The Journal of Systems and Software 113 (2016) 246–256

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Thread-level priority assignment in global multiprocessor scheduling for DAG tasks

Jiyeon Lee [a], Hoon Sung Chwa [a], Jinkyu Lee [b], Insik Shin [a,*]

[a] School of Computing, KAIST, Daejeon, South Korea
[b] Department of Computer Science and Engineering, Sungkyunkwan University, Suwon, South Korea

ARTICLE INFO

Article history:
Received 6 July 2015
Revised 24 November 2015
Accepted 6 December 2015
Available online 15 December 2015

Keywords:
Real-time systems
Intra-parallel task scheduling
Optimal thread-level priority assignment

ABSTRACT

The advent of multi- and many-core processors offers enormous performance potential for parallel tasks that exhibit sufficient intra-task thread-level parallelism. With a growth of novel parallel programming models (e.g., OpenMP, MapReduce), scheduling parallel tasks in the real-time context has received an increasing attention in the recent past. While most studies focused on schedulability analysis under some well-known scheduling algorithms designed for sequential tasks, little work has been introduced to design new scheduling policies that accommodate the features of parallel tasks, such as their multi-threaded structure. Motivated by this, we refine real-time scheduling algorithm categories according to the basic unit of scheduling and propose a new priority assignment method for global task-wide thread-level fixed-priority scheduling of parallel task systems. Our evaluation results show that a finer-grained, thread-level fixed-priority assignment, when properly assigned, significantly improves schedulability, compared to a coarser-grained, task-level assignment.

1. Introduction

The trend for multicore processors is towards an increasing number of on-chip cores. Today, CPUs with 8–10 state-of-the-art cores or 10s of smaller cores AMD are commonplace. In the near future, many-core processors with 100s of cores will be possible Comming soon. A shift from unicore to multicore processors allows *inter-task parallelism*, where several applications (tasks) can execute simultaneously on multiple cores. However, in order to fully exploit multicore processing potential, it entails support for *intra-task parallelism*, where a single task consists of multiple threads that are able to execute concurrently on multiple cores.

Two fundamental problems in real-time scheduling are (1) algorithm design to derive priorities so as to satisfy all timing constrains (i.e., deadlines) and (2) schedulability analysis to provide guarantees of deadline satisfaction. Over decades, those two fundamental problems have been substantially studied for multiprocessor scheduling (Davis and Burns, 2011), generally with a focus on the inter-task parallelism of single-threaded (sequential) tasks. Recently, a growing number of studies have been introduced for supporting multi-threaded (parallel) tasks (Bonifaci et al., 2013; Li et al., 2014; Baruah et al., 2012; Andersson and de Niz, 2012; Li et al., 2013; Chwa et al., 2013; Baruah, 2014; Saifullah et al., 2011; Nelissen et al., 2012a; Lakshmanan et al., 2010; Liu and Anderson, 2010; 2012; Ferry et al., 2013; Axer et al., 2013; Qi Wang, 2014; Li et al., 2015; Kwon et al., 2015; Melani et al., 2015; Sanjoy Baruah, 2015; Shen Li, 2015). Schedulability analysis has been the main subject of much work on thread-level parallelism (Bonifaci et al., 2013; Li et al., 2014; Baruah et al., 2012; Andersson and de Niz, 2012; Li et al., 2013; Chwa et al., 2013; Baruah, 2014; Saifullah et al., 2011; Nelissen et al., 2012a; Lakshmanan et al., 2010; Liu and Anderson, 2010; 2012; Ferry et al., 2013; Axer et al., 2013; Li et al., 2015) for some traditionally well-known scheduling policies, i.e., EDF (Earliest Deadline First) (Liu and Layland, 1973) and DM (Deadline Monotonic) (Leung and Whitehead, 1982). However, a relatively much less effort has been made to understand how to design good scheduling algorithms for parallel tasks.

In a sequential task, a task is a sequence of invocations, or *jobs*, and the task invocation is the unit of scheduling. In general, priority-based real-time scheduling algorithms can fall into three categories according to when priorities change (Davis and Burns, 2011) : *task-wide fixed-priority* where a task has a single static priority over all of its invocations (e.g., DM), *job-wide fixed-priority* where a job has a single fixed priority (e.g., EDF), and *dynamic-priority* where a single job may have different priorities at different times (e.g., LLF Least Laxity First Dertouzos and Mok, 1989).

A parallel task consists of multiple threads, and the invocation of a thread is then the unit of scheduling. This brings a new dimension to the scheduling categories. With a finer granularity of scheduling

* Corresponding author. Tel.: +82 42 350 3524.
E-mail address: insik.shin@cs.kaist.ac.kr, insik.shin@gmail.com (I. Shin).

http://dx.doi.org/10.1016/j.jss.2015.12.004

from task to thread, we can further subdivide each scheduling category into two sub-categories, *task-level* and *thread-level*, according to the unit of priority assignment. To this end, we can refine the scheduling categories to a finer-grained thread level as follows:

- *Task-wide thread-level fixed-priority:* a single thread has a static priority across all of its invocations.
- *Job-wide thread-level fixed-priority:* a single thread has a static priority over one invocation.
- *Thread-level dynamic-priority:* a single thread can have different priorities at different times within one invocation.

In this paper, we aim to explore the possibility of performance enhancement in real-time scheduling by fully exploiting both inter-task and intra-task parallelisms. We hypothesize that a major factor in fully capitalizing on multicore processing potential is priority assignment. The key intuition behind our work is that finding an appropriate priority ordering is as important as using an efficient schedulability test, and that a finer-grained priority ordering at the thread level is more effective than a coarser-grained, task-level one. To this end, in this paper, we focus on priority assignment policies for global[1] task-wide thread-level fixed-priority pre-emptive scheduling.

1.1. Related work

In the recent past, supporting intra-task thread-level parallelism in the context of real-time scheduling has received increasing attention in the recent past (Bonifaci et al., 2013; Li et al., 2014; Baruah et al., 2012; Andersson and de Niz, 2012; Li et al., 2013; Chwa et al., 2013; Baruah, 2014; Saifullah et al., 2011; Nelissen et al., 2012a; Lakshmanan et al., 2010; Liu and Anderson, 2010; 2012; Ferry et al., 2013; Axer et al., 2013; Qi Wang, 2014; Li et al., 2015; Kwon et al., 2015; Melani et al., 2015; Sanjoy Baruah, 2015; Shen Li, 2015). The work in Liu and Anderson (2010); 2012) considers soft real-time scheduling focusing on bounding tardiness upon deadline miss, while hard real-time systems aim at ensuring all deadlines are met. In this paper, we consider hard real-time scheduling.

Fork-join task model. The fork-join task model is one of the popular parallel task models (Lea, 2000), OpenMP, where a 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, Lakshmanan et al. (2010) introduced a task decomposition method that transforms each synchronous parallel task into a set of independent sequential tasks, which can be then scheduled with traditional multiprocessor scheduling techniques. Lakshmanan et al. (2010) presented a resource augmentation bound[2] of 3.42 for partitioned thread-level DM scheduling. Lately, Qi Wang (2014) attempted to implement a system, called *FJOS*, that supports to fork-join intra-task parallelism in a hard real-time environment. They proposed the overhead-aware assignment algorithm based on the analysis presented in Axer et al. (2013).

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. Saifullah et al. (2011) presented decomposition method for synchronous parallel tasks and proved a resource augmentation bound of 4 for global thread-level EDF scheduling and 5 for partitioned thread-level DM scheduling. Building upon this work, Ferry et al. (2013) presented a prototype scheduling service for their *RT-OpenMP* concurrency platform. Nelissen et al. (2012a) also introduced another decomposition method and showed a resource augmentation bound of 2 for a certain class of global scheduling algorithms, such as PD^2 (Srinivasan and Anderson, 2005), LLREF (Cho et al., 2006), DP-Wrap (Levin et al., 2010), or U-EDF (Nelissen et al., 2012b). Some studies (Andersson and de Niz, 2012; Chwa et al., 2013; Axer et al., 2013) developed direct schedulability analysis without task decomposition for synchronous parallel tasks. In this context, Andersson and de Niz (2012) showed a resource augmentation bound of $2 - 1/m$ for global EDF scheduling. Chwa et al. (2013) introduced an interference-based analysis for global task-level EDF scheduling, and Axer et al. (2013) presented a response-time analysis (RTA) for partitioned thread-level fixed-priority scheduling.

DAG task model. Refining the granularity of synchronization from segment-level to thread-level, a DAG (Directed Acyclic Graph) task model is considered, where a node represents a thread and an edge specifies a precedence dependency between nodes. Baruah et al. (2012) showed a resource augmentation bound of 2 for a single DAG task with arbitrary deadlines under global task-level EDF scheduling. For a set of DAG tasks, a resource augmentation bound of $2 - 1/m$ was presented for global task-level EDF scheduling in Bonifaci et al. (2013), Li et al. (2013) and Baruah (2014). Bonifaci et al. (2013) also derived a $3 - 1/m$ resource augmentation bound for global task-level DM scheduling. In addition to those resource augmentation bounds, Li et al. (2013) introduced capacity augmentation bounds that can work as independent schedulability tests, and showed a $4 - 2/m$ capacity augmentation bound for global task-level EDF. In a further study, Li et al. (2015) developed a prototype platform, called *PGEDF*, by combining GNU-OpenMP runtime system and the *LITMUS^RT* system for DAG tasks, and evaluated the schedulability test presented in Li et al. (2013). Later, Li et al. (2014) improved the capacity augmentation bound up to 2.6181 and 3.7321 for global task-level EDF and RM, respectively. Li et al. (2014) also proposed a new scheduling policy, called federated scheduling, and derived a resource augmentation bound of 2 for the proposed approach.

Nowadays, some studies have been introduced for an extended DAG task model, which considers more practical excution environments. Kwon et al. (2015) relaxed the assumption of a pre-defined number of threads in the DAG task model, and exploited multiple parallel options (i.e., runtime selectable numbers of threads) to improve schedulability. The work in Melani et al. (2015) and Sanjoy Baruah (2015) also proposed the extended DAG task model which characterizes the excution flow of conditional branch, and Shen Li (2015) proposed a real-time scheduling of MapReduce workflows based on a hierarchical scheduling scheme.

In summary, much work in the literature introduced and improved schedulability analysis for different parallel task models under different multiprocessor scheduling approaches and algorithms. Table 1 summarizes the global scheduling algorithms that have been considered in the literature, to the best of the author's knowledge. We have two interesting observations from the table. One is that most existing studies considered well-known deadline-based scheduling algorithms (EDF, DM) originally designed for sequential tasks, with a large portion on task-level priority scheduling. Capturing the urgency

[1] Multiprocessor scheduling approaches can broadly fall into two classes: *global* and *partitioned*. Partitioned approaches allocate each task (or thread) to a single processor statically, transforming the multiprocessor scheduling into uniprocessor scheduling with task (or thread) allocation. In contrast, global approaches allow tasks (or threads) to migrate dynamically across multiple processors.

[2] Recently, Li et al. (2013) distinguished resource and capacity augmentation bounds as follows. The resource augmentation bound r of a scheduler $\mathcal{S}$ has the property that if a task set is feasible on m unit-speed processors, then the task set is schedulable under $\mathcal{S}$ on m processors of speed r. For a scheduler $\mathcal{S}$ 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 $\mathcal{S}$ 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 multiprocessor 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 Li et al., 2013 more details).

Table 1
Global scheduling algorithms for parallel tasks.

		Task-wide fixed-priority	Job-wide fixed-priority	Dynamic-priority
Global	Task-level	DM (Bonifaci et al., 2013) / RM (Li et al., 2014)	EDF (Bonifaci et al., 2013; Li et al., 2014; Baruah et al., 2012; Andersson and de Niz, 2012; Li et al., 2013; Chwa et al., 2013; Baruah, 2014; Li et al., 2015)	
	Thread-level	(*OTPA / PADA [this paper]*)	EDF (Saifullah et al., 2011)	PD^2 / U-EDF / LLREF / DP-Wrap (Nelissen et al., 2012a)

of real-time workloads, deadlines are a good real-time scheduling parameter, in particular, for sequential tasks on a single processor (Liu and Layland, 1973). However, deadlines are no longer as effective for parallel tasks on multiprocessors, since deadlines are inappropriate to represent the characteristics of parallel tasks, including the degree of intra-task parallelism (i.e., the number of threads that can run in parallel on multiprocessors) or precedence dependency between threads. The other observation from Table 1 is that little work has explored the task-wide thread-level fixed-priority scheduling category. These motivate our work to develop a new task-wide thread-level fixed-priority assignment method that incorporates the characteristics of DAG tasks.

1.2. *Our approach*

This work is motivated by an attempt to see how good task-wide thread-level fixed-priority assignment, beyond task-level, can be for global multiprocessor scheduling of parallel tasks. To this end, this paper seeks to explore the possibility of using the OPA (Optimal Priority Assignment) algorithm (Audsley, 1991; 2001; Davis and Burns, 2009), which is proven to be optimal in task-wide fixed-priority assignment for independent tasks with respect to some given schedulability analysis. The application of OPA to thread-level priority assignment raises several issues, including how to deal with thread-level dependency and how to develop an efficient thread-level OPA-compatible analysis.

A parallel task typically consists of multiple threads that come with their own precedence dependency. With such a thread-level dependency in the parallel task case, it is thereby non-trivial to make use of OPA for thread-level priority assignment, since OPA is designed for independent tasks. Task decomposition is one of the widely used approaches to deal with the thread-level precedence dependency (Saifullah et al., 2011; Nelissen et al., 2012a; Lakshmanan et al., 2010; Ferry et al., 2013; Axer et al., 2013). Through task decomposition, each individual thread is assigned its own offset and deadline in a way that its execution is separated from those of its predecessors. This allows all threads to be considered as independent as long as their thread-level deadlines can be met. In this paper, we employ such a task decomposition approach to develop an OPA-based thread-level priority assignment method.

Contributions. The main results and contributions of this paper can be summarized as follows. First, we introduce an efficient thread-level interference-based analysis that is aware of the multi-threaded structure of parallel tasks (in Section 3). We also show that the proposed analysis is OPA-compatible (in Section 4). This allows OPA, when using the proposed analysis, to accommodate the characteristics of parallel tasks via its underlying analysis in priority assignment.

Second, we show that the OPA algorithm, originally designed for independent sequential tasks, is applicable to parallel tasks when thread-level precedence dependencies are resolved properly through task decomposition. That is, the algorithm holds optimality in thread-level priority assignment when threads are independent with their own offsets and deadlines with respect to its underlying analysis (in Section 4). With the use of OPA, this study separates thread priority assignment from thread dependency resolution. While most previous decomposition-based studies (Saifullah et al., 2011; Lakshmanan et al., 2010; Ferry et al., 2013) share an approach that resolves between-thread dependencies by determining the relative deadlines of individual threads properly and makes use of thread deadlines for priority ordering, this study decouples thread priorities from deadlines.

Third, we propose a new OPA-based priority assignment method that adjusts thread offsets and deadlines, called PADA (Priority Assignment with Deadline Adjustment), taking into consideration the properties of OPA and its underlying analysis (in Section 5). In the previous studies on fixed-priority scheduling (Saifullah et al., 2011; Lakshmanan et al., 2010; Ferry et al., 2013), thread deadlines are determined, from an individual task perspective, only to resolve intra-task thread dependency. On the other hand, in this study, thread deadlines are adjusted, from the system-wide perspective, to accommodate interference between tasks for schedulability improvement.

Finally, our evaluation results show that the proposed thread-level priority assignment is significantly more effective, in terms of the number of task sets deemed schedulable, than task-level priority assignment in global task-wide fixed-priority scheduling (in Section 6). The results also show that incorporating the features of parallel tasks into priority assignment significantly improves schedulability, compared to traditional deadline-based priority ordering, and that the proposed approach outperforms the existing approaches.

2. System model

2.1. *DAG task*

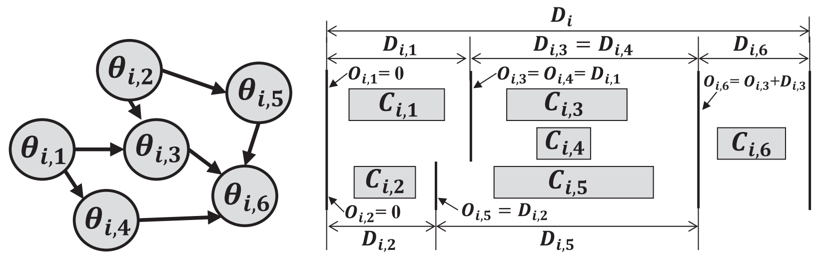
We consider a set of DAG (Directed Acyclic Graph) tasks τ. A DAG task $\tau_i \in \tau$ is represented by a directed acyclic graph as shown in Fig. 1(a). A vertex $v_{i,p}$ in τ_i represents a single thread $\theta_{i,p}$, and a directed edge from $v_{i,p}$ to $v_{i,q}$ represents the precedence dependency such that $\theta_{i,q}$ cannot start execution unless $\theta_{i,p}$ has finished execution. A thread $\theta_{i,p}$ becomes ready for execution as soon as all of its predecessors have completed their execution.

A sporadic DAG task τ_i invokes a series of jobs with the minimum separation of T_i, and each job should finish its execution within D_i (the relative deadline). We denote as J_i^h the h-th job of τ_i.

2.2. *Task decomposition*

A DAG task can be decomposed into a set of independent sequential sub-tasks, capturing the precedence relation between threads by separating the execution windows of the threads. That is, each thread of the DAG task is assigned its own relative offset and deadline in a way that the release time of the thread is no earlier than the latest deadline among the ones of all the predecessors.

We denote τ^{decom} a set of all threads generated from τ through task decomposition, and the number of threads in a decomposed task set τ^{decom} is denoted as n. For a decomposed task τ_i, we define a *primary thread* of the task (denoted by $\theta_{i,1}$), as one of the threads in τ_i that have no predecessors. Then, each thread $\theta_{i,p}$ in τ_i is specified by $(T_{i,p}, C_{i,p}, D_{i,p}, O_{i,p})$, where $T_{i,p}$ is the minimum separation (which equals to T_i), $C_{i,p}$ is the worst-case execution time (which is inherited by the original thread), $D_{i,p}$ is the relative deadline, and $O_{i,p}$ is the relative offset (from $O_{i,1} = 0$). Note that $D_{i,p}$ and $O_{i,p}$ are determined by decomposition methods (more details in Section 5). Fig. 1(b)



(a) A DAG task τ_i (b) A decomposed task τ_i

Fig. 1. A DAG task and its decomposed task.

illustrates a decomposed task, which corresponds to the DAG task in Fig. 1(a).

For a job J_i^h, the primary thread $\theta_{i,1}$ is released at $r_{i,1}^h$, and has absolute deadline $d_{i,1}^h = r_{i,1}^h + D_{i,1}$. Then, the next thread $\theta_{i,p}$ which has dependency with $\theta_{i,1}$ is released at $r_{i,p}^h = r_{i,1}^h + O_{i,p}$, and has deadline $d_{i,p}^h = r_{i,p}^h + D_{i,p}$. The execution window of $\theta_{i,p}$ is then defined as an interval $(r_{i,p}^h, d_{i,p}^h]$.

2.3. Platform and scheduling algorithm

This paper focuses on a multi-core platform, consisting of m identical processors. This paper also considers global task-wide thread-level fixed-priority scheduling, in which each single thread $\theta_{i,p}$ is able to migrate dynamically across processors and assigns a static priority $P_{i,p}$ across all of its invocations. We denote as $hp(\theta_{i,p})$ a set of threads whose priorities are strictly higher than $P_{i,p}$.

3. Schedulability analysis

Once a DAG task is decomposed into individual threads, each thread has its own relative offset and deadline without having to consider precedence dependency any more. This allows to treat each thread as an individual independent sequential task, and it is possible to analyze the schedulability of each thread in a sufficient manner using the existing task-level schedulability analyzes. However, this brings a substantial degree of pessimism since the existing task-level analysis techniques were originally designed for sequential tasks and are thereby oblivious of the intra-task parallelism.

Motivated by this, the goal of this section is to develop a schedulability condition that helps to analyze the schedulability of a thread more efficiently, incorporating the internal thread structures of parallel tasks into analysis. To this end, we consider interference-based analysis as a basis, since interference-based analysis is OPA-compatible (Davis and Burns, 2009).

3.1. Interference-based schedulability analysis

Extending the traditional notion of task-level interference, thread-level interference can be defined as follows.

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

With the above definitions, the relation between $I_{k,q}(a, b)$ and $I_{(i,p)\to(k,q)}(a, b)$ serves as an important basis for deriving a schedulability analysis. Since a thread cannot be scheduled only when m other threads execute, a relation between $I_{k,q}(a, b)$ and $I_{(i,p)\to(k,q)}(a, b)$ can be derived similarly as in Lemma 3 for sequential tasks in Bertogna et al. (2005) as follows:

$$I_{k,q}(a,b) = \frac{\sum_{(i,p)\neq(k,q)} I_{(i,p)\to(k,q)}(a,b)}{m}. \tag{1}$$

Let $J_{k,q}^*$ denote the job that receives the maximum total interference among jobs on $\theta_{k,q}$, and then the worst-case total interference on $\theta_{k,q}$ in the job (denoted by $I_{k,q}^*$) can be expressed as

$$I_{k,q}^* \triangleq \max_h (I_{k,q}(r_{k,q}^h, d_{k,q}^h)) = I_{k,q}(r_{k,q}^*, d_{k,q}^*). \tag{2}$$

Using the above definitions, the studies (Bertogna et al., 2005; 2009) developed the schedulability condition of global multiprocessor scheduling algorithms for sequential tasks, which can be extended to parallel tasks as follows:

Lemma 1 (From Bertogna et al., 2005; 2009). *A set τ^{decom} is schedulable under any work-conserving algorithm on a multiprocessor composed by m identical processors if and only if the following condition holds for every thread $\theta_{k,q}$:*

$$\sum_{\theta_{i,p}\in\tau^{decom}\setminus\{\theta_{k,q}\}} \min(I_{(i,p)\to(k,q)}^*, D_{k,q} - C_{k,q} + 1) < m\cdot(D_{k,q} - C_{k,q} + 1). \tag{3}$$

Then, it is straight-forward that the schedulability of the decomposed task set guarantees that of the original task set, as recorded in the following lemma.

Lemma 2 (From Saifullah et al., 2011). *If τ^{decom} is schedulable, then τ is also schedulable.*

Since it is generally intractable to compute exact interference under a given scheduling algorithm, existing approaches for the sequential task model (Bertogna et al., 2005; 2009; Baker, 2003; Guan et al., 2009; Lee et al., 2010; 2011; Back et al., 2012; Chwa et al., 2012) have derived upper-bounds on the interference under target algorithms, resulting in sufficient schedulability analyzes. We also need to calculate upper-bounds on the interference for decomposed tasks. Since the structure of a decomposed task is different from that of a sequential task, the execution and release patterns that maximize interference should be different, which will be addressed in the next section.

3.2. Workload-based schedulability analysis with offset

As we mentioned, this paper focuses on task-wide thread-level fixed-priority scheduling with task decomposition. Therefore, we need to check whether each thread finishes its execution within the deadline as described in Lemma 1, and the remaining step is to calculate interference of all other threads on a target thread $\theta_{k,q}$, i.e., the LHS of Eq. (3).

One simple approach is to upper-bound thread-to-thread interference (i.e., $I_{(i,p)\to(k,q)}^*$), by calculating the maximum amount of execution of $\theta_{i,p}$ in the execution window of $\theta_{k,q}$, called *workload*. If we take this approach, we can re-use existing task-level schedulability

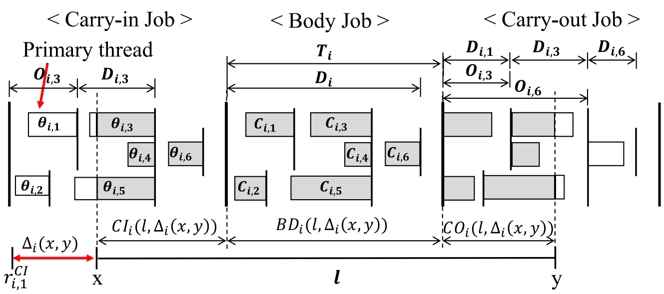


Fig. 2. The maximum workload of all threads in τ_i in $(x, y]$ with given $\Delta_i(x, y)$.

tests for the sequential model. However, the approach entails a significant pessimism because it does not account for the precedence relation among threads in the same task; in other words, if we consider the precedence relation, the situations where the amount of execution of a thread of a task is maximized and that of another thread of the same task is maximized may not happen at the same time.

Therefore, we seek to derive an upper-bound on task-to-thread interference, i.e., the interference of a task τ_i on $\theta_{k,q}$, denoted by $\sum_{\forall\theta_{i,p}\in\tau_i}\min(I^*_{(i,p)\to(k,q)}, D_{k,q}-C_{k,q}+1)$. To achieve this, we first calculate the amount of execution of τ_i in the execution window of $\theta_{k,q}$ when the alignment for the job releases of τ_i is given. Then, we identify the alignment that maximizes the amount of execution of τ_i.

We consider two cases to calculate the maximum workload: when $i \neq k$ and $i = k$. This is because, $i = k$ implies that both interfered and interfering threads belong to the same task, meaning that the alignment for τ_i's job releases is automatically given.

3.2.1. The maximum workload when $i \neq k$

To simplify the presentation, we use the following terms. A job of a task is said to be a *carry-in* job of an interval $(x, y]$ if it is released before x but has a deadline within $(x, y]$, a *body* job if its release time and deadline are both within $(x, y]$, and a *carry-out* job if it is released within $(x, y]$ but a deadline after y. Note that a job is released before x and has a deadline after y is regarded as a carry-in job.

Let us consider the situation in which jobs of τ_i are periodically released. We define $\Delta_i(x, y)$ that is the difference between the release time of the primary thread of the carry-in job in $(x, y]$ and x where is the start point of the execution window of $\theta_{k,q}$ as shown in Fig. 2. For a given $\Delta_i(x, y)$, the interval $(x, y]$ of length l can be partitioned into carry-in, body, and carry-out intervals, and the length of the intervals are denoted as $CI_i(l, \Delta_i(x, y))$, $BD_i(l, \Delta_i(x, y))$, and $CO_i(l, \Delta_i(x, y))$, respectively, and described as

$$CI_i(l, \Delta_i(x,y)) = \min(T_i - \Delta_i(x,y), l), \tag{4}$$

$$BD_i(l, \Delta_i(x,y)) = \left\lfloor \frac{l - CI_i(l, \Delta_i(x,y))}{T_i} \right\rfloor \cdot T_i, \tag{5}$$

$$CO_i(l, \Delta_i(x,y)) = l - CI_i(l, \Delta_i(x,y)) - BD_i(l, \Delta_i(x,y)). \tag{6}$$

Then, with the given $\Delta_i(x, y)$, the workload contribution of each thread in $(x, y]$ (shown in Fig. 2) is calculated as

$$W_{i,p}(l, \Delta_i(x,y)) = W^{CI}_{i,p} + W^{BD}_{i,p} + W^{CO}_{i,p}, \tag{7}$$

where

$$W^{CI}_{i,p} = \Big[\min(O_{i,p} + D_{i,p}, \Delta_i(x,y) + l) - \max(\Delta_i(x,y), O_{i,p})\Big]_0^{C_{i,p}},$$

$$W^{BD}_{i,p} = \left\lfloor \frac{l - CI_i(l, \Delta_i(x,y))}{T_{i,p}} \right\rfloor \cdot C_{i,p},$$

$$W^{CO}_{i,p} = \Big[CO_i(l, \Delta_i(x,y)) - O_{i,p}\Big]_0^{C_{i,p}}.$$

Note that $[X]_a^b$ means $\min(\max(X, a), b)$. Now we will prove that $W^{CI}_{i,p}$, $W^{BD}_{i,p}$, and $W^{CO}_{i,p}$ are respectively the upper-bounds on the amount of execution of a carry-in job, body jobs, and a carry-out job of $\theta_{i,p}$ in an interval $(x, y]$ of length l with given $\Delta_i(x, y)$.

For $W^{CI}_{i,p}$, we first find the interval in which the execution window of the carry-in job of $\theta_{i,p}$ overlaps with $(x, y]$; we denote the interval as $(a, b]$. Without loss of generality, we set $r^{CI}_{i,1}$ to 0. Then, the carry-in job of $\theta_{i,p}$ is released at $O_{i,p}$. If $\Delta_i(x, y) < O_{i,p}$, the time instant a is $O_{i,p}$; otherwise, a is $\Delta_i(x, y)$, as shown in Fig. 2. Also, the deadline of the carry-in job of $\theta_{i,p}$ is $O_{i,p} + D_{i,p}$. If $\Delta_i(x, y) + l > O_{i,p} + D_{i,p}$, the time instant b is $O_{i,p} + D_{i,p}$; otherwise, b is $\Delta_i(x, y) + l$, meaning that only the carry-in job (without body and carry-out jobs) overlaps with $(x, y]$. In summary, a equals to $\max(\Delta_i(x, y), O_{i,p})$, and b equals to $\min(O_{i,p} + D_{i,p}, \Delta_i(x, y) + l)$. In $(a, b]$, the carry-in job cannot execute more than its execution time $C_{i,p}$ and less than 0; therefore, we derive $W^{CI}_{i,p}$ in Eq. (7).

When it comes to $W^{BD}_{i,p}$, the number of body jobs of $\theta_{i,p}$ is simply calculated by $\left\lfloor \frac{l - CI_i(l,\Delta_i(x,y))}{T_{i,p}} \right\rfloor$. Therefore, $W^{BD}_{i,p}$ equals to the number multiplied by the execution time $C_{i,p}$.

The derivation of $W^{CO}_{i,p}$ is similar to that of $W^{CI}_{i,p}$. We find the interval in which the execution window of the carry-out job of $\theta_{i,p}$ overlaps with $(x, y]$; we also denote the interval as $(a, b]$. Without loss of generality, we set $r^{CO}_{i,1}$ to 0, where $r^{CO}_{i,1}$ is the release time of the carry-out job of $\theta_{i,p}$. Then, a and b are $O_{i,p}$ and $CO_i(l, \Delta_i(x, y))$, respectively as shown in Fig. 2. Since the carry-out job cannot execute more than its execution time $C_{i,p}$ and less than 0, we derive $W^{CO}_{i,p}$ in Eq. (7).

For the situation where τ_i invokes its jobs sporadically, we can easily check that the amount of execution of $\theta_{i,p}$ in $(x, y]$ with $\Delta_i(x, y)$ is upper-bounded by $W_{i,p}(l, \Delta_i(x, y))$.

Considering all possible values of $\Delta_i(x, y)$ of task τ_i, the sum of workload of all threads that have a higher priority than thread $\theta_{k,q}$ is an upper bound of the maximum interference of τ_i on thread $\theta_{k,q}$. Thus,

$$W_i(D_{k,q}) = \max_{0 \le \Delta_i(x,y) < T_i} \sum_{\forall\theta_{i,p}\in hp(\theta_{k,q})} \min(W_{i,p}(D_{k,q}, \Delta_i(x,y)), D_{k,q} - C_{k,q} + 1). \tag{8}$$

3.2.2. The maximum workload when $i = k$

In the case of that τ_k is interfered by the same task, the alignment for τ_k's job releases is automatically determined (i.e, Interval $(x, y]$ is set to the execution window of thread $\theta_{k,q}$, and $\Delta_k(r_{k,q}, d_{k,q})$ is fixed with $O_{k,q}$). To calculate the maximum workload when $i = k$, we only need to consider the threads whose execution windows are overlapped with thread $\theta_{k,q}$. The workload contribution of those threads can be similarly calculated using Eq. (7). Thus, the maximum workload of all threads of τ_k that have a higher priority than thread $\theta_{k,q}$ is calculated as

$$W_k(D_{k,q}) = \sum_{\forall\theta_{k,p}\in hp(\theta_{k,q})} \min(W_{k,p}(D_{k,q}, O_{k,q}), D_{k,q} - C_{k,q} + 1). \tag{9}$$

Based on the upper-bound on the interference calculated in Eqs. (8) and (9), we develop the following schedulability test for task-wide thread-level fixed-priority scheduling.

Theorem 1. *A set τ^{decom} is schedulable under task-wide thread-level fixed-priority scheduling on a multiprocessor composed by m identical processors if for every thread $\theta_{k,q}$, the following inequality holds:*

$$\sum_{\forall\tau_i\neq\tau_k} W_i(D_{k,q}) + W_k(D_{k,q}) < m \cdot (D_{k,q} - C_{k,q} + 1). \tag{10}$$

Proof. As we derived, $W_i(D_{k,q})$ (likewise $W_k(D_{k,q})$) is the maximum amount of higher-priority execution of τ_i with $i \neq k$ (likewise τ_k) than $P_{k,q}$ in the execution window of $\theta_{k,q}$. Since an execution A can interfere with another execution B only if the priority of A is higher than that of B under task-wide thread-level fixed-priority scheduling, the

LHS of Eq. (3) is upper-bounded by the LHS of Eq. (10). By Lemma 1, the theorem holds. □

4. Optimal thread-level priority assignment

This paper considers the *thread-level optimal priority assignment* problem that, given a decomposed set τ^{decom}, determines the priority $P_{i,p}$ of every thread $\theta_{i,p} \in \tau^{decom}$ such that the decomposed set is deemed schedulable according to the workload-based schedulability test given in Theorem 1. In this section, we show that the OPA algorithm for sequential tasks is applicable to parallel tasks with decomposition.

The OPA algorithm (Davis and Burns, 2009) aims at assigning a priority to each individual task through iterative priority assignment such that an entire task set is deemed schedulable by some given OPA-compatible schedulability test X under task-wide fixed-priority scheduling. A schedulability test is *OPA-compatible* if the following conditions are satisfied for any given task τ_i:

Condition 1: The schedulability of task τ_k is insensitive to relative ordering of its higher (and lower) priority tasks.

Condition 2: When the priority of τ_k is promoted (or demoted) by swapping the priorities of τ_k and τ_i, τ_k remains schedulable (or unschedulable) after the swap, if it was schedulable (or unschedulable) before the swap.

For thread-level extension of the priority assignment, we now present the Optimal Thread-level Priority Assignment (OTPA) algorithm, applying the OPA algorithm for sequential tasks to decomposed threads in parallel tasks. As described in Algorithm 1, our OTPA algorithm iteratively assigns priorities to the decomposed threads from the lowest one. In the k-th iteration step, the decomposed set τ^{decom} is divided into two disjoint subsets: $A(k)$ and $R(k)$, where

1. $A(k)$ denotes a subset of threads whose priorities have been assigned before the k-th step, and
2. $R(k)$ denotes a subset of remaining threads whose priorities must be assigned from the k-th step onwards.

The OTPA algorithm in Algorithm 1 yields a correct optimal priority assignment, because the schedulability test in Theorem 1 is *OTPA-compatible*, meaning that the test satisfies Conditions 1 and 2 for thread-level schedulability (i.e., substituting $\theta_{k,q}$ for τ_k in the conditions), as stated and proved in the following theorem.

Theorem 2. *The proposed schedulability test given in Theorem 1 is OTPA-compatible.*

Proof. We wish to show that both Conditions 1 and 2 hold for thread-level schedulability according to the proposed schedulability test.

In the LHS of Eq. (10), an upper bound on the interference of each task on thread $\theta_{k,q}$ is computed. The upper bound on the interference of a task is calculated from the sum of workload of all threads that have a higher priority than thread $\theta_{k,q}$. Computing workload of threads having a higher priority does not depend on their relative priority ordering. The other threads that have a lower priority than thread $\theta_{k,q}$ are excluded in calculation. Therefore, Condition 1 holds.

Algorithm 1 OTPA (Optimal Thread-level Priority Assignment).

$k \leftarrow 0, A(1) \leftarrow \emptyset, R(1) \leftarrow \tau^{decom}$
repeat
  $k \leftarrow k+1$
  if Assign-Priority($A(k)$, $R(k)$) = failure **then**
    return unschedulable
  end if
until $R(k)$ is empty
return schedulable

Algorithm 2 Assign-Priority($A(k)$, $R(k)$).

for each thread $\theta_{p,q} \in R(k)$ **do**
  if $\theta_{p,q}$ is schedulable with priority k assuming that all remaining threads in $R(k)$, except $\theta_{p,q}$, have higher priorities than k, according to the schedulability test in Theorem 1 **then**
    assign priority k to $\theta_{p,q}$ ($P_{p,q} \leftarrow k$)
    $R(k+1) \leftarrow R(k) \setminus \{\theta_{p,q}\}$
    $A(k+1) \leftarrow A(k) \cup \{\theta_{p,q}\}$
    return success
  end if
end for
return failure

For Condition 2, we focus on the case where the priority of $\theta_{k,q}$ is promoted by swapping the priorities of $\theta_{k,q}$ and $\theta_{i,p}$. Since the priority of $\theta_{k,q}$ is promoted, $hp(\theta_{k,q})$ becomes only smaller upon the swap. Therefore, $W_i(D_{k,q})$ and $W_k(D_{k,q})$ in Eqs. (8) and (9) get smaller after the swap, resulting in a decrease in the LHS of Eq. (10). This proves the case, and the other case (demoting the priority of τ_i) can be proved in a similar way. Hence, Condition 2 holds. □

During the k-th step, OTPA then invokes a function Assign-Priority($A(k)$, $R(k)$) described in Algorithm 2 to find a thread deemed schedulable according to Theorem 1 under the assumption that all unassigned threads in $R(k)$ have higher priorities.

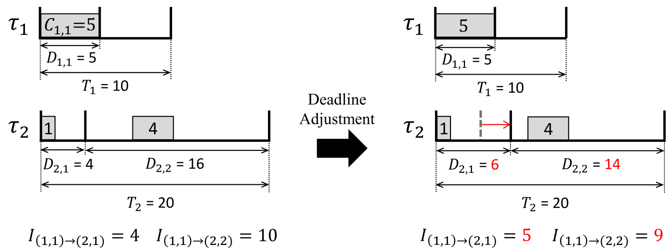
Since the schedulability test in Theorem 1 is OTPA-compatible, the OTPA algorithm has the following properties. First, the algorithm builds a solution incrementally without back-tracking. Once a thread is selected in an iteration step, the thread has no effect on priority assignment in the next iteration steps. This is because the thread is assigned a priority lower than all the unassigned tasks, imposing no interference on them. Second, if there exists only one thread deemed schedulable by our schedulability test at priority level k, OTPA must find it through searching all unassigned threads, which only requires linear time. Third, if there are multiple threads deemed schedulable by our schedulability test at priority level k, it does not matter which thread is selected by OTPA at priority level k. This is because all the other threads deemed schedulable but not selected at priority level k will remain deemed schedulable at the next higher priority level and will be eventually selected for priority assignment at a later level.

Computational complexity. We note that the number of threads in a decomposed task set τ^{decom} is denoted by n. By the above-mentioned three properties, the OTPA algorithm can find a priority assignment that all threads are schedulable according to the schedulability test if any exists, performing the schedulability test at most $\frac{n(n+1)}{2}$ times for n threads.

5. Priority assignment with deadline adjustment

In the previous section, thread-level priority assignment was considered under the assumption that the offset and deadline of each thread are given statically. Relaxing this assumption, in this section, we consider the problem of determining the offset, deadline, and priority of each individual thread such that the parallel task system is deemed schedulable according to the schedulability analysis in Theorem 1.

The existing decomposition approaches (Saifullah et al., 2011; Nelissen et al., 2012a) share in common the principle of density-based decomposition. The density of a segment is defined as a total sum of thread execution times in the segment over the relative deadline of the segment. Saifullah et al. (2011) categorize segments as either heavy or light according to the number of threads in a segment and determine the relative deadline of each segment such that each heavy segment has the same density. Nelissen et al. (2012a) decompose a parallel task such that the maximum density among all



(a) OTPA fails (b) OTPA succeeds

Fig. 3. An example of two tasks on a single processor illustrates situations, where (a) thread-level priority assignment fails when the deadline of each thread in τ_2 is assigned in proportion to thread execution time and (b) thread-level priority assignment becomes successful with the deadline of each thread in τ_2 properly adjusted.

segments in a parallel task is minimized. They assign the relative deadline of an individual segment in a different way according to upper bounds on the density of the segment. Then, those two approaches apply those density bounds to the existing density-based schedulability analysis and derive resource augmentation bounds. In the case of Nelissen et al. (2012a), deadline decomposition is optimal according to a sufficient schedulability test for scheduling algorithms such as PD[2] (Srinivasan and Anderson, 2005), LLREF (Cho et al., 2006), DP-Wrap (Levin et al., 2010), or U-EDF (Nelissen et al., 2012b).

Such a density-based decomposition is still a good principle in thread-level fixed-priority assignment, providing a good basis for high schedulability. However, it leaves room to improve further since the density-based principle does not go perfectly with our case, where the underlying analysis is not based on density. As an example, Fig. 3 shows two tasks with three threads on a single processor[3]. Task τ_2 has two threads with their execution times of 1 and 4. Fig. 3(a) shows the case, where the deadline D_2 of τ_2 is decomposed into $D_{2,1} = 4$ and $D_{2,2} = 16$ such that the resulting densities of two threads $\theta_{2,1}$ and $\theta_{2,2}$ are equal to each other, as in density-based decomposition. This way, OTPA is able to assign the lowest priority to $\theta_{2,2}$ but not able to proceed any more. Let us consider another case, as shown in Fig. 3(b), where $D_{2,1} = 6$ and $D_{2,2} = 14$. This situation can be considered as, from the initial deadline decomposition, thread $\theta_{2,2}$ donating its slack of 2 to thread $\theta_{2,1}$. Then, OTPA is able to assign priorities to $\theta_{2,1}$ and then $\theta_{1,1}$ successfully. This way, we can see that priority assignment can be improved through deadline adjustment, particularly, by passing the slack of one thread to another.

Motivated by this, we aim to develop an efficient method for Priority Assignment with Deadline Adjustment (PADA). In particular, we seek to incorporate the PADA method into the inherent characteristics of the underlying OTPA priority assignment and workload-based schedulability analysis. The basic idea behind PADA is as follows. It first seeks to assign priority through OTPA. When OTPA fails, it adjusts the offsets and deadlines of some threads such that there exists a thread that can be assigned a priority successfully after the deadline adjustment. If it finds such a deadline adjustment, it continues to use OTPA for priority assignment. Otherwise, it is considered as failure. See Algorithm 3.

We define the slack for thread $\theta_{k,q}$ as the minimum distance between the thread finishing time and its deadline, and denoted as $S_{k,q}$. Using our schedulability test presented in Theorem 1, we can approximate the slack $S_{k,q}$ as

$$S_{k,q} = D_{k,q} - C_{k,q} - \left\lfloor \frac{\sum_{\forall \tau_i \neq \tau_k} W_i(D_{k,q}) + W_k(D_{k,q})}{m} \right\rfloor. \quad (11)$$

We further define the normalized slack $\bar{S}_{k,q}$ of thread $\theta_{k,q}$ as $S_{k,q}/D_{k,q}$. When adjusting the slacks of some threads, we call the thread giving its slack to another thread a *donator* thread, the thread receiving the slack from a donator thread a *donee* thread, and the other threads that are not related to slack donation *third-party* threads. In the example shown in Fig. 3(b), $\theta_{2,2}$, $\theta_{2,1}$, and $\theta_{1,1}$ are the donator, the donee, and the third-party thread, respectively.

We design a deadline adjustment method based on the understanding of the underlying priority assignment (OTPA) and analysis methods (see Algorithm 4). There are three key issues in the deadline adjustment: how to determine a donee thread and donator threads, and how to arrange donation. In order to come up with principles to address such issues, we first seek to obtain some understanding about priority assignment with slack donation.

The purpose of slack donation is to assign a priority to a donee thread to make it deemed schedulable. However, the slack donation may impose some undesirable side effect to other threads that are deemed schedulable with their own priorities assigned already.

Observation 1. For some threads $\theta_{i,p}$ and $\theta_{j,q}$, when $D_{i,p}$ decreases, the worst-case interference imposed on $\theta_{j,q}$ may increase.

The above observation implies that when a donator thread decreases its deadline in order to pass some of its slack to a donee thread, it may impose a greater amount of worst-case interference on some other third-party threads, which can lead to violating the schedulability of some third-party threads with their priorities assigned already. This is critical, since it is against one of the most im-

Algorithm 3 PADA (Priority Assignment with Deadline Adjustment).

1: $k \leftarrow 0, A(1) \leftarrow \emptyset, R(1) \leftarrow \tau^{decom}$
2: **repeat**
3: **if** Assign-Priority$(A(k), R(k))$ = failure **then**
4: **if** Adjust-Deadline$(A(k), R(k))$ = failure **then**
5: return unschedule
6: **end if**
7: **end if**
8: **until** $R(k)$ is empty
9: return schedulable

Algorithm 4 Adjust-Deadline($A(k)$, $R(k)$).

1: $F \leftarrow R(k)$
2: **repeat**
3: find the thread with the smallest slack donation request in F to become deemed schedulable according to Theorem 1 (denoted as $\theta^*_{s,e} \in F$)
4: construct a set of donator candidates $DC_s(\theta^*_{s,e})$ such that $\theta_{s,r} \in DC_s(\theta^*_{s,e})$ can donate to $\theta^*_{s,e}$ a slack of at most
5: Ω without violating the schedulability of every already-assigned thread in $A(k)$.
6: save the current offsets and deadlines of all the threads in τ_s
7: **while** $DC_s(\theta^*_{s,e})$ is not empty **do**
8: find the thread with the greatest normalized slack in $DC_s(\theta^*_{s,e})$ (denoted as $\theta^+_{s,i} \in DC_s(\theta^*_{s,e})$)
9: adjust the offsets and deadlines of all the threads in task τ_s to reflect the slack donation of Ω from $\theta^+_{s,i}$ to $\theta^*_{s,e}$
10: **if** thread $\theta^*_{s,e}$ is deemed schedulable by Theorem 1 **then**
11: return success
12: **end if**
13: update the slack of $\theta^+_{s,i}$
14: **end while**
15: restore the offsets and deadlines of all the threads in τ_s that were saved in Line 5.
16: remove $\theta^*_{s,e}$ from F
17: **until** F is emptyreturn failure

[3] For simplicity, we choose to show a case on a uniprocessor platform, but the same phenomenon can also happen on a multiprocessor platform.

portant properties of OTPA, which is that assigning a priority to a thread does not ever affect the schedulability of the already-assigned threads. In order to preserve this important property of OTPA, the deadline adjustment method has a principle of disallowing any slack donation that violates the schedulability of already-assigned threads at this step and pursuing to reduce the potential for such problematic slack donation in the future.

How to determine a donee. The potential for problematic slack donation decreases when the threads with priority assigned have slacks enough to accommodate any potential increase in the worst-case interference that slack donation causes. Thereby, it is important to keep the slacks of threads with priority assigned as much as possible. According to this principle, we seek to minimize the amount of slack donated in total. This way, we select a donee thread (denoted as $\theta^*_{s,e}$) that requires the smallest amount of slack donation to become deemed schedulable according to Theorem 1 (see Line 3 in Algorithm 4).

How to determine a donator. When the donee thread is determined, we construct a set of donator candidate threads (denoted as $DC(\theta^*_{s,e})$) (see Line 4 in Algorithm 4). Each candidate thread $\theta_{s,r} \in DC(\theta^*_{s,e})$ that belongs to the same task τ_s, is deemed schedulable with a priority assigned already, and should be able to donate a slack to the donee without violating the schedulability of all the threads with their priorities assigned already.

The potential for problematic slack donation particularly increases when the smallest slacks of threads with priority assigned become even smaller. This is because we want to avoid even a single case of violating the schedulability of a thread with a priority assigned before. From this perspective, we select a donator thread $\theta^+_{s,r} \in DC(\theta^*_{s,e})$ with the greatest normalized slack among the candidate set (see Line 7 in Algorithm 4).

How to arrange slack donation. The intuition behind how to arrange slack donation is given by the following lemma:

Lemma 3. *For any thread $\theta_{k,q}$, when its deadline $D_{k,q}$ decreases, the worst-case interference imposed on $\theta_{k,q}$ (i.e., $\sum_{\forall\tau_i\neq\tau_k} W_i(D_{k,q}) + W_k(D_{k,q})$) monotonically decreases.*

Proof. In Eqs. (8) and (9), we calculate the maximum amount of execution of τ_i and τ_k in an interval of length $D_{k,q}$. For some given t, $W_{i,p}(D_{k,q}, t)$ and $W_{k,p}(D_{k,q}, t)$ (described in Eq. (7)) only monotonically decreases, as $D_{k,q}$ decreases. In Eq. (8), since $\Delta_i(x, y)$ will be determined to maximize $W_i(D_{k,q})$, there is no case where $W_i(D_{k,q})$ increases as $D_{k,q}$ decreases. Therefore, it is easy to see that $W_i(D_{k,q})$ and $W_k(D_{k,q})$ monotonically decrease as $D_{k,q}$ decreases. Therefore, the lemma holds. □

The above lemma implies that when a donator decreases its deadline to pass a slack to a donee, the donator may get some additional slack after donation. From this implication, we use a reasonably small amount (Ω) of slack in each donation step in order to increase a chance to find such additional slacks (see Line 8), and each donator keeps updating its slack to find some additional slack after donation (see Line 12).

Computational complexity. The PADA algorithm iteratively seeks to assign priorities to individual threads. When it fails to assign a priority through Assign-Priority (Algorithm 2), it invokes Adjust-Deadline (Algorithm 4). In Algorithm 4, constructing a set of donator candidates (Line 4) is a critical factor to the complexity of Algorithm 4, and it performs schedulability tests $O(n^2)$ times for n threads. The while loop (Lines 6–13) repeats at most $S^*_{s,e}/\Omega$ times until the donee thread $\theta^*_{s,e}$ becomes deemed schedulable with slack donation, where $S^*_{s,e} < T_s$. Since the outmost loop (Lines 2–16) repeats at most n times, Algorithm 4 performs schedulability tests $\max\{O(n^3), O(n \cdot T_{max})\}$ times, where T_{max} represents the largest T_i among all tasks $\tau_i \in \tau$. Since PADA invokes Adjust-Deadline at most n times, it thereby performs schedulability tests $\max\{O(n^4), O(n^2 \cdot T_{max})\}$ times.

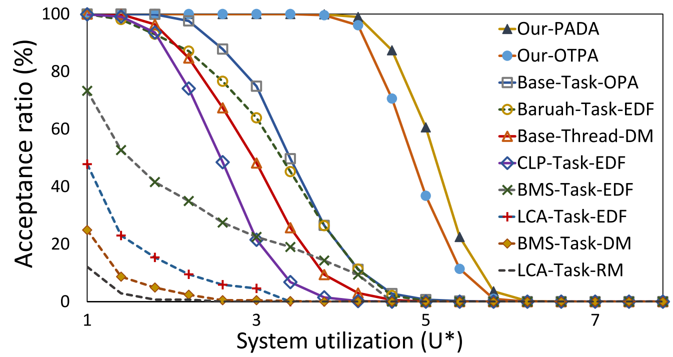


Fig. 4. U^* varying, $N_{max} = 10$, $m = 8$, $p = 0.5$.

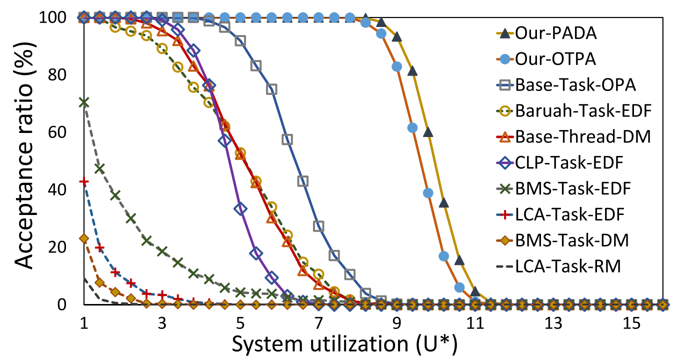


Fig. 5. U^* varying, $N_{max} = 10$, $m = 16$, $p = 0.5$.

6. Evaluation

In this section, we present simulation results to evaluate the proposed thread-level priority assignment algorithms. For presentational convenience, we here define terms. C_i is the sum of the worst-case execution times of all threads of τ_i ($\triangleq \Sigma_q C_{i,q}$), and U_{sys} is the system utilization ($= \sum_{\forall\tau_i\in\tau} C_i/T_i$). Also, L_i is the worst-case execution time of τ_i on infinite number of processors, called *critical execution path*, and LU_{sys} is defined as the maximum L_i/D_i among the tasks $\tau_i \in \tau$.

6.1. Simulation environment

We generate DAG tasks mainly based on the method used in Cordeiro et al. (2010). For a DAG task τ_i, its parameters are determined as follows. The number of nodes (threads) n_i is uniformly chosen in $[1, N_{max}]$, where N_{max} is the maximum number of threads in a task. For each pair of nodes, an edge is generated with the probability of p. The task τ_i and its individual threads $\theta_{i,p}$ are randomly assigned one of three different types: *light*, *medium*, and *heavy*, in which $C_{i,p}$ is determined uniformly in the range of [1, 5], (6, 10], and (11, 40], respectively, and T_i $(= D_i)$[4] is determined such that C_i/T_i is randomly selected in the range of [0.1, 0.3], (0.3, 0.6], or (0.6, 1.0], respectively.

In order to understand how the proposed approaches perform with parallel DAG tasks, we designed three different types of tests using following parameters: 1) system utilization, 2) the degree of parallelism, and 3) the total number of nodes. Figs. 4 and 5 present the schedulability according to U_{sys}, these simulations (annotated as U-test) are designed to show the overall performance with other related methods. We also conduct the second test (annotated as p-test) in order to evaluate our approaches across the different degree of intra-task parallelism. The result is shown in Fig. 6. Finally, we include the

[4] In this section, we only show the results of implicit deadline DAG tasks due to space limit, but the behaviors of constrained deadline DAG tasks are similar to those of implicit ones.

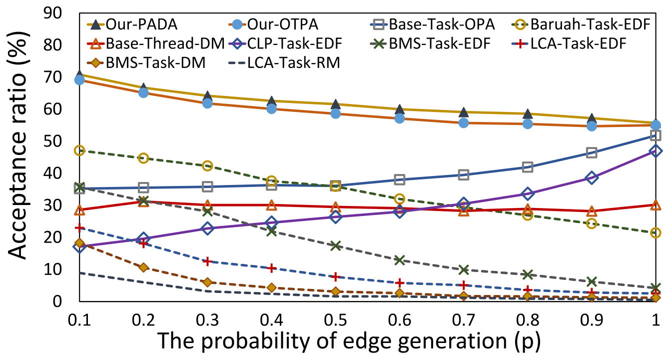


Fig. 6. p varying, $N_{max} = 10$, $m = 8$.

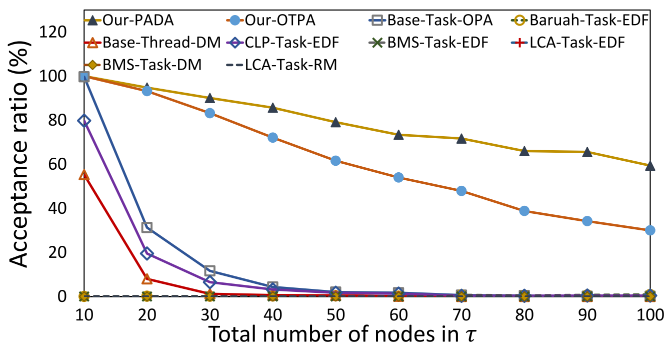


Fig. 7. Total number of nodes varying, $m = 8$, $U^* = 4$, $p = 0.5$.

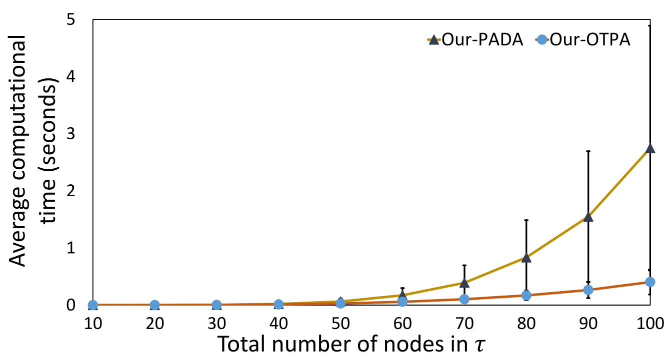


Fig. 8. Total number of nodes varying, $m = 8$, $U^* = 4$, $p = 0.5$.

last test (annoated as node-test) to show the performance according to the total number of nodes in τ for further understanding of the proposed approaches. Results are shown in Figs. 7 and 8.

6.1.1. Task sets generation for U-test (Figs. 4 and 5)

We generate 1000 task sets at each data point with $N_{max} = 10$ for $m = 8$, $m = 16$ respectively, while p is fixed to 0.5.[5]

Due to the difficulty of generating exact U_{sys}, each task set is generated with U^* which falls within in the interval between $U^*_{min} = U^* - 0.005$ and $U^*_{max} = U^* + 0.005$. Then, task sets are generated as following steps.

S1. A random task set is generated by starting with an empty task set, and successively added to as long as $U^* < U^*_{min}$.
S2. If a task is added such that $U^* > U^*_{max}$, we discard this seed set and go to step S1.
S3. If a task is added such that $U^*_{min} \le U^* \le U^*_{max}$, we include this seed set for simulation.

[5] We referred to experimental values (i.e., N_{max}, m, p) in Baruah (2014).

We increase U^* from 1 to 8 for $m = 8$ and 1 to 16 for $m = 16$, in the step of 0.4, which is the sufficient amount to show the tendency. Therefore, we perform simulation with 18,000 and 38,000 task sets in Fig. 4 and Fig. 5, respectively.

6.1.2. Task sets generation for p-test (Fig. 6)

We generate 1000 task sets with $N_{max} = 10$ for $m = 8$, yet leaving p undetermined, as follows.

S1. We first generate a seed task set with m tasks with the parameters determined as described above.
S2. If the U_{sys} 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 one more task into the seed set and go to Step S2 until 1000 task sets are generated.

We now consider constructing edges between nodes (i.e., precedence dependency between threads) with the probability parameter $0 \le p \le 1$. When $p = 0$, there is no edge and thereby no thread has predecessors, maximizing the degree of intra-task parallelism. In contrast, with $p = 1$, 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. As p increases, the number of edges of each DAG task τ_i is increasing, and this generally leads to a longer critical execution path L_i, and then a larger LU_{sys}.

In order to run simulation for different degrees of intra-task parallelism, we perform simulation with 1000 task sets in 10 different cases in terms of p, where we increase p from 0.1 to 1.0 in the step of 0.1, resulting in 10,000 simulations.

6.1.3. Task sets generation for node-test (Figs. 7 and 8)

Most of generation settings in Section 6.1.1 are used for node-test apart from limiting total number of nodes. A task is added to as long as sum of nodes in τ reaches the given total number of nodes, each task can have nodes within [1, the given total number of nodes − sum of nodes in τ] unless task set has only one task. We generate variance to be from 10 to 100 of the total number of nodes for $m = 8$ when p and U^* are fixed to 0.5 and $m/2$, respectively. 1000 simulations are included at each data point, resulting in 10,000 simulations.

6.2. Other approaches for the comparison

We compare our proposed OTPA and PADA approaches (annotated as Our-OTPA and Our-PADA) with other related methods. We first include two baseline approaches: task-level OPA and thread-level DM (annotated as Base-Task-OPA and Base-Thread-DM), in order to evaluate the effectiveness of OTPA and PADA in terms of thread-level priority assignment and incorporation of the characteristics of parallel tasks, respectively. Base-Task-OPA assigns priorities according to the OTPA algorithm, but it restricts that all threads belonging to the same task have the same priority. Base-Thread-DM assigns priorities to threads according to the increasing order of their relative deadlines (i.e., the one having a smaller relative deadline is assigned a higher priority). Both priority assignment algorithms work with our schedulability test in Theorem 1. The above four approaches all require to resolve the precedence dependencies of individual threads through task decomposition. Thus, we transform a DAG task into a synchronous parallel task according to the idea presented in Saifullah et al. (2011), and we use one of the existing task decomposition techniques (Nelissen et al., 2012a) to assign offsets and deadlines to each thread. Although, our approaches can use any decomposition techniques, the technique in Nelissen et al. (2012a) is used since it is designed for different thread execution times.

For the comparison with other known related methods (see Table 1), we include six more methods. For task-level EDF scheduling, four methods are available: pseudo-polynomial time schedulability tests in Bonifaci et al. (2013) (BMS-Task-EDF) and (Baruah, 2014) (Baruah-Task-EDF),[6] an interference-based test in Chwa et al. (2013) (CLP-Task-EDF), and a capacity augmentation bound in Li et al. (2014) (LCA-Task-EDF). For task-level DM scheduling, a pseudo-polynomial time schedulability test in Bonifaci et al. (2013) (BMS-Task-DM) is included. Also, we involve a capacity augmentation bound in Li et al. (2014) (LCA-Task-RM) for task-level RM scheduling. We note that the resource augmentation bounds in Baruah et al. (2012), Andersson and de Niz (2012) and Li et al. (2013) 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. We also omit to include a capacity augmentation bound in Li et al. (2013) since LCA-Task-EDF is the best known capacity augmentation bound of EDF.

For reference, Our-PADA, Baruah-Task-EDF, BMS-Task-EDF, and BMS-Task-DM have pseudo-polynomial time complexity while other methods have polynomial time complexity. We will discuss more details of computational overhead in Section 6.3.

6.3. Simulation results

U-test. Figs. 4 and 5 plot simulation results in terms of the acceptance ratio for $m = 8$ and $m = 16$, respectively, as we varied U^*. The figures show that Our-OTPA and Our-PADA outperform the other methods with wide margins. Showing the effectiveness of thread-level assignment versus task-level, there are large gaps between Our-OTPA and Base-Task-OPA in the both figures. The figures also show that Our-OTPA outperforms Base-Thread-DM significantly as well, indicating that it can be beneficial to incorporate the characteristics of intra-task parallelism into priority assignment. In the figures, Our-OTPA outperforms the other six methods which considered well-known deadline-based scheduling algorithms such as EDF and DM. The results indicate that there is a large room for improving scheduling policies that accommodate the features of parallel tasks. In the figures, Our-PADA is shown to find up to 23.8% (21.3% for $m = 16$) more task sets deemed schedulable, compared to Our-OTPA. Those results show the benefit of deadline adjustment.

p-test. Fig. 6 plots simulation results in terms of acceptance ratio for $m = 8$ as we varied p. As discussed before, an increasing value of p generates a growing number of edges in each DAG task τ_i, leading to a greater degree of precedence constraints between nodes but a smaller degree of intra-task parallelism (i.e., a smaller number of threads in the same segment).

Similar to the above results, Fig. 6 shows that Our-OTPA and Our-PADA perform generally superior to other methods. It demonstrates a need of thread-level priority assignment particularly with a smaller value of p. On the contrary, Our-PADA and Our-OTPA show a comparable performance with Base-Task-OPA and CLP-Task-EDF when p=1, where all the nodes in a DAG task are fully connceted, making τ_i a sequential task. As a consequence, it decreases the efficiency of the finer-grained thread-level scheduling, leading to a comparable result with Task-level approaches.

In spite of this effect, the rest of the five other task-level approaches presented by dash line are shown to perform worse when p increases. This is because those five methods share in common that their schedulability tests check whether LU_{sys} is smaller than or equal to some threshold (i.e., $m/(2m-1)$ in BMS-Task-EDF, approximately $m/(2m-1)$ in Baruah-Task-EDF, $2/(3+\sqrt{5})$ in LCA-Task-EDF, $m/(3m-1)$ in BMS-Task-DM, and $1/(2+\sqrt{3})$ in LCA-Task-RM), and a larger value of p generally increases LU_{sys} for a task set and as described above, it leads to worse schedulability.

node-test. In order to evaluate our approaches over a different number of threads, we run simulation with different value of the total number of nodes for $m = 8$ when $p = 0.5$ and $U^* = 4$. Fig. 7 shows that Our-OTPA and Our-PADA overwhelmingly outperform the other eight methods particularly when task sets have larger number of nodes. Our-PADA is shown to improve schedulability compared to Our-OTPA by up to 98% more. Such an improvement increases with a larger value of nodes. With a fixed value of p, a larger value of nodes ends up with a larger number of segments and this gives a more chance for Our-PADA to adjust segment deadlines for schedulability improvement.

However, according to achieve the improved schedulability, the computational overhead is also increased. We additionally compare the average computational time between Our-OTPA and Our-PADA with the same task sets used in Fig. 7. Fig. 8 presents the increasing gap of computational time between two approaches with larger value of nodes, where Our-PADA is shown to decrease its performance as opposed to Fig. 7. We expect that we can improve this weakness by compromising the trade-off (i.e., limiting iteration numbers in Algorithm 4) as a future study.

7. Conclusion

In the recent past, there is a growing attention to supporting parallel tasks in the context of real-time scheduling (Bonifaci et al., 2013; Li et al., 2014; Baruah et al., 2012; Andersson and de Niz, 2012; Li et al., 2013; Chwa et al., 2013; Baruah, 2014; Saifullah et al., 2011; Nelissen et al., 2012a; Lakshmanan et al., 2010; Liu and Anderson, 2010; 2012; Ferry et al., 2013; Axer et al., 2013; Qi Wang, 2014; Li et al., 2015; Kwon et al., 2015; Melani et al., 2015; Sanjoy Baruah, 2015; Shen Li, 2015). In this paper, we extended real-time scheduling categories, according to the unit of priority assignment, from task-level to thread-level, and we presented, to the best of our knowledge, the first approach to the problem of assigning task-wide thread-level fixed-priorities for global parallel task scheduling on multiprocessors. We showed via experimental validation that the proposed thread-level priority assignment can improve schedulability significantly, compared to its task-level counterpart. Our experiment results also showed that priority assignment can be more effective when incorporating the features of parallel tasks.

This study presented a preliminary result on task-wide thread-level fixed-priority scheduling for parallel tasks, with many further research questions raised. For example, would it be more effective if there exist some new decomposition methods that incorporate the characteristics of the underlying thread-level priority assignment and analysis techniques? Or, would it be better to perform thread-level priority assignment for parallel tasks without task decomposition, if possible? We plan to do further research answering those questions. Another direction of future work is to extend our work taking into account architectural characteristics (Ding and Zhang, 2012; Ding et al., 2013; Ding and Zhang, 2013; Zhang and Ding, 2014; Liu and Zhang, 2015; Liu and Zhang, 2014).

Acknowledgments

This work was supported in part by BSRP (NRF-2010-0006650, NRF-2012R1A1A1014930, NRF-2014R1A1A1035827), NCRC (2010-0028680), IITP (2011-10041313, B0101-15-0557), and NRF (2015M3A9A7067220) funded by the Korea Government (MEST/MSIP/MOTIE). This work was also conducted at High-Speed Vehicle Research Center of KAIST with the support of Defense Acquisition Program Administration (DAPA) and Agency for Defense Development (ADD).

[6] Baruah (2014) improves the schedulability for Global-EDF presented in Bonifaci et al. (2013) by searching all the possible σ in their schedulability test. Due to the time limitation, we follow the same approximation as shown in experiments section in Baruah (2014) instead of searching all the potential space of σ. See the details in Baruah (2014).

References

http://www.amd.com/us/Documents/6000_Series_product_brief.pdf.

Andersson, B., de Niz, D., 2012. Analyzing global-EDF for multiprocessor scheduling of parallel tasks. In: Proceedings of International Conference on Principles of Distributed Systems.

Audsley, N., 1991. Optimal Priority Assignment and Feasibility of Static Priority Tasks With Arbitrary Start Times. Technical Report YCS164. Department of Computer Science, University of York.

Audsley, N.C., 2001. On priority asignment in fixed priority scheduling. Inf. Process. Lett. 79 (1), 39–44.

Axer, P., Quinton, S., Neukirchner, M., Ernst, R., Dobel, B., Hartig, H., 2013. Response-time analysis of parallel fork-join workloads with real-time constraints. In: Proceedings of Euromicro Conference on Real-Time Systems. ECRTS.

Back, H., Chwa, H.S., Shin, I., 2012. Schedulability analysis and priority assignment for global job-level fixed-priority multiprocessor scheduling. In: Proceedings of Real-Time and Embedded Technology and Applications Symposium. RTAS.

Baker, T.P., 2003. Multiprocessor EDF and deadline monotonic schedulability analysis. In: Proceedings of Real-Time and Embedded Technology and Applications Symposium. RTSS.

Baruah, S., 2014. Improved multiprocessor global schedulability analysis of sporadic DAG task systems. In: Proceedings of Euromicro Conference on Real-Time Systems. ECRTS.

Baruah, S., Bonifaci, V., Marchetti-Spaccamela, A., Stougie, L., Wiese, A., 2012. A generalized parallel task model for recurrent real-time processes. In: Proceedings of Real-Time Systems Symposium. RTSS.

Bertogna, M., Cirinei, M., Lipari, G., 2005. Improved schedulability analysis of EDF on multiprocessor platforms. In: Proceedings of Euromicro Conference on Real-Time Systems. ECRTS.

Bertogna, M., Cirinei, M., Lipari, G., 2009. Schedulability analysis of global scheduling algorithms on multiprocessor platforms. IEEE Trans. Parallel Distrib. Syst. 20, 553–566.

Bonifaci, V., Marchetti-Spaccamela, A., Stiller, S., Wiese, A., 2013. Feasibility analysis in the sporadic DAG task model. In: Proceedings of Euromicro Conference on Real-Time Systems. ECRTS.

Cho, H., Ravindran, B., Jensen, E.D., 2006. An optimal real-time scheduling algorithm for multiprocessors. In: Proceedings of Real-Time Systems Symposium. RTSS.

Chwa, H.S., Back, H., Chen, S., Lee, J., Easwaran, A., Shin, I., Lee, I., 2012. Extending task-level to job-level fixed priority assignment and schedulability analysis using pseudo-deadlines. In: Proceedings of Real-Time Systems Symposium. RTSS.

Chwa, H.S., Lee, J., Phan, K.-M., Easwaran, A., Shin, I., 2013. Global EDF schedulability analysis for synchronous parallel tasks on multicore platforms. In: Proceedings of Euromicro Conference on Real-Time Systems. ECRTS.

Comming soon tile-gx100 the first 100 cores processors in the world. http://internalcomputer.com/coming-soon-tile-gx100-the-first-100-cores-processor-in-the-world.computer.

Cordeiro, D., Mouni, G., Perarnau, S., Trystram, D., Vincent, J.-M., Wagner, F., 2010. Random graph generation for scheduling simulations. In: Proceedings of International Conference on Simulation Tools and Techniques. SIMUTools.

Davis, R., Burns, A., 2009. Priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems. In: Proceedings of Real-Time Systems Symposium. RTSS.

Davis, R.I., Burns, A., 2011. A survey of hard real-time scheduling for multiprocessor systems. ACM Comput. Surv. 43, 35:1–35:44.

Dertouzos, M.L., Mok, A.K., 1989. Multiprocessor on-line scheduling of hard-real-time tasks. IEEE Trans. Softw. Eng. 15, 1497–1506.

Ding, Y., Zhang, W., 2012. Multicore-aware code co-positioning to reduce WCET on dual-core processors with shared instruction caches. J. Comput. Sci. Eng. 6 (1), 12–25.

Ding, Y., Wu, L., Zhang, W., 2013. Bounding worst-case DRAM performance on multicore processors. J. Comput. Sci. Eng. 7 (1), 53–66.

Ding, Y., Zhang, W., 2013. Multicore real-time scheduling to reduce inter-thread cache interferences. J. Comput. Sci. Eng. 7 (1), 67–80.

Ferry, D., Li, J., Mahadevan, M., Gill, C., Lu, C., Agrawal, K., 2013. A real-time scheduling service for parallel tasks. In: Proceedings of Real-Time and Embedded Technology and Applications Symposium. RTAS, pp. 261–272.

Guan, N., Stigge, M., Yi, W., Yu, G., 2009. New response time bounds of fixed priority multiprocessor scheduling. In: Proceedings of Real-Time Systems Symposium. RTSS.

Kwon, J., Kim, K.-W., Paik, S., Lee, J., Lee, C.-G., 2015. Multicore scheduling of parallel real-time tasks with multiple parallelization options. In: Proceedings of Real-Time and Embedded Technology and Applications Symposium. RTAS.

Lakshmanan, K., Kato, S., Rajkumar, R., 2010. Scheduling parallel real-time tasks on multi-core processors. In: Proceedings of Real-Time Systems Symposium. RTSS.

Lea, D., 2000. A java fork/join framework. In: Proceedings of the ACM Java Grande Conference.

Lee, J., Easwaran, A., Shin, I., 2010. LLF schedulability analysis on multiprocessor platforms. In: Proceedings of Real-Time Systems Symposium. RTSS.

Lee, J., Easwaran, A., Shin, I., 2011. Maximizing contention-free executions in multiprocessor scheduling. In: Proceedings of Real-Time and Embedded Technology and Applications Symposium. RTAS.

Leung, J., Whitehead, J., 1982. On the complexity of fixed-priority scheduling of periodic real-time tasks. Perform. Eval. 2, 237–250.

Levin, G., Funk, S., Sadowski, C., Pye, I., Brandt, S., 2010. Dp-fair: a simple model for understanding optimal multiprocessor scheduling. In: Proceedings of Euromicro Conference on Real-Time Systems. ECRTS.

Li, J., Agrawal, K., Lu, C., Gill, C.D., 2013. Analysis of global EDF for parallel tasks. In: Proceedings of Euromicro Conference on Real-Time Systems. ECRTS.

Li, J., Chen, J.-J., Agrawal, K., Lu, C., Gill, C., Saifullah, A., 2014. Analysis of federated and global scheduling for parallel tasks. In: Proceedings of Euromicro Conference on Real-Time Systems. ECRTS.

Li, J., Luo, Z., Ferry, D., Agrawal, K., Gill, C., Lu, C., 2015. Global EDF scheduling for parallel real-time tasks. Real-Time Syst. 51 (4), 395–439.

Liu, C., Anderson, J.H., 2010. Supporting soft real-time DAG-based systems on multiprocessors with no utilization loss. In: Proceedings of Real-Time Systems Symposium. RTSS.

Liu, C., Anderson, J.H., 2012. Supporting soft real-time parallel applications on multicore processors. In: Proceedings of Real-Time Computing Systems and Applications. RTCSA, pp. 114–123.

Liu, C., Layland, J., 1973. Scheduling algorithms for multi-programming in a hard-real-time environment. J. ACM 20 (1), 46–61.

Liu, Y., Zhang, W., 2014. Two-level scratchpad memory architectures to achieve time predictability and high performance. J. Comput. Sci. Eng. 8 (4), 215–227.

Liu, Y., Zhang, W., 2015. Scratchpad memory architectures and allocation algorithms for hard real-time multicore processors. J. Comput. Sci. Eng. 9 (2), 51–72.

Melani, A., Bertogna, M., Bonifaci, V., Marchetti-Spaccamela, A., Buttazzo, G.C., 2015. Response-time analysis of conditional DAG tasks in multiprocessor systems. In: Proceedings of Euromicro Conference on Real-Time Systems. ECRTS.

Nelissen, G., Berten, V., Goossens, J., Milojevic, D., 2012a. Techniques optimizing the number of processors to schedule multi-threaded tasks. In: Proceedings of Euromicro Conference on Real-Time Systems. ECRTS.

Nelissen, G., Berten, V., Nelis, V., Goossens, J., Milojevic, D., 2012b. U-EDF: an unfair but optimal multiprocessor scheduling algorithm for sporadic tasks. In: Proceedings of Euromicro Conference on Real-Time Systems. ECRTS.

OpenMP, http://openmp.org.

Qi Wang, G.P., 2014. Fjos: Practical, predictable, and efficient system support for fork/join parallelism. In: Proceedings of Real-Time and Embedded Technology and Applications Symposium. RTAS.

Saifullah, A., Agrawal, K., Lu, C., Gill, C., 2011. Multi-core real-time scheduling for generalized parallel task models. In: Proceedings of Real-Time Systems Symposium. RTSS.

Sanjoy Baruah, A.M.-S., Bonifaci, V., 2015. The global EDF scheduling of systems of conditional sporadic DAG tasks. In: Proceedings of Euromicro Conference on Real-Time Systems. ECRTS.

Shen Li, T.A., Hu, S., 2015. The packing server for real-time scheduling of mapreduce workflows. In: Proceedings of Real-Time and Embedded Technology and Applications Symposium. RTAS.

Srinivasan, A., Anderson, J., 2005. Fair scheduling of dynamic task systems on multiprocessors. J. Syst. Softw. 77(1), 67–80.

Zhang, W., Ding, Y., 2014. Exploiting standard deviation of CPI to evaluate architectural time-predictability. J. Comput. Sci. Eng. 8 (1), 34–42.

Jiyeon Lee received B.S. degree in Computer Science from Dankook Univercity, South Korea in 2012 and M.S. degree in Computer Science from KAIST (Korea Advanced Institute of Science and Technology), South Korea in 2014. She is currently working toward the Ph.D. degree in Computer Science from KAIST. Her research interests include system design and analysis with timing guarantees and resource management in real-time embedded systems and cyber-physical systems.

Hoon Sung Chwa received B.S. and M.S. degrees in Computer Science in 2009 and 2011, respectively, from KAIST (Korea Advanced Institute of Science and Technology), South Korea. He is currently working toward the Ph.D. degree in Computer Science from KAIST. 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.

Jinkyu Lee is an assistant professor in Department of Computer Science and Engineering, Sungkyunkwan University (SKKU), Republic of Korea, where he joined in 2014. He received the B.S., M.S., and Ph.D. degrees in computer science from the Korea Advanced Institute of Science and Technology (KAIST), Republic of Korea, in 2004, 2006, and 2011, respectively. 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.

Insik Shin is currently an associate professor in Department of Computer Science at KAIST, South Korea, where he joined in 2008. He received a B.S. from Korea University, an M.S. from Stanford University, and a Ph.D. from University of Pennsylvania all in Computer Science in 1994, 1998, and 2006, respectively. He has been a post-doctoral research fellow at Malardalen University, Sweden, and a visiting scholar at University of Illinois, Urbana-Champaign until 2008. His research interests lie in cyber-physical systems and real-time embedded systems. He is currently a member of Editorial Boards of Journal of Computing Science and Engineering. He has been co-chairs 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.