

國立臺灣大學電機資訊學院資訊工程學研究所

碩士論文

Graduate Institute of Computer Science and Information Engineering

College of Electrical Engineering and Computer Science

National Taiwan University

Master Thesis

制式多處理器下的最佳動態優先即時排程演算法

Optimal Dynamic-priority Real-Time Scheduling Algorithms
for Uniform Multiprocessors



陳士穎

Chih-Ying Chen


指導教授：薛智文 博士

Advisor: Chih-Wen Hsueh, Ph.D.

中華民國 97 年 7 月

July, 2008

Optimal Dynamic-priority Real-Time Scheduling Algorithms
for Uniform Multiprocessors

The seal of National Taiwan University is a circular emblem. It features a central figure, likely a scholar or historical figure, surrounded by Chinese characters. The outer ring of the seal contains the university's name in Chinese: '國立台灣大學' at the top and '愛·學勵' at the bottom.

By
Shih-Ying Chen
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 2008

制式多處理器下的 最佳動態優先即時排程演算法

指導教授：薛智文 博士 研究生：陳士穎

國立台灣大學資訊工程學研究所



週期性任務在多處理器上的排程是硬性即時環境下其中一個最受矚目的問題，而其中包括令人熟知的制式多處理器排程，在制式多處理器環境下，每個任務在一個處理器上的執行時間是等比例於該處理器的運算能力，在以往的成果中，制式多處理器的線上排程只有合適的近似解；在這篇論文，隨著任務可以遷移到不同的處理器，我們首度提出 $T-L_{er}$ 平面這個嶄新的模型，用來描述任務和處理器的行為，並在 $T-L_{er}$ 平面上提出兩種最佳演算法以在制式多處理器上排程動態優先即時任務，為了讓演算法更符合真實並減少本文切換，我們提出了複雜度在多項式時間以內的演算法來保證一個 $T-L_{er}$ 平面裡重新排程的次數，由於任務遷移在 SOC 多核心平台下較為容易達成，我們的結果也許可以應用到非對稱的多核心平台。

關鍵字：即時，制式，多處理器，最佳，線上，演算法，預防，貪婪，切

Abstract

In hard-real-time environment, scheduling periodic tasks upon multiprocessors is one of the most popular problems where uniform multiprocessor scheduling is a well-known one. In uniform multiprocessor scheduling, execution time of each task in one processor is proportional to the computing capacity of this processor. From previous works, there are only approximate feasible solutions for on-line scheduling on uniform multiprocessors. In this thesis, with task migration, we first present a novel model called $T-L_{er}$ plane for uniform multiprocessors to describe the behavior of tasks and processors, and two optimal algorithms based on $T-L_{er}$ plane to schedule dynamic-priority real-time tasks on uniform multiprocessors. To make it practical and reduce context switches, we also present a polynomial-time algorithm to bound the times of rescheduling or task migration in a $T-L_{er}$ plane and give an experimental evaluation for it. Since task migration is easier in SOC multicore processors, our result might be applicable and adapted to many asymmetric multicore platforms.

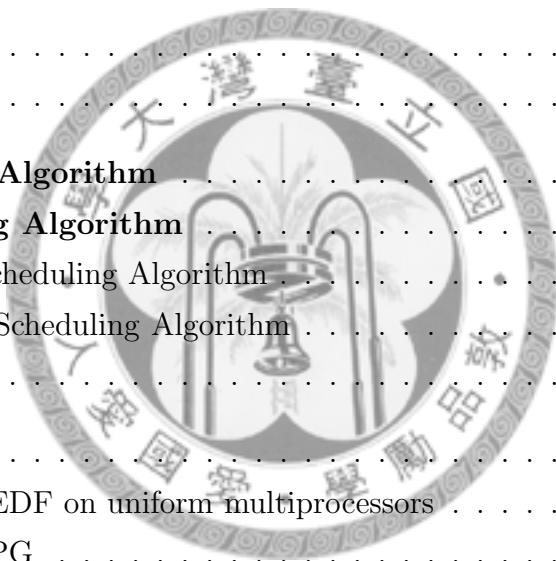
Keyword: real-time, uniform, multiprocessors, optimal, on-line, algorithm, precaution, greedy, cut.

Contents

1	Introduction	1
2	Definitions, Assumptions, and Feasibility Condition for Uniform Multiprocessors	12
2.1	Definitions and Assumptions	12
2.2	Feasibility Condition for Uniform Multiprocessors	14
3	Model and $T-L_{er}$ Plane	17
3.1	P-fair and Fluid Schedule	17
3.2	$T-L_{er}$ Planes	18
3.3	Definitions in One $T-L_{er}$ Plane	21
3.4	$T-L_{er}$ Plane vs T-L Plane	21
4	Optimal Scheduling Algorithms for Uniform Multiprocessors	25
4.1	Precaution Greedy (PG) Scheduling Algorithm	27
4.2	Precaution Cut Greedy (PCG) Scheduling Algorithm	27
4.3	Proof of Optimality	29
5	Experimental Evaluation	38
5.1	Input Generator	38
5.2	PG and PCG Performance Evaluation	38
6	Conclusions	45
	Bibliography	46

List of Figures

1.1	L-C plane	4
1.2	T-L plane	7
1.3	Examples of EDF and PCG	8
2.1	T- L_{er} plane	16
3.1	T- L_{er} Planes	19
3.2	k^{th} T- L_{er} Plane	22
4.1	PG Scheduling Algorithm	26
4.2	PCG Scheduling Algorithm	28
4.3	Example of PG Scheduling Algorithm	30
4.4	Example of PCG Scheduling Algorithm	31
4.5	Task Order	32
5.1	Input Generator	39
5.2	Schedulability of EDF on uniform multiprocessors	40
5.3	Schedulability of PG	40
5.4	Schedulability of PCG	41
5.5	Performance Analysis of PG while Comparing to EDF	42
5.6	Performance Analysis of PCG while Comparing to EDF	42
5.7	The number of increasing cases between PCG and PG	43
5.8	Performance Analysis between PCG and PG	43



Chapter 1

Introduction

Multi-core processor represents a major evolution in computing technology recently [1]. Many operating systems are now benefiting from multi-core processors. It offers cost-effective technology rather than single-core processor and give users the ability to keep working while running tasks in the background. Multi-threaded applications also benefit from it and result in performance increases. It has lots of advantages than single-core processors, so many enterprises support it and develop many technologies.

Asymmetric multicore platform (AMP), which consists of lots of processing units on one or several chips, is capable of executing the same instructions on each processing unit with different performance levels. AMP may ease the transition for software developers from platforms containing a few large, powerful cores to platforms containing tens or hundreds of smaller, simpler cores where parallelism exploiting will be required in order to improve performance. Thus scheduling on AMP may be an important problem for current enterprises,

where many people tried to adapt real-time tasks on this platform and to enhance the total utilization, reducing number of context switching, and improve performance. Calandrino et al. [5] focused on the soft real-time scheduling on AMPs. They implemented their work on both schedsim (Linux scheduler simulator) and Linux kernel, supported periodic real-time tasks, and provided good performance for non-real-time tasks in the presence of a real-time workload.

To schedule multi-core processor, we consider the architecture in hard-real-time environment – multiprocessors, similar to multi-core processor. If we could schedule on multiprocessors, the scheduling algorithm might be adapted on multi-core processors. There are three major kinds of multiprocessor platforms as follows.

Identical parallel machines: All the processors are identical and contain the same computing power. Baruah [2] had presented a strong fairness scheduling algorithm called P-fair, where each task is scheduled resources in proportion to its weight and it is an optimal scheduling for identical multiprocessors. Due to its strong fairness, and quantum-based, it will reschedule many times and have lots of context switches. Holman and Anderson [11] also discussed about P-fair and presented the idea of fluid schedule, where each task executes at a constant rate, ideally but impractical because there will be too much rescheduling. In 1969, Muntz and Coffman [17] had presented a level algorithm for identical multiprocessors. They schedule the tasks in the highest

level first until all the tasks finish their jobs. The tasks in the same level will be scheduled on some processors and evenly share the computing power of the processors. Dertouzos and Mok [8] presented *Laxity and Computation plane* (L-C plane) as shown in Figure 1.1. The laxity of a task is a measurement of its urgency, represented on the x-axis. The computation is the remaining execution time, represented on the y-axis. Clearly, the task, T_1 , with zero laxity must be executed immediately and without interruption.

Cho et al. [7] based on P-fair and L-C plane, extended their idea, created *Time and Local Execution Time Planes* (or T-L planes) with time represented on the x-axis and execution time represented on the y-axis, a token represents task status, and token movement over time in T-L plane forms a line, as shown in Figure 1.2. T-L plane could show the execution behavior of tasks and make it possible for us to envision that entire scheduling over time is just the repetition of T-L planes, so that optimal scheduling in a single T-L plane implies optimal scheduling over all. They provided *Largest Local Remaining Execution Time First* (or LLREF) scheduling algorithm based on T-L plane and proved it is an optimal on-line scheduling algorithm for identical multiprocessors. It reduces the times of rescheduling, defines their own urgency task, and only invokes the scheduler before the emergency of each task. Thus, the performance is optimized.

Uniform parallel machines: Each processor is characterized by its own

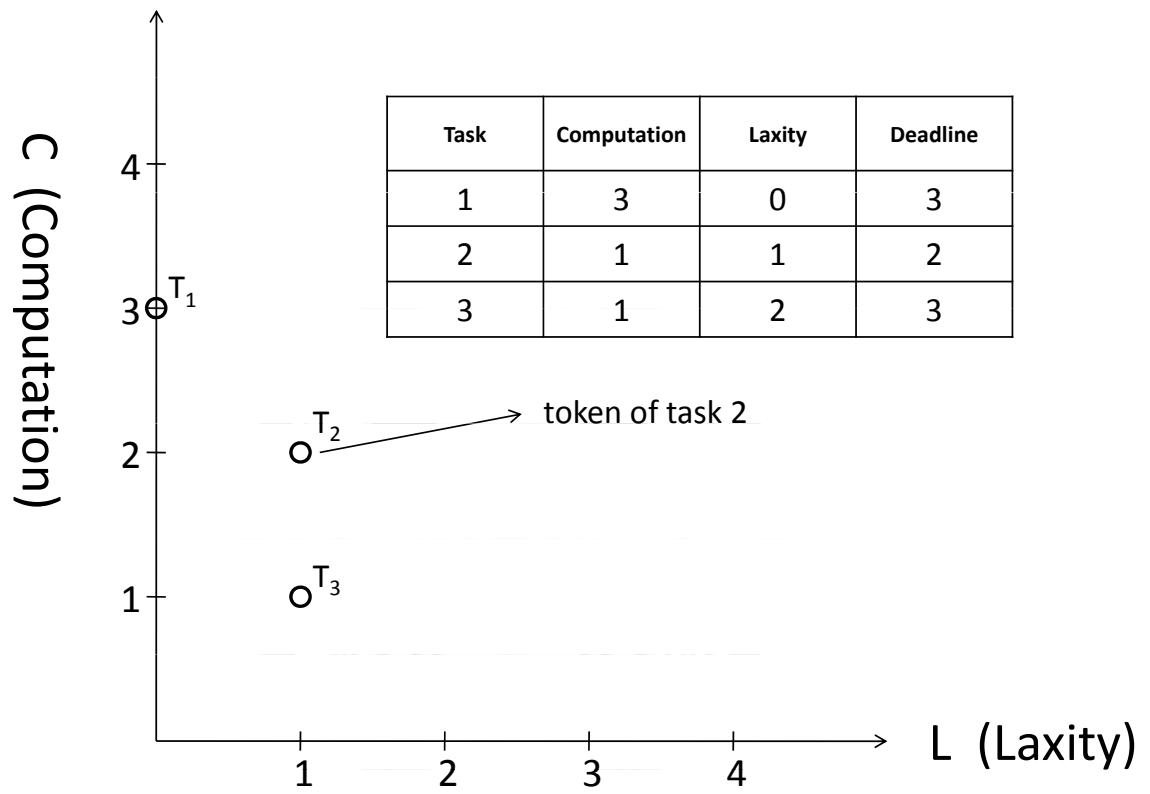


Figure 1.1: L-C plane

computing capacity, with the interpretation that a job that executes on a processor of computing capacity s for t time units completes $s \times t$ units of execution. Each task can migrate to all processors, execution requirement of task is a constant, and execution time of task upon processors is proportional to the computing capacity of each one. Horvath et al. [12] had extended the idea from the level algorithm for identical multiprocessors to uniform multiprocessors. They scheduled non-periodic tasks by level algorithm, used the idea of shared schedule, and proved the minimal length schedule (or makespan problem). Gonzalez and Sahni [3] had given the off-line optimal $O(m \log m)$ scheduling algorithm with the sorted time and no more than $(m - 1)$ preemptions, where m is the number of processors.

There are many works on on-line scheduling upon uniform multiprocessors. Hochbaum and Shmoys [10] presented a polynomial approximation scheme, trying to let the last job finish as quickly as possible. In static priority on-line scheduling, based on RM (Rate Monotonic) scheduling algorithm [15], Baruah and Goossens [4] tried to adapt RM scheduling algorithm upon uniform multiprocessors, they based on the idea of greedy scheduling algorithm, gave higher-priority jobs upon faster processors, and got an RM-feasibility test for this problem, which is the first nontrivial feasibility test for RM scheduling algorithm upon uniform multiprocessors. In dynamic priority on-line scheduling, based on EDF (Earliest Deadline First) scheduling algorithm [15], Funk et al. [9] presented a feasibility condition for periodic task upon uniform multiprocessors, all

the sets of tasks and processors need to satisfy this condition to be schedulable. We call it FG condition in this thesis. They also presented an efficient test to determine whether any instance of hard-real-time jobs known to be feasible on a particular platform can be scheduled by EDF to meet all deadlines upon another platform. They derived a sufficient condition to verify a periodic task set to successfully meet all deadlines when scheduled using EDF, and got a EDF-feasible condition. However, no quick algorithm could schedule any feasible task set under the FG condition. From the definition of AMP and uniform multiprocessor [5], we know the scheduling algorithm for uniform multiprocessor could be adapted to AMP easily. Thus, if we could find a good scheduling algorithm for uniform multiprocessors, this algorithm might also have high performance on AMP to achieve better parallelism.

Unrelated parallel machines: There is an execution rate $r_{i,j}$ associated with each task-processor ordered pair (T_i, P_j) , with the interpretation that task T_i completes $(r_{i,j} \times t)$ units of execution by execution on processor P_j for t time units. To minimize the makespan of this problem, Lenstra [14] presented a polynomial algorithm with no longer than twice of optimum and a polynomial approximation scheme, Jansen and Porkolab [13] also give fully approximation algorithm for this problem. Srivastava [18] presented a tabu search based heuristic for minimizing makespan that can provide good quality solution for practical size problem with a reasonable amount of computational time.

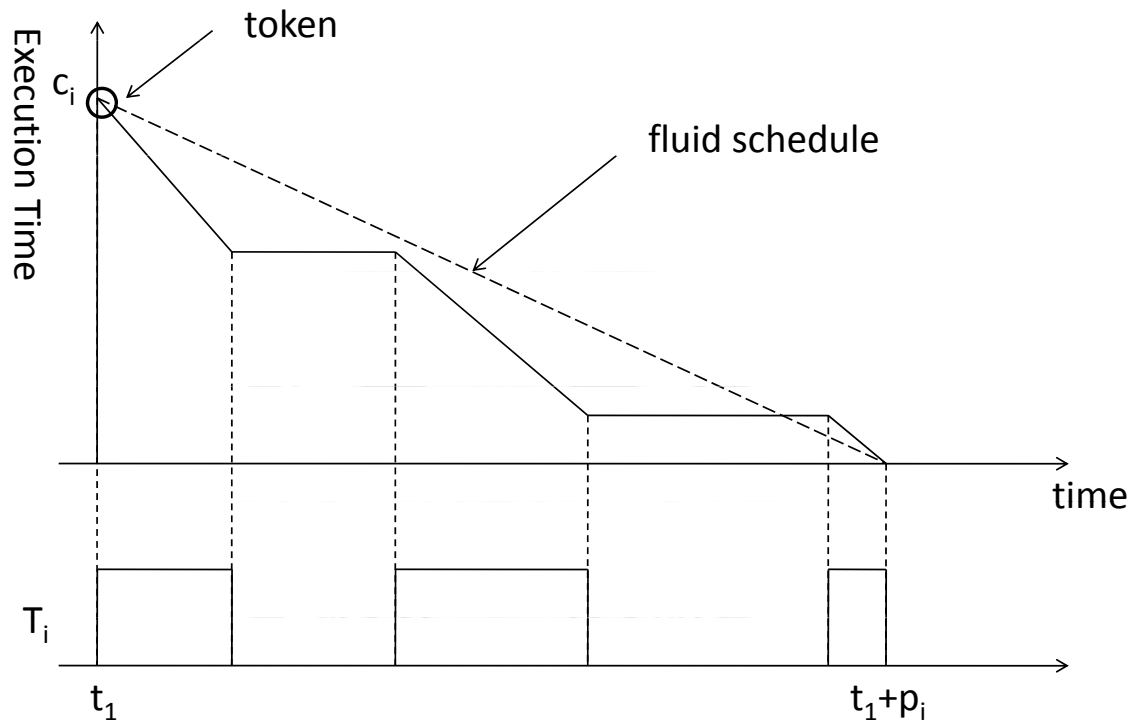
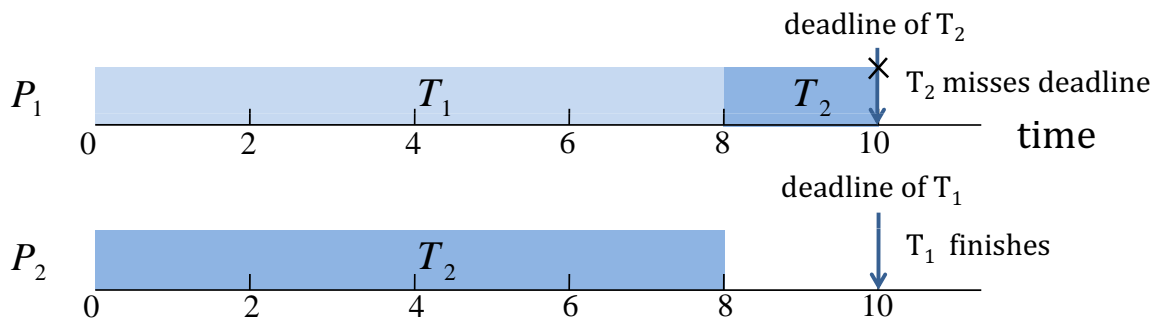
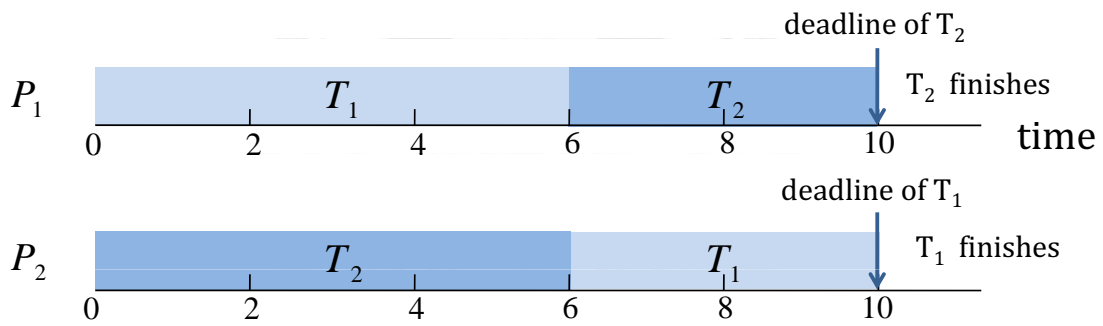


Figure 1.2: T-L plane



(a) Schedule by EDF



(b) Schedule by PCG

Figure 1.3: Examples of EDF and PCG

In this thesis, we focus on dynamic-priority on-line scheduling for uniform multiprocessors. From introduction above, there were only approximately feasible solution, EDF-feasible, and there is not an optimal dynamic-priority on-line scheduling algorithm in terms of feasibility condition. Figure 1.3(a) shows an example for EDF schedule in uniform multiprocessors, task $T_1 = (8, 10)$, (*execution_time, deadline*), $T_2 = (7, 10)$, processor P_1 with *speed* = 1, P_2 with *speed* = 0.5, where P_2 runs 50 percentage as fast as P_1 . Based on the FG condition, the set of tasks and processors is feasible, but not using EDF-feasible. At time 0, the deadline of T_1 is equal to T_2 , we assign T_1 to P_1 and T_2 to P_2 . At time 8, T_1 finishes its job, based on EDF, we assign T_2 to P_1 . At time 10, the remaining execution requirement of T_2 is 1, T_2 misses its deadline. This example shows that even the set of tasks and processors is feasible, it might miss deadline by EDF-feasible scheduling algorithm. However, the task set is schedulable as shown in Figure 1.3(b) using our optimal scheduling algorithm, the details will be discussed later.

Therefore, we would like to derive an optimal scheduling algorithm such that for any task set satisfying the FG condition, it is always schedulable. To derive an optimal scheduling algorithm, based on the concept of T-L plane for identical multiprocessors, we create *Time* and *Local Execution Requirement* plane (or T- L_{er} plane) for uniform multiprocessors. T- L_{er} plane can describe the execution behavior of each task on uniform multiprocessors. It is different from processor budget that processor budget consider about the work that processor

can provide [16], but $T-L_{er}$ plane is focus on the remaining work of each task. To discuss the characteristics of uniform multiprocessors, we create a concept of *processor boundary* in $T-L_{er}$ plane. It is the critical boundary of processors, where rescheduling might be needed when some tokens meet the boundaries. With proper "precaution" ahead to do the rescheduling, that is before current task status might violate the FG condition, we can find the optimal scheduling algorithm.

Based on $T-L_{er}$ plane, we derive two optimal on-line scheduling algorithms, the first one is **P**recaution **G**reedy (or PG) scheduling algorithm. To achieve optimum, we always reschedule at some events precautionously before the FG condition is violated. This is because it might not be schedulable if we do not reschedule until the most urgent event occurs as the LLREF does. When we reschedule, we always assign greedily the task with the largest local remaining execution requirement the fastest processor. However, the number of rescheduling might be indefinitely.

Based on PG, the second one is **P**recaution **C**ut **G**reedy (PCG) scheduling algorithm. The difference between PCG and PG is when rescheduling, local remaining execution requirement of some task will be equal to the computing capacity of some processor. We can assign the task to the processor all the way to the end of $T-L_{er}$ plane without affecting the schedulability. PCG dramatically decrease the times of rescheduling with a upper bound of $n + 1$ in one

T- L_{er} plane, where n is the number of tasks.

Our contributions are as follows:

- We introduce a novel abstraction for reasoning about execution behavior of tasks and processors on uniform multiprocessors, called T- L_{er} plane.
- We first present an optimal on-line scheduling algorithm for uniform multiprocessors called *Precaution Greedy* scheduling algorithm. There were only approximate feasible solutions for this problem.
- We present an optimal on-line scheduling algorithm for uniform multiprocessors called *Precaution Cut Greedy* scheduling algorithm, where the times of rescheduling in one T- L_{er} plane is bounded within n .

The rest of this thesis is organized as follows. The next chapter introduces the problem definitions and assumptions, and the feasibility condition on uniform multiprocessors. In chapter 3, we present a new model called T- L_{er} plane for scheduling on uniform multiprocessors and describe the difference from T-L plane. In chapter 4, we present the PG and PCG scheduling algorithms and prove the optimality. We also derive the upper-bound for PCG algorithm in one T- L_{er} plane. In chapter 5, we analyze the performance of our scheduling algorithms. This thesis is concluded in chapter 6.

Chapter 2

Definitions, Assumptions, and Feasibility Condition for Uniform Multiprocessors

Before we discuss the details of PG and PCG scheduling algorithms on uniform multiprocessors, we introduce our task model.

2.1 Definitions and Assumptions

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

For system environment, we have the following definitions:

- On-line scheduling - makes scheduling decisions at each time-instant based on the characteristics of tasks.
- Static scheduling - operates on a fixed set of tasks and produces a single schedule fixed at all time.

- Dynamic-priority scheduling - executes tasks with arbitrarily priorities at run-time
- Preemptive scheduling - allows task preemption at any time.

A processor P_i is characterized by speed or computing capacity, s_i , and computing capacity in a period of time t . W.l.o.g., we assume s are indexed in a decreasing manner: $s_1 = 1, s_i > s_{i+1}, 1 \leq i < n$, all the values are proportional to speed, positive, and larger than zero. $S_i = \sum_{k=1}^i s_k$ represents the sum of speed from processor 1 to processor i . W.l.o.g., we assume $m = n$, that is the number of tasks is equal to the number of processors, because when $m > n$, the slower processors will never be used, and when $m < n$, we can add dummy processors with speed equal to 0.

A task $T_i = (c_i, p_i)$ is characterized by an execution requirement c_i and a period p_i - all the tasks is periodic and generate a job at each integer multiple of p_i and each has an execution requirement of c_i execution units and must complete by a deadline equal to the next integer multiple of p_i . We define $u_i = c_i/p_i$ to represent the utilization of task i and $C_i = \sum_{k=1}^i c_k$ to represent the sum of execution requirement from task 1 to task i .

For each task, we have the following assumptions.

- Periodic and the deadline is equal to period.
- Independent task - tasks do not share resources or have any precedences.

- Full migration - tasks are allowed to arbitrarily migrate across processors during their execution.

As Funk and Goossens [9] presented, we define the work-conserving scheduling algorithm in the following conditions:

- No processor is idled while there are active jobs awaiting execution
- If at some instant there are fewer than n active tasks awaiting execution, then the active tasks are executed upon the fastest processors.

2.2 Feasibility Condition for Uniform Multiprocessors

Recalled there are many works on uniform multiprocessors, Horvath et al. [12] presented the minimal length schedule for set of tasks and processors. Funk et al. [9] had based on them and presented the feasibility condition upon uniform multiprocessors. We introduce their theorems here and use the FG condition to prove PG and PCG are optimal scheduling algorithms.

Theorem 1 (Horvath et al. [12]) *The level algorithm constructs a minimal length schedule for the set of independent tasks τ with service requirements $c_1 \geq c_2 \geq \dots \geq c_n$ on the processing system $\pi = (s_1 \geq s_2 \geq \dots \geq s_m)$, $m \leq n$. The schedule length is given by*

$$\max \left(\max_{1 \leq i \leq m} \left(\frac{C_i}{S_i} \right), \frac{C_n}{S_m} \right) \quad (2.1)$$

Theorem 2 (Funk et al. [9]) Consider a set $\tau = \{T_1, \dots, T_n\}$ of periodic tasks indexed according to non-increasing utilization (i.e., $u_i \geq u_{i+1}$ for all i , $1 \leq i < n$, where $u_i = \frac{e_i}{p_i}$). Let $U_i = \sum_{j=1}^i u_j$ for all i , $1 \leq i \leq n$. Let π denote a system of $m \leq n$ uniform processors with speeds s_1, s_2, \dots, s_m , $s_i \geq s_{i+1}$ for all i , $1 \leq i < m$. Periodic tasks system τ can be scheduled to meet all deadlines on uniform multiprocessor platform π if and only if the following constraints hold:

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

$$U_k \leq S_k, \text{ for all } k = 1, \dots, m. \tag{2.3}$$



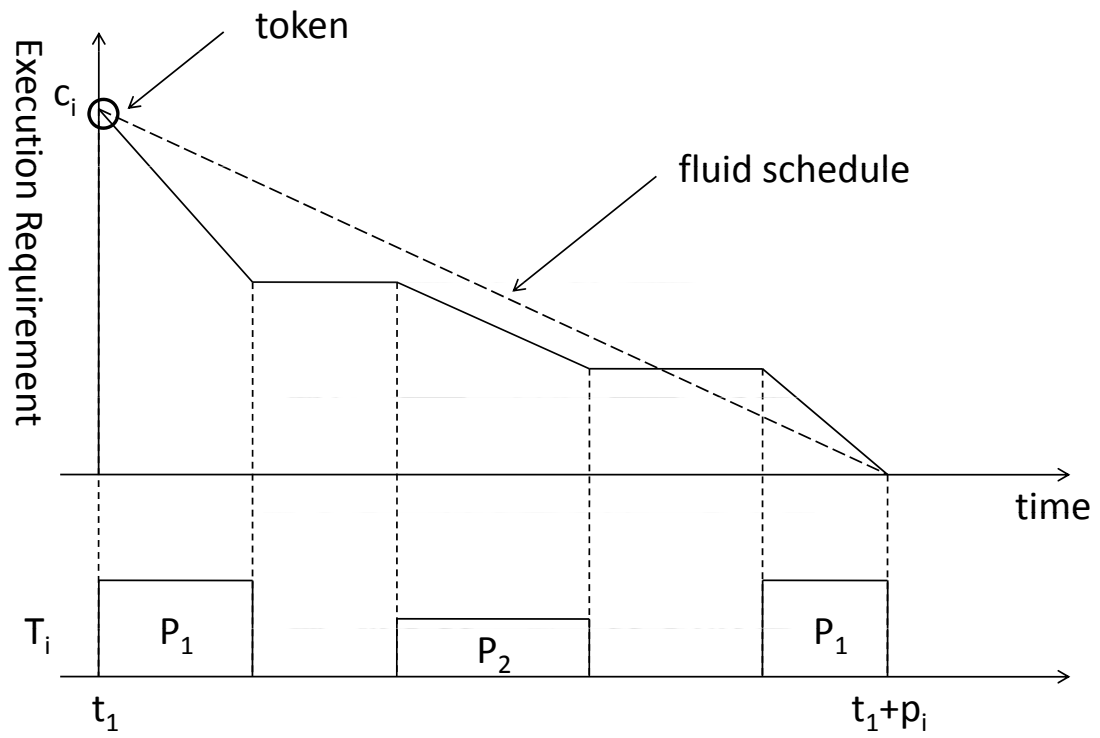


Figure 2.1: $T-L_{er}$ plane

Chapter 3

Model and $T-L_{er}$ Plane

Now, we are ready to introduce our new model for scheduling tasks on uniform multiprocessors.

3.1 P-fair and Fluid Schedule

In the fluid scheduling model, each task in the schedule executes at a constant rate, but the cost of context switching and the times of rescheduling are significant [11]. Based on fluid scheduling model, P-fair scheduling algorithm is optimal in identical multiprocessors with the basic idea fairness. P-fairness is a strong notion of fairness which ensures at any instant, the absolute value of difference between the expected allocation of execution and the actual allocation of execution to every task always be strictly less than 1 (or fluid schedule) [6]. Our idea is also based on P-fair and fluid schedule and extends to uniform multiprocessors.

We know P-fair is optimal in identical multiprocessors, to build an opti-

mal scheduling algorithm, we use the concepts of fluid schedule, fairness, and urgency in P-fair, let the time quantum be as small as possible, and schedule urgent task first. The urgent task means if we do not schedule the task to execute right now, it will miss deadline. The idea of fluid schedule for identical multiprocessors could also work on uniform multiprocessors because it considers the minimum time quantum for each processor, and the computing capacity of all processors could accumulate virtually, and then assign to every task based on their execution requirements.

3.2 T- L_{er} Planes

From Chapter 1, we know T-L plane can present the behavior of tasks for identical multiprocessors. In T-L planes, *Execution Time* is represented on the y-axis, *Time* is represented on the x-axis. We replace *Execution Time* with *Execution Requirement*, extend the model for uniform multiprocessors, and call it **T**ime and **L**ocal **E**xecution **R**equirement planes (or T- L_{er} planes), based on T-L plane and L-C plane [7, 8].

To build a 2D plane for uniform multiprocessors, as shown in Figure 2.1, T_i arrive at time t_1 and its deadline is at time $t_1 + p_i$. The dotted line from $(0, c_i)$ to $(t_1 + p_i, 0)$ indicates the fluid schedule, the slope can be indicated by u_i . Tasks assigned to different processors will have different execution rates. Since the computing capacity of P_1 is larger than P_2 , tasks have higher execution rate

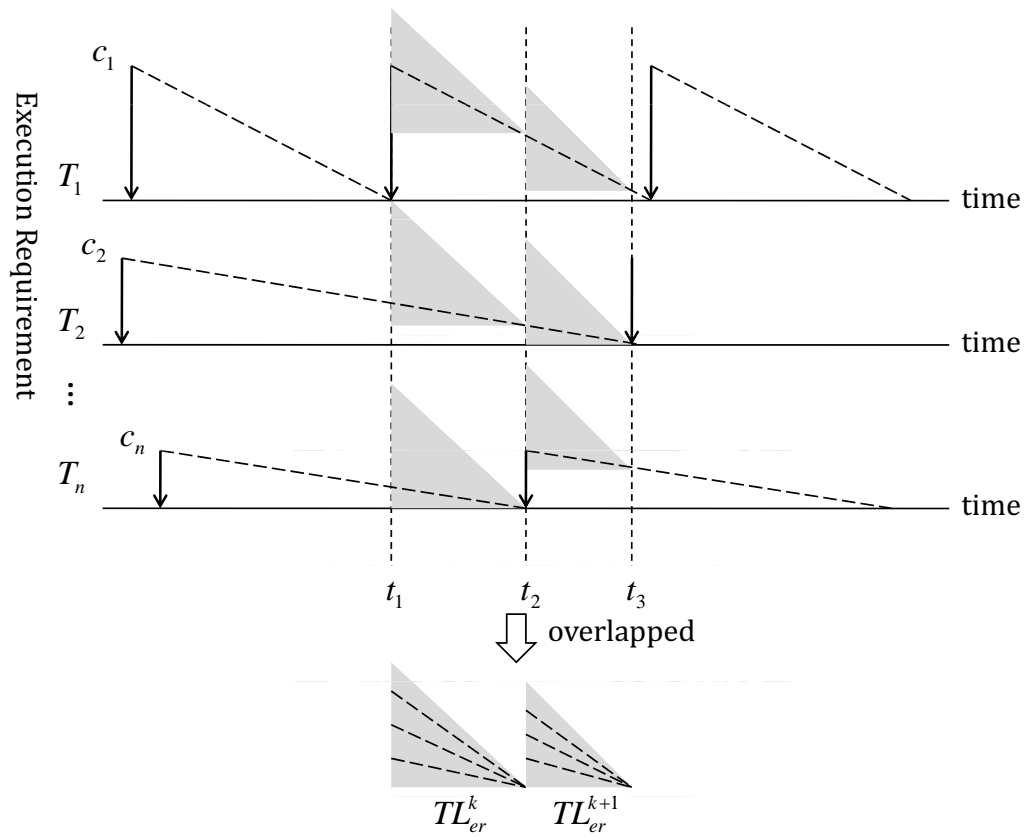


Figure 3.1: T-L_{er} Planes

on P_1 . Thus, the slope of task while assigned to P_1 is larger.

Now we consider n tasks, their fluid schedules can be constructed as shown in Figure 3.1. As T-L plane, a right isosceles triangle (Recalled we assume s_1 to be the biggest and its value is equal to 1, any task assigned to s_1 will move diagonally down) can be found between every two consecutive scheduling events, end of periods. Here we divide all the T- L_{er} planes by the periods of all tasks, the deadline of every task is not within any two consecutive scheduling events and n triangles of each task between every two consecutive scheduling events can be overlapped together. It means we can schedule in one T- L_{er} plane without consider the deadline of each task, just consider the local execution requirement of each task. We called the k^{th} isosceles triangles as TL_{er}^k , where k is simply increasing over time. The bottom side of the triangle represents *time*. The vertical side of the triangle represents remaining execution requirement of tasks, which we call *local remaining execution requirement*. Fluid schedule for each task can be constructed as overlapped in each TL_{er}^k plane with the same slope, and the local remaining execution requirement of task i in k^{th} T-L plane is equal to $u_i \cdot t_f^k$. If we finish all the jobs before the end of T- L_{er} plane, we could give an optimal scheduling algorithm.

3.3 Definitions in One T-L_{er} Plane

In the T-L_{er} plane, we define $l_{i,j}$ to represent the remaining execution requirement of task T_i at time t_j , the value of $l_{i,0}$ is equal to $u_i \cdot t_f^k$. It shows that $l_{i,0} > l_{i+1,0}, \forall i, 1 \leq i < n$. We also define $r_{i,j} = l_{i,j}/(t_f^k - t_j)$ to represent the local utilization of task i at time t_j , the value of $r_{i,0}$ is equal to u_i . To distinguish from r and to verify the set of tasks and processors is still feasible in the T-L_{er} plane, we define $r'_{i,j}$ to represent the $r_{i,j}$ at the time t_j sorted in decreasing order. Therefore, the order of $r'_{i,0}$ in a T-L_{er} plane is equal to $l_{i,0}$ at time 0.

3.4 T-L_{er} Plane vs T-L Plane

As we introduced, T-L_{er} planes are repeated over time. Giving a feasible scheduling algorithm for one T-L_{er} plane will also schedule other T-L_{er} planes. As shown in Figure 3.2, a good scheduling algorithm should keep all the tokens move to t_f^k , make sure all the remaining execution requirement of tasks are equal to 0 at time t_f^k ; in other words, they finish their jobs. As T-L plane, the dashed line represents the fluid schedule for each task, every task is represented by a token. Now we would like to discuss the details of a T-L_{er} plane, and describe the innovation inspired from T-L plane.

Firstly, in T-L plane, execution time is represented on the y-axis, but in T-L_{er} plane, it is replaced by execution requirement. Thus, local remaining execution time is replaced by local remaining execution requirement for uniform

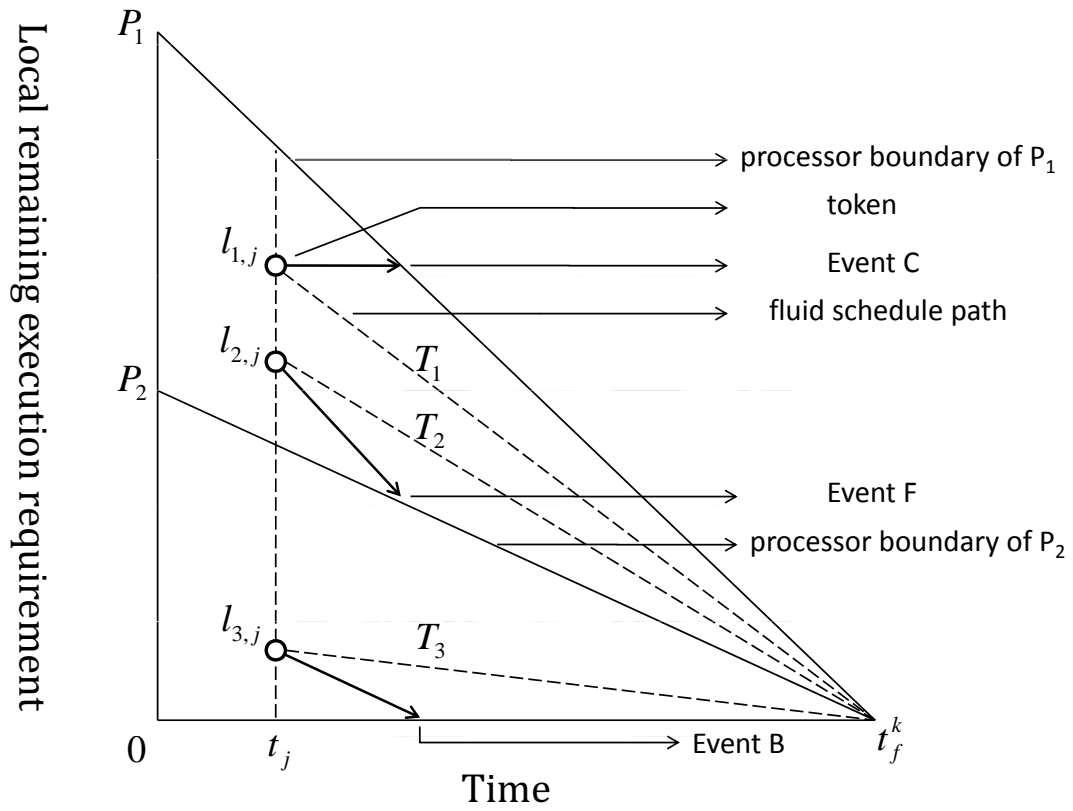


Figure 3.2: k^{th} T- L_{er} Plane

multiprocessors. Secondly, the each processor between the time from 0 to t_f^k can construct a fluid schedule path of a task with full utilization described by solid line. We call it *processor boundary*, indicating it is the critical boundary of processors, where rescheduling might be needed when some tokens meet the boundaries. The slope of the processor boundary for each processor P_i is proportional to s_i . For example, as shown in Figure 3.2, there are two processors P_1 and P_2 in this T- L_{er} plane, s_1 and s_2 are 1 and 0.5 respectively. They have solid lines to represent the remaining computing capacity and connect all the way to t_f^k . Thirdly, in T- L_{er} plane, tasks could be assigned to different processors and move downward in different slopes. For example, as shown in Figure 3.2, when task 2 is assigned to processor 1, its token would move 45 degrees downward, when task 3 is assigned to processor 2, its token would move 22.5 degrees downward due to the speed of P_2 is 0.5.

In T- L_{er} plane, we observe that there are three kinds of time instants where rescheduling is needed. Firstly, when the local remaining execution requirement of a task is equal to 0, it would hit the bottom of a T- L_{er} plane. As T-L plane, we call it *bottom hitting event* (or event B). Secondly, when the local remaining execution requirement of a task is equal to e_1 (the computing capacity of processor P_1), it would hit the ceiling of a T- L_{er} plane. As T-L plane, we call it *ceiling hitting event* (or event C). We should assign this task to processor P_1 , otherwise, it could not finish in this T- L_{er} plane. Thirdly, there is a new event in uniform multiprocessors. When execution requirement of a task is equal to

the computing capacity of processor i , it will hit the processor boundary of processor i . We call it *floor hitting event* (or event F). Although, when event F occurs, it is not necessary to reschedule to satisfy FG condition, it is the precaution time instant to reschedule for our optimal scheduling algorithms. Whenever any of these three events occurs, we will reschedule all the tasks in our optimal scheduling algorithms.



Chapter 4

Optimal Scheduling Algorithms for Uniform Multiprocessors

Our scheduling algorithms for uniform multiprocessors are based on the idea of "precaution". That is we reschedule precautionously when the C, F, and B events occur before the FG condition is violated. When we reschedule, we always assign greedily the task with the largest local remaining execution requirement the fastest processor. Therefore, the two scheduling algorithms we are going to present are called Precaution Greedy (PG) and Precaution Cut Greedy (PCG) scheduling algorithms. The name "Cut" in PCG scheduling algorithm is because we cut the times of rescheduling dramatically. We will present the two scheduling algorithms in the following sections and prove its optimality.

Algorithm *Precaution Greedy*

Input: A set τ of n tasks $\{ T_1, T_2, \dots, T_n \}$ with Utilization u_1, u_2, \dots, u_n .

A set π of n processors $\{ P_1, P_2, \dots, P_n \}$ with Speed $\{ s_1, s_2, \dots, s_n \}$.

1. while (any event $[C|F|B]$ occurs at time t_k) {
2. while (there are ready tasks) {
3. assign task with largest remaining execution
4. requirement to the fastest idle processor.
5. }
6. }

Figure 4.1: PG Scheduling Algorithm

4.1 Precaution Greedy (PG) Scheduling Algorithm

Based on the FG condition, it is easy to check whether a task and processor set is feasible. However, the problem is when to do rescheduling so that the feasible set is schedulable and how to minimize the number of rescheduling. We know that it might not be schedulable if we do not reschedule until the most urgent event occurs as the LLREF does. To achieve optimum, we would like to reschedule at some events precautionously before the FG condition is violated. Fortunately, we find that before the FG condition is violated there must be some events occur earlier. Therefore, PG scheduling algorithm schedules greedily the task with largest local remaining execution requirement first to the fastest processor and reschedules when any event occurs until there is not any idle task or idle processor. It is described in Figure 4.1 and an example is given in Figure 4.3. Although PG is simple, it is the first optimal scheduling algorithm for uniform multiprocessors. However, it might have indefinite times of rescheduling. We will prove its optimality later and propose a more efficient optimal scheduling algorithm.

4.2 Precaution Cut Greedy (PCG) Scheduling Algorithm

Based on PG, PCG scheduling algorithm is also a precaution based scheduling algorithm to reduce the number of rescheduling. It reschedules when event $[B|C|F]$ occurs as shown in Figure 4.2 and an example is given in Figure 4.4.

Algorithm *Precaution Cut Greedy*

Input: A set τ of n tasks $\{ T_1, T_2, \dots, T_n \}$ with Utilization u_1, u_2, \dots, u_n .

A set π of n processors $\{ P_1, P_2, \dots, P_n \}$ with Speed $\{ s_1, s_2, \dots, s_n \}$.

1. while (any event $[C|F|B]$ occurs at time t_k) {
2. if $s_i = r_{j,k}$
3. assign T_j to P_i until the end of T- L_{er} plane
4. remove P_i from π , remove T_j from τ
5. else $r_{j,k} = 0$
6. remove T_j from τ
7. while (there are ready tasks)
8. assign task with largest remaining execution
9. requirement to the fastest idle processor.
10. }

Figure 4.2: **PCG Scheduling Algorithm**

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. PCG removes the task and the processor and the remaining task and processor set is still feasible. PCG also schedules greedily the remaining tasks with largest local remaining execution requirement first to the fastest processors and reschedules when any event occurs until there is not any idle task or idle processor. With the removal of task and processor, the number of rescheduling in PCG decreases dramatically. Its optimality will be proved later.

PCG gives better performance than PG, as shown in Figure 4.3 and Figure 4.4 with 3 tasks and 2 processors, where $T_1 = T_2 = T_3 = (4.6, 10)$, $s_1 = 1$, $s_2 = 0.5$. Obviously, by PCG, the times of rescheduling is 4, but by PG, the times of rescheduling is much more than PCG. At time 0, if the values of $r_{1,0}$, $r_{2,0}$, and $r_{3,0}$ are closer to s_2 , the times of rescheduling will increase dramatically. The worst, the times of rescheduling might be close to ∞ .

4.3 Proof of Optimality

Here, we will prove the optimality of PG and PCG scheduling algorithms. In $T-L_{er}$ plane, we will reschedule when any event $[B|C|F]$ occurs. Actually, all the events occur when the execution requirement of some task is equal to the computing capacity of some processor or 0. To prove the optimality of the algorithm, we define the task order to be the decreasing sequence of task local

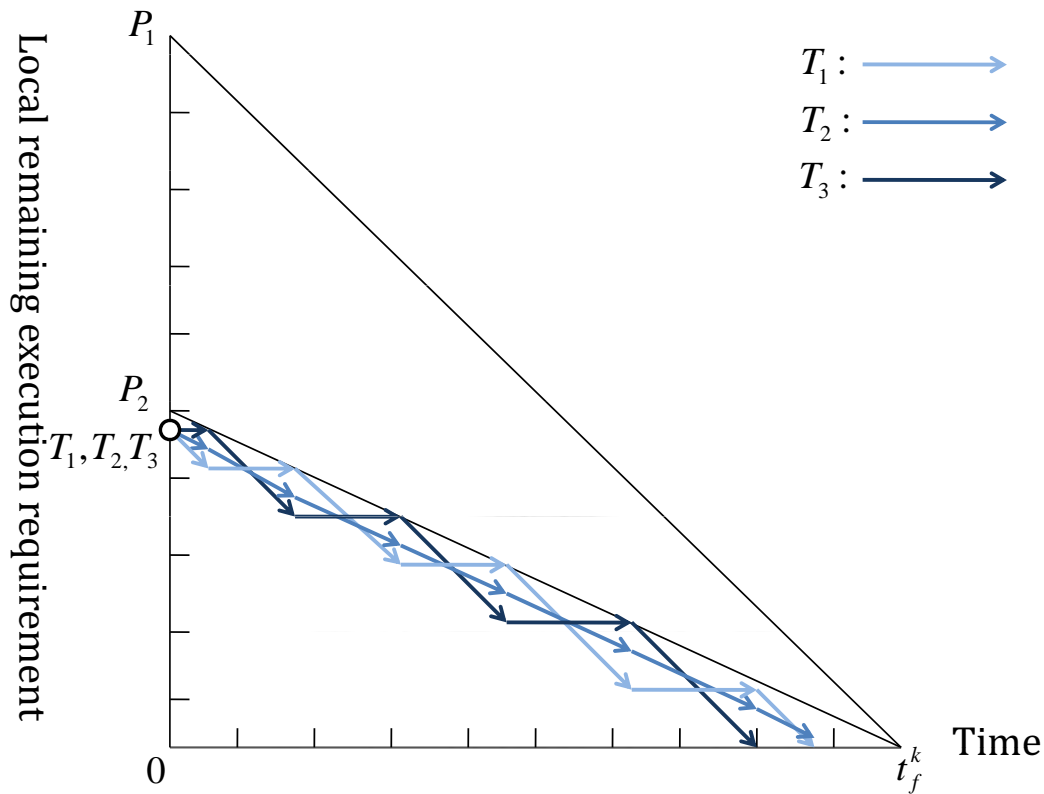


Figure 4.3: Example of PG Scheduling Algorithm

remaining execution requirement. As each task consumes processor computing capacity differently, the task order in a $T-L_{er}$ plane are changed dynamically. Therefore, it is not necessarily equal to the mapping of tasks assigned to processors. As shown in Figure 4.5, at time 0, task 1 has larger remaining execution requirement than task 2, but at time t_j , the order exchanges. However, the task assignment is still the same. If we can guarantee that at any event the FG condition holds according to the local remaining execution requirements in the

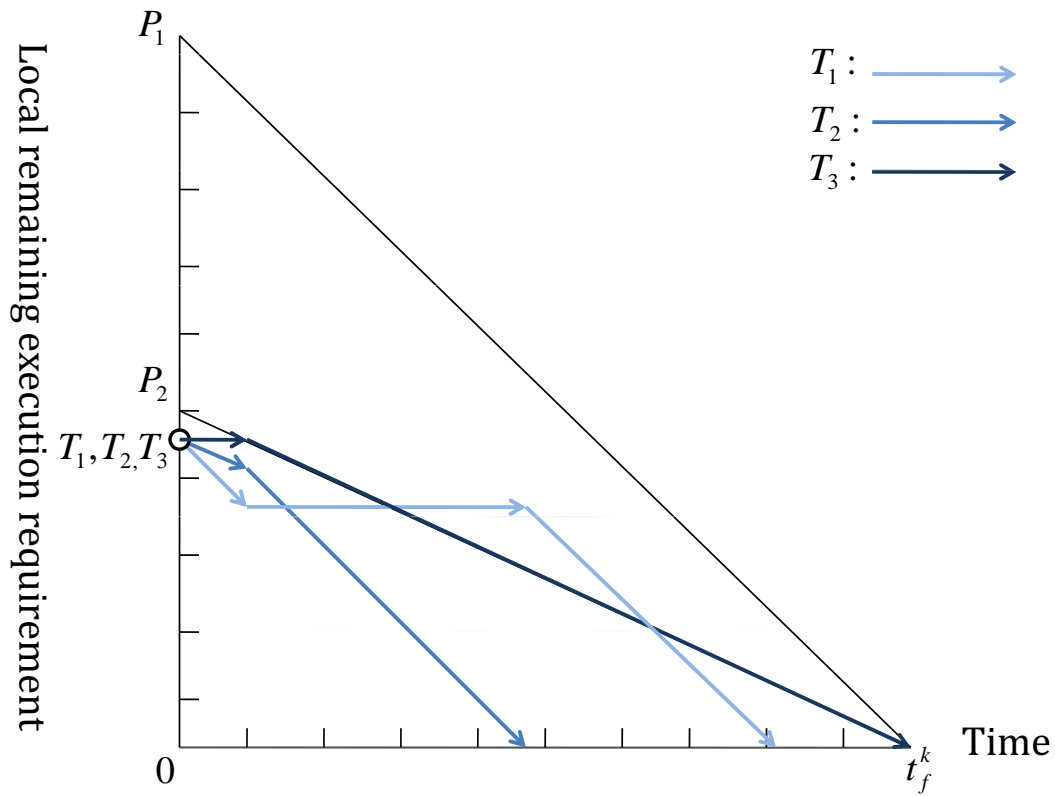


Figure 4.4: Example of PCG Scheduling Algorithm

task order, by the definition of FG condition, the task set is feasible.

Theorem 3 *When any event occurs, the set of tasks and processors is feasible by PG and PCG scheduling algorithms.*

Proof. Since both PG and PCG reschedule when any event $[B|C|F]$ occurs, the feasibility condition is the same at this moment. In the beginning, the time is 0, suppose the event occurs after time t_g lapses. Both PG and PCG assign

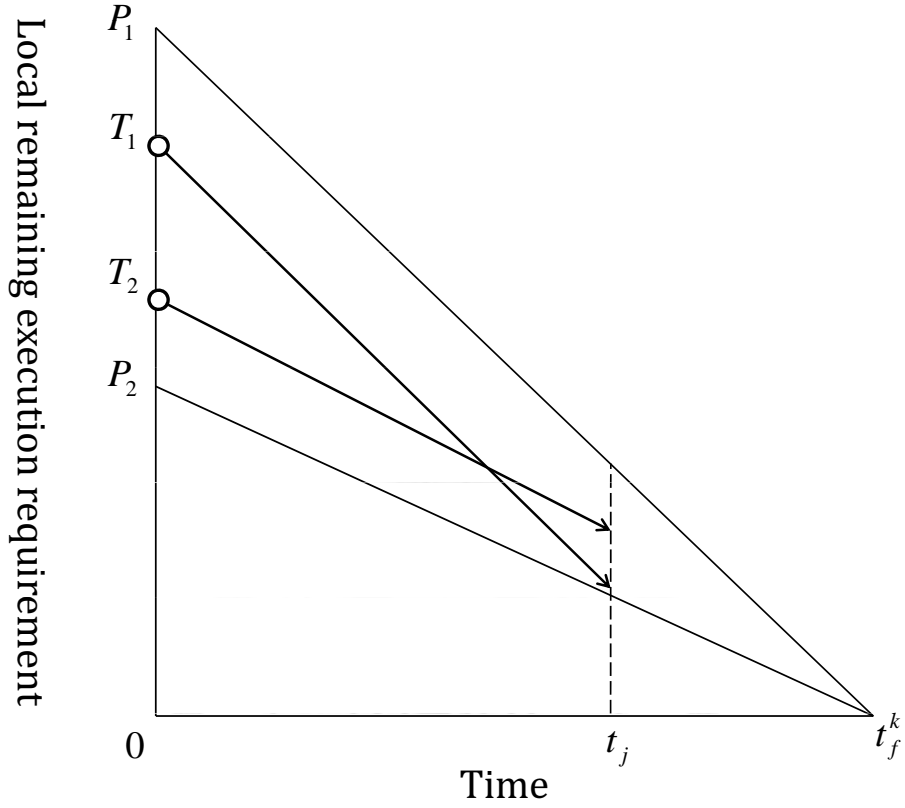


Figure 4.5: Task Order

T_i to P_i , $\forall i$, $1 \leq i \leq n$. Let $r'_{i,g}$ to represent the new $r_{i,g}$ at time t_g , actually $l_{i,0} - s_i \cdot t_g$, $\forall i$, $1 \leq i \leq n$, sorted in decreasing order. As we can see that the order of $r'_{i,g}$ might change. Moreover, we let $R'_{i,g}$ represent $\sum_{j=1}^i r'_{j,g}$ and T'_i represent the new i th task at time t_g in the order of $r'_{i,g}$.

We show that any event occurs earlier than FG condition is violated. According the definition of FG condition, before it is violated, at time t_g , there

must exist that

$$S_i \geq R'_{i,g}, \forall i, 1 \leq i \leq n, \exists S_k = R'_{k,g}, 1 \leq k \leq n \quad (4.1)$$

This is to say the FG condition would be violated if we do not reschedule when, $s_1 \geq r'_{i,g}, \forall i, 1 \leq i \leq k$ (or event C occurs), and $s_k \leq r'_{k,g}$ (because $S_{k-1} \geq R'_{k-1,g}$ and $S_k = R'_{k,g}$). To derive the time t_g when condition 4 holds, we can solve the following equation:

$$\sum_{j=1}^k s_j = \sum_{j=1}^k (l_{i_j,0} - s_{i_j} \cdot t_g) / (t_f^k - t_g), 1 \leq i_{j-1} < i_j \leq n$$

Therefore, we can find some $T'_i, 1 \leq i \leq k, \notin \{T_j | 1 \leq j \leq k\}$, or we could not solve t , that means the FG condition still holds. That is there exists $T_a \in \{T_j | 1 \leq j \leq k\}$ and $T_a \notin \{T'_j | 1 \leq j \leq k\}$, correspondingly, and $T_b \notin \{T_j | 1 \leq j \leq k\}$ and $T_b \in \{T'_j | 1 \leq j \leq k\}$. If $s_1 > r'_{a,g} \geq s_k$, T_a would hit processor boundary of P_{k+1} and event F occurs (because $s_{k+1} \geq r'_{k+1,g}$ when $S_k = R'_{k,g}$). If $s_k > r_{a,g} > 0$, T_b would hit processor boundary of P_k and event F occurs (because $r_{b,g}$ is smaller than $r_{a,g}$ and s_k).

Therefore, there exists an event occurs before FG condition is violated while scheduling by PG or PCG. In other words, rescheduling when any event occurs, the set of tasks and processors will be feasible using PG and PCG scheduling algorithm. \square

Theorem 4 *For a feasible set of n tasks and n processors, the PG scheduling algorithm is optimal and feasible for uniform multiprocessors.*

Proof. Since the set of tasks and processors is feasible, it follows the FG condition. Since PG reschedules at any events, according to *Theorem 3*, the new task set is still feasible in the T- L_{er} plane. Since each T- L_{er} plane is independent, the while schedule is feasible. Therefore, any feasible task set can be schedulable using PG. That is PG scheduling algorithm is optimal. \square

Theorem 5 *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.*

Proof. According to theorem 2, the FG condition, we know if

$$S_i \geq U_i \text{ and } S_i \geq R'_{i,a} \text{ at time } t_a \forall i, 1 \leq i \leq n, \quad (4.2)$$

the set of tasks and processors is feasible. By theorem 3, we know rescheduling when any event occurs, the set of tasks and processors are still feasible, therefore condition 5 still hold. Suppose when event $[C|F]$ occurs at time t_g , we assume task T_j hits processor boundary of P_i , $1 \leq i, j \leq n$, and when event B occurs, we assume task T_j finishes its job. Suppose when T_j is removed from the task set τ , the new $r_{k,g}$ ' will be reindexing as $r'_{k+1,g}$, $\forall j \leq k < n$, The relationship between s_i and $r'_{j,g}$ before removal can be classified into four cases:

Case 1: $s_i = r'_{j,g}$, $1 \leq i = j \leq n$

By condition 5, after removal, we can derive the FG condition of new task and

processor set as

$$S_k \geq R'_{k,g}, \forall k, 1 \leq k \leq i-1$$

$$S_k \geq R'_{k,g} - r'_{i,g}, \forall k, i < k \leq n$$

With proper reindexing, the FG condition still hold.

Case 2: $s_i = r'_{j,g}, 1 \leq j < i \leq n$

By condition 5, after removal, we can derive the FG condition of new task and processor set as

$$S_k \geq R'_{k,g}, \forall k, 1 \leq k \leq j-1$$

and for $1 \leq k \leq j < i \leq m \leq n, s_k \geq s_i = r'_{j,g} \geq r'_{m,g}$, We can derive that

$$S_k \geq R'_{k+1,g} - r'_{j,g}, \forall k, j \leq k < i$$

$$\text{and } S_k - s_i \geq R'_{k,g} - r'_{j,g}, \forall k, i < k \leq n$$

With proper reindexing, the FG condition still hold.

Case 3: $s_i = r'_{j,g}, 1 \leq i < j \leq n$

By condition 5, after removal, we can derive the FG condition of new task and processor set as

$$S_k - s_i \geq R'_{k,g} - r'_{j,g}, \forall k, j < k \leq n$$

$$\text{Since } S_k \geq R'_{k,g} \text{ and } s_k \leq s_i = r'_{j,g} \leq r'_{m,g},$$

$$\forall k, i-1 \leq k \leq n, \forall m, 1 \leq m \leq j-1$$

We can derive that

$$S_k - s_i \geq R'_{k,g} - r'_{k,g}, \forall k, i \leq k < j$$

Finally, as we showed above

$$S_k \geq R_{k,g}, \forall k, 1 \leq k < i$$

With proper reindexing, the FG condition still hold.

Case 4: One task finishes, we assume $r'_{j,g} = 0$

This indicates j equals to n . With proper reindexing after removal, the FG condition still hold. □

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

Proof. Since the set of tasks and processors is feasible, it follows the FG condition. Since PCG reschedules at any events, according to *Theorem 3* and *Theorem 5*, the new task set is still feasible in the T- L_{er} plane. Since each T- L_{er} plane is independent, the while schedule is feasible. Therefore, any feasible task set can be schedulable using PCG. That is PCG scheduling algorithm is optimal. □

In PCG scheduling algorithm, when any event happens, the number of tasks and processors will decrease by 1 respectively. Therefore, the maximal

number of rescheduling in one $T-L_{er}$ plane is n . The times of rescheduling will be dramatically less than PG and fluid schedule.



Chapter 5

Experimental Evaluation

5.1 Input Generator

We construct simulation-based experiments for PG and PCG scheduling algorithms. Before simulation, We give an input generator which will generate set of tasks and processors randomly, and check whether they are feasible by FG condition. Because our simulation is based on multiprocessors, we would like to discuss about the precautionary utilization for all processors as shown in Figure 5.1. As we known, the computing capacity of each processor might be different, it could not be sure the utilization of each processor before scheduling, we consider C_n/E_n to represent the precautionary utilization, the value is only for foreseeing.

5.2 PG and PCG Performance Evaluation

To give a comparison for PG and PCG, we implement EDF according to the following rules presented by Funk [9]:

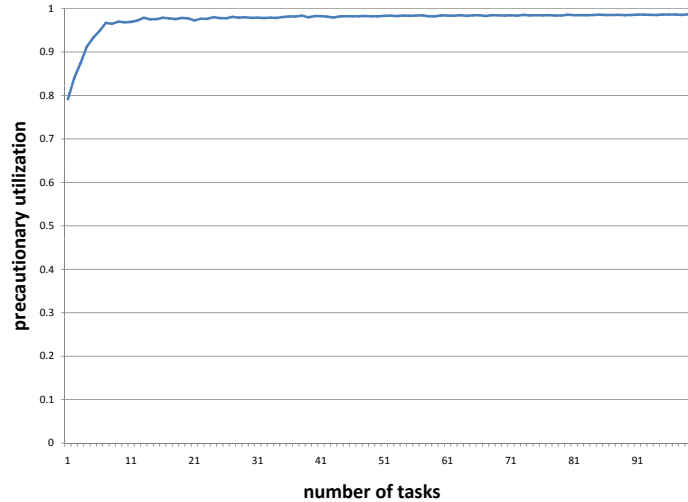


Figure 5.1: Input Generator

- No processor is idled while there is an active job awaiting execution.
- When fewer than m jobs are active, they are required to execution upon the fastest processors while the slowest are idled.
- Higher priority jobs are executed on faster processors.

To analyze the schedulability of EDF on uniform multiprocessors, as shown in Figure 5.2, we generate 1000000 set of tasks and processors for each pair of tasks and processors. In Figure 5.2(a), the number of tasks and processors is equal, in Figure 5.2(b), the number of tasks is 10. It is easy to show while the number of tasks and processors increase, EDF will miss deadline because the complexity of assignment raising. EDF only considers about the deadline of each task, it will miss deadline while the urgent tasks have not be executed.

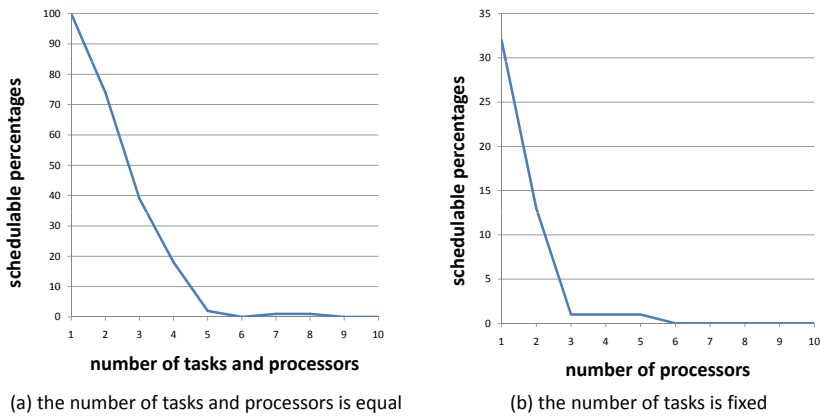


Figure 5.2: Schedulability of EDF on uniform multiprocessors

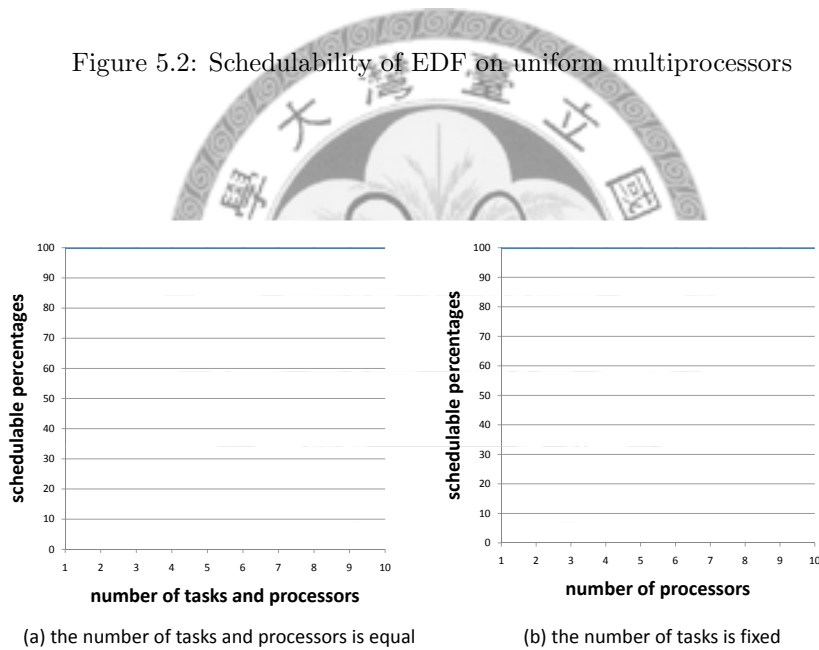


Figure 5.3: Schedulability of PG

To analyze the schedulability of PG and PCG on uniform multiprocessors, as shown in Figure 5.3 and Figure 5.4, we generate 1000000 set of tasks and processors for each pair of tasks and processors. In Figure 5.3(a), the number

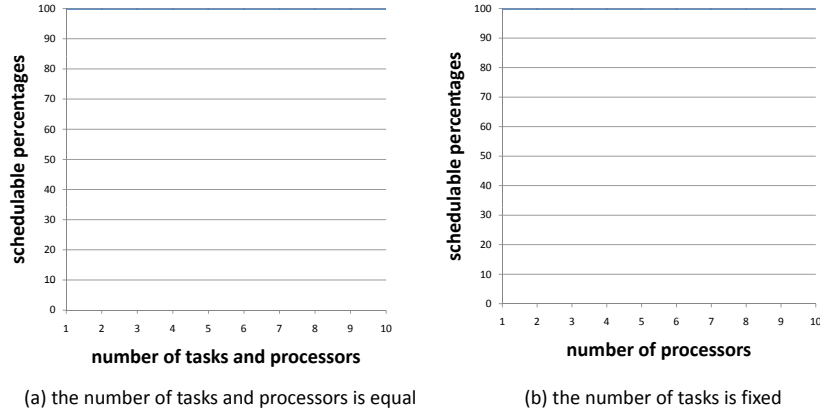
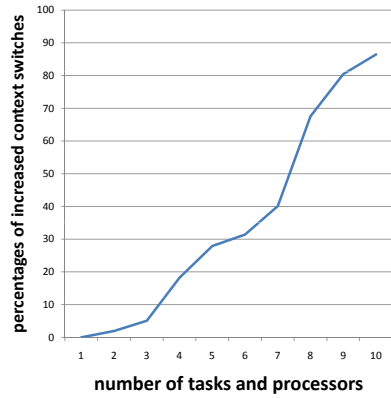


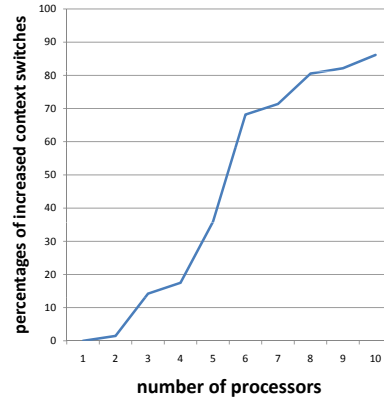
Figure 5.4: Schedulability of PCG

of tasks and processors is equal, in Figure 5.3(b), the number of tasks is 10. In Figure 5.4(a), the number of tasks and processors is equal, in Figure 5.4(b), the number of tasks is 10. PG and PCG could schedule all the set of tasks and processors.

Although PG and PCG is schedulable for all cases on uniform multiprocessors, the times of context switches is increasing based on the number of $T-L_{er}$ planes. As shown in Figure 5.5 and Figure 5.6, we generate 100000000 set for each pair of tasks and processors. In Figure 5.5(a), the number of tasks and processors is equal, in Figure 5.5(b), the number of tasks is 10. In Figure 5.6(a), the number of tasks and processors is equal, in Figure 5.6(b), the number of tasks is 10. We could figure out the times of context switching by PG and PCG is larger than EDF, While the number of tasks and processors increase, the

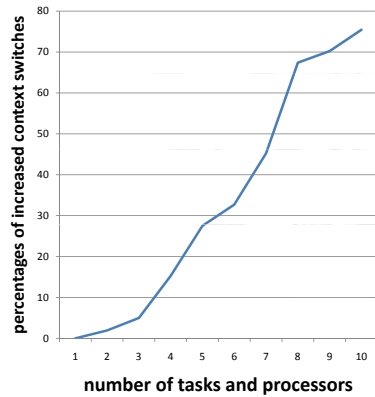


(a) the number of tasks and processors is equal

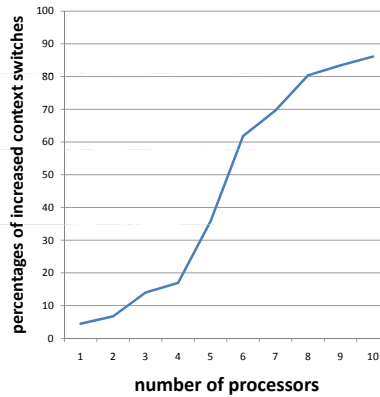


(b) the number of tasks is fixed

Figure 5.5: Performance Analysis of PG while Comparing to EDF



(a) the number of tasks and processors is equal



(b) the number of tasks is fixed

Figure 5.6: Performance Analysis of PCG while Comparing to EDF

times of context switching will increase, too. This is because We generate all the tasks randomly, the period of them is different, PG and PCG will generate lots of $T-L_{er}$ planes and have lots of scheduling within each plane. It shows

although the schedulability of PG and PCG is optimal, the time complexity of them is larger, too.

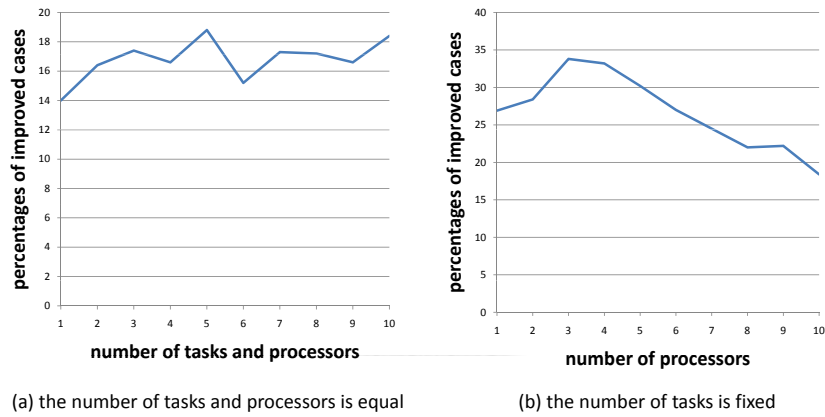


Figure 5.7: The number of increasing cases between PCG and PG

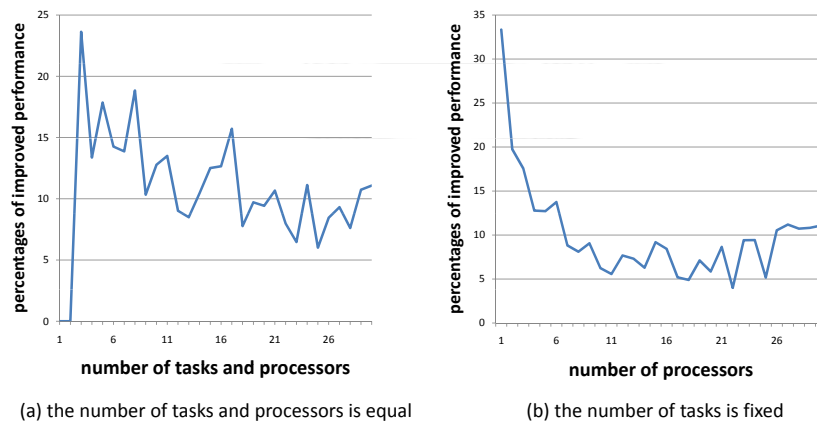
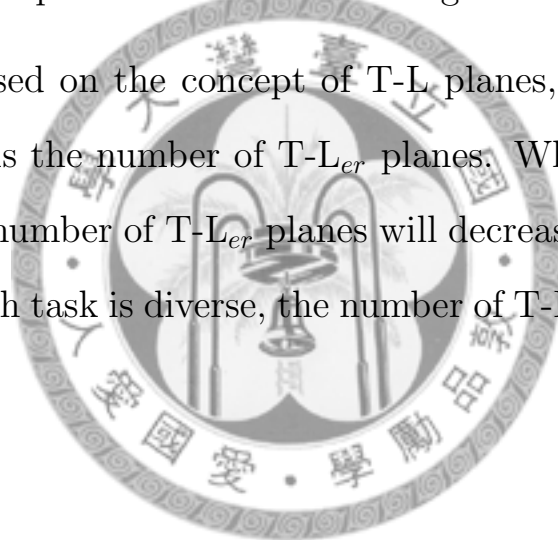


Figure 5.8: Performance Analysis between PCG and PG

Now we want to discuss about the performance between PG and PCG as shown in Figure 5.7 and Figure 5.8. In Figure 5.7(a), the number of tasks and processors is equal, in Figure 5.7(b), the number of tasks is 10. In Figure 5.8(a), the number of tasks and processors is equal, in Figure 5.8(b), the number of tasks is 30. It shows that PCG give better performance than PG, and when the number of tasks and processors increase, PCG will not give better performance. This is because the number of $T-L_{er}$ planes is significant, although PCG give an upper bound n in a $T-L_{er}$ plane, when the number of $T-L_{er}$ planes even larger, it could not be sure the performance of it is still good.

While we are based on the concept of T-L planes, the bottleneck of our scheduling algorithm is the number of $T-L_{er}$ planes. When the period of each task is harmonic, the number of $T-L_{er}$ planes will decrease. On the other hand, when the period of each task is diverse, the number of $T-L_{er}$ planes will increase dramatically.



Chapter 6

Conclusions

Although feasible on-line scheduling algorithm for uniform multiprocessors is difficult, we provide a novel T- L_{er} plane model for uniform multiprocessors to observe the behavior of task and processor easily. We present the Precaution Greedy algorithm, which is the first optimal dynamic-priority scheduling algorithm for uniform multiprocessors and the Precaution Cut Greedy scheduling algorithm, which is also optimal and with the times of rescheduling decreased dramatically. We also prove the optimality of the above algorithms and an upper bound n of the times of rescheduling in a T- L_{er} plane. Finally we give an experimental evaluation for PG, PCG, and EDF scheduling, prove PCG will give better performance than PG. We believe the results might be applicable to current asymmetric multicore platforms of similar uniform multiprocessors, where the processing units are capable of executing the same instruction with different rates, rising the performance in parallel and decreasing the times of context switching. Because the simplicity of our results, it might be also appli-

cable to the most complicated unrelated parallel machines while each task T_i completes $(r_{i,j} \times t)$ units of execution by executing on processor P_j for t time units, the execution work for each task will be an constant value, therefore, we might migrate the same model on unrelated multiprocessors.



Bibliography

- [1] Advanced Micro Devices (AMD). Multi-core processors - the next evolution in computing.
- [2] 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.
- [3] S.K. Baruah and J. Goossens. Preemptive scheduling of uniform processor systems. *Journal of the ACM*, 25(1):92–101, Jan. 1978.
- [4] S.K. Baruah and J. Goossens. Rate-monotonic scheduling on uniform multiprocessors. *IEEE Transactions on Computers*, 52(7):966–970, Jul 2003.
- [5] J.M. Calandrino, D. Baumberger, S. Tong Li, Hahn, and J.H. Anderson. Soft real-time scheduling on performance asymmetric multicore platforms. *Real Time and Embedded Technology and Applications Symposium*, April 2007.
- [6] A Chandra, M Adler, and P Shenoy. Deadline fair scheduling: bridging the theory and practice of proportionate pair scheduling in multiprocessor

- systems. *Real-Time Technology and Applications Symposium*, pages 3–14, June 2001.
- [7] Hyeonjoong Cho, Binoy Ravindran, and E. Douglas Jensen. An optimal real-time scheduling algorithm for multiprocessors. *RTSS*, pages 101–110, Oct. 2006.
- [8] 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.
- [9] S. Funk, J. Goossens, and S. Baruah. On-line scheduling on uniform multiprocessors. *RTSS*, pages 183–192, Dec. 2001.
- [10] Dorit S. Hochbaum and David B. Shmoys. A polynomial approximation scheme for scheduling on uniform processors: Using the dual approximation approach. *SIAM Journal on Computing*, 17(3):539–551, 1988.
- [11] Philip Holman and James H. Anderson. Adapting pfair scheduling for symmetric multiprocessors. *Journal of Embedded Computing*, 1(4):543–564, 2005.
- [12] Edward C. Horvath, Shui Lam, and Ravi Sethi. A level algorithm for preemptive scheduling. *Journal of the ACM*, 24(1):32–43, January 1977.

- [13] Klaus Jansen and Lorant Porkolab. Improved approximation schemes for scheduling unrelated parallel machines. *ACM symposium on Theory of computing*, 1999.
- [14] Jan Karel Lenstra, David B. Shmoys, and Éva Tardos. Approximation algorithms for scheduling unrelated parallel machines. *Mathematical Programming*, 46(1), Jan. 1990.
- [15] 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.
- [16] Jane W.S. Liu. *Real-Time Systems*. Prentice Hall, 2000.
- [17] R.R. Muntz and E.G. Coffman. Optimal preemptive scheduling on two-processor systems. *Computers, IEEE Transactions on*, (11):1014–1020, Nov. 1969.
- [18] B. Srivastava. An effective heuristic for minimising makespan on unrelated parallel machines. *The Journal of the Operational Research Society*, 49(8):886–894, Aug. 1998.