공학박사 학위논문

Conditionally Optimal Parallelization of Real-Time Tasks for Global EDF Scheduling

Global EDF 스케줄링을 위한 실시간 태스크의 조건부 최적 병렬화

2021년 12월

서울대학교 대학원
컴퓨터공학부
조 영 은

Conditionally Optimal Parallelization of Real-Time Tasks for Global EDF Scheduling

지도 교수 이 창 건

이 논문을 공학박사 학위논문으로 제출함

2021년 12월

서울대학교 대학원

컴퓨터공학부

조 영 은

조영은의 공학박사 학위논문을 인준함

2021년 12월

위 원 장 하 순 회 (인)

부위원장 이 창 건 (인)

위 원 오 성 회 (인)

위 원 이 영 기 (인)

위 원 김 종 찬 (인)

Abstract

Conditionally Optimal Parallelization of Real-Time Tasks for Global EDF Scheduling

Youngeun Cho
School of Computer Science and Engineering
The Graduate School
Seoul National University

Real-time applications are rapidly growing in their size and complexity. This trend is apparent even in extreme deadline-critical sectors such as autonomous driving and artificial intelligence. Such complex program models are described by a directed-acyclic-graph (DAG), where each node of the graph represents a task, and the connected edges portray the precedence relation between the tasks. With this intricate structure and intensive computation requirements, extensive system-wide optimization is required to ensure a stable real-time execution.

On the other hand, recent advances in parallel computing frameworks, such as OpenCL and OpenMP, allow us to parallelize a real-time task into many different versions, which is called "parallelization freedom." Depending on the degree of paralleliza-

tion, the thread execution times can vary significantly, that is, more parallelization tends to reduce each thread execution time but increase the total execution time due to parallelization overhead.

By carefully selecting a "parallelization option" for each task, i.e., the chosen number of threads each task is parallelized into, we can maximize the system schedulability while satisfying real-time constraints. Because of this benefit, parallelization freedom has drawn recent attention. However, for global EDF scheduling, G-EDF for short, the concept of parallelization freedom still has not brought much attention. To this extent, this dissertation proposes a way of optimally assigning parallelization options to real-time tasks for G-EDF on a multi-core system. Moreover, we aim to propose a polynomial-time algorithm that can be used for online situations where tasks dynamically join and leave.

We formalize a monotonic increasing property of both tolerance and interference to the parallelization option to achieve this. Using such properties, we develop a uni-directional search algorithm that can assign parallelization options in polynomial time, which we formally prove the optimality.

With the optimal parallelization, we observe significant improvement of schedulability through simulation experiment, and then in the following implementation experiment, we demonstrate that the algorithm is practically applicable for real-world use-cases.

This dissertation first focuses on the traditional task model, i.e., multi-thread task model, then extends also to target the multi-segment (MS) task model and finally discusses the more general directed-acyclic-graph (DAG) task model to accommo-

date a wide range of real-world computing models.

keywords : Real-Time, Multi-Core Scheduling, Parallelization Freedom, Optimal Parallelization

Student Number : 2015-21265

Table of Contents

List of Figures

Chapter 1
Introduction

1.1 Motivation and Objective

Real-time applications are rapidly growing in their size and complexity. This trend is apparent even in extreme deadline-critical sectors such as autonomous driving. To illustrate, AutowareAuto [26], a leading open-source autonomous driving framework, can easily have over 20 tightly integrated modules, just for the perception stack. Such complex program models are described by a directed-acyclic-graph (DAG), where each node of the graph represents a task, and the connected edges portray the precedence relation between the tasks. With this intricate structure and intensive computation requirements, extensive system-wide optimization is required to ensure a stable real-time execution.

On the other hand, recent advances in parallel computing frameworks, such as OpenCL [56] and OpenMP [24], allow us to parallelize a real-time task into many different versions, which is called "parallelization freedom." For example, an object detection program (Euclidean clustering algorithm [39]) can be parallelized into a desirable number of threads as shown in Fig. 1.1. Depending on the degree of parallelization, the thread execution times can vary significantly, that is, more parallelization tends to reduce each thread execution time but increase the total execution time due to parallelization overhead.

By carefully selecting a "parallelization option" for each task, i.e., the chosen number of threads each task is parallelized into, we can maximize the system schedu-

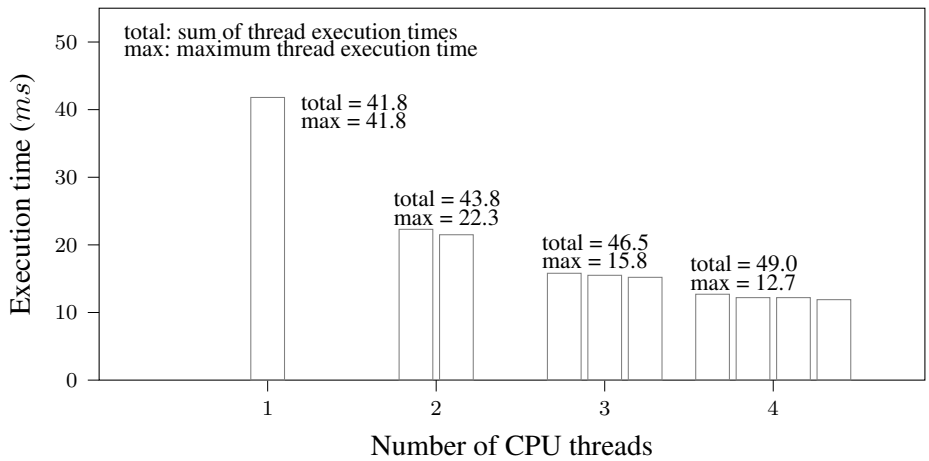


Figure 1.1: Measured thread execution times of Euclidean algorithm

lability while satisfying real-time constraints. Because of this benefit, parallelization freedom has drawn recent attention. Recent works [34, 33] address the parallelization option assignment problem targeting fluid scheduling such as PD^2 [53] and LLREF [18]. However, fluid scheduling is hardly practical due to a massive number of scheduling invocations and context switches. For more widely used global EDF scheduling, G-EDF for short, the concept of parallelization freedom still has not brought much attention. Thus, this dissertation proposes a way of optimally assigning parallelization options to real-time tasks for G-EDF on a multi-core system. Moreover, we aim to propose a polynomial-time algorithm that can be used for online situations where tasks dynamically join and leave.

This dissertation first focuses on the traditional task model, i.e., multi-thread task model, and then extends to also target the multi-segment (MS) task model and then finally discusses the more general directed-acyclic-graph (DAG) task model.

1.2 Approach

To assign parallelization options to tasks, we have to check whether an assignment solution makes tasks schedulable. Therefore, the problem of the parallelization option assignment is, at least, as complex as the schedulability analysis itself. Moreover, combined with the sheer number of possible combinations of parallelization options, the problem space can grow exponentially. Considering such a high complexity nature of the problem, we take advantage of sufficient state-of-the-art schedulability analyses, i.e., BCL analysis [13] for the traditional task model, Chwa-MS [23] for the multi-segment model, and Chwa-DAG [22] for the DAG task model.

The mentioned analyses are categorized as interference-based analysis meaning, the schedulability of a task set is determined by checking whether each task can be allocated enough CPU time before its deadline, despite being maximally interfered by other co-running tasks in the system. We extend their design to correctly consider the dynamic change of the task structure under parallelization freedom.

Applying the extended analyses to their respective task model, we can observe the following trade-off relation. By increasing the parallelization option of a task: (1) for the task itself, its thread execution time is shortened, which increases its tolerance from other tasks' interference. (2) However, due to the increased number of threads and total execution time, worst-case interference to other tasks is increased. This trade-off between "tolerance" and "interference" is formally characterized as monotonic increasing functions of the parallelization option. With the above properties, we propose a uni-directional, polynomial-time algorithm that assigns parallelization options to each task: all tasks start as a single thread, then each task's parallelization options are raised until every task's tolerance is larger than its received interference

from other tasks. We prove that the above algorithm is optimal in accordance with the extended analysis if the parallelization overhead is not significantly large.

1.3 Organization

The rest of this dissertation is organized as follows.

- Chapter 2 surveys related work, including real-time workload model, real-time multi-core scheduling, and parallel computing framework.
- Chapter 3 lays out the optimal parallelization algorithm for the traditional task model. First, in Section 3.3, the BCL schedulability analysis is introduced and extended for parallelization freedom. Then in Section 3.4, the parallelization assignment algorithm is first introduced, and its optimality is formally proved. Finally, in Section 3.5, the algorithm is extensively evaluated in both simulated and implemented workload.
- Chapter 4 extends the algorithm to accommodate the multi-segment task model and the DAG task model. In Section 4.3 the Chwa-MS schedulability is extended for parallelization freedom. Then in Section 4.4 the parallelization assignment algorithm is applied. In Section 4.6 Chwa-DAG schedulability is introduced, and similarly, it is extended to target parallelization freedom. In Section 4.7 parallelization assignment algorithm for DAG is derived. Finally, in Section 4.8 and 4.9, the algorithm is evaluated extensively against real-world autonomous driving framework, i.e., AutowareAuto, to emphasize practicality.
- Chapter 5 concludes the dissertation and discusses future work.

Chapter 2
Related Work

2.1 Real-Time Multi-Core Scheduling

Real-time multi-core scheduling can largely be classified into the following two approaches: partitioned scheduling approach [25, 3, 17, 51, 9, 41] and global scheduling approach [54, 13, 31, 36, 49, 23, 48, 10, 55, 53, 18, 28, 37, 43, 42, 34], according to their consideration of the underlying multi-core computing resource.

In partitioned scheduling approaches, computing tasks are grouped into one or many tasks according to their characteristics, such as their criticality. Each group is exclusively assigned to one or many computing resources so that tasks that belong to other groups cannot access the same computing resource. This approach generally benefits from a tight estimation of the worst-case performance.

On the other hand, all computing tasks can potentially execute in any computing resource in global scheduling approaches. The tasks are managed in a single global queue and are dynamically executed in the available CPU according to the chosen scheduling decision. Global scheduling approach benefits from this centralized approach because they can utilize the system as a whole and thus is exceptionally performant in situations where the computation load fluctuates as in a sporadic server. It is known in queueing theory [52] that a single global queue results in a lower average response time compared to a queue dedicated for every computation resource.

Because of these efficient utilization characteristics, the global scheduling approach has drawn considerable attention: Global fixed-priority (global FP), arguably

the most simple global scheduling approach, assigns static priority to each task. Global earliest deadline first (global EDF) assigns priority to subsequent jobs according to their absolute deadline. This enables the scheduler to be versatile in dynamic load situations and is more practically applicable for real-world use-cases. Another branch of global scheduling is fluid scheduling [10, 55, 53, 1, 18, 28, 37, 43, 42, 34, 33], where the execution of the task is split into fine-grained chunks, and priority is assigned individually to each chunk. However, fluid schedulers suffer from intensive scheduler invocations and context-switching overhead.

2.2 Real-Time Multi-Core Task Model

As multi-core processor computing became widely available, many real-time task models were developed. The previously sequential, single-core task model is first extended to parallel models, e.g., multi-thread task model. In the multi-thread task model, each task can be parallelized into a desirable number of threads that share the same real-time constraints such as deadline and minimum separation time. This model assumes that for each jobs, i.e., an instance of a thread, the number of threads is fixed from its beginning to its end.

Then the "fork-join model" is proposed. Here, tasks can alternate between multi-threaded and single-threaded "segments". This can be thought of as several intermittent synchronized execution during parallel execution, or in other words, threads occasionally join to spend time synchronizing, do sequential work and then fork again to run fast in parallel. This model is further generalized into the "multi-segment(MS) task model", where the alternating single/multi-threaded restriction is revoked so that all segments can run in any number of threads. Therefore, a multi-segment task can

be thought of as a series of parallel executions that must run in sequence.

Finally, the MS model is further generalized to become the "directed-acyclic-graph(DAG)" task model. A DAG task consists of nodes and edges, where a node represents an execution requirement and edges portray the precedence relation of the nodes. Any nodes can execute in a desirable number of threads as long as their precedence conditions are met. The DAG model can represent many real-world workloads. For example, the computation flow of AutowareAuto, a leading open-source autonomous driving platform, conforms to a DAG structure.

However, the listed task models assume a preparallelized execution model with a predefined number of threads and hence do not consider parallelization freedom. In the parallelization freedom model, parallelization can be decided per segment/node and can even change during runtime. Because of this flexible characteristic, parallelization freedom has drawn recent attention. For example, [34, 33] address the fluid scheduling algorithms and was more recently adapted to the widely used global FP [44] and global EDF [19, 20, 21] schedulers.

2.3 Real-Time Multi-Core Schedulability Analysis

Given a set of real-time tasks, a multi-core computing resource, and a scheduling policy, a real-time multi-core schedulability analysis aims to determine, without actually executing, whether each real-time task can be assured their deadline. Real-time multi-core schedulability analyses can be categorized according to their scope: sufficient analysis, necessary analysis, and sufficient and necessary analysis.

A sufficient analysis determines whether a task is schedulable in any possible runtime release scenario. The procedure involves assuming a worst-case release of the

tasks and checking whether the system is schedulable even in such conditions. Such a worst-case condition may not actually occur in reality. Hence sufficient analysis can be pessimistic for determining schedulability. However, they can guarantee proof of fully-safe execution of all tasks, and in many cases in polynomial time. Therefore, many works address this domain, with their priority being reducing the pessimism.

There are numerous available schedulability analyses for G-EDF [13, 29, 6, 5, 14, 8, 12, 11, 57, 7], that can be extended to multi-thread task model. In addition, schedulability analysis for other parallel task models are also available: fork-join task model [2, 25], multi-segment task model [23, 40, 31], and the DAG task model [38].

On the other hand, necessary analyses [6] cannot provide such a guarantee but facilitate mainly as a quick online filter for incoming tasks. [6] checks whether the total sum of utilization does not exceed the number of processors, and a similar variant is used in Linux kernel for SCHED_DEADLINE task admission test [27].

Sufficient and necessary test [7] tries to give accurate scheduling decisions. However, exact analyses narrow down the target scope because of the exponentially vast task arrival pattern, such as assuming integer time and periodic release pattern. The exact analyses operate exponentially and hence cannot be incorporated in online situations where tasks dynamically join and leave.

Chapter 3
Optimal Parallelization of Multi-Thread Tasks

3.1 Introduction

Parallelization freedom allows us to parallelize a real-time task into many different versions. Depending on the degree of parallelization, its thread execution times significantly vary, that is, more parallelization tends to reduce each thread execution time but increases the total execution time due to parallelization overhead. By carefully selecting a "parallelization option" for each task, i.e., the chosen number of threads each task is parallelized into, we can maximize the system schedulability while satisfying real-time constraints.

We first extend the BCL sufficient schedulability analyses to consider parallelization freedom correctly. Then, applying the extended analysis, we can observe the following trade-off relation.

By increasing the parallelization option of a task: (1) for the task itself, its thread execution time becomes shorter, and hence its tolerance for other tasks' interference gets larger. (2) However, due to the increased number of threads and total execution time, worst-case interference to other tasks is increased. We formally model such trade-off relation between "tolerance" and "interference" as monotonic increasing functions of the parallelization option.

Leveraging those monotonic increasing properties, we propose a polynomial-time algorithm that starts from no-parallelization for all the tasks and iteratively raises each task's parallelization option until the tolerance of every task becomes larger than

the interference from other tasks. We formally prove that the proposed algorithm is optimal in terms of the extended analysis when the parallelization overhead is not significantly large.

The effectiveness of the proposed algorithm is validated through both simulation and actual implementation. Our extensive experimental study says that the proposed algorithm can improve the schedulability up to 60%.

3.2 Problem Description

We consider a system with m identical CPU cores and n sporadic tasks scheduled by global EDF. Each sporadic task is denoted by $\tau_k (1 \leq k \leq n)$ and the set of n sporadic tasks is denoted by Γ as follows:

$$\Gamma = \{\tau_1, \tau_2, \cdots, \tau_n\}.$$

Each task τ_k is represented as a 3-tuple, i.e., $\tau_k = (E_k, D_k, T_k)$ where T_k is the minimum inter-release time, D_k is the relative deadline, and E_k is the thread execution time table according to the parallelization option O_k as shown in Fig. 3.1. For a task τ_k, we can choose a parallelization option O_k. The parallelization option O_k ranges from 1 to $O^{\max} = m$ meaning that we have options of parallelizing τ_k into a single thread, i.e., $O_k = 1$, two threads, i.e., $O_k = 2$, and up to $O^{\max}$ threads, i.e., $O_k = O^{\max}$. If we choose the option O_k for a task τ_k, it means that the task is parallelized into O_k sibling threads that share the same release times and deadlines. Thus, we denote such a parallelized task by $\tau_k(O_k)$, which is the set of O_k sibling

threads, i.e.,

$$\tau_k(O_k) = \{\tau_k^1(O_k), \tau_k^2(O_k), \cdots, \tau_k^{O_k}(O_k)\},$$

where $\tau_k^l(O_k)(1 \leq l \leq O_k)$ is the l-th sibling thread. The l-th sibling thread $\tau_k^l(O_k)$ has the worst-case execution time (WCET) $e_k^l(O_k)$ as shown in the E_k table of Fig. 3.1. Note that the above definition gets notation-heavy, and we may omit the subscript or the parentheses when no ambiguity arises.

Without loss of generality, we sort the O_k threads of $\tau_k(O_k)$ in the descending order of $e_k^l(O_k)$ and hence $\tau_k^1(O_k)$ has the largest execution time, i.e.,

$$\max_{\tau_k^l(O_k) \in \tau_k(O_k)} e_k^l(O_k) = e_k^1(O_k).$$

Also, the total computation amount or simply "computation amount" of O_k sibling threads, i.e., $\sum_{l=1}^{O_k} e_k^l(O_k)$, is denoted by $C_k(O_k)$:

$$C_k(O_k) = \sum_{l=1}^{O_k} e_k^l(O_k).$$

In general, a larger parallelization option $O^{'}(> O)$ makes each of the thread execution time smaller, i.e., $e_k^l(O^{'}) < e_k^l(O)$, but makes the computation amount larger, i.e., $C_k(O^{'}) > C_k(O)$, due to parallelization overhead.

For scheduling these n sporadic tasks on m identical CPU cores, we use global EDF. Similar to other researches on global EDF [13, 29, 6, 5, 14, 8, 12, 11, 57, 7], we assume that preemptions, migrations, and scheduling costs are either negligible or included in the WCET. Also, to focus on the scheduling aspect, we assume memory and bandwidth partitioning such as PALLOC [58] and MemGuard [59] to avoid inter-

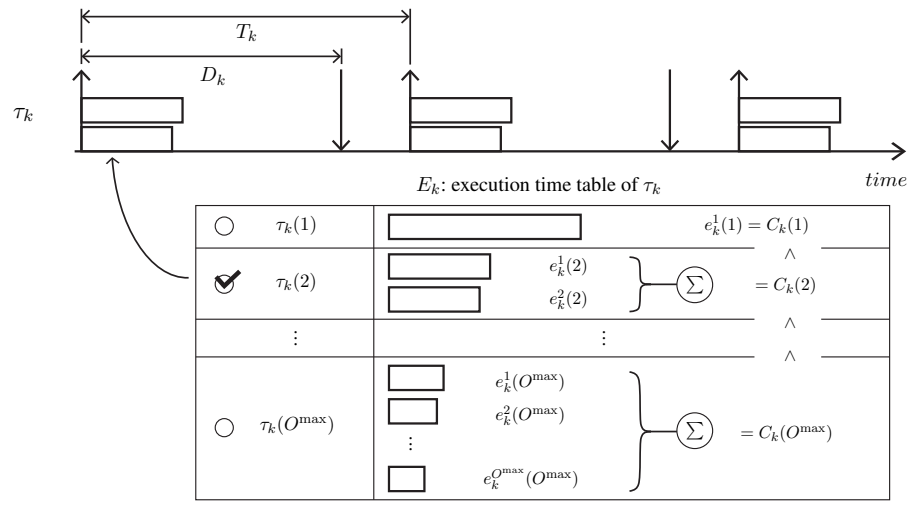


Figure 3.1: Sporadic real-time task with parallelization freedom

core interferences.

Under these assumptions, our problem is formally defined as follows:

Problem Definition: For each task τ_k in the given task set $\Gamma = \{\tau_1, \tau_2, \cdots, \tau_n\}$, our problem is to find its parallelization option O_k, such that all the sibling threads of all the tasks in Γ can be scheduled meeting their deadlines using G-EDF on m identical CPU cores.

3.3 Extension of BCL Schedulability Analysis

To address our problem of the parallelization option assignment, we need a schedulability analysis. There have been various works on G-EDF schedulability analysis for sporadic task systems including exact analysis methods [7] and sufficient analysis methods [13, 29, 6, 5, 14, 8, 12, 11, 57]. Although the exact analysis methods can be used to create the ground truth for the parallelization option assignment, they require

exponential times and hence cannot be used for an online parallelization option assignment. Therefore, we use a sufficient analysis, namely BCL [13] analysis, as the basis for our online parallelization option assignment. Accordingly, our goal is to find an optimal algorithm for the parallelization option assignment in terms of the BCL schedulability analysis.

In this section, we first overview the BCL schedulability analysis and then show how it can be extended to determine the schedulability of tasks with parallelization freedom.

3.3.1 Overview of BCL Schedulability Analysis

The BCL schedulability analysis belongs to a category of sufficient global EDF schedulability analyses based on the concept of interference using the following definitions made by Bertogna et al. [13]:

Within interval $[a, b)$, regarding an interfered task τ_k and an interfering task τ_i:

1. $I_{\tau_k}(a, b)$: Cumulative length of all intervals in which τ_k is ready to execute but it cannot execute due to higher priority jobs. In other words, $I_{\tau_k}(a, b)$ involves all the intervals in which τ_k is ready but all m CPU cores are occupied by other higher priority jobs (See Fig. 3.2).

2. $I_{\tau_i,\tau_k}(a, b)$: Cumulative length of all intervals in which τ_k is ready to execute but it cannot execute due to τ_i's jobs. In other words, out of the cumulative intervals in $I_{\tau_k}(a, b)$, $I_{\tau_i,\tau_k}(a, b)$ involves only intervals where τ_i's job is one of the higher priority jobs occupying the m CPU cores (See Fig. 3.2).

From these definitions, for single-thread tasks, $I_{\tau_k}(a, b)$ is the summation of

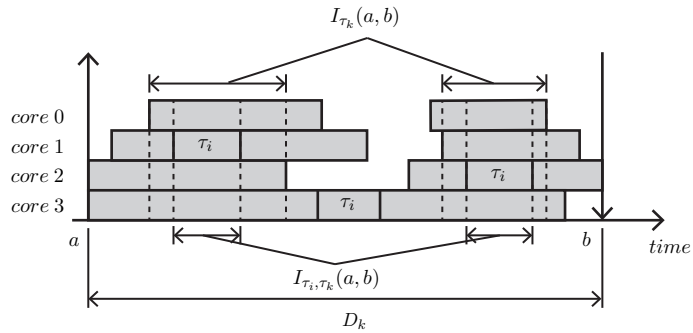


Figure 3.2: Interference of τ_k within interval $[a, b)$

$I_{\tau_i,\tau_k}(a, b)$ of all the interfering tasks τ_is ($\tau_i \in \Gamma, \tau_i \neq \tau_k$) divided by m, i.e.,

$$I_{\tau_k}(a, b) = \frac{1}{m} \sum_{\tau_i \in \Gamma(\tau_i \neq \tau_k)} I_{\tau_i,\tau_k}(a, b). \tag{3.1}$$

Using this, a schedulability condition can be derived for the j-th job of τ_k, having the release time of r_{kj} and the absolute deadline of $d_{kj} = r_{kj} + D_k$. The job is schedulable when it can execute for its WCET denoted by e_k inside its interval $[r_{kj}, d_{kj})$. In other words, for the job not to miss the deadline, the interference must be smaller than or equal to $D_k - e_k$. Combined with Eq. (3.1), this schedulability condition can be formulated as follows, without loss of generality, $a = r_{kj}$ and $b = d_{kj}$:

$$I_{\tau_k}(a, b) = \frac{1}{m} \sum_{\tau_i \in \Gamma(\tau_i \neq \tau_k)} I_{\tau_i,\tau_k}(a, b) \leq D_k - e_k. \tag{3.2}$$

However, Eq. (3.2) cannot be used as-is, since due to the vast number of possible execution scenarios of the sporadic tasks, calculating $I_{\tau_i,\tau_k}(a, b)$ value becomes intractable.

Instead, BCL and most other analyses use an upper bound of the interference

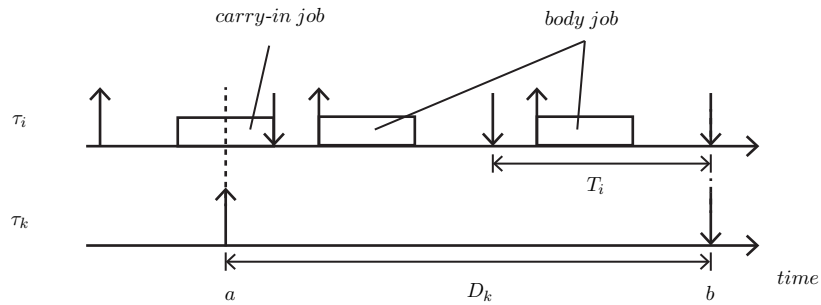


Figure 3.3: Worst-case release pattern of τ_i in $[a, b) = [r_{kj}, d_{kj})$

$I_{\tau_i,\tau_k}(a, b)$. Such an upper bound is given by the total workload of the interfering task τ_i within the interval $[a, b) = [r_{kj}, d_{kj})$. Fig. 3.3 shows the worst-case release pattern of the interfering task τ_i that gives the largest amount of workload into the interval of a τ_k's job in global EDF scheduling. It is when τ_i's last job's deadline coincides with the deadline of τ_k's job and τ_i's previous jobs are released most frequently, that is, with its minimum inter-release time T_i. Considering this worst-case release pattern, the worst-case workload of τ_i included in the interval of a τ_k's job, which is denoted by W_{τ_i,τ_k}, is given as follows:

$$W_{\tau_i,\tau_k} = \left\lfloor \frac{D_k}{T_i} \right\rfloor e_i + \min(e_i, D_k \bmod T_i) = W^{BD}_{\tau_i,\tau_k} + W^{CI}_{\tau_i,\tau_k}. \tag{3.3}$$

In this equation, the first term, i.e., $\left\lfloor \frac{D_k}{T_i} \right\rfloor e_i$, is the workload by *body jobs* that are released and have deadlines within the interval of a τ_k's job, and is notated as $W^{BD}_{\tau_i,\tau_k}$. The second term, i.e., $\min(e_i, D_k \bmod T_i)$, is the workload by a *carry-in job* that are released before the interval but has its deadline within the interval, and is notated as $W^{CI}_{\tau_i,\tau_k}$. This worst-case workload W_{τ_i,τ_k}, from now on, is simply called "workload from τ_i to τ_k".

Since the interference $I_{\tau_i,\tau_k}(a,b)$ cannot be greater than the workload W_{τ_i,τ_k}, the schedulability condition in Eq. (3.2) can be transformed into the following sufficient schedulability condition:

$$\begin{aligned} I_{\tau_k}(a,b) &= \frac{1}{m} \sum_{\tau_i \in \Gamma(\tau_i \neq \tau_k)} I_{\tau_i,\tau_k}(a,b) \\ &\leq \frac{1}{m} \sum_{\tau_i \in \Gamma(\tau_i \neq \tau_k)} W_{\tau_i,\tau_k} \ \leq D_k - e_k. \end{aligned} \tag{3.4}$$

This sufficient schedulability condition gets more tightened in the BCL schedulability analysis with the introduction of "bounded workload", denoted as $\overline{W}_{\tau_i,\tau_k}$. $\overline{W}_{\tau_i,\tau_k}$ is a upper bound of the interference $I_{\tau_i,\tau_k}(a,b)$. This comes from the following consideration: For a job of τ_k to be schedulable, it cannot be interfered more than $D_k - e_k$ by τ_i. Thus, we can limit the workload W_{τ_i,τ_k} by $D_k - e_k$ when considering the interference from τ_i for the schedulability of a τ_k's job. This results in the following bounded workload:

$$\overline{W}_{\tau_i,\tau_k} = \min(W_{\tau_i,\tau_k}, D_k - e_k). \tag{3.5}$$

We call $\overline{W}_{\tau_i,\tau_k}$ "bounded workload from τ_i to τ_k" or "interference from τ_i to τ_k", whenever there is no ambiguity.

By replacing W_{τ_i,τ_k} in Eq. (3.4) with a tighter upper bound, i.e., the bounded workload $\overline{W}_{\tau_i,\tau_k}$ in Eq. (3.5) and moving m to the right-hand side, we finally have

the following BCL schedulability condition for each task τ_k:

$$\sum_{\tau_i \in \Gamma(\tau_i \neq \tau_k)} \overline{W}_{\tau_i,\tau_k} = \sum_{\tau_i \in \Gamma(\tau_i \neq \tau_k)} \min(W_{\tau_i,\tau_k}, D_k - e_k) \ \leq \ m(D_k - e_k). \quad (3.6)$$

In addition, the BCL schedulability condition enforces that at most $m-1$ interfering tasks τ_is have workloads W_{τ_i,τ_k}s exceeding $D_k - e_k$. This is because if m or more τ_is have workloads exceeding $D_k - e_k$, there can be a case where τ_k cannot be guaranteed e_k from its release time to its absolute deadline[13].

From now on, by the "schedulability" of a task τ_k, we mean that τ_k meets this BCL sufficient schedulability condition. Also, when we say that the given task set Γ is schedulable, it means that the BCL condition holds for every task $\tau_k \in \Gamma$.

3.3.2 Properties of Parallelization Freedom

In order to apply the schedulability condition of Eq. (3.6) to a task τ_k with the parallelization freedom, the task τ_k using the parallelization option O_k can simply be considered as a set of O_k individual tasks, i.e., $\tau_k(O_k) = \{\tau_k^1(O_k), \tau_k^2(O_k), \cdots, \tau_k^{O_k}(O_k)\}$, having the same minimum inter-release time T_k and relative deadline D_k, but different execution times, i.e., $e_k^1(O_k), e_k^2(O_k), \cdots, e_k^{O_k}(O_k)$.

By this extension, for a given task set $\Gamma = \{\tau_1, \tau_2, \cdots, \tau_n\}$, when we choose a parallelization option combination $\mathbb{O} = (O_1, O_2, \cdots, O_n)$, we can transform Γ into an extended task set $\Gamma(\mathbb{O}) = \tau_1(O_1) \cup \tau_2(O_2) \cup \cdots \cup \tau_n(O_n)$ containing a total of $O_1 + O_2 + ... + O_n$ number of tasks. Then, we can use Eq. (3.6) on the extended task set $\Gamma(\mathbb{O})$ to determine the schedulability. For this reason, from now on, in the

situations where the threads are considered as individual tasks, we may conveniently refer to them as tasks.

However, note that sibling threads have a clear difference from non-sibling threads. That is, a pair of sibling threads, say $\tau_k^p(O_k)$ and $\tau_k^q(O_k)$, always have the same release times and the same absolute deadlines as shown in Fig. 3.1, which is not the case for non-sibling threads belonging to different tasks. Therefore, only a single body job and no carry-in job of $\tau_k^p(O_k)$ can interfere with $\tau_k^q(O_k)$ and vice versa even in the worst-case. This difference of sibling threads yields the following three properties, which help find the optimal parallelization options.

Property 1. Out of O_k parallelized threads of $\tau_k(O_k)$, the first thread $\tau_k^1(O_k)$, which has the largest WCET can be used as a representative thread for the schedulability of all other sibling threads, because if $\tau_k^1(O_k)$ is schedulable, all other siblings threads are also schedulable. The following lemma formally states this.

Lemma 1. If $\tau_k^1(O_k)$ meets the schedulability condition Eq. (3.6), all its sibling threads $\{\tau_k^2(O_k), \cdots, \tau_k^{O_k}(O_k)\}$ also meet the condition.

Proof. Let τ_k^* be a sibling thread of τ_k^1 whose schedulability we want to prove. Also, a task that is neither τ_k^1 nor τ_k^* is denoted by τ_x and the set of such tasks is denoted by Γ_x. Using these notations, the fact that τ_k^1 meets Eq. (3.6) is rephrased as follows:

$$\sum_{\tau_x \in \Gamma_x} \overline{W}_{\tau_x, \tau_k^1} + \overline{W}_{\tau_k^*, \tau_k^1} \leq m(D_k - e_k^1). \tag{3.7}$$

Also, the schedulability condition, i.e., Eq. (3.6), for τ_k^* is rephrased as follows:

$$\sum_{\tau_x \in \Gamma_x} \overline{W}_{\tau_x, \tau_k^*} + \overline{W}_{\tau_k^1, \tau_k^*} \leq m(D_k - e_k^*). \tag{3.8}$$

We can now prove the lemma by deriving Eq. (3.8) from Eq. (3.7). Let us first consider the first term of the left-hand sides of Eq. (3.7) and Eq. (3.8). For a task $\tau_x \in \Gamma_x$, let us compare its bounded workload to τ_k^1, i.e., $\overline{W}_{\tau_x, \tau_k^1}$ in Eq. (3.7) and that to τ_k^*, i.e., $\overline{W}_{\tau_x, \tau_k^*}$ in Eq. (3.8), which are:

$$\overline{W}_{\tau_{,x}\tau_k^1} = \min(W_{\tau_x, \tau_k^1}, D_k - e_k^1) \text{ and}$$
$$\overline{W}_{\tau_x, \tau_k^*} = \min(W_{\tau_x, \tau_k^*}, D_k - e_k^*),$$

respectively. Since the sibling threads τ_k^1 and τ_k^* have the same release time and deadline, τ_x's workloads to them are the same, i.e., $W_{\tau_x, \tau_k^1} = W_{\tau_x, \tau_k^*}$. However, if W_{τ_x, τ_k^1} exceeds $D_k - e_k^1$, then $\overline{W}_{\tau_x, \tau_k^1} = D_k - e_k^1$. In such case, $\overline{W}_{\tau_x, \tau_k^*}$ can be greater than $\overline{W}_{\tau_x, \tau_k^1}$ due to the larger bound $D_k - e_k^*$. Nevertheless, the difference is at most $e_k^1 - e_k^*$. Therefore,

$$\overline{W}_{\tau_x, \tau_k^1} + (e_k^1 - e_k^*) \geq \overline{W}_{\tau_x, \tau_k^*}.$$

However, the number of τ_xs whose W_{τ_x, τ_k^1} exceeds $D_k - e_k^1$ is at most $m-1$ according to the BCL schedulability condition for τ_k^1. Thus, overall, the following holds for the first term of Eq. (3.7) and Eq. (3.8):

$$\sum_{\tau_x \in \Gamma_x} \overline{W}_{\tau_x, \tau_k^1} + (m-1)(e_k^1 - e_k^*) \geq \sum_{\tau_x \in \Gamma_x} \overline{W}_{\tau_x, \tau_k^*}.$$

Second, let us consider the second term of the left-hand sides of Eq. (3.7) and

Eq. (3.8), which are the bounded workload from τ_k^* to τ_k^1 and vice versa, respectively. They are:

$$\overline{W}_{\tau_k^*,\tau_k^1} = \min(W_{\tau_k^*,\tau_k^1}, D_k - e_k^1) \text{ and}$$

$$\overline{W}_{\tau_k^1,\tau_k^*} = \min(W_{\tau_k^1,\tau_k^*}, D_k - e_k^*),$$

Since τ_k^* and τ_k^1 are sibling threads each other, only one body job of one thread is included in the workload to the other thread, that is, $W_{\tau_k^*,\tau_k^1} = e_k^*$ and $W_{\tau_k^1,\tau_k^*} = e_k^1$. If the workloads do not exceed the bounds, i.e., $W_{\tau_k^*,\tau_k^1} = e_k^* \leq D_k - e_k^1 \Leftrightarrow W_{\tau_k^1,\tau_k^*} = e_k^1 \leq D_k - e_k^*$, the bounded workloads are the same as the workloads, i.e., $\overline{W}_{\tau_k^*,\tau_k^1} = e_k^*$ and $\overline{W}_{\tau_k^1,\tau_k^*} = e_k^1$. Otherwise, they are $\overline{W}_{\tau_k^*,\tau_k^1} = D_k - e_k^1$ and $\overline{W}_{\tau_k^1,\tau_k^*} = D_k - e_k^*$. In both cases, the following holds for the second term of Eq. (3.7) and Eq. (3.8):

$$\overline{W}_{\tau_k^*,\tau_k^1} + (e_k^1 - e_k^*) = \overline{W}_{\tau_k^1,\tau_k^*}.$$

Combining the first term and the second term, the left-hand side of Eq. (3.8) is at most $m(e_k^1 - e_k^*)$ larger than that of Eq. (3.7). On the other hand, the right-hand side of Eq. (3.8) is $m(e_k^1 - e_k^*)$ larger than that of Eq. (3.7). Therefore, when Eq. (3.7) is true, Eq. (3.8) is also true. The lemma follows. □

Thanks to Lemma 1, for each task $\tau_k \in \Gamma$, we only need to check whether its first thread $\tau_k^1(O_k)$ meets the schedulability condition, i.e., Eq. (3.6). Regarding $\tau_k^1(O_k)$, Eq. (3.6) can be rewritten as follows, by separating the bounded workloads from its sibling threads and all others:

$$\sum_{\tau_k^l(O_k) \in \tau_k(O_k), l \neq 1} \overline{W}_{\tau_k^l(O_k),\tau_k^1(O_k)} + \sum_{\tau \notin \tau_k(O_k)} \overline{W}_{\tau,\tau_k^1(O_k)} \leq m(D_k - e_k^1(O_k)).$$

In the left-hand side of this equation, the first term is the bounded workload from the sibling threads to $\tau_k^1(O_k)$ and the second term is the bounded workload from the non-sibling threads, i.e., threads of other tasks, to $\tau_k^1(O_k)$. By moving the first term to the right-hand side, the equation can be rephrased as follows:

$$\sum_{\tau \notin \tau_k(O_k)} \overline{W}_{\tau,\tau_k^1(O_k)} \leq m(D_k - e_k^1(O_k)) \; - \sum_{\tau_k^l(O_k) \in \tau_k(O_k), l \neq 1} \overline{W}_{\tau_k^l(O_k),\tau_k^1(O_k)},$$

meaning that the maximum amount of the bounded workload or interference from other tasks' threads that $\tau_k^1(O_k)$ can tolerate for its schedulability is the right-hand side. Since the schedulability of $\tau_k^1(O_k)$ implies the schedulability of all its sibling threads due to Lemma 1, the right-hand side can be interpreted as the "tolerance" of τ_k for the interference from other tasks, when O_k is chosen for τ_k. Thus, we define the tolerance $tol(\tau_k(O_k))$ by the right-hand side of the above equation, i.e.,

$$tol(\tau_k(O_k)) = \; m(D_k - e_k^1(O_k)) \; - \sum_{\tau_k^l(O_k) \in \tau_k(O_k), l \neq 1} \overline{W}_{\tau_k^l(O_k),\tau_k^1(O_k)}. \tag{3.9}$$

Recall that the bounded workload from a sibling thread $\tau_k^l(O_k)$ to $\tau_k^1(O_k)$ is the minimum of the workload $W_{\tau_k^l(O_k),\tau_k^1(O_k)}$ and the bound $D_k - e_k^1(O_k)$, i.e.,

$$\overline{W}_{\tau_k^l(O_k),\tau_k^1(O_k)} = \min(W_{\tau_k^l(O_k),\tau_k^1(O_k)}, D_k - e_k^1(O_k)).$$

Also, the workload $W_{\tau_k^l(O_k),\tau_k^1(O_k)}$ from a sibling thread $\tau_k^l(O_k)$ is a single body job's execution time $e_k^l(O_k)$. Putting this into Eq. (3.9), the tolerance $tol(\tau_k(O_k))$ is finally

given as a function of the parallelization option O_k as follows:

$$tol(\tau_k(O_k)) = m(D_k - e_k^1(O_k)) - \sum_{\tau_k^l(O_k) \in \tau_k(O_k), l \neq 1} \min(e_k^l(O_k), D_k - e_k^1(O_k)). \tag{3.10}$$

Regarding this tolerance, the following property greatly helps us find the optimal parallelization options by a uni-directional search rather than a back-and-forth combinatorial search.

Property 2. This tolerance of τ_k is a monotonic increasing function of O_k, meaning that, with a larger parallelization option O_k, the task τ_k can tolerate a larger amount of interference from other tasks. The following lemma and corollary formally state this.

Lemma 2. The tolerance $tol(\tau_k(O_k))$ in Eq. (3.10) is a monotonic increasing function of O_k, i.e., for every $O_k < O^{\max}$, $tol(\tau_k(O_k)) < tol(\tau_k(O_k + 1))$ under ideal parallelization with zero parallelization overhead, i.e., $C_k(O_k) = C_k(O_k + 1)$.

Proof. When the parallelization option increases from O_k to $O_k + 1$, the first term of the tolerance in Eq. (3.10) increases from $m(D_k - e_k^1(O_k))$ to $m(D_k - e_k^1(O_k + 1))$. The amount of this increase is $m(e_k^1(O_k) - e_k^1(O_k + 1))$, which is always positive because $e_k^1(O_k) > e_k^1(O_k + 1)$.

Now, let us consider the change of the second term, which is the bounded workload from sibling threads. For this, let us use Fig. 3.4 where the above diagram (shaded area) depicts the second term for O_k while the below diagram (shaded area) depicts that for $O_k + 1$. As we can see in the figure, the second term may increase depending on the amount of siblings' workload increase, i.e., from $C_k(O_k) - e_k^1(O_k)$

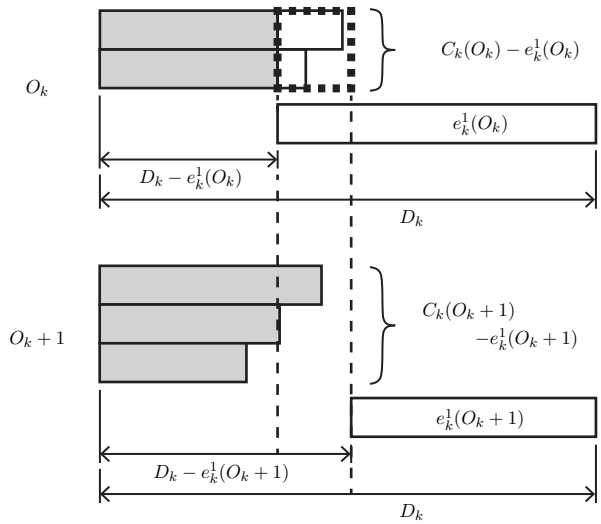


Figure 3.4: Change of bounded workload for $O_k \to O_k + 1$

to $C_k(O_k + 1) - e_k^1(O_k + 1)$. Let us conservatively consider the amount of this increase, i.e., $C_k(O_k + 1) - C_k(O_k) + \big(e_k^1(O_k) - e_k^1(O_k + 1)\big)$, as the increase of the bounded workload.

The second term also may increase due to the increase of the bound from $D_k - e_k^1(O_k)$ to $D_k - e_k^1(O_k + 1)$. See the part surrounded by thick dotted line in the above diagram of Fig. 3.4, which is not counted as the bounded workload due to the small bound $D_k - e_k^1(O_k)$. In the below diagram, however, such a part can be included in the bounded workload due to the larger bound $D_k - e_k^1(O_k + 1)$. Even if such an increase happens for all $O_k - 1$ sibling threads, the total increase cannot be greater than $(O_k - 1)\big(e_k^1(O_k) - e_k^1(O_k + 1)\big)$, because such an increase for each sibling thread is limited by the difference of the two bounds, i.e., $D_k - e_k^1(O_k + 1) - (D_k - e_k^1(O_k)) = e_k^1(O_k) - e_k^1(O_k + 1)$. This amount of increase, i.e., $(O_k - 1)\big(e_k^1(O_k) - e_k^1(O_k + 1)\big)$, is also conservatively considered as the increase of the bounded workload. Summing

up these two increases, the increase of the second term cannot be larger than:

$$C_k(O_k+1) - C_k(O_k) + \big(e_k^1(O_k) - e_k^1(O_k+1)\big) + (O_k-1)\big(e_k^1(O_k) - e_k^1(O_k+1)\big)$$
$$= C_k(O_k+1) - C_k(O_k) + O_k(e_k^1(O_k) - e_k^1(O_k+1))$$

Under ideal parallelization, i.e., when $C_k(O_k+1) - C_k(O_k) = 0$, the increase of the second term is bounded by $O_k\big(e_k^1(O_k) - e_k^1(O_k+1)\big)$.

Since the increase of the first term $m(e_k^1(O_k) - e_k^1(O_k+1))$ is always greater than the increase of the second term $O_k(e_k^1(O_k) - e_k^1(O_k+1))$ where $O_k < O^{\max} = m$, the tolerance monotonically increases: $tol(\tau_k(O_k)) < tol(\tau_k(O_k+1))$. □

This lemma still holds when the parallelization overhead is not significantly large. In order to formally state this, let us define the parallelization overhead $\alpha(O_k, O_k+1)$ as the total computation amount increase, i.e., $C_k(O_k+1) - C_k(O_k)$, for unit reduction of the first thread's execution time, that is,

$$\alpha(O_k, O_k+1) = \frac{C_k(O_k+1) - C_k(O_k)}{e_k^1(O_k) - e_k^1(O_k+1)}. \tag{3.11}$$

Corollary 1. If $\alpha(O_k, O_k+1) < m - O_k$ where $O_k < O^{\max} = m$, Lemma 2 still holds.

Proof. As stated in the proof of Lemma 2, the increase amount of the second term in Eq. (3.10) cannot be larger than $C_k(O_k+1) - C_k(O_k) + O_k\big(e_k^1(O_k) - e_k^1(O_k+1)\big)$. Using Eq. (3.11), this bound for the second term increase is rewritten as:

$$\big(\alpha(O_k, O_k+1) + O_k\big)\big(e_k^1(O_k) - e_k^1(O_k+1)\big).$$

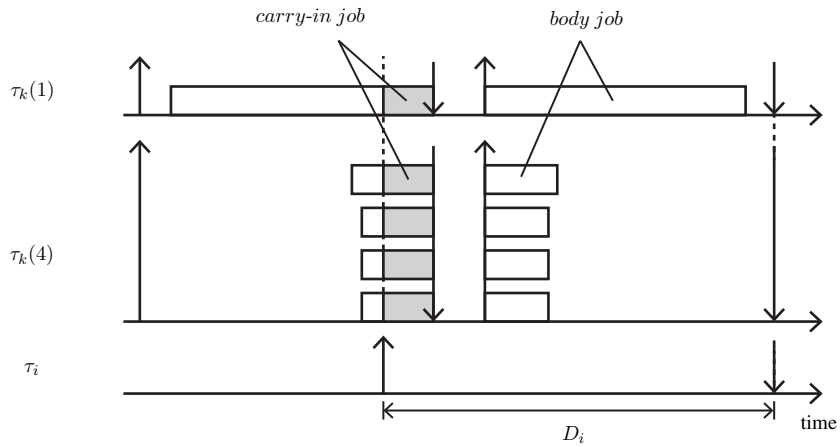


Figure 3.5: Increase of τ_k's workload when increasing O_k

Since the increase of the first term in Eq. (3.10) is $m\big(e_k^1(O_k) - e_k^1(O_k+1)\big)$, if $\alpha(O_k, O_k+1) + O_k < m$, the second term increase is smaller than the first term increase. In other words, if $\alpha(O_k, O_k+1) < m - O_k$, the tolerance increases, i.e., $tol(\tau_k(O_k)) < tol(\tau_k(O_k+1))$. □

Lemma 2 and Corollary 1 say that for a task τ_k, increasing parallelization option O_k may be advantageous for its own schedulability. However, for other tasks τ_is ($\tau_i \in \Gamma, \tau_i \neq \tau_k$), increasing τ_k's parallelization option O_k may be disadvantageous. Fig. 3.5 illustrates how two different parallelization options of τ_k, i.e., $O_k = 1$ and $O_k = 4$, can give the interference to one instance interval of another task τ_i. It shows that a larger parallelization option $O_k = 4$ of τ_k may give a larger amount of interference to another task τ_i. To formally discuss this, for a parallelization option O_k chosen for τ_k, let us define its interference function given to another task τ_i as

follows:

$$int(\tau_k, \tau_i) = \sum_{l=1}^{O_k} \overline{W}_{\tau_k^l(O_k),\tau_i^1(O_i)} = \sum_{l=1}^{O_k} \min\left(W_{\tau_k^l(O_k),\tau_i^1(O_i)}, D_k - e_i^1(O_i)\right), \tag{3.12}$$

$$\begin{aligned} \text{where} \quad W_{\tau_k^l(O_k),\tau_i^1(O_i)} &= W^{BD}_{\tau_k^l(O_k),\tau_i^1(O_i)} + W^{CI}_{\tau_k^l(O_k),\tau_i^1(O_i)} \\ &= \left\lfloor \frac{D_i}{T_k} \right\rfloor e_k^l(O_k) + \min(e_k^l(O_k), D_i \bmod T_k). \end{aligned} \tag{3.13}$$

Note that this interference function $int(\tau_k, \tau_i)$ counts the sum of bounded workloads, i.e., $\sum_{l=1}^{O_k} \overline{W}_{\tau_k^l(O_k),\tau_i^1(O_i)}$, from all O_k threads of $\tau_k(O_k)$ to only the first thread $\tau_i^1(O_i)$ of $\tau_i(O_i)$. This is because the schedulability of the first thread $\tau_i^1(O_i)$ implies the schedulability of all the sibling threads of $\tau_i(O_i)$ by Lemma 1, regardless of O_i chosen for τ_i.

Also, since the interference from each thread $\tau_k^l(O_k)$ of $\tau_k(O_k)$ to $\tau_i^1(O_i)$ is upper bounded by the bounded workload $\overline{W}_{\tau_k^l(O_k),\tau_i^1(O_i)}$, the interference function $int(\tau_k, \tau_i)$ sums up each bounded workload $\overline{W}_{\tau_k^l(O_k),\tau_i^1(O_i)}$, which is the minimum of the workload $W_{\tau_k^l(O_k),\tau_i^1(O_i)}$ and the bound $D_i - e_i^1(O_i)$ as in Eq. (3.12), where the workload $W_{\tau_k^l(O_k),\tau_i^1(O_i)}$ is the sum of the body jobs' workload (the first term of Eq. (3.13)) and the carry-in job's workload (the second term of Eq. (3.13)).

Regarding this interference function, the following property also helps us find the optimal parallelization options by a uni-directional search.

Property 3. The above-defined interference from τ_k to another task τ_i is a monotonic increasing function of O_k, meaning that, if a larger parallelization option O_k is chosen

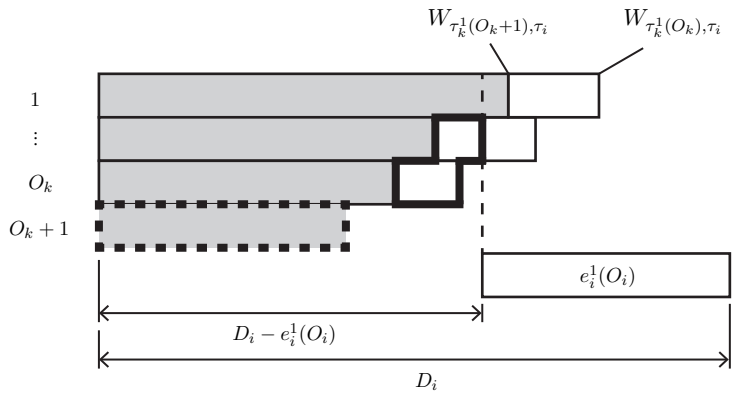


Figure 3.6: Change of bounded workload from τ_k to τ_i for $O_k \to O_k + 1$

for τ_k, it gives a larger amount of interference to another task τ_i. The following lemma formally states this.

Lemma 3. The interference $int(\tau_k, \tau_i)$ in Eq. (3.12) is a monotonic increasing function of O_k, i.e., for every $O_k < O^{\max}$, $int(\tau_k(O_k), \tau_i) < int(\tau_k(O_k + 1), \tau_i)$.

Proof. To prove this lemma, let us use Fig. 3.6. In the figure, each horizontal bar represents the workload from each thread of τ_k to the first thread of τ_i. The right-end of this horizontal bar is the workload when the option O_k is used, i.e., $W_{\tau_k^l(O_k),\tau_i^1(O_i)}$ in Eq. (3.13), while the shaded portion of the bar represents the workload when the option $(O_k + 1)$ is used, i.e., $W_{\tau_k^l(O_k+1),\tau_i^1(O_i)}$.

Note that the shaded portion is always smaller than the right-end, i.e.,

$$W_{\tau_k^l(O_k+1),\tau_i^1(O_i)} \leq W_{\tau_k^l(O_k),\tau_i^1(O_i)}, \text{because}$$

$$W^{BD}_{\tau_k^l(O_k),\tau_i^1(O_i)} \leq W^{BD}_{\tau_k^l(O_k+1),\tau_i^1(O_i)} \Leftrightarrow \left\lfloor \frac{D_i}{T_k} \right\rfloor e_k^l(O_k + 1) \leq \left\lfloor \frac{D_i}{T_k} \right\rfloor e_k^l(O_k), \text{and}$$

$$W^{CI}_{\tau_k^l(O_k),\tau_i^1(O_i)} \leq W^{CI}_{\tau_k^l(O_k+1),\tau_i^1(O_i)} \Leftrightarrow$$
$$\min(e_k^l(O_k+1), D_i \bmod T_k) \leq \min(e_k^l(O_k), D_i \bmod T_k).$$

Also, the workload from $(l+1)$-th thread is smaller than or equal to that from l-th thread since the threads are sorted in the descending order of execution times, i.e., $e_k^{l+1}(O_k) \leq e_k^l(O_k)$. Thus, the figure shows the stair shape of the right-ends and the dark portions of the horizontal bars. Note that the bottom-most horizontal bar is the workload from (O_k+1)-th thread when the option (O_k+1) is used and hence does not exist when the option O_k is used.

Now let us consider the dashed vertical line at $D_i - e_i^1(O_i)$, which limits the workload to the bounded workload. In the figure, for each horizontal bar, only the portion left to this vertical line is considered as the bounded workload. Thus, when the option O_k is used, the interference $int(\tau_k(O_k), \tau_i)$ includes the first O_k horizontal bars' portion left to the vertical line. On the other hand, when the option (O_k+1) is used, the interference $int(\tau_k(O_k+1), \tau_i)$ includes the shaded portions of the (O_k+1) horizontal bars left to the vertical line. Note that the white-area surrounded by the thick-solid line is included in $int(\tau_k(O_k), \tau_i)$ but not in $int(\tau_k(O_k+1), \tau_i)$. On the other hand, the shaded area surrounded by the thick-dotted line is included only in $int(\tau_k(O_k+1), \tau_i)$.

We now show that the white-area surrounded by the thick-solid line is smaller than the dark-area surrounded by the thick-dotted line, meaning that $int(\tau_k(O_k), \tau_i)$ $< int(\tau_k(O_k+1), \tau_i)$, as follows:

1. The total workload of $\tau_k(O_k)$, i.e.,

$$\begin{aligned}\sum_{l=1}^{O_k} W_{\tau_k^l(O_k),\tau_i^1(O_i)} &= \sum_{l=1}^{O_k} \left(W^{BD}_{\tau_k^l(O_k),\tau_i^1(O_i)} + W^{CI}_{\tau_k^l(O_k),\tau_i^1(O_i)} \right) \\ &= \sum_{l=1}^{O_k} \left(\left\lfloor \frac{D_i}{T_k} \right\rfloor e_k^l(O_k) + \min\left(e_k^l(O_k), D_i \bmod T_k\right) \right) \\ &= \left\lfloor \frac{D_i}{T_k} \right\rfloor C_k(O_k) + \sum_{l=1}^{O_k} \min\left(e_k^l(O_k), D_i \bmod T_k\right),\end{aligned}$$

   is smaller than or equal to that of $\tau_k(O_k+1)$, i.e.,

$$\begin{aligned}\sum_{l=1}^{O_k+1} W_{\tau_k^l(O_k+1),\tau_i^1(O_i)} &= \sum_{l=1}^{O_k+1} \left(W^{BD}_{\tau_k^l(O_k+1),\tau_i^1(O_i)} + W^{CI}_{\tau_k^l(O_k+1),\tau_i^1(O_i)} \right) \\ &= \sum_{l=1}^{O_k+1} \left(\left\lfloor \frac{D_i}{T_k} \right\rfloor e_k^l(O_k+1) + \min(e_k^l(O_k+1), D_i \bmod T_k) \right) \\ &= \left\lfloor \frac{D_i}{T_k} \right\rfloor C_k(O_k+1) + \sum_{l=1}^{O_k+1} \min(e_k^l(O_k+1), D_i \bmod T_k),\end{aligned}$$

   because $C_k(O_k) \le C_k(O_k+1)$, and

$$\sum_{l=1}^{O_k} \min(e_k^l(O_k), D_i \bmod T_k) \le \sum_{l=1}^{O_k+1} \min(e_k^l(O_k+1), D_i \bmod T_k).$$

2. This means that the whole white-area of the horizontal bars—which is included in $\sum_{l=1}^{O_k} W_{\tau_k^l(O_k),\tau_i^1(O_i)}$ but not in $\sum_{l=1}^{O_k+1} W_{\tau_k^l(O_k+1),\tau_i^1(O_i)}$— is smaller than or equal to the dark-area surrounded by the thick-dotted line that is newly included only in $\sum_{l=1}^{O_k+1} W_{\tau_k^l(O_k+1),\tau_i^1(O_i)}$.
3. Therefore, the white area surrounded by the thick solid line is also smaller than

the shaded area surrounded by the thick-dotted line. The lemma follows.

□

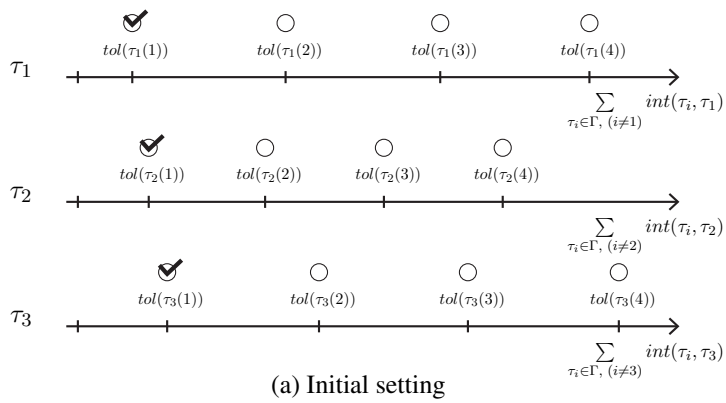
Wrapping up the above discussions, the schedulability condition of τ_k can be rewritten using the monotonic increasing functions of tolerance and interference as follows:

$$\sum_{\tau_i(O_i)\in\Gamma,\ (i\neq k)} int(\tau_i(O_i), \tau_k(O_k)) \ \leq \ tol(\tau_k(O_k)) \tag{3.14}$$

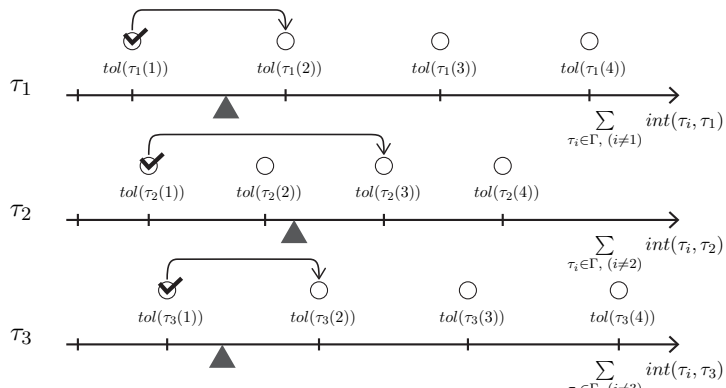
3.4 Optimal Assignment of Parallelization Options

The monotonic increasing properties of the tolerance function $tol(\tau_k)$ and the interference function $int(\tau_k, \tau_i)$ of τ_k in Lemma 2 and Lemma 3 hint that we should choose the "barely tolerable option" for τ_k in order to give the smallest possible interference to other tasks for the schedulability of all the tasks in Γ. Motivated by this, we propose an iterative algorithm, which starts from the minimum parallelization options, i.e., $\mathbb{O} = (O_1, O_2, \cdots, O_n) = (1, 1, \cdots, 1)$, and iteratively increases the options to the "barely tolerable ones" until all the tasks' tolerances become larger than the interference given by other tasks. We first intuitively explain this algorithm using an example task set consisting of three tasks, i.e., $\Gamma = \{\tau_1, \tau_2, \tau_3\}$, in Fig. 3.7.

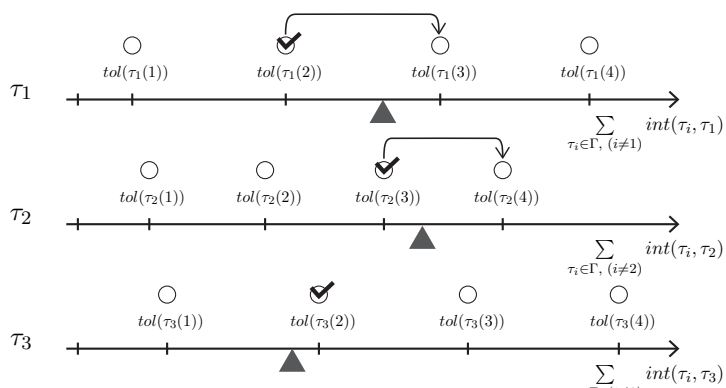
Fig. 3.7(a) shows our initial setting. On each horizontal axis representing each task τ_k, the tolerance $tol(\tau_k(O_k))$ with each O_k is marked by a circle above the axis. Notice that we initially set the first option for all the three tasks, as marked by a '✓.' Considering these initial options, in the first iteration shown in Fig. 3.7(b), we calculate the interferences given to the three tasks as marked by triangles underneath the



(a) Initial setting

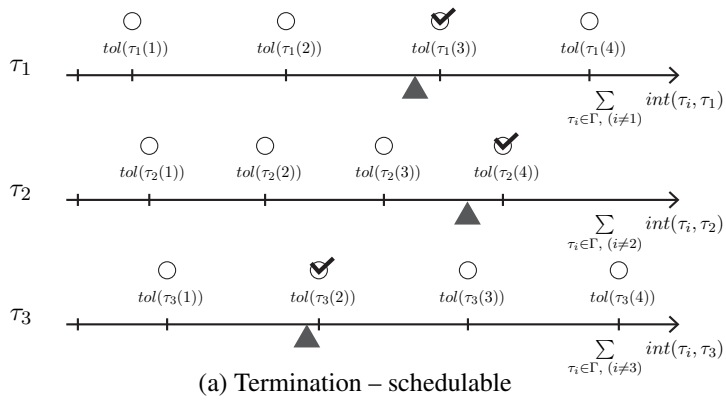


(b) First iteration

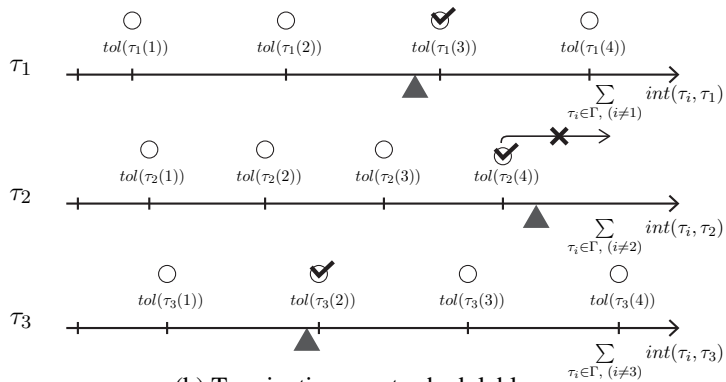


(c) Second iteration

Figure 3.7: Optimal parallelization assignment — initial setting and iterations



(a) Termination – schedulable



(b) Termination – not schedulable

Figure 3.8: Optimal parallelization assignment — termination conditions

axes. Note that the interference exceeds the current tolerance for all three tasks. Thus, as shown by the arrows, we raise the options to those whose tolerances barely exceed the interferences. Then, in the second iteration shown in Fig. 3.7(c), we recalculate the interferences with the raised options. Note that the amount of interference given to τ_1 and τ_2 still exceed the tolerances, and thus we raise their options again.

We repeat this until one of the following two conditions holds: (1) for all three tasks, tolerances are greater than or equal to the interferences—schedulable case as shown in Fig. 3.8(a) and (2) at least one task reaches its maximum parallelization

option but the interference to it still exceeds its tolerance—unschedulable case as shown in Fig. 3.8(b).

3.4.1 Optimal Parallelization Assignment Algorithm

This algorithm can be formally described as the pseudo-codes in Algorithm 1. The algorithm takes the set of tasks Γ as input and produces the parallelization option combination $\mathbb{O}$, if any, that makes the given task set schedulable as output. The **for**-loop from Line 1 to Line 5 pre-calculates the tolerance $tol(\tau_k(O_k))$ for each task $\tau_k \in \Gamma$ and for each option $O_k \leq O^{\max}$ using Eq. (3.10) as in Fig. 3.7(a). In Line 6, we create an array $\mathbb{S}$ where each element $S_k \in \mathbb{S}$ is initialized to *Unknown*, meaning the schedulability of τ_k is yet unknown. Then, Line 7 initializes the parallelization options to $\mathbb{O}^{\text{cur}} = (1, 1, ..., 1)$ and Line 8 sets the "updated" flag as true.

The **while**-loop from Line 9 to Line 24 iteratively increases the parallelization options to the barely tolerable ones until it turns out "schedulable" or "not schedulable". For this, in each iteration of the **while**-loop, Line 10 first resets the updated flag to false, meaning that we will terminate the **while**-loop if no parallelization option is updated. Then Line 11 sets the previous parallelization option combination $\mathbb{O}^{\text{pre}}$ the same as the current option combination $\mathbb{O}^{\text{cur}}$. Then, the **for**-loop from Line 12 to Line 22 tries to update each task τ_k's parallelization option O_k^{cur} to the barely tolerable ones as in Fig. 3.7(b). For this, within the **for**-loop, Line 13 calculates the total interference given to τ_k, i.e., $\sum_{\tau_i \in \Gamma(\tau_i \neq \tau_k)} int(\tau_i(O_i^{\text{pre}}), \tau_k)$, using Eq. (3.12) with the previous parallelization options.

The **while**-loop from Line 14 to Line 21 checks whether τ_k's tolerance is smaller than the interference from other tasks. If so, Line 15 increases O_k^{cur} to the next one,

and Line 16 sets the updated flag as true, indicating that some options are updated. This is repeated until the barely tolerable option is found. In the meantime, however, if O_k^{cur} becomes larger than $O^{\max}$ as in Line 17, it means that τ_k's tolerance even with $O^{\max}$ is still smaller than the interference from other tasks and hence τ_k is unschedulable. In this case, Line 18 marks S_k as false and sets $O_k^{\text{cur}} = O^{\max}$. Line 19 then breaks the **while**-loop from Line 14 to Line 21 and goes back to Line 12 to continue with unchecked tasks for finding their barely tolerable options. After checking all the tasks in the current iteration, Line 23 checks if there is a task that turns out unschedulable, and if so, it breaks the **while**-loop from Line 9 to Line 24.

When the **while**-loop from Line 9 to Line 24 terminates, the **if-else** statement from Line 25 to Line 29 returns the "not schedulable" or "schedulable" conclusion. When the given task set is schedulable, it also returns the found parallelization option combination $\mathbb{O}^{\text{cur}}$.

Algorithm 1 Optimal Parallelization Assignment (OPA):

Input: Set of tasks $\Gamma = \{\tau_1, \tau_2, \cdots, \tau_n\}$.
Output: (1) Schedulability,
(2) Parallelization option combination $\mathbb{O} = (O_1, O_2, \cdots, O_n)$
begin procedure
1. **for** $\tau_k \in \Gamma$ **do**
2. **for** $O_k = 1$ **to** $O^{\max}$ **do**
3. $tol(\tau_k(O_k)) \leftarrow$ Eq. (3.10)
4. **end for**
5. **end for**
6. initialize $\mathbb{S} \leftarrow (Unknown, Unknown, ..., Unknown)$
7. initialize $\mathbb{O}^{\text{cur}} \leftarrow (1, 1, ..., 1)$
8. initialize updated $\leftarrow$ True
9. **while** updated **do**

10. updated ← False
11. $\mathbb{O}^{\text{pre}} \leftarrow \mathbb{O}^{\text{cur}}$
12. **for** $\tau_k \in \Gamma$ **do**
13. $\sum_{\tau_i \in \Gamma,\ (i \neq k)} int(\tau_i(O_i^{\text{pre}}), \tau_k) \leftarrow$ Eq. (3.12)
14. **while** $tol(\tau_k(O_k^{\text{cur}})) < \sum_{\tau_i \in \Gamma,\ (i \neq k)} int(\tau_i(O_i^{\text{pre}}), \tau_k)$ **do**
15. $O_k^{\text{cur}} \leftarrow O_k^{\text{cur}} + 1$
16. updated ← True
17. **if** $O_k^{\text{cur}} > O^{\max}$ **then**
18. $S_k \leftarrow False$ and $O_k^{\text{cur}} \leftarrow O^{\max}$
19. **break** // goto Line 12 to continue with next τ_k
20. **end if**
21. **end while**
22. **end for**
23. **if** any $S_k \in \mathbb{S}$ is $False$ **then break end if**
24. **end while**
25. **if** any $S_k \in \mathbb{S}$ is $False$ **then**
26. **return** not schedulable
27. **else**
28. **return** schedulable, $\mathbb{O}^{\text{cur}}$
29. **end if**

end procedure

3.4.2 Optimality of Algorithm 1

Theorem 1. Algorithm 1 is optimal in terms of the BCL schedulability analysis when the parallelization overhead is not significantly large, i.e., when $\alpha(O_k, O_k + 1) < m - O_k$. More specifically, if Algorithm 1 cannot find a parallelization option combination that makes the given task set schedulable by schedulability condition (Eq. 3.14), no other option combinations can do it either, when $\alpha(O_k, O_k + 1) <$

$m - O_k$.

Proof. For this proof, let us define the "$\leq$" relation between two parallelization option combinations $\mathbb{O}^X$, $\mathbb{O}^Y$ as follows:

$$\mathbb{O}^X \leq \mathbb{O}^Y \coloneqq \{O_k^X \leq O_k^Y \mid \forall k,\ 1 \leq k \leq n\}.$$

Now, suppose that Algorithm 1 reaches an unschedulable parallelization option combination $\mathbb{O}^A = (O_1^A, O_2^A, \cdots, O_n^A)$ meaning that, for at least one task, say τ_k, its option O_k^A already reaches the maximum, but its tolerance is still smaller than its received interference from other tasks. Then, we will show that any other option combination $\mathbb{O}^S$ cannot be schedulable for all the following cases:***case (1)*** $\mathbb{O}^A \leq \mathbb{O}^S$, ***case (2)*** $\mathbb{O}^S \leq \mathbb{O}^A$, and ***case (3)*** $\mathbb{O}^A \not\leq \mathbb{O}^S$ and $\mathbb{O}^S \not\leq \mathbb{O}^A$.

case (1): When $\mathbb{O}^A \leq \mathbb{O}^S$, due to the monotonic increasing property of the interference (Lemma 3), interference given to τ_k must be greater with the $\mathbb{O}^S$ option combination. Since already in $\mathbb{O}^A$, O_k^A was at maximum parallelization, so tolerance of τ_k cannot increase anymore. $\mathbb{O}^S$ is not schedulable.

case (2): For case $\mathbb{O}^S \leq \mathbb{O}^A$, we denote the x-th iteration of Algorithm 1 as $\mathbb{O}^{A(x)} = (O_1^{A(x)}, O_2^{A(x)}, \cdots, O_n^{A(x)})$. For example, the initial option combination is $\mathbb{O}^{A(0)} = (O_1^{(1)}, O_2^{(1)}, \cdots, O_n^{(1)})$, and the final option combination, say the X-th iteration, is $\mathbb{O}^{A(X)} = \mathbb{O}^A$. While in the iteration, each parallelization option monotonically increases, i.e.,

$$\mathbb{O}^{A(x)} \leq \mathbb{O}^{A(y)}\ \ \forall x, y,\ 1 \leq x < y \leq X.$$

Because the iteration starts from the lowest parallelization and $\mathbb{O}^S \leq \mathbb{O}^A$, among the X iterations, there is always one iteration, e.g., y-th iteration, such that $\mathbb{O}^{A(y)} \leq \mathbb{O}^S \leq \mathbb{O}^A$. Let us call the largest among them $\mathbb{O}^{A(z)}$, and hence $\mathbb{O}^{A(z+1)} \not\leq O^S$. Therefore, there exists at least one element $O_k^{A(z+1)}$ in which $O_k^S < O_k^{A(z+1)}$.

On the other hand, according to Algorithm 1, $\mathbb{O}^{A(z+1)}$ is the smallest possible option combination that can tolerate interference with combination $\mathbb{O}^{A(z)}$ in the previous iteration. Then because $O_k^S < O_k^{A(z+1)}$, τ_k's tolerance is lower with $\mathbb{O}^{A(z)}$, and hence is not schedulable with $\mathbb{O}^{A(z)}$. To make this worse, since $\mathbb{O}^{A(z)} \leq \mathbb{O}^S$, the interference τ_k receives with $\mathbb{O}^S$ is actually larger than or equal to that of $\mathbb{O}^{A(z)}$ — due to the monotonic increasing property of interference. Therefore τ_k is still not schedulable with $\mathbb{O}^S$.

case (3): For the case where $\mathbb{O}^A \not\leq \mathbb{O}^S$ and $\mathbb{O}^S \not\leq \mathbb{O}^A$, we prove by contradiction.

Suppose that there exists a schedulable option combination $\mathbb{O}^S$. Then, there must be at least one element of $\mathbb{O}^S$ larger than the one in $\mathbb{O}^A$, i.e., $\exists x,\ O_x^A < O_x^S$. Similarly, at least one element must be smaller, i.e., $\exists y,\ O_y^S < O_y^A$. Let us classify the former elements into Group-X, and the latter into Group-Y. Then we make another combination $\mathbb{O}^{S'}$ by taking the smaller elements from either group, i.e.,

$$\mathbb{O}^{S'} = \Big(\min(O_1^A, O_1^S), \min(O_2^A, O_2^S), \cdots, \min(O_n^A, O_n^S) \Big).$$

Using this, we will derive a contradiction, that if $\mathbb{O}^S$ is schedulable, $\mathbb{O}^{S'}$ ($\leq \mathbb{O}^A$) is also schedulable, which is not possible because of ***case (2)***.

For a task τ_x in Group-X, because $O_x^A < O_x^S$, O_x^A is not at the maximum option $O^{(\max)}$. Therefore τ_x's tolerance is larger than its received interference with option

$\mathbb{O}^A$. Since $\mathbb{O}^{S'} \leq \mathbb{O}^A$, τ_x's received interference is decreased with smaller option $\mathbb{O}^{S'}$ (Lemma 3). Hence, τ_x remains schedulable with $\mathbb{O}^{S'}$.

For a task τ_y in Group-Y, because we assumed $\mathbb{O}^S$ is schedulable, τ_x's tolerance is larger than its received interference with option $\mathbb{O}^S$. Since $\mathbb{O}^{S'} \leq \mathbb{O}^S$, τ_y's received interference is decreased with the smaller option $\mathbb{O}^{S'}$, which makes τ_y remain schedulable.

Therefore, if $\mathbb{O}^S$ is schedulable, $\mathbb{O}^{S'} (\leq O^A)$ is also schedulable, which is a contradiction due to ***case (2)***. $\mathbb{O}^S$ is not schedulable. □

3.4.3 Time Complexity of Algorithm 1

Algorithm 1 conducts a uni-directional search in the outer **while**-loop from Line 9 to Line 24 by leveraging the monotonic increasing property of both tolerance and interference. Thus, for n tasks with the constant number $O^{\max}$ of parallelization options, the number of options increases is at most $n \cdot O^{\max}$ and hence $O(n)$. For each of such increases, the algorithm may calculate the interference in Line 13 with the time complexity of $O(n)$ for all n tasks by the **for**-loop from Line 12 to Line 22. This becomes $O(n^2)$ time complexity. Thus, the overall time complexity of the algorithm is $O(n^3)$. Our python3 implementation of Algorithm 1 takes $1.44ms$ to complete for $n = 10$ tasks on an Intel Core i7-8700 CPU (6-cores) machine clocked at $3.2GHz$.

3.5 Experiment Results

In this section, we show the effectiveness of the proposed task parallelization algorithm by both simulation with synthetic tasks and actual implementation with real autonomous driving tasks. We first show the evaluated schedulability using synthetic

tasks in Section 3.5.1, then in Section 3.5.2 we perform a series of artificial scheduling to investigate the ceiling of our approach. Next, in Section 3.5.3, we survey the extreme task set cases to reveal the limitation and effectiveness of parallelization and the BCL schedulability analysis. Finally, in Section 3.5.4, we demonstrate the performance through actual implementation with real autonomous driving tasks and additionally with a real-time benchmark.

3.5.1 Simulation Results

For our simulation, a synthetic task $\tau_k = (T_k, D_k, E_k)$ is randomly generated as follows: (1) T_k is randomly generated from uniform$[600, 2000]$, (2) D_k is randomly generated from uniform$[400, T_k]$, and (3) the thread execution time table E_k is created starting from $e_k^1(O_k = 1) = C_k(O_k = 1) =$ uniform$[300, 1000]$, $C_k(O_k + 1)$ where $O_k < O^{\max}$ is computed using $\alpha(O_k, (O_k + 1)) = \frac{C_k(O_k+1) - C_k(O_k)}{e_k^1(O_k) - e_k^1(O_k+1)}$, $C_k(O_k + 1)$ is divided into $(O_k + 1)$ pieces by Unifast [15] algorithm, and those pieces are sorted in the descending order and assigned to $e_k^1(O_k + 1), e_k^2(O_k + 1), \cdots, e_k^{O_k+1}(O_k + 1)$.

Using this way of generating a synthetic task, we generate 10^6 task sets with different number of tasks n and different task set utilization $\sum_{k=1}^{n} U_k = \sum_{k=1}^{n} C_k(1)/T_k$ as follows:

1. Create an empty task set Γ.
2. Generate a new synthetic task τ_k and add it to Γ.
3. Check whether Γ passes the necessary condition, i.e., $\sum_{\tau_k \in \Gamma} C_k(1)/T_k < m$.
4. If true, it becomes one task set Γ. Then, with a copy of Γ, we add one more task by repeating the above two steps to make another task set with one more

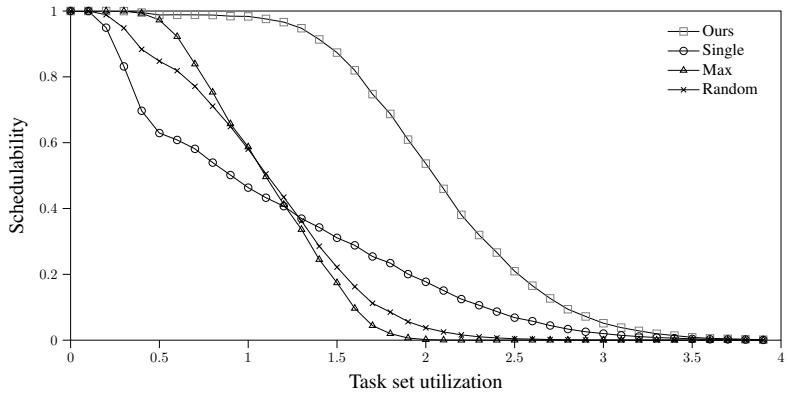
task.

5. If not, meaning that Γ is already large enough violating the necessary schedulability condition, we start over from the beginning with a new empty task set Γ.

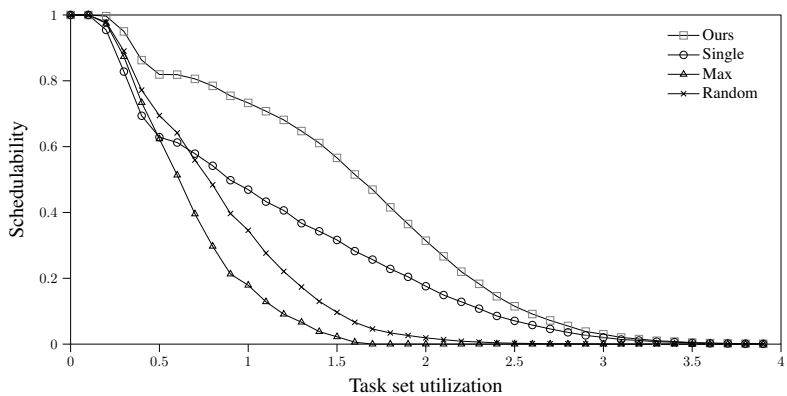
With such generated 10^6 task sets and four CPU cores, i.e, $m = 4$, we compare the schedulability resulting from the following four different parallelization approaches:

- Ours: Each task parallelized using our proposed algorithm, i.e., Algorithm 1.
- Single: No parallelization is used. Each task executes as a single thread.
- Max: Every task τ_k is parallelized into the maximum number of threads, i.e., $O_k = O^{\max}$.
- Random: For each task τ_k, the parallelization option is randomly chosen from uniform$[1, O^{\max}]$.

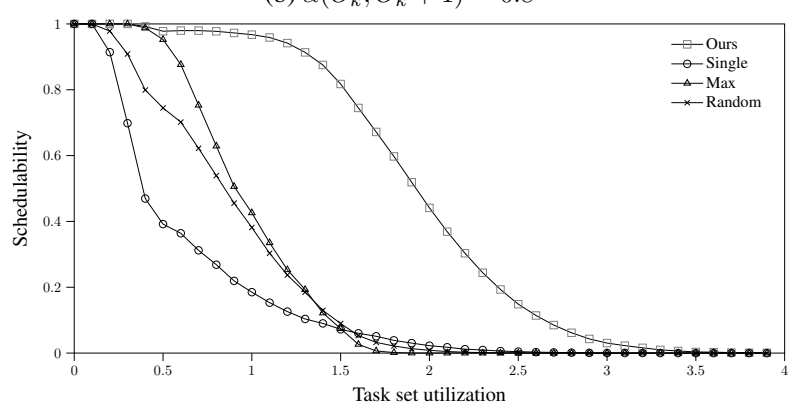
Fig. 3.9(a) compares the schedulability of the above four approaches for the whole spectrum of the task set utilization of 10^6 task sets ranging from 0 to 4 assuming the parallelization overhead $\alpha(O_k, O_k+1) = 0.3$ for every $O_k < O^{\max} = 4$. In the figure, the x-axis represents the task set utilization, and the y-axis represents the schedulability, i.e., the proportion of schedulable task sets out of all the task sets having the corresponding task set utilization. Comparing "Max" and "Single", the former performs better in low task set utilization while the latter performs better in high task set utilization, making a crossing point around the task set utilization of



(a) $\alpha(O_k, O_k + 1) = 0.3$



(b) $\alpha(O_k, O_k + 1) = 0.8$



(c) $\alpha(O_k, O_k + 1) = 0.3$, 20% tightened deadlines

Figure 3.9: Simulation result with $m = 4$ CPU cores

1.3. This is because, in low utilization, the schedulability is mostly affected by deadlines, and hence reducing the thread execution time by parallelization helps. On the other hand, in high utilization, the schedulability is mostly affected by the amount of interferences among tasks and hence keeping the number of threads small help. "Random" is between "Max" and "Single". "Ours" performs significantly better in any task set utilization, e.g., 60% more schedulable task sets at the crossing point of "Max" and "Single"[‡]. This is because Algorithm 1 optimally trades off the tolerance and interference in the parallelization option selection for the schedulability of all the tasks in the given task set.

Fig. 3.9(b) shows the results when the parallelization overhead is larger, i.e., when $\alpha(O_k, O_k+1) = 0.8$. In this case, "Single" is not affected, but "Max" becomes worse since parallelization benefits are severely sacrificed due to the large parallelization overhead. "Ours" is similarly affected. Nevertheless, "Ours" still significantly outperforms all other approaches in this case of large parallelization overhead.

Fig. 3.9(c) shows the results when the deadline for each task is 20% tighter than Fig. 3.9(a). In this case, "Single" becomes significantly worse because it does not take advantage of thread execution time reduction by parallelization. On the other hand, the performance of "Max" and "Random" also drop but not that much compared with "Single" thanks to the parallelization. The performance of "Ours" slightly drops since it can well overcome the tightened deadlines by optimal parallelization.

[‡]By applying the improved interference-based schedulability analysis called RTA [12] to "Single", "Max", and "Random", our experiments say that their schedulability improves, but less than 5%.

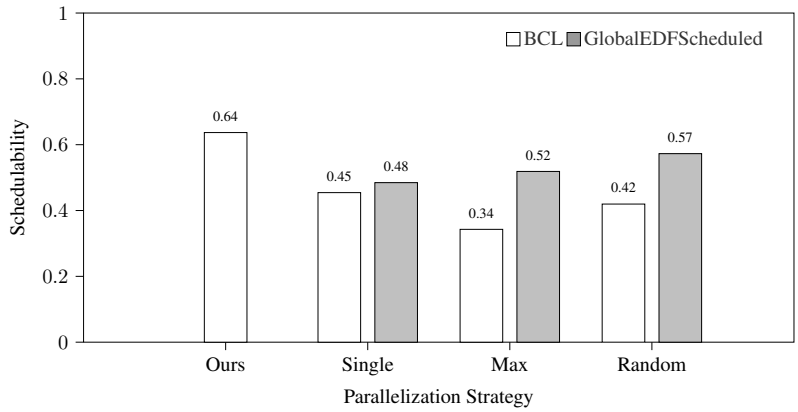


Figure 3.10: Performance of simulated global EDF schedule

3.5.2 Simulated Schedule Results

Parallelization freedom for multi-segment tasks is realized based on a sufficient, interference-based schedulability analysis, i.e., BCL. Therefore the proposed algorithm inevitably inherits the built-in pessimism from BCL. Intuitively speaking, if the schedulability gap of BCL is too large, the scheduling benefit that our proposed algorithm brings can be diluted. To investigate and quantify this effect, we conducted a simulated global EDF schedule with synthetic tasks. The results are drawn in Fig 3.10.

The simulated global EDF schedule is administered in the following procedure: (1) Task Sets generated through the same procedure as the base case in Section 3.5.1, where parallelization overhead is $\alpha = 0.3$ (2) Schedulability of each task set is first checked using BCL. (3) If not schedulable, the schedulability is rechecked with simulated global EDF scheduling.

Because our problem definition considers the sporadic multi-segment task, there can be infinitely many release patterns to check. Therefore, we can only check for

a limited number of scenarios. For this experiment, we consider every task follows a strictly periodic release pattern with zero release offset. In other words, all tasks $\tau_k \in \Gamma$ are assumed to release simultaneously at $t = 0$, and subsequent jobs are released most frequently as possible, according to their minimum separation time T_k.

A time-based simulation is run, wherein each step $m = 4$ threads are selected to execute according to global EDF scheduling. If at any time stop, a job has remaining execution time after its absolute deadline expired, we consider the deadline is violated and return "Not schedulable".

For each task set, the simulation is run until the hyper-period H, which is the least common multiple of all T_k, because the release pattern will only repeat after H. If no deadline miss is observed until H, the task set is deemed schedulable. Note that H can quickly become intractably large. Therefore we cap H with a value of 10^6. Any time step later than this upper bound will not be considered.

Fig 3.10 compares the schedulability of BCL and simulated global EDF according to different parallelization strategies. The unfilled boxes represent the BCL schedulability, and the filled boxes represent the simulated global EDF scheduling with the previously described procedure. Note "Ours" does not output a valid parallelization option for "deemed not schedulable" task sets. Therefore simulated global EDF is not conducted for "Ours".

As shown in the figure, the simulated global EDF schedule outpaces BCL by a great margin. The gap in schedulability is exaggerated as parallelization increases, as it peaks at the "Max" strategy. This is because of two reasons, (1) because parallelization overhead is relatively low, $\alpha = 0.3$, choosing higher parallelization is beneficial in actual scheduling. (2) Because hyper-period H capped to 10^6 when the

number of threads grows, we omit a more significant portion of release scenarios in high parallelization scenarios.

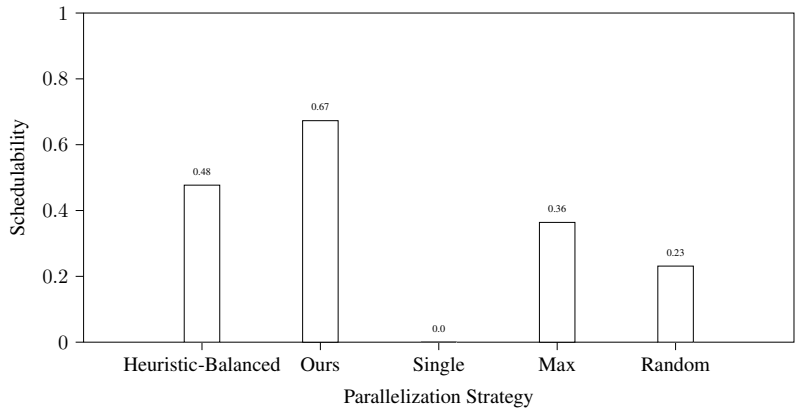
However, “Ours” can make optimal use of the BCL schedulability analysis and outperforms all cases, even in such an “easy-win” scenario for simulated global EDF scheduling due to restricted simulation conditions.

3.5.3 Survey on the Boundary Condition of the Parallelization Freedom

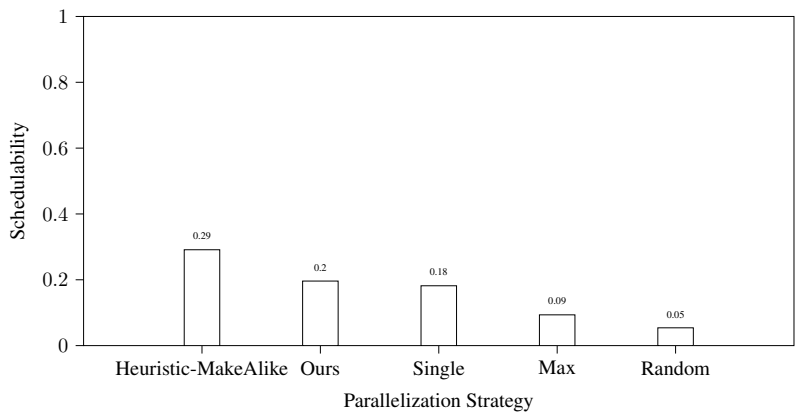
BCL schedulability analysis is a sufficient analysis meaning, the analysis is absolutely confident for the “schedulable” results but is not for the “unschedulable” results. This creates a gap of pessimism, as shown in the previous section. However, also shown in the last section, “Ours” can optimally select the parallelization options according to the BCL analysis, and as a result, “Ours” can even outpace the “exhaustive” competitors. In other words, we “squeezed out” maximal possible schedulability gain from the BCL analysis.

Thus, the potential schedulability gain through Algorithm 1 is heavily limited by the performance of the BCL analysis itself. Furthermore, being tightly coupled with the BCL analysis, the performance of Algorithm 1 may highly suffer in the same situations where the BCL analysis falls short.

To point out the circumstances and evaluate the magnitude of the performance loss, we survey the extreme boundary cases of the BCL analysis in this section. In particular, we focus on situations where an intuitive parallelization strategy can be thought of and applied with ease. Through this section, we attempt to affirm the reach of our proposed algorithm by fully disclosing the limitations. Those cases are: (1) number of tasks is equal to the number of processors, each task’s utilization confined



(a) Task set of large utilization tasks



(b) Task set of mixed utilization tasks

Figure 3.11: Parallelization freedom — boundary cases

under 1, (2) same as (1), but all task's utilization is over 1, and (3) few tasks with a large utilization mixed with many small utilization tasks. For all cases, we assume the implicit deadline case where $D_k = T_k$ for all tasks.

For each case, we suggest a heuristic to be coupled with, which can be thought up rather intuitively. The suggested heuristics aim to be simple and try to maximize schedulability by exploiting the nature of the given situation. For each case, the coupled heuristic is evaluated using a simulated global EDF schedule and compared

against the performance of Algorithm 1. The suggested heuristics are independent of the BCL analysis, thus attempting to maximize the schedulability for the given situation. On the other hand, Algorithm 1 relies on sufficient BCL schedulability analysis, so the schedulability implies pessimism.

In case (1), the number of tasks equals the number of processors, and each task's utilization is under 1. This is a case where the schedulability can be trivially deduced, i.e., true, with the "Single" strategy. Thus, the coupled heuristic would make all threads run single, which can be called "Heuristic-Single". In this situation, "Heuristic-Single" has 100% schedulability for all possible generated cases. However, this also applies to Algorithm 1 since BCL can consider this situation, and Algorithm 1 also starts from all single threads. Therefore, for this case, "Ours" and "Heuristic-Single" have an identical performance of 100%.

In case (2), the number of tasks equals the number of processors, and each task's utilization is over 1. In such a case, schedulability cannot be easily guessed. However, the "Single" strategy surely fails because the deadline is shorter than the required execution for all tasks. Therefore the parallelization option must be increased for all tasks. A viable strategy will be to raise the parallelization of each task equally, say all threads 2 or all threads 3, and so on. All tasks are raised until every task's largest thread's utilization falls below 1. Considering that the parallelization overhead is compounded by increasing the options, incrementing options in a balanced fashion is both a simple and a reasonable approach. We call this "Heuristic-Balanced".

A comparison of "Heuristic-Balanced" and other strategies including "Ours" is shown in Fig. 3.11(a), where "Heuristic-Balanced" is evaluated using simulated global EDF schedule assuming strictly periodic release until the hyper-period. We can see

that “Heuristic-Balanced” performs significantly better than the other naive approaches, i.e., “Single”, “Max”, and “Random”, due to its exploitation of the given situation. However, “Ours” still outpaces even in such a situation by utilizing the BCL analysis to its full.

Finally, in case (3), the task sets are comprised of a few large utilization tasks mixed with many small utilization tasks. In this case, an intuitive approach would be to parallelize the larger utilization tasks first to make them more likely to be schedulable. Let us call the group of larger tasks group-A and smaller utilization tasks group-B. Thus for tasks in group-A, we parallelize them until (1) their individual threads have similar(or lower) utilization compared to the average utilization of group-B or (2) they reach maximum parallelization. Generally speaking, this strategy tries to even out the utilization for all threads so that no threads are especially harder to schedule. We can call this heuristic “Heuristic-MakeAlike”.

Fig. 3.11(b) compares “Heuristic-MakeAlike” and other approaches. Where again, “Heuristic-MakeAlike” is evaluated using a simulated global EDF schedule assuming strictly periodic release until the hyper-period. “Heuristic-MakeAlike” performs better than any other strategies. Since all naive strategies cannot favor both group-A and group-B at the same time, their performance is relatively low. “Ours” innately considers the tasks individually and assigns higher parallelization options to those needed. Therefore “Ours” is more performant than the naive strategies.

3.5.4 Autonomous Driving Task Implementation Results

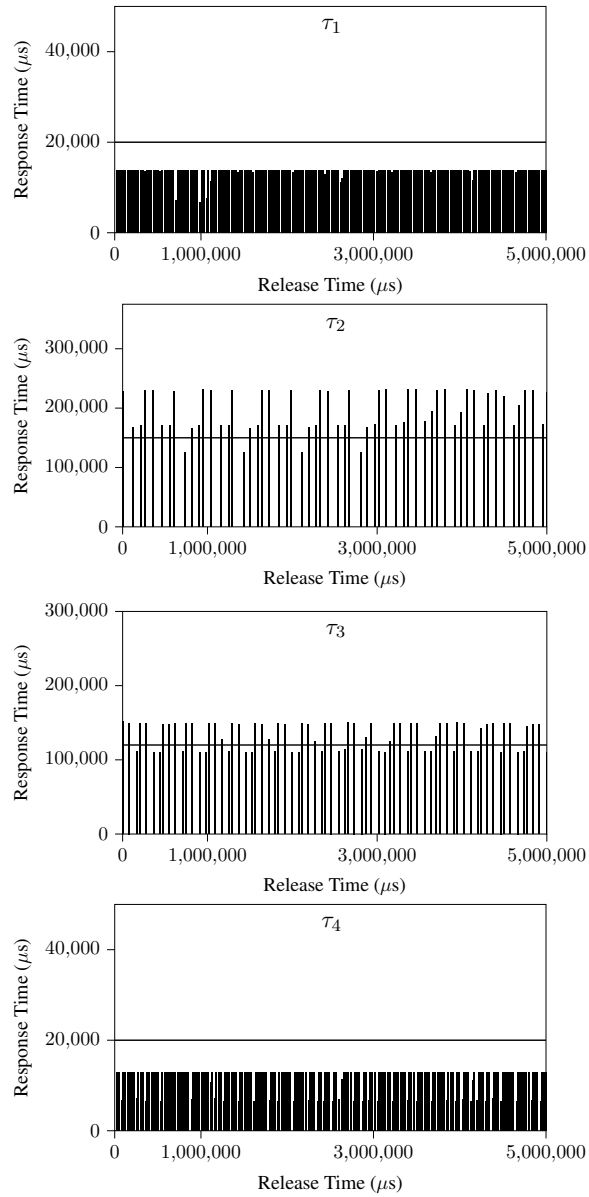
For the practical justification of our proposed approach, we composed two implementation experiments: Autoware [32] based autonomous workload evaluation and

RT-App [50] based benchmark test.

Both experiments are conducted on PREEMPT_RT patched on top of Linux kernel 4.19. All tasks are scheduled according to G-EDF (SCHED_DEADLINE) scheduler on a PC equipped with Intel Core i7-8700 CPU (6-cores). We use CPUSET to limit the execution of our tasks into $m = 4$ cores, fix the clock frequency at 3.20 GHz, and disable GPU.

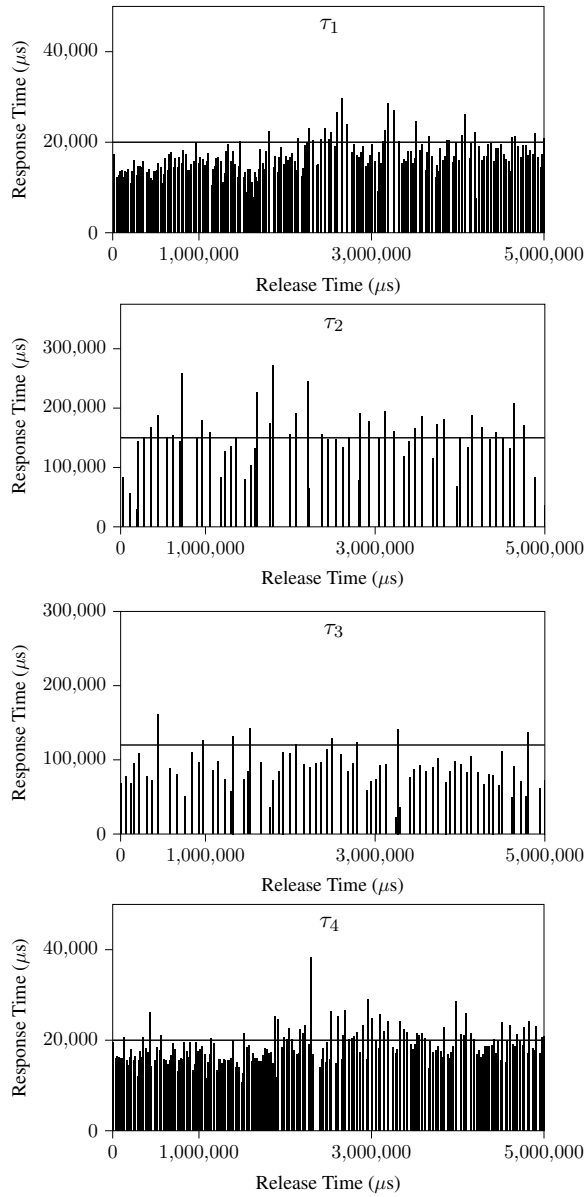
For the first implementation experiment, we execute the following four real programs used for autonomous driving, i.e., (1) τ_1: sensor read program emulating a multi-channel camera module, (2) τ_2: lane tracking program used by Autoware [32], (3) τ_3: darknet [45]-based object detection and labeling program used by Apollo [4], and (4) τ_4: steering actuation program emulating PID motor controller. Their minimum inter-release times and deadlines are set as $(T_1 = D_1 = 20,000)$, $(T_2 = D_2 = 150,000)$, $(T_3 = D_3 = 120,000)$, and $(T_4 = D_4 = 20,000)$, where the time unit is μs.

Fig. 3.12 shows the measured response times of the largest thread $\tau_k^1(O_k)$ of each task τ_k. In each graph, the x-axis is the release time of each job, and the y-axis is its corresponding response time. The horizontal line on each graph is the deadline D_k, and thus the response time above it means a deadline violation.



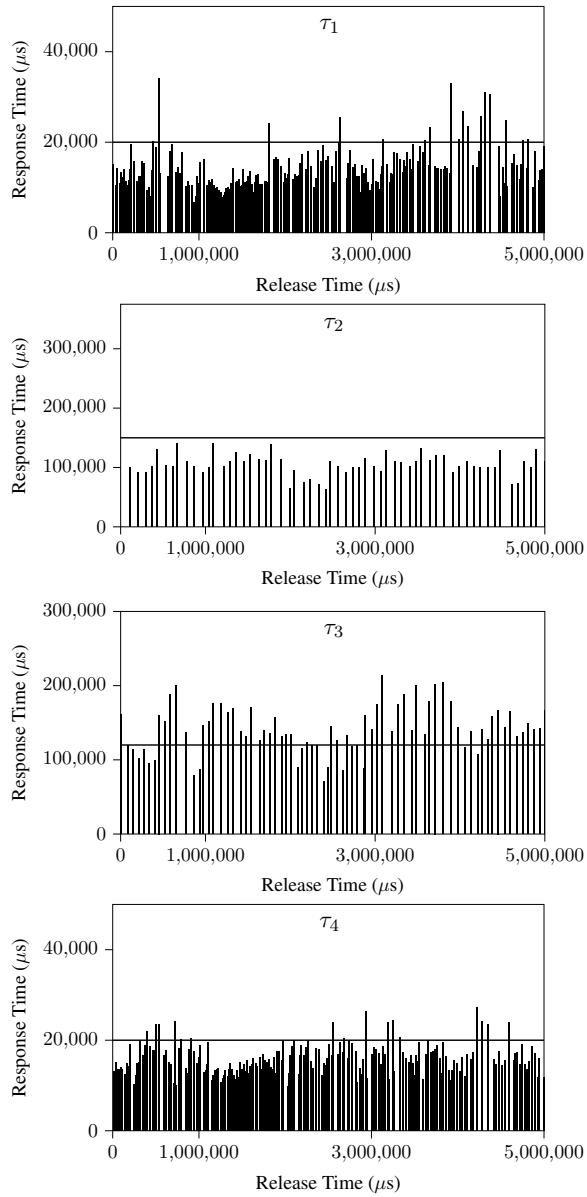
(a) Strategy: “Single”

Figure 3.12: Measured response times of autonomous driving tasks



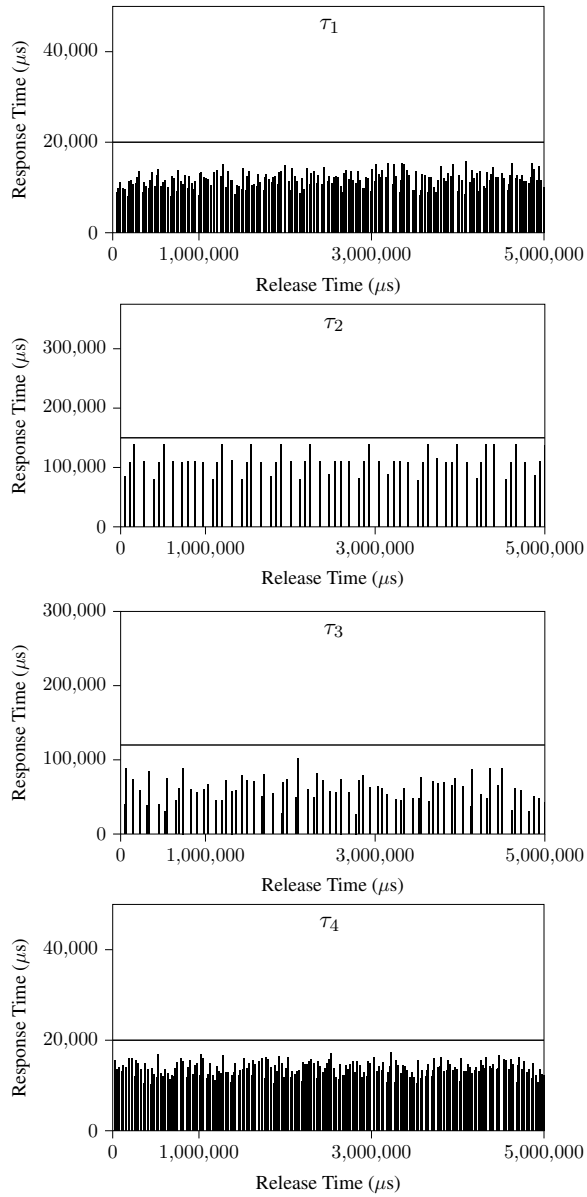
(b) Strategy: “Max”

Figure 3.12: Measured response times of autonomous driving tasks(cont.)



(c) Strategy: “Random”

Figure 3.12: Measured response times of autonomous driving tasks(cont.)



(d) Strategy: “Ours”

Figure 3.12: Measured response times of autonomous driving tasks(cont.)

In "Single", we can observe many deadline misses of τ_2 and τ_3. This is because they are time-taking tasks, and hence without parallelization, their execution times often are longer than the deadlines. In "Max" and "Random", the number of τ_2's and τ_3's deadline misses decrease thanks to the parallelization, but now we observe deadline misses of the short tasks τ_1 and τ_4 due to increased interferences. On the other hand, we can observe that "Ours" meet all the deadlines of all four tasks by the optimal parallelization.

Our proposed algorithm is additionally evaluated using real-time benchmark RT-App [50]. RT-App is one of the most used real-time CPU workload benchmarks and is notably used in the Linux mainline kernel [27]. To evaluate our proposed algorithm with RT-App, the same task sets used in Section 3.5.1 are used. The experiment is conducted in the following manner:

1. Tasks are generated through the same procedure as in Section 3.5.1, with parallelization overhead $\alpha = 0.3$. Configure RT-App tasks to have such parameters.

2. Create a task set with a single task, then execute with RT-App on $m = 4$ CPU cores for 10 seconds. Repeat for all parallelization strategies.

3. If there was no deadline miss during the run, the task set is deemed schedulable according to the selected parallelization strategy. Add another task, then execute RT-App again.

4. Repeat the above step until all strategies experience a deadline miss. In such a case, discard the task set and start over from Step 2.

The above procedure is conducted for 10^4 task sets, and the results are depicted in

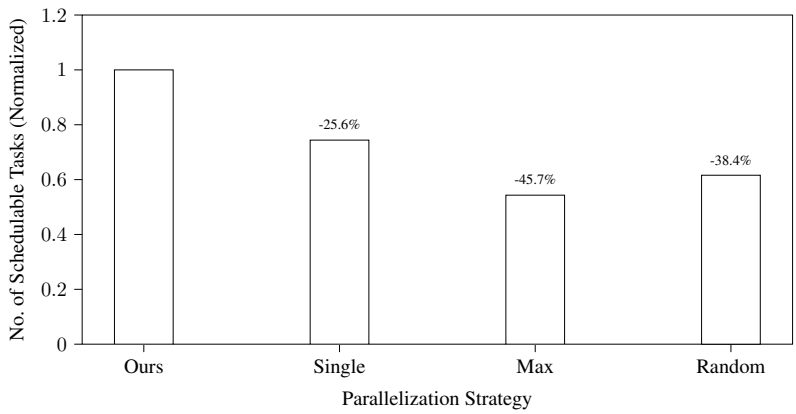


Figure 3.13: RT-App benchmark evaluation results with $m = 4$ CPU cores

Figure 3.13. For different parallelization strategies, their average number of schedulable tasks is shown normalized.

As immediately shown in the figure, the "Ours" strategy significantly outpaces other strategies. "Max" suffers harshly, recording a 45.7% drop in relative performance. "Single" performs poorly, but not as much as "Max". "Random" is in-between "Single" and "Max". Overall, the effect of parallelization overhead is exaggerated in the benchmark experiment than in the previous simulation-based experiments. This is because the threads here are real and suffer from actual overheads like syncing and context switching. These "actual" overheads are additionally added to the $\alpha = 0.3$ overhead, which is already assigned to RT-App tasks.

Chapter 4
Conditionally Optimal Parallelization of Multi-Segment and DAG Tasks

4.1 Introduction

The multi-segment (MS) task model can portray a wide range of workloads that have intermittent synchronization points. In other words, the multi-segment model describes a series of parallel executions that must run in sequence. Many computation-intensive real-time workloads, such as convolution neural network (CNN), can be characterized as a multi-segment task.

On the other hand, real-time applications are rapidly growing in their size and complexity. Such complex program models conform to a directed-acyclic-graph (DAG) task model, where each node of the graph represents a task, and the connected edges portray the precedence relation between the tasks. Thus, the DAG task model can be viewed as a generalization of the multi-segment model, where execution may take place in multiple paths, as long as the precedence relation holds.

In this chapter, we first show how parallelization freedom can be brought to the multi-segment model. First, the parallelization option assignment algorithm is extended so that each segment of the task can have different values to eventually optimize the system schedulability. To achieve this, we base on the state-of-the-art schedulability analysis for multi-segment, namely Chwa-MS [23]. Next, we extend the analysis for parallelization freedom and formally construct the tolerance and in-

terference function for the multi-segment model. We then show that the monotonic increasing property of both tolerance and interference is preserved when following a selected parallelization route, which helps us assign a parallelization option. Finally, we derive a near-optimal parallelization option assignment algorithm for the multi-segment task model by utilizing those functions.

Continued in this chapter, we extend the concept of parallelization freedom to the DAG task model so that each node can be freely parallelized into a desirable number of threads. The state-of-the-art schedulability analysis for multi-segment, i.e., Chwa-DAG [22], is extended for parallelization freedom. Like the multi-segment model, we formally model the monotonic increasing property of both tolerance and interference, according to a selected parallelization route. Then the parallelization option is assigned node-wise to maximize schedulability.

Finally, the effectiveness of the algorithms each targeting the multi-segment or the DAG task model is extensively evaluated with both simulated and implemented workloads. AutowareAuto [26], a leading open-source autonomous driving framework, is used for the implementation evaluation.

This chapter is organized as follows: Section 4.2 formally describes the multi-segment model. Then Section 4.3 introduces the Chwa-MS schedulability analysis and derives tolerance and interference properties. Section 4.4 presents and utilizes the parallelization route to assign parallelization option to segments.

Then addressing the DAG task model, Section 4.5 formally describes the DAG task model. Then Section 4.6 introduces the Chwa-DAG schedulability analysis and derives tolerance and interference properties. Afterwards, Section 4.7 presents and utilizes the parallelization route to assign parallelization option to segments.

Finally, Section 4.8 reports our experiment results for the multi-segment task model and Section 4.9 reports our empirical results of DAG parallelization algorithm using simulated and implemented workload, i.e., AutowareAuto.

4.2 Multi-Segment Task Model

A multi-segment task τ_k, depicted in Fig. 4.1, consists of S_k segments that must execute sequentially. Each segment can only start to execute after the previous segment has finished executing. A segment of τ_k is denoted τ_{ks}, where $s \in \{1, ..., S_k\}$.

$$\tau_k = \{\tau_{k1}, \tau_{k2}, \cdots, \tau_{kS_k}\}.$$

τ_k requires complete execution of all segments within the relative deadline D_k, and consequent jobs are separated at least by the minimum inter-release time T_k.

Enjoying the parallelization freedom, we consider that those S_k segments can freely be parallelized into a desirable number of threads. The number of threads of each segment O_{ks} can range from 1 to $O^{\max}$, i.e., the system maximum. Likewise, they are sibling threads $\tau_{ks}(O_{ks})$:

$$\tau_{ks}(O_{ks}) = \{\tau_{ks}^1(O_{ks}), \tau_{ks}^2(O_{ks}), \cdots, \tau_{ks}^{O_{ks}}(O_{ks})\}.$$

When parallelization option is decided for all S_k segments, we denote such parallelized multi-segment task as $\tau_k(O_{k1}, O_{k2}, ..., O_{kS_k})$. For example, the parallelized multi-segment task depicted in Fig. 4.1, i.e. $\tau_k(2, 1, 3)$, has 2, 1, 3 threads in each of its segments, respectively. For simplicity, we will now on denote the set of chosen option for all segments:

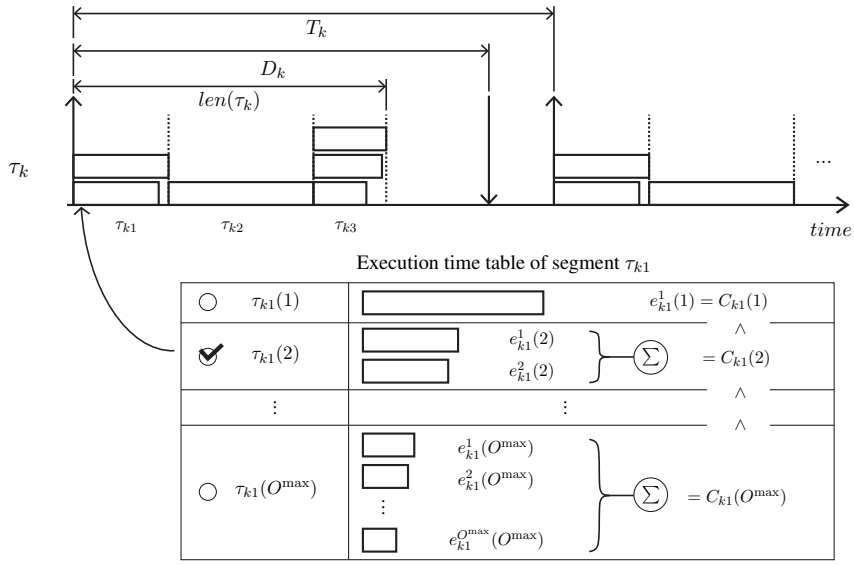


Figure 4.1: Multi-segment task with parallelization freedom

$$O_k = \{O_{k1}, O_{k2}, ..., O_{kS_k}\}.$$

The l-th sibling thread $\tau^l_{ks}(O_{ks})$ has the WCET (worst-case execution time) $e^l_{ks}(O_{ks})$ as indicated in the execution time table in the lower part of Fig. 4.1. Without loss of generality, we sort the O_{ks} threads of $\tau_{ks}(O_{ks})$ in descending order of WCET, and hence $\tau^1_{ks}(O_{ks})$ has the largest WCET, i.e.,

$$\max_{\tau^l_{ks} \in \tau_{ks}} e^l_{ks}(O_{ks}) = e^1_{ks}(O_{ks})$$

The total computation amount of these O_{ks} sibling threads is denoted as:

$$C_{ks}(O_{ks}) = \sum_{l=1}^{O_{ks}} e^l_{ks}(O_{ks}).$$

Finally, the critical path is defined as the collection of the largest thread, i.e., the first thread, from each segment. Then, its length is defined as:

$$len(\tau_k(O_k)) = \sum_{s=1}^{S_k} e_{ks}^1(O_{ks}).$$

Under these assumptions, our problem is formally defined as follows:

Problem Definition: For each multi-segment tasks τ_k in the given task set $\Gamma = \{\tau_1, \tau_2, \cdots, \tau_n\}$, our problem is to find a parallelization option O_{ks} for its segments τ_{ks}, such that all the tasks in Γ can be scheduled meeting their deadlines using global EDF on m identical CPU cores.

4.3 Extension of Chwa-MS Schedulability Analysis

4.3.1 Chwa-MS Schedulability Analysis

In the work of Chwa et al., BCL analysis was extended to target the multi-segment task model [23], by significantly tightening the overestimation of parallel threads' interference. We will use this analysis to address our problem of assigning parallelization options for the multi-segment task system.

Being an extension of BCL, the mutual interference of the multi-segment tasks is also bounded by their total worst-case workload. An worst-case example is depicted in Fig. 4.2, where τ_i is interfering τ_k's execution. In such an example, the worst-case workload of τ_i is given to τ_k, when τ_i's deadline is aligned to that of τ_k, and the previous jobs of τ_i are released most frequently. Additionally, all threads of τ_i's *carry-in job* are executed at the latest time possible, i.e., all threads of the last segment τ_{i3} complete their execution at the absolute deadline of the *carry-in job*, and all previous

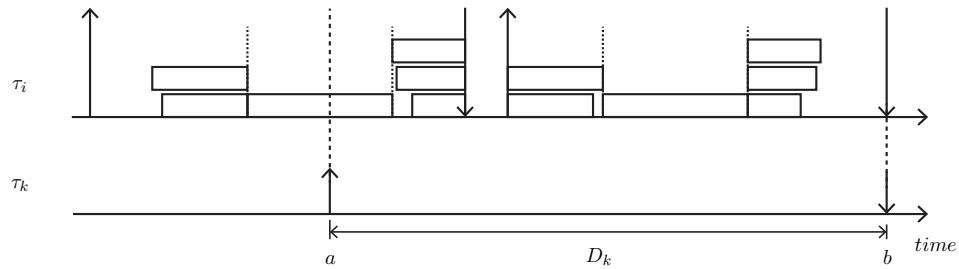


Figure 4.2: Worst-case release pattern of τ_i in $[a, b) = [r_{kj}, d_{kj})$

segments are tightly packed to the deadline.

Since threads of a multi-segment task may execute simultaneously on multiple cores, the actual contribution of τ_i's workload can be bounded by calculating the maximal simultaneous core usage. Chwa introduces this concept as p-depth critical interference, which means the interval where at least p number of cores are simultaneously being occupied by τ_i's job, blocking τ_k's critical path. We denote such p-depth critical interference of τ_i to τ_k as: $\overline{W}^p_{\tau_i,\tau_k}$.

The value of $\overline{W}^p_{\tau_i,\tau_k}$ can be calculated for each p, by adding up the workload (given by Eq. (3.3)) of the p-th largest thread from each segment, and then bounding the sum with $D_k - len(\tau_k(O_k))$. Because sibling threads are sorted in descending order of the thread execution time, p-depth critical interference is calculated by simply adding workload of the p-th thread from each segment and then bounding it by $D_k - len(\tau_k(O_k))$:

$$\overline{W}^p_{\tau_i,\tau_k} = \min\Big(\sum_{s=1}^{S_i} \{ \left\lfloor \frac{D_k}{T_i} \right\rfloor e^p_{is} + \min\left(e^p_{is}, D_k \bmod T_i\right)\}, D_i - len(\tau_k(O_k)) \Big). \tag{4.1}$$

Note that for a single-segment task, i.e. $p = 1$, this value is exactly the same as

the bounded workload (Eq. (3.5)) of the original BCL analysis.

When we sum up all p-depth critical interference for $p \in (1, 2, \cdots, m)$, and for all multi-segment tasks $\tau_i \in \Gamma$, we get the total p-depth critical interference that τ_k receives. This includes the interference from the non-critical execution path of τ_k, i.e., the interference between sibling threads, which is simply $p \geq 2$-depth critical interference of τ_k. Notice that since the segments belonging to the same task cannot compete with each other, their mutual interference must be zero, and the analysis correctly considers this.

When the summation of the above critical interference value is smaller than or equal to $D_k - len(\tau_k((O_k))$, then $\tau_k(O_k)$ is schedulable. Thus the final scheduling condition is derived:

$$\sum_{\tau_i \in \Gamma, i \neq k} \sum_{p=1}^{m} \overline{W}^p_{\tau_i,\tau_k} + \sum_{p=2}^{m} \overline{W}^p_{\tau_k,\tau_k} \leq D_k - len(\tau_k(O_k)). \tag{4.2}$$

4.3.2 Tolerance and Interference of Multi-Segment Tasks

Now tolerance function can be derived from Eq. (4.2), by moving the interference term of the sibling threads, i.e. $\sum_{p=2}^{m} \overline{W}^p_{\tau_k,\tau_k}$, to the right-hand side:

$$tol(\tau_k(O_k)) \coloneqq D_k - len(\tau_k(O_k)) - \sum_{p=2}^{m} \overline{W}^p_{\tau_k,\tau_k}, \tag{4.3}$$

Then the left-hand side becomes the interference given to τ_k by other tasks $\tau_i \in \Gamma$:

$$\sum_{\tau_i \in \Gamma,\ \tau_i \neq \tau_k} int(\tau_i(O_i), \tau_k(O_k)) \coloneqq \sum_{\tau_i \in \Gamma,\ i \neq k} \sum_{p=1}^{m} \overline{W}^p_{\tau_i,\tau_k}. \tag{4.4}$$

This yields the following scheduling condition:

$$\sum_{\tau_i \in \Gamma,\ \tau_i \neq \tau_k} int(\tau_i(O_i), \tau_k(O_k)) \leq tol(\tau_k(O_k)). \tag{4.5}$$

4.4 Assigning Parallelization Options to Multi-Segments

4.4.1 Parallelization Route

We showed in the previous chapter that both tolerance and interference function monotonically increase with the parallelization option in the traditional task model. However, this is only partly true for the multi-segment case. We illustrate this in the following example. Consider a two-segment task with option $O_k = (O_{k1}, O_{k2})$. In such example, there are two ways to increment the given option: (a) $(O_{k1} + 1, O_{k2})$ or (b) $(O_{k1}, O_{k2} + 1)$. In both cases, only a single segment is incremented while the other remains the same, which makes tolerance and interference functions increase compared to (O_{k1}, O_{k2}). However, between (a) and (b), the inequality does not always hold in one way but may vary according to the thread execution table of each segment.

On the other hand, our parallelization option assignment algorithm requires that tolerance and interference functions both monotonically increase with the parallelization option. Therefore the algorithm cannot be used with both (a) and (b) options.

To address this problem, we first define the *binary relation* between option combinations and then use the relation to create a subset of τ_k's option combinations that preserve *strict total ordering*.

Binary relation between option combinations can be defined as follows. An op-

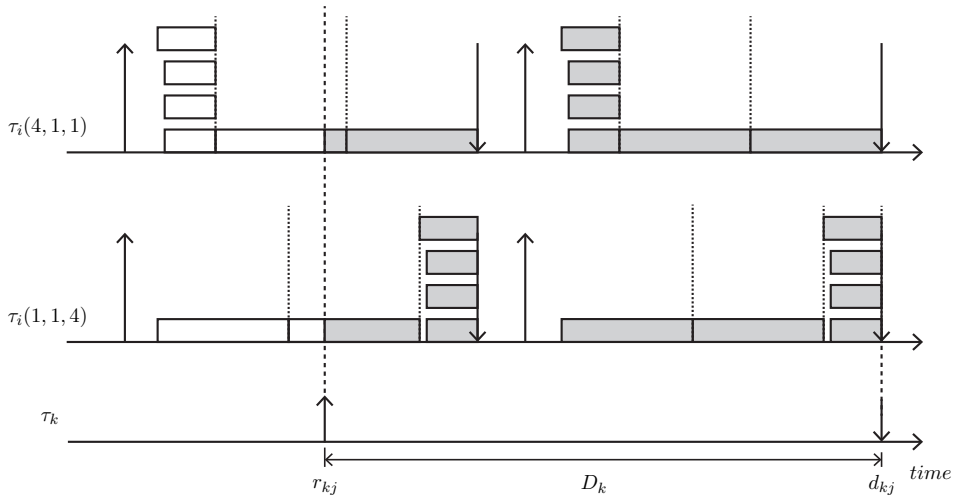


Figure 4.3: Different carry-in contribution of $\tau_i(4,1,1)$ and $\tau_i(1,1,4)$ to τ_k

tion O_k is strictly larger than another option O'_k, i.e. $O_k > O'_k$, when

(1) $\forall s \in \{1, 2, \cdots, S_k\}, O_{ks} \geq O'_{ks}$, and

(2) $\exists s^* \in \{1, 2, \cdots, S_k\}, O_{ks^*} > O'_{ks^*}$.

In other words, all segments of O_k must have greater or equal options than their corresponding O'_k segment's option, and there must be at least one segment of O_k that has a greater option than its O'_k counterpart.

Next, by using this relation, we can create a subset of τ_k's option combinations that preserve *strict total ordering*. We will denote such subset as $\vec{O}_k = \{O_k^1, O_k^2, \cdots\}$. To create such set, we start with $O_k^1 = \{1, 1, \cdots, 1\}$, then we choose and increment a segment's option by 1 to create the next element O_k^2. We repeat this process until we reach $\{O^{\max}, O^{\max}, \cdots, O^{\max}\}$, the last element. Through this process all elements of $\vec{O}_k$ monotonically increases, i.e. $\forall i < j, O_k^i < O_k^j$, thus monotonic increasing property of tolerance and interference functions is preserved for $\vec{O}_k$.

Despite this restriction, there are still vastly many valid $\vec{O}_k$s, which increases

exponentially with the number of segments. Therefore, in the remainder of this section, we propose a heuristic selection of $\vec{O}_k$ that performs near-optimal compared to exhaustive search.

To this end, we propose a heuristic selection of $\vec{O}_k$ that aims to reduce interference contribution to other tasks. Intuitively speaking, we leverage the fact that the carry-in workload is included starting from the deadline, so we preferably increment the nodes farther away from the deadline.

The intuition behind this heuristic is depicted in Fig. 4.3. In the figure two different parallelization of τ_i is shown: $\tau_i(4,1,1)$ and $\tau_i(1,1,4)$. We show the difference of their behavior upon giving interference to τ_k. $\tau_i(4,1,1)$ represents our heuristic where the first segment's parallelization option is increased to 4 while others segments remain at 1. We can observe that the first segment is not included for the *carry-in* contribution of $\tau_i(4,1,1)$. This greatly reduces the amount of interference $\tau_i(4,1,1)$ gives to τ_k. On the other hand, $\tau_i(1,1,4)$ increments the last segment first. We can see that this results in all threads of the last segment is included in the *carry-in* contribution. This increases interference given to τ_k.

To sum up, when calculating *carry-in* contribution of the interference function using Eq. (4.1), all tasks are considered starting from the deadline. Therefore we want to pack as least workload as possible closer to the deadline, and this is done by incrementing the 'farthest from deadline segment' first.

Keeping these in mind, we can build parallelization route $\vec{O}_k$ as the following:

1. Start from the lowest parallelization for all segments, i.e., $O_k^{(1)} = (1, 1, \cdots, 1)$.

2. Increment the 'farthest from deadline segment' first, e.g., if τ_{k1} is the farthest,

$O_k^{(2)} = (2, 1, \cdots, 1)$, $O_k^{(3)} = (3, 1, \cdots, 1)$, and so on until τ_{k1} reaches the system maximum $(O^{\max}, 1, \cdots, 1)$.

3. Continue with the next farthest segment $(O^{\max}, 2, \cdots, 1)$,

4. ... until we finally reach $O_k^{(\max)} = (O^{\max}, O^{\max}, \cdots, O^{\max})$.

4.4.2 Assigning Parallelization Options to Multi-Segment Tasks

Within a given parallelization route, the monotonic increasing property of both tolerance and interference is preserved. Leveraging such properties, we can extend the parallelization option assignment algorithm presented in the previous chapter to the multi-segment tasks, i.e., (1) all tasks begin with the lowest parallelization, (2) iteratively increase parallelization according to the *parallelization route*, until all tasks can "barely tolerate" the received interference.

This algorithm can be formally described as the pseudo-codes in Algorithm 2. The algorithm takes the set of tasks Γ as input and produces the parallelization option combination $\mathbb{O}$, if any, that makes the given task set schedulable as output.

The **for**-loop from Line 1 to Line 7 pre-calculates the tolerance $tol(\tau_k(O_k))$ for each task $\tau_k \in \Gamma$ and for each option O_k using Eq. (4.3). In Line 8, we create an array $\mathbb{S}$ where each element $\mathbb{S}_k \in \mathbb{S}$ is initialized to *Unknown*, meaning the schedulability of τ_k is yet unknown. Then, Line 9 initializes the parallelization options to $\mathbb{O}^{\text{cur}} = (\{1, 1, \cdots, 1\}, \cdots, \{1, 1, \cdots, 1\})$ and Line 10 sets the "updated" flag as true.

Then, the **while**-loop from Line 11 to Line 26 iteratively increases the parallelization options to the barely tolerable ones until it turns out "schedulable" or "not schedulable". For this, in each iteration of the **while**-loop, Line 12 first resets the up-

dated flag to false, meaning that we will terminate the **while**-loop if no parallelization option is updated. Then Line 13 sets the previous parallelization option combination $\mathbb{O}^{\text{pre}}$ the same as the current option combination $\mathbb{O}^{\text{cur}}$.

Then, the **for**-loop from Line 14 to Line 24 tries to update each task τ_k's parallelization option O_k^{cur} to the barely tolerable ones. For this, within the **for**-loop, Line 15 calculates the total interference given to τ_k, i.e., $\sum_{\tau_i \in \Gamma(\tau_i \neq \tau_k)} int(\tau_i(O_i^{\text{pre}}), \tau_k)$, using Eq. (4.4) with the previous parallelization options.

Then, the **while**-loop from Line 16 to Line 23 checks whether τ_k's tolerance is smaller than the interference from other tasks. If so, Line 17 increases O_k^{cur} to the next one, and Line 18 sets the updated flag as true, indicating that some options are updated. This is repeated until the barely tolerable option is found.

In the meantime, however, if O_k^{cur} becomes larger than $\{O^{\text{max}}, \cdots, O^{\text{max}}\}$ as in Line 19, it means that τ_k's tolerance even with $\{O^{\text{max}}, \cdots, O^{\text{max}}\}$ is still smaller than the interference from other tasks and hence τ_k is unschedulable. In this case, Line 20 marks $\mathbb{S}_k$ as false and sets $O_k^{\text{cur}} = \{O^{\text{max}}, \cdots, O^{\text{max}}\}$. Line 21 then breaks the **while**-loop from Line 16 to Line 23 and goes back to Line 14 to continue with unchecked tasks for finding their barely tolerable options.

After checking all the tasks in the current iteration, Line 25 checks if there is a task that turns out unschedulable, and if so, it breaks the **while**-loop from Line 11 to Line 26. When the **while**-loop from Line 11 to Line 26 terminates, the **if-else** statement from Line 27 to Line 31 returns the "not schedulable" or "schedulable" conclusion. When the given task set is schedulable, it also returns the found parallelization option combination $\mathbb{O}^{\text{cur}}$.

Algorithm 2 Parallelization Option Assignment for Multi-Segment Tasks

Input: (1) Set of tasks $\Gamma = \{\tau_1, \tau_2, \cdots, \tau_n\}$

(2) Parallelization routes $\{\vec{O}_1, \vec{O}_2, ..., \vec{O}_n\}$

Output: (1) Schedulability,

(2) Parallelization option combination $\mathbb{O} = (O_1, O_2, \cdots, O_n)$

begin procedure

1. **for** $\tau_k \in \Gamma$ **do**
2. $O_k = \{1, 1, \cdots, 1\}$
3. **while** $O_k \neq \{O^{\max}, O^{\max}, \cdots, O^{\max}\}$
4. $tol(\tau_k(O_k)) \leftarrow$ Eq. (4.3)
5. $O_k \leftarrow$ next element of $\vec{O}_k$
6. **end while**
7. **end for**
8. initialize $\mathbb{S} \leftarrow (Unknown, Unknown, ..., Unknown)$
9. initialize $\mathbb{O}^{\text{cur}} \leftarrow (\{1, 1, \cdots, 1\}, \cdots, \{1, 1, \cdots, 1\})$
10. initialize updated $\leftarrow$ True
11. **while** updated **do**
12. updated $\leftarrow$ False
13. $\mathbb{O}^{\text{pre}} \leftarrow \mathbb{O}^{\text{cur}}$
14. **for** $\tau_k \in \Gamma$ **do**
15. $\sum_{\tau_i \in \Gamma, \tau_i \neq \tau_k} int(\tau_i(O_i^{\text{pre}}), \tau_k) \leftarrow$ Eq. (4.4)
16. **while** $tol(\tau_k(O_k^{\text{cur}})) < \sum_{\tau_i \in \Gamma(\tau_i \neq \tau_k)} int(\tau_i(O_i^{\text{pre}}), \tau_k)$ **do**
17. $O_k^{\text{cur}} \leftarrow$ next element of $\vec{O}_k$
18. updated $\leftarrow$ True
19. **if** $O_k^{\text{cur}} > \{O^{\max}, O^{\max}, \cdots, O^{\max}\}$ **then**
20. $\mathbb{S}_k \leftarrow False$ and $O_k^{\text{cur}} \leftarrow \{O^{\max}, \cdots, O^{\max}\}$
21. **break** // goto Line 14 to continue with next τ_k
22. **end if**
23. **end while**
24. **end for**

25. **if** any $\mathbb{S}_k \in \mathbb{S}$ is $False$ **then break end if**
26. **end while**
27. **if** any $\mathbb{S}_k \in \mathbb{S}$ is $False$ **then**
28. **return** not schedulable
29. **else**
30. **return** schedulable, $\mathbb{O}^{\text{cur}}$
31. **end if**

end procedure

4.4.3 Time complexity of Algorithm 2

Algorithm 2 conducts a uni-directional search in the outer **while**-loop (Lines 11-26) by leveraging the monotonic increasing property of both tolerance and interference. For n tasks, the maximum possible number of option increments is $n \cdot |\vec{O}_k|$. Because each S_k segments increments exactly $O^{\max} - 1$ times, thus $O(n \cdot |\vec{O}_k|) = O(n \cdot S_k \cdot (O^{\max} - 1)) = O(n \cdot S_k \cdot O^{\max})$. For each of the increment, we need to recalculate interference (Line 15), and this takes $O(n)$ for all other tasks within the **for**-loop from Lines 14-24, hence $O(n^2)$ complexity. Therefore the overall time complexity of the algorithm is $O(n^3 \cdot S_k \cdot O^{\max})$.

4.5 DAG (Directed Acyclic Graph) Task Model

We consider a system with m identical CPU cores and n sporadic DAG (Directed Acyclic Graph) tasks scheduled by G-EDF. Each sporadic DAG task is denoted $\tau_k (1 \leq k \leq n)$, and such set of tasks is represented as Γ as follows:

$$\Gamma = \{\tau_1, \tau_2, \cdots, \tau_n\}.$$

In the sporadic DAG task model, a task $\tau_k \in \Gamma$ is represented as a 3-tuple, i.e., $\tau_k = (G_k, D_k, T_k)$, where G_k is a directed acyclic graph shown in the upper part of Fig. 4.4, D_k is the relative deadline, and T_k is the minimum inter-release time. The DAG G_k is specified by $G_k = (V_k, E_k)$, where V_k is the set of n_k nodes, i.e.,

$$V_k = \{\tau_{k1}, \tau_{k2}, ..., \tau_{kn_k}\},$$

and E_k is a set of directed edges between nodes. An edge $(\tau_{kp}, \tau_{kq}) \in E_k$ represents the precedence relation between τ_{kp} and τ_{kq}, i.e., τ_{kq} can only start to execute after τ_{kp} finishes its execution. In such a case, τ_{kp} is a *predecessor* of τ_{kq}, and τ_{kq} is a *successor* of τ_{kp}.

A node without any *predecessor*/*successor* is called a *source*/*sink*, respectively. For example, in Fig. 4.4, τ_{k1} is a *source*, and τ_{k7} is a *sink*. Without loss of generality, we consider DAGs have exactly one *source* and one *sink*[†].

Enjoying the parallelization freedom, we consider that each node can be parallelized into a desirable number of threads[‡]. This is illustrated in Fig. 4.4, as τ_{k1} having a choice ranging from 1 to $O^{\max}$ threads, i.e., the system maximum. The choice of such a number of threads (2 in this case) is called the *parallelization option*, i.e., $O_{k1} = 2$, and the resulting *sibling threads* are notated as $\tau_{k1}(2) = \{\tau^1_{k1}(2), \tau^2_{k1}(2)\}$.

[†]DAGs with multiple sources or sinks can be converted to the considered form by adding dummy nodes with zero execution requirement[30].

[‡]Parallelization does not change the precedence relation of the nodes. A node can only start to execute after *all* threads of *all* the preceding nodes complete their execution.

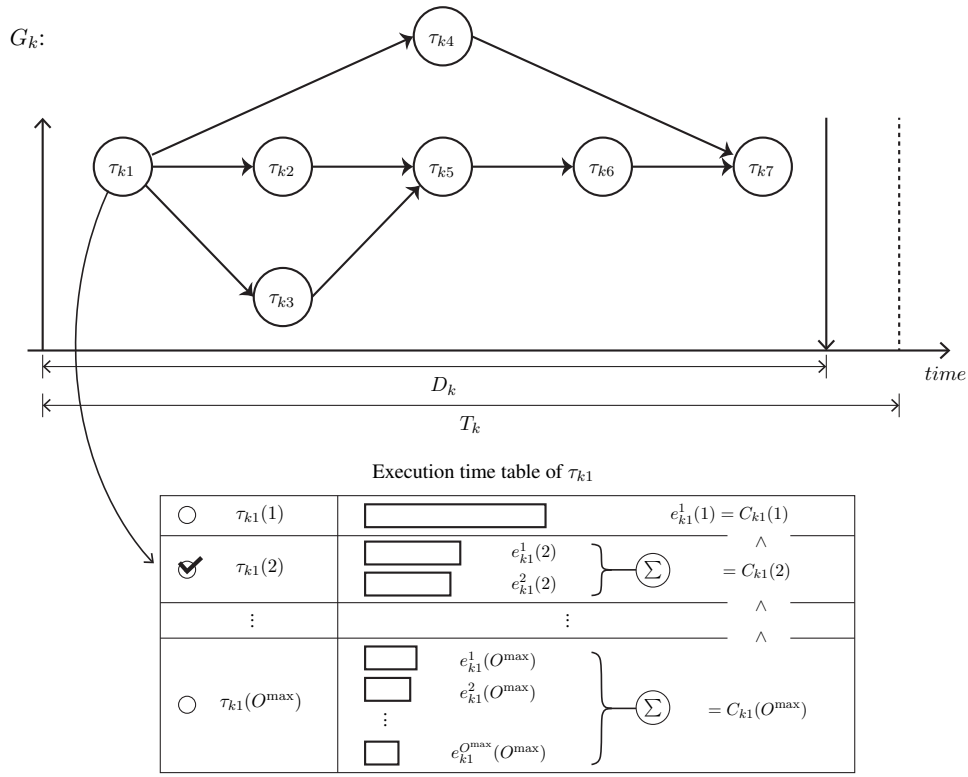


Figure 4.4: DAG task model with parallelization freedom

In a more general form, a node τ_{kp} parallelized into O_{kp} threads yields:

$$\tau_{kp}(O_{kp}) = \{\tau^1_{kp}(O_{kp}), \tau^2_{kp}(O_{kp}), ..., \tau^{O_{kp}}_{kp}(O_{kp})\},$$

where $\tau^l_{kp}(O_{kp})(1 \leq l \leq O_{kp})$ is the l-th sibling thread. Note that the above definition gets notation-heavy, and we may omit the subscript or the parentheses when no ambiguity arises.

The l-th sibling thread $\tau^l_{kp}(O_{kp})$ has the WCET (worst-case execution time) of $e^l_{kp}(O_{kp})$ as indicated in the execution time table in the lower part of Fig. 4.4. Without loss of generality, we sort the O_k threads of $\tau_{kp}(O_k)$ in descending order of WCET,

and hence $\tau^1_{kp}(O_{kp})$ has the largest WCET, i.e., $\max_{\tau^l_{kp} \in \tau_{kp}} e^l_{kp}(O_{kp}) = e^1_{kp}(O_{kp})$. The total computation amount of these O_k sibling threads is denoted as $C_{kp}(O_{kp}) = \sum_{l=1}^{O_{kp}} e^l_{kp}(O_{kp})$.

In general, a larger parallelization option $O^{'} (> O)$ makes each individual thread execution time smaller, i.e., $e^l_{kp}(O^{'}) < e^l_{kp}(O)$, but makes the computation amount larger, i.e., $C_{kp}(O^{'}) > C_{kp}(O)$, due to parallelization overhead.

Now looking back at G_k, we define a *path* λ as a sequence of nodes $[\tau_{k1}$, ..., τ_{kp}, τ_{kq}, ..., $\tau_{kn_k}]$ that starts from the *source* (τ_{k1}), ends at the *sink* (τ_{kn_k}). Every consecutive node in the path has a connected edge, i.e., $(\tau_{kp}, \tau_{kq}) \in E_k$. For example, in Fig. 4.4, $[\tau_{k1}, \tau_{k2}, \tau_{k5}, \tau_{k6}, \tau_{k7}]$ is one of the possible paths in G_k. The length of a path λ is defined as:

$$len(\lambda) = \sum_{\tau_{kp} \in \lambda} e^1_{kp}(O_{kp}),$$

which is the summation of the longest thread's WCET, i.e., the first thread, for all nodes of the path. The length of the longest path, or the *critical path* λ^*, which also defines the length of the task, is then given as:

$$len(\tau_k) = len(\lambda^*) = \max_{\lambda | G_k} len(\lambda),$$

and the consisting threads of such path are called the *critical threads*. Note that the longest path is not static; a path may vary as thread execution time changes with different parallelization options. Consequently, the critical threads can also change depending on the parallelization.

The sum of execution times of entire threads of τ_k, or the "total computation

amount" of τ_k, is called the volume of τ_k:

$$vol(\tau_k) = \sum_{\tau_{kp} \in \tau_k} C_{kp}.$$

Problem Definition: For each DAG tasks τ_k in the given task set $\Gamma = \{\tau_1, \tau_2, \cdots, \tau_n\}$, our problem is to find a parallelization option O_{kp} for its nodes τ_{kp}, such that all the tasks in Γ can be scheduled meeting their deadlines using global EDF on m identical CPU cores.

4.6 Extension of Chwa-DAG Schedulability Analysis

This section derives a schedulability analysis for DAG tasks with parallelization freedom scheduled with global EDF. In later sections, we build upon this schedulability analysis to assign a valid parallelization option to each node of the tasks.

We base our schedulability analysis on the state-of-the-art analysis targeted for sporadic DAG tasks, i.e., Chwa-DAG analysis [22]. For this, we extend the calculation of interference to consider threads from parallelized nodes correctly.

In the following sub-sections, we first provide an overview of Chwa schedulability analysis. Afterward, we show how it can be extended to determine the schedulability of tasks with parallelization freedom.

4.6.1 Chwa-DAG Schedulability Analysis

Chwa-DAG analysis is an interference-based schedulability analysis. Similar to other interference-based analyses, Chwa-DAG analysis formulates each task's worst possible execution scenario by calculating their worst-case received interference. Only

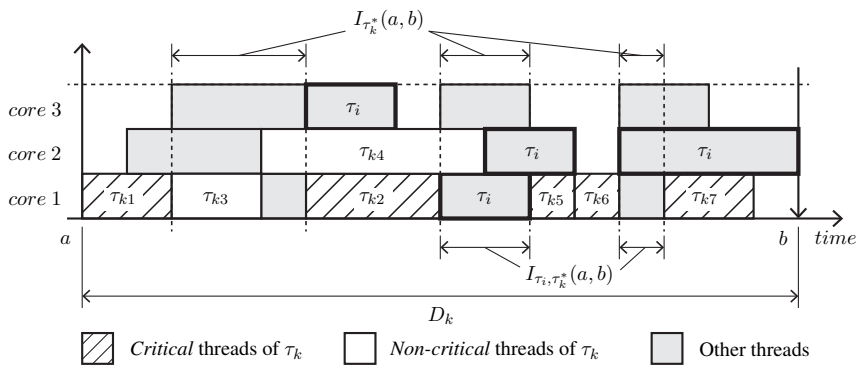


Figure 4.5: Critical interference received by τ_k

when all the tasks are schedulable in such conditions, the task set is deemed schedulable.

To apply such a method to the DAG task model, the notion of *critical interference* is introduced. Within an interval $[a, b)$, regarding an interfered task τ_k and an interfering task $\forall \tau_i \in \Gamma$,

1. $I_{\tau_k^*}(a, b)$: Cumulative length of all intervals in which a *critical thread* τ_k^* of τ_k is ready to execute, but it cannot execute due to higher priority threads in $[a, b)$.

2. $I_{\tau_i,\tau_k^*}(a, b)$: Cumulative length of all intervals in which τ_k^* is ready to execute but it cannot execute due to at least one thread of τ_i in $[a, b)$.

An example is depicted in Fig. 4.5 using the same DAG task τ_k from Fig. 4.4. To explain the above concept of critical interference, we assume all nodes of τ_k execute as a single thread, i.e., no-parallelization. Also, without loss of generality, we assume that the critical threads of τ_k are $\tau_k^* = \{\tau_{k1}, \tau_{k2}, \tau_{k5}, \tau_{k6}, \tau_{k7}\}$, identified as striped boxes in the figure. Consequently, the non-critical threads of τ_k are $\{\tau_{k3}, \tau_{k4}\}$, and are drawn as unfilled boxes. On the other hand, the filled boxes represent other co-

running tasks in the system. Among them, τ_i's threads are indicated with thick boxes.

As marked in the figure, $I_{\tau_k^*}(a,b)$ involves all the intervals in which τ_k^* is ready, but other higher priority threads occupy all CPU cores. Out of the cumulative intervals in $I_{\tau_k^*}(a,b)$, $I_{\tau_i,\tau_k^*}(a,b)$ involves only the intervals where at least one of interfering task's threads ($\tau_i \in \Gamma$) occupy the CPU cores. Note that the non-critical threads of τ_k can also affect τ_k^*'s execution and thus are counted as interference.

From these definitions, $I_{\tau_k^*}(a,b)$ can be calculated by adding up $I_{\tau_i,\tau_k^*}(a,b)$ from all the interfering tasks, including the non-critical threads, then dividing the sum by m, i.e.,

$$I_{\tau_k^*}(a,b) = \frac{1}{m} \sum_{\tau_i \in \Gamma, (\tau_i \neq \tau_k^*)} I_{\tau_i,\tau_k^*}(a,b). \tag{4.6}$$

Using this, a schedulability condition can be derived for the critical threads of τ_k. Recall that the length of the *critical path* is given as $len(\lambda^*) = len(\tau_k)$. For the critical threads to not miss the deadline, they must be granted at least $len(\tau_k)$ CPU time before the deadline D_k. Applying this to Eq. (4.6) yields the following schedulability condition:

$$I_{\tau_k^*}(a,b) = \frac{1}{m} \sum_{\tau_i \in \Gamma, (\tau_i \neq \tau_k^*)} I_{\tau_i,\tau_k^*}(a,b) \leq D_k - len(\tau_k). \tag{4.7}$$

It is shown in [22] that τ_k^* represents τ_k's schedulability, or in other words, when τ_k^* is schedulable, its original task τ_k is also schedulable.

However, because of the sheer number of possible execution scenarios of the sporadic tasks, the calculation of $I_{\tau_i,\tau_k^*}(a,b)$ becomes computationally intractable. Instead, Chwa-DAG analysis adopts an upper bound of the interference, leveraging

the fact that τ_i cannot interfere τ_k^* more than its worst-case workload W_{τ_i,τ_k^*} within the same interval, i.e., $I_{\tau_i,\tau_k^*}(a,b) \leq W_{\tau_i,\tau_k^*}$.

Fig. 4.6 shows a DAG task τ_i's graph G_i and its worst-case release pattern when it interferes with τ_k^*'s execution. To illustrate how the worst-case release pattern is formulated, we introduce the following two terms: (1) *carry-in* job: a job of τ_i released before but finishes within the considered interval $[a,b)$, and (2) *body* jobs: jobs of τ_i that is both released and finished within the considered interval $[a,b)$. As indicated in the figure, the worst-case release pattern of τ_i where W_{τ_i,τ_k^*} can be maximized happens when (1) τ_i's last *body* job's deadline coincides with the deadline of τ_k's job, (2) previous jobs of τ_i are released as frequently as possible, i.e., with its minimum inter-release time T_i, and (3) all threads of the *carry-in* job are executed as late as possible. Then W_{τ_i,τ_k^*} can be represented as a sum of the workload of the body jobs ($W^{BD}_{\tau_i,\tau_k^*}$) and the carry-in job ($W^{CI}_{\tau_i,\tau_k^*}$):

$$W_{\tau_i,\tau_k} = W^{BD}_{\tau_i,\tau_k^*} + W^{CI}_{\tau_i,\tau_k^*}. \tag{4.8}$$

The workload from the body jobs $W^{BD}_{\tau_i,\tau_k^*}$ can be calculated by measuring the number of occurring instances of the body job within the interval $[a,b)$ and multiplying it by the total workload of τ_i, which is $vol(\tau_i)$:

$$W^{BD}_{\tau_i,\tau_k^*} = \left\lfloor \frac{D_k}{T_i} \right\rfloor vol(\tau_i). \tag{4.9}$$

We note that the non-critical threads of τ_k also contribute workload, i.e., W_{τ_k,τ_k^*}. Since the critical and non-critical threads are from the same task, the non-critical

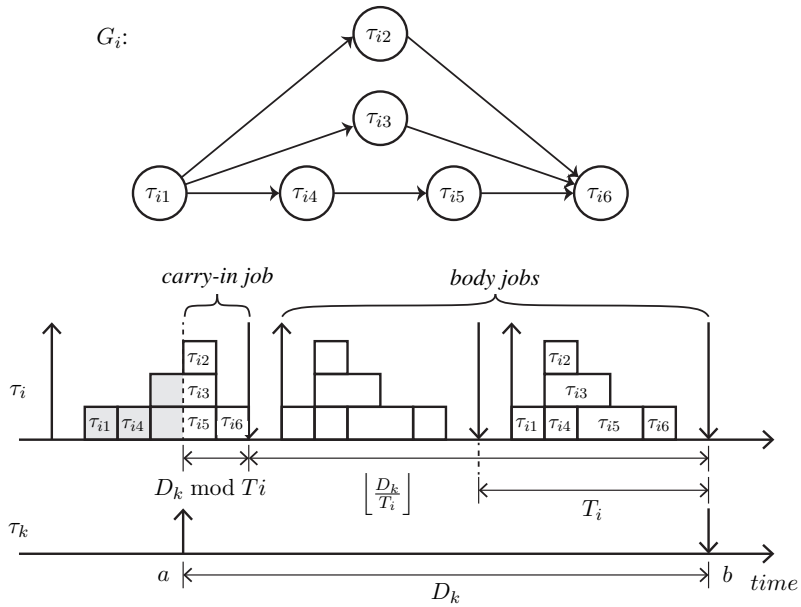


Figure 4.6: Worst-case release pattern of τ_i within D_k

threads can only contribute a single body job. Thus, its value is:

$$W_{\tau_k,\tau_k^*} = W^{BD}_{\tau_k,\tau_k^*} = vol(\tau_k) - len(\tau_k). \tag{4.10}$$

On the other hand, as depicted in Fig. 4.6, $W^{CI}_{\tau_i,\tau_k}$ is calculated by summing up the maximum possible inclusion of τ_i inside the carry-in interval. This is when every thread of τ_i is packed as much as possible towards its deadline, i.e., the finish time of every thread is set to the fastest start time among its successors. As a result, individual threads $\tau_{ip} \in \tau_i$ contribute carry-in by:

1. C_{ip}, for entire inclusion, e.g., τ_{i2}, τ_{i6},
2. $f[\tau_{ip}] - (D_i - (D_k \bmod T_i))$, partial inclusion, e.g., τ_{i3}, τ_{i5},
3. zero, if not included at all, e.g., τ_{i1}, τ_{i4},

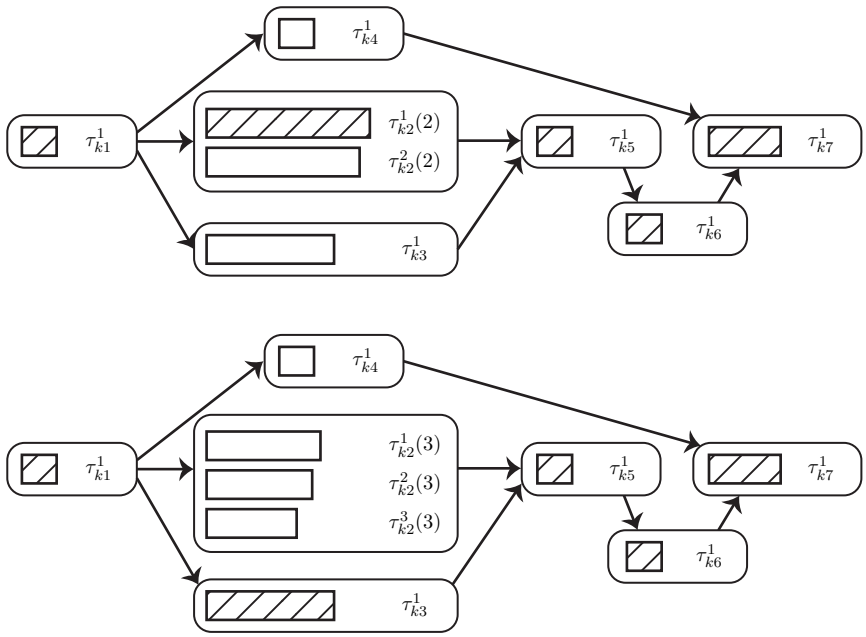


Figure 4.7: τ_k with different parallelization option $O_{k2} = 2, 3$

where $f[\tau_{ip}]$ denotes the finish time of τ_{ip}. Portions that are not included are shaded in gray in the figure. We note that it is assumed here that threads can use as many processors as possible for their execution when calculating $W^{CI}_{\tau_i,\tau^*_k}$.

With the calculated worst-case workload $W^{BD}_{\tau_i,\tau^*_k}$ and $W^{CI}_{\tau_i,\tau^*_k}$, we can now derive a workload-based schedulability condition from Eq. (4.7). A DAG task τ_k is schedulable when the following condition is met:

$$\sum_{\tau_i \in \Gamma, (\tau_i \neq \tau^*_k)} \left(W^{BD}_{\tau_i,\tau^*_k} + W^{CI}_{\tau_i,\tau^*_k} \right) \leq m(D_k - len(\tau_k)). \tag{4.11}$$

4.6.2 Tolerance and Interference of DAG Tasks

The scheduling condition presented in Eq. (4.11), i.e., Chwa-DAG analysis, considers each node is run as a single thread. In this sub-section, we present an extension of

Chwa-DAG analysis to consider DAG tasks with parallelization freedom correctly.

With parallelization freedom, each node can be parallelized into a desirable number of threads. Refer to the upper half of Fig. 4.7, as the second node of τ_k is split into two sibling threads $\tau^1_{k2}(2)$ and $\tau^2_{k2}(2)$. The length of the boxes represents the execution time of each thread. As shown in the figure, $\tau^1_{k2}(2)$ has a longer execution time than τ^1_{k3}, i.e., $e^1_{k2}(2) > e^1_{k3}$. Therefore, the critical path is identified as $[\tau^1_{k1}, \tau^1_{k2}(2), \tau^1_{k5}, \tau^1_{k6}, \tau^1_{k7}]$, as indicated as diagonally striped boxes in the figure. Then the length of the critical path is the sum of WCET of those threads:

$$len(\tau_k) = len(\lambda^*) = e^1_{k1} + e^1_{k2}(2) + e^1_{k5} + e^1_{k6} + e^1_{k7}.$$

As a consequence, the remaining sibling thread, i.e., $\tau^2_{k2}(2)$, becomes a non-critical thread. Therefore, $\tau^2_{k2}(2)$ must be counted as interfering workload when calculating Eq. (4.10).

Next, let us consider the case where τ_{k2} is parallelized even more, becoming three threads: $\tau^1_{k2}(3)$, $\tau^2_{k2}(3)$, and $\tau^3_{k2}(3)$. The situation is presented in the lower half of Fig. 4.7. As individual thread execution time decreases for τ_{k2} due to parallelization, $\tau^1_{k2}(3)$ became shorter than τ^1_{k3}, i.e., $e^1_{k2}(3) < e^1_{k3}$. This alters the critical path to $[\tau^1_{k1}, \tau^1_{k3}, \tau^1_{k5}, \tau^1_{k6}, \tau^1_{k7}]$. Consequently, all threads of τ_{k2} are now non-critical and therefore treated as interfering workload.

To formally characterize this behavior, we define the notion of *tolerance* of task $\tau_k(O_k)$ as follows:

$$tol(\tau_k(O_k)) \coloneqq m(D_k - len(\tau_k(O_k))) - W_{\tau_k(O_k),\tau^*_k(O_k)}, \tag{4.12}$$

where O_k is the collection of chosen parallelization options of each node of $\tau_k(O_k)$, i.e.,

$$O_k = \{O_{k1}, O_{k2}, \cdots, O_{kn_k}\}.$$

In other words, tolerance can be thought of as the limit of the workload $\tau_k(O_k)$ can receive from all other tasks $\tau_i \neq \tau_k$ for $\tau_k(O_k)$ to remain schedulable. This definition is derived from Eq. (4.11), where we move the non-critical threads' contribution of workload, i.e., $W_{\tau_k(O_k),\tau_k^*(O_k)}$, to the right-hand side.

We also define the order between two option sets O_k and O'_k. O_k is strictly larger than O'_k, i.e., $O_k > O'_k$, when:

$$(1)\ \forall p,\ O_{kp} \geq O'_{kp},\ \text{ and }\ (2)\ \exists q,\ O_{kq} > O'_{kq}. \tag{4.13}$$

In other words, all nodes of τ_k must have greater or equal options than their corresponding τ'_k node's option, and there must be at least one node of τ_k that has a greater option than its τ'_k counterpart.

We present the following property of parallelization of a DAG task using the definition of tolerance, which helps us find the optimal parallelization options by a uni-directional search rather than a back-and-forth combinatorial search. The following lemma and corollary formally state this:

Lemma 4. The tolerance $tol(\tau_k(O_k))$ in Eq. (4.12) is a monotonic increasing function of O_k, i.e., when $O_k < O'_k$, $tol(\tau_k(O_k)) \leq tol(\tau_k(O'_k))$, under ideal parallelization with zero parallelization overhead, i.e., $\forall p,\ C_{kp}(O_{kp}) = C_{kp}(O'_{kp})$.

Proof. We prove for consecutive O_k and O'_k, when only a single node τ_{kp}'s parallelization is increased by one, i.e., $O'_{kp} = O_{kp} + 1$ and $O'_{kq} = O_{kq}$ $(\forall q \neq p)$. Such proof is sufficient because we can always get from any O_k to another O'_k $(O_k < O'_k)$ with series of consecutive increments.

Next, using Eq. (4.10), we can express the definition of tolerance using the length and the volume of $\tau_k(O_k)$:

$$tol(\tau_k(O_k)) = \ m\big(D_k - len(\tau_k(O_k))\big) \ - \big(vol(\tau_k(O_k)) - len(\tau_k(O_k))\big). \quad (4.14)$$

Now we consider the following two cases of parallelization for node τ_{kp}, depending on $\tau^1_{kp}(O_k)$'s criticality: ***case (1)*** when it is a non-critical thread, or ***case (2)*** when it is a critical thread.

case (1): If $\tau^1_{kp}(O_k)$ is a non-critical thread, $\tau^1_{kp}(O_k + 1)$ must also be a non-critical thread because parallelization only decreases the WCET of each thread, i.e., $e^1_{kp}(O_k) > e^1_{kp}(O_k{+}1)$. Hence critical path, i.e., set of longest threads, is not affected.

Therefore, neither $vol(\tau_k(O_k))$ nor $len(\tau_k(O_k))$ is changed upon the increase of parallelization, and this results in identical tolerance for O_k and $O_k + 1$, that is, $tol(\tau_k(O_k)) = tol(\tau_k(O_k + 1))$.

case (2): If $\tau^1_{kp}(O_k)$ is a critical thread, we divide the case further into two sub-cases: ***sub-case (2-1)*** $\tau^1_{kp}(O_k + 1)$ remains a critical thread, and ***sub-case (2-2)*** $\tau^1_{kp}(O_k + 1)$ is no longer a critical thread.

sub-case (2-1): If $\tau^1_{kp}(O_k + 1)$ is a critical thread, the length of the critical path shrinks:

$$len(\tau_k(O_k + 1)) - len(\tau_k(O_k)) = e^1_{kp}(O_k + 1) - e^1_{kp}(O_k) < 0.$$

Under ideal parallelization, $vol(\tau_k(O_k))$ is the same for both O_k and O_k+1. According to Eq. (4.14), tolerance increases as $len(\tau_k(O_k))$ decreases. Therefore tolerance increases for O_k and $O_k + 1$ for this sub-case.

sub-case (2-2): If $\tau^1_{kp}(O_k + 1)$ is not a critical thread, a different critical path has a longer length. Let us denote the original path as $\lambda(\tau^1_{kp}(O_k))$ and the resulting non-critical path created from replacing $\tau^1_{kp}(O_k)$ with $\tau^1_{kp}(O_k + 1)$ as $\lambda(\tau^1_{kp}(O_k + 1))$. Also, let us denote the new critical path λ^*. Then we know from ***sub-case (2-1)*** that $\lambda(\tau^1_{kp}(O_k + 1)) < \lambda(\tau^1_{kp}(O_k))$, and we know that $\lambda^* < \lambda(\tau^1_{kp}(O_k + 1))$. By transitivity, $\lambda^* < \lambda(\tau^1_{kp}(O_k))$. Thus from Eq. (4.14), tolerance increases as well for this sub-case.

In all cases, tolerance is increased or remains the same. The lemma follows. □

This lemma still holds with parallelization overhead considered, with the following restrictions: (1) only the critical threads are parallelized, and (2) the decrease of the critical chain length outpaces the parallelization overhead, i.e.,

$$len(\tau_k(O_{kp} + 1)) - len(\tau_k(O_{kp})) > \frac{1}{1-m}\Big(C_k(O_{kp} + 1) - C_k(O_{kp})\Big) \quad (4.15)$$

We formally state this in the following corollary:

Corollary 2. If τ_{kp} is a critical thread and condition Eq. (4.15) holds, Lemma 4 still holds.

Proof. We can calculate the difference of tolerance of O_{kp} and $O_{kp}+1$ using Eq. (4.14):

$$
\begin{aligned}
&tol(\tau_k(O_{kp}+1)) - tol(\tau_k(O_{kp})) = \\
&\qquad (1-m)\{len(\tau_k(O_{kp}+1)) - len(\tau_k(O_{kp}))\} \\
&\qquad -\{vol(\tau_k(O_{kp}+1)) - vol(\tau_k(O_{kp}))\}.
\end{aligned} \tag{4.16}
$$

Considering parallelization overhead, the total execution time of $\tau_k(O_{kp})$ is increased by the excess WCET, i.e., $vol(\tau_k(O_{kp}+1)) - vol(\tau_k(O_{kp})) = C_k(O_{kp}+1) - C_k(O_{kp})$. Substituting this to Eq. (4.16) yields the following:

$$
(1-m)\{len(\tau_k(O_{kp}+1)) - len(\tau_k(O_{kp}))\} - \{C_k(O_{kp}+1) - C_k(O_{kp})\},
$$

which according to condition Eq. (4.15) is positive. Thus the tolerance increases, i.e., $tol(\tau_k(O_{kp}+1)) > tol(\tau_k(O_{kp}))$. □

Now we know from Lemma 4 and Corollary 2 that increasing parallelization option O_k increases tolerance of task τ_k. This allows τ_k to endure more interfering workload from other tasks, which is advantageous for its own schedulability. However, this increase of τ_k's parallelization may hinder the schedulability of other tasks $\tau_i \neq \tau_k$.

Fig. 4.8 shows τ_k with two different parallelizations interfering with τ_i's execution. In the upper part of the figure, τ_k's second node is parallelized into two threads $\{\tau^1_{k2}(2), \tau^1_{k2}(2)\}$. The same node is parallelized into three threads $\{\tau^1_{k2}(3), \tau^2_{k2}(3), \tau^3_{k2}(3)\}$, as shown in the lower part of the figure. A worst-case release scenario for τ_i is depicted for both cases, and we zoomed in on the carry-in job portion for clarity.

As shown in the figure, some part of τ_k's workload is included in τ_i's execution interval, while others are not. To simplify the discussion, we call this "interfering

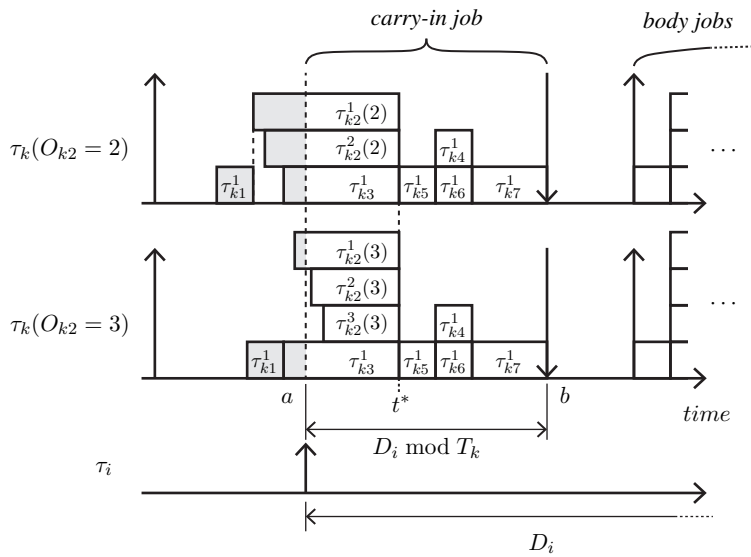


Figure 4.8: Carry-in of τ_k with different parallelization $O_{k2} = 2, 3$

workload of τ_k inside τ_i's execution interval", and abbreviated as the "interference" of τ_k to τ_i, using the following notation:

$$int(\tau_k, \tau_i) = W_{\tau_k, \tau_i^*} = W^{BD}_{\tau_k, \tau_i^*} + W^{CI}_{\tau_k, \tau_i^*} \tag{4.17}$$

Looking at the figure, we can see that more portions of τ_{k2} get included inside interval $[a, b)$ in the higher parallelization case. This is because τ_k's worst-case release scenario requires all threads of its carry-in job to execute as late as possible. As a result, all sibling threads are considered to have identical finish times, indicated as t^* in the figure. Therefore as parallelization increases for node τ_{k2}, more threads are packed towards t^*, punching in greater interference to τ_i. We discuss this effect formally in the following lemma. Later, this will also help us construct a uni-directional parallelization option assignment:

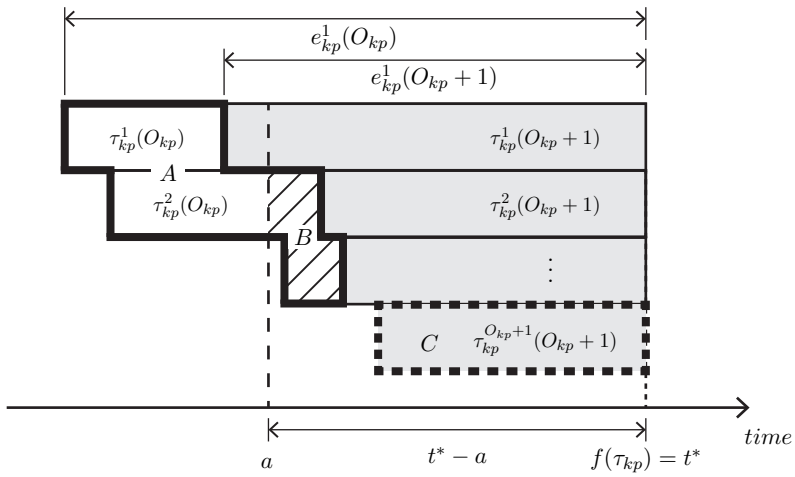


Figure 4.9: Workload of τ_{kp} with parallelization $O_{kp}, O_{kp}+1$

Lemma 5. Interference from τ_k to τ_i, i.e., $int(\tau_k, \tau_i)$, is a monotonic increasing function of O_k, i.e., when $O_k < O'_k$, $int(\tau_k(O'_k), \tau_i) \leq int(\tau_k(O_k), \tau_i)$.

Proof. Similar to Lemma 4, we prove for consecutive O_k and O'_k, when only a single node τ_{kp}'s parallelization is increased by one, i.e., $O'_{kp} = O_{kp} + 1$ and $O'_{kq} = O_{kq}$ $(\forall q \neq p)$.

According to Eq. (4.17), interference of τ_k to τ_i is split into carry-in and body job, i.e.,

$$int(\tau_k, \tau_i) = W^{BD}_{\tau_k,\tau^*_i} + W^{CI}_{\tau_k,\tau^*_i}$$

Because parallelization of a node does not change the number of body jobs, $W^{BD}_{\tau_k,\tau^*_i}$ remains the same. In fact, considering parallelization overhead, $W^{BD}_{\tau_k,\tau^*_i}$ may only increase. Therefore, if $W^{CI}_{\tau_k,\tau^*_i}$ monotonically increases, W_{τ_k,τ^*_i} monotonically increases as well. For this, in the remainder of the proof, we focus only on the carry-in job's workload, i.e., $W^{CI}_{\tau_k,\tau^*_i}$. The carry-in interval is denoted $[a, b)$, as indicated in

Fig. 4.8.

The worst-case release scenario of τ_k considers each thread of the carry-in job executes as late as possible. In such a case, τ_{kp}'s finish time, i.e., $f(\tau_{kp})$, is decided by the starting time of its successors. Since this is invariant to parallelization, $f(\tau_{kp})$ is identical for both O_k and $O_k + 1$. For example, in Fig. 4.8 we marked $f(\tau_{k2})$ as t^*, notice t^* is the same for $O_{k2} = 2, 3$.

Now we consider the following cases of parallelization of node τ_{kp}, depending on $f(\tau_{kp})$'s relative position to the carry-in interval $[a, b)$: ***case (1)*** $f(\tau_{kp}) \leq a$ and ***case (2)*** $a < f(\tau_{kp}) < b$.

case (1): If $f(\tau_{kp}) \leq a$, then no thread of both $\tau_{kp}(O_k)$ or $\tau_{kp}(O_k + 1)$ is counted as workload. Therefore, in this case W_{τ_k, τ_i^*} remains the same.

case (2): For case $a < f(\tau_{kp}) < b$, refer to Fig. 4.9, where we show a generalized example case based on τ_{k2} from the previous figure (Fig. 4.8). In this example, τ_{kp} is parallelized from O_{kp} to $O_{kp}+1$, where the resulting threads $\tau_{kp}(O_{kp})$ and $\tau_{kp}(O_{kp}+1)$ are drawn as unfilled and filled boxes, respectively. All threads are drawn aligned to the right, i.e., $f(\tau_{kp}) = t^*$, and the threads of $\tau_{kp}(O_{kp} + 1)$ are drawn over the threads of $\tau_{kp}(O_{kp})$, when the thread index is the same, i.e., $\tau_{kp}^1(O_{kp} + 1)$ overlaps $\tau_{kp}^1(O_{kp})$, and so on. $\tau_{kp}(O_{kp})$ has O_{kp} threads, so the last thread of $\tau_{kp}(O_{kp} + 1)$, i.e., $\tau_{kp}^{O_{kp}+1}(O_{kp} + 1)$ does not overlap any thread of $\tau_{kp}(O_{kp})$. We indicated the combined portion of $\tau_{kp}(O_{kp})$, where $\tau_{kp}(O_{kp})$ is longer than $\tau_{kp}(O_{kp} + 1)$, with thick solid lines and marked the portion as A in the figure.

From the definition of the carry-in workload, all part of threads inside interval $[a, t^*)$ is considered interference. As we can see from the figure, for $\tau_{kp}(O_{kp} + 1)$, the diagonally striped portion (indicated as B) was no longer included, compared

to $\tau_{kp}(O_{kp})$. However, the part inside the thick dashed box (C) was additionally included. By comparing the value of B and C, we can know in which case more workload was included.

For ideal parallelization, the combined WCET must be the same, i.e., $C_{kp}(O_{kp}) = C_{kp}(O_{kp} + 1)$, we know that $A = C$. In fact, when considering parallelization overhead, combined WCET increases, i.e., $C_{kp}(O_{kp}) \leq C_{kp}(O_{kp} + 1)$, which makes $A \leq C$. Also, because B is included in A, $B \leq A$. Finally, by transitivity, $B \leq C$. Therefore, more workload is included inside interval $[a, t^*)$ as parallelization increases.

In all the cases, interference is increased or remains the same. The lemma follows.

□

Wrapping up the above discussions, the schedulability condition of $\tau_k(O_k)$ can be rewritten using the monotonic increasing functions of tolerance and interference as follows:

$$\sum_{\tau_i \in \Gamma, (\tau_i \neq \tau_k(O_k))} int(\tau_i, \tau_k(O_k)) \leq tol(\tau_k(O_k)) \tag{4.18}$$

4.7 Assigning Parallelization Options to DAG Tasks

4.7.1 Parallelization Route for DAG Task Model

The monotonic increasing property of both tolerance and interference derived from the extended analysis helps us assign of parallelization option to DAG tasks. However, we still face a similar problem from multi-segment tasks: the "increase of the parallelization" can be applied to any node.

Thus we similarly specify the exact trace of the increase as parallelization *route*

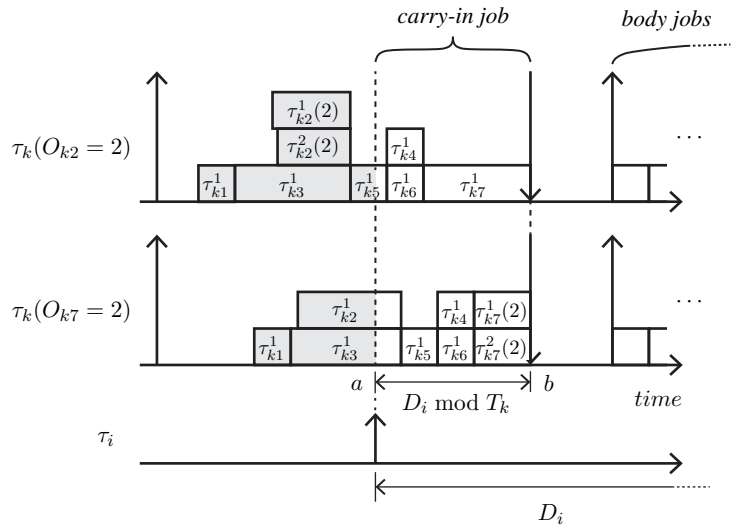


Figure 4.10: Carry-in job of τ_k with $O_{k2} = 2$ or $O_{k7} = 2$

$\vec{O}_k$. $\vec{O}_k$ is a *strictly totally ordered* subset of all possible parallelization of τ_k:

$$\vec{O}_k = \left(O_k^{(1)}, O_k^{(2)}, ..., O_k^{(|\vec{O}_k|)}\right), \tag{4.19}$$

where $O_k^{(i)}$ is a parallelization option for τ_k, i.e.,

$$O_k^{(i)} = \{O_{k1}^{(i)}, O_{k2}^{(i)}, \cdots, O_{kn_k}^{(i)}\},$$

and elements of $\vec{O}_k$ strictly increases according to Eq. (4.13), i.e.,

$$\forall (i < j), \;\; O_k^{(i)} < O_k^{(j)}.$$

A similar heuristic selection of $\vec{O}_k$ that aims to reduce interference contribution to other tasks is applied to DAG tasks. Intuitively speaking, we leverage the fact that the

carry-in workload is included starting from the deadline, so we preferably increment the nodes farther away from the deadline.

To see how this works, refer to Fig. 4.10, where we show τ_k interfering τ_i's execution. We show two cases where a different node of τ_k, i.e., τ_{k2} or τ_{k7}, is parallelized. Notice that compared to τ_{k7}, τ_{k2} is farther away from the deadline (indicated as b in the figure). We can infer from the figure that parallelization of τ_{k2} results in a much smaller carry-in interference to τ_i.

This is mainly due to two reasons. First, recall from Fig. 4.6 of Section 4.6 that the carry-in workload is calculated considering the worst-case finish time of each node. Upon parallelizing a node, all of its predecessors' finish times are also shifted forward. This "packing towards deadline" effect is cascaded upwards to the source node. To minimize this effect, we should pick the node with the smallest combined number of predecessors, the node farthest from the deadline.

The second reason is also due to the difference in the finish time. If the finish time is before the start of the carry-in interval, i.e., $[a, b)$, the parallelized threads are not carry-in workload. Notice from the upper part of the figure where τ_{k2} is parallelized, both τ_{k2}'s threads are not included as carry-in (a no-cost parallelization). Additionally, if the finish time is inside $[a, b)$ but the start time is before the interval, only a portion of the thread is included, i.e., $f[\tau_{kp}] - (D_k - (D_i \bmod T_k))$. Although not as dramatic, some part of the thread is excluded from the carry-in. Hence picking the farthest from the deadline node for parallelization is a superior strategy.

Keeping these in mind, we can build parallelization route $\vec{O}_k$ as the following:

1. Start from the lowest parallelization for all nodes, i.e., $O_k^{(1)} = (1, 1, \cdots, 1)$.

2. Increment the 'farthest from deadline node' first, e.g., if τ_{k1} is the farthest,

$O_k^{(2)} = (2, 1, \cdots, 1)$, $O_k^{(3)} = (3, 1, \cdots, 1)$, and so on until τ_{k1} reaches the system maximum $(O^{\max}, 1, \cdots, 1)$.

3. Continue with the next farthest node $(O^{\max}, 2, \cdots, 1)$.

4. ... until we finally reach $O_k^{(\max)} = (O^{\max}, O^{\max}, \cdots, O^{\max})$.

4.7.2 Assigning Parallelization Options to DAG Tasks

Within a given parallelization route, the monotonic increasing property of both tolerance and interference is preserved. Leveraging such properties, we can extend the parallelization option assignment algorithm presented in the previous chapter to the DAG tasks, i.e., (1) all tasks begin with the lowest parallelization, (2) iteratively increase parallelization according to the *parallelization route*, until all tasks can "barely tolerate" the received interference.

This algorithm can be formally described with the pseudo-code in Algorithm 3. The algorithm takes a task set Γ and each task's parallelization route $\vec{O}_k$ as input and outputs the schedulability of the given task set. If the given task set turns out to be schedulable, the algorithm also outputs the parallelization option combination $\mathbb{O}$ that made it schedulable.

From Lines 1-6, the algorithm creates the initial setting. To do so, we first precalculate tolerance using Eq. (4.12) for all tasks for every parallelization option (Lines 1-5). Immediately (Line 6), we set all task's current option $\mathbb{O}^{\text{cur}}$ to the lowest option, i.e., $O_k^{(1)}$.

Next, in Line 7-22, we iteratively calculate interference and increase options. We first set the 'updated' flag to true (Line 7) and enter the **while** loop (Line 8-21), which

will spin until we arrive at a scheduling decision. Afterward, we reset the 'updated' flag to false (Line 9), and store the current options in $\mathbb{O}^{\text{pre}}$ (Line 10).

Then for each task (Lines 11-20), we calculate its received interference from others using Eq. (4.17) (Line 12). We then compare the tolerance and received interference (Line 13) and try to increase the option to make the task barely tolerable (Line 13-19). However, if the task was already at the max option (Line 14), we terminate the algorithm and output a 'not schedulable' decision (Line 15). Otherwise, the option is successfully incremented (Line 17), and we set the updated flag to true (Line 18) so that we can start another round of iteration.

After checking all tasks, and all tasks turn out to be schedulable, the **while** loop terminates. We return the 'schedulable' decision (Line 22) with the corresponding parallelization options $\mathbb{O}^{\text{cur}}$.

4.7.3 Time Complexity of Algorithm 3

Algorithm 3 conducts a uni-directional search in the outer **while**-loop (Lines 8-21) by leveraging the monotonic increasing property of both tolerance and interference. For n tasks, the maximum possible number of option increments is $n \cdot |\vec{O}_k|$. As will be shown in the next sub-section, each n_k nodes increments exactly $O^{\max} - 1$ times, thus $O(n \cdot |\vec{O}_k|) = O(n \cdot n_k \cdot (O^{\max} - 1)) = O(n \cdot n_k \cdot O^{\max})$. For each of the increment, we need to recalculate interference (Line 12), and this takes $O(n)$ for all other tasks within the **for**-loop from Lines 11-20, hence $O(n^2)$ complexity. Therefore the overall time complexity of the algorithm is $O(n^3 \cdot n_k \cdot O^{\max})$.

Algorithm 3. Parallelization Option Assignment for DAG Tasks

Input: (1) Task set $\Gamma = \{\tau_1, \tau_2, \cdots, \tau_n\}$,
(2) Parallelization routes $\{\vec{O}_1, \vec{O}_2, ..., \vec{O}_n\}$

Output: (1) Schedulability,
(2) Parallelization option combination $\mathbb{O} = (O_1, O_2, \cdots, O_n)$

begin procedure

for $\tau_k \in \Gamma$ **do**
  for $O_k \in \vec{O}_k$ **do**
    $tol(\tau_k(O_k)) \leftarrow$ Eq. (4.12)
  end for
end for
initialize $\mathbb{O}^{\text{cur}} \leftarrow (O_1^{(1)}, O_2^{(1)}, ..., O_n^{(1)})$
initialize updated $\leftarrow$ True
while updated **do**
  updated $\leftarrow$ False
  $\mathbb{O}^{\text{pre}} \leftarrow \mathbb{O}^{\text{cur}}$
  for $\tau_k \in \Gamma$ **do**
    $\sum_{\tau_i \in \Gamma, (\tau_i \neq \tau_k)} int(\tau_i(O_i^{\text{pre}}), \tau_k(O_k^{\text{pre}})) \leftarrow$ Eq. (4.17)
    while $tol(\tau_k(O_k^{\text{pre}})) < \sum_{\tau_i \in \Gamma, (\tau_i \neq \tau_k)} int(\tau_i(O_i^{\text{pre}}), \tau_k(O_k^{\text{pre}}))$ **do**
      if $O_k^{\text{cur}} = O_k^{(\max)}$ **then** // reached max option
        return not schedulable
      end if
      $O_k^{\text{cur}} \leftarrow$ next element of $\vec{O}_k$
      updated $\leftarrow$ True
    end while
  end for
end while
return schedulable, $\mathbb{O}^{\text{cur}}$

end procedure

4.8 Experiment Results: Multi-Segment Task Model

For the evaluation of Algorithm 2 on the multi-segment task model, τ_k was synthesized in the following manner: (1) T_k is randomly drawn from uniform$[1000, 5000]$, (2) D_k is randomly selected from uniform$[1000, T_k]$, (3) number of segments S_k randomly selected from uniform$[1, 8]$, and (4) for each segments, the thread execution time table E_{ks} is created starting from $e^1_{ks}(O_{ks} = 1) =$ uniform$[200, 800]$, $C_{ks}(O_k + 1)$ where $O_{ks} < O^{\max}$ is computed using $\alpha(O_{ks}, (O_{ks}+1)) = \frac{C_{ks}(O_{ks}+1) - C_{ks}(O_{ks})}{e^1_{ks}(O_{ks}) - e^1_{ks}(O_{ks}+1)}$, $C_{ks}(O_{ks} + 1)$ is divided into $(O_{ks} + 1)$ pieces by Unifast [15] algorithm, and those pieces are sorted in the descending order and assigned to $e^1_{ks}(O_{ks} + 1), e^2_{ks}(O_{ks} + 1), \cdots, e^{O_{ks}+1}_{ks}(O_{ks} + 1)$.

Using this way of generating a synthetic task, 10^5 task sets containing different number of tasks n and with different task set utilization $\sum_{k=1}^{n} U_k = \sum_{k=1}^{n} C_k(1)/T_k$ was generated as follows:

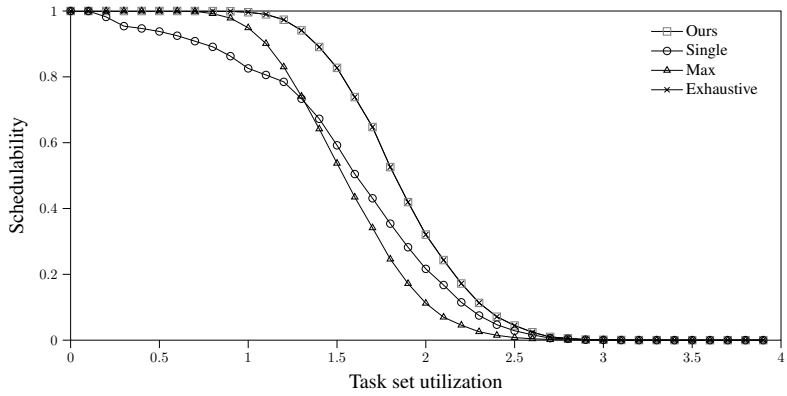
1. Create an empty task set Γ.
2. Generate a new synthetic task τ_k and add it to Γ.
3. Check Γ passes the necessary schedulability test, i.e., $\sum_{\tau_k \in \Gamma} C_k(1)/T_k < m$.
4. If true, it becomes a task set Γ. Then, with a copy of Γ, we add one more task by repeating the above two steps to make another task set with another task.
5. If not, meaning that Γ is already violating schedulability condition, we start over from the beginning with a new empty task set Γ.

With such generated 10^5 task sets and four CPU cores, i.e, $m = 4$, we com-

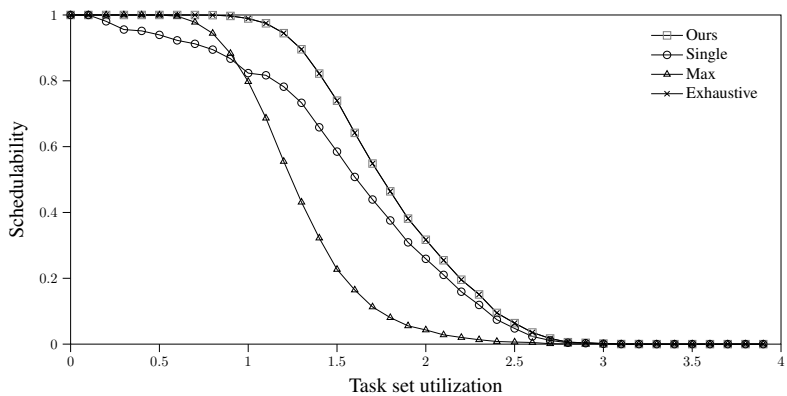
pare the schedulability resulting from the following four different parallelization approaches:

- Ours (Algorithm 2): The segments are parallelized using our proposed algorithm.
- Single: No parallelization is used. All segments execute as a single thread.
- Max: Each segment is parallelized into maximum number of threads, i.e., $O_{ks} = O^{\max}$.
- Exhaustive: For each task, try every possible combination of $O_k = \{ \{1, 1, \cdots, 1\}, \cdots, \{O^{\max}, O^{\max}, \cdots, O^{\max}\} \}$.

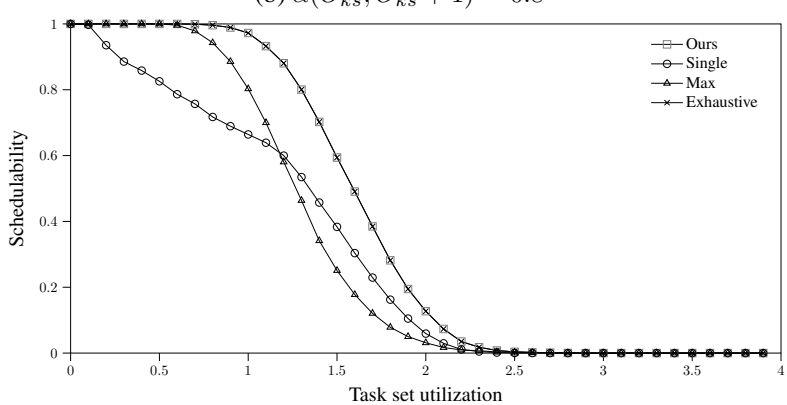
Fig. 4.11(a) compares the schedulability of the above four approaches for the whole spectrum of the task set utilization of 10^5 task sets ranging from 0 to 4 assuming the parallelization overhead $\alpha(O_{ks}, O_{ks} + 1) = 0.3$ for every $O_{ks} < O^{\max} = 4$. In the figure, the x-axis represents the task set utilization, and the y-axis represents the schedulability, i.e., the proportion of schedulable task sets out of all the task sets having the corresponding task set utilization. Comparing "Max" and "Single", the former performs better in low task set utilization while the latter performs better in high task set utilization, making a crossing point around the task set utilization of 1.3. This is because, in low utilization, the schedulability is mostly affected by deadlines, and hence reducing the thread execution time by parallelization helps. On the other hand, in high utilization, the schedulability is mostly affected by the amount of interferences among tasks, and hence keeping the number of threads small help. "Ours" performs significantly better in any task set utilization, nearly as effectively as



(a) $\alpha(O_{ks}, O_{ks}+1) = 0.3$



(b) $\alpha(O_{ks}, O_{ks}+1) = 0.8$



(c) $\alpha(O_{ks}, O_{ks}+1) = 0.3$, 20% tight deadlines

Figure 4.11: Simulation result with $m = 4$ CPU cores

the exhaustive results. This is because Algorithm 2 optimally trades off the tolerance and interference in the parallelization option selection for the schedulability of all the tasks in the given task set.

Fig. 4.11(b) shows the results when the parallelization overhead is larger, i.e., when $\alpha(O_{ks}, O_{ks}+1) = 0.8$. In this case, "Single" is not affected, but "Max" becomes worse since the benefit by parallelization is severely sacrificed due to the large parallelization overhead. "Ours" is similarly affected. Nevertheless, "Ours" still significantly outperforms all other approaches in this case of large parallelization overhead.

Fig. 4.11(c) shows the results when the deadline for each task is 20% tighter than Fig. 4.11(a). In this case, "Single" becomes significantly worse because it does not take advantage of thread execution time reduction by parallelization. On the other hand, the performance of "Max" also dropped but not that much compared with "Single" thanks to the parallelization. The performance of "Ours" just slightly drops since it can well overcome the tightened deadlines by optimal parallelization.

4.9 Experiment Results: DAG Task Model

This section shows the effectiveness of the proposed task parallelization algorithm targeting the DAG task model by simulation with synthetic tasks in Section 4.9.1. Then, Section 4.9.2 reports the actual implementation result with real autonomous driving tasks.

4.9.1 Simulation Results

A DAG task $\tau_k = (G_k, D_k, T_k)$ is randomly generated as follows: (1) number of nodes n_k randomly selected from uniform$[3, 10]$, (2) D_k, T_k randomly chosen with uniform$[2000, 3000]$ considering implicit deadline, i.e., $D_k = T_k$, (3) utilization per node ($\overline{U}_{kp} = C_{kp}/T_k$) randomly generated with normal$[0.3, 0.1]$ (4) for each node τ_{kp}, its single-thread execution time is calculated as $e^1_{kp}(1) = C_{kp}(1) = \overline{U}_{kp}T_k$, (5) then considering *parallelization overhead*, notated as α, the total execution time of consecutive parallel option is calculated as $C_{kp}(O_{kp} + 1) = (1 + \alpha)C_{kp}(O_{kp})$, (6) thread execution time table is created by dividing the total execution time into the number of threads using Unifast [15] algorithm then sorting those in the descending order, (7) each pair of τ_k's nodes is connected by odds of $Pr = 0.3$, creating set of edges E_k, and (8) conducting a breadth-first search, graph G_k is finalized with a single source and sink, adding dummy nodes with zero execution time if necessary.

Next, using this method of generating a synthetic DAG task, we create 10^6 task sets in the following manner: (1) initialize an empty task set Γ, (2) generate a DAG task using the above method then append to Γ, (3) check schedulability of Γ according to necessary schedulability condition, i.e., $\sum_{\tau_k \in \Gamma} vol(\tau_k)/T_k < m$, (4) if true, Γ becomes a valid task set. In such case, a copy of Γ is made, we add additional tasks by going back to step (2), (5) if false, we discard Γ and start over from step (1).

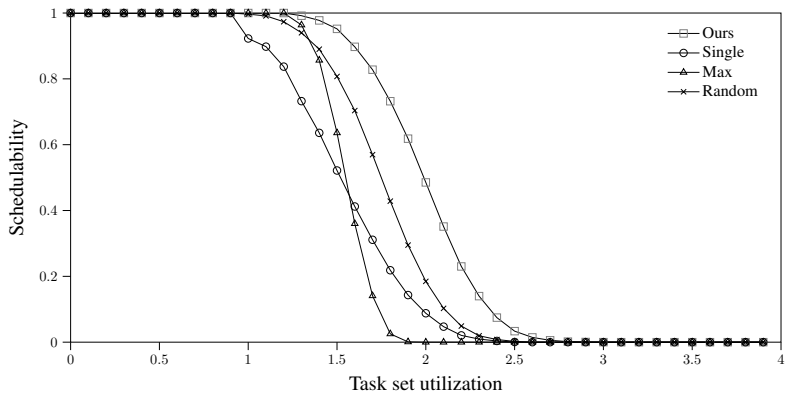
Using the above task sets, we compare the difference in schedulability of the following parallelization strategies on four virtual CPU cores ($m = 4$):

- Ours: Tasks are parallelized according to Algorithm 3.
- Single: No tasks are parallelized, i.e. single-threaded.

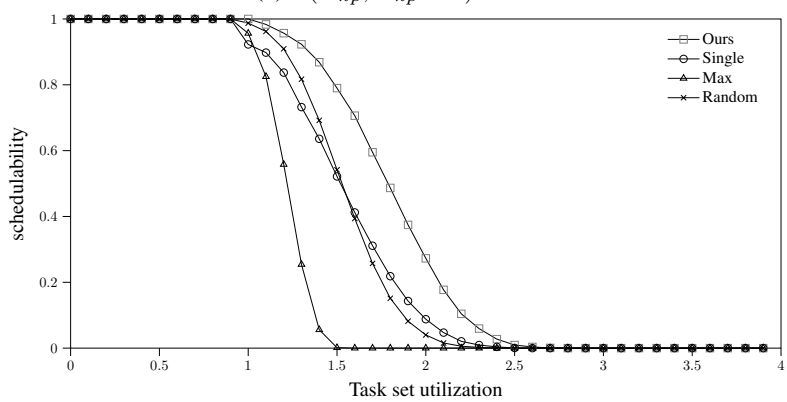
- Max: Tasks are maximally parallelized, i.e., $O^{\max} = m$.
- Random: For each nodes, parallelization option is randomly selected, i.e., $O_{kp} = \text{uniform}[1, O^{\max}]$.

Fig. 4.12(a) shows the difference in schedulability using the above parallelization strategies with comparably low parallelization overhead case, i.e., $\alpha = 30\%$. The task set utilization is represented in the x-axis, while the task set schedulability is indicated in the y-axis. Hence each point in the figure represents the ratio of the schedulable task sets to the total generated task sets that have the same utilization. Compared to any other strategies, "Ours" outdistances by a large margin, e.g., for task set utilization 2, "Ours" have schedulability of 0.49 while "Single" and "Max", and "Random" are each at 0.09, 0, and 0.19, respectively, thanks to the optimally assigned parallelization option by Algorithm 3. Next, comparing "Single" and "Max", we can see that "Max" performs better in the lower utilization, while this is flipped around utilization 1.5, hence making "Single" better in the high utilization. This is because, in the low utilization, the schedulability is affected mostly by the deadline. Therefore decreasing the execution time by parallelizing into more threads can increase the schedulability. With high utilization, however, the schedulability is mainly affected by induced interference from other tasks, thus decreasing the total number of threads can improve the schedulability. Notice both "Single" and "Max", are extreme cases and hence perform even worse than "Random".

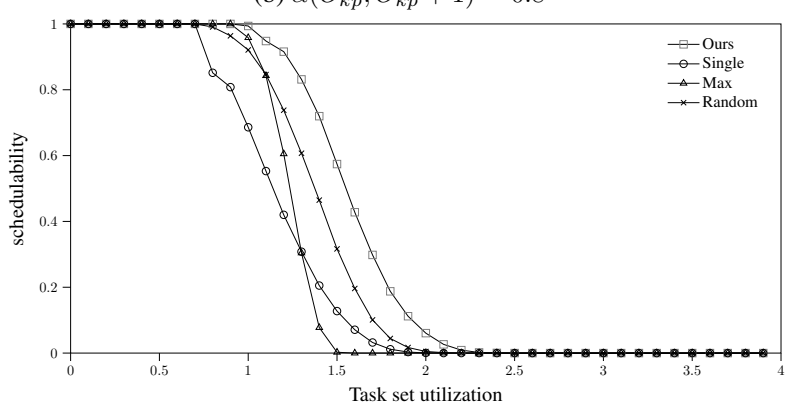
Fig. 4.12(b) shows when the parallelization overhead is increased. Because no threads are parallelized in "Single", the schedulability is not affected. "Max", however, is affected severely due to the harsh penalty of parallelization. "Ours" is also impacted yet still outperforms others by a large margin.



(a) $\alpha(O_{kp}, O_{kp}+1) = 0.3$



(b) $\alpha(O_{kp}, O_{kp}+1) = 0.8$



(c) $\alpha(O_{kp}, O_{kp}+1) = 0.3$, 20% tight deadlines

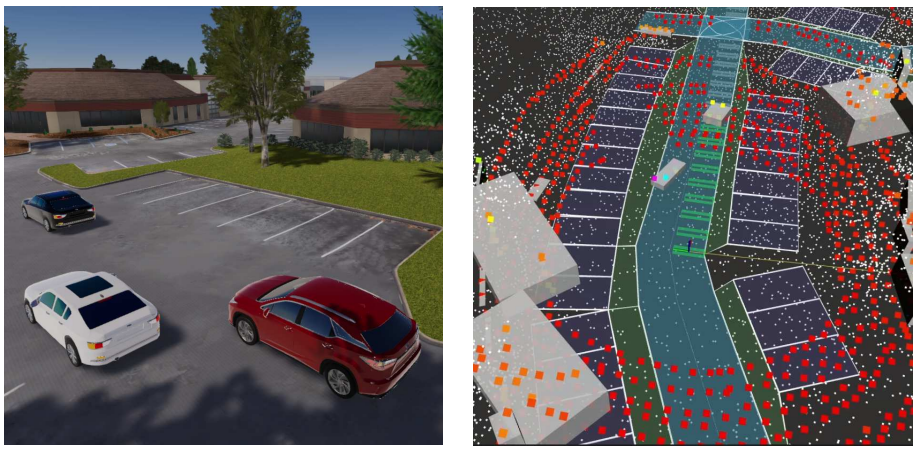
Figure 4.12: Simulation result with $m = 4$ CPU cores

Fig. 4.12(c) is when each task's deadline is tightened by 20%. Since "Single" is affected mainly by the deadline, this results in a drastic drop in performance. "Max" is also affected, but the decrease is comparably small. "Ours" can assign optimal parallelization options, hence affected minimally.

4.9.2 Implementation Results

In this section, we provide practical implementation of parallelization assignment algorithm, on top of an actual autonomous driving program AutowareAuto [26]. AutowareAuto is the leading open-source autonomous driving (AD) framework, with full-stack AD capabilities ranging from environment perception to vehicle dynamics planning. These AD functions conform to a DAG structure. Hence our proposed algorithm is applicable to assign parallelization options.

The experiment was conducted on an Intel Core i7-8700 CPU (6-cores) clocked at $3.20GHz$, with GPU disabled. The tasks are scheduled using SCHED_DEADLINE scheduler on Linux kernel 5.11.4 patched with PREEMP_RT. Our implementation using AutowareAuto is forked from the April 2021 snapshot, and involves a total of 15 nodes. Among them, 9 nodes are assigned real-time constraints: 4 `point_cloud_tf` nodes, `point_cloud_fusion` node, `euclidean_cluster` node, `ray_ground_classifier` node, and two custom implemented nodes for demonstration purposes which are `collision_avoidance` and `trajectory_planning` node. These real-time nodes are given exclusive access to $m = 4$ CPU cores, and hence other non-real-time nodes are separated to run in the remaining cores. An unique `ROS2_SingleThreadExecutor` instance is assigned to each real-time nodes to bypass ROS2 native scheduling [16]. The environmental data and vehicle kinematic information that are fed to the AD



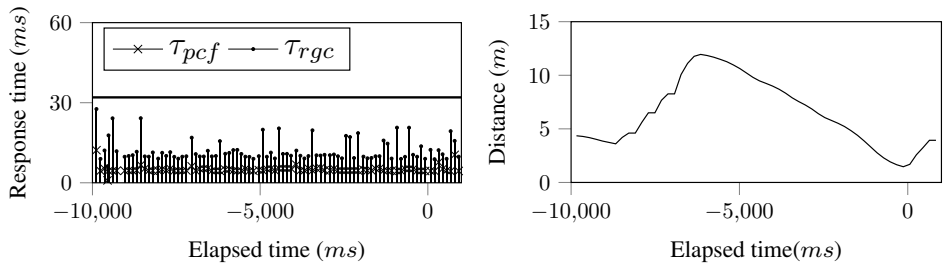
(a) SVL driving environment (b) Visualized surroundings (RViz[47])

Figure 4.13: Driving environment provided by SVL

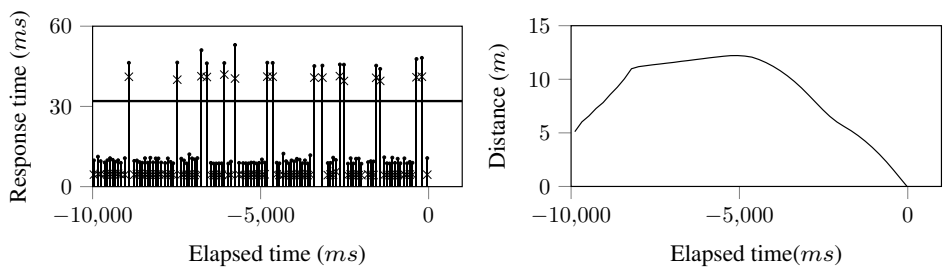
stack are supplied from a separate simulation environment running on different PC, i.e., SVL [35] simulator (Fig. 4.13). SVL is capable of real-time simulation of vehicle dynamics and environment, suitable for verifying our implementation.

Using the above setup, we designed a way-point following scenario with traffic, in which the AD vehicle must maintain a safe distance from any obstacles or other vehicles in front to avoid a collision. To achieve this, the AD stack must be alert to the dynamic change of the surroundings and be prompt in any maneuvers if necessary. For this reason, a deadline violation of any part of the AD stack can result in failure. We show that such failure can occur for the naive parallelization assignment, i.e., "Single", "Max", and "Random", while "Ours" can guarantee a safe execution, i.e., no deadline violation, thanks to the optimal parallelization.

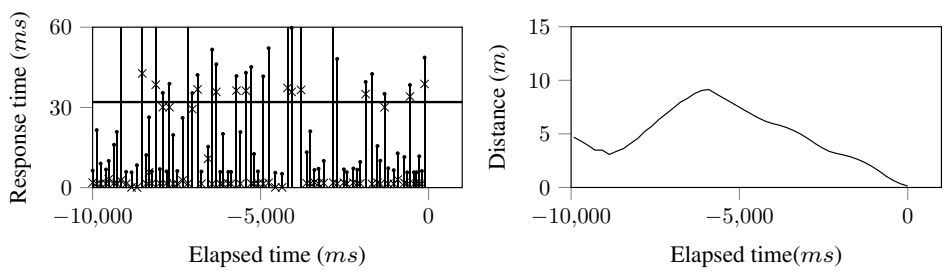
Fig. 4.14 reports the results for different strategies mentioned above. To the left, among the real-time DAG tasks, τ_{AP}, i.e. advanced perception task which consists of `point_cloud_fusion`(τ_{pcf}) $\rightarrow$ `ray_ground_classifier`(τ_{rgc}) node is selected for visu-



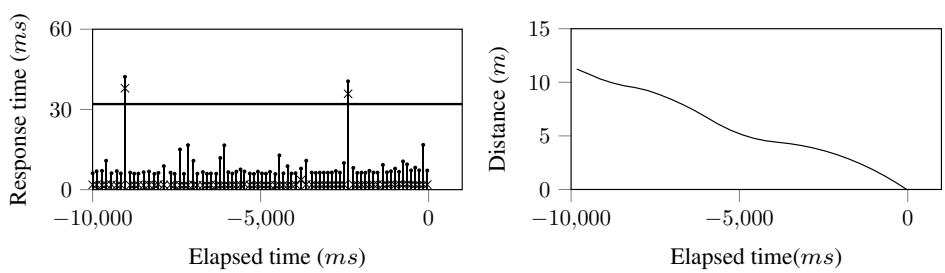
(a) Ours



(b) Single



(c) Max



(d) Random

Figure 4.14: Measured response times of τ_{AP} and distance to nearest object

alization. τ_{AP}'s execution behavior is plotted, with the x-axis representing the elapsed time and the y-axis representing the response time of the longest path, i.e., $len(\tau_{AP})$. The horizontal bar represents the deadline. For each job, we indicated the contribution of each node with different markers. To the right, the distance to the nearest object is plotted. As shown in the figure, except "Ours", other strategies experienced a collision, which is aligned to the end, i.e., $t = 0$. Time-points to the left, preceding traces are shown. As indicated in the figure, all other strategies missed their deadline, which eventually led to a collision. "Ours" executed safely, meeting all the deadlines.

Chapter 5
Conclusion

5.1 Summary

This dissertation presents an optimal parallelization of real-time tasks on multi-core processors for global EDF. For this, state-of-the-art interference-based schedulability analyses are extended, and notions of tolerance and interference parallelization freedom are developed. Next, the monotonic increasing property of both tolerance and interference according to the increase of parallelization is derived. Leveraging such properties, a parallelization option assignment algorithm with polynomial time complexity is proposed, and its optimality is formally proved.

In the following chapters, the scope of the proposed algorithm is extended to target multi-segment and extended again for DAG task models. Furthermore, the monotonic increasing property of both tolerance and interference is derived anew from respective schedulability analysis and parallelization route. As a result, node/segment-wise parallelization assignment algorithms are derived.

The effectiveness of the proposed algorithms is extensively evaluated with both simulated and implemented workload, and a significant gain in schedulability is observed. Furthermore, through actual implementation using autonomous driving tasks, i.e., AutowareAuto [26], a leading open-source autonomous driving framework, the algorithm's practicality and its applicability are demonstrated.

5.2 Future Work

With the work presented in this dissertation, parallelization freedom is brought to a wide range of task models for global EDF. Fig. 5.1 depicts the current picture of parallelization freedom, where each row represents a real-time scheduling policy and each column represents a real-time task model. In each cell, a schedulability analysis available for the respective scheduling policy and task model is specified. Then followed by an arrow, the corresponding solution for parallelization assignment is presented. For example, parallelization freedom is brought to DAG tasks for global EDF by Cho [20] by extending Chwa-DAG [22] analysis. This dissertation covers all cells of the top-most row, i.e., global EDF.

Possible extension points for parallelization freedom is also depicted in Fig. 5.1.

- (a) Extension to a tighter schedulability analysis: Although BCL is a foundational work on global EDF multi-threads schedulability analysis, it remains a sufficient analysis, and tighter analyses such as RTA [12] and RTA-LC-EDF [57] exists. The assignment algorithm of this dissertation is optimal but optimal concerning the BCL analysis. By employing such analyses, the overall gain in schedulability can be expected.

- (b) Extension to different real-time task models: DAG task model can represent a wide range of real-world workloads such as autonomous driving, i.e., AutowareAuto [26]. However, different models that do not strictly fall into the DAG task model are emerging, such as ROS2 [46] model. Therefore, a different approach must be taken, and a novel schedulability analysis is needed for parallelization freedom to be successfully incorporated into such task models.

	Multi-Thread	Multi-Segment	DAG	···
Global EDF	BCL [13] → Cho [19] (a)	Chwa-MS [23] → Cho [20]	Chwa-DAG[22] → Cho [21]	(b)
Global FP	BCL [13] → Park [44]	Chwa-MS [23] → ---	Chwa-MS [22] → ---	
Fluid	Cho [18] → Kim [33]	Cho [18] → Kim [33]	Guan[48] → ---	
⋮	(c)			

Figure 5.1: Status quo and unexplored regions of parallelization freedom

- (c) Extension to different scheduling policies: Global EDF, Global FP, and Fluid schedulers cover a significant portion of global schedulers. However, many scheduling policies, especially the partitioned approaches, are left unexplored by parallelization freedom. In addition, many of these schedulers are widely used in industries; thus, extending to different scheduling policies will improve the practicality of parallelization freedom.

References

[1] James H Anderson and John M Calandrino. Parallel Real-time Task Scheduling on Multicore Platforms. In *27th IEEE Real-Time Systems Symposium (RTSS)*, 2006.

[2] Philip Axer, Sophie Quinton, Moritz Neukirchner, Rolf Ernst, Björn Döbel, and Hermann Härtig. Response-time analysis of parallel fork-join workloads with real-time constraints. In *2013 25th Euromicro Conference on Real-Time Systems*, pages 215–224. IEEE, 2013.

[3] Benjamin Bado, Laurent George, Pierre Courbin, and Joël Goossens. A Semi-partitioned Approach for Parallel Real-time Scheduling. In *20th ACM International Conference on Real-Time and Network Systems*, 2012.

[4] Baidu. Apollo apollo 3.5 open-source autonomous driving. http:/http://apollo.auto/index.html. Accessed: 2019-05-30.

[5] Theodore P Baker. An analysis of fixed-priority schedulability on a multiprocessor. *Real-Time Systems*, 32(1-2):49–71, 2006.

[6] Theodore P Baker and Michele Cirinei. A necessary and sometimes sufficient condition for the feasibility of sets of sporadic hard-deadline tasks. In *2006 27th IEEE International Real-Time Systems Symposium (RTSS'06)*, pages 178–190. IEEE, 2006.

[7] Theodore P Baker and Michele Cirinei. Brute-force determination of multiprocessor schedulability for sets of sporadic hard-deadline tasks. In *International Conference On Principles Of DIstributed Systems*, pages 62–75. Springer, 2007.

[8] Sanjoy Baruah. Techniques for multiprocessor global schedulability analysis. In *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*, pages 119–128. IEEE, 2007.

[9] Sanjoy Baruah. Partitioned edf scheduling: a closer look. *Real-Time Systems*, 49(6):715–729, 2013.

[10] SK Baruah, NK Cohen, CG Plaxton, and DA Varvel. Proportionate Progress: a Notion of Fairness in Resource Allocation. In *25th annual ACM symposium on Theory of computing*, 1993.

[11] Marko Bertogna and Sanjoy Baruah. Tests for global edf schedulability analysis. *Journal of systems architecture*, 57(5):487–497, 2011.

[12] Marko Bertogna and Michele Cirinei. Response-time analysis for globally scheduled symmetric multiprocessor platforms. In *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*, pages 149–160. IEEE, 2007.

[13] Marko Bertogna, Michele Cirinei, and Giuseppe Lipari. Improved schedulability analysis of edf on multiprocessor platforms. In *17th Euromicro Conference on Real-Time Systems (ECRTS)*, 2005.

[14] Marko Bertogna, Michele Cirinei, and Giuseppe Lipari. Schedulability analysis of global scheduling algorithms on multiprocessor platforms. *IEEE Transactions on parallel and distributed systems*, 20(4):553–566, 2008.

[15] Enrico Bini and Giorgio C Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1):129–154, 2005.

[16] Daniel Casini, Tobias Blaß, Ingo Lütkebohle, and Björn B Brandenburg. Response-time analysis of ros 2 processing chains under reservation-based scheduling. In *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[17] Jian-Jia Chen and Samarjit Chakraborty. Partitioned packing and scheduling for sporadic real-time tasks in identical multiprocessor systems. In *24th Euromicro Conference on Real-Time Systems (ECRTS)*, 2012.

[18] Hyeonjoong Cho, Binoy Ravindran, and E Douglas Jensen. An Optimal Real-time Scheduling Algorithm for Multiprocessors. In *27th IEEE Real-Time Systems Symposium (RTSS)*, 2006.

[19] Youngeun Cho, Do Hyung Kim, Daechul Park, Seung Su Lee, and Chang-Gun Lee. Conditionally optimal task parallelization for global edf on multi-core systems. In *2019 Real-Time Systems Symposium (RTSS)*. IEEE, 2019.

[20] Youngeun Cho, Do Hyung Kim, Daechul Park, Seung Su Lee, and Chang-Gun Lee. Optimal parallelization of single/multi-segment real-time tasks for global edf. *IEEE Transactions on Computers*, 2021.

[21] Youngeun Cho, Dongmin Shin, Jaeseung Park, and Chang-Gun Lee. Conditionally optimal parallelization of real-time dag tasks for global edf. In *2021 Real-Time Systems Symposium (RTSS)*. IEEE, 2021.

[22] Hoon Sung Chwa, Jinkyu Lee, Jiyeon Lee, Kiew-My Phan, Arvind Easwaran, and Insik Shin. Global edf schedulability analysis for parallel tasks on multi-

core platforms. *IEEE Transactions on Parallel and Distributed Systems*, 28(5):1331–1345, 2016.

[23] Hoon Sung Chwa, Jinkyu Lee, KieuMy Phan, Arvind Easwaran, and Insik Shin. Global EDF Schedulability Analysis for Synchronous Parallel Tasks on Multicore Platforms. In *25th Euromicro Conference on Real-Time Systems (ECRTS)*, 2013.

[24] Bronis R de Supinski, Thomas RW Scogland, Alejandro Duran, Michael Klemm, Sergi Mateo Bellido, Stephen L Olivier, Christian Terboven, and Timothy G Mattson. The ongoing evolution of openmp. *Proceedings of the IEEE*, 106(11):2004–2019, 2018.

[25] Frédéric Fauberteau, Serge Midonnet, and Manar Qamhieh. Partitioned Scheduling of Parallel Real-time Tasks on Multiprocessor Systems. *ACM SIGBED Review*, 8(3):28–31, 2011.

[26] Autoware Foundation. Autowareauto. https://www.autoware.auto/, 2021.

[27] Linux Foundation. Linux deadline task scheduling. https://www.kernel.org/doc/Documentation/scheduler/sched-deadline.txt, 2021.

[28] Shelby Funk. LRE-TL: An Optimal Multiprocessor Algorithm for Sporadic Task Sets with Unconstrained Deadlines. *Real-Time Systems*, 46(3):332–359, 2010.

[29] Joël Goossens, Shelby Funk, and Sanjoy Baruah. Priority-driven scheduling of periodic task systems on multiprocessors. *Real-time systems*, 25(2-3):187–205, 2003.

[30] Qingqiang He, Nan Guan, Zhishan Guo, et al. Intra-task priority assignment in real-time scheduling of dag tasks on multi-cores. *IEEE Transactions on Parallel and Distributed Systems*, 30(10):2283–2295, 2019.

[31] Shinpei Kato and Yutaka Ishikawa. Gang EDF Scheduling of Parallel Task Systems. In *30th IEEE Real-Time Systems Symposium (RTSS)*, 2009.

[32] Shinpei Kato, Shota Tokunaga, Yuya Maruyama, Seiya Maeda, Manato Hirabayashi, Yuki Kitsukawa, Abraham Monrroy, Tomohito Ando, Yusuke Fujii, and Takuya Azumi. Autoware on board: Enabling autonomous vehicles with embedded systems. In *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*, pages 287–296. IEEE, 2018.

[33] Kang-Wook Kim, Youngeun Cho, Jeongyoon Eo, Chang-Gun Lee, and Junghee Han. System-wide time versus density tradeoff in real-time multicore fluid scheduling. *IEEE Transactions on Computers*, 67(7):1007–1022, 2018.

[34] Jihye Kwon, Kang-Wook Kim, Sangyoun Paik, Jihwa Lee, and Chang-Gun Lee. Multicore scheduling of parallel real-time tasks with multiple parallelization options. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2015.

[35] LG Silicon Valley Lab. Svl simulator. https://www.svlsimulator.com/, 2021.

[36] Karthik Lakshmanan, Shinpei Kato, and Ragunathan Rajkumar. Scheduling Parallel Real-time Tasks on Multi-core Processors. In *31st IEEE Real-Time Systems Symposium (RTSS)*, 2010.

[37] Greg Levin, Shelby Funk, Caitlin Sadowski, Ian Pye, and Scott Brandt. DP-FAIR: A Simple Model for Understanding Optimal Multiprocessor Scheduling. In *22nd Euromicro Conference on Real-Time Systems (ECRTS)*, 2010.

[38] Jing Li, Jian Jia Chen, Kunal Agrawal, Chenyang Lu, Chris Gill, and Abusayeed Saifullah. Analysis of federated and global scheduling for parallel real-time tasks. In *2014 26th Euromicro Conference on Real-Time Systems*, pages 85–96. IEEE, 2014.

[39] Aristidis Likas, Nikos Vlassis, and Jakob J Verbeek. The global k-means clustering algorithm. *Pattern recognition*, 36(2):451–461, 2003.

[40] Cláudio Maia, Marko Bertogna, Luís Nogueira, and Luis Miguel Pinho. Response-time analysis of synchronous parallel tasks in multiprocessor systems. In *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*, page 3. ACM, 2014.

[41] Ernesto Massa, George Lima, Paul Regnier, Greg Levin, and Scott Brandt. Quasi-partitioned scheduling: optimality and adaptation in multiprocessor real-time systems. *Real-Time Systems*, 52(5):566–597, 2016.

[42] Geoffrey Nelissen, Vandy Berten, Joël Goossens, and Dragomir Milojevic. Techniques optimizing the number of processors to schedule multi-threaded

tasks. In *2012 24th Euromicro Conference on Real-Time Systems*, pages 321–330. IEEE, 2012.

[43] Geoffrey Nelissen, Vandy Berten, Vincent Nélis, Joël Goossens, and Dragomir Milojevic. U-EDF: An Unfair but Optimal Multiprocessor Scheduling Algorithm for Sporadic Tasks. In *24th Euromicro Conference on Real-Time Systems (ECRTS)*, 2012.

[44] Daechul Park, Youngeun Cho, and Chang-Gun Lee. Conditionally optimal parallelization for global fp on multi-core systems. In *2020 3rd International Conference on Information and Computer Technologies (ICICT)*, pages 403–412. IEEE, 2020.

[45] Joseph Redmon. Darknet: Open source neural networks in c. http://pjreddie.com/darknet/, 2013–2016.

[46] Open Robotics. Ros2. https://docs.ros.org/en/galactic/index.html, 2021.

[47] Open Robotics. Rviz. https://github.com/ros-visualization/rviz, 2021.

[48] Abusayeed Saifullah, David Ferry, Jing Li, Kunal Agrawal, Chenyang Lu, and Christopher D Gill. Parallel real-time scheduling of dags. *IEEE Transactions on Parallel and Distributed Systems*, 25(12):3242–3252, 2014.

[49] Abusayeed Saifullah, Jing Li, Kunal Agrawal, Chenyang Lu, and Christopher Gill. Multi-core Real-time Scheduling for Generalized Parallel Task Models. *Real-Time Systems*, 49(4):404–435, 2013.

[50] Scheduler-Tools. Rtapp benchmark. https://github.com/scheduler-tools/rt-app, 2021.

[51] Saeed Senobary and Mahmoud Naghibzadeh. Ss-drm: Semi-partitioned scheduling based on delayed rate monotonic on multiprocessor platforms. *Journal of Computing Science and Engineering*, 8(1):43–56, 2014.

[52] John F Shortle, James M Thompson, Donald Gross, and Carl M Harris. *Fundamentals of queueing theory*, volume 399. John Wiley & Sons, 2018.

[53] Anand Srinivasan and James H Anderson. Fair Scheduling of Dynamic Task Systems on Multiprocessors. *Journal of Systems and Software*, 77(1):67–80, 2005.

[54] Anand Srinivasan and Sanjoy Baruah. Deadline-based scheduling of periodic task systems on multiprocessors. *Information Processing Letters*, 84(2):93–98, 2002.

[55] Anand Srinivasan, Philip Holman, James H Anderson, and Sanjoy Baruah. The case for fair multiprocessor scheduling. In *International Symposium on Parallel and Distributed Processing Symposium (IPDPS)*, 2003.

[56] John E Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66, 2010.

[57] Youcheng Sun and Giuseppe Lipari. Response time analysis with limited carry-in for global earliest deadline first scheduling. In *2015 IEEE Real-Time Systems Symposium*, pages 130–140. IEEE, 2015.

[58] H. Yun, R. Mancuso, Z. P. Wu, and R. Pellizzoni. Palloc: Dram bank-aware memory allocator for performance isolation on multicore platforms. In *2014*

IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS), pages 155–166, April 2014.

[59] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 55–64, April 2013.

초 록

이 논문은 Global EDF 스케줄러에서의 실시간 태스크의 최적 병렬화 기법에 대해 기술한다. 이를 위해 병렬화 옵션 증가에 대해 tolerance와 interference가 단조 증가함을 수학적으로 밝힌다. 이러한 특성을 이용하여, 다항식 시간안에 수행될 수 있는 실시간 태스크의 병렬화 기법을 제안하고, 수학적으로 최적의 방법임을 증명한다. 시뮬레이션 실험을 통해, 실시간 태스크의 스케줄링 성능의 비약적인 상승을 관찰하고, 이어지는 실제 세계 워크로드를 이용한 구현 실험에서 실제 세계 적용 가능성을 점검한다. 이 논문은 먼저 전통적인 멀티쓰레드 태스크 모델을 대상으로 최적 병렬화 방법을 논하며, 이후 멀티 세그먼트, DAG 태스크 모델까지 확장하는 방법을 기술한다.

주요어 : 실시간 시스템, 멀티코어 스케줄링, 자유 병렬화, 최적 병렬화
학 번 : 2015-21265

감사의 글

설레는 마음으로 박사과정을 시작한 게 엊그제 같은데, 벌써 박사 논문의 마지막 장을 쓰고 있습니다. 그동안 정말 많은 분께 과분한 도움과 사랑을 받았습니다. 감사의 말을 드리고 싶은 분들이 정말 많습니다. 이 글을 빌어 감사 인사를 전하고자 합니다.

가장 먼저 오랜 시간 지도해주시고 격려해주신 이창건 교수님께 깊은 감사를 드립니다. 바쁘신 가운데에서도 직접 연구를 챙겨주시고, 언제나 진심 어린 조언을 아끼지 않으시는 모습이 제게는 정말 큰 감동이었습니다. 교수님 연구실에서 박사과정을 보낼 수 있었던 것이 제게는 정말로 큰 행운이었습니다. 교수님께 받은 은혜와 가르침을 따라 살아가도록 노력하겠습니다.

다음으로 흔쾌히 제 박사 학위 심사위원을 수락하여 주시고, 심사기간동안 귀중한 시간 내주어 지도해주신 하순회 교수님, 오성회 교수님, 이영기 교수님, 김종찬 교수님께 감사의 말씀을 전합니다. 교수님들께서 지도해주시고 심사해주신 보람이 있도록 더 열심히 노력하고 연구하겠습니다.

연구실에서 많은 분들게 도움을 받았습니다. 가장 오랜 기간 연구실에서 함께 했고, 제게는 늘 선망의 대상이었던 권오철 박사님, 김강욱 박사님, 위경수 박사님께 먼저 감사의 말씀을 전하고 싶습니다. 존경합니다. 그리고 제게는 너무나 소중한 선배님들로, 연구실 초반 많이 보살펴 주시고, 아낌없이 조언과 도움 주셨던 지화형, 상윤형, 승곤형, 환석형, 항영형께 감사드립니다. 가장 바쁘고 힘들었던 시기 함께한 원재, 원석, Alena, 그리고 승수. 정말 기억에 많이 남고 항상 고맙고 미안한 마음뿐입니다. 같은 학번 동기인 대철, 혜진누나, 정윤누나, 찬희형, 그리고 Nushaba가 있어서 정말 든든했습니다. 고맙습니다. 또 나이가 같아 의지가 많이 된 재환, Taylor에게도 특별히 고마운 마음을 전하고 싶습니다. 지금은 연구실에

없는 이재우 박사님, Artem 박사님, 동완형, Alex, Diane, Victoria, 선준, 그리고 진에게도 깊은 감사를 드립니다. 현재 연구실에서 석사 과정을 밟고 있는 병규, 종우, 서환형, 그리고 해주, 정말 고생이 많아요. 고맙습니다. 큰 응원 보냅니다. 마지막으로 연구실을 앞으로 이끌 박사과정, 도형형, 하연, 성현, 그리고 동민. 진심으로 응원합니다. 힘든 길, 잘 헤쳐 나갈 수 있었으면 좋겠습니다. 그동안 정말 감사했습니다. 앞으로도 늘 소식 전하며 인연 닿기를 바랍니다.

마지막으로 아버지, 어머니, 그리고 가은. 글로 다 표현할 수 없을 만큼 고맙고 사랑합니다. 끝없는 사랑과 지지 덕분에 여기까지 올 수 있었습니다. 언제나 화수분 같은 사랑과 응원을 보내주신 외할아버지, 외할머니, 그리고 제 졸업소식을 누구보다 기뻐하셨을 하늘에 계신 할아버지, 할머니께 이 논문을 드립니다.