공학석사 학위논문

Conditionally Optimal Parallelization for Global FP on Multi-core Systems

멀티코어 시스템 상의 Global FP를 위한 조건적 최적 병렬화 기법

2020년 8월

서울대학교 대학원
컴퓨터공학부
박대철

Conditionally Optimal Parallelization for Global FP on Multi-core Systems

지도교수 이창건

이 논문을 공학석사 학위논문으로 제출함

2020년 6월

서울대학교 대학원
컴퓨터공학부
박대철

박대철의 석사 학위논문을 인준함
2020년 7월

위 원 장 하순회 (인)

부위원장 이창건 (인)

위 원 김지홍 (인)

Abstract

Conditionally Optimal Parallelization for Global FP on Multi-core Systems

Park Daechul
Department of Computer Science and Engineering
The Graduate School
Seoul National University

Throughout the last decade, the importance of parallel computing has risen greatly to match the ever-increasing computational demand. Frameworks such as OpenMP and OpenCL allow easy parallelization of computing tasks into desirable number of threads, opening up a chance to greatly utilize the parallel computing resources. We call this "parallelization freedom". However, this does not come for free, as parallelization overhead increase with parallelization option (i.e. the number of thread each task is parallelized). Thus parallelization option must be carefully decided to better utilize a given computing resource. This paper addresses the problem of assigning parallelization option to each task for global FP scheduler. For this, we extend the approaches made by Cho, which is limited to the global EDF scheduler case. We prove that a conditionally optimal parallelization assignment of parallelization option also exists for the global FP case. Through extensive simulations and autonomous driving module task sets, we show a significant improvement of schedulability.

keywords : scheduling; parallelization freedom; global FP

Student Number : 2016-21201

Contents

List of Figures

1 Introduction

Today, as computational demand continuously grows, the importance of parallel computing has risen greatly. This trend is also relevant in the real-time community, as real-time safe-critical tasks are also growing in its complexity and size. Using a parallelization framework such as OpenMP [3] and OpenCL [20], we can parallelize such tasks, each into multiple threads. This is called "parallelization freedom." Such frameworks allow easy parallelization of computing tasks. However, there is no such thing as a free lunch. The parallelization overhead increases with the parallelization option (i.e., the number of threads each task is parallelized) [1]. Therefore, the parallelization option must be decided carefully to utilize a given computing resource better.

This problem has recently drawn attention in the real-time community [13], and several methods have been proposed for Fluid scheduling [12] [8] and for global EDF [4]. However, both scheduling method is not practical, as both scheduling methods accompanies a large context switch and migration overhead. Global FP [2], on the other hand, does not suffer from such drawbacks and thus is widely accepted and used throughout the industry. However, the problem of assigning the parallelization option has not yet been discussed for the global FP case. For this, this paper proposes a method to assign an optimal parallelization option for the global FP scheduler.

To validate the assigned parallelization option for each task, we need a schedulability analysis. For this, we use BCL [6], a sufficient, polynomial time schedulability analysis for global EDF and global FP [5], and one of the foundational work on interference-based schedulability analysis domain. BCL checks whether each task

could complete its execution before the deadline, even maximally interfered by other tasks [19].

In [9], Cho et al. observed that when an assignment of parallelization option is analyzed through BCL, the following two properties emerge: 1) Increase of parallelization option of a task acquire greater "tolerance" (i.e. greater room for tolerating interference). 2) However, at the same time, the task may inflict greater "interference" on other tasks. Cho et al. proves both properties monotonically increase with parallelization option, and based on that proposes an optimal parallelization option assignment algorithm: 1) All tasks start at no-parallelization, 2) Parallelization option of each task is increased when its "tolerance" is smaller than its experienced "interference," 3) iterate overall tasks, until all tasks can tolerate the received interference (schedulable), or cannot while reaching maximum possible option (unschedulable).

In this paper we prove that the same monotonic increasing property of both "interference" and "tolerance" regarding to the parallelization option hold for the global FP case [16]. We trade-off the two properties to derive an optimal parallelization option assignment for global FP. The effectiveness of the proposed algorithm is validated through extensive simulation, and we observed a significant improvement of schedulability.

The rest of this paper is organized as follows. Section II formally defines our problem of parallelization option assignment for global FP scheduling. In Section III we extend the BCL schedulability analysis for global FP scheduler, and for tasks with parallelization. In Section IV, we introduce the two properties of parallelization, i.e. tolerance and interference, and identify a trade-off relation between the two. In section V, the properties of parallelization is used to derive an optimal assignment

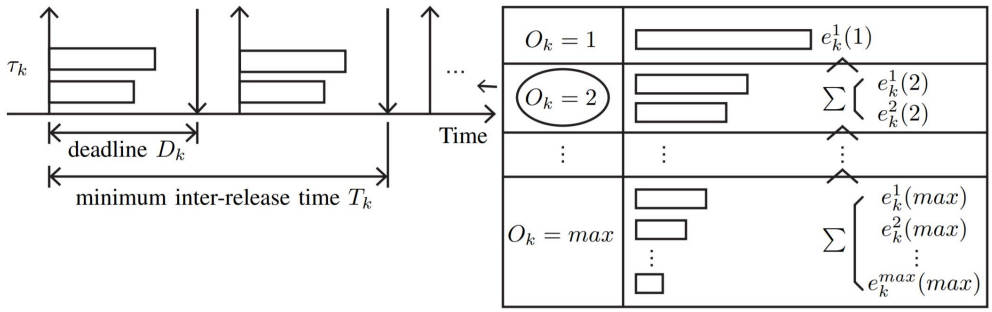


Figure 1: Task with parallelization freedom and it's execution table e_k.

algorithm. Section VI reports our experimental results. Finally, Section VII states our conclusions.

2 Problem Description

A system with m homogeneous CPU cores and n sporadic tasks is considered. Those tasks are scheduled by global FP, and each tasks, where the k-th task denoted as τ_k, is assigned a priority π_k for scheduling. Without loss of generality, we assume tasks are sorted by their priority in the descending order, i.e. $i \leq j \Leftrightarrow \pi_i \geq \pi_j$.[14] The tasks are contained in a task set Γ which is represented as follows:

$$\Gamma = \{\tau_k \mid 1 \leq k \leq n\}.$$

Each tasks are expressed by the following four parameters:

$$\tau_k = (\pi_k, T_k, D_k, e_k),$$

where π_k is the previously mentioned assigned priority, 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 Figure 1. O_k can have a value ranging from 1 (no-parallelization) to $O^{max} = m$ (number of CPU) threads. When τ_k is parallelized into O_k number of threads, they share the same priority [11], release times and deadlines. Thus we call those O_k threads as sibling threads, and denote such a parallelized task by $\tau_k(O_k)$:

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

where $\tau_k^l(O_k)(1 \leq l \leq O_k)$ is the l-th sibling thread. The execution time of the l-th sibling thread $e_k^l(O_k)$ can be obtained from thread execution table (Figure 1). Without loss of generality, we assume that the execution time of sibling threads are sorted in the descending order, i.e. $i \leq j \Leftrightarrow e_k^i(O_k) \geq e_k^j(O_k)$. Therefore for any given sibling threads, the first thread among them $e_k^1(O_k)$ has the largest execution time, i.e. $\max_{\tau_k^l(O_k) \in \tau_k(O_k)} e_k^l(O_k) = e_k^1(O_k) := e_k^{max}(O_k)$. The sum of all thread execution time of sibling threads of parallelization option O_k, i.e., $\sum_{l=1}^{O_k} e_k^l(O_k)$, is denoted by $C_k(O_k)$, and will be referred to as the total computation time of $\tau_k(O_k)$.

As the parallelization option increases $O_k \rightarrow O_k'$ individual thread execution time decreases, i.e. $e_k^l(O_k) \geq e_k^l(O_k')$, but the total computation time increases $C_k(O_k) \geq C_k(O_k')$. Which is due to the parallelization overhead. We define the parallelization overhead $\alpha(O_k \rightarrow O_k + 1)$ 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,

$$\alpha(O_k \rightarrow O_k + 1) = \frac{C_k(O_k + 1) - C_k(O_k)}{e_k^1(O_k) - e_k^1(O_k + 1)}. \quad (1)$$

The tasks are scheduled on m homogeneous CPU cores using global FP scheduler, with the priority that is statically assigned to each task and unchanged throughout execution. Similar to other researches on global FP scheduler[6], we assume that all task can be preempted and migrated at any time with negligible scheduling cost.

Problem Definition: For a given task set Γ, our problem is to find a parallelization option O_k for each task $\tau_k \in \Gamma$, that makes all the sibling threads of all the tasks in Γ can be scheduled meeting their deadlines using global FP on m homogeneous CPU cores.

3 Global FP Extension of BCL Schedulability Analysis for Tasks with Parallelization Freedom

A schedulability analysis is needed to validate the assigned parallelization option [10] [18]. Many works on schedulability analysis for global FP exists[2], targeted for sporadic tasks scheduled on multi-core systems. Some of them are exact, meaning a true determination of schedulability can be done in sacrifice of the computational time. They require exponential time and thus are ruled out of consideration for this paper. Thus we will have to use the sufficient variants, that operate in polynomial time.

However, to the best of our knowledge, currently there are no sufficient schedulability analysis that targets global FP and tasks with parallelization freedom. For this, we extended a sufficient schedulability analysis, i.e. BCL [6], which is one of the foundation work on the sufficient schedulability domain.

In this section we first overview the BCL schedulability analysis targeted for

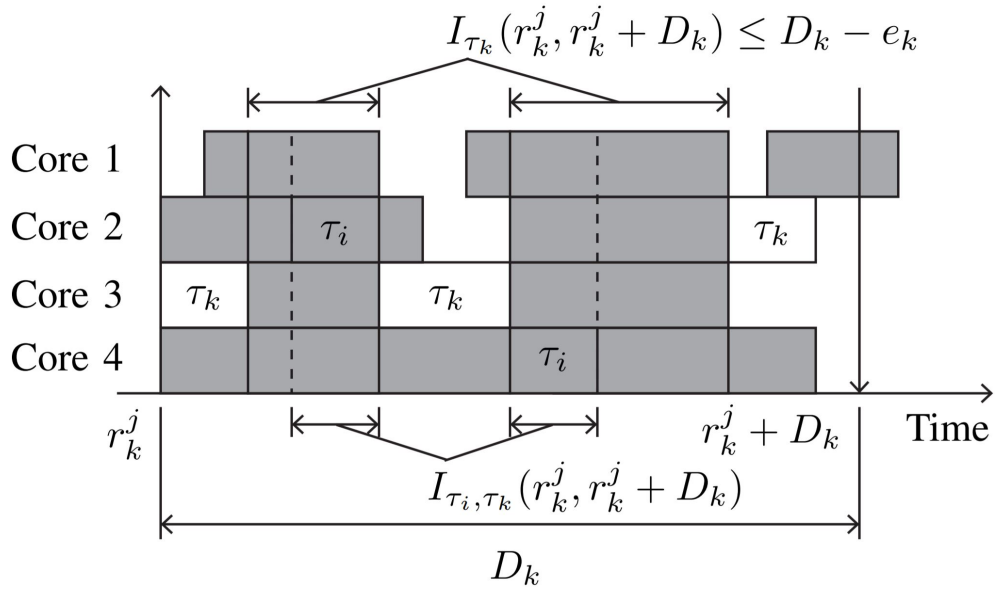


Figure 2: Interference of τ_k within the interval $[r_{kj}, r_{kj} + D_k]$.

global FP on multi-core systems. Then we extend the analysis to accommodate tasks with parallelization freedom.

3.1 Overview of BCL Schedulability Analysis for Global FP

Developed by Bertogna [6] et al., BCL schedulability analysis is a sufficient analysis targeted for sporadic tasks [5] on multicore systems, scheduled with global FP. The analysis uses the following definitions: (See Figure 2)

1) $I_{\tau_k}(a, b)$: Accumulated length of all intervals in which τ_k is ready to execute (released), but cannot execute due to higher priority tasks occupying all the CPU cores.

2) $I_{\tau_i,\tau_k}(a, b)$: Accumulated length of all intervals in which τ_k is ready to execute (released), but cannot execute due to τ_i's jobs occupying the CPU cores. Note

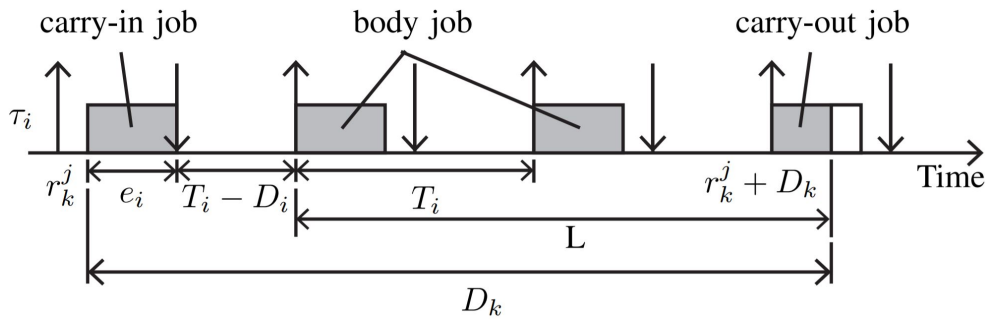


Figure 3: Worst-case workload of τ_i within the interval $[r_k^j, r_k^j + D_k]$.

that the interval $I_{\tau_i,\tau_k}(a,b)$ is always a subset of the interval $I_{\tau_k}(a,b)$.

Summing up $I_{\tau_i,\tau_k}(a,b)$ for all the tasks that has higher priority than τ_k, $(\tau_i \in \Gamma, \pi_i \geq \pi_k)$,and dividing with the number of CPU cores m yields $I_{\tau_k}(a,b)$:

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

For any job of τ_k^j is to be executed, it must complete e_k amount of execution within D_k time unit after its release(r_k^j). Therefore for the job to successfully complete its execution, the maximal amount of interference it can suffer is $D_k - e_k$. A schedulability condition can be derived by combining this observation with the above equation (2):

$$I_{\tau_k}(r_k^j, r_k^j + D_k) = \frac{1}{m} \sum_{i \leq k} I_{\tau_i,\tau_k}(r_k^j, r_k^j + D_k) \leq D_k - e_k. \tag{3}$$

Although the above schedulability condition is exact, it is not practically useful, because for a sporadic task system, all tasks may release at arbitrary time points. This requires checking exponentially many possible values of $I_{\tau_i,\tau_k}(r_k^j, r_k^j + D_k)$. To

overcome this, BCL upper-bounds the interference $I_{\tau_i,\tau_k}(r_k^j, r_k^j + D_k)$ to the total worst-case workload of the task within the same interval. In other words, the interference of τ_i to τ_k cannot exceed τ_i's total execution. Figure 3 shows the maximal workload of τ_i that can squeeze inside an interval of length D_k. It is when the jobs of τ_i is aligned and executed with the following conditions

1) τ_i's first job's execution begins exactly at the start of the interval, and finishes executing at its absolute deadline. This job is called a carry-in job.

2) Subsequent jobs of τ_i is executed most frequently with its minimum inter-release time T_k. These jobs are called body jobs.

3) Last job of τ_i is also released most frequently, and executes immediately. This job is called a carry-out job.

Summing up the above three cases, we can calculate the worst-case workload of τ_i within τ_k, i.e., W_{τ_i,τ_k} as follows:

$$W_{\tau_i,\tau_k} = \min(e_i, D_k) + \lfloor \frac{L}{T_i} \rfloor e_i + \min(e_i, L \bmod T_i), \tag{4}$$

where $L = D_k - e_i - (T_i - D_i)$ is the remaining interval of D_k when the carry-in job is removed (See Figure 3). In the above equation, the first term $\min a(e_i, D_k)$ is the workload from the carry-in job, the second term $\lfloor \frac{L}{T_i} \rfloor e_i$ is from the body jobs, and the last term $\min(e_i, L \bmod T_i)$ is from the carry-out job.

Bertogna [5], et al. observes that when τ_k is schedulable, the maximum interference that each interfering task τ_i gives to τ_k cannot exceed $D_k - e_k$. Therefore when $W_{\tau_i,\tau_k} \geq D_k - e_k$, the excess workload can be thought to have ran in parallel with τ_k,

thus we should not consider the excess as interference. In other words, we can limit the contribution of each W_{τ_i,τ_k} to $D_k - e_k$. We can call this "bounded workload" from τ_i to τ_k, i.e., $\overline{W}_{\tau_i,\tau_k}$:

$$\overline{W}_{\tau_i,\tau_k} = \min(W_{\tau_i,\tau_k}, D_k - e_k) \tag{5}$$

Worst-case workload of τ_i within the interval $[r_k^j, r_k^j + D_k)$. Now, using the concept of "bounded workload", we can extend the previous exact schedulability condition, i.e. Eq. (3), to derive a sufficient schedulability condition for τ_k:

$$I_{\tau_k}(r_k^j, r_k^j + D_k) = \frac{1}{m}\sum_{i \le k} I_{\tau_i,\tau_k}(r_k^j, r_k^j + D_k) \le \frac{1}{m}\sum_{i \le k} \overline{W}_{\tau_i,\tau_k} \le D_k - e_k. \tag{6}$$

Finally, the schedulability condition Eq. (6) can be extended to the entire task set Γ, by checking whether Eq. (6) is satisfied for $\forall \tau_k \in \Gamma$. From now on, we will use the term "schedulable" meaning that the task or the task set is "BCL schedulable" according to Eq. (6).

3.2 Extension of BCL Schedulability Analysis for Tasks with Parallelization Freedom

The schedulability analysis derived in the previous section Eq. (6) is targeted for single threaded tasks, therefore it cannot be applied directly to our model which involves parallelized tasks. Therefore in this section we present an extension of the analysis for tasks with parallelization freedom.

Recall from Chapter 2, a task with parallelization option O_k, i.e. $\tau_k(O_k)$ creates a set of O_k threads, i.e. $\tau_k(O_k) = \{\tau_k^1(O_k), \tau_k^2(O_k), \cdots, \tau_k^{O_k}(O_k)\}$, which shares the

same priority π_k, deadline D_k, minimum inter-release time T_k, but different execution time, i.e. $\{e_k^1(O_k), e_k^2(O_k), \cdots, e_k^{O_k}(O_k)\}$. Now consider the set of parallelization option $\mathbb{O} = \{O_1, O_2, \cdots, O_n\}$, where O_k is the parallelization option of τ_k. We can extend the original task set $\mathbf{\Gamma}$, by combining the set of threads created with parallelization option $\mathbb{O}$:

$$\Gamma(\mathbb{O}) = \tau_1(O_1) \cup \tau_2(O_2) \cup \cdots \cup \tau_n(O_n).$$

We will call this an extended task set $\Gamma(\mathbb{O})$, or simply a task set whenever there is no ambiguity. This extended task set has $O_1 + O_2 + \cdots + O_n$ number of individual threads. Now Eq. (6) can be applied to the extended task set $\Gamma(\mathbb{O})$ to determine the schedulability of the original task set $\mathbf{\Gamma}$.

Moreover, additional improvement of the schedulability analysis can be made by leveraging the property of the sibling threads; that they share the same release time and deadline. This is unlike the case of non-sibling threads that belong to different tasks, which has no correlation with each other at all. Since the interval between the release and the deadline of sibling threads overlap exactly with each other, we can be sure that all the sibling threads receive exactly the same workload from other non-sibling tasks. In such condition, it can be proved that the largest thread among the siblings, i.e. $e_k^1(O_k)$, is the hardest to schedule[9]. Using this property, now we can determine the schedulability of all the siblings only by checking the schedulability of its largest thread. Therefore for $\tau_k(O_k)$ to be schedulable, we need to check for only $\tau_k^1(O_k)$:

$$\sum_{\tau_i \in \Gamma(\mathbb{O}), \tau_i \neq \tau_k^1(O_k), \pi_i \geq \pi_k} \overline{W}_{\tau_i, \tau_k^1(O_k)} \leq m(D_k - e_k^1(O_k)) \tag{7}$$

4 A Trade-Off of Parallelization: Tolerance VS. Interference

When applying the extended BCL schedulability analysis on tasks with parallelization freedom, we can observe a following trade-off relation: Upon increase of a task's parallelization option:

- The task's execution time is shortened. Since its deadline stays the same, its "tolerance" to other tasks' interference increases. In other words, the task may be schedule even receiving more interference from other tasks.
- However, as a side-effect, the additional threads created by the parallelization may induce greater "interference" to other tasks.

In this section we formally discuss such effects of parallelization of a task to its "tolerance" and "interference" to others. We prove that both effects in fact, comprises a monotonic relation regarding to the parallelization option.

4.1 Monotonic Increasing Property of Tolerance

Tolerance is defined for each task of selected option $\tau_k(O_k)$, as a threshold of interference a task can experience for it to be schedulable. This value can be obtained from the extended BCL schedulability condition for $\tau_k(O_k)$, i.e. Eq. (7), by separating the bounded workload it receives from sibling and non-sibling threads. This is

represented as:

$$\sum_{\tau_k^l(O_k)\in\tau_k(O_k),l\neq 1} \overline{W}_{\tau_k^l(O_k),\tau_k^1(O_k)} + \sum_{\tau_i\notin\tau_k(O_k),\pi_i\geq\pi_k} \overline{W}_{\tau_i,\tau_k^1(O_k)} \leq m(D_k - e_k^1(O_k) \tag{8}$$

where the first term, i.e. $\sum_{\tau_k^1(O_k)\in\tau_k(O_k),l\neq 1} \overline{W}_{\tau_k^l(O_k),\tau_k^1(O_k)}$, is $\tau_k^1(O_k)$'s received bounded workload from its siblings and the second term, i.e. $\sum_{\tau_i\notin\tau_k(O_k),\pi_i\geq\pi_k} \overline{W}_{\tau_i,\tau_k^1(O_k)}$, is from its

$$\sum_{\tau_i\notin\tau_k(O_k),\pi_i\geq\pi_k} \overline{W}_{\tau_i,\tau_k^1(O_k)} \leq m(D_k - e_k^1(O_k)) - \sum_{\tau_k^l(O_k)\in\tau_k(O_k),l\neq 1} \overline{W}_{\tau_k^l(O_k),\tau_k^1(O_k)} \tag{9}$$

The left-hand side is the amount of the bounded workload or interference from other tasks' threads that $\tau_k(O_k)$ receives. This value needs to be smaller than the right-hand side for $\tau_k(O_k)$ to be schedulable. Therefore the right-hand side can be interpreted as tolerance of $\tau_k(O_k)$, and denoted as:

$$\overline{W}_{\tau_k}^{tolerance}(O_k)6 = m(D_k - e_k^1(O_k)) - \sum_{\tau_k^l(O_k)\in\tau(O_k),l\neq 1} \overline{W}_{\tau_k^l(O_k),\tau_k^1(O_k)} \tag{10}$$

This definition of tolerance has a monotonic increasing property in regards of the parallelization option O_k of a task, which is proven for the global EDF case in [9].

Since the definition of tolerance is the same for the global FP case, the monotonic increasing property of tolerance still holds for the global FP case, which can be formally written:

Property 1. $\overline{W}_{\tau_k}^{tolerance}(O_k)$ is a monotonic increasing function of $O_k(< O^{max} =$

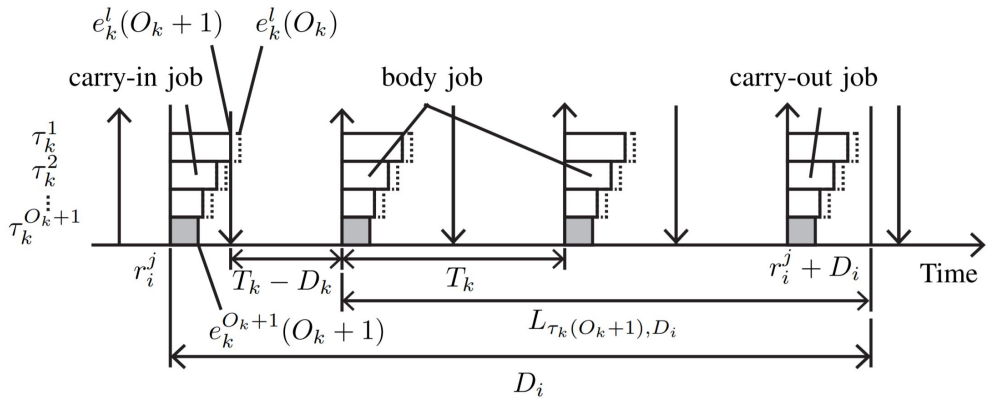


Figure 4: Worst-case workload of sibling threads of τ_k within the interval $[r_i^j, r_i^j + D_i)$.

$m)$.

4.2 Monotonic Increasing Property of Interference

Now with Property 1, we are able to improve the schedulability of a given task by increasing its parallelization option. However, this does not come for free. For the additional workload threads created by τ_k's increased parallelization may inflict greater interference to other tasks in the system. We explain the intuition behind how this happens in Figure 4, where τ_k's parallelization option increases from O_k=1 to O_k=4 within $\tau_i \neq \tau_k$'s execution interval D_i. We can see that in the higher parallelization case of τ_k, i.e. $\tau_k(4)$, more carry-in and carry-out jobs (shaded area) gets included inside τ_i's interval.

Being precise, $\tau_k(O_k)$'s threads' individual contribution of interference toward τ_i does not increase. However it is the summation of interference from all the sibling threads of $\tau_k(O_k)$ that increases as O_k increases. Thus we define the interference function $\tau_k(O_k)$ given to another task τ_i as a sum of all the interference from sibling

threads of $\tau_k(O_k)$:

$$\overline{W}^{interference}_{\tau_k,\tau_i}(O_k) = \sum_{l=1}^{O_k} \overline{W}_{\tau_k^l(O_k),\tau_i^1(O_i)} = \sum_{l=1}^{O_k} \min(W_{\tau_k^l(O_k),\tau_i^1(O_i)}, D_k - e_i^1(O_i)) \tag{11}$$

where $W_{\tau_k^l(O_k),\tau_i^1(O_i)}$, is calculated using Eq. 4

Recall that all sibling threads share the same release time. This suggests, as shown in Figure 4, the worst case execution pattern should be different for the sibling threads. To represent this situation for the sibling thread, we define $L_{\tau_k(O_k)}(D_i)$, as the remaining interval of D_i, i.e. $D_i - e_k^1(O_k) - (T_k - D_k)$.

The sum of the execution time of all sibling threads increases for higher option due to the parallelization overhead[12], i.e., $C_k(O_k) \leq C_k(O_k+1), (\forall O_k, 1 \leq O_k \leq m)$. Also as shown in Figure 3, the execution time of the sibling thread, the length of the horizontal bar, decreases when we choose higher parallelization option, i.e., $e_k^l(O_k) \leq e_k^l(O_k+1), (\forall l \leq O_k)$. Therefore, the execution time of the newly created $(O_k + 1)$-th sibling thread, depicted as a dark horizontal bar in Figure 4, is greater if not equal than the sum of the decreased execution time of other sibling threads, likewise Eq. 12.

$$e_k^{O_k+1}(O_k + 1) \geq \sum_{l=1}^{O_k} (e_k^l(O_k) - e_k^l(O_k + 1)) \tag{12}$$

We can use the definition of interference function $\overline{W}^{interference}_{\tau_k,\tau_i}(O_k)$ to formally write the second property of parallelization, and prove it in the following lemma.

Property 2. $\overline{W}^{interference}_{\tau_k,\tau_i}(O_k)$ is a monotonic increasing function of $O_k (< O^{max} = m)$.

Lemma 1. $\overline{W}^{interference}_{\tau_k,\tau_i}(O_k) < \overline{W}^{interference}_{\tau_k,\tau_i}(O_k+1)$, when $O_k (< O^{max} = m)$.

Proof. To prove this lemma, we will first show that the total workload of τ_k within τ_i's interval increases when the parallelization increases, i.e. $W_{\tau_k,\tau_i}(O_k) < W_{\tau_k,\tau_i}(O_k+1)$. Then we will show that the interference function increases as well, i.e. $\overline{W}^{interference}_{\tau_k,\tau_i}(O_k) < \overline{W}^{interference}_{\tau_k,\tau_i}(O_k+1)$.

The total workload of τ_k within τ_i's interval is expressed using Eq. 4. It is a sum of the following three terms:

1) Carry-in job: $\min(e^l_k(O_k), D_i)$,

2) Body jobs: $\lfloor \frac{L_{\tau_k(O_k)}(D_i)}{T_k} \rfloor e^l_k(O_k)$,and

3) Carry-out job: $\min(e^l_k(O_k), L_{\tau_k(O_k)}(D_i) \bmod T_k)$.

where, $L_{\tau_k(O_k)}(D_i) = D_i - e^1_k(O_k) - (T_k - D_k)$.

In the following, we prove that all these individual terms increase with the parallelization option, thus making the sum of all three terms increase as well.

- Carry-in job

Here we are comparing:

$$\sum_{l=1}^{O_k} \min(e^l_k(O_k), D_i) < \sum_{l=1}^{O_k+1} \min(e^l_k(O_k+1), D_i)$$

Since the right-hand side of above equation is the sum of $O_k + 1$ threads, let us separate it into two parts: $\{1, \cdots, O_k\}$, and $\{O_k + 1\}$. This way, we can individually (thread-wise) compare left and right-hand side with thread of $\{1, \cdots, O_k\}$ and finally

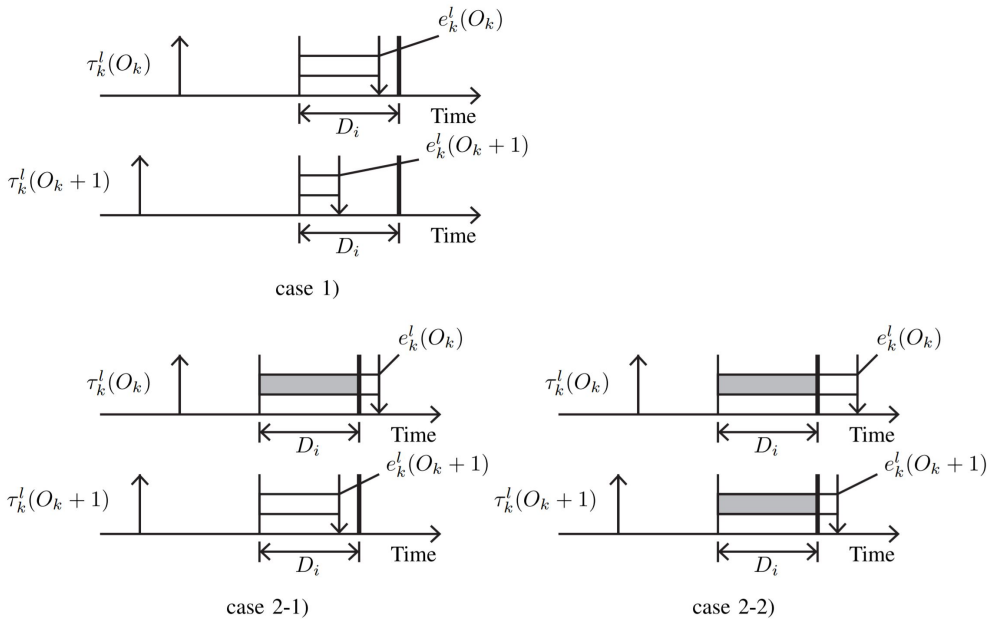


Figure 5: Worst-case workload carry-in job of $\tau_k^l(O_k)$ and $\tau_k^l(O_k+1)$ within the interval D_i.

append the last thread $\{O_k+1\}$ for the right-hand side only:

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

More specifically, we are going to show that even though the individual workload may decrease for threads with index $1 \leq l \leq O_k$, the combined workload will always increase when the last thread $l = O_k + 1$, i.e. $e_k^{O_k+1}(O_k+1)$ is considered. We will use the following term to represent the decrease of carry-in workload for the l-th thread:

$$\Delta_{carry-in}^l = \min(e_k^l(O_k), D_i) - \min(e_k^l(O_k+1), D_i).$$

Let us compare the $1, \cdots, O_k$ threads of each term side by side, with thread index

$1 \leq l \leq O_k$. As shown in Figure 5, there can only be three relations of $\tau_k^l(O_k)$ and $\tau_k^l(O_k+1)$:

Case 1) Note that the thread of higher parallelization is always shorter if not equal, i.e. $e_k^l(O_k) \geq e_k^l(O_k+1)$. Therefore in this case $\min(e_k^l(O_k+1), D_i) = e_k^l(O_k+1)$.

$$\Rightarrow \Delta_{carry-in}^l = e_k^l(O_k) - e_k^l(O_k+1).$$

In Figure 5, for each thread, the shaded part represents the included workload. For this case, we can see both thread's workload is entirely included as workload. This means that the decrease of carry-in workload is $e_k^l(O_k) - e_k^l(O_k+1)$, which is shown formally above.

Case 2) $\min(e_k^l(O_k), D_i) = D_i$:

There are two sub-cases depending on $e_k^l(O_k+1)$.

Case 2-1) $\min(e_k^l(O_k+1), D_i) = e_k^l(O_k+1)$:

$$\Rightarrow \Delta_{carry-in}^l = D_i - e_k^l(O_k+1) < e_k^l(O_k) - e_k^l(O_k+1).$$

Case 2-2) $\min(e_k^l(O_k+1), D_i) = D_i$:

$$\Rightarrow \Delta_{carry-in}^l = D_i - D_i = 0.$$

In all above cases, $\Delta_{carry-in}^l \leq e_k^l(O_k) - e_k^l(O_k+1)$. Adding up O_k threads, we get $\sum_{l=1}^{O_k} \Delta_{carry-in}^l \leq \sum_{l=1}^{O_k} e_k^l(O_k) - e_k^l(O_k+1)$. Which, as you may have noticed, is always smaller or equal to the last thread's workload $e_k^{O_k+1}(O_k+1)$ according to Eq. 12. Therefore the carry-in job's workload always increases or remain the same.

- Body job

Similarly as in the carry-in case, we divide the body job workload into two parts $\{1 \cdots O_k\}$ and $\{O_k + 1\}$, and denote the difference in body job workload of l-th thread as Δ^l_{body}:

$$\Delta^l_{body} = \lfloor \frac{L_{\tau_k(O_k)}(D_i)}{T_k} \rfloor e^l_k(O_k) - \lfloor \frac{L_{\tau_k(O_{k+1})}(D_i)}{T_k} \rfloor e^l_k(O_k + 1) \tag{13}$$

where, $L_{\tau_k(O_k)}(D_i) = D_i - e^1_k(O_k) - (T_k - D_k)$ and $L_{\tau_k(O_k+1)}(D_i) = D_i - e^1_k(O_k + 1) - (T_k - D_k)$.

Because $e^l_k(O_k) \geq e^l_k(O_k + 1)$, we can know that

$$\lfloor \frac{L_{\tau_k(O_k)}(D_i)}{T_k} \rfloor \leq \lfloor \frac{L_{\tau_k(O_{k+1})}(D_i)}{T_k} \rfloor \tag{14}$$

This means that there may be additional body job of $\tau_k(O_k + 1)$ included inside interval D_i. However for this proof, we restrict the number of body job to be the same, i.e. both have $\lfloor \frac{L_{\tau_k(O_k)}(D_i)}{T_k} \rfloor$ number of body jobs. Note that this is the harder case, and when we consider the (may existing) additional body job of $\tau_k(O_k + 1)$, the total body job workload will only increase.

Then, for the $\lfloor \frac{L_{\tau_k(O_k)}(D_i)}{/T_k} \rfloor$ body jobs, an upper-bound of Δ^l_{body} can be derived from Eq. 13: $\Delta^l_{body} \leq \lfloor \frac{L_{\tau_k(O_k)}(D_i)}{T_k} \rfloor \{e^l_k(O_k) - e^l_k(O_k + 1)\}$. Which, for$O_k$threads:

$$\sum_{l=1}^{O_k} \Delta^l_{body} \lfloor \frac{L_{\tau_k(O_k)}(D_i)}{T_k} \rfloor \leq \sum_{l=1}^{O_k} \{e^l_k(O_k) - e^l_k(O_k + 1)\}. \tag{15}$$

Lastly we compare the above value to the body job workload of the leftover $(O_k+$

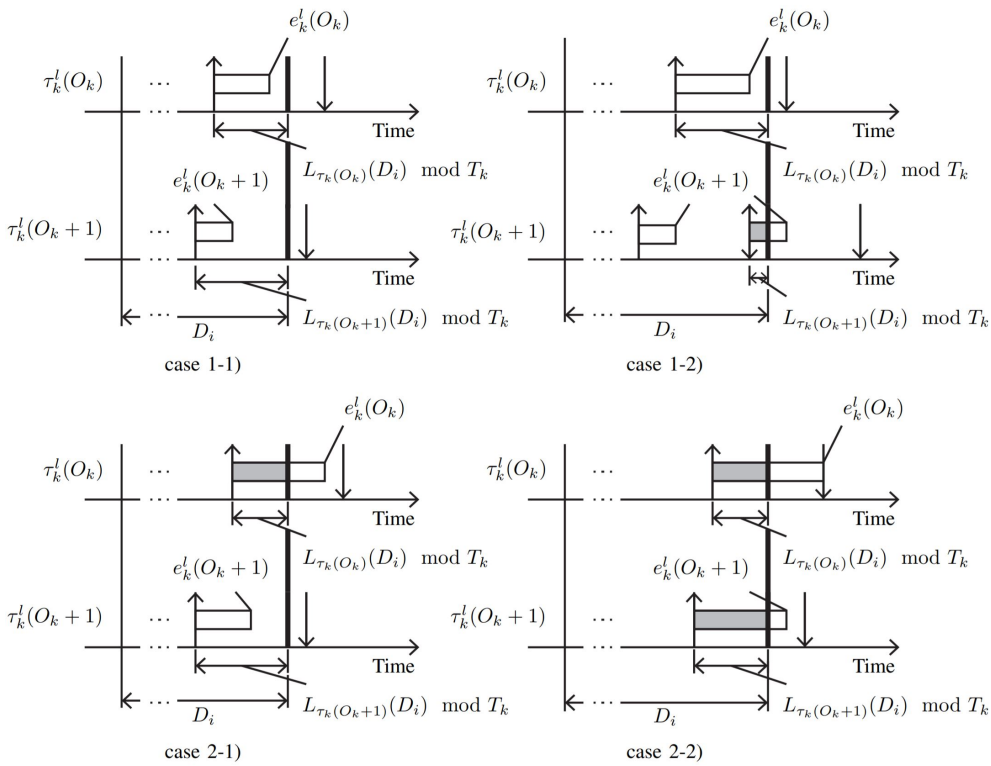


Figure 6: Worst-case workload carry-out job of $\tau_k^l(O_k)$ and $\tau_k^l(O_k+1)$ within the interval D_i.

1)-th thread, i.e. $\lfloor \frac{L_{\tau_k(O_k)}(D_i)}{T_k} \rfloor e_k^{O_k+1}(O_k+1)$. Likewise, according to Eq. 12, the addition of the leftover thread ((O_k+1)-th thread) exceeds or equal to the value of Eq. 15. Therefore the sum of body job's workload always increases or remains the same.

- Carry-out job

Similar for the body job case, we denote the difference in carry-out job workload of l-th thread as $\Delta_{carry-out}^l$.

$$\Delta^l_{carry-out} = \min(e^l_k(O_k), L_{\tau_k(O_k)}(D_i) \bmod T_k) \\ - \min(e^l_k(O_k+1), L_{\tau_k(O_k+1)}(D_i) \bmod T_k), \tag{16}$$

where, $L_{\tau_k(O_k)}(D_i)$ and $L_{\tau_k(O_k+1)}(D_i)$ each defined the same as the body job case.

There are four possible cases of $\tau^l_k(O_k)$ and $\tau^l_k(O_k+1)$'s relation, as depicted in Figure 6. Likewise, the included part of the carry-out workload is represented as a shaded box in the figure:

Case 1) $\min(e^l_k(O_k), L_{\tau_k(O_k)}(D_i) \bmod T_k) = e^l_k(O_k)$: We can further branch into the following two sub-cases.

Case 1-1) $\min(e^l_k(O_k+1), L_{\tau_k(O_k+1)}(D_i) \bmod T_k) = e^l_k(O_k+1)$:

$$\Rightarrow \Delta^l_{carry-out} = e^l_k(O_k) - e^l_k(O_k+1).$$

This is when both thread's carry-out job is entirely included as workload, as shown in Figure 6. The decrease of included carry-out job workload is $e^l_k(O_k) - e^l_k(O_k+1)$, as formally shown above.

Case 1-2) $\min(e^l_k(O_k+1), L_{\tau_k(O_k+1)}(D_i) \bmod T_k)$

$$= L_{\tau_k(O_k+1)}(D_i) \bmod T_k :$$

This is an odd case when the included carry-out workload seems to decrease, i.e. $\Delta^l_{carry-out} < 0$. However, this is not true. It is because in this case, an additional body job must have existed for $\tau_k(O_k+1)$, and we intentionally left it out of consideration from the body job's proof. The existence of an additional job can be

shown in the following way: We know that $e_k^l(O_k) \geq e_k^l(O_k+1)$. Also we know that $\lfloor \frac{L_{\tau_k(O_k)}(D_i)}{T_k} \rfloor \leq \lfloor \frac{L_{\tau_k(O_k+1)}(D_i)}{T_k} \rfloor$ from Eq. 14.

Because we are applying a same modulo T_k operations to both $L_{\tau_k(O_k)}(D_i)$ and $L_{\tau_k(O_k+1)}(D_i)$, the only way case 1-2) could arise is when $\lfloor \frac{L_{\tau_k(O_k)}(D_i)}{T_k} \rfloor < \lfloor \frac{L_{\tau_k(O_k+1)}(D_i)}{T_k} \rfloor$.

Therefore for $\tau_k(O_k+1)$, there is a carry-out job plus an additional entire body job. Therefore, the carry-out workload always increases for this case.

Case 2) $\min(e_k^l(O_k), L_{\tau_k(O_k)}(D_i) \bmod T_k) = L_{\tau_k(O_k)}(D_i) \bmod T_k$:

We can further branch into the following two sub-cases.

Case 2-1) $\min(e_k^l(O_k+1), L_{\tau_k(O_k+1)}(D_i) \bmod T_k) = e_k^l(O_k+1)$:

$$\Rightarrow \Delta^l_{carry-out} = L_{\tau_k(O_k)}(D_i) \bmod T_k - e_k^l(O_k+1) < e_k^l(O_k) - e_k^l(O_k+1)$$

Case 2-2) $\min(e_k^l(O_k+1), L_{\tau_k(O_k+1)}(D_i) \bmod T_k) = L_{\tau_k(O_k+1)}(D_i) \bmod T_k$:

$$\begin{aligned}
&\Rightarrow \Delta^l_{carry-out} \\
&= L_{\tau_k(O_k)}(D_i) \bmod T_k - L_{\tau_k(O_k+1)}(D_i) \bmod T_k \\
&= \{D_i - e_k^l(O_k) - (T_k - D_k)\} - \{D_i - e_k^l(O_k+1) - (T_k - D_k)\} \\
&= e_k^l(O_k) - e_k^l(O_k+1).
\end{aligned}$$

In all above cases, $\Delta^l_{carry-out} \leq e_k^l(O_k) - e_k^l(O_k+1)$. Adding up O_k threads, we get $\sum_{l=1}^{O_k} \Delta^l_{carry-out} \leq \sum_{l=1}^{O_k}(e_k^l(O_k) - e_k^l(O_k+1))$. This, also according to Eq. 12 is smaller than or equal to the last thread's workload $e_k^(O_k+1)(O_k+1)$. Therefore the carry-out job's workload always increases or remain the same.

Summing the results, all carry-in, body, and carry-out job's workload increase

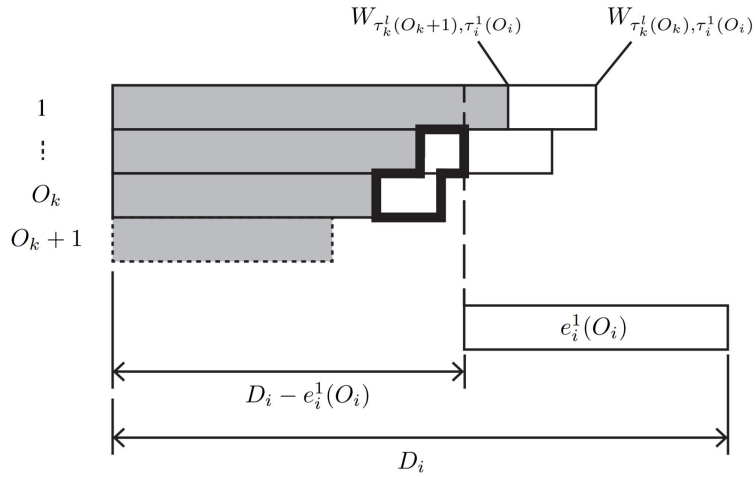


Figure 7: Change of workload bound from τ_k to τ_i for $O_k \rightarrow O_k + 1$.

with the increase of parallelization. Therefore the total workload of τ_k within τ_i's interval increases when the parallelization option increases, i.e. $W_{\tau_k,\tau_i}(O_k) < W_{\tau_k,\tau_i}(O_k+1)$.

For our last step, we will show using Figure 7. that the interference function also increases with the parallelization option. In Figure 7, each horizontal bar represents τ_k's individual total workload inside τ_i^1's interval, i.e. D_i. Note that this includes all the workload from carry-in, body and carry-out jobs of τ_k. In the figure, $\tau_k(O_k)$thread's workload is represented as horizontal bar, and $\tau_k(O_k + 1)$thread's workload is shown as a shaded portion of the bar.

The interference function is defined in Eq. 4 as:

$$\overline{W}^{interference}_{\tau_k,\tau_i}(O_k) = \sum_{l=1}^{O_k} \overline{W}_{\tau_k^l(O_k),\tau_i^1(O_i)} = \sum_{l=1}^{O_k} \min(W_{\tau_k^l(O_k),\tau_i^1(O_i)}, D_k - e_i^1(O_i)).$$

This means that for each bar in the figure, the part left to the dashed line $D_i - e_i^1(O_i)$ is counted as interference.

As we proved in the previous step, sum of the shaded area, i.e. $W_{\tau_k,\tau_i}(O_k+1)$, is larger than equal to the sum of the entire white bars (including the overlapping shaded portion), i.e. $W_{\tau_k,\tau_i}(O_k)$. Thus $\tau_k(O_k+1)$'s last thread, depicted as dotted shaded bar is larger than the white area surrounded by the thick solid line. The lemma follows. □

To sum up all the above discussions, the schedulability condition of τ_k can be finally written, using both the monotonic increasing property of tolerance and interference as follows:

$$\overline{W}^{interference}_{\tau_k,\tau_i}(O_k) < \overline{W}^{interference}_{\tau_k,\tau_i}(O_k+1) \tag{17}$$

5 Optimal Parallelization Options Assignment Algorithm

In the previous section, two monotonic increasing property of parallelization freedom, and their trade-off has been introduced. As the parallelization option increases for τ_k, its "tolerance" monotonically increases. However its "interference" to other task τ_i also increases.

This hints that a great way to make Γ schedulable, is to assign only the "barely tolerable" option for each task $\tau \in \Gamma$. In other words, each tasks should try to keep its parallelization option low, because increasing the option beyond its current need is pointless, and would only result in interfering other task more.

This was an original motivation in [9], where task would start from lowest parallelization, i.e. no parallelization, and iteratively granted a "barely tolerable option" over calculated current interference. This is called the 'Optimal Parallelization Op-

tion Assignment (OPOA)' algorithm. OPOA was originally targeted for global EDF scheduling policy only. However we discovered that the requirement for OPOA, i.e. monotonic increasing property of both tolerance and interference function also holds for global FP case, and thus OPOA is directly applicable for the global FP case as well. In the following we introduce the extended OPOA algorithm converted and targeted for global FP.

An illustration of OPOA for global FP using two tasks τ_1, τ_2 is depicted in Fig 8. In each sub figures, the horizontal axis represent the received interference from all higher priority tasks, i.e. $\overline{W}^{interference}_{\tau_i,\tau_k}(O_k)$. The tolerance according to the parallelization of each task is marked on the axis, i.e. $\overline{W}^{tolerance}_{\tau_k}(O_k)$, and in this case the maximum parallelization option is 4. Above the marked tolerance, the current chosen option is shown with a check 'checkmark', indicating the task is parallelized into the checked option.

Initially all tasks start with no-parallelization, which is shown in left-side of Fig 8(a). Next in Fig 8(b), task with highest priority is drawn, and its received interference is calculated, and the value is marked underneath the axis with a triangle. This case it is the first thread τ_1, and its received interference is zero, because it has the highest priority among all. Then the option with barely schedulable tolerance for τ_1 is selected, which in this case is $O_1 = 1$. In Fig 8(c) this process is repeated for τ_2, but this time τ_2 experiences interference from τ_1 only, and thus τ_2 is raised to option 2 to tolerate the received interference.

The right side of Figure 8 shows when two or more tasks have the same priority. In this case, τ_3 and τ_4 has the same priority, and thus iterative measure has to be applied. Firstly, we calculate the received interference of both tasks. Note that τ_3

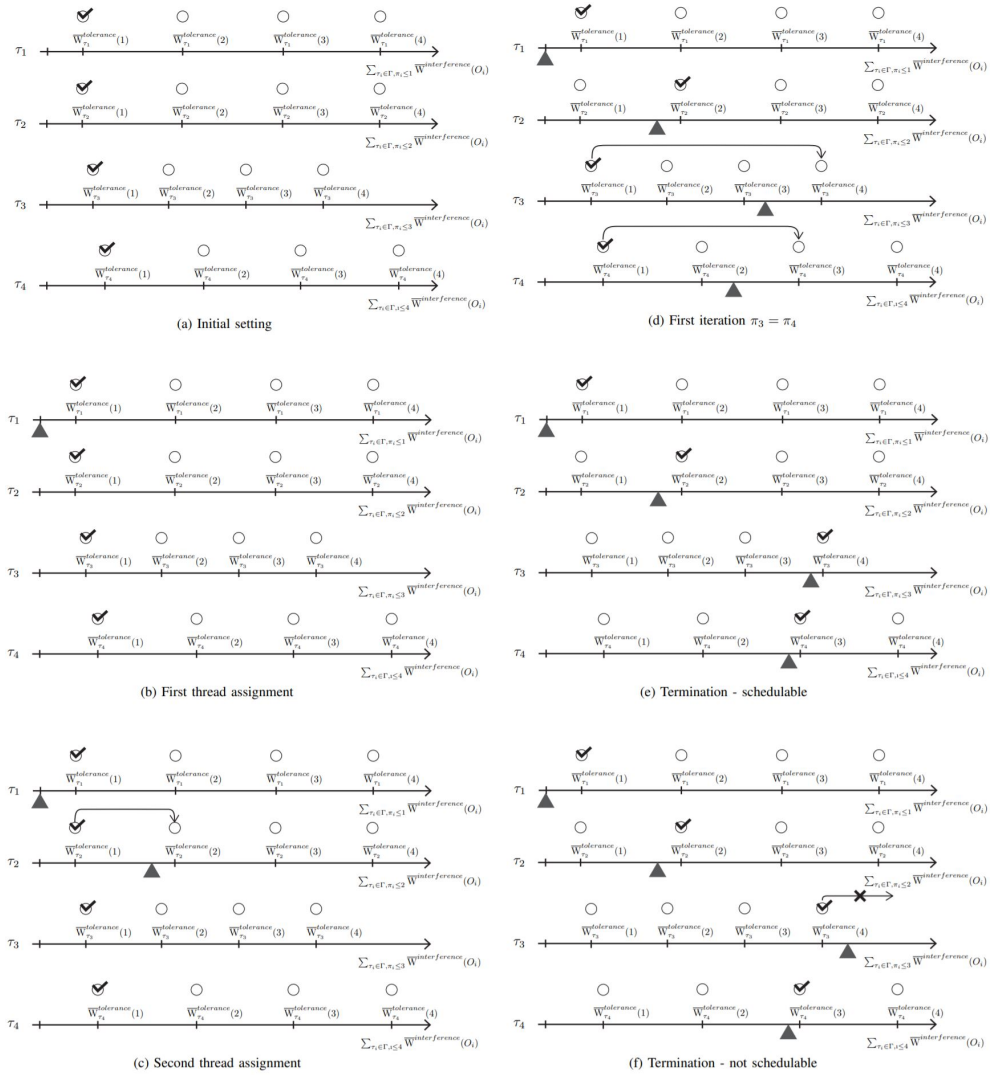


Figure 8: Optimal parallelization option assignment

and τ_4 interfere each other mutually according to the selected option. If the received interference exceeds the current tolerance for any task, we raise its parallelization option likewise to the barely tolerable option Fig 8(e). This step is repeated until one of the following condition is met: (1) For all tasks, its tolerances is greater than its received interference Fig. 8(d) — schedulable case according to Eq. 19, or (2) For any task, it reached its maximum parallelization but cannot tolerate the received interference Fig. 8(f) — unschedulable case.

The pseudo-code in Algorithm 1 describes the OPOA algorithm for global FP. The algorithm takes a set of tasks Γ as input, and returns two outputs. 1) Schedulability of the given task set and 2) Parallelization option combination $\mathbb{O}$, which represents a parallelization option of each task when the task set is schedulable. The for-loop in Line 1 through Line 5 calculates and stores the tolerance $\overline{W}_{\tau_k}^{tolerance}(O_k)$ depends on parallelization option O_k according to Eq. 7. The initialization in Line 6 creates an empty group $\mathbb{G}$, where which means each task τ_k's schedulability is yet unknown.

The initialization in Line 11 creates an array $\mathbb{O}^{cur}$ representing the current selected parallelization option for each tasks, which currently all of them is initialized as 1. The initialization in Line 12 creates an 'updated' flag, which represents that at some point in the following iteration some task option has been updated. The for-loop from Line 13 to Line 31 check the schedulability for each task set g in same priority group mathbbGuntil it is decided 'schedulable' or 'not schedulable'. Note that each step starts from the next highest priority group. The while-loop from Line 14 to Line 29 iteratively compares the tolerance and the step-wise calculated interference of each task and will end when there is no more updates. For each iteration, Line 15 sets the 'updated' flag back to false, and at the end of the while-loop, if still 'up-

Algorithm 1: Optimal Parallelization Option Assignment (OPOA) for global FP scheduling

Input: Set of tasks $\Gamma = \tau_1, \tau_2, \cdots, \tau_n$
Output: (1) Schedulability, (2) Parallelization option combination $O = (O_1, O_2, \cdots, O_n)$

1 **for** $\tau_k \in \Gamma$ **do**
2 **for** $O_k = 1$ *to* O_k^{max} **do**
3 $\overline{W}_(\tau_k)^{tolerance}(O_k) \leftarrow$ Eq.7
4 **end**
5 **end**
6 initialize $\mathbb{G} \leftarrow emptygroup$
7 **for** $\tau_k \in \Gamma$ **do**
8 $\mathbb{G}[\pi_k].append(\tau_k)$
9 **end**
10 initialize $\mathbb{S} \leftarrow (Unknown, Unknown, \cdots, Unknown)$
11 initialize $\mathbb{O}^{cur} \leftarrow (1, 1, \cdots, 1)$
12 initialize updated $\leftarrow$ True
13 **for** *next highest priority* $g \in \mathbb{G}$ **do**
14 **while** *updated* **do**
15 updated $\leftarrow$ False
16 $\mathbb{O}_k^{pre} \leftarrow \mathbb{O}_k^{cur}$
17 **for** $\tau_k \in g$ **do**
18 $\sum_{\tau_i \in \Gamma, \tau_i d \tau_k} \overline{W}_{\tau_i,\tau_k}^{interference}(O_i^{pre}) \leftarrow$ Eq.11
19 **while** $\overline{W}_{\tau_k}^{tolerance}(\mathbb{O}_k^{cur}) < \sum_{\tau_i \in \Gamma, \pi_i d \pi_k} \overline{W}_{\tau_i,\tau_k}^{interference}(O_i^{pre})$ **do**
20 $\mathbb{O}_k^{cur} \leftarrow \mathbb{O}_k^{cur} + 1$
21 updated $\leftarrow$ True
22 **if** $\mathbb{O}_k^{cur} > O_k^{max}$ **then**
23 $S_k \leftarrow$ False and $\mathbb{O}_k^{cur} \leftarrow O^{max}$
24 break //goto Line 17, continue with next τ_k
25 **end**
26 **end**
27 **end**
28 **if** *any* $S_k \in \mathbb{S}$ *is False* **then**
29 break
30 **end**
31 **end**
32 **if** *any* $S_k \in \mathbb{S}$ *is False* **then**
33 break
34 **end**
35 **end**
36 **if** *any* $S_k \in \mathbb{S}$ *is False* **then**
37 return not schedulable
38 **end**
39 **else**
40 return schedulable, $\mathbb{O}^{cur}$
41 **end**

dated' flag is false, then we will terminate the while-loop. Line 16 copies and stores the current parallelization option combination $\mathbb{O}_k^{cur}$ as $\mathbb{O}_k^{pre}$. Then, the for-loop from Line 17 to Line 27 calculates the total interference and increases the parallelization options till the 'barely tolerable parallelization option' of each task τ_k. Line 18 calculates the total interference from higher or equal priority tasks using Eq. 11 with previous parallelization option combination $\mathbb{O}^{pre}$. Then, the while-loop from Line 19 to 26 compares the tolerance with the calculated interference. If the tolerance is smaller, then Line 20 increases the parallelization option $\mathbb{O}_k^{cur}$ one at a time and Line 21 sets the 'update' flag as true. Line 22 checks whether $\mathbb{O}_k^{cur}$ exceeded the limit O^{max}, and when it does Line 23 sets S_k as a false, meaning the task is unschedulable and $\mathbb{O}_k^{cur}$ back to O^{max}. Also in such case Line 24 breaks the while-loop from Line 19 to Line 26 and goto Line 17 to continue with next τ_k to calculate rest of the tasks before terminating. After checking all task inside the priority group g , Line 28 checks whether any $S_k \in \mathbb{S}$ was false, and if true it breaks the for-loop from Line 17 to Line 29 and goto Line 30 then breaks the while-loop from Line 14 to Line 29 and finally return 'not schedulable' in Line 33. If all $S_k \in \mathbb{S}$ was true, however, we goto Line 29 to check the 'updated' flag. If 'updated' flag is true then we goto Line 14 to continue iteration. After the while-loop from Line 14 to Line 29. The if-else statement from Line 32 to Line 36 returns the output 'schedulable' or 'not schedulable'. And if the output is 'schedulable', then the parallelization option combination $\mathbb{O}^{cur}$ is also returned.

Instead of exhaustive search, Algorithm 1 executes a one-way search in the for-loop from Line 13 to Line 31 using the monotonic property of both tolerance and interference. For n tasks with maximum O^{max} parallelization option, the maximum

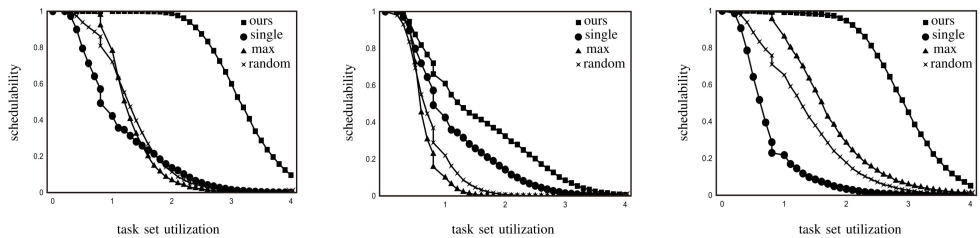
increase of options is at most $n \cdot O^{max}$. The maximum number of group $g \in G$ is n, hence O(n) complexity. And the for-loop from Line from 17 to Line 27 is $O(n^2)$. In each iteration, the algorithm calculate the interference in Line 18 with the time complexity of O(n) for all n tasks by the for-loop form Line 17 to Line 27. Therefore, the overall time complexity of Algorithm 1 is $O(n^4)$.

6 Experiment

6.1 Simulation Results

In this section, we show the performance of the proposed OPOA for global FP algorithm through results from extensive simulation [7]. For our simulation, a total of 10^6 different task sets were evaluated. Each task $\tau_k = (\pi_k, T_k, D_k, e_k)$ was generated in the following way: (1) π_k randomly drawn from uniform integer[0,10]. (2) T_k randomly selected from uniform[500,3000]. (3) D_k drawn from uniform[400,T_k]. (4) Execution time table e_k constructed with $e_k^l(O_k = 1) = C_k(O_k = 1) = uniform[300, 1000]$. (5) Thread execution time of higher parallelization option $O_k+1, \forall O_k, 1 \leq O_k < m, C_k(O_k + 1)$ is computed using $\alpha(O_k \rightarrow O_k + 1) = \frac{C_k(O_k+1)-C_k(O_k)}{e_k^1(O_k)-e_k^1(O_k+1)}$, i.e. Eq. 10. (6) $C_k(O_k + 1)$ is divided into $O_k + 1$ threads by Unifast algorithm [15], and those threads are sorted in descending order of their thread execution time, i.e., $e_k^l(O_k) \geq e_k^l(O_k + 1), \forall l, 1 \leq l \leq O_k$.

Using the task generated by the above method, each task set is created in the following way: (1) Create an empty task set Γ. (2) Generate a new task τ_k and append to Γ. (3) Check whether Γ passes necessary schedulability test, i.e. $\sum_{\tau_k \in \Gamma} \frac{C_k(1)}{T_k} < m$. (4) If it passes the test, sort Γ in descending order of each tasks priority π_k, and



(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 9: Simulation result with m=4 CPU cores.

return Γ. Afterwards, repeat from step (2). (5) If Γ fails the necessary test, discard Γ, and start anew from step (1).

We run the simulation using the generated task sets with m=4 CPU cores, and compare the schedulability result of the following four different parallelization approaches:

- Ours: Each task parallelized using Algorithm 1
- Single: No-parallelization. All task, single threaded.
- Max: Maximum parallelization. Every task is parallelized into the maximum possible parallelization option, i.e., $O_k = O^{max} = m$.
- Random: Each task is parallelized with randomly chosen parallelization option from uniform $[1, O^{max}]$.

The experiment result is shown in Figure (9) In each sub-figures, the x-axis represent the task set utilization $U_k = \sum_{\tau_k \in \Gamma} C_k(1)$, and the y-axis represent the schedulability. Fig 10(a) compares the schedulability of the four approaches when the parallelization overhead $\alpha(O_k \rightarrow O_k + 1) = 0.3$ for every $O_k < O^{max} = m = 4$. First comparing "Single" and "Max", we can see that while "Max" shows better per-

formance in the low task set utilization region, it performs poorly at high task set utilization region, and the trend flips for the "Single" approach, where it performs better at high utilization. This shows that in low utilization, the schedulability is mostly affected by deadlines, and thus reducing the execution time of the threads by parallelizing into higher option is advantageous. On the contrary, in the case with higher utilization, schedulability is mostly shaped by the number of tasks that are in the system. Therefore keeping the number of threads small by reducing parallelization option is a better strategy. "Random" performs in-between "Single" and "Max". "Ours" significantly out performs in all task set utilization. This is because the OPOA algorithm assigns optimal parallelization option through balancing trade-off relation of tolerance and interference.

Fig 10(b) shows the result parallelization overhead is much higher, i.e., $\alpha(O_k \rightarrow O_k + 1) = 0.8$ for every $O_k \leq O^{max} = m = 4$. Looking at "Single", it is not affected by higher parallelization overhead, since no thread undergoes parallelization. On the other hand, performance of "Max" is greatly hindered, because benefiting from parallelization became much harder. "Random" is between "Single" and "Max". Here, "Ours" is also affected by the high parallelization overhead, but still performs much better than any other approaches.

Fig 10(c) shows the result when all task's deadline is tightened by 20%. This emphasizes the deadline's effect to the schedulability. Thus for "Single" strategy, since each thread's execution time is large, is greatly affected by the reduced deadline, with highly reduced performance compared to previous experiments. However the "Max" strategy, which is affected less by the reduced deadline, performs similarly to the base experiment, i.e. Fig 10(a). "Ours" is similarly affected but still performs best

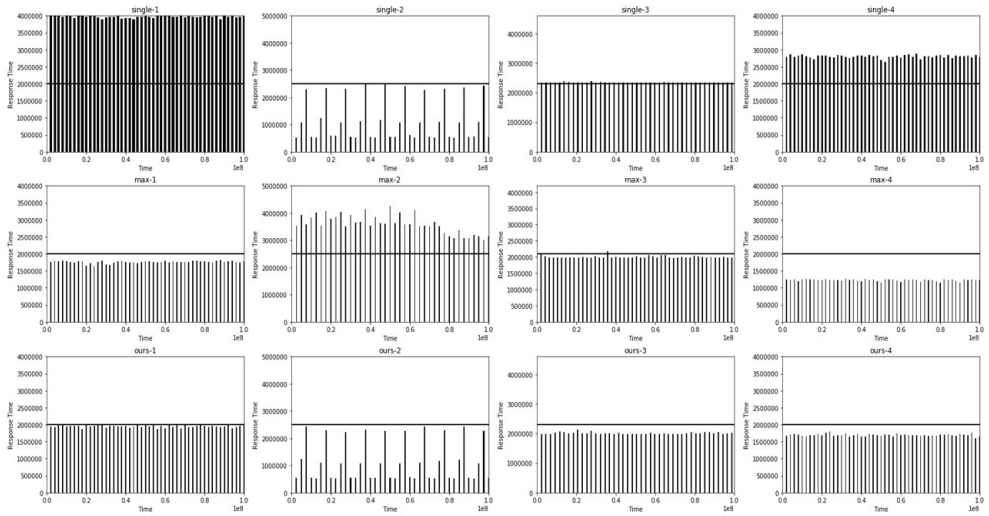


Figure 10: Implementation results

in all utilization.

6.2 Implementation Results

To justify the practicality of our proposed approach, we implemented real-time task sets using Linux kernel 4.19 with PREEMPT_RT patch. The tasks are scheduled according to G-FP (SCHED_RR) scheduler on PC with Intel Core i7-8700 CPU (6-cores). We use CPUSET to set the running of our tasks in m = 4 cores, fix the clock frequency at 3.20 GHz, and disable GPU.

On this system, we execute the four real programs used for autonomous driving, (1)τ_1: sensor read program emulating a multi-channel camera module, (2) τ_2: lane tracking program used by Autoware, (3) τ_3: darknet-based object detection and labeling program used by APOLLO, and (4) τ_4: steering actuation program emulating PID motor controller. Their minimum inter-release times and deadlines are set

as $(T_1 = D_1 = 2500000), (T_2 = D_2 = 2500000), (T_3 = D_3 = 2300000), (T_4 = D_4 = 2000000)$, where the time units is μs.

Fig. 10 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 if the response time exceed this line, it means a deadline violation. In "single", we can observe many deadline misses of τ_1 and τ_4. This is because they are heavy workload tasks and without parallelization, their execution times are longer than deadlines. In "max", the τ_1's and τ_4's deadline misses decrease because of the parallelization but now we observe deadline misseds of the τ_2 and τ_3 due to the increased interference. On the other hand, we can observe that "ours" meet all the deadlines of all the four tasks by optimal parallelization.

7 Conclusion

In this paper, we presented an algorithm that optimally assign parallelization option for global FP. To do this, we extended the BCL schedulability condition for tasks with parallelization freedom scheduled with global FP. Next, a trade-off relation of a task's tolerance and interference was identified, by deriving a monotonic increasing property of parallelization. Using the properties, an optimal parallelization assignment algorithm for global FP with polynomial time complexity was derived. Through extensive simulation we showed a significant improvement of schedulability. Our plan is to extend the approach to accommodate other schedulers, and to other task models such as a DAG model [17].

References

[1] J. H. Anderson and J. M. Calandrino. Parallel real-time task scheduling on multicore platforms. In *27th IEEE Real-Time Systems Sympo-sium*, RTSS '06. IEEE, 2006.

[2] Andersson and Jonsson. Fixed-priority preemptive multiprocessor scheduling: to partition or not to partition. In *7th International Conference on Real-Time Computing Systems and Applications*, RTCSA '00. IEEE, 2000.

[3] Ayguade, Copty, and Duran. The design of openmp tasks. *IEEE Transactions on Parallel and Distributed Systems*, 20:404–418, 2008.

[4] TP Baker. Multiprocessor edf and deadline monotonic schedulability analysis. In *24th IEEE Real-Time Systems Symposium*, RTSS '03. IEEE, 2003.

[5] Bertogna, Cirinei, and Lipari. Schedulability analysis of global scheduling algorithms on multiprocessor platforms. *IEEE Transactions on Parallel and Distributed Systems*, 20:553–566, 2008.

[6] Bertogna, Marko, and Cirinei. Improved schedulability analysis of edf on multiprocessor platforms. In *17th Euromicro Conference on RealTime Systems*, ECRTS '05, pages 209–218. IEEE, 2005.

[7] Bini and Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30:129–154, 2005.

[8] J.-J. Chen and S. Chakraborty. Partitioned packing and scheduling for sporadic real-time tasks in identical multiprocessor systems. In *24th Euromicro Conference on Real-Time Systems*, ECRTS'12. IEEE, 2012.

[9] Youngeun Cho, Dohyung kim, Daechul Park, Seungsu Lee, , and Chang-Gun Lee. Conditionally optimal parallelization for global edf on multi-core systems. In *40th IEEE Real-Time Systems Symposium*, RTSS '19. IEEE, 2019.

[10] Guo, Santinelli, and Yang. Edf schedulability analysis on mixed-criticality systems with permitted failure probability. In *21st International Conference on Embedded and Real-Time Computing Systems and Applications*, RTCSA '15. IEEE, 2015.

[11] S. Funk J. Goossens and S. Baruah. Priority-driven scheduling of periodic task systems on multiprocessors. *Real-time systems*, 25:187–205, 2003.

[12] Kim, Cho, et al., and Han. System-wide time versus density tradeoff in real-time multicore fluid scheduling. *IEEE Transactions on Computers*, 18:7, 2018.

[13] Kwon, et al., and Lee. Multicore scheduling of parallel real-time tasks with multiple parallelization options. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, RTETAS '15. IEEE, 2015.

[14] Kato Lakshmanan and Rajkumar. Scheduling parallel real-time tasks on multicore processors. In *31th IEEE Real-Time Systems Symposium*, RTSS '10. IEEE, 2010.

[15] F Markovic, J Carlson, and R Dobrin. Preemption point selection in limited preemptive scheduling using probabilistic preemption costs. In *28th Euromicro Conference on Real-Time Systems*, ECRTS '16. IEEE, 2016.

[16] Daechul Park, Youngeun Cho, and Chang-Gun Lee. Conditionally optimal parallelization for global fp on multi-core systems. In *3th International Conference on Information and Computer Technologies*, ICICT '20. IEEE, 2020.

[17] A. Saifullah, J. Li D. Ferry, K. Agrawal, C. Lu, and C. D. Gill. Parallel real-time scheduling of dags. *Transactions on Paralle and Distributed Systems*, 25:3242–3252, 2014.

[18] Sheng, Gao, Xi, and Zhou. Schedulability analysis for multicore global scheduling with model checking. In *11th International Workshop on Microprocessor Test and Verification*, IWMTV'10. IEEE, 2010.

[19] A. Srinivasan and S. Baruah. Deadline-based scheduling of periodic task systems on multiprocessors. *Information Processing Letters*, 84:93–98, 2002.

[20] Stone, Gohara, Shi, and et al. Opencl: A parallel programming standard for heterogeneous computing. *Computing in science and engineering*, 12:66–73, 2010.

요약(국문초록)

지난 몇 십년 간, 컴퓨테이션 요구의 증가와 함께 병렬 컴퓨팅의 중요성이 크게 증대되고 있다. OpemMP나 OpenCL과 같은 프레임워크들은 컴퓨팅 테스크를 원하는 수의 쓰레드로 쉽게 병렬화 할 수 있도록 한다. 이를 본 논문에서는 '병렬화 자유도'라고 명명했다. 그러나 병렬화는 댓가를 필요로 하며 병렬화 옵션(각 태스크를 병렬화 하는 쓰레드의 갯수)에 따른 병렬화 오버헤드를 발생시킨다. 이에 주어진 컴퓨팅 리소스의 보다 나은 활용을 위해 병렬화 옵션은 신중하게 정해질 필요가 있다. 본 논문은 global FP 스케쥴러 상에서 각 테스크에 병렬화 옵션을 할당하는 문제를 다룬다. 이를 위해 기존에 global EDF 상으로 한정되어 있는 조의 접근방법을 확장한다. global FP 상에서도 병렬화 옵션의 조건적 최적 할당이 존재함을 증명하였다. 또한 광범 위한 시뮬레이션과 자율주행 태스크셋을 통해 스케쥴러빌리티의 확실한 향상을 보였다.

주요어 : 스케쥴링, 병렬화자유도, global FP
학 번 : 2016-21201