석사 학위논문
Master's Thesis

병렬 작업을 위한 thread 단위 스케줄링 기법

Thread-level Priority Assignment in Global Multiprocessor Scheduling for Parallel Tasks

이 지 연 (李 智 娟 Lee, Jiyeon)
전산학과
Department of Computer Science

KAIST

2014

병렬 작업을 위한 thread 단위 스케줄링 기법

Thread-level Priority Assignment in Global Multiprocessor Scheduling for Parallel Tasks

Thread-level Priority Assignment in Global Multiprocessor Scheduling for Parallel Tasks

Advisor : Professor Shin, Insik

by

Lee, Jiyeon

Department of Computer Science

KAIST

A thesis submitted to the faculty of KAIST in partial fulfillment of the requirements for the degree of Master of Science in the Department of Computer Science . The study was conducted in accordance with Code of Research Ethics[1].

2013. 12. 16.

Approved by

Professor Shin, Insik

[Advisor]

---

[1]Declaration of Ethical Conduct in Research: I, as a graduate student of KAIST, hereby declare that I have not committed any acts that may damage the credibility of my research. These include, but are not limited to: falsification, thesis written by someone else, distortion of research findings or plagiarism. I affirm that my thesis contains honest conclusions based on my own careful research under the guidance of my thesis advisor.

병렬 작업을 위한 thread 단위 스케줄링 기법

이 지 연

위 논문은 한국과학기술원 석사학위논문으로
학위논문심사위원회에서 심사 통과하였음.

2013년 12월 16일

심사위원장 신 인 식 (인)

심사위원 최 성 희 (인)

심사위원 김 기 응 (인)

MCS
20100000

이 지 연. Lee, Jiyeon. Thread-level Priority Assignment in Global Multiprocessor Scheduling for Parallel Tasks. 병렬 작업을 위한 thread 단위 스케줄링 기법. Department of Computer Science . 2014. 22p. Advisor Prof. Shin, Insik. Text in English.

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.

Contents

Abstract i
Contents ii
List of Tables iii
List of Figures iv

Chapter 1. Introduction **1**
1.1 Related Work 2
1.2 Our Approach 3

Chapter 2. System Model **5**

Chapter 3. Schedulability Analysis **6**
3.1 Interference-based Schedulability Analysis 6
3.2 Workload-based Schedulability Analysis with Offset 7

Chapter 4. Optimal Thread-level Priority Assignment **10**

Chapter 5. Priority Assignment with Deadline Adjustment **12**

Chapter 6. Evaluation **16**

Chapter 7. Conclusion **20**

References **21**

Summary (in Korean) **23**

List of Tables

1.1 Global scheduling algorithms for parallel tasks 2

List of Figures

2.1 Optional caption for list of figures 5

3.1 The maximum workload of all threads in τ_i in $(x, y]$ with given $\Delta_i(x, y)$ 7

5.1 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. 12

6.1 Schedulability as LU_{sys} changes 17
6.2 Schedulability as m changes 18
6.3 Schedulability as p changes 18
6.4 LU_{sys} as p changes 19
6.5 Schedulability as m changes 19

Chapter 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 [8] are commonplace. In the near future, manycore processors with 100s of cores will be possible [9]. 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 [10], 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 [1–7, 11–15]. Schedulability analysis has been the main subject of much work on thread-level parallelism [1–7, 11–15] for some traditionally well-known scheduling policies, i.e., EDF (Earliest Deadline First) [16] and DM (Deadline Monotonic) [17]. However, a relatively much less effort has been made to understand how to design good scheduling algorithms for parallel tasks.

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 [10]: *task-wide fixed-priority:* a task has a single static priority over all of its invocations (e.g., DM); *job-wide fixed-priority:* a job has a single fixed priority (e.g., EDF); and *dynamic-priority:* a single job may have different priorities at different times (e.g., LLF (Least Laxity First [18])).

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 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] Multiprocessor scheduling approaches can broadly fall into two classes: *global* and *partitioned*. Partitioned approaches

		Task-wide Fixed-Priority	Job-wide Fixed-Priority	Dynamic-Priority
Global	Task-level	DM [1]	EDF [1–5]	
	Thread-level	(*OTPA / PADA [this paper]*)	EDF [6]	PD^2 / U-EDF / LLREF / DP-Wrap [7]

Table 1.1: Global scheduling algorithms for parallel tasks

1.1 Related Work

Recently, supporting intra-task thread-level parallelism in the context of real-time scheduling has received increasing attention in the recent past [1–7, 11–15]. The work in [12, 13] 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.

The fork-join task model is one of the popular parallel task models [19, 20], 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.* [11] 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.* [11] presented a resource augmentation bound[2] of 3.42 for partitioned thread-level DM scheduling.

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.* [6] presented another decomposition method and proved a capacity 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.* [14] presented a prototype scheduling service for their *RT-OpenMP* concurrency platform. Nelissen, *et al.* [7] 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 [21], LLREF [22], DP-Wrap [23], or U-EDF [24]. Recently, Chwa, *et al.* [5] introduced an interference-based analysis for global task-level EDF scheduling, and Axer, *et al.* [15] presented a response-time analysis (RTA) for partitioned thread-level fixed-priority scheduling.

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.* [2] 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

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.* [4] 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 [4] more details).

scheduling [1,3,4]. Li, *et al.* [4] also showed a $4-2/m$ capacity augmentation bound for global task-level EDF, and Bonifaci, *et al.* [1] also derived a $3-1/m$ resource augmentation bound for global task-level DM scheduling.

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 1.1 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 of real-time workloads, deadlines are a good real-time scheduling parameter, in particular, for sequential tasks on a single processor [16]. 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 1.1 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 parallel 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 [25–27], 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 [6,7,11,14,15]. 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 [6, 11, 14] 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 [6,11,14], 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.

Chapter 2. System Model

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 Figure 2.1(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 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.

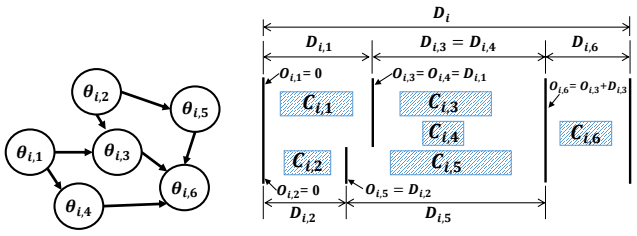
A sporadic DAG task τ_i invokes a series of jobs with the minimum separation of T_i, and each job should finish its execution within D_i (the relative deadline). We denote as J_i^h the h-th job of τ_i.

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 as 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. Figure 2.1(b) illustrates a decomposed task, which corresponds to the DAG task in Figure 2.1(a).

For a job J_i^h, the primary thread $\theta_{i,1}$ is released at $r_{i,1}^h$, and has deadline $d_{i,1}^h = r_{i,1}^h + D_{i,1}$. Then, the other thread $\theta_{i,p}$ 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 $(r_{i,p}^h, d_{i,p}^h]$.

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 $P_{i,p}$.



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

Figure 2.1: A DAG task and its decomposed task

Chapter 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 analyses. 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 [27].

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 [28] as follows:

$$I_{k,q}(a,b) = \frac{\sum_{(i,p)\neq(k,q)} I_{(i,p)\to(k,q)}(a,b)}{m}. \tag{3.1}$$

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

$$I^*_{k,q} \triangleq \max_h(I_{k,q}(r^h_{k,q}, d^h_{k,q})) = I_{k,q}(r^*_{k,q}, d^*_{k,q}). \tag{3.2}$$

Using the above definitions, the studies [28, 29] developed the exact schedulability condition of global multiprocessor scheduling algorithms for sequential tasks, which can be extended to parallel tasks as follows:

Lemma 1 (from [28, 29]) *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 $\theta_{k,q}$:*

$$\sum_{\theta_{i,p}\in\tau^{decom}\setminus\{\theta_{k,q}\}} \min(I^*_{(i,p)\to(k,q)}, D_{k,q} - C_{k,q} + 1) < m \cdot (D_{k,q} - C_{k,q} + 1). \tag{3.3}$$

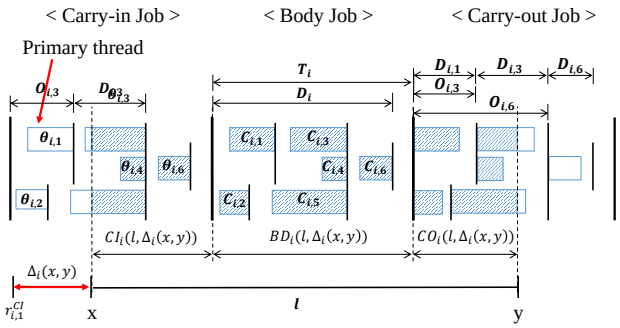


Figure 3.1: The maximum workload of all threads in τ_i in $(x, y]$ with given $\Delta_i(x, y)$

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 [6]) *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 [28–35] have derived upper-bounds on the interference under target algorithms, resulting in sufficient schedulability analyses. 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 subsection.

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. (3.3).

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 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.

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 refer to the difference between the release time of the primary thread of the carry-in job in $(x, y]$ as $\Delta_i(x, y) = x - r_{i,1}^{CI}$ as shown in Figure 3.1. 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

$$CI_i(l, \Delta_i(x, y)) = \min(T_i - \Delta_i(x, y), l), \tag{3.4}$$

$$BD_i(l, \Delta_i(x, y)) = \left\lfloor \frac{l - CI_i(l, \Delta_i(x, y))}{T_i} \right\rfloor \cdot T_i, \tag{3.5}$$

$$CO_i(l, \Delta_i(x, y)) = l - CI_i(l, \Delta_i(x, y)) - BD_i(l, \Delta_i(x, y)). \tag{3.6}$$

Then, with the given $\Delta_i(x, y)$, the workload contribution of each thread in $(x, y]$ (shown in Figure 3.1) is calculated as

$$W_{i,p}(l, \Delta_i(x, y)) = W_{i,p}^{CI} + W_{i,p}^{BD} + W_{i,p}^{CO}, \tag{3.7}$$

where

$$W_{i,p}^{CI} = \Big[\min(O_{i,p} + D_{i,p}, \Delta_i(x, y) + l) - \max(\Delta_i(x, y), O_{i,p}) \Big]_0^{C_{i,p}},$$
$$W_{i,p}^{BD} = \left\lfloor \frac{l - CI_i(l, \Delta_i(x, y))}{T_{i,p}} \right\rfloor \cdot C_{i,p},$$
$$W_{i,p}^{CO} = \Big[CO_i(l, \Delta_i(x, y)) - O_{i,p} \Big]_0^{C_{i,p}}.$$

Note that $[X]_a^b$ means $\min(\max(X, a), b)$.

We will prove that $W_{i,p}^{CI}$, $W_{i,p}^{BD}$, and $W_{i,p}^{CO}$ 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 $W_{i,p}^{CI}$, 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 $r_{i,1}^{CI}$ 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 Figure 3.1. 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 $W_{i,p}^{CI}$ in Eq. (3.7).

When it comes to $W_{i,p}^{BD}$, 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, $W_{i,p}^{BD}$ equals to the number multiplied by the execution time $C_{i,p}$.

The derivation of $W_{i,p}^{CO}$ is similar to that of $W_{i,p}^{CI}$. 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 $r_{i,1}^{CO}$ to 0, where $r_{i,1}^{CO}$ 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 Figure 3.1. Since the carry-out job cannot execute more than its execution time $C_{i,p}$ and less than 0, we derive $W_{i,p}^{CO}$ in Eq. (3.7).

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,

$$W_i(D_{k,q}) = \max_{0 \le \Delta_i(x,y) < T_i} \sum_{\forall \theta_{i,p} \in hp(\theta_{k,q})} \min(W_{i,p}(D_{k,q}, \Delta_i(x, y)), D_{k,q} - C_{k,q} + 1). \tag{3.8}$$

When $i = k$. In the case of $i = k$, 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. (3.7). Thus, the maximum workload of all threads of τ_k that have a higher priority than thread $\theta_{k,q}$ is calculated as

$$W_k(D_{k,q}) = \sum_{\forall \theta_{k,p} \in hp(\theta_{k,q})} \min(W_{k,p}(D_{k,q}, O_{k,q}), D_{k,q} - C_{k,q} + 1). \tag{3.9}$$

Based on the upper-bound on the interference calculated in Eqs. (3.8) and (3.9), 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:*

$$\sum_{\forall \tau_i \neq \tau_k} W_i(D_{k,q}) + W_k(D_{k,q}) < m \cdot (D_{k,q} - C_{k,q} + 1). \tag{3.10}$$

***proof* 1** *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.* (3.3) *is upper-bounded by the LHS of Eq.* (3.10)*. By Lemma 1, the theorem holds.*

Chapter 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 [27] 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

$A(k)$ denotes a subset of threads whose priorities have been assigned before the k-th step, and

$R(k)$ 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* 2** *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. (3.10)*, 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.*

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.

Algorithm 1 OTPA (Optimal Thread-level Priority Assignment)

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

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

1: **for** each thread $\theta_{p,q} \in R(k)$ **do**
2: **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**
3: assign priority k to $\theta_{p,q}$ ($P_{p,q} \leftarrow k$)
4: $R(k+1) \leftarrow R(k) \setminus \{\theta_{p,q}\}$
5: $A(k+1) \leftarrow A(k) \cup \{\theta_{p,q}\}$
6: return success
7: **end if**
8: **end for**
9: return failure

Therefore, $W_i(D_{k,q})$ and $W_k(D_{k,q})$ in Eqs. (3.8) *and* (3.9) *get smaller after the swap, resulting in a decrease in the LHS of Eq.* (3.10)*. 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.

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.

Chapter 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 [6, 7] 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.* [6] decompose a parallel task such that the density of each segment is upper bounded by some value, and Nelissen, *et al.* [7] decompose a parallel task such that the maximum density among all segments in a parallel task is minimized. Then, they apply those density bounds to the existing density-based schedulability analysis and derive resource augmentation bounds. In case of [7], deadline decomposition is optimal according to a sufficient schedulability test for scheduling algorithms such as PD^2 [21], LLREF [22], DP-Wrap [23], or U-EDF [24].

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, Figure 5.1 shows two tasks with three threads on a single processor[1]. Task τ_2 has two threads with their execution times of 1 and 4. Figure 5.1(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 Figure 5.1(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

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

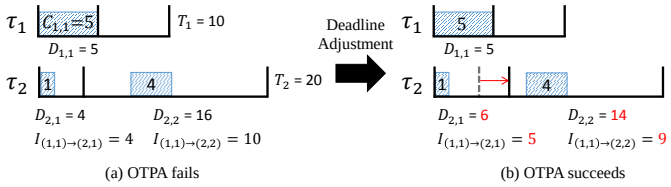


Figure 5.1: An example of two tasks on a single processor illustrates situations, where (a) thread-level priority assignment fails when the deadline of each thread in τ_2 is assigned in proportion to thread execution time and (b) thread-level priority assignment becomes successful with the deadline of each thread in τ_2 properly adjusted.

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

$$S_{k,q} = D_{k,q} - C_{k,q} - \Big\lfloor \frac{\sum_{\forall \tau_i \neq \tau_k} W_i(D_{k,q}) + W_k(D_{k,q})}{m} \Big\rfloor. \tag{5.1}$$

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 Figure 5.1(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 important 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

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 unscheduble
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
Ω without violating the schedulability of every already-assigned thread in $A(k)$.
5: save the current offsets and deadlines of all the threads in τ_s
6: **while** $DC_s(\theta^*_{s,e})$ is not empty **do**
7: find the thread with the greatest normalized slack in $DC_s(\theta^*_{s,e})$ (denoted as $\theta^+_{s,i} \in DC_s(\theta^*_{s,e})$)
8: 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 $\theta^*_{s,e}$
9: **if** thread $\theta^*_{s,e}$ is deemed schedulable by Theorem 1 **then**
10: return success
11: **end if**
12: update the slack of $\theta^+_{s,i}$
13: **end while**
14: restore the offsets and deadlines of all the threads in τ_s that were saved in Line 5.
15: remove $\theta^*_{s,e}$ from F
16: **until** F is empty
return failure

$\theta_{s,r} \in DC(\theta^*_{s,e})$ 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* 3** *In Eqs.* (3.8) *and* (3.9)*, 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.* (3.7)*) only monotonically decreases, as* $D_{k,q}$ *decreases. In Eq.* (3.8)*, 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).

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.

Chapter 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 ($\triangleq \sum_q C_{i,q}$), and U_i is the task utilization ($\triangleq C_i/T_i$). 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/T_i among the tasks $\tau_i \in \tau$.

Simulation environment. We generate DAG tasks mainly based on the method used in [36]. For a DAG task τ_i, its parameters are determined as follows. 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 p. The task τ_i and its individual threads are randomly assigned one of three different types: *light*, *medium*, and *heavy*, in which C_i is determined uniformly in the range of $[1, 5]$, $(5, 20]$, and $(20, 80]$, respectively, and T_i $(= D_i)$[1] 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.

We generate 1,000 task sets for $m = 4$, where m is the number of processors, yet leaving p undetermined, as follows.

S1 We first generate a seed task set with m tasks with the parameters (except p) determined as described above.

S2 If the system utilization U_{sys} (i.e., $U_{sys} = \sum_{\forall \tau_i \in \tau} C_i/T_i$) 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 one more task into the seed set and go to Step S2 until 1,000 task sets are generated.

We now consider constructing edges between nodes (i.e., precedence dependency between threads) with the probability parameter $0 \leq p \leq 1$. 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_i as shown in Figure 6.4. In order to run simulation for different degrees of intra-task parallelism, we perform simulation with 1,000 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.

We compare our proposed OTPA and PADA approaches (annotated as Our-OTPA and Our-PADA) with other related methods. In order to evaluate the effectiveness of OPTA and PADA in terms of thread-level priority assignment and incorporation of the characteristics of parallel tasks, we developed two baseline approaches: task-level OPA and thread-level DM (annotated as Base-Task-OPA and Base-Thread-DM, 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

[1] 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.

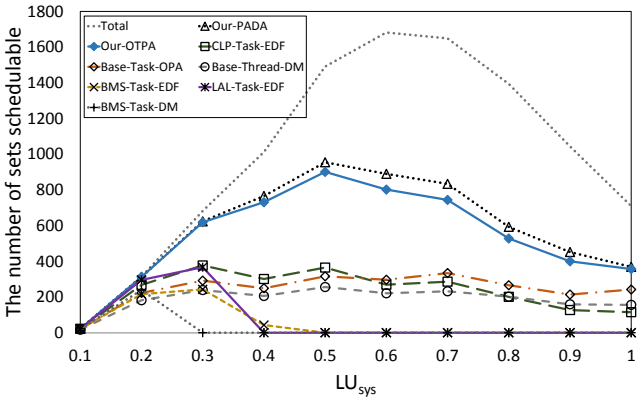


Figure 6.1: Schedulability as LU_{sys} changes

dependencies of individual threads through task decomposition. Thus, we transform a DAG task into a synchronous parallel task according to the idea presented in [6], and we use one of the existing task decomposition techniques [7] to assign offsets and deadlines to each thread. The technique in [7] is used since it is designed for different thread execution times.

For the comparison with other known related methods (see Table 1.1), we include four more methods. For task-level EDF scheduling, three methods are available: a schedulability test (i.e., Theorem 21) in [1] (BMS-Task-EDF), a capacity augmentation bound in [4] (LAL-Task-EDF), and an interference-based test (i.e., Theorem 2) in [5] (CLP-Task-EDF). For task-level DM scheduling, a schedulability test (i.e., Theorem 22) in [1] (BMS-Task-DM) is included. We note that the resource augmentation bounds in [1–4] 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.

Simulation results. Figure 6.1 plots the results of 10,000 simulations for $m = 4$ in terms of the number of task sets deemed schedulable over different values of LU_{sys}, where $LU_{sys} = \max_{\tau_i \in \tau}\{L_i/D_i\}$. In the figure, the line annotated as 'Total' represents the number of task sets used in the simulation. The figure shows that Our-OTPA outperforms Base-Task-OPA by a wide margin, showing the effectiveness of thread-level assignment versus task-level. The figure also shows 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 (through its underlying analysis). In the figure, Our-PADA is shown to find up to 12% (7% in average) more task sets deemed schedulable, compared to Our-OTPA, showing the benefit of deadline adjustment.

In the figure, the proposed Our-OTPA and Our-PADA outperform the four related methods. Unlike the others, the three methods of LAL-Task-EDF, BMS-Task-EDF, and BMS-Task-DM are shown to be much sensitive to LU_{sys}. This is because those three methods share in common that their schedulability tests check whether LU_{sys} is smaller than or equal to some threshold (e.g., $1/(4\text{-}2/m)$ in LAL-Task-EDF, $1/3$ in BMS-Task-EDF). On the other hand, the other five methods are able to find some task sets schedulable as long as $LU_{sys} \leq 1$.

We seek to compare the eight different methods in terms of the smallest number of processors needed to make some given task sets schedulable. To this end, Figure 6.2 shows the number of task sets deemed schedulable as we varied m with $0 < LU_{sys} \leq 0.4$[2]. We note that we run simulation with 2,000 task

[2]Due to space limit, another figure with $0 < LU_{sys} \leq 1$ is presented in Appendix B.

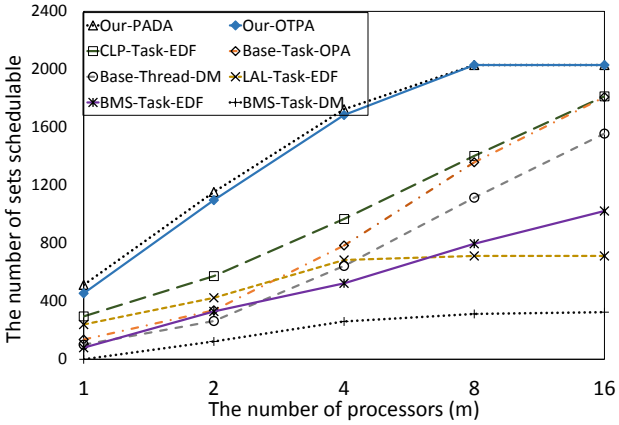


Figure 6.2: Schedulability as m changes

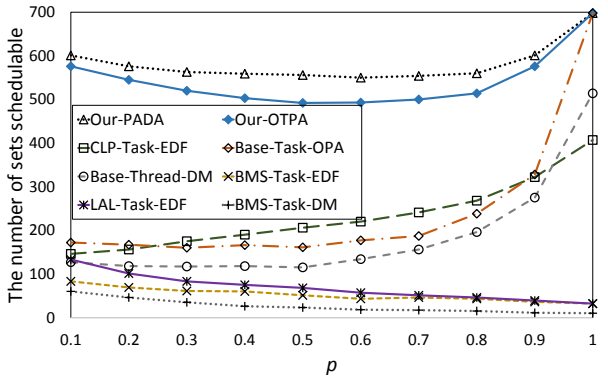


Figure 6.3: Schedulability as p changes

sets, and all those task sets have their task utilization $U_{sys} \leq 4.0$. The figure shows that Our-OTPA and Our-PADA outperform the other methods, requesting a much smaller number of processors to support the timing constraints of parallel tasks.

Figure 6.3 plots the number of task sets deemed schedulable as we varied p for $m = 4$. With the increase of p, each DAG task τ_i has an increasing number of edges and this generally leads to a longer L_i and a greater LU_{sys} (as shown in Figure 6.4).

As mentioned in Section **??**, when $p = 0$, there is no edge at all, representing no precedence dependency between threads. This corresponds to the case where the degree of intra-task parallelism is maximized. In contrast, when $p = 1$, all the nodes are fully connected with directed edges, representing a situation where the maximum degree of precedence dependency is imposed on threads and only one single thread can execute at a time but no two or more threads can run in parallel. This is then the case where the degree of intra-task parallelism is minimized.

The figure shows that the proposed Our-OTPA and Our-PADA outperform all the other methods by wide margins, except $p = 1$. In particular, when $p = 1$, the three approaches, Our-OTPA, Our-PADA, Base-Task-OPA, yield the same results. This is because our task decomposition method transforms a DAG-based parallel task into a sequential task when $p = 1$. We note that our task decomposition method is mainly based on the technique in [7] with one additional adjustment. The additional adjustment is made for two consecutive segments each of which has only a single thread. In such a case, those two consecutive segments are combined into a single segment, with two corresponding threads combined into one as well. Since every segment in a task τ_i has exactly only one thread when $p = 1$, our task decomposition method

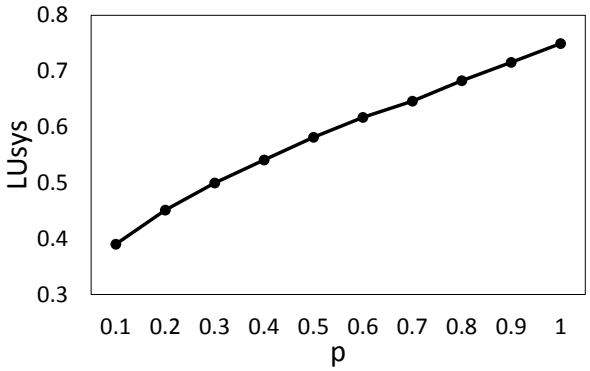


Figure 6.4: LU_{sys} as p changes

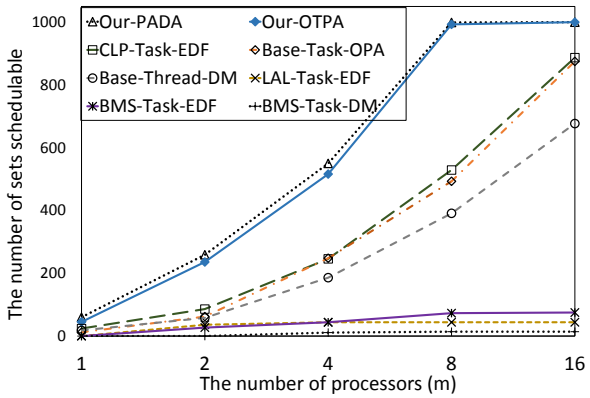


Figure 6.5: Schedulability as m changes

combines all the threads/segments into one thread/segment, making τ_i a sequential task. With such a transformation, the figure shows that the two methods, Our-OTPA and Base-Task-OPA, behave in the same way.

The methods of LAL-Task-EDF, BMS-Task-EDF, and BMS-Task-DM are sensitive to LU_{sys} due to their own schedulability analysis. As shown in Figure 6.4, LU_{sys} increases as p increases. Hence, those three methods find a smaller number of task sets deemed schedulable as p increases, while the other methods stay less sensitive except $p = 1$.

Figure 6.5 compares the eight different methods in terms of the smallest number of processors needed to make some given task sets schedulable. The figure shows the number of task sets deemed schedulable as we varied m with $0 < LU_{sys} \leq 1.0$. We note that we run simulation with 10,000 task sets, and all those task sets have their task utilization $U_{sys} \leq 4.0$. The trend shown in this figure is similar to Figure 6.2. The figure shows that Our-OTPA and Our-PADA outperform the other methods, requesting a much smaller number of processors to support the timing constraints of parallel tasks.

Chapter 7. Conclusion

In the recent past, there is a growing attention to supporting parallel tasks in the context of real-time scheduling [1–7, 11–15]. 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.

References

[1] V. Bonifaci, A. Marchetti-Spaccamela, S. Stiller, and A. Wiese, "Feasibility analysis in the sporadic DAG task model," in *ECRTS*, 2013.

[2] S. Baruah, V. Bonifaci, A. Marchetti-Spaccamela, L. Stougie, and A. Wiese, "A generalized parallel task model for recurrent real-time processes," in *RTSS*, 2012.

[3] B. Andersson and D. de Niz, "Analyzing global-EDF for multiprocessor scheduling of parallel tasks," in *Proceedings of Int. Conf. on Principles of Distributed Systems*, 2012.

[4] J. Li, K. Agrawal, C. Lu, and C. D. Gill, "Analysis of global EDF for parallel tasks," in *ECRTS*, 2013.

[5] 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 *ECRTS*, 2013.

[6] A. Saifullah, K. Agrawal, C. Lu, and C. Gill, "Multi-core real-time scheduling for generalized parallel task models," in *RTSS*, 2011.

[7] G. Nelissen, V. Berten, J. Goossens, and D. Milojevic, "Techniques optimizing the number of processors to schedule multi-threaded tasks," in *ECRTS*, 2012.

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

[9] http://internalcomputer.com/coming-soon-tile-gx100-the-first-100-cores-processor-in-the-world.computer.

[10] R. I. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM Computing Surveys*, vol. 43, pp. 35:1–35:44, October 2011.

[11] K. Lakshmanan, S. Kato, and R. Rajkumar, "Scheduling parallel real-time tasks on multi-core processors," in *RTSS*, 2010.

[12] C. Liu and J. H. Anderson, "Supporting soft real-time DAG-based systems on multiprocessors with no utilization loss," in *RTSS*, 2010.

[13] ——, "Supporting soft real-time parallel applications on multicore processors," in *RTCSA*, 2012.

[14] D. Ferry, J. Li, M. Mahadevan, C. Gill, C. Lu, and K. Agrawal, "A real-time scheduling service for parallel tasks," in *RTAS*, 2013.

[15] 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 *ECRTS*, 2013.

[16] C. Liu and J. Layland, "Scheduling algorithms for multi-programming in a hard-real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[17] J. Leung and J. Whitehead, "On the complexity of fixed-priority scheduling of periodic real-time tasks," *Performance Evaluation*, vol. 2, pp. 237–250, 1982.

[18] M. L. Dertouzos and A. K. Mok, "Multiprocessor on-line scheduling of hard-real-time tasks," *IEEE Transactions on Software Engineering*, vol. 15, pp. 1497–1506, 1989.

[19] D. Lea, "A java fork/join framework," in *Proceedings of the ACM Java Grande Conference*, 2000.

[20] *OpenMP*, http://openmp.org .

[21] A. Srinivasan and J. Anderson, "Fair scheduling of dynamic task systems on multiprocessors," *Journal of Systems and Software*, vol. 77(1), pp. 67–80, 2005.

[22] H. Cho, B. Ravindran, and E. D. Jensen, "An optimal real-time scheduling algorithm for multiprocessors," in *RTSS*, 2006.

[23] G. Levin, S. Funk, C. Sadowski, I. Pye, and S. Brandt, "Dp-fair: A simple model for understanding optimal multiprocessor scheduling," in *ECRTS*, 2010.

[24] G. Nelissen, V. Berten, V. Nelis, J. Goossens, and D. Milojevic, "U-EDF: An unfair but optimal multiprocessor scheduling algorithm for sporadic tasks," in *ECRTS*, 2012.

[25] N. Audsley, "Optimal priority assignment and feasibility of static priority tasks with arbitrary start times," Department of Computer Science, University of York, Tech. Rep. YCS164, 1991.

[26] N. C. Audsley, "On priority asignment in fixed priority scheduling," *Inf. Process. Lett.*, vol. 79, no. 1, pp. 39–44, May 2001.

[27] R. Davis and A. Burns, "Priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems," in *RTSS*, 2009.

[28] M. Bertogna, M. Cirinei, and G. Lipari, "Improved schedulability analysis of EDF on multiprocessor platforms," in *ECRTS*, 2005.

[29] M. Bertogna, M. Cirinei, and G. Lipari, "Schedulability analysis of global scheduling algorithms on multiprocessor platforms," *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, pp. 553–566, 2009.

[30] T. P. Baker, "Multiprocessor EDF and deadline monotonic schedulability analysis," in *RTSS*, 2003.

[31] N. Guan, M. Stigge, W. Yi, and G. Yu, "New response time bounds of fixed priority multiprocessor scheduling," in *RTSS*, 2009.

[32] J. Lee, A. Easwaran, and I. Shin, "LLF Schedulability Analysis on Multiprocessor Platforms," in *RTSS*, 2010.

[33] ——, "Maximizing contention-free executions in multiprocessor scheduling," in *RTAS*, 2011.

[34] H. Back, H. S. Chwa, and I. Shin, "Schedulability analysis and priority assignment for global job-level fixed-priority multiprocessor scheduling," in *RTAS*, 2012.

[35] 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 *RTSS*, 2012.

[36] D. Cordeiro, G. Mouni, S. Perarnau, D. Trystram, J.-M. Vincent, and F. Wagner, "Random graph generation for scheduling simulations," in *SIMUTools*, 2010.

Summary

Thread-level Priority Assignment in Global Multiprocessor Scheduling for Parallel Tasks

최근 멀티코어 기반 시스템이 급격히 증가하면서, 병렬성을 극대화할 수 있는 프로그래밍 환경이 꾸준히 소개되고 있다. 예를 들어, OpenMP 및 맵-리듀스 등과 같은 프로그래밍 환경은 한 task가 여러 개의 thread들로 나누어져 동시에 다중코어에서 수행 가능한 병렬 작업을 지원하고 있다. 실시간 시스템 분야에서 이러한 병렬 작업을 위한 스케줄링 기법에 대한 연구가 최근에 많은 관심을 받고 있다. 대부분의 기존 연구에서는 순차 작업을 위해 개발된 성능이 우수한 스케줄링 기법을 병렬 작업에 적용하고 있다. 하지만 순차 작업을 위한 스케줄링 기법은 병렬 작업의 task가 갖는 다중 thread 구조를 고려하고 있지 않아 낮은 수준의 schedulability 성능을 보인다. 이와 같이 실시간 시스템 분야에서 병렬 작업을 위한 스케줄링 연구는 아직 초기 수준에 머물고 있다. 이에, 본 연구는 병렬 작업의 스케줄링 기본 단위를 세분화 하여 각 단위에 따라 기존의 스케줄링 알고리즘을 분류하고, 병렬 작업의 thread 단위 스케줄링을 위한 새로운 우선순위 할당 기법을 소개한다. 본 연구의 결과는 thread와 같은 세분화된 단위의 병렬 작업 스케줄링이 task와 같은 큰 단위의 스케줄링에 비해 schedulability가 크게 향상됨을 보인다.

감 사 의 글

먼저 아무것도 할줄 모르는 부족한 저를 이 자리까지 이끌어 주신 신인식 교수님께 진심으로 감사드립니다. CPS 연구실의 부원으로 뽑힌건 제가 태어나서 가장 잘한 일이 아닌가 생각됩니다. 앞으로 박사과정동안 열심히 하는 것 뿐만 아니라 잘 하는 학생으로 성장해서 교수님께 부끄럽지 않은 제자가 되도록 더욱 더 노력하겠습니다. 그리고 바쁘신 가운데 논문심사를 맡아주시고 좋은 조언을 해주신 최성희 교수님과 김기응 교수님께 감사드립니다. 또한 저에게 연구자로서의 미래를 꿈꾸게 해주신 저의 영원한 우상이신 단국대 최종무 교수님과 서울대 민상렬 교수님 진심으로 존경합니다. 그리고 같이 생활하는 CPS 연구실 식구들에게도 감사의 말씀 드립니다. 남자들끼리만 생활하다가 여학생인 제가 들어와서 불편한 점도 많았을텐데 항상 먼저 챙겨주고 배려해 주셔서 감사드립니다. 특히 이 논문을 준비하면서 많은 도움을 주셨던 진규오빠와 훈승오빠께 진심으로 감사드립니다.

끝으로 자주뵙지는 못하지만 누구보다도 저를 사랑해 주시는 저희 가족에게 감사하다는 말씀을 드립니다. 항상 저를 격려해 주시고 응원해 주시는 아버지와 제 투정을 묵묵히 다 받아주시고 집에 갈때마다 든든한 밥 지어주시는 어머니께 감사드립니다. 그리고 어려운 고민이 있을 때 이야기를 들어주고 소중한 조언을 해주는 언니와 새 식구가 된 형부께도 감사말씀 전합니다. 앞으로 더욱 자랑스러운 딸, 듬직한 동생이 되도록 노력하겠습니다. 감사합니다.

이 력 서

이 름 : 이 지 연

생 년 월 일 : 1987년 8월 20일

주 소 : 대전 유성구 구성동 한국과학 기술원 전산학동(E3-1) 4427호

E-mail 주 소 : jy.lee@cps.kaist.ac.kr

학 력

2007. 3. – 2012. 2. 단국대학교 컴퓨터공학과 (B.S.)

2012. 2. – 2014. 2. 한국과학기술원 전산학과 (M.S.)

학 회 활 동

1. **Jiyeon Lee**, Hoon Sung Chwa, and Insik Shin, *Optimal Priority Assignment for Parallel Tasks under Fixed Priority Scheduling in Real-time Systems*, 40th Korean Institute of Information Scientists and Engineers, Jeju (Korea), November., 2013.