M.S. THESIS

Scheduling Algorithms for Parallel Real-Time Tasks with Multiple Parallelization Options on Multicore/GPGPU System

멀티코어/GPGPU 시스템 상에서 복수 병렬화 옵션을 가지는 병렬 실시간 태스크 스케줄링 방법

AUGUST 2014

서울대학교 대학원

전기·컴퓨터공학부

권 지 혜

Scheduling Algorithms for Parallel Real-Time Tasks
with Multiple Parallelization Options
on Multicore/GPGPU System

지도교수 이 창 건

이 논문을 공학석사학위논문으로 제출함

2014 년 8 월

서울대학교 대학원

전기·컴퓨터공학부

권 지 혜

권지혜의 석사학위논문을 인준함

2014 년 8 월

위 원 장 : 김 형 주 (인)

부위원장 : 이 창 건 (인)

위 원 : 이 재 진 (인)

Abstract

Scheduling Algorithms for Parallel Real-Time Tasks with Multiple Parallelization Options on Multicore/GPGPU System

Jihye Kwon

School of Electrical Engineering and Computer Science

College of Engineering

The Graduate School

Seoul National University

Past researches on multicore/GPGPU scheduling assume that a computational unit has a pre-fixed number of CPU and GPU threads. However, with recent technologies such as OpenCL, a computational unit can be parallelized in many different ways with runtime selectable numbers of CPU and GPU threads. This paper proposes algorithms for optimally parallelizing and scheduling a set of parallel tasks with multiple parallelization options on multiple CPU cores and multiple GPU devices. Our experimental study says that the proposed algorithms can successfully schedule up to two times more tasks compared with other algorithms assuming pre-fixed parallelization. To the best of our knowledge, this is the first work addressing the

problem of scheduling parallel tasks with multiple parallelization options on multiple heterogeneous resources.

keywords : parallelization, multicore, gpgpu, real-time, scheduling

student number : 2012-20739

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Along with increasing features and advanced functionalities, many real-time applications become more and more data-intensive and computation-intensive. For example, a vehicle with vision, lidar, radar, and GPS sensors can be augmented into an autonomously driving vehicle via real-time processing of massive sensory data and sophisticated control algorithms [1]. For such real-time handling of massive data and sophisticated algorithms, it has been a recent hot topic to optimally use the multiple CPU cores, and GPU (Graphics Processing Unit) devices for GPGPU (General Purpose computing on GPU) [2]. For example, [3] and [4] address the problem of scheduling sequential tasks using homogeneous CPU cores. On the other hand, [5] and [6] tackle scheduling of parallel tasks where each parallel task is modeled as a sequence of segments and each segment has a number of parallel threads. [7] addresses more general parallel tasks where each parallel task is modeled as a DAG (Directed Acyclic Graph) and each vertex in the DAG has one sequential thread. In addition to these works addressing CPU resources only, [2] addresses scheduling sequential tasks on multiple CPU cores and GPU devices.

All these works consider that each computational unit, i.e., a task in [2], [3],

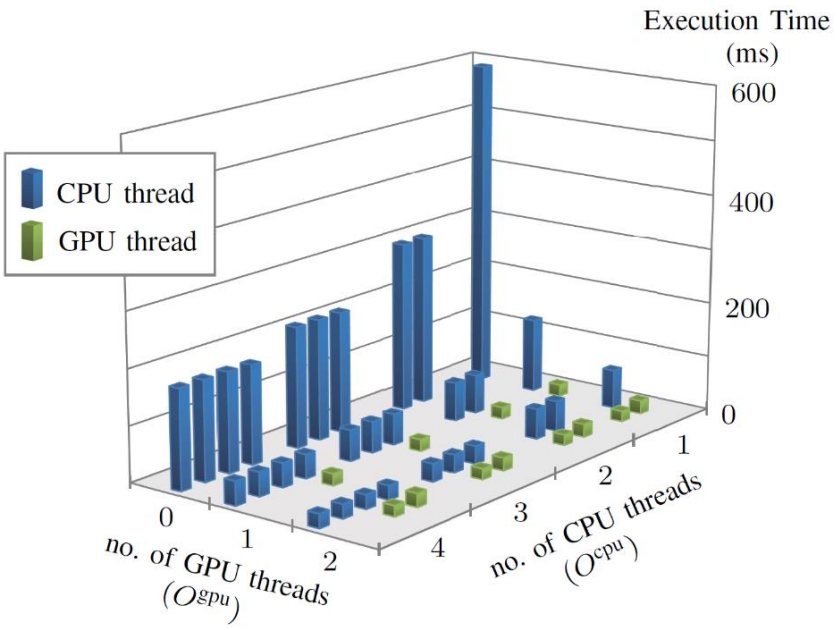


Figure 1.1 Measured thread execution times for an edge-detection program

(Intel Core i7, 2 × Nvidia GeForce GTX 650 Ti)

[4], a segment in [5], [6] and a vertex in [7], is implemented as a pre-fixed number of threads. However, recent heterogeneous programming frameworks such as OpenCL [8] support multi-version parallelism even for a single computational unit. More specifically, an OpenCL kernel (main program) can be parallelized into a different number of CPU/GPU threads with runtime arguments and launched on that number of CPU cores and GPU devices. Depending on the number O^{cpu} of CPU threads and the number O^{gpu} of GPU threads that the program is parallelized into, its corresponding thread execution times significantly vary. Figure 1.1 shows the variation of the thread execution times of each parallelization option, i.e., $(O^{\mathrm{cpu}}, O^{\mathrm{gpu}})$ for an edge-detection program that can be used for lane detection in an autonomously driving vehicle application. The figure clearly shows the significant

differences among thread execution times of different runtime parallelization options.

Motivated by this finding, this paper aims at exploiting the freedom of selecting one out of multiple parallelization options to improve the schedulability of given set of parallel tasks. Specifically, we address the problem of optimally parallelizing and scheduling a set of sporadic parallel tasks, where each task is described by a DAG (Directed Acyclic Graph) and each vertex of the DAG has multiple runtime parallelization options on multiple preemptive/non-preemptive resources. In this problem, we consider a CPU core as a representative preemptive resource and a GPU device as a representative non-preemptive resource. For the given problem, we first tackle a simplified problem where the resource model has only CPU cores not GPU devices and the DAG of each task is a simple directed sequence of vertexes. For this simplified problem, we propose an optimal algorithm. Second, we extend this algorithm for the heterogeneous resource model with both CPU cores and GPU devices. Finally, we further extend the algorithm for the general DAG-based task model. Our experimental study says that our proposed algorithms can successfully schedule up to two times more tasks than other algorithms assuming pre-fixed parallelization. To the best of our knowledge, this is the first work addressing the problem of scheduling parallel tasks with multiple parallelization options on multiple CPU cores and multiple GPU devices.

The rest of this thesis is organized as follows. Chapter 2 surveys the related works on real-time multicore scheduling algorithms. Then, Chapter 3 describes the target problem. In Chapter 4, we explain our proposed scheduling algorithms. Chapter 5 presents the experimental results. Finally, Chapter 6 concludes the thesis.

Chapter 2

Related Works

There have been numerous researches on real-time multi-core scheduling since Dhall and Liu first addressed it in 1978 [9]. They can be grouped as in Table 1.1 according to the workload and resource models. For the "sequential workload model" where each task is a sequential program, the first group [10], [11], [12], [13], [14], [15], [3], [4], [16] addresses the problem of scheduling the tasks on multiple homogeneous resources. Specifically, in 1993, [10] first presented an optimal algorithm for scheduling periodic sequential tasks on homogeneous multiprocessors. The algorithm was theoretically optimal, in terms of the competitiveness in schedulability, but it was impractical due to its heavy scheduling, preemption, and migration overheads. Thereafter, many researchers have proposed optimal scheduling algorithms with less overheads [11], [12], [13], [14], [15], [3], [4], and [16].

The second group for the sequential workload model [17], [2] includes GPUs in their resource model. However, each job is dedicated to a specific resource type, either a CPU core or a GPU device, meaning that it cannot be executed on other type resources. That is, the GPU (portions of) jobs are statically fixed and

Table 1.1 Categorization of real-time multicore scheduling problems

Workload / Resource	Sequential	Parallel (multi-segment)	Parallel (DAG with pre-fixed parallelism)	Parallel (DAG with multi-version parallelism)
Homogeneous	[10] [11] [12] [13] [14] [15] [3] [4] [16]	[21] [22] [23] [5] [6] [24] [25] [26]	[5] [6] [28] [29] [7]	[30]
Heterogeneous (dedicated)	[17] [2]	[27]		
Heterogeneous (interchangeable)	[18] [19] [20]			This work

cannot be executed on multicore CPUs, which limits the freedom of resource scheduling.

The third group for the sequential workload model [18], [19], [20] considers the interchangeable heterogeneous resource model. Specifically, [19] and [20] present scheduling algorithms for heterogeneous resource model where jobs are allowed to use any type of resources. However, all the resource types considered are preemptive. In contrast, [18] considers preemptive CPUs and non-preemptive GPUs for distributing real-time data streams depending on their rates and deadlines.

The workload model also has been extended from the sequential workload model to parallel workload models. The first parallel workload model called a "multi-segment parallel workload model" assumes that each task is modeled as a sequence of segments and each segment consists of multiple parallelized threads. For this workload model, the first group [21], [22], [23], [5], [6], [24], [25], [26] addresses the

problem of scheduling such parallel tasks on homogeneous resources. Specifically, [21] presents a scheduling algorithm for a single-segment parallel workload model on identical multiprocessors. [22] and [23] present scheduling algorithms for the fork-join workload model where single-thread and multi-thread segments alternate, that is, a single-thread forks into multi-threads in the next segment and they join into a single thread in the next segment and so on. [5], [6], [24], [25] and [26] address a more general multi-segment parallel workload model without alternating fork and join points.

For the multi-segment parallel workload model, the second group [27] addresses the problem of scheduling multi-segment parallel tasks on multiple types of resources including both preemptive and non-preemptive ones. However, the resource model is limited in that jobs are dedicated to specific resource types. It is still open issue how to schedule multi-segment parallel tasks on interchangeable heterogeneous resources.

A more general workload model is a "DAG-based parallel workload model" where each task is modeled as a DAG and each vertex of the DAG has one sequential thread. For this workload model, [5], [6], [28], [29], [7] address the problem of scheduling such tasks on homogeneous resources.

In all of the aforementioned work, a task in the sequential workload model, a segment in the multi-segment parallel workload model, and a vertex in the DAG-based parallel workload model has a pre-fixed number of threads and hence it is modeled as a pre-fixed worst case execution time. Unlike this assumption, recent work [30] addresses a general DAG-based parallel workload model where each vertex has a freedom on the degree of parallelism. A vertex has a fixed amount of computation requirements, but the system can choose how many homogeneous resources it will use to execute the vertex. Thus, the execution time of the vertex is

modeled as its total computation requirement divided by the number of homogeneous resources used for executing it. However, such a simple execution time model for multi-version parallelism is not practical since such perfect parallelization is not possible due to parallelization overheads. Also, the resource model is limited to the homogeneous resource model.

In this paper, we aim at finding a solution for the most general workload model, i.e., DAG-based parallel workload model with multi-version parallelism and the most general resource model, i.e., interchangeable heterogeneous resource model with multiple CPU cores and multiple GPU devices. This work is positioned right-most and bottom-most in Table 1.1.

Chapter 3

Problem Description

In this chapter, we formally define the problem. The resource in the target system and the workload to be executed are depicted in Figure 3.1. We consider a system with m^{cpu} preemptive resources, i.e., CPU cores, and m^{gpu} non-preemptive resources, i.e., GPU devices as in the figure. Thus, the set $\mathcal{R}$ of all the resources is modeled as:

$$\mathcal{R} = \{r_h^{\mathrm{cpu}} | 1 \leq h \leq m^{\mathrm{cpu}}\} \cup \{r_h^{\mathrm{gpu}} | 1 \leq h \leq m^{\mathrm{gpu}}\}.$$

On top of this set of resources, the workload to be executed is modeled as a set of n independent sporadic hard real-time tasks as in Fig. 3.1:

$$\Gamma = \{\tau_i | 1 \leq i \leq n\}.$$

Each task τ_i is characterized by DAG G_i, minimum inter-release time T_i, and relative deadline D_i $(D_i \leq T_i)$ as follows:

$$\tau_i = (G_i, T_i, D_i).$$

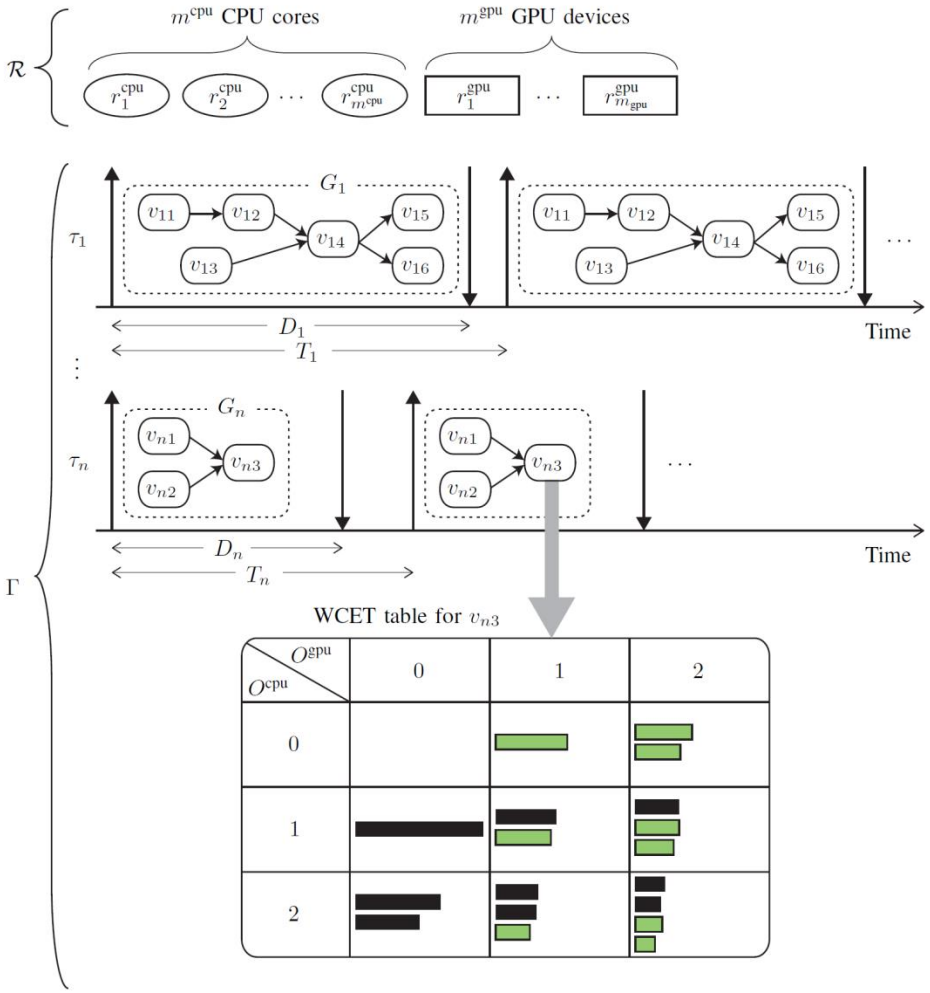


Figure 3.1 Multiple CPU/GPU resources $\mathcal{R}$ and DAG-based workload Γ with multiple parallelization options

τ_i is repeatedly released with the minimum inter-release time T_i. Once τ_i is released, it should execute a parallel program represented by the DAG G_i. Its execution must be completed within the relative deadline D_i.

The DAG G_i consists of a set V_i of vertexes and a set E_i of directed edges. k-th vertex of DAG G_i is denoted by v_{ik} where $1 \leq k \leq |V_i|$. A directed

edge $(v_{ik_1}, v_{ik_2}) \in E_i$ represents a precedence constraint meaning that v_{ik_2} can start its execution only after v_{ik_1} finishes the execution. Each vertex v_{ik} of DAG G_i has multiple parallelization options as depicted in the WCET table of Figure 3.1. An option is represented by $(O^{\text{cpu}}, O^{\text{gpu}})$ meaning that the vertex is parallelized into O^{cpu} threads for CPU cores and O^{gpu} threads for GPU devices.[1] Out of $O^{\text{cpu}} + O^{\text{gpu}}$ threads for executing v_{ik} with option $(O^{\text{cpu}}, O^{\text{gpu}})$, ℓ-th thread's worst case execution time (WCET) is denoted by $WCET_{ik}((O^{\text{cpu}}, O^{\text{gpu}}), \ell)$ and assumed to be given through offline program analysis [31] or measurements [32].[2]

Problem Definition: For the given resource $\mathcal{R}$ and workload Γ, our problem has two folds:

- *Parallelization*: choose parallelization option $(O^{\text{cpu}}, O^{\text{gpu}})$ for every vertex v_{ik} of DAG G_i for every task $\tau_i = (G_i, T_i, D_i)$ in $\Gamma = \{\tau_i | 1 \leq i \leq n\}$ and
- *Scheduling*: determine when the parallelized CPU and GPU threads should be executed on which CPU cores and GPU devices,

meeting all the deadlines and precedence constraints.

Our goal is to find the optimal or near-optimal solutions for the problem such that as many as possible tasks can be feasibly scheduled with the given CPU cores and GPU devices.

---

[1] This general model can cover practical scenarios where a CPU thread vertex launches the next vertex's CPU/GPU threads and their results are combined by the next CPU thread vertex and so on.

[2] Communication overheads between CPU and GPU are defined to be a part of the thread's WCET, as in [2]. Preemption and migration costs are assumed to be zero. In the future, we plan to extend our work for a more practical task model.

Chapter 4

Proposed Solution

To solve the problem defined in the previous chapter, in Section 4.1, we first tackle a simplified problem with only CPU cores and sporadic tasks modeled by simple DAGs. Then, Section 4.2 extends this algorithm for the heterogeneous resource model with both CPU cores and GPU devices. Finally, Section 4.3 further extends the algorithm for the general DAG-based task model.

4.1 Solution for CPU Cores and Simple DAG-based Tasks

This section proposes an optimal solution for a simplified problem. In the simplified problem, the resource set $\mathcal{R}$ has only m^{cpu} CPU cores and each task τi in the sporadic task set $\Gamma = \{\tau_i | 1 \leq i \leq n\}$ is modeled by a simple DAG, i.e., a directed sequence of $|V_i|$ vertexes. For this problem, we have to jointly address the two issues, (1) parallelization, i.e., determining a parallelization option for each vertex and (2)

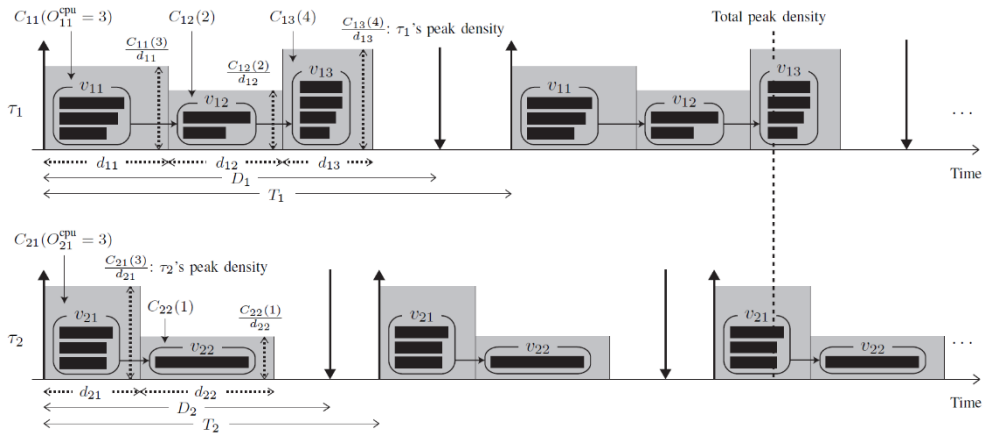


Figure 4.1 Simple DAG workload $\{\tau_1, \tau_2\}$ with fixed parallelization options and assigned local deadlines

scheduling, i.e., determining when the parallelized threads should be executed on which CPU core.

For the scheduling issue, we take the similar scheduling model as in [4]. Specifically, once the parallelization option O_{ik}^{cpu} is fixed for each vertex v_{ik}, we assign a "local deadline" d_{ik}, which roles as the same deadline for all the threads of v_{ik} as shown in Figure 4.1. All the threads of v_{ik} are assumed to be released at the same time when $v_{i(k-1)}$'s deadline $d_{i(k-1)}$ has been reached and have to be completed before the same deadline d_{ik}. These parallelized threads with assigned local deadlines are scheduled by an existing optimal multicore scheduling algorithm such as LLREF[10] and they can meet their local deadlines on m^{cpu} CPU cores if the sum of the densities of all active threads is not greater than m^{cpu} at all times. Note that a thread is defined active from its release time to its absolute deadline and its density is defined as its WCET divided by its relative deadline. Thus, if each vertex v_{ik} chooses a parallelization options O_{ik}^{cpu} and a deadline d_{ik} (See Figure 4.1), its

density is $\frac{C_{ik}(O_{ik}^{\text{cpu}})}{d_{ik}}$ where $C_{ik}(O_{ik}^{\text{cpu}})$ is the total execution time requirement, i.e.,

$$C_{ik}(O_{ik}^{\text{cpu}}) = \sum_{\ell=1}^{O_{ik}^{\text{cpu}}} WCET_{ik}(O_{ik}^{\text{cpu}}, \ell).$$

Since at most one vertex of τ_i is active at a time, τ_i's peak density is the largest one among all vertex densities, that is, $\max_{1 \le k \le |V_i|} \frac{C_{ik}(O_{ik}^{\text{cpu}})}{d_{ik}}$. Moreover, the sum of densities of all active threads becomes largest when each task's peak density meets all together as marked as "total peak density" in Figure 4.1.

Therefore, to minimize the total peak density to schedule as many as possible tasks with m^{cpu} CPU cores, the problem boils down to minimizing each task τ_i's peak density independently by optimally determining the parallelization option O_{ik}^{cpu} and the deadline d_{ik} for each vertex v_{ik} of τ_i. This optimization problem can be formulated as follows:

Optimization Problem 1.

Minimize
$O_{ik}^{\text{cpu}}, d_{ik}\ (1 \le k \le |V_i|)$ $\quad \max_{1 \le k \le |V_i|} \frac{C_{ik}(O_{ik}^{\text{cpu}})}{d_{ik}}$

Subject to $\quad C_{ik}^{\min}(O_{ik}^{\text{cpu}}) \le d_{ik}, \forall 1 \le k \le |V_i| \quad (1)$

$$\sum_{k=1}^{|V_i|} d_{ik} \le D_i \quad (2)$$

The first constraint Equation (1) says that each vertex v_{ik}'s deadline d_{ik} should be greater than or equal to the minimum time requirement $C_{ik}^{\min}(O_{ik}^{\text{cpu}})$

needed to complete it with the selected parallelization option O_{ik}^{cpu}. $C_{ik}^{\min}(O_{ik}^{\text{cpu}})$ is the WCET of the longest thread out of O_{ik}^{cpu} threads, i.e.,

$$C_{ik}^{\min}(O_{ik}^{\text{cpu}}) = \max_{1 \le \ell \le O_{ik}^{\text{cpu}}} WCET_{ik}(O_{ik}^{\text{cpu}}, \ell) \sum_{\ell=1}^{O_{ik}^{\text{cpu}}} WCET(O_{ik}^{\text{cpu}}, \ell).$$

The second constraint Equation (2) says that the sum of local deadlines of all the vertexes should be smaller than or equal to D_i to meet the task τ_i's original deadline requirement D_i.

To solve this problem, our algorithm consists of the following five steps:

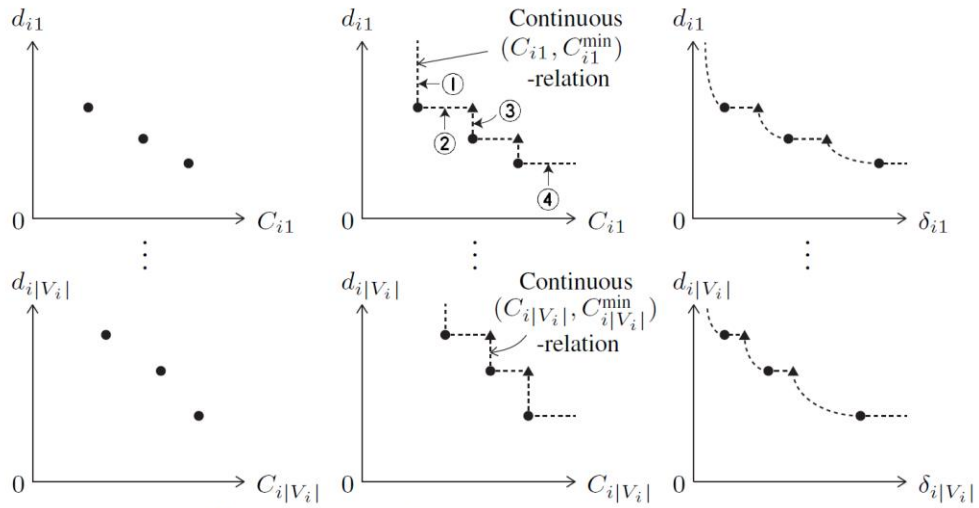
Algorithm for CPU cores and Simple DAG-based Task

Input: Simple DAG G_i, the parallelization options O_{ik}^{cpu}s and their corresponding $(C_{ik}(O_{ik}^{\text{cpu}}), C_{ik}^{\min}(O_{ik}^{\text{cpu}}))$ pairs for each vertex v_{ik} in G_i, and deadline D_i of task τ_i.

Output: Minimized peak density δ_i for task τ_i and optimal parallelization option O_{ik}^{cpu} and optimal deadline d_{ik} for each vertex v_{ik} of τ_i.

- **Step 1**: From $(C_{ik}(O_{ik}^{\text{cpu}}), C_{ik}^{\min}(O_{ik}^{\text{cpu}}))$ pairs of each vertex v_{ik} in Figure 4.2 (a), we form a continuous relation between C_{ik} and d_{ik} as in Figure 4.2 (b).

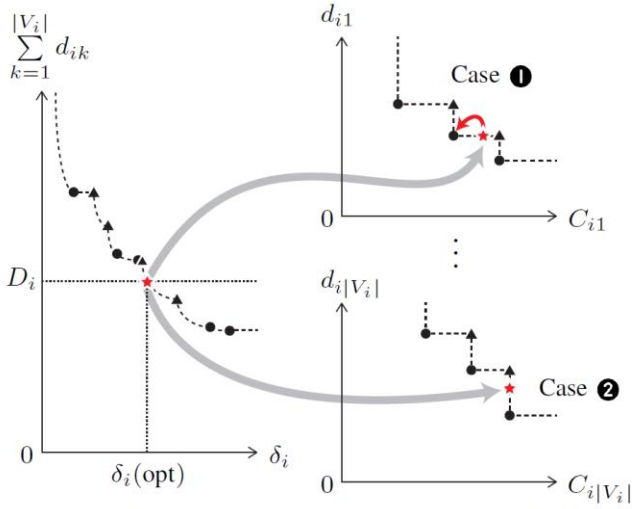
- **Step 2**: We transform the (C_{ik}, d_{ik})-relation of each vertex v_{ik} in Figure 4.2 (b) to the relation between vertex's density $\delta_{ik} = \frac{C_{ik}}{d_{ik}}$ and deadline d_{ik} as in Figure 4.2 (c).



(a) $(C_{ik}(O_{ik}^{\text{cpu}}),$ $C_{ik}^{\min}(O_{ik}^{cpu}))$ points

(b) Continuous (C_{ik}, d_{ik})-relation

(c) Continuous (δ_{ik}, d_{ik})-relation



(d) Finding optimal peak density $\delta_i(\text{opt})$

(e) Optimal solution $(C_{ik}(\text{opt}), d_{ik}(\text{opt}))$

Figure 4.2 Illustration of the Algorithm for CPU cores and Simple DAG-based Task

- **Step 3**: We merge the (δ_{ik}, d_{ik})-relation of every vertex v_{ik} in Figure 4.2 (c) into the relation between task's density δ_i and the deadline sum $\sum_{k=1}^{|V_i|} d_{ik}$ as in Figure 4.2 (d).

- **Step 4**: From the $(\delta_i, \sum_{k=1}^{|V_i|} d_{ik})$-relation in Figure 4.2 (d), we find $\delta_i(\text{opt})$ where $\sum_{k=1}^{|V_i|} d_{ik}$ becomes equal to D_i. That $\delta_i(\text{opt})$ is the optimal objective value of Optimization Problem 1.

- Step 5: By reverse mapping the found $\delta_i(\text{opt})$ to the (C_{ik}, d_{ik})-relations as in Figure 4.2 (e), we find the optimal O_{ik}^{cpu} and d_{ik} for each vertex v_{ik} of τ_i.

We now give the details of each step and explain why the algorithm indeed finds the optimal solution.

Step 1: Generally, for a vertex v_{ik}, as increasing the number O_{ik}^{cpu} of threads for executing it, the minimum required execution time $C_{ik}^{\min}(O_{ik}^{\text{cpu}})$ decreases but the total execution time requirement $C_{ik}(O_{ik}^{\text{cpu}})$ increases due to parallelization overhead. In Figure 4.2 (a), $(C_{ik}(O_{ik}^{\text{cpu}}), C_{ik}^{\min}(O_{ik}^{\text{cpu}}))$ pairs corresponding to the discrete parallelization options O_{ik}^{cpu}s are depicted by black dots in the 2-dimensional space of C_{ik} and d_{ik}. These discrete $(C_{ik}(O_{ik}^{\text{cpu}}), C_{ik}^{\min}(O_{ik}^{\text{cpu}}))$ pairs for each vertex v_{ik} are extended to an imaginary continuous $(C_{ik}, C_{ik}^{\min})$-relation as follows (see Figure 4.2 (b)): (1) We draw a vertical line segment from ∞ down to the first discrete point of $(C_{ik}(O_{ik}^{\text{cpu}}), C_{ik}^{\min}(O_{ik}^{\text{cpu}}))$ (See part ① in Figure 4.2 (b)). (2) From there, we draw a horizontal line segment until C_{ik} becomes the same as that of the

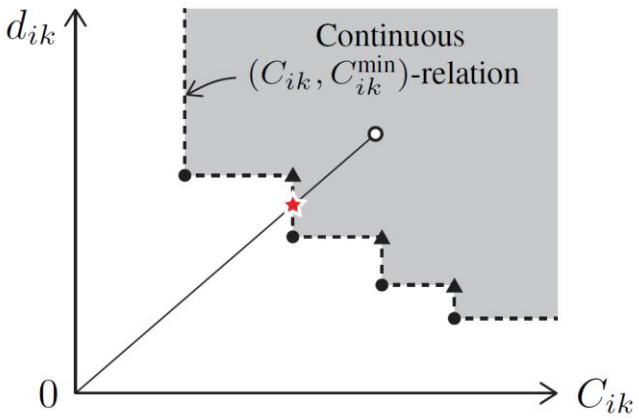


Figure 4.3 Illustration of a non-optimal solution point (white dot) on a (C_{ik}, d_{ik})-plane

second discrete point of $(C_{ik}(O_{ik}^{\text{cpu}}), C_{ik}^{\min}(O_{ik}^{\text{cpu}}))$ (See part ② in Figure 4.2 (b)). The end of such a horizontal line segment is called a pivot point and marked as a black triangle. (3) From there, we draw a vertical line down to the second discrete point of $(C_{ik}(O_{ik}^{\text{cpu}}), C_{ik}^{\min}(O_{ik}^{\text{cpu}}))$ (See part ③ in Figure 4.2 (b)). (4) We continue this until the last discrete point of $(C_{ik}(O_{ik}^{\text{cpu}}), C_{ik}^{\min}(O_{ik}^{\text{cpu}}))$. (5) After that, we draw a horizontal line toward ∞ (See part ④ in Figure 4.2 (b)).

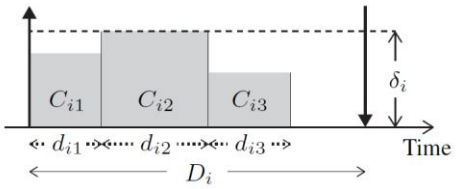
With such formed continuous $(C_{ik}, C_{ik}^{\min})$-relation, we can form an "imaginary continuous solution space" where not only d_{ik} but also C_{ik} are continuous. Lemma 1 says that the optimal solution $C_{ik}(\text{opt})$ and $d_{ik}(\text{opt})$ in the imaginary continuous solution space, which we call "continuous optimal solution", is located on the $(C_{ik}, C_{ik}^{\min})$-relation of each vertex v_{ik}.

Lemma 1. Any continuous optimal solution $C_{ik}(\text{opt})$, $d_{ik}(\text{opt})$ $(1 \leq k \leq |V_i|)$ for Optimization Problem 1 is located on $(C_{ik}, C_{ik}^{\min})$-relation $(1 \leq k \leq |V_i|)$.

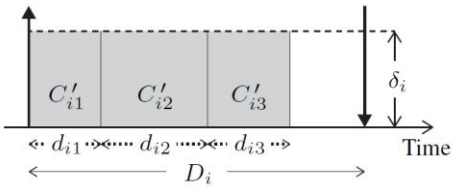
Proof. For any vertex v_{ik}, consider a 2-dimensional space of C_{ik} and d_{ik} as in Figure 4.3. Since d_{ik} should be greater than or equal to $C_{ik}^{\min}$, any valid solution (C_{ik}, d_{ik}) should be located above or on the $(C_{ik}, C_{ik}^{\min})$-relation, that is, the shaded area in the figure. Consider a solution (C_{ik}, d_{ik}) located above the $(C_{ik}, C_{ik}^{\min})$-relation, for example, the white dot in the figure. We show that this solution is not optimal as follows: Note that when we draw a line from $(0, 0)$ to (C_{ik}, d_{ik}), its slope $\frac{d_{ik}}{C_{ik}}$ is the inverse of the vertex density $\frac{C_{ik}}{d_{ik}}$. Thus, following this line, we can find a point (C_{ik}', d_{ik}') on the $(C_{ik}, C_{ik}^{\min})$-relation, i.e., the star mark in the figure. This point has the same slope and hence the same vertex density, i.e., $\frac{C_{ik}'}{d_{ik}'} = \frac{C_{ik}}{d_{ik}}$, but $C_{ik}' < C_{ik}$ and $d_{ik}' < d_{ik}$. By replacing (C_{ik}, d_{ik}) with (C_{ik}', d_{ik}'), the original solution can be transformed into another solution with the same vertex densities. However, this transformed solution has a deadline gap of $d_{ik} - d_{ik}'$. By distributing this deadline gap to all the vertexes, we can reduce the densities of all the vertexes and hence the peak density. This means the original solution cannot be optimal. Thus, the lemma follows.

Due to Lemma 1, we can consider $(C_{ik}, C_{ik}^{\min})$-relation as the set of (C_{ik}, d_{ik}) candidates for the continuous optimal solution. Thus, we treat $(C_{ik}, C_{ik}^{\min})$-relation as (C_{ik}, d_{ik})-relation for the continuous optimal solution.

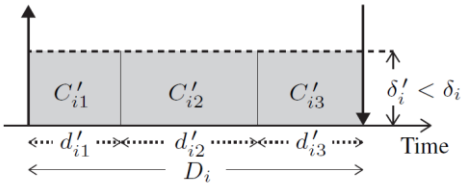
Step 2: For every point on (C_{ik}, d_{ik})-relation in Figure 4.2 (b), its density δ_{ik} is simply given as $\frac{C_{ik}}{d_{ik}}$, i.e., the inverse of its slope. Thus, by transforming every (C_{ik}, d_{ik}) point to its corresponding (δ_{ik}, d_{ik}) point, we can construct (δ_{ik}, d_{ik})-relation as in Figure 4.2 (c).



(a) A solution with non-uniform vertex densities



(b) A solution with uniform vertex densities



(c) A solution with uniform vertex densities and tight deadline

Figure 4.4 Illustration of solutions for Optimization Problem 1

Step 3: In this step, we vertically add d_{ik} values on the (δ_{ik}, d_{ik})-relations of all vertexes in Figure 4.2 (c) for each density value δ_i, resulting in $(\delta_i, \sum_{k=1}^{|V_i|} d_{ik})$-relation as in Figure 4.2 (d). Lemma 2 allows this addition without loss of optimality by saying that an optimal solution can be found from the cases where all the vertex densities are the same, i.e., $\delta_{i1} = \delta_{i2} = \cdots = \delta_{i|V_i|} = \delta_i$, as in Figure 4.4 (b).

Lemma 2. For any feasible solution C_{ik}, d_{ik} $(1 \le k \le |V_i|)$ with the peak density $\delta_i = \max_{1 \le k \le |V_i|} \frac{C_{ik}}{d_{ik}}$, there exists another solution C_{ik}', d_{ik} $(1 \le k \le |V_i|)$ where all the vertex densities are uniform as δ_i, i.e.,

$$\frac{C_{i1}'}{d_{i1}} = \frac{C_{i2}'}{d_{i2}} = \cdots = \frac{C_{i|V_i|}'}{d_{i|V_i|}} = \delta_i.$$

See Figure 4.4 (b).

Proof. If a feasible solution C_{ik}, d_{ik} $(1 \le k \le |V_i|)$ with the peak density δ_i has non-uniform vertex densities as in Figure 4.4 (a), it can always be modified into another feasible solution C_{ik}', d_{ik} $(1 \le k \le |V_i|)$ with uniform vertex densities δ_i as in Figure 4.4 (b) as follows. For every vertex v_{ik} with a smaller density than the peak density δ_i, we increase its total execution time requirement C_{ik} to $C_{ik}' = \delta_i \cdot d_{ik}$ such that its density becomes the same as δ_i. This increase of C_{ik} is valid since (1) C_{ik} is a continuous variable until ∞ and (2) increasing C_{ik} results in the same or a smaller $C_{ik}^{\min}$ and hence the constraint of $C_{ik}^{\min} \le d_{ik}$ in Equation (1) is still met. Thus, the lemma follows.

Step 4: We find the point where $(\delta_i, \sum_{k=1}^{|V_i|} d_{ik})$-relation and the horizontal line at D_i meet, as the star mark in Figure 4.2 (d), that gives the continuous optimal solution with the minimum uniform density $\delta_i(\text{opt})$ by Lemma 2 and the following lemma.

Lemma 3. Any continuous optimal solution $C_{ik}(\text{opt})$, $d_{ik}(\text{opt})$ $(1 \le k \le |V_i|)$ for Optimization Problem 1 satisfies

$$\sum_{k=1}^{|V_i|} d_{ik}(\text{opt}) = D_i.$$

See Figure 4.4 (c).

Proof. For any feasible solution C_{ik}', d_{ik} $(1 \le k \le |V_i|)$ with $\sum_{k=1}^{|V_i|} d_{ik} < D_i$ as in Figure 4.4 (b), we can construct another feasible solution C_{ik}', d_{ik}' $(1 \le k \le |V_i|)$ with a smaller peak density as in Figure 4.4 (c) by distributing the deadline gap $D_i - \sum_{k=1}^{|V_i|} d_{ik}$ to all the vertexes to lengthen their deadlines, i.e., to decrease their densities. Thus, the lemma follows.

If the horizontal line at D_i is completely below the $(\delta_i, \sum_{k=1}^{|V_i|} d_{ik})$-relation and hence there is no crossover point, it means that there is no feasible solution, that is, such small D_i cannot be satisfied even with the maximum parallelization options for all vertexes.

Step 5: By reverse mapping $\delta_i(\text{opt})$ to C_{ik} and d_{ik} using (C_{ik}, d_{ik})-relations, we can eventually find the continuous optimal solution $C_{ik}(\text{opt})$, $d_{ik}(\text{opt})$ $(1 \le k \le |V_i|)$. If the found $C_{ik}(\text{opt})$, $d_{ik}(\text{opt})$ for a vertex v_{ik} is on the vertical line segment as in case ❷ of Figure 4.2 (e), its corresponding O_{ik}^{cpu} and $d_{ik}(\text{opt})$ is the valid optimal solution for v_{ik}. Otherwise, that is, case ❶ of Figure 4.2 (e), we use O_{ik}^{cpu} of the leftmost point of the horizontal line segment and the same deadline $d_{ik}(\text{opt})$ as the valid solution for v_{ik}. Theorem 1 says that such found solution O_{ik}^{cpu} and d_{ik} $(1 \le k \le |V_i|)$ is the optimal solution for Optimization Problem 1.

Theorem 1. The above algorithm finds the optimal solution O_{ik}^{cpu}, d_{ik} $(1 \le k \le |V_i|)$ for Optimization Problem 1.

Proof. $C_{ik}(\text{opt})$, $d_{ik}(\text{opt})$ $(1 \le k \le |V_i|)$ found by the above algorithm is the continuous optimal solution with the minimum possible uniform density $\delta_i(\text{opt})$ of τ_i due to Lemma 1, Lemma 2, and Lemma 3. Its mapping to the O_{ik}^{cpu} and d_{ik} in Step 5 results in a valid solution with the same peak density $\delta_i(\text{opt})$ since the vertex densities of case ❶ in Figure 4.2 (e) decrease but those of case ❷ in Figure 4.2 (e) remain the same.

The above steps can be practically carried out using only the discrete points, i.e., black dots and black triangles in Figure 4.2, as follows. For Steps 1 and 2, the

discrete points in Figure 4.2 (b) and Figure 4.2 (c) can be formed with time complexity of $O(n_{\mathrm{O}}|V_i|)$ where n_{O} is the number of parallelization options for each vertex. For Step 3, the discrete points in Figure 4.2 (d) can be formed with time complexity of $O(n_{\mathrm{O}}|V_i|\log|V_i|)$. Only with these discrete points, for Step4, we can find in between which two discrete points the continuous optimal solution is located, with the time complexity of $O(n_{\mathrm{O}}|V_i|)$. For Step 5, these two discrete points can be reverse-mapped to the discrete points of (C_{ik}, d_{ik})-relation as in Figure 4.2 (e), and thus identify in between which two discrete points of (C_{ik}, d_{ik}) of each vertex v_{ik} the continuous optimal point is located, and whether it is case ❶ or case ❷. For every vertex v_{ik} of case ❶, we clearly know its deadline $d_{ik}(\mathrm{opt})$ since the d_{ik} is constant on the horizontal line segment. Thus, $D_i - \sum_{v_{ik} \text{ of } ❶} d_{ik}(\mathrm{opt})$ is the remaining deadline that can be given to all the vertexes of case ❷. For every vertex v_{ik} of case ❷, we clearly know its $C_{ik}(\mathrm{opt})$ since C_{ik} is constant on the vertical line segment. Also, the densities of these vertexes are uniform as $\delta_i(\mathrm{opt})$. Thus, for these vertexes, their average density $\frac{\sum_{v_{ik} \text{ of } ❷} C_{ik}(\mathrm{opt})}{D_i - \sum_{v_{ik} \text{ of } ❶} d_{ik}(\mathrm{opt})}$ now gives the exact value of $\delta_i(\mathrm{opt})$. From $\delta_i(\mathrm{opt})$, we can now determine d_{ik} for each vertex of case ❷ as $\frac{C_{ik}(\mathrm{opt})}{\delta_i(\mathrm{opt})}$. This step takes $O(n_{\mathrm{O}}|V_i|)$. Overall, we can find the optimal solution with time complexity of $O(n_{\mathrm{O}}|V_i|\log|V_i|)$.

If such optimal solutions are determined for all the tasks $\tau_1, \tau_2, \ldots, \tau_n$, their optimal densities $\delta_i(\mathrm{opt})$s are summed up. If the sum, i.e., $\sum_{i=1}^{n} \delta_i(\mathrm{opt})$, is smaller than or equal to the number of CPU cores, i.e., m^{cpu}, then all the tasks can be feasibly scheduled using an optimal multicore scheduling algorithm such as LLREF.

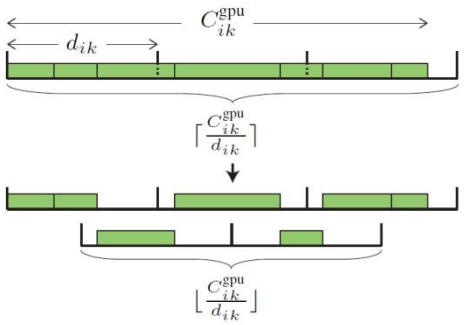
Theorem 2. For the simplified problem, the proposed parallelization and scheduling algorithm is optimal (amongst the algorithms adding intermediate deadlines) when the schedulability test is based on the density.

Proof. The proposed algorithm minimizes each task's peak density and hence the total peak density of the given set of sporadic tasks. Thus, if the proposed algorithm cannot successfully schedule the given task set, that is, the minimum of possible total peak density is greater than the number m^{cpu} CPU cores, then the given task set is not schedulable by any other algorithm that assigns intermediate deadlines and tests the schedulability based on the total peak density.

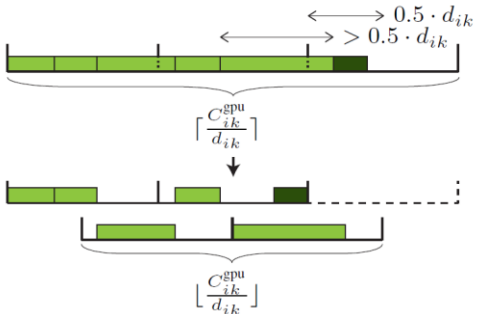
4.2 Extension for GPU Devices

This section extends the optimal algorithm for the simplified problem to a near optimal algorithm for heterogeneous resources with both preemptive CPU cores and non-preemptive GPU devices. For this, we need GPU's total density bound like CPU's total density bound of m^{cpu}. Note that once a GPU thread enters into a GPU device, it is executed by the GPU-internal scheduler for which we have no control until the result comes out.[3] Thus, a GPU device is usually modeled as a non-preemptive resource. In order to derive a total density bound for such non-preemptive m^{gpu} GPU devices that works for any GPU-internal scheduling, we consider a partitioned scheduling approach where GPU threads are partitioned and a GPU device is dedicated for each partition.

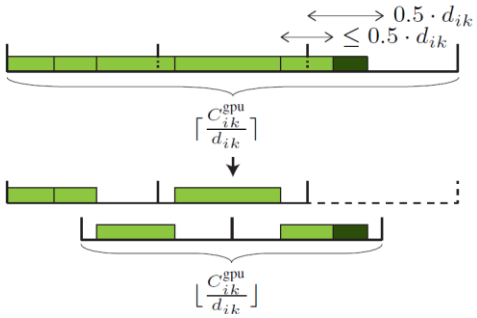
[3] In this work, the term *GPU thread* refers to a sequence of GPU operations (host-to-device memory copy, kernel execution, and device-to-host memory copy).



(a) When the last group is more than 50% full



(b) When the last group $\leq 50\%$ full and the last split thread $> 0.5 \cdot d_{ik}$



(c) When the last group $\leq 50\%$ full and the last split thread $\leq 0.5 \cdot d_{ik}$

Figure 4.5 Partitioning GPU threads of vertex v_{ik} into $\left\lfloor 2 \cdot \frac{C_{ik}^{\text{gpu}}}{d_{ik}} \right\rfloor$ bins of size d_{ik}

Let us explain such GPU thread partitioning algorithm assuming a fixed parallelization option $(O_{ik}^{\text{cpu}}, O_{ik}^{\text{gpu}})$ and a fixed local deadline d_{ik} for each vertex v_{ik} of task τ_i. From now on, we use $C_{ik}^{\text{cpu}}(O_{ik}^{\text{cpu}}, O_{ik}^{\text{gpu}})$ to denote the total CPU execution time requirement, i.e., the sum of WCETs of the O_{ik}^{cpu} CPU threads and $C_{ik}^{\text{gpu}}(O_{ik}^{\text{cpu}}, O_{ik}^{\text{gpu}})$ to denote the total GPU execution time requirement, i.e., the sum of WCETs of the O_{ik}^{gpu} GPU threads. Also, we use $C_{ik}^{\text{min}}(O_{ik}^{\text{cpu}}, O_{ik}^{\text{gpu}})$ to denote the minimum time requirement, that is, the WCET of the longest threads out of all the

$O_{ik}^{\text{cpu}} + O_{ik}^{\text{gpu}}$ threads. For the notational simplicity, if no ambiguity, we use C_{ik}^{cpu}, C_{ik}^{gpu}, and $C_{ik}^{\min}$ omitting the parallelization option.

Figure 4.5 depicts our GPU thread partitioning for a vertex v_{ik}. First, we put all the GPU threads of v_{ik} in a row, whose length becomes C_{ik}^{gpu}, and divide this row into groups of size d_{ik}, as in the top part of Figure 4.5 (a). This results in $\left\lceil \frac{C_{ik}^{\text{gpu}}}{d_{ik}} \right\rceil$ groups with at most $\left\lfloor \frac{C_{ik}^{\text{gpu}}}{d_{ik}} \right\rfloor$ threads split into two groups. Since GPU threads are non-preemptive, their execution cannot be split. Thus, we add $\left\lfloor \frac{C_{ik}^{\text{gpu}}}{d_{ik}} \right\rfloor$ extra groups and move the split threads to the added groups as in the bottom part of Figure 4.5 (a). As a result, the GPU threads of v_{ik} are partitioned into $\left\lceil \frac{C_{ik}^{\text{gpu}}}{d_{ik}} \right\rceil + \left\lfloor \frac{C_{ik}^{\text{gpu}}}{d_{ik}} \right\rfloor$ groups. For each partition, we dedicate a GPU device. Then, the threads of each partition can be scheduled within d_{ik} by any GPU-internal work-conserving scheduling.

On top of this base partitioning, if the rightmost group is less than 50% full as in the top part of Figure 4.5 (b) and Figure 4.5 (c), we can pack the non-split threads of the rightmost group, i.e., dark boxes in Figure 4.5 (b) and Figure 4.5 (c), either into the second rightmost group as in the case of Figure 4.5 (b) or into the rightmost extra group as in the case of Figure 4.5 (c) and remove the rightmost group. Thus, in this case, the GPU threads are partitioned into $\left\lceil \frac{C_{ik}^{\text{gpu}}}{d_{ik}} \right\rceil + \left\lfloor \frac{C_{ik}^{\text{gpu}}}{d_{ik}} \right\rfloor - 1$ groups and hence can be scheduled within d_{ik} with that number of GPU devices. Note that this packing is possible only when $\frac{C_{ik}^{\text{gpu}}}{d_{ik}} > 1$.

In both cases of non-packing (Figure 4.5 (a)) and packing (Figure 4.5 (b) and Figure 4.5 (c)), we assume $\frac{C_{ik}^{\text{gpu}}}{d_{ik}} \geq 0.5$. Thus, if $\frac{C_{ik}^{\text{gpu}}}{d_{ik}} < 0.5$ for a vertex, we

conservatively treat its GPU density as 0.5 in the GPU schedulability check. Thus, vertex v_{ik}'s GPU density denoted by δ_{ik}^{gpu} is defined as $\max(\frac{C_{ik}^{\text{gpu}}}{d_{ik}}, 0.5)$.

With this partitioned GPU scheduling, the following theorem says that GPU's total density bound is $\frac{m^{\text{gpu}}}{2}$.

Theorem 3. All GPU threads of simple DAG-based tasks with the total GPU peak density δ^{gpu} not greater than $\frac{m^{\text{gpu}}}{2}$ can meet their local deadlines using the aforementioned partitioned scheduling on m^{gpu} GPU devices.

Proof. In the non-packing case (Figure 4.5 (a)), all the GPU threads of v_{ik} can be scheduled within d_{ik} with $\left\lceil \frac{C_{ik}^{\text{gpu}}}{d_{ik}} \right\rceil + \left\lfloor \frac{C_{ik}^{\text{gpu}}}{d_{ik}} \right\rfloor$ GPU devices. In this case, since $\frac{C_{ik}^{\text{gpu}}}{d_{ik}}$'s fractional part is greater than or equal to 0.5, $\left\lceil \frac{C_{ik}^{\text{gpu}}}{d_{ik}} \right\rceil + \left\lfloor \frac{C_{ik}^{\text{gpu}}}{d_{ik}} \right\rfloor$ is less than or equal to $\left\lfloor 2 \cdot \frac{C_{ik}^{\text{gpu}}}{d_{ik}} \right\rfloor$. Also, in the packing case (Figure 4.5 (b) and Figure 4.5 (c)), all the GPU threads of v_{ik} can be scheduled within d_{ik} with $\left\lceil \frac{C_{ik}^{\text{gpu}}}{d_{ik}} \right\rceil + \left\lfloor \frac{C_{ik}^{\text{gpu}}}{d_{ik}} \right\rfloor - 1$ GPU devices. In this case, $\left\lceil \frac{C_{ik}^{\text{gpu}}}{d_{ik}} \right\rceil + \left\lfloor \frac{C_{ik}^{\text{gpu}}}{d_{ik}} \right\rfloor - 1$ is less than or equal to $\left\lfloor 2 \cdot \frac{C_{ik}^{\text{gpu}}}{d_{ik}} \right\rfloor$. Combining these two cases, any vertex v_{ik} with density δ_{ik}^{gpu} can be scheduled within d_{ik} with $\lfloor 2 \cdot \delta_{ik}^{\text{gpu}} \rfloor$ GPU devices. For a task τ_i, since no two vertexes are simultaneously active, all the GPU threads of τ_i can be scheduled meeting their deadlines with $\max_{1 \le k \le |V_i|} \lfloor 2 \cdot \delta_{ik}^{\text{gpu}} \rfloor = \lfloor 2 \cdot \delta_i^{\text{gpu}} \rfloor$ GPU devices where δ_i^{gpu} is τ_i's GPU peak density, i.e., $\max_{1 \le k \le |V_i|} \delta_{ik}^{\text{gpu}}$. Therefore, all GPU threads of a task set with the total GPU

peak density $\delta^{\text{gpu}} = \sum_{i=1}^{n} \delta_i^{\text{gpu}}$ can be successfully scheduled on $\sum_{i=1}^{n} \lfloor 2 \cdot \delta_{ik}^{\text{gpu}} \rfloor \leq 2 \cdot \delta^{\text{gpu}}$ GPU devices. Thus, the theorem follows.

In order to schedule as many as possible tasks with the CPU's total density bound m^{cpu} and GPU's total density bound $\frac{m^{\text{gpu}}}{2}$, for each task τ_i, we aim at minimizing a greater density out of CPU peak density δ_i^{cpu} and GPU peak density δ_i^{gpu} normalized by their respective bounds. More specifically, we minimize $\max\{\frac{\delta_i^{\text{cpu}}}{m^{\text{cpu}}}, \frac{\delta_i^{\text{gpu}}}{m^{\text{gpu}}/2}\}$ by optimally determining a CPU/GPU parallelization option $(O_{ik}^{\text{cpu}}, O_{ik}^{\text{gpu}})$ and a local deadline d_{ik} for each vertex v_{ik} $(1 \leq k \leq |V_i|)$. Thus, the problem is formulated as follows:

Optimization Problem 2.

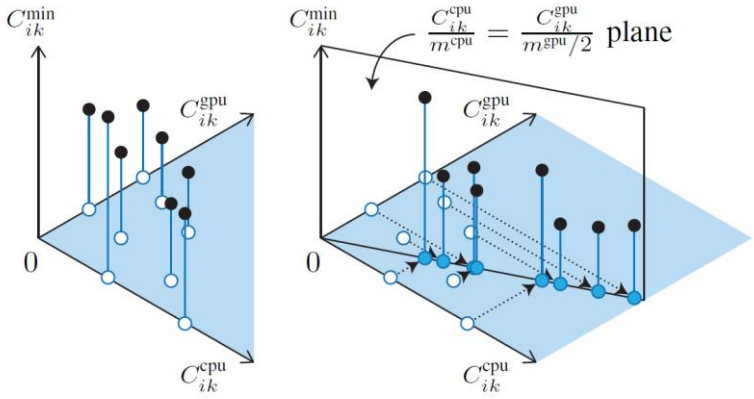
$$\text{Minimize}_{O_{ik}^{\text{cpu}}, O_{ik}^{\text{gpu}}, d_{ik}\ (1 \leq k \leq |V_i|)} \quad \max\{\frac{\delta_i^{\text{cpu}}}{m^{\text{cpu}}}, \frac{\delta_i^{\text{gpu}}}{m^{\text{gpu}}/2}\}$$

$$\text{Subject to} \quad C_{ik}^{\min}\left(O_{ik}^{\text{cpu}}, O_{ik}^{\text{gpu}}\right) \leq d_{ik}, \forall 1 \leq k \leq |V_i|$$

$$\sum_{k=1}^{|V_i|} d_{ik} \leq D_i$$

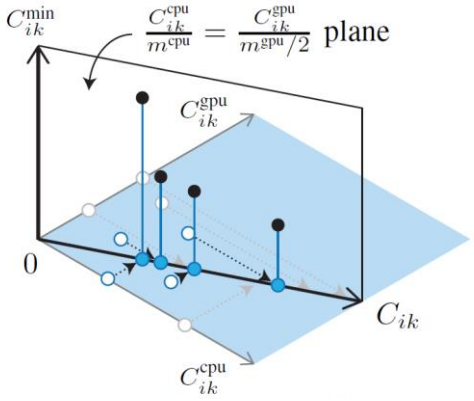
This optimization problem can be reduced to Optimization Problem 1 via the following procedure for each vertex v_{ik}.

- Black dots in Figure 4.6 (a) denote the $(C_{ik}^{\text{cpu}}(O_{ik}^{\text{cpu}}, O_{ik}^{\text{gpu}}), C_{ik}^{\text{gpu}}(O_{ik}^{\text{cpu}}, O_{ik}^{\text{gpu}}), C_{ik}^{\min}(O_{ik}^{\text{cpu}}, O_{ik}^{\text{gpu}}))$ points for the actual discrete parallelization options $(O_{ik}^{\text{cpu}}, O_{ik}^{\text{gpu}})$.



(a) Actual $(C_{ik}^{\text{cpu}}, C_{ik}^{\text{gpu}}, C_{ik}^{\min})$

(b) Mapping actual points onto $\frac{C_{ik}^{\text{cpu}}}{m^{\text{cpu}}} = \frac{C_{ik}^{\text{gpu}}}{m^{\text{gpu}}/2}$ plane



(c) Decreasing $(C_{ik}, C_{ik}^{\min})$ points

Figure 4.6 Reduction procedure of Optimization Problem 2 to Optimization Problem 1 for vertex v_{ik}

- We map each $(C_{ik}^{\text{cpu}}, C_{ik}^{\text{gpu}}, C_{ik}^{\min})$ point onto the $\frac{C_{ik}^{\text{cpu}}}{m^{\text{cpu}}} = \frac{C_{ik}^{\text{gpu}}}{m^{\text{gpu}}/2}$ plane as in Figure 4.6 (b). That is, on the $(C_{ik}^{\text{cpu}}, C_{ik}^{\text{gpu}})$-plane, the points below the $\frac{C_{ik}^{\text{cpu}}}{m^{\text{cpu}}} = \frac{C_{ik}^{\text{gpu}}}{m^{\text{gpu}}/2}$ line move upward, and those over the line move rightward,

until they reach the line while keeping the same $C_{ik}^{\min}$ values.

- Out of the points mapped onto the $\frac{C_{ik}^{\text{cpu}}}{m^{\text{cpu}}} = \frac{C_{ik}^{\text{gpu}}}{m^{\text{gpu}}/2}$ plane, we eliminate the non-decreasing points and obtain decreasing $(C_{ik}, C_{ik}^{\min})$ points as in Figure 4.6 (c).

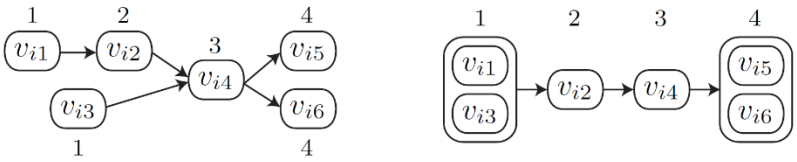
With such obtained $(C_{ik}, C_{ik}^{\min})$ points for each vertex v_{ik}, we apply the same algorithm in Figure 4.2 to find the valid optimal solution (C_{ik}, d_{ik}). By reverse mapping the found C_{ik} point to its corresponding $(C_{ik}^{\text{cpu}}(O_{ik}^{\text{cpu}}, O_{ik}^{\text{gpu}}), C_{ik}^{\text{gpu}}(O_{ik}^{\text{cpu}}, O_{ik}^{\text{gpu}}))$ point using Figure 4.6 (c), we can determine the CPU/GPU parallelization option $(O_{ik}^{\text{cpu}}, O_{ik}^{\text{gpu}})$.

By applying this optimization for each task, we can accept tasks until either CPU's total density bound m^{cpu} or GPU's total density bound $\frac{m^{\text{gpu}}}{2}$ is reached. After that, if there still remains room for the CPU's density, we accept tasks using CPU only parallelization options as explained in Section 4.1.

4.3 Extension for General DAG-based Tasks

We address the problem of general DAG-based tasks by transforming each task's general DAG into a simple DAG as in Figure 4.7 and then applying the algorithm in the previous sections. Specifically, for each task τ_i, we transform its DAG into a simple DAG as follows.

- We calculate the depth of each vertex as in Figure 4.7 (a).



(a) Calculating vertex depths of general DAG

(b) Forming simple DAG based on vertex depths

Figure 4.7 Transforming general DAG to simple DAG for task τ_i

- The vertexes of the same depth are combined into a single vertex as the first and the fourth vertexes in Figure 4.7 (b).
- We connect the resulting vertexes with a directed edge from the vertex of each depth to the next as in Figure 4.7 (b).

Such simplified DAG respects all of the original DAG's precedence relations. Also, for a new vertex v_{ik}' that consists of two or more vertexes from the original DAG, a combination of the parallelization options of the original vertexes, e.g., $((O_{i1}^{\text{cpu}}, O_{i1}^{\text{gpu}}), (O_{i3}^{\text{cpu}}, O_{i3}^{\text{gpu}}))$ for the first vertex in Figure 4.7 (b), becomes a parallelization option determining a point of $(C_{ik}^{\text{cpu}}, C_{ik}^{\text{gpu}}, C_{ik}^{\text{min}})$, e.g., a black dot in Figure 4.6 (a). Thus, we can apply the same algorithm as in Figure 4.6 and Figure 4.2 to find the solution.

Even though this simple DAG transformation is not optimal, it performs close to the optimal transformation as will be shown in Chapter 5, since we enjoy the freedom of both total computation requirement and local deadline.

Chapter 5

Experiments

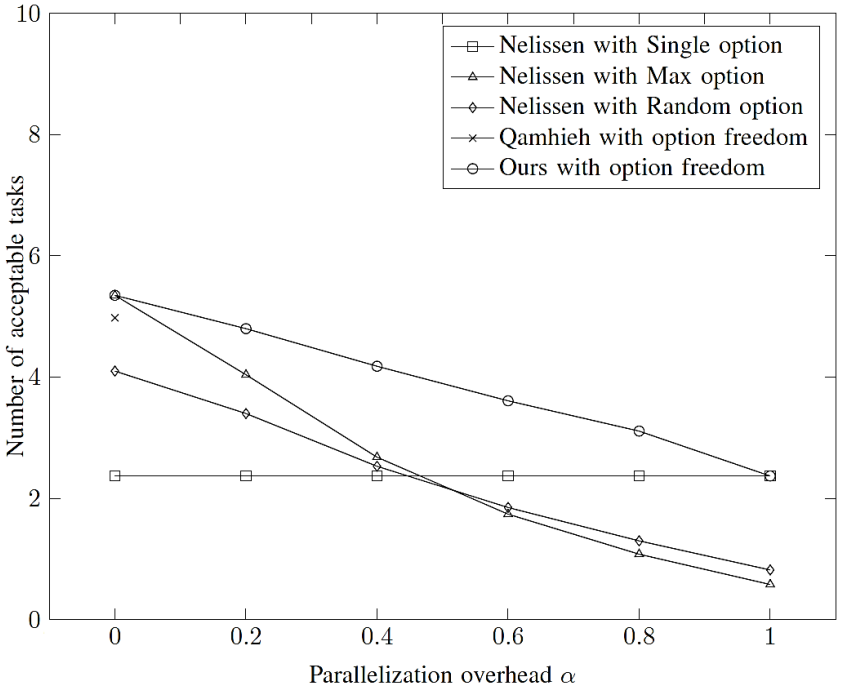
In order to study how much more real-time tasks are schedulable by enjoying the freedom of multiple parallelization options, this section presents our simulation results. As a hardware platform, we consider a system like ASUS ESC4000 G2 [33] equipped with 8 CPU cores and 7 GPU devices.

In our first experiment, we consider the problem of scheduling simple DAG-based tasks on 8 CPU cores. For this we randomly generate a list of tasks. Each task τ_i's DAG is formed as a directed sequence of $|V_i|$ vertexes where $|V_i|$ is randomly generated from 4 to 10. For each vertex v_{ik}, we generate a random number E from the range of (100 ms, 400 ms) and use it as v_{ik}'s WCET for the single CPU thread version. Then, its multiple CPU thread versions up to $O_{ik}^{\text{cpu}} = 4$ are made such that each thread for O_{ik}^{cpu} thread version has the WCET of $E/O_{ik}^{\text{cpu}} + \alpha(E - E/O_{ik}^{\text{cpu}})$ on average, where α models the parallelization overhead ranging from 0 to 1. That is, $\alpha = 0$ means perfect parallelization with zero overhead and $\alpha = 1$ means no reduction of a thread execution time due to parallelization. Each task τ_i is assumed to have the implicit deadline $D_i = T_i$ where T_i is randomly generated in the range of (0.2, 1.4) times the sum of single CPU thread version's WCETs of all vertexes. This deadline range is selected because beyond this range, the deadlines are either too

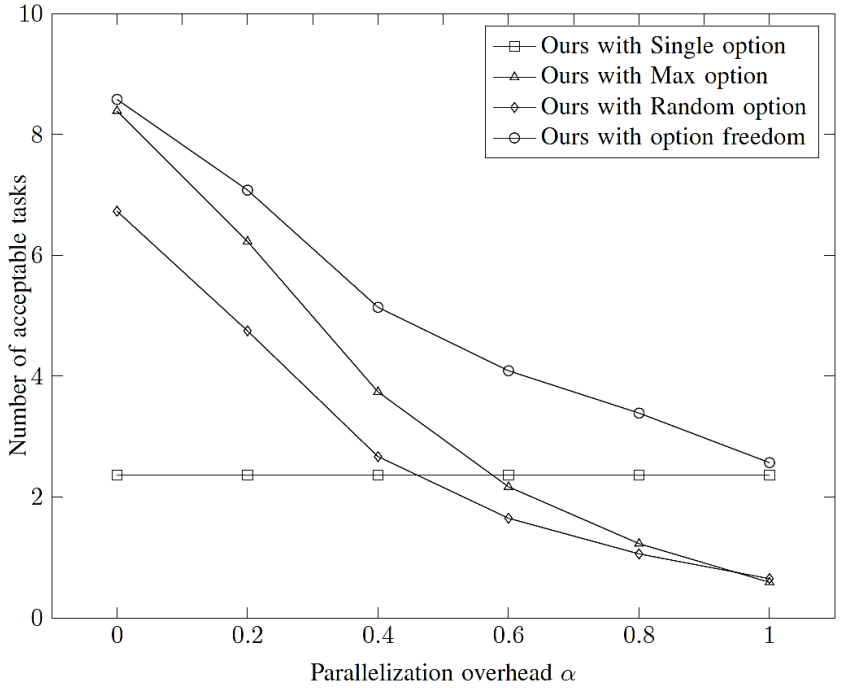
tight or too loose making no difference among methods. For such generated task list, we admit tasks one by one until the system is no longer schedulable to obtain the number of acceptable tasks. We repeat this experiment for 100 different task lists and average the number of acceptable tasks. Figure 5.1 (a) compares the average numbers of acceptable tasks by the following five methods as increasing the parallelization overhead α:

- Nelissen et al.'s method [6] assuming, for every vertex, the fixed single thread option (Nelissen with Single option), the fixed maximum parallelization option (Nelissen with Max option), and a randomly selected option (Nelissen with Random option),
- Qamhieh et al.'s method [30] that tries to enjoy the freedom of parallelization but assumes zero parallelization overhead (Qamhieh with option freedom), and
- Our proposed method (Ours with option freedom) explained in Section 4.1.

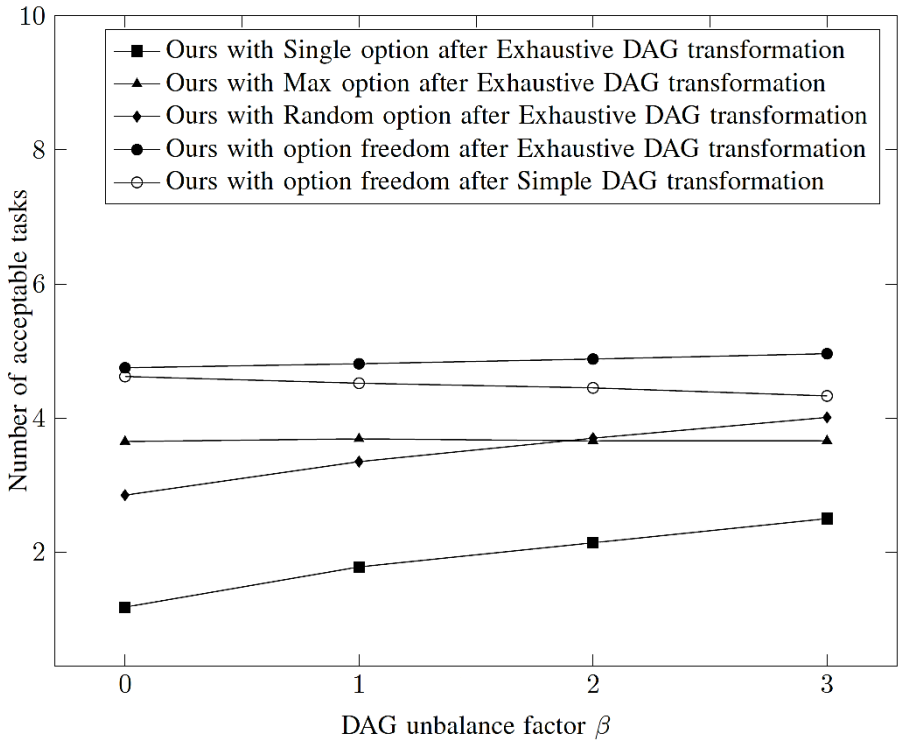
"Nelissen with Single option" method can accept the same number of tasks regardless of parallelization overhead since it always uses the non-parallelized option. On the other hand, "Nelissen with Max option" can accept much more tasks when the parallelization overhead is small but it becomes worse than "Nelissen with Single option" as the parallelization overhead becomes large. "Nelissen with Random option" is in between "Nelissen with Single option" and "Nelissen with Max option" depending on the parallelization overhead. "Qamhieh with option freedom" can accept more tasks than "Nelissen with Single option" and "Nelissen+Random" when the parallelization overhead is zero but it does not deal with other cases. Compared to these four existing methods, our proposed method can always accept more tasks,



(a) Simple DAG-based task and CPU resource model



(b) Simple DAG-based task and heterogeneous resource model



(c) General DAG-based task and heterogeneous resource model

Figure 5.1 Average numbers of acceptable tasks by different methods

i.e., up to two times more than “Nelissen with Single option”, up to four times more than “Nelissen with Max option”, and up to three times more than “Nelissen with Random option”, by enjoying the freedom of multiple parallelization options.

In the second experiment, we consider the problem of scheduling simple DAG-based tasks on 8 CPU cores and 7 GPU devices. For this, we similarly generate a list of tasks. However, in this case, based on each vertex’s WCET for the single CPU thread version, i.e., E, we make its thread versions up to $O_{ik}^{\text{cpu}} = 4$ and $O_{ik}^{\text{gpu}} = 2$ such that each thread of parallelization option $(O_{ik}^{\text{cpu}}, O_{ik}^{\text{gpu}})$ has the WCET of $E/(O_{ik}^{\text{cpu}} + 2 \cdot O_{ik}^{\text{gpu}}) + \alpha(E - E/(O_{ik}^{\text{cpu}} + 2 \cdot O_{ik}^{\text{gpu}}))$ on average, modeling more reduction of thread WCETs by using GPU threads. Figure 5.1 (b)

shows the results. Since there is no existing solution for scheduling parallel tasks on interchangeable CPU/GPU resources, we compare our proposed method that exercises full freedom of parallelization options (Ours with option freedom) with our methods assuming a fixed single thread option (Ours with Single option), a fixed maximum parallelization option (Ours with Max option), and a randomly selected option (Ours with Random option). From this, we can note that enjoying the parallelization freedom can make significant improvement also in the coexistence of CPU cores and GPU devices.

Finally, we consider the problem of scheduling general DAG-based tasks on 8 CPU cores and 7 GPU devices. For this, we randomly generate a general DAG with 8 vertexes for each task τ_i. The way for forming a DAG with 8 vertexes is controlled by a DAG unbalance factor β that models the length difference between the longest path and the shortest path in the DAG. The unbalance factor β ranges from 0 to 3 where β = 0 models a balanced DAG with all the same length paths and β = 3 models a largely unbalanced DAG with path length difference up to 3 vertexes. For each vertex of such formed DAG, we make its multiple parallelization options up to $O_{ik}^{\mathrm{cpu}} = 2$ and $O_{ik}^{\mathrm{gpu}} = 2$ in the same way. For each task τ_i', we randomly assign implicit deadline $D_i = T_i$ from the range of (0.7, 1.2) times the sum of single CPU thread version's WCETs of vertexes on the longest path. The parallelization overhead α is randomly selected in the range of (0, 1). Figure 5.1 (c) shows the results. In order to study the loss by our simple DAG transformation, we compare our method using simple DAG transformation denoted by "Ours with option freedom after Simple DAG transformation" with the best one obtained by exhaustive DAG transformation denoted by "Ours with option freedom after Exhaustive DAG transformation". "Ours with option freedom after Simple DAG transformation" shows very close performance to "Ours with option freedom after Exhaustive DAG transformation"

when β is small. Even when β is large, the loss by our simple DAG transformation is not that significant. This can be explained as follows. With the simple DAG transformation, unbalance can happen in terms of number of vertexes merged into one in the transformed simple DAG. However, such unbalance can be even out in terms of vertex density by controlling both local deadline and parallelization option, i.e., total computation requirement and minimum required computation time. On the other hand, if we cannot enjoy the freedom of multiple parallelization options, even the exhaustive DAG transformations, denoted by "Ours with Single option after Exhaustive DAG transformation", "Ours with Max option after Exhaustive DAG transformation", and "Ours with Random option after Exhaustive DAG transformation" cannot beat "Ours with option freedom after Simple DAG transformation" in the all range of β.

Chapter 6

Conclusion

This paper proposes optimal and near optimal algorithms for parallelizing and scheduling real-time tasks with multiple parallelization options on multiple CPU cores and multiple GPU devices. For a simplified problem with the simple DAG task model and the CPU resource model, an optimal algorithm is proposed. Extending this optimal algorithm, near optimal algorithms are proposed for the problem with the CPU/GPU resource model and the general DAG-based task model. Our simulation study says that the proposed algorithms can schedule up to two times more tasks with the same number of CPU cores and GPU devices by enjoying the freedom of multiple parallelization options. In the future, we plan to investigate how to trade such increased capacity for minimizing the energy consumption. We also plan to mitigate conservativeness of the partitioned scheduling for GPU devices, and to extend the system model so that it practically considers the communication, preemption, and migration overheads.

Bibliography

[1] C. Urmson, J. Anhalt, D. Bagnell, C. Baker, R. Bittner, M. Clark, J. Dolan, D. Duggins, T. Galatali, C. Geyer et al., "Autonomous driving in urban environments: Boss and the urban challenge," Journal of Field Robotics, vol. 25, no. 8, pp. 425 - 466, 2008.

[2] G. A. Elliott and J. H. Anderson, "Globally scheduled real-time multiprocessor systems with gpus," Real-Time Systems, vol. 48, no. 1, pp. 34 - 74, 2012.

[3] G. Levin, S. Funk, C. Sadowski, I. Pye, and S. Brandt, "Dp-fair: A simple model for understanding optimal multiprocessor scheduling," in 22nd Euromicro Conference on Real-Time Systems (ECRTS), 2010.

[4] P. Regnier, G. Lima, E. Massa, G. Levin, and S. Brandt, "Run: Optimal multiprocessor real-time scheduling via reduction to uniprocessor," in 32nd IEEE Real-Time Systems Symposium (RTSS), 2011.

[5] A. Saifullah, K. Agrawal, C. Lu, and C. Gill, "Multi-core real-time scheduling for generalized parallel task models," in 32nd IEEE Real-Time Systems Symposium (RTSS), 2011.

[6] G. Nelissen, V. Berten, J. Goossens, and D. Milojevic, “Techniques optimizing the number of processors to schedule multi-threaded tasks,” in 24th Euromicro Conference on Real-Time Systems (ECRTS), 2012.

[7] A. Saifullah, D. Ferry, J. Li, K. Agrawal, C. Lu, and C. Gill, “Parallel real-time scheduling of dags,” in IEEE Transactions on Parallel and Distributed Systems, 2014.

[8] Khronos Group, https://www.khronos.org/opencl/.

[9] S. K. Dhall and C. Liu, “On a real-time scheduling problem,” Operations Research, vol. 26, no. 1, pp. 127.140, 1978.

[10] S. Baruah, N. Cohen, C. Plaxton, and D. Varvel, “Proportionate progress: a notion of fairness in resource allocation,” in 25th annual ACM symposium on Theory of computing, 1993.

[11] D. Zhu, D. Mosse, and R. Melhem, “Multiple-resource periodic scheduling problem: how much fairness is necessary?” in 24th IEEE Real-Time Systems Symposium (RTSS), 2003.

[12] A. Srinivasan and J. H. Anderson, “Fair scheduling of dynamic task systems on multiprocessors,” Journal of Systems and Software, vol. 77, no. 1, pp. 67 - 80, 2005.

[13] H. Cho, B. Ravindran, and E. D. Jensen, “An optimal real-time scheduling algorithm for multiprocessors,” in 27th IEEE Real-Time Systems Symposium (RTSS), 2006.

[14] K. Funaoka, S. Kato, and N. Yamasaki, “Work-conserving optimal real-time scheduling on multiprocessors,” in 20th Euromicro Conference on Real-Time Systems (ECRTS), 2008.

[15] S. Kato and Y. Ishikawa, “Gang edf scheduling of parallel task systems,” in 30th IEEE Real-Time Systems Symposium (RTSS), 2009.

[16] G. Nelissen, V. Berten, V. N´elis, J. Goossens, and D. Milojevic, “U-edf: An unfair but optimal multiprocessor scheduling algorithm for sporadic tasks,” in 24th Euromicro Conference on Real-Time Systems (ECRTS), 2012.

[17] B. Andersson and A. Easwaran, “Provably good multiprocessor scheduling with resource sharing,” Real-Time Systems, vol. 46, no. 2, pp. 153 - 159, 2010.

[18] U. Verner, A. Schuster, and M. Silberstein, “Processing data streams with hard real-time constraints on heterogeneous systems,” in International Conference on Supercomputing (ICS), 2011.

[19] A. Wiese, V. Bonifaci, and S. Baruah, “Partitioned edf scheduling on a few types of unrelated multiprocessors,” Real-Time Systems, vol. 49, no. 2, pp. 219 - 238, 2013.

[20] G. Raravi, B. Andersson, V. N´elis, and K. Bletsas, “Task assignment algorithms for two-type heterogeneous multiprocessors,” Real-Time Systems, vol. 50, no. 1, pp. 87 - 141, 2014.

[21] J. H. Anderson and J. M. Calandrino, “Parallel real-time task scheduling on multicore platforms,” in 27th IEEE Real-Time Systems Symposium (RTSS), 2006.

[22] K. Lakshmanan, S. Kato, and R. Rajkumar, “Scheduling parallel real-time tasks on multi-core processors,” in 31st IEEE Real-Time Systems Symposium (RTSS), 2010.

[23] F. Fauberteau, S. Midonnet, and M. Qamhieh, “Partitioned scheduling of parallel real-time tasks on multiprocessor systems,” ACM SIGBED Review, vol. 8, no. 3, pp. 28 - 31, 2011.

[24] L. Nogueira, J. C. Fonseca, C. Maia, and L. M. Pinho, “Dynamic global scheduling of parallel real-time tasks,” in 15th IEEE International Conference on Computational Science and Engineering (CSE), 2012.

[25] B. Bado, L. George, P. Courbin, and J. Goossens, “A semi-partitioned approach for parallel real-time scheduling,” in 20th ACM International Conference on Real-Time and Network Systems, 2012.

[26] P. Courbin, I. Lupu, and J. Goossens, “Scheduling of hard real-time multi-phase multi-thread (mpmt) periodic tasks,” Real-time systems, vol. 49, no. 2, pp. 239 - 266, 2013.

[27] M. Holenderski, R. J. Bril, and J. J. Lukkien, “Parallel-task scheduling on multiple resources,” in 24th Euromicro Conference on Real-Time Systems (ECRTS), 2012.

[28] C. Liu and J. H. Anderson, “Supporting soft real-time dag-based systems on multiprocessors with no utilization loss,” in 31st IEEE Real-Time Systems Symposium (RTSS), 2010.

[29] S. Baruah, V. Bonifaci, A. Marchetti-Spaccamela, L. Stougie, and A. Wiese, “A generalized parallel task model for recurrent real-time processes,” in 33rd IEEE Real-Time Systems Symposium (RTSS), 2012.

[30] M. Qamhieh, S. Midonnet, and L. George, “Dynamic scheduling algorithm for parallel real-time graph tasks,” ACM SIGBED Review, vol. 9, no. 4, pp. 25 - 28, 2012.

[31] H. Ozaktas, C. Rochange, and P. Sainrat, “Automatic wcet analysis of real-time parallel applications,” in 13th International Workshop on Worst-Case Execution Time Analysis, 2013.

[32] R. Mangharam and A. A. Saba, “Anytime algorithms for gpu architectures,” in 32nd IEEE Real-Time Systems Symposium (RTSS), 2011.

[33] Asus, https://www.asus.com/Commercial Servers Workstations/ESC4000 G2/.

요약 (국문 초록)

멀티코어/GPGPU 실시간 스케줄링에 관한 기존의 연구들은 하나의 계산 유닛 (순차 태스크 모델의 태스크, 멀티 세그먼트 병렬 태스크 모델의 세그먼트, DAG 기반 병렬 태스크 모델의 정점) 이 미리 정해진 개수의 CPU 쓰레드와 GPU 쓰레드를 가지는 것으로 가정하였다. 그러나, 최근 OpenCL 프레임워크 등의 발전으로, 하나의 계산 유닛이 런타임에 동적으로 정하는 개수의 CPU와 GPU 쓰레드로 병렬화될 수 있게 되었다. 본 논문은 복수 병렬화 옵션을 가진 실시간 병렬 태스크들을 여러 개의 CPU 코어와 GPU 디바이스 자원 위에 최적으로 병렬화하고 스케줄하는 알고리즘들을 제안한다. 본 논문의 실험 결과에 의하면 제안하는 알고리즘들은 미리 정해진 병렬화 옵션을 사용하는 다른 알고리즘보다 두 배까지 많은 태스크들을 성공적으로 스케줄할 수 있다. 본 논문은 다수의 이종 자원 위에서 복수의 병렬화 옵션을 가지는 실시간 병렬 태스크들을 스케줄하는 문제의 해결을 최초로 시도하고 있다.

주요어 : 병렬화, 멀티코어, GPGPU, 실시간, 스케줄

학 번 : 2012-20739