國立臺灣大學電機資訊學院資訊工程學系

碩士論文

Department of Computer Science and Information Engineering

College of Electrical Engineering and Computer Science

National Taiwan University

Master Thesis

制式多處理器下即時排程的增進

Improvements on On-Line Uniform Multiprocessors Scheduling

朱百一

Bie-I Chu

指導教授：薛智文 博士

Advisor: Chih-Wen Hsueh, Ph.D.

中華民國 99 年 8 月

August, 2010

制式多處理器下即時排程的增進

指導教授：薛智文　　　研究生：朱百一

國立台灣大學資訊工程學研究所

摘要

在硬性即時環境下，處理週期性任務在多處理器平台上的排程，是電腦科學中一個基礎的問題。在這篇論文中，我們要考慮問題是對一個由 n 個獨立的任務和 m 個制式多處理器所形成的集合作線上排程。我們提出了一個最佳演算法，它能夠對所有可行的集合進行排程，並且保證所有的任務都能夠在期限內完成。在以往的成果中，過去的最佳演算法在每一次重新排程時，他的時間複雜度是 $O(n \lg n)$，並會產生至多 $O(n)$的工作遷移；但是我們的演算法在每一次重新排程時，除了改進時間複雜度，將之減少為 $O(\lg n)$之外，亦將工作遷移的數量減少至 $O(1)$。除此之外，若是對非週期性任務在制式多處理器下進行排程，我們的演算法亦可保證他的排程長度是最短的。

關鍵字：即時，制式，多處理器，最佳，線上，演算法，排程

Improvements on On-Line Uniform Multiprocessors Scheduling

By
Bie-I Chu
A Thesis Submitted To
Institute of Computer Science and Information Engineering
College of Electrical Engineering and Computer Science
National Taiwan University
For The Degree of Master
in
Computer Science and Information Engineering
July 2010

Abstract

Scheduling periodic tasks on multiprocessor platform in a hard-real-time environment is one of the fundamental problems in computer science. In this thesis, we consider the problem of on-line scheduling a set of n independent periodic tasks on m uniform processors. We present an optimal scheduling algorithm in the sense that the algorithm is able to schedule any feasible set to meet all deadlines. From previous works, the optimal algorithm gave an $O(n)$ bound for number of task migration and an $O(n \lg n)$ bound for time complexity on each rescheduling. But for our algorithm, we reduce both number of task migration and time complexity to $O(1)$ and $O(\lg n)$ respectively. Our algorithm also guarantees minimal schedule length for scheduling non-periodic tasks on uniform multiprocessors.

Keyword: real-time, uniform, multiprocessor, optimal, on-line, algorithm, scheduling.

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

1.1 Background

1.1.1 Multiprocessor Platforms

The scheduling of a set of tasks on parallel multiprocessors platform is a basic problem in computer science with a large number of applications. From previous works [5] [2], there are at least three different kinds of multiprocessors platforms that scheduling theorists are interested in:

- *Identical multiprocessors platform.* All processors are identical. In other words, all processors have the same computing capacity.
- *Uniform multiprocessors platform.* Each processor in the platform is characterized by its own computing capacity. Once a task is scheduled onto a uniform processor with speed s for t time units, then $s \times t$ units of execution requirements are completed.
- *Unrelated multiprocessors platform.* There exists an execution rate $r_{i,j}$ be-

tween each pairs of processor P_i and task T_j. Once a task T_j is scheduled onto processor P_i for t time units, then $r_{i,j} \times t$ units of execution requirements are completed.

1.1.2 Optimal Scheduler

The time to finish a given set of tasks on multiprocessor platform provides a measure for the performance of scheduling algorithms. Therefore, in general, a scheduling algorithm is optimal if and only if the algorithm can schedule tasks on multiprocessor platform with minimal schedule length (or finish time) [7]. But for periodic tasks, the schedule will not finish. Hence, we cannot use the schedule length to measure the performance of scheduling algorithms. In stead, a scheduling algorithm is said to be optimal if and only if the algorithm can schedule all feasible set of periodic task on multiprocessor platforms that all task complete by their deadlines [2] [5].

There are still other criterions for measuring the performance of scheduling algorithm such as numbers of task migrations and time complexity.

1.1.3 On-line Scheduling

On-line scheduling algorithms make scheduling decisions at each time-instant based upon the characteristics of the tasks that have arrived thus far [5]. An example of on-line scheduling algorithm would be the earliest deadline first (EDF) scheduling algorithm [8].

1.2 Motivations

There were many works had been done under different platforms. In 1969, Muntz and Coffman [11] had presented an optimal static level algorithm for scheduling tasks on identical multiprocessors platforms. Then in 2006, Cho et al. [3] based on P-fair [1] and L-C plane [4], created the *Time and Local Execution Time Planes* (or T-L planes) model and provided the *Largest Local Remaining Execution Time First* (or LLREF) optimal on-line scheduling algorithm for scheduling tasks on identical multiprocessors platforms. Finally in 2008, Chen et al. [2] extended the T-L plane model, created the *Time and Local Execution Requirement plane* (or T-L_{er} plane) model and presented the *Precaution Cut Greedy* (or PCG) optimal on-line scheduling algorithm for scheduling tasks on uniform multiprocessors platforms, which is the first on-line optimal scheduling algorithm for uniform multiprocessors platforms. Although the problem of scheduling a set of periodic tasks on uniform multiprocessors platform had been solved by the PCG algorithm, the scheduling cost of the PCG algorithm is high. We are interested in developing another optimal on-line scheduling algorithm that would solve the problem efficiently and with lower time complexities and task migration costs.

1.3 Contributions and Organizations

Our contributions are as follows:

- We present an optimal on-line scheduling algorithm for uniform multiprocessors called *Random Divide* scheduling algorithm, which uses a divide-and-conquer method to solve the problem.
- We present an optimal on-line scheduling algorithm for uniform multiprocessors called *Precaution Group Merge* scheduling algorithm, which is based on the Random Divide and the PCG algorithm and reduces both the time complexity and the number of task migrations.
- We present an optimal on-line scheduling algorithm for uniform multiprocessors called *Preprocessed Precaution Group Merge* scheduling algorithm, which is based on the PGM algorithm and guarantees an $O(1)$ bound for the number of task migrations and an $O(\lg n)$ bound for time complexity on each rescheduling.

The remainder of this thesis is organized as follows. In chapter 2, we describe our problems, definitions and assumptions, and the feasibility conditions for uniform multiprocessors. In chapter 3, we introduce the T-L$_{er}$ plane model and the PCG scheduling algorithm [2] presented by Chen as they are the basis of our work. In chapter 4, we present the Random Divide (RD), the Precaution Group Merge (PGM), and the Preprocessed Group Merge (PPGM) scheduling algorithms and prove their optimalities. In chapter 5, we analyze the performances and complexities of our scheduling algorithms. This thesis is concluded in chapter 6.

Chapter 2

Definitions, Assumptions, and Feasibility Condition for Uniform Multiprocessors

Before we discuss about the details of the scheduling algorithms, we introduce the basic definitions and assumptions of uniform multiprocessor scheduling.

2.1 Definitions and Assumptions

We discuss the problem of dynamic-priority scheduling of hard-real-time systems on a uniform multiprocessors platform of m processors and n tasks.

For processors and tasks, we have the following definitions:

Definition 1. *A processor $P_i = (s_i)$ is characterized by a positive constant s_i, where s_i represents the computing capacity (or speed) of P_i.*

Definition 2. *A task $T_i = (c_i, p_i)$ is characterized by two positive constants*

c_i and p_i, where c_i represents the execution requirement and p_i represents the period of T_i. We define the utilization of a task T_i to be $u_i = c_i/p_i$.

Moreover, for tasks, we have the following assumptions:

- All tasks are periodic and the deadline is equal to the end of the period.
- All tasks are independent that tasks do not share resources or have any precedence.
- Each task can be assigned with only one processor at a time.
- Tasks are allowed to arbitrarily migrate across processors during their execution.

Throughout this thesis, we will use the set $\mathcal{P} = \{P_i | 1 \leq i \leq m\}$ to represent the m-processor uniform multiprocessor platform and the set $\mathcal{T} = \{T_i | 1 \leq i \leq n\}$ to represent n periodic tasks system. Without loss of generality, we assume that both $\mathcal{P}$ and $\mathcal{T}$ are indexed in a non-increasing manner that $s_i \geq s_{i+1}$ for all $1 \leq i < m$ and $u_i \geq u_{i+1}$ for all $1 \leq i < n$.

We further assume $m \leq n$, because when $m > n$, the slower processors will never be used.

2.2 Feasibility Condition for Uniform Multiprocessors

Many works had been done for uniform multiprocessors scheduling. Funk et al. [5] presented the feasibility condition for uniform multiprocessors. We introduce their theorems here.

Theorem 1. *(Funk et al. [5]) Consider a set $\mathcal{T} = \{T_i | 1 \leq i \leq n\}$ of n periodic tasks indexed according to non-increasing utilization and a set $\mathcal{P} = \{P_i | 1 \leq i \leq m\}$ of $m \leq n$ uniform processors indexed according to non-increasing speed. Let $S_k = \sum_{i=1}^{k} s_i$ for all $1 \leq i \leq m$ and let $U_k = \sum_{i=1}^{k} u_i$ for all $1 \leq i \leq n$. Periodic tasks system $\mathcal{T}$ can be scheduled to meet all deadlines on uniform multiprocessor platform $\mathcal{P}$ if and only if the following constraints hold*

$$S_k \geq U_k, \forall 1 \leq k \leq m \tag{2.1}$$

$$S_m \geq U_n \tag{2.2}$$

We call the set $\{\mathcal{T}, \mathcal{P}\}$ a feasible set.

2.3 Independent Feasible Sets

Based on feasibility conditions, we introduce independent feasible sets:

Definition 3. *An independent feasible set is a feasible set of n tasks and m uniform processors such that*

$$\textit{if } 1 < m \leq n, s_i \neq u_j, \forall 1 \leq i \leq m, 1 \leq j \leq n, \tag{2.3}$$

$$S_k > U_k, \forall 1 \leq k \leq m-1, \tag{2.4}$$

and

$$S_m \geq U_n. \tag{2.5}$$

In other words, an independent feasible set is a feasible set with stricter constraints: $S_k > U_k$, for all $1 \leq k \leq m-1$; $s_i \neq u_j$, for all $1 \leq i \leq m$ and $1 \leq j \leq n$, if $1 < m \leq n$.

We call $S_k > U_k$ the kth feasibility constraint and $S_m \geq U_n$ the mth feasibility constraint for an independent feasible set.

There are two reasons that the set is named independent:

- Each independent feasible set can be scheduled independently that no context switch would occur across independent feasible sets. In other words, context switch would only occur on the tasks within an independent feasible set.
- Each task in an independent feasible set can be scheduled onto processor independently in an arbitrary order.

We will prove the above properties in the following chapters.

2.4 Separability of Feasible Sets

Lemma 1. *For a feasible set of n tasks and m uniform processors, if there exist k such that $S_k = U_k$, then the set can be reduced into one independent feasible* *set containing k' tasks and k' processors and one feasible set containing $n-k'$ tasks and $m-k'$ processors, where k' is the least k such that $S_k = U_k$.*

Proof. Suppose $S_{k'} = U_{k'}$ for the original set. We separate the original set into $\mathcal{S}_1 = \{\mathcal{T}_1, \mathcal{P}_1\}$ and $\mathcal{S}_2 = \{\mathcal{T}_2, \mathcal{P}_2\}$ where $\mathcal{T}_1 = \{T_i | 1 \le i \le k'\}$, $\mathcal{P}_1 = \{P_i | 1 \le i \le k'\}$, $\mathcal{T}_2 = \{T_i | k' < i \le n\}$, and $\mathcal{P}_2 = \{P_i | k' < i \le m\}$.

That is, we make the first k' tasks and first k' processors form one set and the rest tasks and processors form the other set. Since k' is the least k such that $S_k = U_k$, we have

$$S_i > U_i, \forall 1 \le i \le k' - 1.$$

Therefore $\mathcal{S}_1$ is an independent feasible set by definition.

For $\mathcal{S}_2$, the jth, $1 \le j \le m-k$, feasibility constraints becomes

$$S_j = \sum_{i=k'+1}^{k'+j} s_i = S_{k'+j} - S_{k'} \ge U_{k'+j} - U_{k'} = \sum_{i=k'+1}^{k'+j} u_i = U_j.$$

For all $1 \le j \le m-k'$, the inequality would hold, thus $\mathcal{S}_2$ is a feasible set. □

Lemma 2. *For a feasible set of n tasks and m uniform processors, if there exist $s_k = u_{k'}$, where $1 \le k \le m$ and $1 \le k' \le n$ (i.e. the speed of a processor is equal to the utilization of a task), then the set can be reduced into one feasible set containing $n-1$ tasks and $m-1$ processors and one independent feasible set containing one task and one processor.*

Proof. Suppose $s_k = u_{k'}$ for the original set, we separate the original set into

$\mathcal{S}_1 = \{\mathcal{T}_1, \mathcal{P}_1\}$ and $\mathcal{S}_2 = \{\mathcal{T}_2, \mathcal{P}_2\}$ where $\mathcal{T}_1 = \{T_{k'}\}$, $\mathcal{P}_1 = \{P_k\}$, $\mathcal{T}_2 = \{T_i | 1 \le i \le n$ where $i \ne k'\}$, and $\mathcal{P}_2 = \{P_i | 1 \le i \le m$ where $i \ne k\}$.

$\mathcal{S}_1$ is an independent feasible set by definition since $S_1 = s_k \ge u_{k'} = U_1$.

For $\mathcal{S}_2$, there are two cases: if $k \le k'$, then the jth, $1 \le j \le m-1$, feasibility constraints becomes

$$S_j = \sum_{i=1}^{j} s_i \ge \sum_{i=1}^{j} u_i = U_j, \text{ for } 1 \le j < k,$$

$$S_j = \sum_{i=1}^{k'} s_i - s_k - \sum_{i=j+2}^{k'} s_i \ge \sum_{i=1}^{k'} u_i - \sum_{i=j+1}^{k'} u_i = U_j, \text{ for } k \le j < k',$$

and

$$S_j = \sum_{i=1}^{j} s_i - s_k \ge \sum_{i=1}^{j} u_i - u_{k'} = U_j, \text{ for } k' \le j \le m-1.$$

Otherwise, if $k \ge k'$, then the jth, $1 \le j \le m-1$, feasibility constraints becomes

$$S_j = \sum_{i=1}^{j} s_i \ge \sum_{i=1}^{j} u_i = U_j, \text{ for } 1 \le j < k',$$

$$S_j = \sum_{i=1}^{j} s_i \ge \sum_{i=1}^{k'-1} u_i + \sum_{i=k'+1}^{j+1} u_i = U_j, \text{ for } k' \le j < k,$$

and

$$S_j = \sum_{i=1}^{j} s_i - s_k \ge \sum_{i=1}^{j} u_i - u_{k'} = U_j, \text{ for } k \le j \le m-1.$$

For both cases, all jth, $1 \le j \le m-1$, feasibility constraints would hold, thus $\mathcal{S}_2$ is a feasible set. $\square$

Theorem 2. *All feasible sets of n tasks and m processors can be reduced into* *independent feasible sets that scheduling these independent feasible sets is equivalent of scheduling the original feasible set.*

Proof. According to lemma 1 and lemma 2, for any given feasible set, if there exists k such that $E_k = L_k$ or any j, i such that $e_j = l_i$, we can recursively apply the reduction algorithm to the separate the set until all of the sets are not reducible. That is, all sets become independent feasible sets. Furthermore, since independent feasible sets are by definition feasible sets, these independent feasible sets can schedule independently. Therefore, no context switch would occur across independent feasible sets. □

Chapter 3

Precaution Cut Greedy Algorithm and The T-L$_{er}$ Plane

Our works are mainly based on the Precaution Cut Greedy (PCG) algorithm and the T-L$_{er}$ planes presented by Chen et al. [2]. Therefore in this chapter, we will introduce the PCG algorithm and the T-L$_{er}$ plane model.

3.1 T-L$_{er}$ Planes

Chen extended the T-L plane model [3] and the L-C plane model [4], created the T-L$_{er}$ plane model [2]. T-L$_{er}$ planes stands for **T**ime and **L**ocal **E**xecution **R**equirement planes, which models the behaviors of tasks in uniform multiprocessor platform. As shown in figure 3.1, task T_i arrives at time t and its deadline is $t + p_i$. Also, as the figure shows, T_i is assigned with processors P_1 and P_2 alternatively in different time intervals. Since P_1 is faster than P_2, task T_i would have higher execution rate on P_1 than P_2. Therefore, as shown in figure 3.1, the

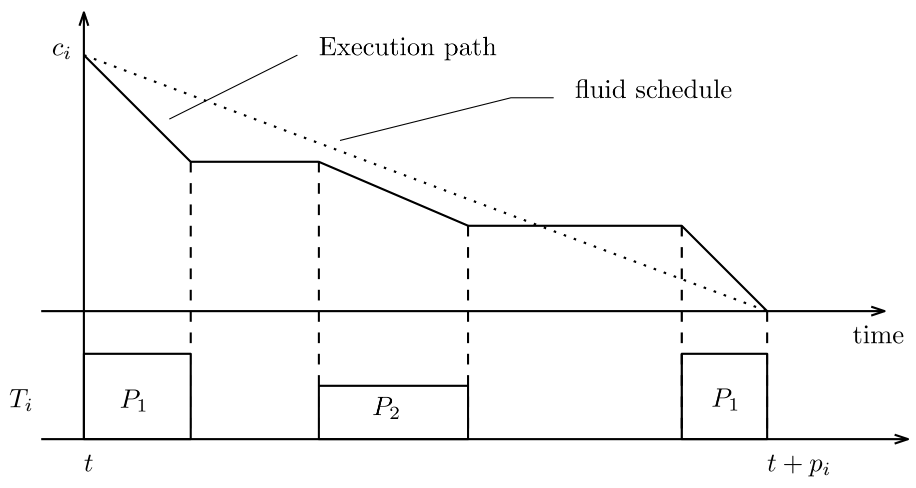

Figure 3.1: The T-L$_{er}$ plane

slope of the execution path is larger while the task is assigned with P_1. Finally, the fluid schedule of T_i represents the average execution rate during the entire period, which is shown in figure 3.1 by a dotted line. The average execution rate of T_i can be easily calculated by c_i/p_i, which is equal to the utilization of T_i.

While n tasks are considered together, their fluid schedules can be constructed as shown in figure 3.2. Similar to the T-L plane [3], a right triangle can be found between every two consecutive end of periods and these right triangles would divide the T-L$_{er}$ plane into contiguous time intervals. We called the kth right triangle as the kth T-L$_{er}$ plane and let the length of the corresponding time interval be t_f^k. Since no two right triangle would overlapped with each other, we could schedule in one kth T-L$_{er}$ plane without considering the deadline of each task [6] – just to consider the local remaining execution requirement in time interval of length t_f^k. Since the local remaining execution requirement

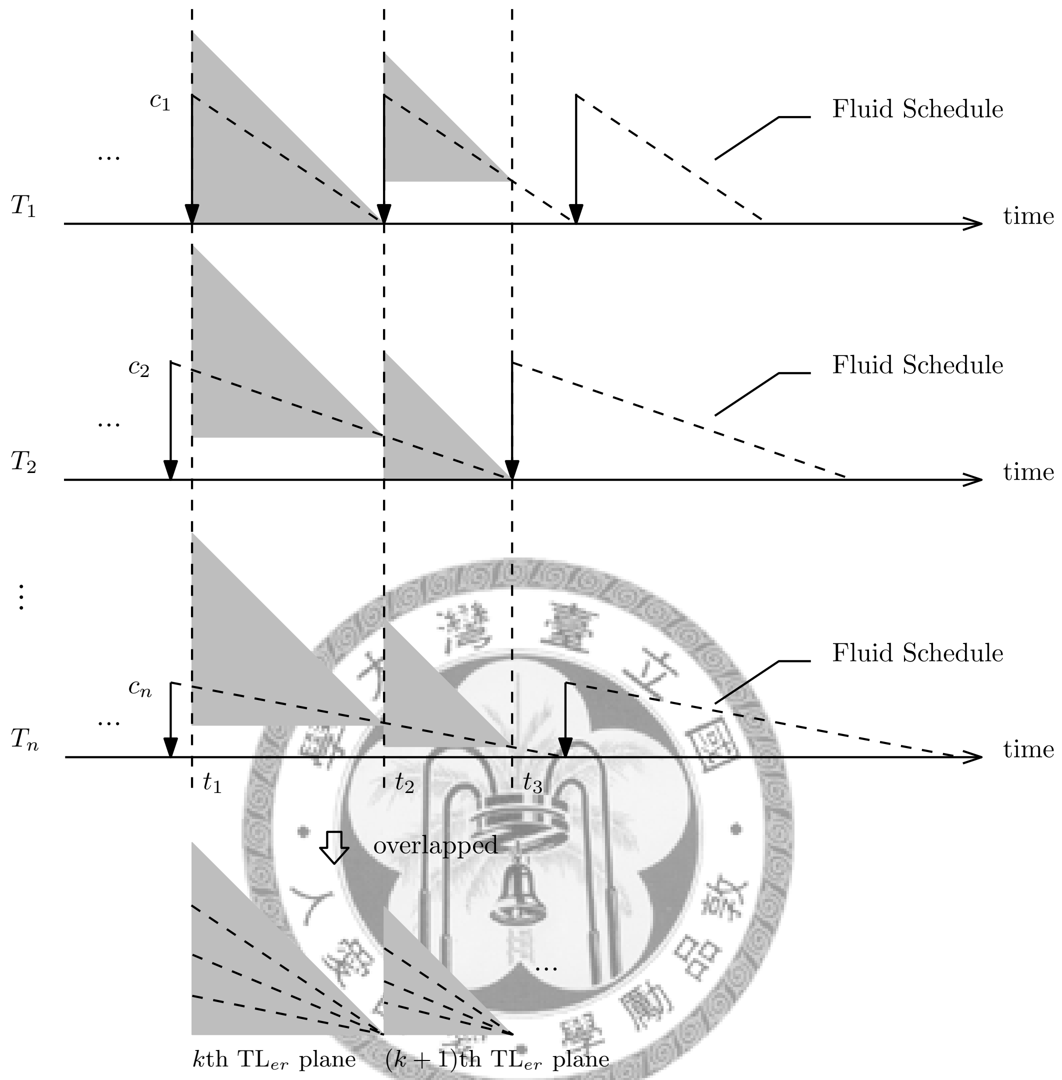

Figure 3.2: T-L$_{er}$ plane

of each task is proportional to the average execution rate and the length of the time interval, all kth T-L$_{er}$ planes are similar to each other. As long as we can finish all the jobs before the end of the T-L$_{er}$ plane, we could give an optimal scheduling algorithm.

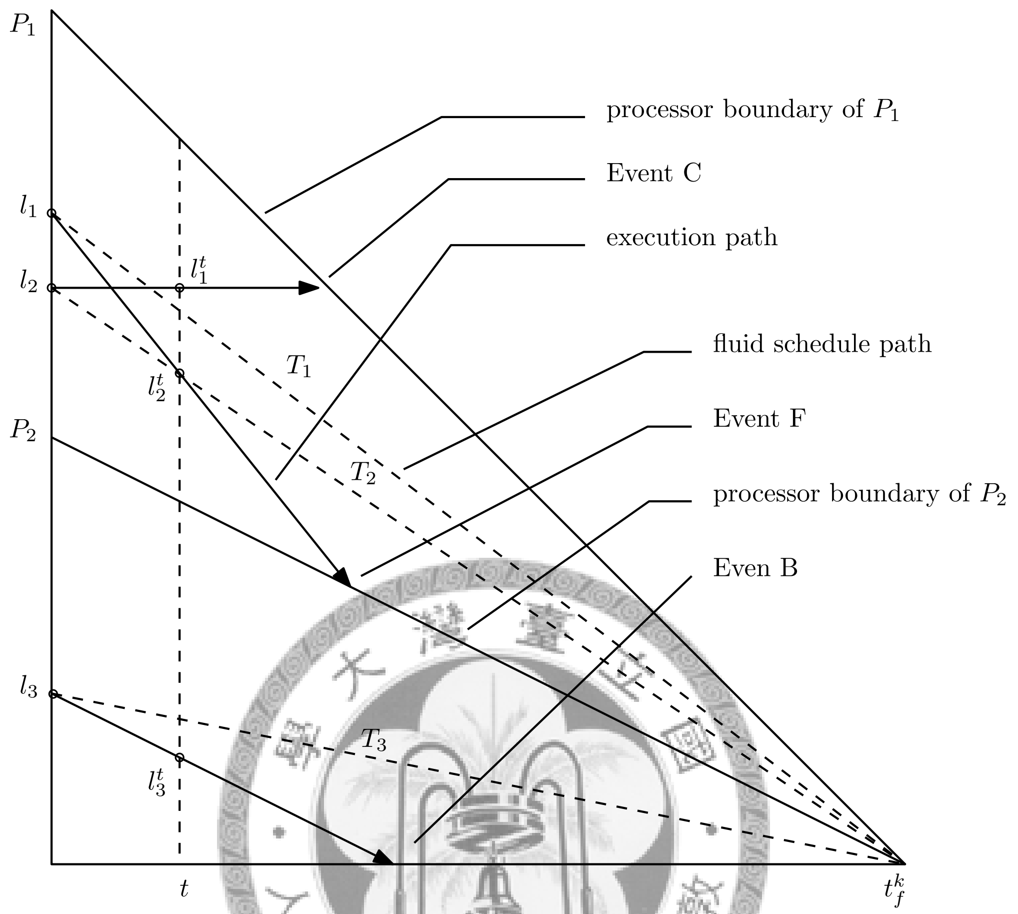

Figure 3.3: kth T-L$_{er}$ plane

3.2 Definitions in One T-L$_{er}$ Plane

In the kth T-L$_{er}$ plane at time $0 \leq t \leq t_f^k$, we define the local computing capacity for processor P_i to be $e_i^t = s_i \times (t_f^k - t)$ and $E_k^t = \sum_{i=1}^{k} e_i^t$. The local computing capacity represents the actual computing capacity of a processor in the T-L$_{er}$ plane. For simplicity, we define $e_i = e_i^0$ and $E_k = E_k^0$.

Chen defined $l_{i,j}$ to represent the local remaining execution requirement of task T_i at time t_j. The value of $l_{i,0}$ is equal to $u_k \times t_f^k$. Chen also defined the local utilization $r_{i,j} = l_{i,j}/(t_f^k - t_j)$ of task T_i at time t_j in the T-L$_{er}$ plane. But for simplicity, we rewrite t_j as t and define the remaining execution requirement of task T_i at time t as $l_{i,t}$ and the local utilization of task T_i at time t as $r_{i,t} = l_{i,t}/(t_f^k - t)$ in the T-L$_{er}$ plane.

As shown in figure 3.3, during the execution, the execution paths of tasks might intersect with each others. Thus in a T-L$_{er}$ plane, at time t, $l_{i,t} \geq l_{i+1,t}$ would not guaranteed to hold unless $t = 0$. Therefore, to distinguish with Chen, we define l_i^t to be the ith largest local remaining execution requirement at time t and $L_k^t = \sum_{i=1}^{k} l_i^t$. Also, we define $l_i = l_i^0$ and $L_k = L_k^0$ for simplicity. Note that here l_i^t do not guaranteed to be the local remaining execution requirement of T_i at time t unless $t = 0$. Also note that $l_i = l_i^0 = l_{i,0}$.

3.3 Events in One T-L$_{er}$ Plane

In T-L$_{er}$ plane, Chen observed three kinds of time instances (or events) that rescheduling is needed. These events are bottom hitting events (or Event B), ceiling hitting events (or Event C), and floor hitting events (or Event F).

Definition 4. *(Chen et al. [2]) Event B occurs when the local remaining execution requirement of a task T_i is equal to 0, and the execution path of the task would hit the bottom of the T-L$_{er}$ plane.*

Definition 5. *(Chen et al. [2]) Event C occurs when the local remaining execution requirement of a task T_i is equal to e_1 (i.e. the computing capacity of* *processor P_1), and the execution path of the task would hit the ceiling (i.e. the processor boundary of P_1) of the T-L_{er} plane.*

Definition 6. *(Chen et al. [2]) Event F occurs when the local remaining execution requirement of a task T_i is equal to the computing capacity of a processor P_j, and the execution path of the task T_i would hit the processor boundary of processor P_j in the T-L_{er} plane.*

Note that there are two situations for Event F. If the local remaining execution requirement of task T_i is originally greater than the local computing capacity of processor P_j, then the execution path of T_i would hit the processor boundary of P_j from top. Otherwise, if the local remaining execution requirement of task T_i is less than the local computing capacity of processor P_j, then the execution path of T_i would hit the processor boundary of P_j from bottom.

Whenever Event B or Event C occurs, it is apparent that a rescheduling is required. For Event F, Chen stated that [2]: *Although, when event F occurs, it is not necessary to reschedule to satisfy FG condition (i.e. the feasibility conditions), it is the precaution time instance to reschedule for our optimal scheduling algorithm. Whenever any of these three events occurs, we will reschedule all the tasks in our optimal scheduling algorithm.*

In our work, we defined a new time instance called Event V:

Definition 7. *For a feasible set of n tasks and m processors, in a T-L_{er} plane, Event V happens at time t whenever $E_k^t = L_k^t$, $k \neq m$, holds.*

In other words, Event V is the time instance that the kth feasibility constraint is about to be violated. It is apparent that whenever Event V occurs, a rescheduling is required. Otherwise the feasibility conditions would be violated and the set would become infeasible and non-schedulable.

3.4 The Precaution Cut Greedy Algorithm

The Precaution Cut Greedy (PCG) scheduling algorithm is a work-conserving algorithm [5] based on the idea of "precaution." The algorithm always assigns the task with largest remaining execution requirement to the fastest processor, and will reschedule on the occurrence of any C, F, or B event.

When any event occurs, there must be a task on a processor boundary in T-L_{er} plane otherwise the execution requirement of the tasks is equal to 0. PCG will remove the task and the processor associate with the event and reschedule by greedily assigning the task with largest remaining execution requirement to the fastest processor.

Algorithm 1. *Precaution Cut Greedy, Chen et al. [2]*

```
    Input: A set $\tau$ of $n$ tasks $\{T_1, T_2, ..., T_n\}$ with utilization $u_1, u_2, ..., u_n$.
           A set $\pi$ of $m$ processors $\{P_1, P_2, ..., P_m\}$ with speed $s_1, s_2, ..., s_m$.
 1. while any event [C|F|B] occurs at time $t$ do
 2.     if $s_i = r_{j,t}$ then
 3.         assign $T_j$ to $P_i$ until the end of the T-L$_{er}$ plane
 4.         remove $P_i$ from $\pi$, remove $T_j$ from $\tau$
 5.     else if $r_{j,k} = 0$ then
 6.         remove $T_j$ from $\tau$
 7.     while there are ready tasks do
 8.         assign task with largest remaining execution requirement to the
            fastest idle processor
 9.     end while
10. end while
```

Chen proved that the set of tasks and processors is feasible while any event occurs.

Theorem 3. *(Chen et al. [2]) When any event occurs, the set of tasks and processors is feasible by PG and PCG scheduling algorithm.*

Moreover, Chen proved that when any event occurs, the removal of the task and the processor associated with the event will not lead to the violation of feasibility conditions.

Theorem 4. *(Chen et al. [2]) When any event occurs, there exists a task on a* *processor boundary in T-L_{er} plane or the execution requirement of this task is equal to 0. If we remove the task and the processor, the remaining set of tasks and processors will still be feasible.*

Actually, theorem 4 can be explained by the separation of feasible set of theorem 2 in chapter 2. Therefore theorem 2 is a generalized form for Chen's theorem. Furthermore, we will state another generalized form of Chen's theorem 3 in the next chapter.

Chapter 4

Optimal Scheduling Algorithm for Uniform Processors

In this chapter, we will present the three scheduling algorithms: the Random Divide (RD) algorithm, the Precaution Group Merge (PGM) algorithm, and the Preprocessed Precaution Group Merge (PPGM) algorithm in the following sections.

4.1 Task Groups and Group Order

Before introducing our scheduling algorithms, three terms must be defined: *task groups, processor groups*, and *group order*.

Definition 8. *For a feasible set of* n *tasks and* m *uniform processors, we define task group* $\mathcal{G}_i = \{T_j | e_i < l_j < e_{i+1}, \forall 1 \leq i < m, 1 \leq j \leq n\}$.

In other words, a task group is a set of tasks that all tasks in the set are bounded by the same pair of consecutive processors in the same T-L$_{er}$ plane.

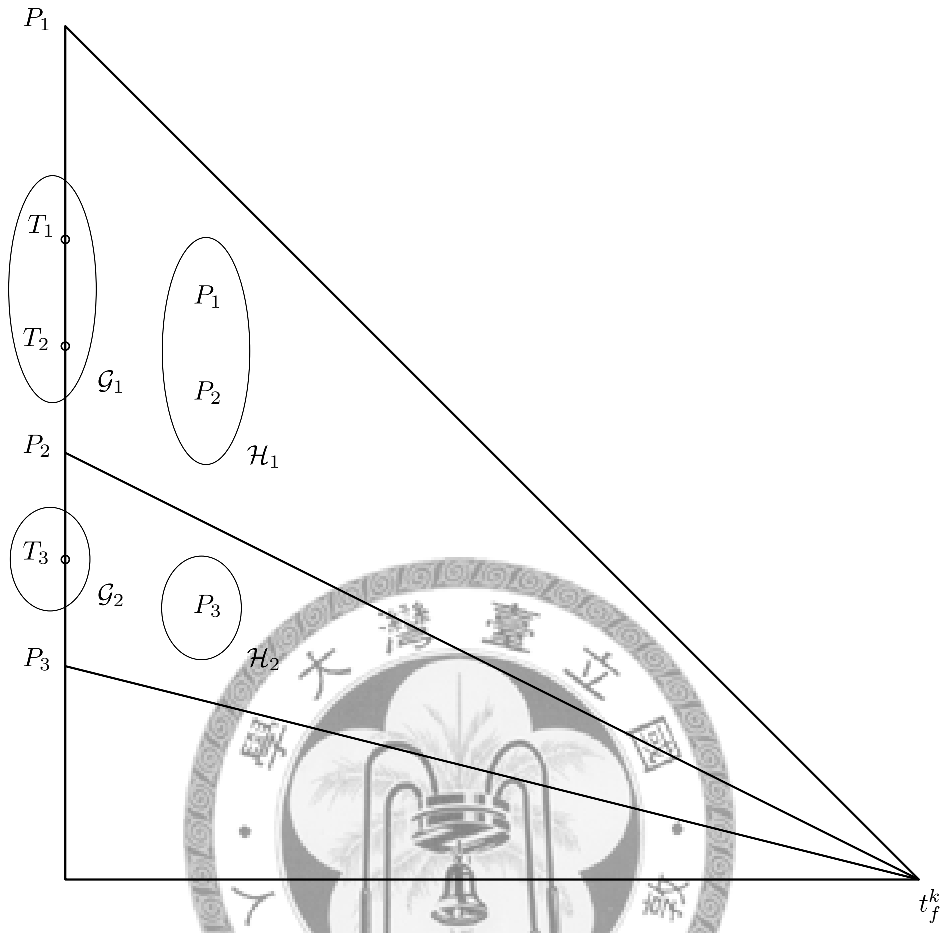

Figure 4.1: An Example of Tasks Groups and Processors Groups

Similar to task groups, a set $\mathcal{P}$ of processors can be divided into processor groups:

Definition 9. *For a feasible set of* n *tasks and* m *uniform processors, we define processor group* $\mathcal{H}_i = \{P_j\}$ *to be a set of processors where* $|\mathcal{H}_i| = |\mathcal{G}_i|$*, for all* $1 \leq i < m$*.*

Note here that we did not specify the construction of processor group $\mathcal{H}_i$. As long as $|\mathcal{H}_i| = |\mathcal{G}_i|$ would hold for all $1 \leq i < m$, we consider the processor groups as valid.

Definition 10. *A group order is an order on scheduling tasks onto processors that task T_i would be assigned processor P_j if and only if $T_i \in \mathcal{G}_k$ and $P_j \in \mathcal{H}_k$.*

In other words, we would schedule tasks and processors according to their group index: tasks would be assigned processors if and only if they have the same group index. If there are more than one tasks and processors have the same index, then the assignment is a random order for these tasks and processors.

The simplest way to form processor groups is to make faster processors into prior groups as shown in algorithm 2. As mentioned in the previous chapter, PCG scheduling algorithm is a work-conserving algorithm that would always assign processors to tasks in an order that the task with largest remaining execution requirement always gets the fastest processor. In other words, in a T-L$_{er}$ plane, if we make processor groups according to the computing capacity of processors, then the PCG algorithm would assign processors to tasks firstly in a group order for tasks in different groups, then secondly in a greedy order that the task with largest remaining execution requirement would get the fastest processor for tasks in the same group. Hence we can view the order of assignment of PCG algorithm as a subset of group order.

Algorithm 2. *Make Group*

```
Input  : An independent feasible set $\{\mathcal{T}, \mathcal{P}\}$ of $n$ tasks and $m$
         uniform processors
Output: Tasks groups $\mathcal{G}$ and processors groups $\mathcal{H}$
     // Initialize
1.   $i \leftarrow 1$, $j \leftarrow 1$
     // Scan $\mathcal{T}$ and $\mathcal{P}$ to make group
2.   while $i < n$ do
3.     $\mathcal{G}_i \leftarrow \phi$
4.     $\mathcal{H}_i \leftarrow \phi$
5.     while $l_j > e_{i+1}$ and $j < n$ do
6.       $\mathcal{G}_i \leftarrow \mathcal{G}_i \cup T_j$
7.       $\mathcal{H}_i \leftarrow \mathcal{H}_i \cup P_j$
8.       $j \leftarrow j + 1$
9.     end while
10.    $i \leftarrow i + 1$
11.  end while
```

4.2 The Random Divide Algorithm

The Random Divide (RD) scheduling algorithm is a divide-and-conquer algorithm that is shown in algorithm 3.

The RD scheduling algorithm takes a feasible set as input. Initially the

Algorithm 3. *Random Divide*

```
    Input: A set $\mathcal{T}$ of $n$ tasks $\{T_1, T_2, ..., T_n\}$ with utilization $u_1, u_2, ..., u_n$.
           A set $\mathcal{P}$ of $m$ processors $\{P_1, P_2, ..., P_m\}$ with speed $s_1, s_2, ..., s_m$.
1.  repeat
2.    if B event occurs then
3.      remove the task with local remaining execution requirement
        equals to 0
4.      assign another ready task to the idle processor
5.    else
6.      separate $\{\mathcal{T}, \mathcal{P}\}$ into independent feasible sets
7.      for each independent feasible set, assign processors to tasks in a
        random order
8.    end if
9.  until Event B or Event V occurs
```

algorithm would separate the feasible set into independent feasible sets, then for each independent feasible set, the algorithm would schedule tasks onto processors in a random order. The algorithm would keep this random order until Event V or Event B occurs. If Event B occurs, the task associates with the event is removed since its remaining execution requirement equals to 0, and then another ready task would be assigned to the idle processor. Otherwise, Event V occurs, then the RD algorithm will again separate those feasible sets associated with the event into independent feasible sets, and will reassign processors to tasks in a random order.

We give an example of execution in figure 4.2 that in the T-L$_{er}$ plane, initially the feasible set contains three processors P_1, P_2, and P_3 with local computing capacity $e_1 = 1$, $e_2 = 1/2$, $e_3 = 1/4$ respectively and three tasks T_1, T_2, and T_3 with local remaining execution requirement $l_1 = 3/4$, $l_2 = 5/8$, and $l_3 = 3/8$ respectively. At the beginning, all the feasibility constraints holds, and T_1, T_2, and T_3 are scheduled onto P_1, P_2, and P_3 respectively in a random order. Then at time t_1, Event V occurs that $E_1^{t_1} = L_1^{t_1}$. Hence we separate the original feasible set into independent feasible sets $\mathcal{S}_1 = \{\{T_1\}, \{P_1\}\}$ and $\mathcal{S}_2 = \{\{T_2, T_3\}, \{P_2, P_3\}\}$ and again reschedule these two sets by assigning P_1 to T_1, P_2 to T_3, and P_3 to T_2. Note here that $\mathcal{S}_1$ contains only one processor and one task, so it would not need to be rescheduled until the end of the T-L$_{er}$ plane. Then finally at time t_2, another Event V occurs in $\mathcal{S}_2$ that both $E_1^{t_2} = L_1^{t_2}$ and $E_2^{t_2} = L_2^{t_2}$. Therefore, we further separate $\mathcal{S}_2$ into independent feasible sets

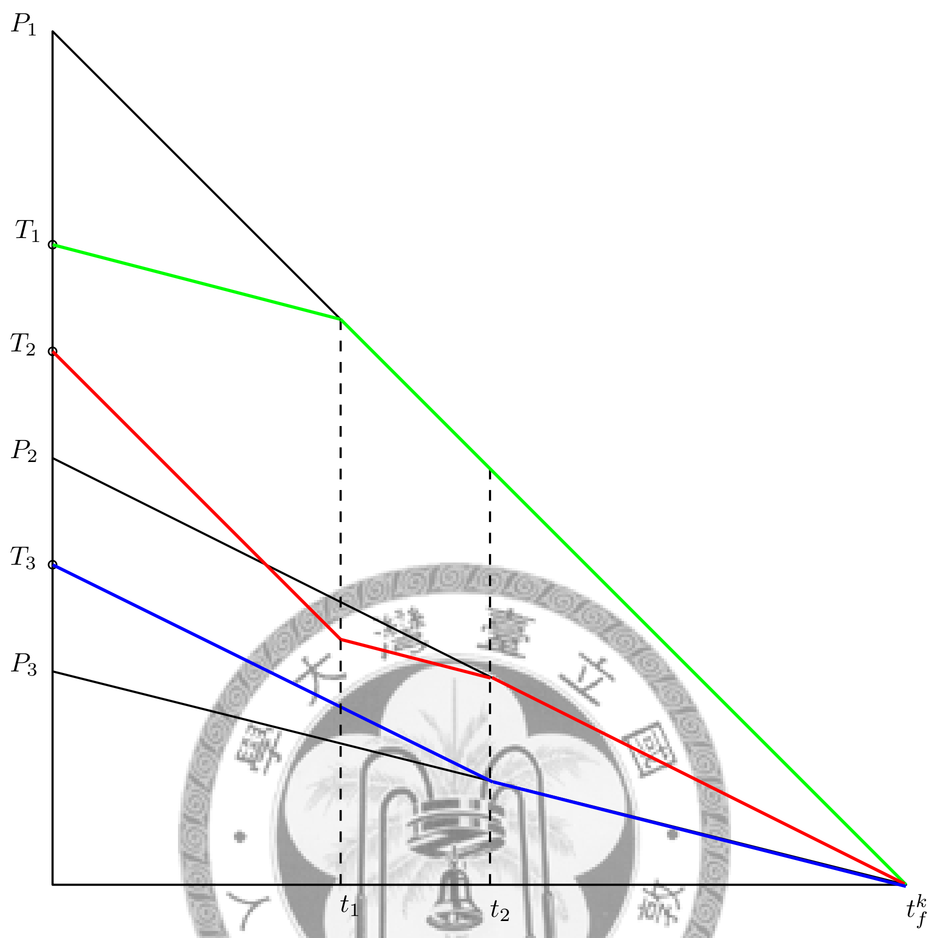

Figure 4.2: RD Scheduling Algorithm

$\{\{T_2\},\{P_2\}\}$ and $\{\{T_3\},\{P_3\}\}$. Now both independent feasible sets contain only one processor and one task, so they would not need to be rescheduled until the end of the T-L$_{er}$ plane. Hence we finish the schedule.

4.3 Proof of Optimality of RD Scheduling Algorithm

We will prove the optimality of RD scheduling algorithm in this section.

Theorem 5. *For a feasible set of n tasks and m uniform processors, in a T-L$_{er}$ plane, as long as all processors are fully utilized, the mth feasibility condition* *will always hold.*

Proof. The proof is straight forward. Since the set is feasible at the beginning, at time $t = 0$ in the T-L$_{er}$ plane, we have the mth feasibility condition hold at the time. That is,

$$S_m \geq U_n,$$

or equivalently,

$$E_m \geq L_n.$$

Now since we keep all processors to be fully utilized, we assign all m processors to tasks and there are no idle processors. Therefore, at any time t, in the same T-L$_{er}$ plane, the mth feasible condition becomes

$$E_m^t = E_m - t \times \sum_{i=1}^{m} s_i \geq L_n - t \times \sum_{i=1}^{m} s_i = L_n^t \tag{4.1}$$

The inequality $E_m^t \geq L_n^t$ will always hold regardless of t, thus we finish the proof.

□

Theorem 6. *For a feasible set of n tasks and m processors, the RD scheduling algorithm is optimal and feasible for uniform multiprocessors.*

Proof. We prove by showing that in a T-L$_{er}$ plane, whenever an Event V occurs, we can reschedule by reseparating the feasible sets into independent feasible sets, so that no feasibility constraints would be violated during the schedule.

According to the algorithm, we initially separate the given feasible set into independent feasible sets. Therefore, for each independent feasible set $\mathcal{S}_i$, initially we have

$$s_j \neq u_k, \forall 1 \leq j \leq m_i, 1 \leq k \leq n_i,$$

$$S_k > U_k, \forall 1 \leq k \leq m_i - 1,$$

and

$$S_{m_i} \geq U_{n_i},$$

or equivalently,

$$e_j \neq l_k, \forall 1 \leq j \leq m_i, 1 \leq k \leq n_i, \tag{4.2}$$

$$E_k > L_k, \forall 1 \leq k \leq m_i - 1, \tag{4.3}$$

and

$$E_{m_i} \geq L_{n_i}. \tag{4.4}$$

Where m_i and n_i represent the number of processors and the number of tasks of an independent feasible set $\mathcal{S}_i$. Note that here we assume $m_i > 1$ because if $m_i = 1$, then the independent feasible set would contain only one processor. Moreover, due to the feasibility constraints of independent feasible sets, it is apparent that the set is always feasible.

Since the scheduling algorithm would always assign all processors to tasks, all processors are kept fully utilized during the schedule. Therefore, according to theorem 5, we know for all independent feasible sets, the mth feasibility constraint would always hold.

We now focus on the independent feasible set $\mathcal{S}_j$ that Event V occurs during the schedule. Because $E_k > L_k \forall 1 \leq k \leq m_j - 1$, would hold for all independent feasible set $\mathcal{S}_j$, no matter that how the order of assigning processors to tasks is, before any of the feasibility constraints of $\mathcal{S}_j$ is violated, at time $t > 0$, there must be that

$$\exists k, 1 \leq k \leq m_i - 1, E_k^t = L_k^t. \tag{4.5}$$

In other words, at time t, Event V occurs and independent feasible set $\mathcal{S}_i$ becomes feasible sets at this moment.

We then separate $\mathcal{S}_j$ again by algorithm described in theorem 2. Therefore at time t, we would have another set of independent feasible sets and the initial conditions would be invariant.

In a T-L$_{er}$ plane, by no more than $m-1$ reseparations, the original set will be separated into m independent feasible sets with each set containing only one processor, which is the base case of tasks scheduling, and is apparently to be feasible and schedulable as mentioned before. Therefore, any feasible set can be schedulable using RD scheduling algorithm. That is, RD scheduling algorithm is optimal.

$\square$

4.4 The Precaution Group Merge Algorithm

Based on the RD and PCG scheduling algorithm, we develop the Precaution Group Merge (PGM) scheduling algorithm as shown in algorithm 4.

Algorithm 4. *Precaution Group Merge*

```
    Input:  A set $\mathcal{T}$ of $n$ tasks $\{T_1, T_2, ..., T_n\}$ with utilization $u_1, u_2, ..., u_n$.
            A set $\mathcal{P}$ of $m$ processors $\{P_1, P_2, ..., P_m\}$ with speed $s_1, s_2, ..., s_m$.
 1. repeat
 2.    if B event occurs then
 3.       remove the task with local remaining execution requirement
          equals to 0
 4.       assign another ready task to the idle processor
 5.    else
 6.       separate $\{\mathcal{T}, \mathcal{P}\}$ into independent feasible sets
 7.       for each independent feasible set, make group by algorithm 2
 8.       for each independent feasible set, assign tasks with processors in a
          group order
 9.    end if
10. until any [C|F|B] event occurs
```

The PGM scheduling algorithm borrows the idea of precaution of PCG algorithm, and would reschedule at the occurrence of any C, F, or B event.

The algorithm takes a feasible set as input. Initially the algorithm would separate the feasible set into independent feasible sets. Then for each independent feasible set, the algorithm would make task groups and processor groups and would assign tasks with processors in a group order.

The algorithm would keep this order until that any C, F, or B event to occur. When B event occurs, the algorithm simply removes the task associated with the event and reschedules another ready task to the idle processor. When C or F event occurs, we know there must be at least one task on a processor boundary in T-L_{er} plane, that is, we have remaining execution requirement of a task equal to the computing capacity of a processor. Therefore, we can apply the separation algorithm in theorem 2 to remove (or simply we say cut) the task and processor associated with the event from the current T-L_{er} plane. After such removal, the number of processors and tasks in the T-L_{er} plane both decreases by one, and we expect a merge of two task groups whose processor boundary is the same as the removed processor.

We use the same feasible set in the previous section for RD scheduling algorithm as an example to demonstrate the execution of PGM scheduling algorithm. The example is shown in figure 4.3.

At the beginning, all the feasibility constraints holds. Since $s_1 > u_1 > u_2 > s_2 > u_3 > s_3$, we make task groups by $\mathcal{G}_1 = \{T_1, T_2\}$, $\mathcal{G}_2 = \{T_3\}$ and processor groups by $\mathcal{H}_1 = \{P_1, P_2\}$, $\mathcal{H}_2 = \{P_3\}$. Following the group order, T_1 and T_2 would be assigned with P_1 and P_2, and T_3 would be assigned with P_3, thus we

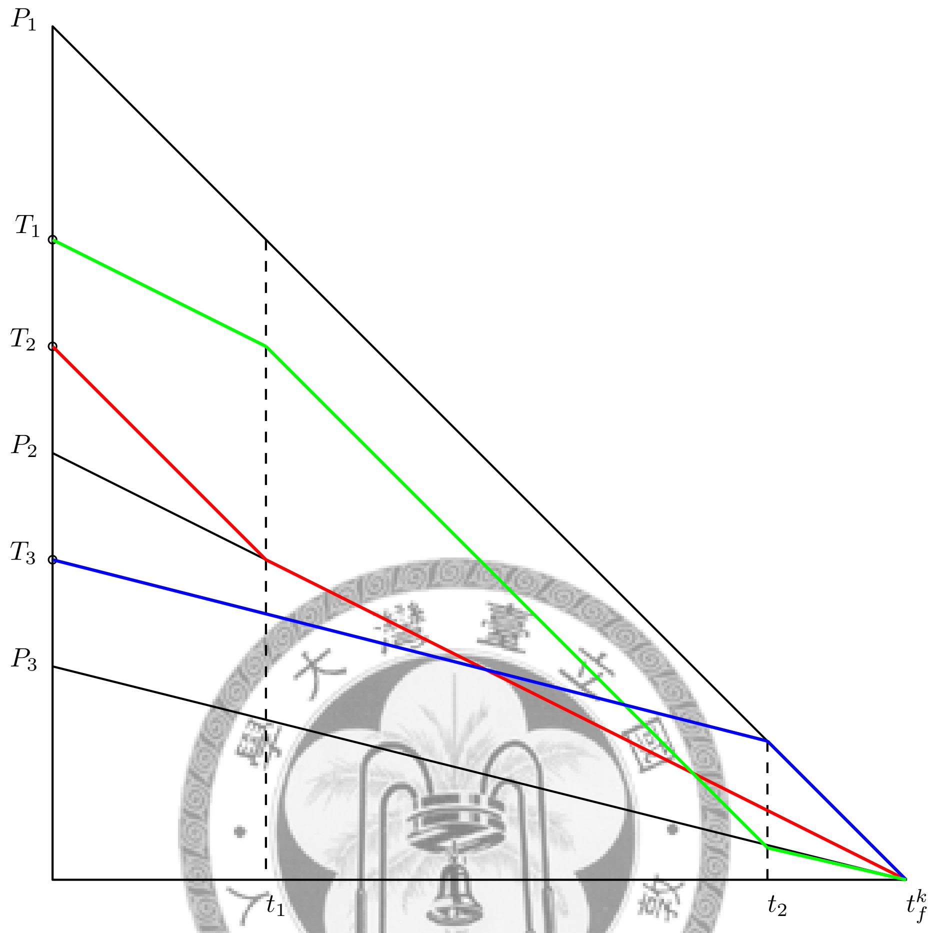

Figure 4.3: Example of PGM Scheduling Algorithm

assign T_1, T_2, and T_3 with P_2, P_1, and P_3 respectively. Then, at time t_1, Event F occurs that the local remaining execution requirement of T_2 is equal to the local computing capacity of P_2, therefore, we separate the original feasible set into independent feasible sets $\mathcal{S}_1 = \{\{T_1, T_3\}, \{P_1, P_3\}\}$ and $\mathcal{S}_2 = \{\{T_2\}, \{P_2\}\}$. Now for $\mathcal{S}_2$, it is an independent feasible set with only one processor and one task, therefore as long as we assign P_2 to T_2, it would not need to be reschedule until the end of the T-L$_{er}$ plane. For $\mathcal{S}_1$, it is the original feasible set with T_2 and P_2 removed, and we assign P_1 to T_1 and P_3 to T_3 to satisfy the group order.

Finally at time t_2, another Event F occurs when the execution path of T_1 hits the boundary of P_3 (note that at this time, the execution path of T_3 also hits the boundary of P_1). Now we again reschedule by remove $\mathcal{S}_3 = \{\{T_1\}, \{P_3\}\}$ from $\mathcal{S}_1$. Hence both $\mathcal{S}_1$ and $\mathcal{S}_3$ contain only one processor, so as long as we assign P_3 to T_1 and P_1 to T_3, they would not require further rescheduling until the end of the T-L$_{er}$ plane, and we finish the schedule.

4.5 Proof of Optimality of PGM Scheduling Algorithm

The PGM scheduling algorithm is similar to the PCG scheduling algorithm except that PGM would assign processors to tasks in a group order. But in another point of view, as long as we can prove that the violation of feasibility constraint (or Event V) would occur no earlier than the first occurrence of any C, F, or B event, PGM scheduling algorithm can be thought as a subset of the RD scheduling algorithm because group order is a subset of random order. Hence the PGM scheduling algorithm is feasible and optimal.

Theorem 7. *For an independent feasible set of n tasks and m processors, if we schedule using PGM algorithm, then Event V would occur no earlier than Event F in the first task group.*

Proof. Suppose there are n' tasks in the first task group, we have $\mathcal{G}_1 = \{T_i | 1 \leq i \leq n'\}$, $\mathcal{H}_1 = \{P_i | 1 \leq i \leq n'\}$, and $s_1 > u_1 \geq u_2 \geq ... \geq u_{n'} > s_2 \geq s_3 \geq ... \geq s_{n'}$. Also, the feasibility constraints should hold, therefore, we have

$E_k \geq L_k, \forall 1 \leq k \leq n'$.

Since we assign the fastest n' processors to tasks in the first task group, according to theorem 5, the n'th feasibility constraint would never be violated. Hence it would be no need to prove the occurrence of the violation of the n'th feasibility constraint.

Suppose t is the time that kth feasibility constraint is about to be violated, that is $E_k^t = L_k^t$, for all $1 \leq k < n'$. Because processors are assigned to tasks in a group order, it is actually a random order for tasks in the same group. Therefore, we assume initially at time 0, we assign processor P_i' to tasks T_i', and now at time t, the local remaining execution requirement of T_i' is $l'_{i,t}$ and $l'_{i,t} \geq l'_{i+1,t}$ for $1 \leq i < n'$. Let s'_i be the speed of processor P_i', then t can be calculated as follow:

$$E_k^t = L_k^t,$$

$$E_k^t = E_k - t \times \sum_{i=1}^{k} s_i = E_k - t \times S_k,$$

$$L_k^t = \sum_{i=1}^{k} l'_{i,0} - t \times \sum_{i=1}^{k} s'_i = L_k' - t \times \sum_{i=1}^{k} s'_i = L_k' - t \times S_k'.$$

Therefore,

$$t = \frac{E_k - L_k'}{S_k - S_k'}, \tag{4.6}$$

where $1 \leq k < n'$.

Suppose initially T_j is the task that is assigned with P_1, then the time of occurrence of Event F can be calculated as

$$\frac{l_j - e_2}{s_1 - s_2}. \tag{4.7}$$

Furthermore, we know for $1 \le k < n'$,

$$\begin{aligned} S_k - S'_k &= (s_1 + s_2 + ... + s_k) - (s'_1 + s'_2 + ... + s'_k) \\ &\le (s_1 + s_2 + ... + s_k) - (s_{n'} + s_{n'-1} + ... + s_{n'-k+1}) \\ &= S_k - \sum_{i=n'-k+1}^{n'} s_i, \end{aligned}$$

and

$$\begin{aligned} E_k - L'_k &= (e_1 + e_2 + ... + e_k) - (l'_1 + l'_2 + ... + l'_k) \\ &\ge (e_1 + e_2 + ... + e_k) - (l_1 + l_2 + ... + l_k) \\ &= E_k - L_k. \end{aligned}$$

Now we consider two cases. If $U_k \le S'_k$, then

$$\begin{aligned} \frac{E_k - L'_k}{S_k - S'_k} &\ge t_f \times \frac{S_k - U_k}{S_k - S'_k} \\ &\ge t_f \times 1 \\ &\ge t_f \times \frac{u_j - s_2}{s_1 - s_2} \\ &= \frac{l_j - e_2}{s_1 - s_2}. \end{aligned} \tag{4.8}$$

Otherwise, $U_k > S'_k$ implies $j > k$ because if $j \leq k$, we have

$$\begin{aligned}
U_k &= u_1 + u_2 + ... + u_k \\
&\leq u_1 + u_2 + ... + u_k + (u_{k+1} + u_{k+2} + ... + u_{n'}) - (s'_{k+1} + s'_{k+2} + ... + s'_{n'}) \\
&\leq (s'_1 + s'_2 + ... + s'_{j-1}) + s_1 + (s'_{j+1} + s'_{j+2} + ... + s'_k) \\
&= S'_k.
\end{aligned}$$

Hence,

$$\begin{aligned}
\frac{E_k - L'_k}{S_k - S'_k} &\geq t_f \times \frac{S_k - U_k}{S_k - \sum_{i=n'-k+1}^{n'} s_i} \\
&\geq t_f \times \frac{\sum_{i=k+1}^{n'} u_i - \sum_{i=k+1}^{n'} s_i}{\sum_{i=1}^{k} s_i - \sum_{i=n'-k+1}^{n'} s_i} \\
&\geq t_f \times \frac{\sum_{i=k+1}^{n'} u_i - \sum_{i=k+1}^{n'} s_i - \sum_{i=2}^{k} s_i + \sum_{i=n'-k+1}^{n'-1} s_i}{\sum_{i=1}^{k} s_i - \sum_{i=n'-k+1}^{n'} s_i - \sum_{i=2}^{k} s_i + \sum_{i=n'-k+1}^{n'-1} s_i} \\
&= t_f \times \frac{\sum_{i=k+1}^{n'} u_i - \sum_{i=2}^{n'} s_i + \sum_{i=n'-k+1}^{n'-1} s_i}{s_1 - s_{n'}} \\
&= t_f \times \frac{\sum_{i=k+1}^{n'} u_i - \sum_{i=2}^{n'-k} s_i - s_{n'}}{s_1 - s_{n'}} \\
&\geq t_f \times \frac{u_{k+1} - s_{n'}}{s_1 - s_{n'}} \\
&= \frac{l_{k+1} - e_{n'}}{s_1 - s_{n'}} \\
&\geq \frac{l_j - e_{n'}}{s_1 - s_{n'}} \\
&\geq \frac{l_j - e_2}{s_1 - s_2} \qquad (4.9)
\end{aligned}$$

Both cases implies the occurrence of Event F is earlier than the occurrence of Event V for all $1 \leq k < n'$, therefore we finish the proof.

□

Theorem 8. *For a feasible set of n tasks and m processors scheduled using PGM scheduling algorithm, there must be at least one Event C, Event B, or Event F occurs before Event V.*

Proof. Theorem 7 shows that Event V would occur no earlier‘ than Event F in the first task group. Therefore, we now consider the groups other than the first group.

Suppose at time t Event V occurs, and the kth feasibility constraint is about to be violated. Because at time t, $E^t_{k-1} \geq L^t_{k-1}$, $E^t_k = L^t_k$, and $E^t_{k+1} \geq L^t_{k+1}$, we have $e^t_k \leq l^t_k$ and $e^t_{k+1} \geq l^t_{k+1}$. Now we consider three cases at time 0 before time t:

- Case 1: Suppose initially at time 0, $e_k \geq l_k$. Then there must be an intersection at time $0 \leq t' \leq t$ (i.e. $e^{t'}_k = l^{t'}_k$), hence F event occurs.
- Case 2: Suppose initially at time 0, $e_{k+1} \leq l_{k+1}$. Then there must be an intersection at time $0 \leq t' \leq t$ (i.e. $e^{t'}_{k+1} = l^{t'}_{k+1}$), hence F event occurs.
- Case 3: Suppose initially at time 0, $e_k < l_k$ and $e_{k+1} > l_{k+1}$. Then $E^t_k = E^t_{k-1} + e^t_k < L^t_{k-1} + l^t_k = L^t_k$, which leads to a contradiction indicating that the kth feasibility constraint is not the one that is about to be violated. Therefore, there must exist $k' < k$ such that the k'th feasibility constraint has already been violated at time $t' < t$. Then we move to time t' and the

problem becomes the same except that $k' < k$. By such reduction, the case 3 will be recursively reduced to either case 1, case 2, or the base case (i.e. first group), and turn out that there must be an event to occur earlier than the feasibility conditions is violated.

Therefore we finish the proof.

□

As mentioned in earlier chapters, PCG scheduling algorithm assigns processors to tasks in a greedy manner that the task with largest local remaining execution requirement will get the fastest processor, and the order of assignment of PCG scheduling algorithm is a subset of group order. Therefore, theorem 8 is a generalized theorem of Chen's theorem in theorem 3.

Theorem 9. *For a feasible set of n tasks and m processors, the PGM scheduling algorithm is optimal and feasible for uniform multiprocessors.*

Proof. Since the set of tasks and processors is feasible, it follows the feasibility conditions. Since PGM reschedules at any event, according to theorem 8, the new task set is still feasible in the T-L$_{er}$ plane. Since each T-L$_{er}$ plane is independent, the entire schedule is feasible. Therefore, any feasible tasks set can be scheduled to meet all deadlines using PGM algorithm, and PGM scheduling algorithm is optimal. □

The main drawback of PGM algorithm is that the algorithm would not

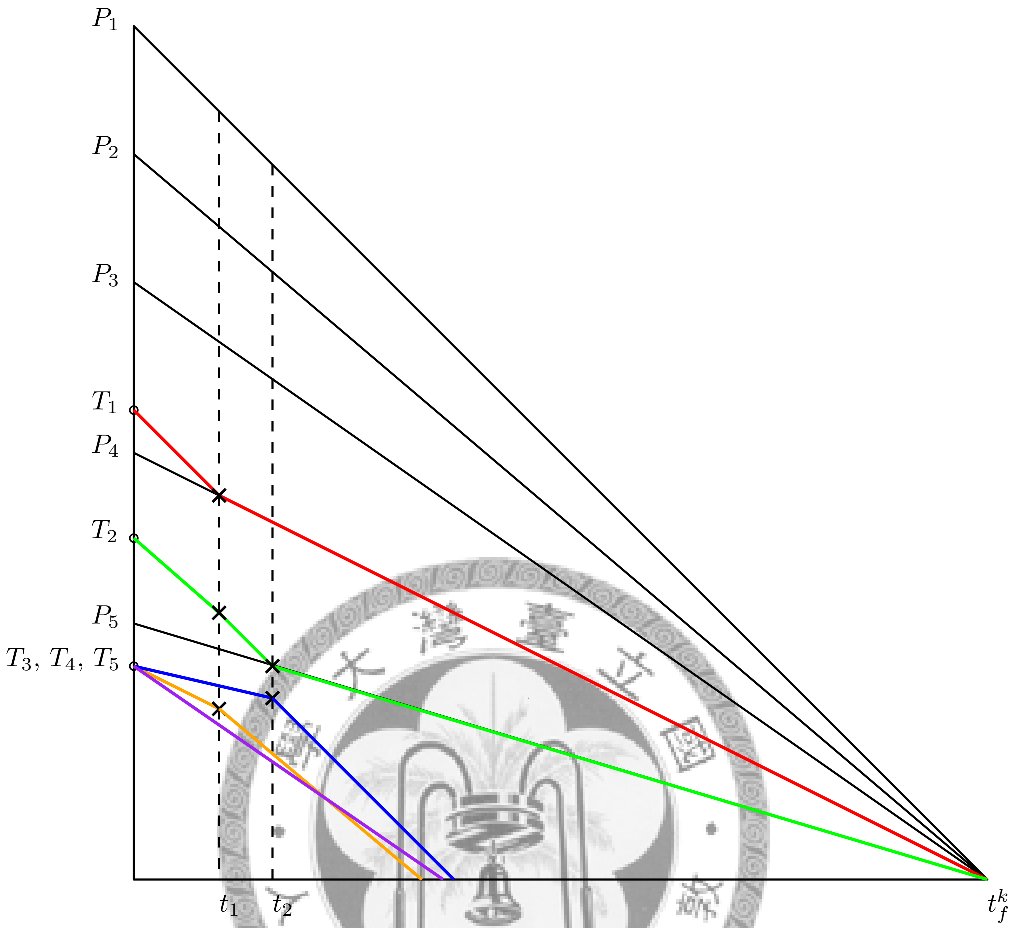

Figure 4.4: Yet Another Example of PGM Scheduling Algorithm

guarantee a constant bound for task migrations while rescheduling. An example is shown in figure 4.4 where initially the feasible set contains five processors P_1, P_2, P_3, P_4, and P_5 with local computing capacity $e_1 = 1$, $e_2 = 0.85$, $e_3 = 0.7$, $e_4 = 0.5$, and $e_5 = 0.3$ respectively and five tasks T_1, T_2, T_3, T_4, and T_5 with local remaining execution requirement $l_1 = 0.55$, $l_2 = 0.4$, and $l_3 = l_4 = l_5 = 0.25$ respectively. Initially, tasks T_1, T_2, T_3, T_4, and T_5 is assigned with processors P_1, P_2, P_3, P_4, and P_5 respectively to satisfy the group order. At time t_1, the execution path of T_1 hits the boundary of P_4, Event F occurs. Therefore, T_1

and P_4 are removed from the feasible set, and the remaining tasks T_2, T_3, T_4, and T_5 are assigned with processors P_1, P_3, P_2, and P_5 respectively to satisfy the group order. At the time instance t_1, we see that it requires at least three tasks migrations as indicated by a cross bar in figure 4.4.

4.6 The Preprocessed Precaution Group Merge Algorithm

Based on the PGM scheduling algorithm, we develop the Preprocessed Precaution Group Merge (PPGM) scheduling algorithm. The only distinction between the PPGM and the PGM algorithms is that we change the way of making group by first preprocessed the tasks and processors sets as shown in algorithm 5.

Unlike the make group algorithm in algorithm 2, the preprocessed make group algorithm would scan through the entire processors set to make the best processors groups to reduce tasks migrations. Rather then simply grouping the faster processors into prior groups, while the preprocessed make group algorithm is making a processor group for a task group, the algorithm would count the number of processors with local computing capacity larger than the boundary processor of the task group, and make the most efficient use of these processors. For example, in figure 4.4, the original make group algorithm 2 generates the processor groups of $\{P_1\}$, $\{P_2\}$, and $\{P_3, P_4, P_5\}$. But in fact, there are three processors (i.e. P_1, P_2, and P_3) with local computing capacity larger than the local remaining execution requirement of T_1, and the most efficient way is

Algorithm 5. *Preprocessed Make Group*

```
      Input   : An independent feasible set $\{\mathcal{T}, \mathcal{P}\}$ of $n$ tasks and $m$
                uniform processors
      Output: Tasks groups $\mathcal{G}$ and processors groups $\mathcal{H}$
      // Initialize
 1.   $i \leftarrow 1$, $j \leftarrow 1$, $k \leftarrow 1$
 2.   $Stack \leftarrow \phi$
      // Scan $\mathcal{T}$ and $\mathcal{P}$ to make group
 3.   while $i < n$ do
 4.     $Stack.push(P_k)$ , $k \leftarrow k + 1$
 5.     $\mathcal{G}_i \leftarrow \phi$ , $\mathcal{H}_i \leftarrow \phi$
 6.     while $l_j > e_{i+1}$ and $j < n$ do
 7.       $\mathcal{G}_i \leftarrow \mathcal{G}_i \cup T_j$
 8.       if $Stack.empty()$ then
 9.         $\mathcal{H}_i \leftarrow \mathcal{H}_i \cup P_k$ , $k \leftarrow k + 1$
10.       else
11.         $\mathcal{H}_i \leftarrow \mathcal{H}_i \cup Stack.pop()$
12.       end if
13.       $j \leftarrow j + 1$
14.     end while
15.     $i \leftarrow i + 1$
16.   end while
```

to assign T_1 with P_3 and T_2 with P_4. Therefore, the preprocessed make group algorithm would instead generate processor groups $\{P_3\}$, $\{P_4\}$, and $\{P_1, P_4, P_5\}$.

The Preprocessed Precaution Group Merge scheduling algorithm is shown in algorithm 6.

Algorithm 6. *Preprocessed Precaution Group Merge*

```
    Input: A set $\mathcal{T}$ of $n$ tasks $\{T_1, T_2, ..., T_n\}$ with utilization $u_1, u_2, ..., u_n$.
           A set $\mathcal{P}$ of $m$ processors $\{P_1, P_2, ..., P_m\}$ with speed $s_1, s_2, ..., s_m$.
 1. repeat
 2.     if B event occurs then
 3.         remove the task with local remaining execution requirement
            equals to 0
 4.         assign another ready task to the idle processor
 5.     else
 6.         separate $\{\mathcal{T}, \mathcal{P}\}$ into independent feasible sets
 7.         for each independent feasible set, make group by algorithm 5
 8.         for each independent feasible set, assign tasks with processors in a
            group order
 9.     end if
10. until any [C|F|B] event occurs
```

As mentioned earlier, the only distinction between PGM and PPGM algorithms is the make group algorithm. By making different processors group, the task schedule would be different among these two scheduling algorithms.

We use the same feasible set in figure 4.4 as an example to demonstrate the execution of PPGM scheduling algorithm. The example is shown in figure 4.5. Initially, algorithm 5 make tasks groups $\{T_1\}$, $\{T_2\}$, and $\{T_3, T_4, T_5\}$, and processor groups $\{P_3\}$, $\{P_4\}$, and $\{P_1, P_2, P_5\}$ respectively. Hence we assign T_1, T_2, T_3, T_4, and T_5 with P_3, P_4, P_1, P_2, and P_5 respectively. While at time t_1, Event F occurs and the local remaining execution requirement of T_1 is equal to the local computing capacity of P_4. Therefore, T_1 and P_4 would be removed from the feasible set, and now T_2 is assigned with idle processor P_3. Again at time t_2, another Event F occurs when the local remaining execution requirement of T_2 is equal to the local computing capacity of P_5. Therefore T_2 and P_5 would be removed from the feasible set, and T_5 is assigned with idle processor P_3. And the we will keep this order until the end of the T-L$_{er}$ plane.

4.7 Proof of Optimality of PPGM Scheduling Algorithm

Theorem 7 and theorem 8 shows that there must be at least one Event C, Event B, or Event F occurs before Event V if we schedule using PGM algorithm. For PPGM scheduling algorithm, the only distinction from PGM scheduling algorithm is the algorithm of making groups. Therefore, as long as we can prove that the tasks and processors groups generated by algorithm 5 would hold the same properties as generated by algorithm 2, we can prove that the PPGM scheduling is optimal.

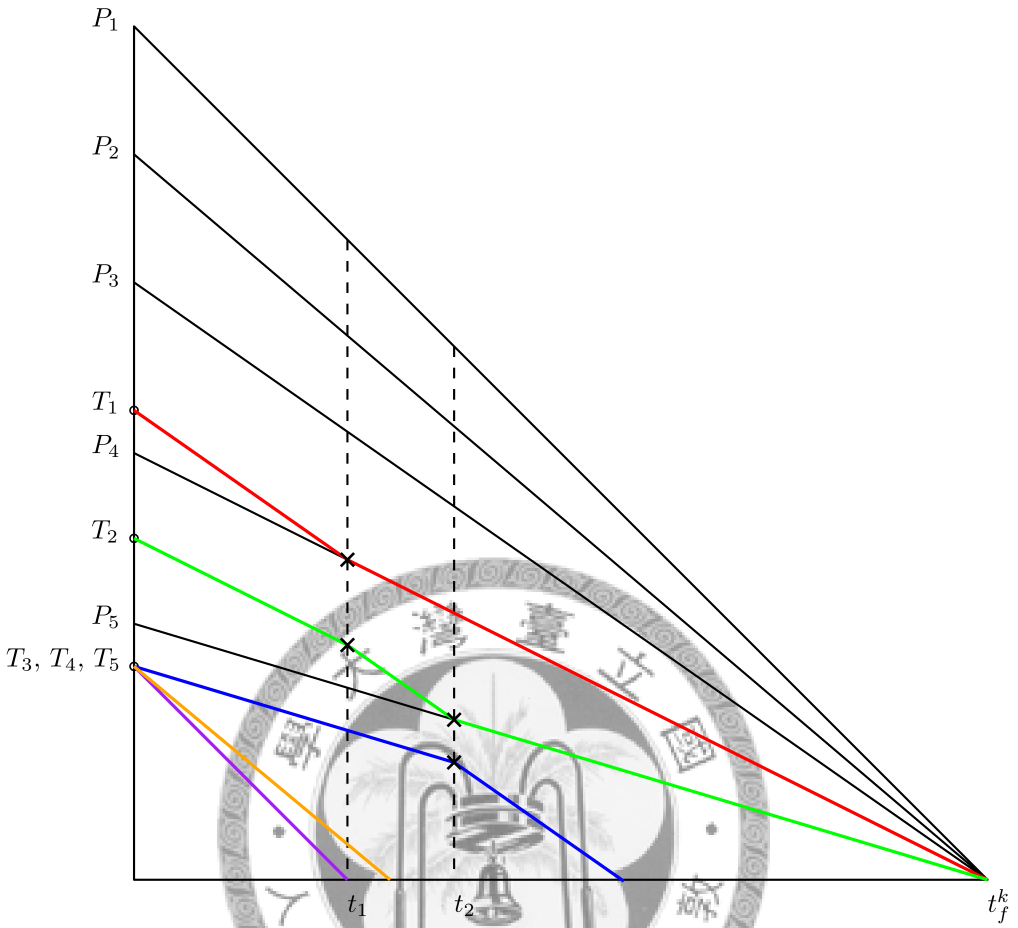

Figure 4.5: Example of PPGM Scheduling Algorithm

Theorem 10. *For a feasible set of n tasks and m processors scheduled using PPGM scheduling algorithm, there must be at least one Event C, Event B, or Event F occurs before Event V.*

Proof. We start the proof first by examining algorithm 5. While generating any tasks group $\mathcal{G}_i = \{T_j | e_i > l_j > e_{i+1}\}$, suppose there are k processors that have been pushed into the stack by algorithm 5. Let $\mathcal{G}_i$ contain g tasks, then according to algorithm 5, there are two cases:

- If $k \geq g$, then g processors would be popped out from the stack and grouped into $\mathcal{H}_i$. In this case, all g processors in $\mathcal{H}_i$ would have local computing capacity larger than the local remaining execution requirement of tasks in $\mathcal{G}_i$.

- If $k < g$, then the algorithm would group k processors in the stack and $g - k$ consecutive processors into $\mathcal{H}_i$. In this case, the first k processors in $\mathcal{H}_i$ would have local computing capacity larger than the local remaining execution requirement of tasks in $\mathcal{G}_i$.

For the first case, the feasibility constraints would hold because all processors in the processors group have local computing capacity larger than the local remaining execution requirement of tasks in the tasks group. For the second case, it is obvious that the processors in the processor group generated by algorithm 5 is always faster than the processors in the processor group generated by algorithm 2. Therefore, for both cases, the feasibility constraints would hold for each tasks group and processors group. Therefore, according to theorem 8, the initial conditions are the same. Hence we concluded that for any feasible set of n tasks and m processors scheduled using PPGM scheduling algorithm, there must be at least one Event C, Event B, or Event F occurs before Event V.

$\square$

Theorem 11. *For a feasible set of n tasks and m processors, the PPGM* *scheduling algorithm is optimal and feasible for uniform multiprocessors.*

Proof. Since the set of tasks and processors is feasible, it follows the feasibility conditions. Since PPGM reschedules at any event, according to theorem 8, the new task set is still feasible in the T-L$_{er}$ plane. Since each T-L$_{er}$ plane is independent, the entire schedule is feasible. Therefore, any feasible tasks set can be scheduled to meet all deadlines using PPGM algorithm, and PPGM scheduling algorithm is optimal. □

Theorem 12. *For a set of n independent non-periodic tasks and m uniform processors, the PPGM scheduling algorithm is optimal.*

Proof. Non-periodic tasks are characterized by their execution requirements. Therefore, similar to the definitions in chapter 2, we can define a non-periodic task $T_i = (c_i, \infty)$ by simply changing the period to infinity.

Early works [7], [10], [9] showed that the optimal finish time ω of scheduling a set of n independent tasks with execution requirement $c_1 \geq c_2 \geq ... \geq c_n$ on m uniform processors with speed $s_1 \geq s_2 \geq ... \geq s_m$ would be

$$\omega = \max\left(\max_{1\leq i\leq m}\left(\frac{C_i}{S_i}\right), \frac{C_n}{S_m}\right), \tag{4.10}$$

where $C_i = \sum_{j=1}^{i} c_j$ and $S_i = \sum_{j=1}^{i} s_i$.

We schedule the set by first construct a T-L$_{er}$ plane with corresponding time interval $[0, t_f) = [0, \omega)$. That is, if we can schedule the set in the T-L$_{er}$

plane, we can finish the schedule in optimal finish time. Then by definition, for each non-periodic task T_i, the utilization can be calculated by $u_i = c_i/\omega$.

According to PPGM algorithm, for a set of n tasks and m uniform processors, if the feasibility constraint would hold at the beginning of the T-L$_{er}$ plane, the set would be schedulable by PPGM scheduling algorithm. Therefore, for all $1 \leq k \leq m$, the kth feasibility constraint of the problem becomes

$$\begin{aligned} U_k &= \sum_{i=1}^{k} u_i = \frac{1}{\omega} \times \sum_{i=1}^{k} c_i \\ &\leq \frac{\sum_{i=1}^{k} s_i}{\sum_{i=1}^{k} c_i} \times \sum_{i=1}^{k} c_i \\ &\leq \sum_{i=1}^{k} s_i = S_k. \end{aligned} \tag{4.11}$$

Similarly, the mth feasibility constraint becomes

$$\begin{aligned} U_n &= \frac{1}{\omega} \times C_n \leq \frac{S_m}{C_n} \times C_n \\ &\leq S_m. \end{aligned} \tag{4.12}$$

Equation 4.11 and equation 4.12 show that the kth feasibility constraint would hold for all $1 \leq k \leq m$. Therefore, PPGM algorithm is able to schedule the set of n non-periodic tasks on m uniform processors in time interval $[0, \omega)$. Hence PPGM scheduling algorithm is also optimal for scheduling non-periodic tasks on uniform multiprocessors.

$\square$

Chapter 5

Complexity Analysis

5.1 Analysis of RD Scheduling Algorithm

The RD scheduling algorithm would assign tasks with processors in a random order and reschedule at the occurrence of any Event B or Event V. Once Event B occurs, the algorithm simply removes the task with local remaining execution requirement equals to 0 and assign another ready task with the idle processor, which is obvious an $O(1)$ manner for time complexity. But if Event V occurs, the algorithm will re-separate the set of tasks and processors into independent feasible sets and re-assign processors to tasks in a random order. According to algorithm 7, once Event V occurs, both the time complexity and the complexity of task migrations would be $O(n)$ for one rescheduling.

The RD scheduling algorithm is hard to implement because the algorithm detects Event V as the time instance for rescheduling. Event V is hard to detect because the execution paths of tasks would intersect with each others at any time, and the rank of local remaining execution requirements would change.

Algorithm 7. *Separate Feasible Set*

```
    Input  : A set $\mathcal{T}$ of $n$ tasks $\{T_1, T_2, ..., T_n\}$. A set $\mathcal{P}$ of $m$ processors
             $\{P_1, P_2, ..., P_m\}$.
    Output: An independent feasible set or empty set $\phi$ on the end

    // Scan for $s_j = u_k$
 1. $j \leftarrow 1$, $k \leftarrow 1$
 2. while $j \leq m$ and $k \leq n$ do
 3.     if $s_j = u_k$ then
 4.         return $\{\{T_k\}, \{P_j\}\}$
 5.     else if $s_j < u_k$ then
 6.         $k \leftarrow k + 1$
 7.     else
 8.         $j \leftarrow j + 1$
 9.     end if
10. end while
    // Scan for $s_j = u_k$
11. $\mathcal{Q} \leftarrow \phi$, $\mathcal{R} \leftarrow \phi$
12. for $k = 1$ to $m$ do
13.     $\mathcal{Q} \leftarrow \mathcal{Q} \cup T_k$, $\mathcal{R} \leftarrow \mathcal{R} \cup P_k$
14.     if $S_k = U_k$ then
15.         return $\{\mathcal{Q}, \mathcal{R}\}$
16.     end if
17. end for
18. return $\phi$
```

Therefore, only if we can simulate the behaviors of all tasks, we are unable to decide L_k^t because we can not decide the local remaining execution requirements of all tasks.

5.2 Analysis of PGM Scheduling Algorithm

Unlike the RD algorithm, the PGM scheduling algorithm detects the first occurrence of Event B, Event C, or Event F for rescheduling. By algorithm 8, we see that the time complexity decreases significantly to $O(1)$ for each task, and hence $O(n)$ for the entire task set.

While rescheduling, since the algorithm would assign tasks with processors in a group order, which is a random order in the same task group, the algorithm would require at most $O(m)$ task migrations to move tasks and processors among different groups. Therefore, while rescheduling, the algorithm has an $O(n)$ bound for time complexity and an $O(m)$ bound for task migration.

If we applied the PGM algorithm to an identical multiprocessors platform, the complexity decreases significantly. Because there the computing capacity are the same for all processors in identical multiprocessors platform, there is only one task group and the rescheduling would only cost $O(1)$ for task migration.

5.3 Analysis of PPGM Scheduling Algorithm

The PPGM scheduling algorithm is similar to PGM but making tasks and processors groups by preprocessed make group algorithm. The preprocessed make group algorithm would divide the processors set into a structure that for any task group $\mathcal{G}_i$, the boundary processors P_i and P_{i+1} would be grouped into processors groups $\mathcal{H}_{i-1}$, $\mathcal{H}_i$, or $\mathcal{H}_{i+1}$. Therefore, whenever an event occurs, the removal of task and processor would not lead to a change to the structure of the tasks groups and processors groups. Therefore, once a processor is removed, we can thought it as a merge between neighboring tasks groups and processors groups. Since the structure of the tasks groups and processors groups is not changed, the removal can be simply done by a context switch, which leads to at most 2 task migrations. Therefore, the PPGM scheduling algorithm would cause only $O(1)$ task migrations on each rescheduling, and at most $2(m-1)$ task migrations in a T-L$_{er}$ plane. Also, because there are only one context switch on each rescheduling, the overhead of calculation for the next event becomes $O(1)$. We can simply maintain a heap structure sorted by time for all events to get the first occurrence of any events. Since it takes $O(\lg n)$ to complete an insertion to a heap, the time complexity for PPGM algorithm becomes $O(\lg n)$ on each rescheduling.

Algorithm 8. *Next Event*

Input : A task T_i, a processor P_j that is assigned to T_i, two processors P_a and P_b that represents the upper and lower boundaries for T_i.

Output: Occurrence time, type, and associated task and processor of the next event

```
    // Scan for Event B
1.  $t_B \leftarrow \frac{l_i}{s_j}$
    // Scan for Event F
2.  if $s_j > s_b$ then
3.      $t_F \leftarrow \frac{l_i - e_b}{s_j - s_b}$
4.      $P \leftarrow P_b$
5.  else
6.      $t_F \leftarrow \frac{e_a - l_i}{s_a - s_j}$
7.      $P \leftarrow P_a$
8.  end if
9.  if $t_F > t_B$ then
10.     return $\{t_B, \text{"Event B"}, T_i, P\}$
11. else
12.     return $\{t_F, \text{"Event F"}, T_i, P\}$
13. end if
```

Chapter 6

Conclusions

Although in 2008, Chen presented the PCG scheduling algorithm, which is the first optimal on-line dynamic-priority scheduling algorithm for uniform multiprocessors systems, the cost of the algorithm is high. In this thesis, first, we introduce the idea of independent feasible sets and the idea of group order. Furthermore, based on the ideas and the T-L_{er} plane model for uniform multiprocessors platform, we extend the Precaution Cut Greedy algorithm, and present three optimal on-line dynamic-priority scheduling algorithms for uniform multiprocessors systems with lower time complexities and less task migration costs.

The first algorithm presented is the RD scheduling algorithm. The RD scheduling algorithm is optimal, but is hard to implement. The RD algorithm could be thought as an one-directional proof of Funk's theorem that once the set of periodic tasks and uniform processors satisfies the feasibility condition, the set is schedulable to meet all deadlines.

The second algorithm would be the PGM scheduling algorithm. The algorithm is similar to PCG scheduling algorithm but assigns tasks with processors in a group order, which generalize the concept of PCG algorithm. The algorithm is optimal, with $O(n)$ time complexity and $O(m)$ task migrations on each rescheduling for an uniform multiprocessors platform. But if the algorithm is applied to an identical multiprocessors platform, the performance would raise significantly to $O(n)$ time complexity and $O(1)$ task migrations on each rescheduling.

Finally, we present the PPGM scheduling algorithm, which is based on the PGM algorithm but with a few modifications. The algorithm is also optimal, and gives an $O(\lg n)$ time complexity but $O(1)$ task migrations on each rescheduling for an uniform multiprocessors platform. Comparing to others, currently the PPGM algorithm is an on-line dynamic-priority scheduling algorithm with the least task migration cost among all algorithms.

In addition to periodic tasks, the PPGM scheduling algorithm is also optimal for scheduling non-periodic tasks. The algorithm would generate a schedule for n non-periodic tasks on m uniform processors with optimal finish time and at most $2(m-1)$ task migrations.

Because of the simplicity and the low task migration cost of the PPGM scheduling algorithm, we believe that the PPGM algorithm could be applied to many asymmetric multicore platforms that are similar to uniform multiprocessor platform.

Bibliography

[1] S.K. Baruah, N.K. Cohen, C.G. Plaxton, and D.A. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15(6):600–625, June 1996.

[2] Shih-Ying Chen and Chih-Wen Hsueh. Optimal dynamic-priority real-time scheduling algorithms for uniform multiprocessors. In *Proc. IEEE Real-Time Systems Symposium*, pages 147–156, Barcelona, Spain, December 2008.

[3] Hyeonjoong Cho, Binoy Ravindran, and E. Douglas Jensen. An optimal real-time scheduling algorithm for multiprocessors. *RTSS*, pages 101–110, Oct. 2006.

[4] M.L. Dertouzos and A.K Mok. Multiprocessor online scheduling of hard-real-time tasks. *IEEE Transcation on Software Engineering*, 15(12):1497–1506, Dec. 1989.

[5] S. Funk, J. Goossens, and S. Baruah. On-line scheduling on uniform multiprocessors. *IEEE Real-Time Systems Symposium*, pages 183–192, Dec.

2001.

[6] K.S. Hong and J.Y.-T. Leung. On-line scheduling of real-time tasks. *RTSS*, pages 244–250, Dec. 1988.

[7] Edward C. Horvath, Shui Lam, and Ravi Sethi. A level algorithm for preemptive scheduling. *Journal of the ACM*, 24(1):32–43, January 1977.

[8] C. L. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 10(1):46–61, 1973.

[9] Jane W. S. Liu and C. L. Liu. Performance analysis of heterogeneous multiprocessor computing systems. *Computer Architectures and Networks*, 1974.

[10] Jane W. S. Liu and Ai-Tsung Yang. Optimal scheduling of independent tasks on heterogeneous computing systems. *ACM Annual Conference*, 1:38–45, 1974.

[11] R.R. Muntz and E.G. Coffman. Optimal preemptive scheduling on two-processor systems. *IEEE Transactions on Computers*, (11):1014–1020, Nov. 1969.