implicit deadlines. Building upon this work, [34] presented a prototype scheduling service for their *RT-OpenMP* concurrency platform. [26] also showed a resource augmentation bound of 2 for a certain class of global scheduling algorithms, such as PD^2 [35], LLREF [36], DP-Wrap [16], or U-EDF [18] to schedule sporadic tasks with constrained deadlines.

Those studies [21], [22], [26] share a common principle of task decomposition for schedulability analysis. They decompose a single synchronous parallel task into multiple independent sequential sub-tasks through intermediate deadline assignment. This approach is safe—satisfying the intermediate deadlines of all sub-tasks leads to meeting the deadlines of their aggregate synchronous parallel tasks. They then employ existing schedulability analysis for those sequential sub-tasks. [22] decomposes a parallel task into a set of sequential sub-tasks such that the density of each segment is upper bounded by some value, and [26] decomposes a parallel task such that the maximum density among all segments in a parallel task is minimized. However, such an indirect analysis via task decomposition can be pessimistic, because task decomposition can incur non-trivial overheads. Furthermore, it requires modifications to existing operating systems to support task decomposition [34].

Recently, some studies [23], [24], [25], [27] developed direct schedulability analysis without task decomposition for synchronous parallel tasks. [24] showed a resource augmentation bound of $2 - 1/m$ for sporadic tasks with constrained deadlines under global EDF scheduling. [23] introduced an interference-based analysis for global EDF scheduling of sporadic tasks with constrained deadlines. [25] and [27] presented a response-time analysis (RTA) for sporadic tasks under global fixed-priority scheduling and partitioned fixed-priority scheduling, respectively.

DAG Task Model. Refining the granularity of synchronization from segment-level to thread-level, a Directed Acyclic Graph (DAG) task model is considered, where a node represents a thread and an edge specifies a precedence dependency between nodes. A thread can execute only after all of its predecessors have been executed. [30] showed a resource augmentation bound of 2 for a single DAG task with arbitrary deadlines under global EDF scheduling. For a set of DAG tasks, a resource augmentation bound of $2 - 1/m$ was presented for global EDF scheduling [28], [29]. [29] also derived a $3 - 1/m$ resource augmentation bound for global DM scheduling. In addition to those resource augmentation bounds, which serve as comparative schedulability tests, [28] introduced capacity augmentation bounds that can work as independent schedulability tests for sporadic DAG tasks with arbitrary deadlines under global EDF and rate-monotonic (RM) scheduling. [31] later improved the capacity augmentation bounds for global EDF and RM scheduling and proposed a new scheduling policy, called federated scheduling, for DAG tasks. [29] also presented a simple polynomial-time independent schedulability test for global EDF scheduling, and this work was later extended in [32] to yield an improved pseudo-polynomial time independent schedulability test. This paper proposes a new interference-based, independent schedulability test for a DAG task set under global EDF scheduling, significantly improving the schedulability compared to the existing techniques.

3 System Model

We consider a multi-core platform, where sporadic Directed Acyclic Graph tasks run over m identical processors under global EDF scheduling. A set of tasks is denoted by τ. In the sporadic DAG task model, a task $\tau_i \in \tau$ is specified by (G_i, D_i, T_i), where G_i is a directed acyclic graph as shown in Fig. 1, D_i is the relative deadline, and T_i is the minimum separation. The DAG G_i is specified as $G_i = (V_i, E_i)$, where V_i is a set of nodes and E_i is a set of directed edges between two nodes. Each node $\theta_{i,u} \in V_i$ represents a sequential operation (a "thread"), and is characterized by the worst-case execution time requirement (WCET) $C_{i,u}$. The number of threads in τ_i is denoted by N_i. A directed edge $(\theta_{i,u}, \theta_{i,v}) \in E_i$ represents the precedence dependency that $\theta_{i,v}$ cannot start execution unless $\theta_{i,u}$ has finished execution.

2. The resource augmentation bound r of a scheduler S has the property that if a task set is feasible on m unit-speed processors, then the task set is schedulable under S on m processors of speed r. For a scheduler S and its corresponding schedulability condition X, their capacity augmentation bound c has the property that if the given condition X is satisfied with a task set, the task set is schedulable by S on m processors of speed c. Since the resource augmentation bound is connected to an ideal optimal schedule, it is hard (if not impossible) to use it as a schedulability test due to the difficulty of finding an optimal schedule in many multi-core scheduling domains. On the other hand, the capacity augmentation bound has nothing to do with an optimal schedule, and this allows it to serve as an easy schedulability test (see [28] more details).

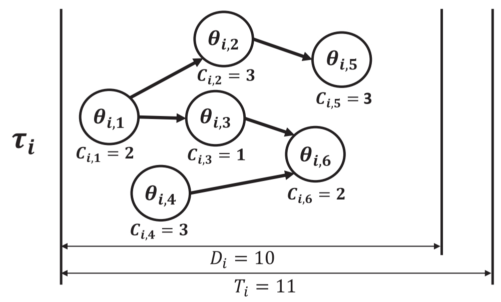

Fig. 1. A DAG task τ_i.

A thread $\theta_{i,u}$ becomes ready for execution as soon as all of its predecessors have completed their execution.

A *path* in the sporadic DAG task τ_i is a sequence of threads $\theta_{i,1}, \theta_{i,2}, \ldots, \theta_{i,f}$ such that $(\theta_{i,j}, \theta_{i,j+1})$ is an edge in G_i for $1 \leq j < f$. The length of this path is defined as the sum of the WCETs of all its threads: $\sum_{j=1}^{f} C_{i,j}$. The length of a longest path, denoted by LC_i can be computed in linear time in the number of nodes and the number of edges in G_i, when its nodes are sorted and processed in a topological order. In Fig. 1, the longest path of τ_i is the sequence of $\theta_{i,1}$, $\theta_{i,2}$ and $\theta_{i,5}$, with the longest path length LC_i equal to $2 + 3 + 3 = 8$. We also define C_i as the total WCET of τ_i, and it is presented as

$$C_i = \sum_{\theta_{i,j} \in V_i} C_{i,j}. \tag{1}$$

Note that LC_i is the minimum amount of time needed to execute all threads in τ_i assuming that it can use as many processors as possible for its execution, and C_i is the maximum amount of time to complete the execution of all the threads in τ_i on a single core.

A sporadic DAG task τ_i invokes a series of jobs, and successive jobs are released with a duration of at least T_i time units apart. If a job of τ_i is released at time instant t then all $|V_i|$ threads $\theta_{i,u} \in V_i$ are released at time instant t and must complete execution by the absolute deadline $t + D_i$. We consider a constrained deadline D_i such that $D_i \leq T_i$. It should be $LC_i \leq D_i$ but not necessarily $C_i \leq D_i$. Let U_i denote the utilization of τ_i and be defined as $U_i = C_i/T_i$. We denote the lth job of a task τ_i with J_i^l. We will omit the superscript in the notation for simplicity when no ambiguity arises. For a job J_i^l, r_i^l and d_i^l are its release time and deadline. The *execution window* of a job J_i^l is then defined as interval $[r_i^l, d_i^l)$.

In EDF scheduling, threads are assigned priorities according to their absolute deadline: the earlier the deadline of a thread, the higher its priority. Thereby, threads in different jobs may have different priorities, but all threads within a single job have the same priority since they share the same absolute deadline. With global EDF, each thread ready to execute is placed in a system-wide queue, ordered by non-decreasing absolute deadline, and the first m threads are extracted from the queue to execute on the available processors at every time instant. In this paper, we assume quantum-based time and without loss of generality, let one time unit denote the quantum length. All task parameters are assumed to be specified as multiples of this quantum length.

We summarize the notation used throughout the paper in the supplement, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2016.2614669.

4 Schedulability Analysis for DAG Tasks

In this section, we derive schedulability analysis of global EDF scheduling for sporadic DAG task systems with constrained deadlines. To this end, we extend the concept of interference towards the DAG task model and introduce a new concept called *critical interference*. With the notion of critical interference, we investigate what happens if there is a deadline miss and identify a necessary condition for a job to miss its deadline, which serves as a basis for schedulability analysis. In this section, a run-time schedule of a task set is assumed to be known. However, a worst-case situation where a task suffers the worst possible interference is generally unknown for global scheduling of sporadic task systems. This prevents calculation of the exact interference without knowledge of run-time schedule. Therefore, in later sections, we derive a safe upper bound on the interference and transform the necessary condition into an efficient schedulability test.

4.1 The Concept of Critical Interference

In the real-time scheduling literature, the notion of interference has been employed in many schedulability analysis methods [10], [11], [37], [38], [39], using the following definitions:

- Interference $I_k(a, b)$: the sum of all intervals in which τ_k is ready for execution but cannot execute due to other higher-priority tasks in $[a, b)$.
- Interference $I_{i,k}(a, b)$: the sum of all intervals in which τ_i is executing and τ_k is ready to execute but not executing in $[a, b)$.

With the above definitions, the relation between $I_k(a, b)$ and $I_{i,k}(a, b)$ serves as an important basis for deriving schedulability analysis. In the single-thread task case, it is intuitive to construct such a relation on m processors as follows [10]:

$$I_k(a, b) = \frac{1}{m} \sum_{\tau_i \in \tau} I_{i,k}(a, b). \tag{2}$$

However, it is not straightforward to build such a relation in the multi-thread task case, as illustrated in the following example.

Example 4.1. As an example, suppose that two threads of higher-priority task τ_i and one thread of lower-priority task τ_k are ready for execution on two processors at time t. Then, the two threads of τ_i will run on two processors in $[t, t+1)$, delaying the execution of τ_k. According to the above definitions, τ_i imposes interference on τ_k in $[t, t+1)$, yielding $I_k(t, t+1) = 1$ and $I_{i,k}(t, t+1) = 1$. However, Eq. (2) no longer supports such definitions.

The above example suggests a need for extending the concept of interference for the parallel task model, and this raises three problems: (i) how to calculate the interference on τ_k when only some (but not all) threads of τ_k are interfered, (ii) how to calculate the interference of τ_i on τ_k when only some (but not all) threads of τ_i interfere with τ_k, and (iii) how to calculate the intra-task interference of threads of τ_k on other threads of the same task τ_k.

To address problem (i), we represent the concept of interference of a task τ_k by considering the interference on a chain[3] of its threads. This chain is called a *critical chain*, and threads belonging to a critical chain are called *critical threads*. The detailed description of a critical chain is provided in Section 4.2. With the notion of critical threads, we can now extend the traditional definition of interference towards the DAG task model and introduce a new concept called *critical interference* as follows:

- Critical interference $\mathcal{I}_k(a,b)$: the sum of all intervals in which *a critical thread of* τ_k is ready for execution but cannot execute due to other higher-priority *threads* in $[a,b)$.
- Critical interference $\mathcal{I}_{i,k}(a,b)$: the sum of all intervals in which *at least one thread of* τ_i is executing and *the critical thread of* τ_k is ready to execute but not executing in $[a,b)$.

Note that when all tasks have a single thread, then the single thread is equal to the critical thread and our definition is the same as the traditional definition of interference.

To address problem (ii), we introduce a new concept called *p-depth critical interference*. The p-depth critical interference of a task τ_i on τ_k characterizes not only the length of the delay τ_i causes to τ_k but also the number of threads of τ_i that cause the delay.

To address problem (iii), we incorporate the notion of intra-task interference into both the critical interference and the p-depth critical interference such that they include interference on a critical thread by other non-critical threads of the same task.

In the remainder of this section, we describe the notion of critical threads and address the problems raised the above in detail.

4.2 Necessary Condition for the First Deadline Miss of a DAG Task

In this section we formally define a critical chain of a job. We first seek to identify a necessary condition for any DAG task to miss a deadline on m processors. Let a *last-completing thread* of a job be a thread that completes last among the job's threads. A job misses deadline if its last-completing thread misses deadline. A thread is said to be a *last-completing predecessor* of $\theta_{k,u}$ if it finishes last among all of the predecessors of $\theta_{k,u}$.[4] A thread can only be ready when its last-completing predecessor completes. If we recursively track all the concatenating last completing predecessors from a last completing thread until there is no predecessor, we can construct a *critical* chain λ_k^l of job J_k^l. Each thread in the critical chain is defined as a *critical* thread, and the length of the critical chain of J_k^l, denoted by CP_k^l, is defined as the sum of the WCETs of all its critical threads. A DAG task is then considered as *complete* as soon as its critical threads complete execution.

We define interference on a critical thread $\theta_{k,u} \in \lambda_k$ over interval $[a,b)$ (denoted as $\mathcal{I}_{<k,u>}(a,b)$) as the cumulative length of all intervals in which the critical thread $\theta_{k,u}$ is ready to execute but not executing due to the execution of higher-priority threads that belong to not only other tasks but also the same task. To avoid any confusion, it is worth noting that $\mathcal{I}_{<k,u>}(a,b)$ includes intra-task interference that a critical thread $\theta_{k,u} \in \lambda_k$ receives from other threads $\theta_{k,v} \notin \lambda_k$ of the same task τ_k. According to our definition, $\mathcal{I}_k(a,b)$ is a total interference imposed collectively on all the critical threads of τ_k, i.e.,

$$\mathcal{I}_k(a,b) = \sum_{\theta_{k,u} \in \lambda_k} \mathcal{I}_{<k,u>}(a,b). \tag{3}$$

In order to derive schedulability analysis using the concept of critical interference, we investigate what happens when the "first" deadline miss occurs and identify necessary conditions for a job to miss its deadline. Generally, a deadline miss happens since there is a large amount of higher priority execution that blocks the remaining execution of critical threads of a job until its deadline. We consider any legal sequence of job requests of task set τ, on which EDF misses a deadline. Suppose that a job of task τ_k, denoted by J_k^*, is the first job to miss a deadline among all the jobs of all tasks. Then, by definition, all the jobs of earlier deadlines than the deadline of J_k^* complete execution before their deadlines, and the task system remains underloaded until the first deadline miss. For J_k^*, r_k^* and d_k^* are its release time and deadline. Then, one can see that at least one critical thread $\theta_{k,u} \in \lambda_k^*$ of J_k^* must execute for less than $C_{k,u}$ time units, and the total execution time taken by all critical threads must be less than CP_k^* time units. In order for all critical threads of J_k^* to execute for strictly less than CP_k^* time units over $[r_k^*, d_k^*)$, it is necessary that its critical interference $\mathcal{I}_k(r_k^*, d_k^*)$ be strictly more than $(D_k - CP_k^*)$ time units. This observation yields a necessary condition for job J_k^* to miss a deadline, i.e.,

$$\mathcal{I}_k(r_k^*, d_k^*) > D_k - CP_k^*. \tag{4}$$

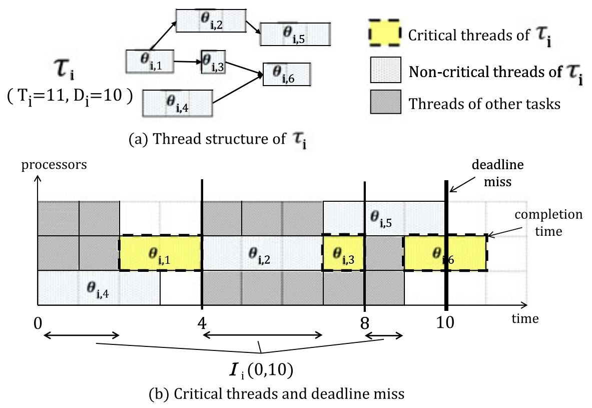

Fig. 2. An example of DAG task τ_i on 3 processors. (a) Task τ_i consists of 6 threads with $T_i = 11$ and $D_i = 10$. (b) A job J_i^l of task τ_i misses a deadline at 10. Here, critical threads are $\theta_{i,1}$, $\theta_{i,3}$, and $\theta_{i,6}$, and those critical threads were blocked in 3 time intervals (i.e., [0,2), [4,7), and [8,9)), respectively. This yields $\mathcal{I}_i(0,10) > D_i - CP_i^l$, where CP_i^l is the sum of the execution times of all critical threads in job J_i^l.

Fig. 2 illustrates a situation where a job of DAG task τ_i misses a deadline. Fig. 2a shows the thread structure of task τ_i and its parameters that we will consider throughout this paper. In Fig. 2b, a job J_i^l of τ_i is released at 0 with a deadline

3. A chain indicates a particular path at run-time.
4. We note that if there are multiple threads that finish last among all of the predecessors of $\theta_{k,u}$ at the same time, we can choose any of them. Without loss of generality, we choose the one that has the lowest index among them.

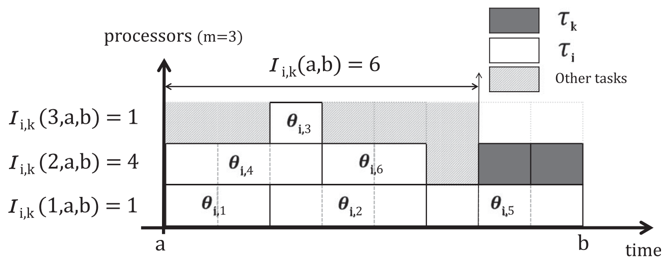

Fig. 3. The notion of p-depth critical interference. Suppose that task τ_k has a lower priority than τ_i. A job of τ_k is released at time instant a, but it cannot execute in $[a, a+6)$ due to the execution of other higher priority tasks. In this example, τ_i executes a single thread in interval $[a+5, a+6)$, which corresponds to 1-depth critical interference on τ_k. This yields $\mathcal{I}_{i,k}(1,a,b) = 1$. τ_i executes two threads in intervals $[a, a+2)$ and $[a+3, a+5)$, leading to $\mathcal{I}_{i,k}(2,a,b) = 4$. And $\mathcal{I}_{i,k}(3,a,b) = 1$.

of 10. Thread $\theta_{i,6}$ is the last completion thread in τ_i, and it is released when $\theta_{i,3}$ has finished at time instant 8 (i.e., $\theta_{i,3}$ is a last completing predecessor of $\theta_{i,6}$). $\theta_{i,1}$ is a last completing predecessor of $\theta_{i,3}$. Since $\theta_{i,1}$ has no predecessor, critical threads of J_i^l are $\theta_{i,1}$, $\theta_{i,3}$, and $\theta_{i,6}$. The execution of those critical threads were delayed in interval [0,2), [4,7), and [8,9), leading to $\mathcal{I}_{<i,1>}(0,10) = 2$, $\mathcal{I}_{<i,3>}(0,10) = 3$, and $\mathcal{I}_{<i,6>}(0,10) = 1$. Then, the critical interference $\mathcal{I}_k(0,10)$ on τ_i during interval [0,10) is $2+3+1=6$, resulting in $\mathcal{I}_k(0,10) > D_i - CP_i^l$. This makes it infeasible for the last thread $\theta_{i,6}$ to fully execute for $C_{i,6}$ time units before the deadline of 10. This leads to the deadline miss of task τ_i.

As shown in Example 4.1, it is not as straightforward as Eq. (2) to build the relation between $\mathcal{I}_k(a,b)$ and $\mathcal{I}_{i,k}(a,b)$. This is mainly because $\mathcal{I}_{i,k}(a,b)$ does not capture how many threads of τ_i interfere with the critical threads of τ_k. We thereby introduce a new concept of p-depth critical interference that characterizes the number of interfering threads, and this new notion will bridge $\mathcal{I}_k(a,b)$ and $\mathcal{I}_{i,k}(a,b)$ effectively for DAG tasks. Let us define the *p-depth critical interference*$\mathcal{I}_{i,k}(p,a,b)$ of task τ_i on task τ_k during interval $[a,b)$ as the cumulative length of all intervals in which (1) a critical thread of τ_k is ready to execute but does not execute and (2) exactly p number of threads of τ_i are executing (see Fig. 3). It is worth noting that when it comes to the intra-task interference case, $\mathcal{I}_{k,k}(p,a,b)$ corresponds to a case where a critical thread of τ_k is not executing while exactly p number of other non-critical threads of τ_k are executing. The p-depth critical interference enables to represent the behavior of parallel execution in more detail, allowing to figure out exactly how many threads of a task τ_i are executing simultaneously when τ_i delays the execution of another task τ_k. A total critical interference $\mathcal{I}_{i,k}(a,b)$ can be decomposed into individual p-depth critical interferences as follows:

$$\mathcal{I}_{i,k}(a,b) = \sum_{p=1}^{m} \mathcal{I}_{i,k}(p,a,b). \tag{5}$$

The p-depth critical interference also makes it easy to constitute a total interference $\mathcal{I}_k(a,b)$ out of individual interferences of each task on task τ_k on m processors as follows.

Lemma 1. *For any work-conserving algorithm, the total critical interference $\mathcal{I}_k(a,b)$ imposed on task τ_k in interval $[a,b)$ is equal to the total amount of contribution of individual threads to the interference on each critical thread divided by the number of processors, i.e.,*

$$\mathcal{I}_k(a,b) = \frac{1}{m}\sum_{\tau_i \in \tau}\sum_{p=1}^{m} \mathcal{I}_{i,k}(p,a,b) \times p. \tag{6}$$

Proof. Since the scheduling algorithm is work-conserving, in the time instants in each of which a critical thread of a task is ready but not executing, each processor must be occupied by all the other threads of another task and including itself. The total amount of the contribution to the critical interference on τ_k is $\sum_{\tau_i \in \tau}\sum_{p=1}^{m} \mathcal{I}_{i,k}(p,a,b) \times p$. If it is divided by the number of processors, we get exactly the length of cumulative intervals in which a critical thread of τ_k is ready to execute but cannot in an interval $[a,b)$. □

Building upon the notion of p-depth critical interference and Lemma 1, the necessary condition for task τ_k to miss a deadline (presented in Eq. (4)) can be rewritten as follows:

$$\frac{1}{m}\sum_{\tau_i \in \tau}\sum_{p=1}^{m} \mathcal{I}_{i,k}(p, r_k^*, d_k^*) \times p + CP_k^* > D_k. \tag{7}$$

Conversely, in order for a task to be schedulable, it is sufficient to demonstrate that for all of its jobs, Eq. (7) cannot be satisfied. Hence to show that task set τ is schedulable under global EDF scheduling, this condition must be checked for each task in τ, which serves as a basis of a schedulability condition. In order to leverage the condition for schedulability analysis properly, we need to calculate the terms in the left-hand side (LHS) of Eq. (7) accurately. Unfortunately, it is hard to compute those terms precisely without knowledge of run-time schedule. Thereby, we wish to derive safe but tight upper bounds on the terms and transform the necessary condition into schedulability tests based on those upper bounds.

5 Workload-Based EDF Schedulability Test

This section derives a polynomial-time schedulability test of global EDF scheduling for DAG tasks. We note that any sporadic task system has infinitely many different legal job arrival sequences. Hence, checking Eq. (7) for all such sequences is computationally intractable. Moreover, it is generally difficult to calculate the interference terms of $\mathcal{I}_{i,k}(p, r_k^*, d_k^*) \times p$ and the critical chain term of CP_k^*, since they are decided at run-time according to the schedules of other tasks that interfere with the job. Therefore, we instead seek to derive upper bounds on the LHS of Eq. (7) for each task.

To this end, we first consider a job J_k having a particular length of its critical chain and derive an upper bound on the interference term $\mathcal{I}_{i,k}(p, r_k, d_k)$ imposed on the job. By definition, a critical chain of a job in τ_k is determined as one of all possible paths in G_k, and its length ranges between the shortest path length in G_k (denoted by SC_k) and the longest path length in G_k (i.e., LC_k). When we derive an upper bound on interference, we use the concept of *workload* that has been widely used in the literature [10], [12], [38], [39], [40]: the workload $W_i(a,b)$ of τ_i is the sum of all intervals in which τ_i is executing in interval $[a,b)$. The interference imposed on a job can be divided into inter-task interference received from threads of other tasks and intra-task interference received from threads of the same task τ_k. We note that bounding inter-task interference is independent of the length of a critical chain, but bounding intra-task interference is dependent on it. We derive upper bounds on inter-task and intra-task interferences, respectively, based on workload. Then, in order to ensure that there exists no job that causes the first deadline miss, we need to check Eq. (7) for all possible lengths of critical chains that a task can have. To this end, we will show that our proposed schedulability test needs to be conducted for only one case, rather than exploring all the possible cases, to get the upper bound of the LHS of Eq. (7) for all jobs.

5.1 Bounding Interference

We define the workload $W_{<i,v>}(a,b)$ of thread $\theta_{i,v}$ in τ_i is the sum of all intervals in which $\theta_{i,v}$ is executing in interval $[a,b)$. Then, the following inequality holds for any J_k of τ_k:

$$\sum_{p=1}^{m} \mathcal{I}_{i,k}(p, r_k, d_k) \times p \le \sum_{\theta_{i,v} \in hp_i(J_k)} W_{<i,v>}(r_k, d_k) \stackrel{\text{def.}}{=} \widehat{W_{i,k}}, \tag{8}$$

where $hp_i(J_k)$ is a set of threads in τ_i that have a priority higher than or equal to a critical thread of J_k.

According to Inequality (8), an upper-bound on the interference term can be obtained by finding an upper-bound on $\widehat{W_{i,k}}$ in any scheduling window of a job of τ_k. Note that when the upper-bound of $\widehat{W_{i,k}}$ is calculated, we assume that there is no deadline miss. This is because our schedulability test aims to derive necessary conditions for the first deadline miss. Therefore, we will assume this for derivation of the upper-bound of $\widehat{W_{i,k}}$ in the rest of the paper. In addition, we are not relying on any particular execution behavior of a task as well as the number of processors when deriving the upper bound on $\widehat{W_{i,k}}$. We next consider two cases to discuss a worst-case release pattern for maximizing $\widehat{W_{i,k}}$: inter-task ($i \neq k$) and intra-task cases ($i = k$).

5.1.1 *Bounding Inter-Task Workload*

To simplify the presentation, we use the following terms. A job is said to be a *carry-in* job of an interval $[a,b)$ if it is released before a but has a deadline within $[a,b)$ or a *body* job if its release time and deadline are both within $[a,b)$.

Worst-Case Release Pattern. Fig. 4 shows a worst-case release pattern, in which task τ_i has the maximum amount of $\widehat{W_{i,k}}$ that interferes with job J_k over interval $[r_k, d_k)$ under global EDF scheduling. As shown in the figure, all the jobs of τ_i are released periodically, and its last body job (J_i') of the interval $[r_k, d_k)$ has a deadline equal to that of J_k (i.e., $d_i' = d_k$). All individual threads in $[r_k, d_k)$ have a priority higher than or equal to J_k and then execute as long as their WCETs. For the carry-in job, we consider a worst-case situation in which all threads of the carry-in job are executed as late as possible subject to satisfying the deadline of the carry-in job. With this release pattern we can include the largest number of threads of τ_i having higher priority than J_k in the interval $[r_k, d_k)$, thus bounding the value of $\widehat{W_{i,k}}$. We recapitulate the above result in the following lemma:

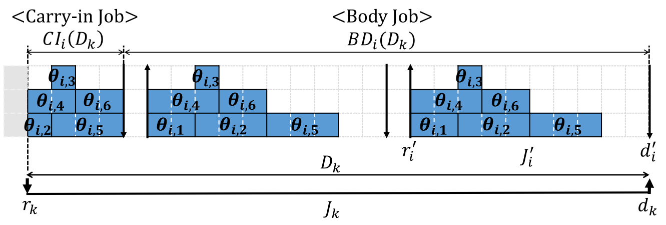

Fig. 4. A worst-case release pattern in which $\widehat{W_{i,k}}$ is maximized.

Lemma 2. *For any task τ_i under the assumption that all jobs meet their deadlines as long as all threads in a job execute at most their WCETs, a release pattern of τ_i that maximizes $\widehat{W_{i,k}}$ is: (a) τ_i releases jobs with a minimum inter-arrival time of T_i time units, (b) the deadline of a job of τ_i aligns with the deadline of the job of τ_k, and (c) all threads of the carry-in job of τ_i are executed as late as possible (right before the deadline of the carry-in job).*

Proof. Fig. 4 illustrates a worst-case release pattern in which task τ_i satisfies requirements (a), (b) and (c).

First of all, the contribution of jobs to $\widehat{W_{i,k}}$ cannot be larger than when the release times of jobs are exactly periodic. That is, moving the release times of some jobs farther apart cannot increase $\widehat{W_{i,k}}$.

In Fig. 4, we then consider what happens to the contribution if we simultaneously shift all the release times and deadlines of τ_i earlier or later. The maximum shift we need to consider in either direction is T_i, since for longer shifts the effect occurs periodically. From Fig. 4, if we shift their deadlines earlier, the contribution of carry-in and body jobs to $\widehat{W_{i,k}}$ cannot increase. The job having its deadline after d_k cannot achieve higher priority than J_k, so it cannot contribute to $\widehat{W_{i,k}}$. Therefore, shifting their deadline earlier cannot increase the maximum contribution to $\widehat{W_{i,k}}$. If we shift their deadlines later, the absolute deadline of the last body job J_i' becomes later than the one of J_k, so it cannot contribute to $\widehat{W_{i,k}}$. Thereby, $\widehat{W_{i,k}}$ is decreased by C_i. The shift may increase the contribution of the carry-in job, but by at most C_i. Therefore, shifting their deadline later cannot increase the maximum contribution to $\widehat{W_{i,k}}$. From the two cases, we can see that the maximum contribution of the body jobs is achieved when a deadline of a job of τ_i aligns with the deadline of the job of τ_k, and the interval of a carry-in job in $[r_k, d_k)$ is maximized.

For the carry-in job of τ_i, only threads executing in $[r_k, d_k)$ can contribute to $\widehat{W_{i,k}}$. If all threads of the carry-in job are executed as late as possible right before the deadline, it can maximize the number of threads executing in $[r_k, d_k)$. It is worthy noting that we assume that all jobs meet their deadlines. Thus, each thread must complete execution so that all of its successive threads are guaranteed to finish before the deadline of the carry-in job although all threads execute for their WCETs. The maximum contribution of the carry-in job is achieved with the release pattern shown in Fig. 4. □

Calculating an Upper Bound on Worst-Case Workload. By using a worst-case release pattern, we can now determine the interval of length D_k which maximizes $\widehat{W_{i,k}}$. Define $\mathcal{W}_{<i,v>}(D_k)$ as the value of $W_{<i,v>}(r_k, d_k)$ in such an interval. Under a worst-case release pattern, there exist only body and carry-in jobs in any interval of length D_k, and all of the threads in those jobs have higher priority than τ_k.

The interval $[r_k, d_k)$ of length D_k can be partitioned into body and carry-in intervals, and the length of the intervals are denoted as $BD_i(D_k)$ and $CI_i(D_k)$, respectively, and described as

$$BD_i(D_k) = \left\lfloor \frac{D_k}{T_i} \right\rfloor \cdot T_i, \tag{9}$$

$$CI_i(D_k) = D_k - BD_i(D_k). \tag{10}$$

Let us consider a bound $\mathcal{W}^{BD}_{<i,v>}(D_k)$ on the body job workload in any interval of length D_k. The maximum number of body jobs of τ_i over an interval of length D_k is $\left\lfloor \frac{D_k}{T_i} \right\rfloor$, and each thread $\theta_{i,v}$ in a body job can fully execute for its WCET. Then, $\mathcal{W}^{BD}_{<i,v>}(D_k)$ is calculated as

$$\mathcal{W}^{BD}_{<i,v>}(D_k) = \left\lfloor \frac{D_k}{T_i} \right\rfloor \cdot C_{i,v}. \tag{11}$$

Now, we consider a bound $\mathcal{W}^{CI}_{<i,v>}(L')$ on the carry-in job workload in any carry-in interval of length L'. For the carry-in job case, only some of threads can contribute to $\mathcal{W}^{CI}_{<i,v>}(L')$ in an interval of length L', and their execution can partially fit into the carry-in interval due to the precedence relation. Thereby, we need to figure out the execution interval of each thread under the worst-case release pattern. Under a worst-case release pattern, all threads of the carry-in job are executed as late as possible as long as all the threads finish their execution before the deadline of the job. We assume that all threads can use as many processors as possible for their execution when we calculate $\mathcal{W}^{CI}_{<i,v>}(L')$. Then, when each thread begins to execute, it exclusively occupies one processor without any interruption until executing for its WCET. We note that contribution of each thread on $\mathcal{W}^{CI}_{<i,v>}(L')$ is maximized under the assumption. If we know each thread's WCET and precedence dependency between threads, we can calculate $\mathcal{W}^{CI}_{<i,v>}(L')$ for each thread $\theta_{i,v}$.

Algorithm 1 shows how to calculate $\mathcal{W}^{CI}_{<i,v>}(L')$ for a carry-in job under the worst-case release pattern shown in Fig. 5b. It computes each thread's starting time and finishing time under the worst-case release pattern. At first, the starting and finishing time of each thread are initialized to D_i (lines 1-3). We reverse the direction of all edges in E_i (denoted by E_i^T) and define DAG G_i^T as $G_i^T = (V_i, E_i^T)$ (line 4). Algorithm 1 then performs a topological sort of G_i^T (line 5). The topological sort is a linear ordering of all threads such that if G_i^T contains a directed edge from $\theta_{i,u}$ to $\theta_{i,v}$, then $\theta_{i,u}$ appears before $\theta_{i,v}$ in the ordering. It traverses each thread in topologically sorted order and calculates its starting and finishing time (lines 6-13). The finishing time $f[u]$ of thread $\theta_{i,u}$ is determined as the earliest one among the starting times of the threads that have the edge to $\theta_{i,u}$ in G_i^T. The starting time $s[u]$ of thread $\theta_{i,u}$ is then determined as its finishing time minus its WCET. We note that this presents the worst-case release pattern for a carry-in job in which all threads of the carry-in job are executed as late as possible. If we compare each thread's starting time and finishing time with the carry-in interval, we can calculate the contribution on $\mathcal{W}^{CI}_{<i,v>}(L')$ for each thread $\theta_{i,v}$ (lines 14-24). In the example shown in Fig. 5a, the threads in τ_i are topologically sorted according to G_i^T. The finishing time $f[1]$ of $\theta_{i,1}$ is determined as 4, which is the starting time of $\theta_{i,2}$, and the finishing time $s[1]$ is determined as 2, which is $f[1] - C_{i,1}$. In Fig. 5b, each thread in τ_i executes its WCET from its starting time to its finishing time.

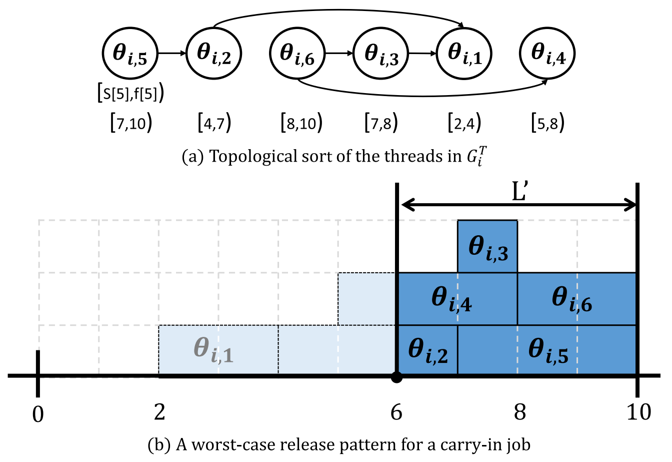

Fig. 5. Calculation of an upper bound on carry-in job workload under a worst-case release pattern.

Algorithm 1. Calculate-CarryinJob-Workload (τ_i, L')

```
1:  for each thread $\theta_{i,u} \in V_i$ do
2:      $s[u] \leftarrow D_i, f[u] \leftarrow D_i$
3:  end for
4:  $G_i^T \leftarrow (V_i, E_i^T)$
5:  topologically sort the threads in $G_i^T$
6:  for each thread $\theta_{i,u}$, taken in topologically sorted order do
7:      for each thread $\theta_{i,v} \in Parent[u]$ do
8:          if $s[v] < f[u]$ then
9:              $f[u] \leftarrow s[v]$
10:         end if
11:     end for
12:     $s[u] \leftarrow f[u] - C_{i,u}$
13: end for
14: for each thread $\theta_{i,u}$ in $G_i^T$ do
15:     if $s[u] \geq D_i - L'$ then
16:         $\mathcal{W}^{CI}_{<i,u>}(L') = C_{i,u}$
17:     else
18:         if $s[u] < D_i - L'$ and $f[u] > D_i - L'$ then
19:             $\mathcal{W}^{CI}_{<i,u>}(L') = f[u] - (D_i - L')$
20:         end if
21:     else
22:         $\mathcal{W}^{CI}_{<i,u>}(L') = 0$
23:     end if
24: end for
```

A bound on the workload $W_{<i,v>}(r_k, d_k)$ that will contribute to the worst case as shown in Fig. 4, for $i \neq k$, is expressed as follows:

$$\mathcal{W}_{<i,v>}(D_k) = \mathcal{W}^{CI}_{<i,v>}(CI_i(D_k)) + \mathcal{W}^{BD}_{<i,v>}(BD_i(D_k)). \tag{12}$$

We note that both $\mathcal{W}^{CI}_{<i,v>}(CI_i(D_k))$ and $\mathcal{W}^{BD}_{<i,v>}(BD_i(D_k))$ in Eq. (12) are independent of a critical chain of a job of τ_k. This means that we always get the same result of $\mathcal{W}^{CI}_{<i,v>}(CI_i(D_k))$ and $\mathcal{W}^{BD}_{<i,v>}(BD_i(D_k))$, no matter what a critical chain of length is. We can then compute bounds on the amount of inter-task interference on any job of τ_k as follows:

$$\widehat{W_{i,k}} \le \sum_{\theta_{i,v}\in V_i} \mathcal{W}_{<i,v>}(D_k). \tag{13}$$

5.1.2 Bounding Intra-Task Workload

The critical threads in a critical chain of a job of τ_k can get interference from the other threads belonging to the same job. For the intra-task interference, the worst-case release pattern is already determined as the execution window of a job of τ_k. Then, all threads within a single job except the critical threads can interfere on the critical threads, and it is clear that the interference of a single thread $\theta_{k,v}$ on the critical threads is upper bounded by $C_{k,v}$. Thereby, the sum of WCETs of all threads within a single job except the ones of critical threads can contribute on $\widehat{W_{k,k}}$.

Unlike inter-task workload, intra-task workload is dependent on the length of a critical chain of a job. We define $\mathcal{W}_k(x)$ as a bound on intra-task workload of a job with a critical chain of length x. The sum of WCETs of all threads within a single job of τ_k except the ones of the critical threads can contribute to $\mathcal{W}_k(x)$. We recall that the critical chain length x is the sum of WCETs of the critical threads, so $\mathcal{W}_k(x)$ is computed as

$$\mathcal{W}_k(x) = C_k - x. \tag{14}$$

We can then compute bounds on the amount of intra-task interference on a job with a critical chain of length x as

$$\widehat{W_{k,k}} \le \mathcal{W}_k(x). \tag{15}$$

5.2 Deriving a Schedulability Test

Putting together inter-task and intra-task workloads as an upper-bound on the interference on a job of τ_k with a critical chain of length x ($SC_k \le x \le LC_k$), we can check the schedulability of the job for a given task set under global EDF scheduling as follows.

Lemma 3. *Suppose that a task set τ is scheduled by global EDF scheduling on m identical processors. Then, a job invoked by $\tau_k \in \tau$ having a critical chain length x ($SC_k \le x \le LC_k$) does not cause the first deadline miss, if the following inequality holds:*

$$\frac{1}{m}\Big(\sum_{i\neq k}\sum_{\theta_{i,v}\in V_i}\mathcal{W}_{<i,v>}(D_k) + \mathcal{W}_k(x)\Big) + x \le D_k. \tag{16}$$

Proof. From Lemma 2 and Inequality (8), the following inequality holds for a job of τ_k having a critical chain length of x:

$$\frac{1}{m}\sum_{\tau_i\in\tau}\sum_{p=1}^{m}\mathcal{I}_{i,k}(p, r_k, d_k)\times p + x \le \frac{1}{m}\Big(\sum_{i\neq k}\sum_{\theta_{i,v}\in V_i}\mathcal{W}_{<i,v>}(D_k) + \mathcal{W}_k(x)\Big) + x. \tag{17}$$

Then, if Eq. (16) is satisfied for the job, the job fails to satisfy the necessary condition of Eq. (7) that triggers the first deadline miss. □

In order to derive a schedulability test for a task set, it should be guaranteed that all individual jobs satisfy Lemma 3. Then, we need to check Eq. (16) for all possible critical chain lengths of each task τ_k ranging between SC_k and LC_k. We claim that our workload-based schedulability test needs to be conducted for only one critical chain length, which is the longest path length (i.e., LC_k).

Lemma 4. *For a given task set τ, a task τ_k satisfies the followings:*

$$\frac{1}{m}\Big(\sum_{i\neq k}\sum_{\theta_{i,v}\in V_i}\mathcal{W}_{<i,v>}(D_k) + \mathcal{W}_k(x)\Big) + x \le \frac{1}{m}\Big(\sum_{i\neq k}\sum_{\theta_{i,v}\in V_i}\mathcal{W}_{<i,v>}(D_k) + \mathcal{W}_k(LC_k)\Big) + LC_k. \tag{18}$$

Proof. The term $\frac{1}{m}\sum_{i\neq k}\sum_{\theta_{i,v}\in V_i}\mathcal{W}_{<i,v>}(D_k)$ is included in both LHS and RHS of inequality (18). Therefore, we show that $\frac{1}{m}\mathcal{W}_k(x) + x \le \frac{1}{m}\mathcal{W}_k(LC_k) + LC_k$ as follows.

$$\begin{aligned}
&\frac{1}{m}\mathcal{W}_k(x) + x\\
&= \frac{1}{m}(C_k - x + mx)\\
&\quad \text{(From Eq. (14))}\\
&= \frac{1}{m}(C_k - x + mx + LC_k - LC_k + mLC_k - mLC_k)\\
&= \frac{1}{m}(C_k - LC_k + mLC_k + (LC_k - x) - m(LC_k - x))\\
&\quad \text{(Re-arrange the terms)}\\
&\le \frac{1}{m}(C_k - LC_k + mLC_k)\\
&\quad (\because (LC_k - x) \le m(LC_k - x))\\
&\quad (\because \text{By definition, } x \le LC_k \text{ and } m > 0)\\
&= \frac{1}{m}\mathcal{W}_k(LC_k) + LC_k\\
&\quad \text{(From Eq. (14)).}
\end{aligned}$$

□

Finally, we develop a workload-based schedulability test for DAG tasks under global EDF scheduling.

Theorem 1. *A task set τ is schedulable under global EDF scheduling on m identical processors if for each task $\tau_k \in \tau$, the following inequality holds:*

$$\sum_{i\neq k}\sum_{\theta_{i,v}\in V_i}\mathcal{W}_{<i,v>}(D_k) + \mathcal{W}_k(LC_k) \le m(D_k - LC_k). \tag{19}$$

Proof. From Lemmas 3 and 4, $\frac{1}{m}\Big(\sum_{i\neq k}\sum_{\theta_{i,v}\in V_i}\mathcal{W}_{<i,v>}(D_k) + \mathcal{W}_k(LC_k)\Big) + LC_k$ is an upper bound on the LHS of Eq. (7) for all jobs of a task τ_k. Then, if Eq. (19) is satisfied for all tasks in a task set τ, no job can trigger the first deadline miss, and the task set is schedulable under global EDF scheduling on m identical processors. □

Complexity. We denote the number of tasks in a task set by n. For each task τ_i, Algorithm 1 is performed in $O(|V_i| + |E_i|)$ time to calculate an upper bound on carry-in workload. The longest path length LC_i can be calculated in $O(|V_i| + |E_i|)$ time. This repeats n times for all tasks in a task set to calculate LC_i. Independently, it requires $O(n)$ time to calculate Eq. (19) for a given τ_k, and it repeats n times for all tasks in a task set to check schedulability of a task set. Therefore, the schedulability test in Theorem 1 requires $\max\{O(n(|V_{max}| + |E_{max}|)), O(n^2)\}$ time for a task set, where $|V_{max}|$ and $|E_{max}|$ represents the largest $|V_i|$ and $|E_i|$ among all tasks $\tau_i \in \tau$.

6 Slack-Based Iterative Schedulability Test

In general, the computation of workload bounds in Section 5 involves much pessimism due to the overestimation on calculating the worst-case workload of a carry-in job. In particular, the worst-case release pattern for a carry-in job shown in Fig. 5b does not consider the fact that a task can finish its execution earlier than its deadline. Once we identify such an early completion of a task's execution, we can reduce the amount of workload of a carry-in job using slack values, where the slack S_k of task τ_k is defined as a length of the minimum time interval between finishing time and deadline of a job of task τ_k. The idea of exploiting slack values is introduced to reduce such pessimism effectively for the sequential task model [11], [37]. In this section, building upon our workload-based schedulability test, we derive an improved schedulability test by taking advantage of slack values for DAG tasks. To this end, we first introduce how to calculate slack values for each task in the following lemma.

Lemma 5. *The slack of task τ_k is given by*

$$S_k = D_k - CP_k^* - \left\lceil \frac{\sum_{\tau_i \in \tau} \sum_{p=1}^{m} \mathcal{I}_{i,k}(p, r_k^*, d_k^*) \times p}{m} \right\rceil, \quad (20)$$

if $S_k > 0$.

Proof. We re-arrange Eq. (20) as follows:

$$\begin{aligned}
&\left\lceil \frac{\sum_{\tau_i \in \tau} \sum_{p=1}^{m} \mathcal{I}_{i,k}(p, r_k^*, d_k^*) \times p}{m} \right\rceil = D_k - CP_k^* - S_k \\
&\Rightarrow \frac{\sum_{\tau_i \in \tau} \sum_{p=1}^{m} \mathcal{I}_{i,k}(p, r_k^*, d_k^*) \times p}{m} \le D_k - CP_k^* - S_k \\
&\Leftrightarrow \mathcal{I}_k(r_k^*, d_k^*) \le D_k - CP_k^* - S_k \\
&\quad (\because \text{By Lemma 1}) \\
&\Leftrightarrow \mathcal{I}_k(r_k^*, d_k^*) + CP_k^* \le D_k - S_k.
\end{aligned} \quad (21)$$

According to Eq. (21), the sum of the critical interference and total execution time taken by all critical threads is less than or equal to $D_k - S_k$. Since we assume that $S_k > 0$, task τ_k has already finished its execution at S_k time units ahead of its deadline. □

By exploiting the bounds on the interference derived in Section 5, a lower bound S_k^{lb} on the slack S_k of a task τ_k under global EDF scheduling is then given by

$$S_k^{lb} = D_k - LC_k - \left\lceil \frac{1}{m} \Big(\sum_{i \ne k} \sum_{\theta_{i,v} \in V_i} \mathcal{W}_{<i,v>}(D_k) + \mathcal{W}_k(LC_k) \Big) \right\rceil, \quad (22)$$

when this term is positive.

We now exploit slack values for reducing the pessimism calculating the worst-case carry-in job workload of τ_k. When a lower bound on the slack of a task is available, it is possible to give a tighter upper bound on interference. If the value of S_k^{lb} is positive, every job of τ_k will complete at least S_k^{lb} time units before its deadline. Using this information, we can reduce pessimism on calculating the worst-case carry-in job workload of τ_k shown in Algorithm 1. It can be easily incorporated by replacing the initial starting and finishing times of each thread (line 2 in Algorithm 1) with $s[u] \leftarrow D_i - S_i^{lb}$ and $f[u] \leftarrow D_i - S_i^{lb}$. Then, a worst-case workload can be calculated in a similar way to Eq. (12), and we denote the workload incorporating a lower bound on the slack of task τ_i by $\mathcal{W}_{<i,v>}(D_k, S_i^{lb})$. Note that when a lower bound on S_i^{lb} is not known, we can simply use $S_i^{lb} = 0$. Finally, a lower bound on the slack of a task τ_k under global EDF scheduling on m identical processors is given by

$$S_k^{lb} = D_k - LC_k - \left\lceil \frac{1}{m} \Big(\sum_{i \ne k} \sum_{\theta_{i,v} \in V_i} \mathcal{W}_{<i,v>}(D_k, S_i^{lb}) + \mathcal{W}_k(LC_k) \Big) \right\rceil, \quad (23)$$

when this term is positive.

Algorithm 2. Slack-Based Iterative Schedulability Test (τ)

```
1: $Updated \leftarrow true, Nround \leftarrow 0$
2: for each task $\tau_k \in \tau$ do
3:     $S_k^{lb} \leftarrow 0$
4: end for
5: repeat
6:     $Feasible \leftarrow true$
7:     $Updated \leftarrow false$
8:     for $k \leftarrow 1$ to $|\tau|$ do
9:         $NewBound \leftarrow$ Eq. (23)
10:        if $NewBound < 0$ then
11:            $Feasible \leftarrow false$
12:        else
13:            if $NewBound > S_k^{lb}$ then
14:                $S_k^{lb} \leftarrow NewBound$
15:                $Updated \leftarrow true$
16:            end if
17:        end if
18:    end for
19:    $Nround++$
20:    if $Feasible$ is $true$ then
21:        return schedulable
22:    end if
23: until $Updated$ is $true$ and $Nround \le NroundLimit$
24: return unschedulable
```

Then, the approaches [11], [37] of exploiting a slack value can be adopted into our workload-based schedulability test presented in Theorem 1. Therefore, we derive a slack-based iterative schedulability test of a task set under global EDF

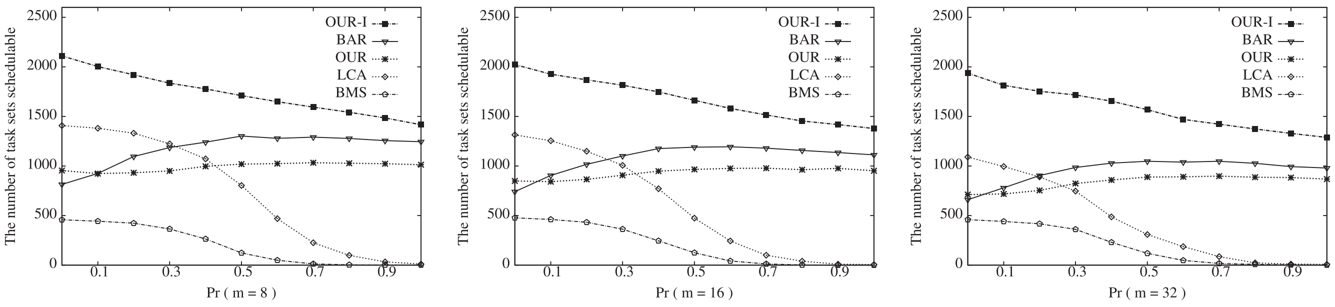

Fig. 6. Schedulability with different values of Pr for DAG tasks.

scheduling, and it is presented in Algorithm 2. As shown in Algorithm 2, for every task in a task set, a lower bound value on the slack of the task is initialized to zero (line 3). Eq. (23) is used to compute a new value of the lower bound on the slack of a task (line 9). If the computed value is negative, the task is considered to be unschedulable (line 11). If it is positive, the lower bound is accordingly updated (lines 13-15). It is repeated for every task in the task set until the lower bound on the slack for every task is not updated (lines 5-23). If no computed value is negative, the task set is deemed schedulable. If the iteration stops, the task set is deemed unschedulable (line 24).

Complexity. Similar to the workload-based schedulability test in Section 4, the slack-based iterative schedulability test requires $O(n(|V_{max}| + |E_{max}|))$ time to calculate LC_i for all tasks. Additionally, the complexity of the iterative test depends on the number of iterations of slack updates. A single iteration of slack updates requires $O(n^2)$ time. The total number of iterations of the *repeat* cycle at line 5 is upper-bounded by $O(n \cdot \max_{\tau_i \in \tau} D_i)$ [37]. Therefore, the overall time complexity is $\max\{O(n(|V_{max}| + |E_{max}|)), O(n^3 \cdot \max_{\tau_i \in \tau} D_i)\}$. We note that the complexity can be significantly reduced if the test is stopped after a finite number $NroundLimit$ of iterations. If this is the case, the overall complexity becomes $\max\{O(n(|V_{max}| + |E_{max}|)), O(n^2 \cdot NroundLimit)\}$. However, limiting the number of iterations may reduce the number of schedulable task sets, rejecting some feasible task set that could be deemed schedulable after a few more iterations. We examine such schedulability loss for different values of $NroundLimit$ in Section 7.

7 Evaluation

In this section, we present simulation results to evaluate our global EDF schedulability analysis that is directly applicable to a set of parallel tasks.

7.1 Simulation Environment

We generate DAG tasks mainly based on the method used in [41]. For a DAG task τ_i, its parameters are determined as follows. Period and deadline of τ_i ($T_i = D_i$)[5] are uniformly chosen in $[100, 1000]$. The number of nodes (threads) N_i is uniformly chosen in $[1, 30]$. For each pair of nodes, an edge is generated with the probability of Pr; for each edge, its orientation is chosen to ensure acyclicity. For individual threads $\theta_{i,u}$, the WCET ($C_{i,u}$) is randomly selected in the range of $[1, T_i/N_i]$.

5. In this section, we only show the results of implicit deadline DAG tasks because some of related works consider the implicit deadline task case only, but our proposed analysis shows similar behaviors in constrained deadline DAG tasks compared to those of implicit ones.

In order to understand how our proposed approaches perform with DAG tasks, we experiment by varying the following parameters: the degree of parallelism in a DAG task and the number of processors. In addition, we examine the effect of $NroundLimit$ in our slack-based iterative schedulability test presented in Section 6. Other experiments when varying the number of threads and variation of WCETs among threads in a DAG task are reported in the supplement, available online.

In each experiment, we compare our proposed schedulability tests with some related methods (shown in Table 1) for the DAG task model under global EDF scheduling. More specifically, we consider the following schedulability tests:

- our workload-based schedulability test in Theorem 1 (denoted by OUR)
- our slack-based iterative schedulability test allowing the maximum number of iterations in Algorithm 2 (denoted by OUR-I)
- the EDF schedulability test in [32] (denoted by BAR).
- the capacity augmentation bound (i.e., Theorem 4) in [31] (denoted by LCA)
- the EDF schedulability test (i.e., Theorem 21) in [29] (denoted by BMS).

As mentioned in Section 2, the five schedulability tests shown in Fig. 6 are classified as independent schedulability analysis. OUR-I and BAR are of pseudo-polynomial time complexity, while OUR, LCA, and BMS are of polynomial time complexity.

We note that other related analysis techniques are not included in our evaluation, because the one in [30] is applicable only to the single DAG task case, while the multiple parallel task case is of our interest. The capacity augmentation bound in [28] is excluded in our evaluation, because LCA is the best known capacity augmentation bound of EDF. In [29], a pseudo-polynomial time schedulability test was also proposed in addition to BMS, but it is not included in this comparison because BAR strictly dominates the pseudo-polynomial test. We also note that the resource augmentation bounds in [28], [29] are not included in this comparison, because those bounds can serve as schedulability tests only when an optimal schedule is known. However, no optimal schedule for parallel tasks has been developed so far, and therefore, it is difficult (if not impossible) to check the feasibility of a task set through simulation.

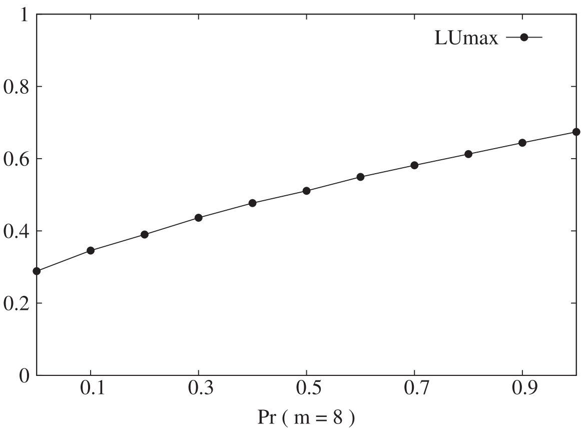

Fig. 7. LU_{max} as Pr changes.

7.2 Simulation Results

Effect of the Degree of Parallelism. Our first simulations were performed to evaluate our approaches with different degrees of intra-task parallelism. We generate 4,000 task sets for $m = 8$, 16, and 32, where m is the number of processors, yet leaving Pr undetermined, as follows.

S1 We first generate a seed task set with two tasks with the parameters determined as described above.

S2 If the system utilization U_{sys} (i.e., $U_{sys} = \sum_{\tau_i \in \tau} U_i$) of the seed task set is greater than m, we discard this seed set and go to Step S1.

S3 We include this seed set for simulation. We then add $m/4$ more task into the seed set and go to Step S2 until 4,000 task sets are generated.

We now consider constructing edges between nodes (i.e., precedence dependency between threads) with the probability parameter $0 \leq Pr \leq 1$. When $Pr = 0$, there is no edge and thereby no thread has predecessors, maximizing the degree of intra-task parallelism. In contrast, with $Pr = 1$, each node is fully connected to all the other nodes, representing no single thread can execute in parallel with any other threads in the same task. An increasing value of Pr generates a growing number of edges in each DAG task, leading to a greater degree of precedence constraints between nodes but a smaller degree of intra-task parallelism. Thereby, as Pr increases, each task is highly likely to have a larger longest path length LC_i. We define LU_{max} as the maximum LC_i/T_i among the tasks $\tau_i \in \tau$, and Fig. 7 shows a tendency for LU_{max} as Pr increases. In order to run simulation for different degrees of intra-task parallelism, we perform simulation with 4,000 task sets in 11 different cases in terms of Pr, where we increase Pr from 0.0 to 1.0 in the step of 0.1, resulting in 44,000 simulations.

Fig. 6 compares different schedulability tests (OUR, OUR-I, BAR, LCA, and BMS) in terms of the number of task sets deemed schedulable with different values of Pr for $m = 8$, 16, and 32. The figure shows that OUR-I outperforms the other existing methods throughout all the values of Pr. In cases of $m = 8$, 16, and 32, OUR-I finds 47, 54, and 65 percent more schedulable task sets than BAR does, and OUR-I is also shown to improve schedulability, compared to OUR, by 75, 80, and 89 percent more, respectively. OUR finds 35, 60, and 90 percent additional task sets, which are deemed unschedulable by LCA for $m = 8$, 16, and 32, respectively. We note that OUR-I finds almost all task sets deemed schedulable by the other tests except only 124, 12, and 1 task sets for $m = 8$, 16, and 32, respectively. We also note that OUR-I shows better performance compared to other tests for cases with a larger number of processors.[6]

Looking at pseudo-polynomial-time schedulability tests (OUR-I and BAR), it is interesting to see that the performance gap between OUR-I and BAR becomes larger with a smaller value of Pr. Looking at polynomial-time schedulability tests (OUR, LCA, and BMS), when Pr increases the two methods of LCA and BMS are shown to perform worse, while OUR is relatively much insensitive to the value of Pr. Thereby, the performance gap between OUR and those two methods (LCA and BMS) becomes larger as the degree of precedence constraint increases.

To understand such performance results of OUR and OUR-I as Pr varies, we examine the effect of the degree of parallelism on schedulability with respect to our interference-based analysis. (i) In the perspective of a task τ_k receiving interference from other tasks in a task set, LC_k generally decreases when the degree of parallelism increases (i.e., when Pr decreases). This gives task τ_k more room to accommodate larger interference from other tasks, likely leading to better schedulability. (ii) On the other hand, in the perspective of a task τ_i imposing interference on τ_k, the larger number of threads of τ_i has a chance to delay execution of τ_k at the same time when the degree of parallelism increases, leading to an increase in interference on τ_k. Moreover, to derive a safe upper bound of the interference suffered by τ_k, OUR assumes that every higher priority task τ_i has carry-in. This is an over-pessimistic assumption, since in a real scheduling sequence, it may be the case that some task τ_i's carry-in job has finished before the beginning of the execution window of τ_k, thus it actually does not contribute any carry-in to the interference on τ_k. Such pessimism on bounding interference increases when the degree of parallelism increases. Statements (i) and (ii) are conflicting with each other in terms of schedulability. Fig. 6 shows that the performance of OUR stays insensitive to Pr with Statements (i) and (ii) having comparable impacts on schedulability.

The figure also shows that OUR-I can compensate for the loss of performance from the pessimism involved in bounding interference, and the benefit of slack-based iterative method is getting bigger with a smaller value of Pr. Based on statement (i), it is highly likely to have larger slack values of τ_k when the degree of parallelism increases. Then, those slack values effectively reduce the pessimism associated with the estimation of the carry-in of an interfering task according to statement (ii), which leads to better schedulability. Such an effect makes a larger performance gap between OUR-I and BAR with a smaller value of Pr.

The two existing polynomial-time tests (LCA and BMS) are sensitive to LU_{max} due to their schedulability analysis, since those two methods share in common that their schedulability tests check whether LU_{max} is smaller than or equal to some threshold (e.g., $(3+\sqrt{5})/2$ in LCA, $1/3$ in BMS). A larger value of Pr generally increases the maximum LU_{max} for a task set (as shown in Fig. 7), leading to worse schedulability.

6. We also conducted simulations for $m = 48$, but the results are shown in the supplement, available online, since the trends are similar.

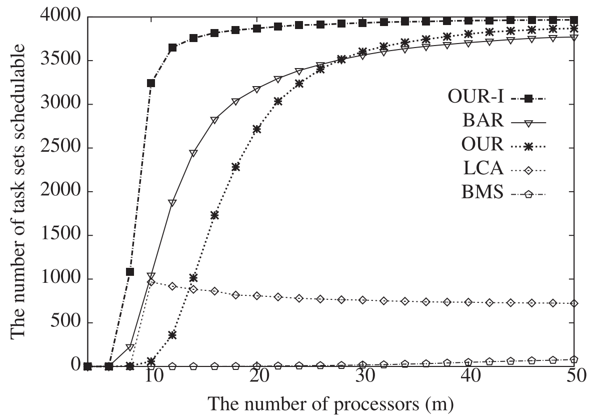

Fig. 8. Schedulability with different number of processors.

Effect of the Number of Processors. Our second simulations were performed to show schedulability with a different number of processors. We generate 4,000 task sets whose utilization U_{sys} is in $[3.9, 4.1]$. Each task set is obtained by repeatedly adding tasks until the system utilization is in $[3.9, 4.1]$ while each individual task is generated with the parameters described in Section 7.1 and the value of *Pr* fixed as 0.5.[7]

Fig. 8 shows the number of task sets deemed schedulable when the number of processors (m) is varied from 4 to 50. OUR-I significantly outperforms the other tests. OUR-I requires a much lower number of processors (around 12) to schedule 90 percent of the task sets while OUR and BAR require more than 30 processors to do so. LCA and BMS behave even worse in the sense that they cannot admit a large portion of the generated task sets even with a very large number of processors. This is because their schedulability analysis is highly dependent on the value of LU_{max} irrelevant to the number of processors, as discussed in the first simulation results. We note that the running time of BAR increases as m increases by reflecting its analytical procedure shown in [32] while the running time of other tests including OUR-I and OUR is relatively stable to the number of processors. In particular, the average running time of OUR-I was 0.5 ms for all values of m, while the average running time of BAR was increased from 1 to 121 ms when changing m from 4 to 50.

Effect of $NroundLimit$ *in* OUR-I. Our third simulations were performed to investigate the schedulability loss of OUR-I for different values of $NroundLimit$: 1, 2, 4, 8, 16. We ran simulations on the same task sets used for Figure 6 with $m = 8$, 16, and 32, and Table 2 shows the schedulability ratio of OUR-I with different values of $NroundLimit$ relative to the case of the maximum value of $NroundLimit$ (i.e., $NroundLimit = n \cdot \max_{\tau_i \in \tau} D_i$). For example, for $NroundLimit = 1$, OUR-I finds a solution 73.9, 70.0, and 66.4 percent close to the case of the maximum value of $NroundLimit$ when $m = 8$, 16, and 32, respectively. When two slack updates for each task are allowed (i.e., $NroundLimit = 2$), the number of task sets deemed schedulable by OUR-I increases rapidly. With $NroundLimit = 16$, OUR-I finds every task set that can be detected using an unbounded $NroundLimit$ for all $m = 8$, 16, and 32. We note that the average running time of OUR-I with $NroundLimit = 16$ was 0.4 ms to check the schedulability of a task set, while the average running time of BAR was 11.1 ms when $m = 8$.

TABLE 2
Schedulability Ratio of OUR − I with Different Values of $NroundLimit$ Relative to the Maximum Value of $NroundLimit$

	m	Iteration limit ($NroundLimit$)				
		1	2	4	8	16
Schedulability ratio (%)	8	73.9	97.7	99.9	100	100
	16	70.0	97.7	99.9	100	100
	32	66.4	97.3	99.9	99.9	100

Summary. In summary, OUR outperforms the other existing polynomial-time schedulability tests. In addition, OUR-I significantly improves the schedulability of EDF with good efficiency and thus shows the best performance compared to the state-of-the-art independent schedulability tests available for DAG tasks. We identify that our proposed schedulability tests are adaptive to different degrees of intra-task parallelism and scalable to the number of processors in terms of both performance and complexity.

8 Conclusion

The motivation for our work was the desire to understand the thread-level parallelism of DAG tasks in the context of hard real-time multi-core scheduling. In this paper, we extended the notion of interference formalizing it at a finer-grained thread level and building a connection to the notion at a task level. We then generalized interference-based analysis methods according to the new proposed notion of interference, introducing global EDF schedulability conditions that are directly applicable to a set of DAG tasks. Our evaluation results showed that it significantly improves the state-of-the-art analysis techniques available for parallel tasks.

This paper incorporated thread-level parallelism directly into schedulability analysis focusing on the global EDF algorithm. However, we believe the schedulability of parallel tasks can be advanced much more significantly if thread-level parallelism is directly reflected into scheduling algorithms as well. Hence, a direction of our future work includes developing new real-time scheduling algorithms that support intra-task parallelism and synchronization directly.

Acknowledgment

This work was supported in part by IITP (B0101-15-0557), BSRP (NRF-2015R1D1A1A01058713, NRF-2014R1A1A1035827), NCRC (2010-0028680), and NRF (2015M3A9A7067220) funded by the Korea Government (MEST/MSIP/MOTIE). This work was also funded in part by MoE Tier-2 grant (MOE2013-T2-2-029, ARC9/14) of the Singapore Government.

References

[1] Intel, "Intel Xeon Phi product family," (2016). [Online]. Available: http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html

7. If the system utilization exceeds the limit in generating a task set, we discard the lastly added task, add a new task, and repeat the procedure until the system utilization is in the desired range.

[2] Cavium, "ThunderX ARM processors," (2016). [Online]. Available: http://www.cavium.com/ThunderX_ARM_Processors.html
[3] OpenMP, "Openmp," 1997. [Online]. Available: http://openmp.org
[4] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the Cilk-5 multithreaded language," *ACM SIGPLAN Notices*, vol. 33, no. 5, pp. 212–223, 1998.
[5] J. Reinders, *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. Sebastopol, CA, USA: O'Reilly Media Inc., 2007.
[6] K.-F. Faxén, "Wool user's guide," (2015). [Online]. Available: https://www.sics.se/~kff/wool/users-guide.pdf
[7] B. L. Chamberlain, D. Callahan, and H. P. Zima, "Parallel programmability and the chapel language," *Int. J. High Performance Comput. Appl.*, vol. 21, no. 3, pp. 291–312, 2007.
[8] C. Liu and J. Layland, "Scheduling algorithms for multi-programming in a hard-real-time environment," *J. ACM*, vol. 20, no. 1, pp. 46–61, 1973.
[9] T. P. Baker, "Multiprocessor EDF and deadline monotonic schedulability analysis," in *Proc. IEEE Real-Time Syst. Symp.*, 2003, pp. 120–129.
[10] M. Bertogna, M. Cirinei, and G. Lipari, "Improved schedulability analysis of EDF on multiprocessor platforms," in *Proc. Euromicro Conf. Real-Time Syst.*, 2005, pp. 209–218.
[11] M. Bertogna and M. Cirinei, "Response-time analysis for globally scheduled symmetric multiprocessor platforms," in *Proc. IEEE Real-Time Syst. Symp.*, 2007, pp. 149–160.
[12] S. Baruah, "Techniques for multiprocessor global schedulability analysis," in *Proc. IEEE Real-Time Syst. Symp.*, 2007, pp. 119–128.
[13] N. Guan, M. Stigge, W. Yi, and G. Yu, "New response time bounds of fixed priority multiprocessor scheduling," in *Proc. IEEE Real-Time Syst. Symp.*, 2009, pp. 387–397.
[14] R. Davis and A. Burns, "Priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems," in *Proc. IEEE Real-Time Syst. Symp.*, 2009, pp. 398–409.
[15] S. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel, "Proportionate progress: A notion of fairness in resource allocation," *Algorithmica*, vol. 15, no. 6, pp. 600–625, 1996.
[16] G. Levin, S. Funk, C. Sadowski, I. Pye, and S. Brandt, "DP-FAIR: A simple model for understanding optimal multiprocessor scheduling," in *Proc. Euromicro Conf. Real-Time Syst.*, 2010, pp. 3–13.
[17] P. Regnier, G. Lima, E. Massa, G. Levin, and S. A. Brandt, "Run: Optimal multiprocessor real-time scheduling via reduction to uniprocessor," in *Proc. IEEE Real-Time Syst. Symp.*, 2011, pp. 104–115.
[18] G. Nelissen, V. Berten, V. Nelis, J. Goossens, and D. Milojevic, "U-EDF: An unfair but optimal multiprocessor scheduling algorithm for sporadic tasks," in *Proc. Euromicro Conf. Real-Time Syst.*, 2012, pp. 13–23.
[19] P. Jayachandran and T. Abdelzaher, "Transforming distributed acyclic systems into equivalent uniprocessors under preemptive and non-preemptive scheduling," in *Proc. Euromicro Conf. Real-Time Syst.*, 2008, pp. 233–242.
[20] N. Serreli, G. Lipari, and E. Bini, "The demand bound function interface of distributed sporadic pipelines of tasks scheduled by EDF," in *Proc. Euromicro Conf. Real-Time Syst.*, 2010, pp. 187–196.
[21] K. Lakshmanan, S. Kato, and R. Rajkumar, "Scheduling parallel real-time tasks on multi-core processors," in *Proc. IEEE Real-Time Syst. Symp.*, 2010, pp. 259–268.
[22] A. Saifullah, K. Agrawal, C. Lu, and C. Gill, "Multi-core real-time scheduling for generalized parallel task models," in *Proc. IEEE Real-Time Syst. Symp.*, 2011, pp. 217–226.
[23] H. S. Chwa, J. Lee, K.-M. Phan, A. Easwaran, and I. Shin, "Global EDF schedulability analysis for synchronous parallel tasks on multicore platforms," in *Proc. Euromicro Conf. Real-Time Syst.*, 2013, pp. 25–34.
[24] B. Andersson and D. de Niz, "Analyzing global-EDF for multiprocessor scheduling of parallel tasks," in *Proc. Int. Conf. Principles Distrib. Syst.*, 2012, pp. 16–30.
[25] C. Maia, M. Bertogna, L. Nogueira, and L. M. Pinho, "Response-time analysis of synchronous parallel tasks in multiprocessor systems," in *Proc. Int. Conf. Real-Time Netw. Syst.*, 2014, Art. no. 3.
[26] G. Nelissen, V. Berten, J. Goossens, and D. Milojevic, "Techniques optimizing the number of processors to schedule multi-threaded tasks," in *Proc. Euromicro Conf. Real-Time Syst.*, 2012, pp. 321–330.
[27] P. Axer, S. Quinton, M. Neukirchner, R. Ernst, B. Dobel, and H. Hartig, "Response-time analysis of parallel fork-join workloads with real-time constraints," in *Proc. Euromicro Conf. Real-Time Syst.*, 2013, pp. 215–224.
[28] J. Li, K. Agrawal, C. Lu, and C. D. Gill, "Analysis of global EDF for parallel tasks," in *Proc. Euromicro Conf. Real-Time Syst.*, pp. 3–13.
[29] V. Bonifaci, A. Marchetti-Spaccamela, S. Stiller, and A. Wiese, "Feasibility analysis in the sporadic DAG task model," in *Proc. Euromicro Conf. Real-Time Syst.*, 2013, pp. 225–233.
[30] S. Baruah, V. Bonifaci, A. Marchetti-Spaccamela, L. Stougie, and A. Wiese, "A generalized parallel task model for recurrent real-time processes," in *Proc. IEEE Real-Time Syst. Symp.*, 2012, pp. 63–72.
[31] J. Li, J.-J. Chen, K. Agrawal, C. Lu, C. Gill, and A. Saifullah, "Analysis of federated and global scheduling for parallel real-time tasks," in *Proc. Euromicro Conf. Real-Time Syst.*, 2014, pp. 85–96.
[32] S. Baruah, "Improved multiprocessor global schedulability analysis of sporadic DAG task systems," in *Proc. Euromicro Conf. Real-Time Syst.*, 2014, pp. 97–105.
[33] N. Fisher, J. Goossens, and S. Baruah, "Optimal online multiprocessor scheduling of sporadic real-time tasks is impossible," *Real-Time Syst.*, vol. 45, pp. 26–71, 2010.
[34] D. Ferry, J. Li, M. Mahadevan, C. Gill, C. Lu, and K. Agrawal, "A real-time scheduling service for parallel tasks," in *Proc. IEEE Real-Time Technol. Appl. Symp.*, 2013, pp. 261–272.
[35] A. Srinivasan and J. Anderson, "Fair scheduling of dynamic task systems on multiprocessors," *J. Syst. Softw.*, vol. 77, no. 1, pp. 67–80, 2005.
[36] H. Cho, B. Ravindran, and E. D. Jensen, "An optimal real-time scheduling algorithm for multiprocessors," in *Proc. IEEE Real-Time Syst. Symp.*, 2006, pp. 101–110.
[37] M. Bertogna, M. Cirinei, and G. Lipari, "Schedulability analysis of global scheduling algorithms on multiprocessor platforms," *IEEE Trans. Parallel Distrib. Syst.*, vol. 20, no. 4, pp. 553–566, Apr. 2009.
[38] J. Lee, A. Easwaran, and I. Shin, "LLF Schedulability Analysis on Multiprocessor Platforms," in *Proc. IEEE Real-Time Syst. Symp.*, 2010.
[39] H. S. Chwa, H. Back, S. Chen, J. Lee, A. Easwaran, I. Shin, and I. Lee, "Extending task-level to job-level fixed priority assignment and schedulability analysis using pseudo-deadlines," in *Proc. IEEE Real-Time Syst. Symp.*, 2012, pp. 51–62.
[40] T. Baker, "An analysis of EDF schedulability on a multiprocessor," *IEEE Trans. Parallel Distrib. Syst.*, vol. 16, no. 8, pp. 760–768, Aug. 2005.
[41] D. Cordeiro, G. Mouni, S. Perarnau, D. Trystram, J.-M. Vincent, and F. Wagner, "Random graph generation for scheduling simulations," in *Proc. 3rd Int. ICST Conf. Simulation Tools Techn.*, 2010, Art. no. 60.

Hoon Sung Chwa received the BS, MS, and PhD degrees from Korea Advanced Institute of Science and Technology, all in computer science, in 2009, 2011, and 2016, respectively. He is a postdoctoral researcher in the School of Computing, Korea Advanced Institute of Science and Technology, South Korea. His research interests include system design and analysis with timing guarantees and resource management in real-time embedded systems and cyber-physical systems. He won two best paper awards from the 33rd IEEE Real-Time Systems Symposium (RTSS) in 2012 and from the IEEE International Conference on Cyber-Physical Systems, Networks, and Applications (CPSNA) in 2014. He is a member of IEEE.

Jinkyu Lee received the BS, MS, and PhD degrees in computer science from Korea Advanced Institute of Science and Technology, South Korea, in 2004, 2006, and 2011, respectively. He is an assistant professor in the Department of Computer Science and Engineering, Sungkyunkwan University, South Korea, where he joined in 2014. He has been a research fellow/visiting scholar in the Department of Electrical Engineering and Computer Science, University of Michigan until 2014. His research interests include system design and analysis with timing guarantees, QoS support, and resource management in real-time embedded systems and cyber-physical systems. He won the best student paper award from the 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), in 2011, and the Best Paper Award from the 33rd IEEE Real-Time Systems Symposium (RTSS), in 2012. He is a member of the IEEE.

Jiyeon Lee received the BS degree in computer science from Dankook University, South Korea, in 2012, and the MS degree in computer science from Korea Advanced Institute of Science and Technology (KAIST), South Korea, in 2014. She is currently working toward the PhD degree in computer science at KAIST. Her research interests include system design and analysis with timing guarantees and resource management in real-time embedded systems and cyber-physical systems. She is a student member of the IEEE.

Kiew-My Phan received the BS degree in computer science from Korea Advanced Institute of Science and Technology, South Korea, in 2013. Her research interests include system design and analysis with timing guarantees and resource management in real-time embedded systems and cyber-physical systems. She is a student member of the IEEE.

Arvind Easwaran received the BE degree from the University of Mumbai, India, and the MSc and PhD degrees from the University of Pennsylvania, Pennsylvania, all in computer science & engineering. He is an assistant professor in the School of Computer Science and Engineering, Nanyang Technological University (NTU), Singapore. Prior to joining NTU in 2013, he has been an Invited research scientist at the Polytechnic Institute of Porto, Portugal, and an R&D scientist at Honeywell Aerospace, Phoenix, Arizona. His research interests include cyber-physical systems, real-time systems, and formal methods. He has published in several leading conferences and journals in these areas, some of which are highly cited and have won awards.

Insik Shin received the BS degree from Korea University, the MS degree from Stanford University, and the PhD degree from the University of Pennsylvania, all in computer science, in 1994, 1998, and 2006, respectively. He is currently an associate professor in the Department of Computer Science, KAIST, South Korea, where he joined in 2008. He has been a postdoctoral research fellow with Malardalen University, Sweden, and a visiting scholar with the University of Illinois, Urbana-Champaign until 2008. His research interests include cyber-physical systems and real-time embedded systems. He is currently a member of the editorial board of the *Journal of Computing Science and Engineering*. He has been cochair of various workshops including satellite workshops of RTSS, CPSWeek, and RTCSA and has served various program committees in real-time embedded systems, including RTSS, RTAS, ECRTS, and EMSOFT. He received best paper awards, including Best Paper Awards from RTSS, in 2003 and 2012, Best Student Paper Award from RTAS, in 2011, and Best Paper runner-ups at ECRTS and RTSS, in 2008. He is a member of the IEEE.