Extensions to Fixed Priority with Preemption Threshold and Reservation-Based Scheduling

by

Jiongxiong Chen

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2005

I hereby declare that I am the sole author of this thesis.

I authorize the University of Waterloo to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the University of Waterloo to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Abstract

This thesis focuses on two real-time scheduling algorithms: fixed-priority with preemption threshold (FPPT) and the reservation-based algorithm (RBA).

FPPT is an important form of real-time scheduling algorithm that generalizes both fixed-priority preemptive (FPP) and fixed-priority non-preemptive (FPNP), i.e., FPP and FPNP are the two boundary cases for FPPT. The critical instant for FPPT is rephrased, and it is shown that FPPT is robust under its critical instant. When a task set is schedulable by FPPT with predefined regular priorities, there may exist multiple valid preemption threshold assignments, which are proved to be delimited by two special assignments, called minimal and maximal assignment respectively. This thesis presents effective algorithms to compute the minimal and maximal assignment by starting from FPNP. In addition, algorithms are presented to compute the minimal (maximal) assignment when the other is known. The maximal assignment reduces the number of context switches and can eliminate some resource sharing issues. It is shown that the algorithm to find the maximal assignment can be used to perform a schedulability test for FPPT. Resource management protocols for FPPT are not addressed in this work, which is left for future study.

RBA is a kind of bandwidth reservation algorithm in which an even duration of time (the bandwidth) is made available for executing aperiodic tasks to improve their response time. This thesis provides the theoretical support to compute the maximal size of the reserved bandwidth after guaranteeing schedulability of a periodic task set for earliest deadline first (EDF), rate monotonic (RMA), and FPPT algorithms. When RMA is used to schedule periodic tasks, the known algorithm to calculate the reservation size is complicated and ineffective. It is shown that the combination of RBA and FPPT can obtain a reservation size no smaller than that of RMA and avoid the high run-time cost of EDF. Combined with a background server, the response time of aperiodic tasks can be further improved. However, this thesis does not consider all practical issues. For example, the bandwidth may be too small and the practical costs of additional context switching may be expensive.

To verify the research results of this thesis, a small real-time scheduling tool-kit was created as part of the thesis work. The tool-kit allows scheduling algorithms to be created and tested (via simulations), where scheduling results are presented as step-by-step Gantt-charts. The tool-kit was invaluable in generating examples to prove or disprove different aspects of the scheduling work for the thesis. The tool-kit has been extended with multiple scheduling algorithms, not just those specific to this thesis.

Acknowledgments

I would like to thank my supervisor Dr. Peter Buhr for his enthusiasm, encouragement, and help. His suggestions were always ready especially when I was struggling with the difficult parts of my research. I also would like to thank Dr. Forbes Burkowski for his help and tolerance of academic freedom to explore another interesting research field. I would like to thank my wife Guozhen Zheng and son Grant Jiayi Chen for their love and support. I also would like to thank my family members in mainland China for their patience and understanding in allowing me to pursue my academic target in both China and Canada.

I also would like to thank the following students in the programming language laboratory: Ashif Harji, Rodolfo Esteves, Richard Bilson, Roy Krischer, Josh Lessard, Michael Laszlo, Brad Lushman, Thomas Lynam, and David Yeung. Among them, I really would like to say special thanks to Ashif Harji and Rodolfo Esteves for their patience and help.

I would like to thank all members in my Ph.D. Examining Committee – Dr. Jia Xu, Dr. Mark Aagaard, Dr. Peter van Beek, Dr. Charlie Clarke, and Dr. Peter Buhr.

Dedication

To my family

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Real-time systems are becoming more and more important as they have entered into people's daily lives through, for example, manufacturing, transportation, and medical operation controls [66, 16]. Unlike traditional computation that only requires functional correctness, a real-time system is restricted by time constraints, as the correctness of a computation also depends on time. If a result is obtained within some specific time called the *deadline*, it is valid; otherwise, it is useless or its usage is degraded. If a real-time task misses its deadline, resulting in a catastrophic effect, it is called a *hard real-time task* such as an atomic power plant control task. If a real-time task misses its deadline from time to time, resulting in a degraded result, it is called a *soft real-time task* such as transmitting an image frame during videoconferencing. If a real-time system contains at least one hard real-time task, it is called a *hard real-time system*; otherwise it is called a *soft real-time system*. An alternative viewpoint is that a real-time system must be predictable and the system reacts to events in a timely way. The ability to react to events requires an appropriate execution order of the tasks, which is called a real-time *schedule* of tasks. The algorithm applied to find a schedule is called a *scheduling algorithm*. Many real-time scheduling algorithms have been created for different applications from static to dynamic [45, 57] from uniprocessor to multiprocessor [5, 6], from serial to parallel algorithms [44, 35, 24, 54], using both weak and strong assumptions [61, 61, 58, 11]. However, even a simple real-time scheduling problem under simple constraints for a uniprocessor or multiprocessor environment can be NP-Complete [71, 25]; hence, scheduling is one of the most difficult parts in the development of a real-time system.

In traditional concurrent programming, tasks do not have deadline constraints. The purpose of

a real-time scheduling algorithm is to guarantee the deadlines of all real-time tasks in the system, which is different from the purpose of other types of scheduling. For example, other scheduling algorithms may target load balancing in a multiprocessor system [14, 75] or minimizing memory usage. Both of these kinds of scheduling algorithms are popular in concurrent programming environments. However, these kinds of scheduling algorithms do not guarantee the deadlines of tasks.

Many real-time systems are designed for an *embedded system*, where the characteristics of the application and the operating environment can be known. In this case, the real-time system can be finely tuned to the desired degree of performance. Embedded systems may also have additional constraints that must be considered, such as a small amount of memory, power consumption, weight, or physical size. These very specific constraints are in contrast to the demands of flexibility and functionality typically available in non-real-time or general real-time systems.

In addition to a deadline parameter, real-time tasks have other operational parameters, such as ready time, starting time, and *worst case computation time* [50, 52]. Ready time and starting time are defined later in Section 1.3. Worst case computation time is the maximal amount of time required for a task to execute exclusively until it finishes, and is called *computation time* for simplification.

1.1 Real-Time Task

For real-time scheduling purposes, tasks in a task set are often classified depending on arrival patterns [66], which is shown in Figure 1.1.

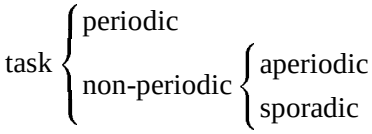


Figure 1.1: Task Categorization

1. *periodic task*

   A periodic task has a fixed release frequency that is periodic in nature, the fixed interval between two consecutive releases of a periodic task is called its *period*.

A periodic task has many instances during its lifetime. An instance of a periodic task is called a *job* of that periodic task. A periodic task consists of all of its jobs[1].

2. *non-periodic task*
   A non-periodic task does not have a fixed release frequency. Non-periodic tasks are further subdivided into two sub-categories: *aperiodic* and *sporadic*. Again, the difference between these categories lies in release frequencies.

   - *aperiodic task*
     An aperiodic task has a release frequency that is unbounded. In the extreme, this property could lead to an arbitrarily large number of simultaneously active tasks.
   - *sporadic task*
     A sporadic task is an aperiodic task with a guaranteed minimum inter-arrival time between releases.

1.2 Real-Time Scheduler

A run-time system performs scheduling through a component called a *scheduler*. When a scheduler performs a computation trying to find a valid schedule for a task set, it is performing a *schedulability test*. One very important factor when a scheduler performs a schedulability test for a task set is whether the execution of a task can be preempted during its execution by another task. If preemption is allowed, the scheduler is called *preemptive*. A preemptive scheduler may allow tasks to be interrupted at arbitrary points during their execution, and then a *context switch* allows a new or possibly the same task to continue its execution. Preemption gives greater flexibility in scheduling as task execution can be subdivided into arbitrary time intervals to facilitate time layout and achieve higher processor utilization. However, the cost of each preemption must be factored into the scheduling, and the amount of time required for a context switch must be significantly less than the computation time of a task. If no preemption is allowed, the scheduler is called *non-preemptive*. When a non-preemptive scheduler is used, the execution of a task is never preempted by another task, resulting in less context switches than for a preemptive scheduler. Furthermore, such a scheduler can be used as a cheap mechanism to ensure mutual exclusion to a shared resources, as long as resource usage does not span a task's computation (job).

[1] *Task* and *job* are used interchangeably in many real-time papers.

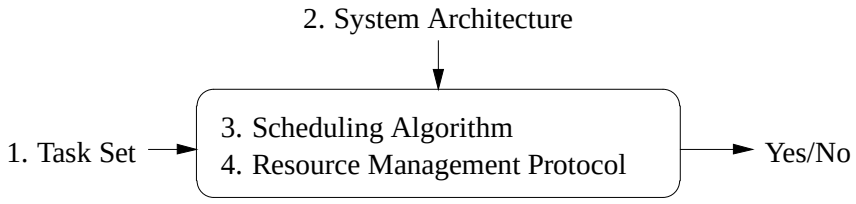


Figure 1.2: Real-Time Scheduler

No matter which scheduler is applied, the provided execution order for tasks is enforced by a *dispatcher*. Sometimes a scheduler and a dispatcher are combined and sometimes they are separate, depending on the system.

A real-time scheduler often has to consider more than the task set, such as resource management protocols, and the system architecture, as indicated in Figure 1.2. Based on the task set, system architecture, and resource management protocol, the scheduling algorithm performs a schedulability test and provides an answer of "yes" or "no". If yes, the scheduler provides one schedule and possibly provides system-status information such as the utilization of the system resources. For example, if the processor utilization is less than 1, then the unused time left by periodic tasks can be used to service aperiodic tasks. Each of these four factors is discussed in detail.

1. Task Sets

   As mentioned, tasks are divided into three types in a real-time system: periodic, aperiodic, and sporadic. Most research work attempts to transfer sporadic tasks into periodic tasks with worst case cost estimation and create special servers to service aperiodic tasks without guaranteeing deadlines. Periodic, aperiodic, and sporadic tasks require different schedulers to guarantee their deadlines or to make them complete their computations as soon as possible due to their different properties and requirements. Often periodic tasks are scheduled by a well-known scheduler such as RMA, while the unused time left by the periodic tasks is used to service aperiodic and sporadic tasks. In a uniprocessor environment, it is complicated to consider scheduling three types of tasks. However, in a multiprocessor environment, it is possible to assign a processor(s) to each kind of tasks so the different task sets can be scheduled by different schedulers on different processors. Periodic tasks can also be grouped and assigned to different processors so as to improve the processor utilization or minimize the number of processors required for the tasks [17]. Furthermore, the scheduler on each processor can be relatively simple and fast. Though many task sets

unschedulable in a uniprocessor environment may be schedulable in a multiprocessor environment, the scheduling problem in a multiprocessor environment is more complicated than in a uniprocessor environment.

2. System Architectures

   Originally, only uniprocessor computers were available. Now system architecture is experiencing a change from uniprocessor to multiprocessor, and from stand-alone machines to networked systems. Both of these changes improve the processing ability of the systems but also challenge the next generation of software, which greatly affects scheduling. One task set may be unschedulable in a uniprocessor environment, but it may be schedulable in a multiprocessor environment. While it seems that multiprocessor systems enlarge the range of problems that can be solved practically, the scheduling problem in a multiprocessor environment is more difficult in a uniprocessor environment. For example, it may seem counter-intuitive, but certain schedulers are optimal in a uniprocessor environment but may not be optimal in a multiprocessor environment [43]. The traditional schedulers and/or resource management protocols originally created for uniprocessor architectures must be rechecked when they are adopted for multiprocessor systems. In a uniprocessor environment, concurrent programming is very popular to structure complex, independent operations often associated with real-time systems. In addition, concurrent programs can immediately take advantage of any available hardware parallelism, as in current multiprocessor environments. This thesis does not consider multiprocessor environments; it only considers uniprocessor environments.

3. Scheduling Algorithms

   Many real-time schedulers are available [45, 43, 33, 41, 7, 26]. In general, each particular scheduler attempts to take advantage of some aspect of a given task set. As a result, there is no general scheduler that works for all task sets. However, it is unclear which scheduler is the most appropriate for a particular task set. In addition, there may exist different schedulability tests for a given scheduler. For example, RMA has at least three different schedulability tests [45, 33, 41]. Furthermore, a scheduler may be implemented in a number of ways, including using a parallel algorithm for schedule searching. For example, the earliest deadline first (EDF) [45] can be implemented with a parallel algorithm for schedule searching.

4. Resource Management Protocols

Task execution is seldom independent. First, there may exist precedence constraints among tasks. For example, one task can only start after another task finishes successfully. Second, tasks often communicate and/or access exclusively-shared[2] resources, which imposes additional constraints on scheduling as tasks may block when attempting these operations. As such, many real-time systems provide *resource management protocol*s [53, 1, 2, 3, 11]. Among them, priority inheritance protocol (PIP) [62], priority ceiling protocol (PCP) [27], session reservation protocol (SRP) [4], and stack-based protocol (SBP) [11] are very important ones.

There are multiple approaches to categorize real-time schedulers. For example, based on whether a real-time scheduler is event-driven or time-driven; based on whether a real-time scheduler supports multiprocessor architectures; based on whether a real-time scheduler allows preemption. Each subclass can be further categorized if necessary. For example, if a real-time scheduler is event-driven and assigns priorities to the responses to different events, it is called priority-based. Priority-based real-time schedulers can be further categorized into two types: *fixed-priority* or *dynamic-priority*, based on whether the priority of a task is fixed or dynamic during its execution.

RMA is fixed-priority preemptive (FPP). The schedulability test of a fixed-priority scheduler can be performed offline (statically). In FPP, the running task can be interrupted at an arbitrary location, i.e., suspended (blocked), and another task can become the running task. For FPP, preemption occurs when a task with a higher priority becomes ready while a task with a lower priority is running. After RMA, much research work has been done on FPP scheduling algorithms on uniprocessor environments [8, 10, 13, 29, 28, 19, 23, 41, 70, 33, 38, 39, 47, 49, 55, 56]. FPP scheduling algorithms have also been extended to multiprocessor and/or distributed environments [6, 5, 38, 47, 55, 73].

One important assumption of RMA is task independence [45], which implies that tasks do not communicate and they do not access exclusively-shared resources. However, this assumption is often impractical because tasks may interact or compete for exclusively-shared resources during their executions. A direct consequence of dependent tasks is that a task with a lower priority may hold some exclusively-shared resources when a task with a higher priority is ready. If the task with a higher priority also requires an exclusively-shared resource held by another blocked task with a lower priority, then a *priority inversion* occurs [58]. As a result, exclusive access

[2] A resource may not be exclusively-shared such as a read-only data file.

to shared resources may preclude the desired schedule to ensure tasks achieve their deadlines. To solve the problem, specific resource management protocols have been proposed [27, 61, 62, 65]. In addition, preemptions introduce context switches, whose cost may not be negligible in practice. For example, extra processor time and memory are required. Dealing with the cost of a context switch is discussed in Section 2.2.5. Interestingly, if preemption is disallowed, then appropriate access to exclusively-shared resources can be guaranteed and the cost of context switches introduced by preemptions can be saved, too. These observations suggest using fixed-priority non-preemption (FPNP) when possible.

FPNP is based on a fixed-priority assignment, with the added requirement that the execution of a task cannot be preempted by another task. Non-preemptive scheduling is important in real-time distributed applications such as on-line transaction processing. For example, each packet should be transmitted successfully or not. Otherwise, such a packet is considered useless or lost by the receiver.

Fixed-Priority with preemption threshold (FPPT) [74] fills the gap between FPP and FPNP, and it is a mixture of FPP and FPNP, where FPP and FPNP are the two boundary cases of FPPT. Each task has a pair of priorities: *regular priority* and *preemption threshold*, where the preemption threshold of a task is equal to or higher than its regular priority. The regular priority of a task is fixed and used to compete for the processor when the task enters a system. When a task starts to run, its running priority is upgraded to its preemption threshold. Only those tasks with regular priorities higher than the preemption threshold of the running task can preempt the execution of the running task. All preemption thresholds of a task set form a *preemption threshold assignment* or *assignment* for simplification. Similarly, all regular priorities of a task set form a *regular priority assignment*. If an assignment can make each task schedulable by FPPT, then the assignment is called *valid*. Based on the schedulability test for FPPT, a known algorithm [74] can find a valid preemption threshold assignment if the task set is schedulable by FPPT when the regular priority is predefined.

Finally, supporting both periodic tasks and aperiodic tasks is also very important. Most often, periodic tasks are scheduled by a well-known real-time scheduling algorithm such as RMA, EDF, and FPPT. After guaranteeing that all periodic tasks meet their deadlines, the unused time left by periodic tasks is used to schedule aperiodic tasks. The well-known approaches to manage such unused time are background server, polling server, slack stealing algorithm, and bandwidth reservation algorithm, which are discussed in Section 2.3. The reservation-based algorithm (RBA) is an important bandwidth reservation algorithm.

1.3 Terminology and Assumption

Periodic tasks are denoted as $\tau_1, \tau_2, \ldots$ and the jth job of task τ_i is denoted as $J_{i,j}$. Some of the research work enumerates all jobs of a task with the index starting from 0 and some of them with 1, which depends on personal taste and preference. In this thesis, the starting index for the research work is 0, but when discussing other research work, the starting index is clearly specified.

The time when a job enters a system is called its *ready time*. The *starting time* of a job is the time when the job starts to execute for the first time. The duration between the time when a job enters a system and when it finishes is called the *response time* of the job. In the lifetime of a system, the *worst case response time* of a task is equal to the maximal response time of all of its jobs.

The time when the first job of a task is ready is called the *phase* of the task. If all tasks in a task set have the same phase, they are called *in phase* and their phase can be considered to be 0.

The major cycle of a task set is denoted as T_{mc}, which is equal to the least common multiple of the periods of all periodic tasks in the task set. The *unit cycle* of a task set is denoted as T_{uc}, which is equal to the greatest common divisor of the periods of all the periodic tasks in the task set.

The time granularity in a real-time system determines the granularity of the time parameters of a task. For example, if a microsecond is the minimal time-unit supported by a clock in a real-time system, any part less than 1 microsecond is impossible. Practically, the time parameters of a task in a real-time system are presented as the number of smallest time-units of the clock in the system. To simplify discussion and calculation, a time value can be scaled to an integer of the fundamental time-unit. Therefore, without loss of generality, all periods of periodic tasks are assumed to be integers for convenience in this thesis. Furthermore, the time-unit to specify a periodic task is assumed to be much larger than the fundamental time-unit because it is generally true in practice. For example, the time-unit for a periodic task may be millisecond while the fundamental time-unit may be microsecond. In the following discussion, the time-unit is unspecified; instead, it is assumed that the time-unit used makes sense practically.

A periodic task is denoted as τ = (*D, T, C, P*), where *D* stands for the deadline, *T* the period, *C* the computation time, and *P* the phase. If the deadline is equal to the period, then τ = (*T, C, P*) is used. If the phase is 0, then τ = (*T, C*) is used. A periodic task set S with n periodic tasks is denoted as $S = \{\tau_1, \tau_2, \ldots, \tau_n\}$. For convenience, the deadline of task τ_i is denoted by D_i, the

period by T_i, the computation time by C_i, and the phase by P_i. For example, task $\tau = (7, 5, 2, 1)$ indicates a periodic task with deadline 7, period 5, computation time 2, and phase 1.

Given a periodic task set, if tasks neither communicate nor access exclusively-shared resources during their executions, then such a task set is called an *independent task set*.

A *harmonic periodic task set* satisfies the condition that for each task its period is an exact multiple of the period of every other task with a shorter period. For example, a task set composed of four tasks with periods 2, 4, 8, and 16 is harmonic. But a task set composed of four tasks with periods 2, 3, 6, and 12 is not harmonic.

RMA timing analysis is based on a worst case scenario as follows. At time 0, the first job of each task is ready. Such a scenario is called a *critical instant* [45]. For the RMA critical instant, periodic tasks are assumed to satisfy the following assumptions:

1. A periodic task has a hard deadline.
2. A periodic task's computation time cannot be greater than its period.
3. All periodic tasks are independent.
4. The worst case computation time of a periodic task is constant.

In addition, any aperiodic task does not have a hard, critical deadline. Two additional assumptions are as follows in this thesis:

1. A periodic task's deadline can be greater than its period.
2. Tasks τ_1, τ_2, . . ., τ_n are listed in decreasing order based on their fixed-priorities, the priority of task τ_i is equal to i. Note, this thesis follows the standard convention that the smaller a priority in value, the higher the priority.

Based on the assumptions, for task $\tau_i = (D_i, T_i, C_i, 0)$, $J_{i,j}$ is ready at time $T_i * j$ with period T_i, computation time C_i, phase 0, and deadline D_i. Given task τ_i, if FPPT is used, then its regular priority and preemption threshold are denoted by a pair of numbers (π_i, γ_i), where π_i stands for the regular priority and γ_i for the preemption threshold. By default, $\pi_i = i$. For a task set with n periodic tasks, the regular priority assignment is denoted as $\pi = \{\pi_1, \pi_2, \ldots, \pi_n\}$ and the preemption threshold assignment is denoted as $\gamma = \{\gamma_1, \gamma_2, \ldots, \gamma_n\}$. The assumption that each task has a unique regular priority is reasonable. Given a task set with predefined regular priority,

if two tasks have the same regular priority. The situation can be dealt with as follows. The tasks can be sorted in increasing order based on their importance breaking ties arbitrary. Based on the sorted result, a unique regular priority is assigned to each task. Thus, if two tasks have the same regular priority, it is easy to reassign a unique regular priorities to them.

Based on the definition of preemption threshold, the preemption threshold γ_i of task τ_i is between i and 1 inclusively. When a task set is schedulable by FPPT, there may exist multiple valid assignments. Among a group of valid assignments for a task set, if the preemption threshold of each task is minimal, such an assignment is called *minimal*; if the preemption threshold of each task is maximal, such an assignment is called *maximal*. The minimal and maximal assignments are formally defined in Section 3.5.2.

As mentioned, this thesis does not consider multiprocessor architectures, only uniprocessor architecture is considered. At the same time, sporadic tasks are not considered and only periodic and aperiodic tasks are considered. Extra assumptions are introduced in each section if necessary.

1.4 Objective and Contributions

The field of real-time scheduling is a mature research area, with many contributions over the past 30+ years covering theoretical and practical developments. This thesis dovetails with this large body of work by providing incremental advancements in two existing, commonly used algorithms, and demonstrates these advancements with other existing approaches through a newly developed real-time scheduling tool-kit. This work targets two of the most important problems in real-time scheduling:

1. How to effectively schedule a set of periodic real-time tasks on a uniprocessor computer (with some thought towards multiprocessor scheduling).
2. How to effectively schedule aperiodic real-time tasks among the periodic tasks to take advantage of any unused execution-time.

These two problems are core issues in real-time scheduling, and hence, this work contributes to both the fundamental theoretical and practical development of the field.

Specifically, this thesis takes the existing FPPT scheduler for scheduling periodic tasks and bandwidth reservation algorithm for scheduling aperiodic tasks, and extends each with new theoretical and practical developments. As well, a new real-time scheduling tool-kit was developed to

verify these new algorithmic approaches in a quick and precise manner, and to provide a mechanism to compare and contrast the new algorithms with other approaches. The results are a broader understanding of the two existing approaches, and a better implementation for scheduling real-time task sets. The deficiencies and/or problems of the existing research work in these three areas are explained.

1. FPPT

   Wang and Saksena present a schedulability test for FPPT [74] when the regular priority of each periodic task is predefined. They present an effective algorithm to compute the minimal assignment by starting from FPP, one of the boundary cases of FPPT; in addition, they present an effective algorithm [60] to compute the maximal assignment by starting from a valid assignment such as the minimal assignment.

   However, the original schedulability test for FPPT is incorrect [59]. In addition, when a task set is schedulable by FPPT, there may exist multiple valid preemption threshold assignments. A mechanism to generate additional valid preemption threshold assignments is not discussed. Furthermore, the relationship among valid preemption threshold assignments is not considered such as all valid assignments are delimited by a minimal and maximal assignment. Finally, computing the minimal and maximal assignments by starting from FPNP, the other boundary case of FPPT, is not discussed.

2. RBA

   RBA is one of the bandwidth reservation algorithms, in which a periodic task acts as a periodic server to service aperiodic tasks. When RMA is used to guarantee the deadlines of periodic tasks, Kang and Yi-Chieh [63] present an algorithm to calculate the reservation size.

   However, the algorithm is complicated and its correctness is suspect. In addition, the algorithm does not consider the maximal reservation bandwidth when other schedulers such as EDF and FPPT are used to guarantee the deadlines of periodic tasks.

3. tool-kit

   Though many real-time scheduling tool-kits are available, there is none that can satisfy the requirements to generate all of the examples or counter examples required by this thesis. One tool-kit is required to integrate different parts from different tool-kits.

   Another reason to develop such a tool-kit is to verify the correctness of the ideas that are guessed by the author during the research work. For example, when considering a sample

task set based on an even perfect number, generating the Gantt-chart of such a task set step by step in its first critical instant is required.

The research work of this thesis is based on the above problems and observations. The contributions of this thesis are as follows.

1. Prove that FPPT is robust under its critical instant (robustness is discussed in Section 2.1.2); provide theoretical support for the mechanism to generate more valid preemption threshold assignments when a task set is schedulable by FPPT; prove that all valid assignments for a task set are delimited by the minimal and maximal assignments; present algorithms to compute the minimal (maximal) assignment by starting from the maximal (minimal) assignment; present algorithms to compute the minimal and maximal assignments by starting from FPNP; and present the relationships among all algorithms to compute maximal and minimal preemption threshold assignments from both boundary cases of FPPT – FPNP and FPP. The research work in this thesis shows that the maximal threshold assignment can be calculated from either boundary case with equivalent complexity and the minimal and maximal threshold assignments can be calculated if the other is known.

2. Provide theoretical support for a new algorithm to calculate the maximal bandwidth reserved in RBA with RMA and EDF to guarantee the deadlines of periodic tasks, which is simpler and more effective than the known algorithm. When RBA is combined with FPPT, it can obtain a better reservation size than RMA but avoid the high run-time cost of EDF.

3. The creation of a small real-time scheduling tool-kit with a graphical interface to verify the research results for this thesis. The research work of this thesis does not focus on the tool-kit. Instead, the tool-kit is only a by-product of the research work. Though the tool-kit is ad hoc, full of personal tastes and preferences, it does contain some novel features that may be of interest in a larger commercial scheduling tool-kit.

1.5 Thesis Organization

Related work is presented in Chapter 2. In Chapter 3, FPPT is proved robust under its critical instant; a mechanism to generate additional valid preemption threshold assignments is presented; all valid assignments for a task set are delimited by the minimal and maximal assignments, which

is specified by a necessary condition; the algorithm to calculate the minimal and maximal assignments by starting from FPNP are presented and proved that they can be used to perform a schedulability test, too. In Chapter 4, a theoretical support is proved to compute the maximal reservation size when RMA is used in RBA. When RBA is combined with FPPT, an algorithm to search for the maximal assignment under which the maximal reservation size can be calculated is presented. In Chapter 5, a small real-time scheduling tool-kit to verify the research work in the thesis is discussed. Conclusion and further research work are discussed in Chapter 6.

Chapter 2

Related Work

Within the large corpus of literature on real-time scheduling, this chapter focuses on that material necessary to understand the subsequent new material presented in Chapters 3, 4, and 5.

2.1 Rate Monotonic Algorithm

Liu and Layland [45] formally proposed RMA to schedule real-time periodic tasks in a uniprocessor environment based on the RMA critical instant. Given a task set, the priority of a task is inversely proportional to its period under RMA. Alternatively speaking, the longer the period of a task, the lower the priority of the task.

Given a periodic task set $S = \{\tau_i = (T_i, C_i) : 1 \leq i \leq n\}$, $T_1 \leq T_2 \leq \ldots \leq T_n$, its processor utilization U is defined by Formula 2.1 [45].

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \tag{2.1}$$

The Liu and Layland's schedulability test calculates an expected processor utilization $U_e(n)$ with Formula 2.2 [45], where n is the number of tasks.

$$U_e(n) = n(2^{\frac{1}{n}} - 1) \tag{2.2}$$

When $n \to \infty, U_e(n) \to 0.69$. Given a task set of n tasks, if $U_e(n) \geq U$, then it is schedulable by RMA. This schedulability test is sufficient with time efficiency $O(n)$, but pessimistic. When a

task set does not satisfy the condition or it fails this schedulability test, it may still be schedulable by RMA. For example, periodic tasks $\tau_1 = (30, 9)$, $\tau_2 = (50, 15)$, and $\tau_3 = (70, 14)$ fail this RMA schedulability test because $U_e(3) \approx 0.78$ and $U = 0.8$. But in fact, these tasks are schedulable by RMA based on other sufficient and necessary schedulability tests.

Since Liu and Layland proposed RMA, many other approaches have been proposed to improve the performance of RMA and/or extend RMA from a uniprocessor to a multiprocessor environment [41, 8, 15, 33, 70, 19, 23, 13, 29, 28, 38, 56]. In uniprocessor environments, the following three approaches have been used to improve the performance of RMA based on improving the schedulability test and/or converting a task set into another task set under which a schedulability test is potentially easier to solve. As most real-time scheduling problems are NP-Complete, no polynomial algorithm is currently known for RMA. The cost for the current necessary and sufficient RMA schedulability tests are pseudo-polynomial.

1. improve schedulability test

   (a) sufficient and necessary schedulability test

   The Liu and Layland's schedulability test is only sufficient, the first sufficient and necessary schedulability test for RMA was proposed by Lehoczky et al [41], which is rephrased in detail in Section 2.1.1.

   Audsley et al [8] proposed a sufficient and necessary schedulability test based on the response time, denoted as R_i, of each job in the first critical instant. Hence, the response time of $J_{i,0}$ is checked for $1 \leq i \leq n$. A task set is schedulable by RMA if and only if $R_i \leq T_i$ for $1 \leq i \leq n$. This approach is also explained in detail in [15]. Furthermore, this research work is based on Joseph and Pandya's research work in [33].

   Tjandrasa et al [70] proposed another sufficient and necessary schedulability test based on the exact amount of execution time, denoted as E_i, required by those jobs of tasks $\tau_1, \tau_2, \ldots, \tau_i$ ready over $[0, T_i]$ for $1 \leq i \leq n$. A task set is schedulable by RMA if and only if $E_i \leq T_i$ for $1 \leq i \leq n$.

   The time efficiency of these approaches is pseudo-polynomial. Up to now, there is no RMA schedulability test that is sufficient and necessary with polynomial time efficiency.

   (b) increase the threshold, i.e., moving the processor utilization from 0.69 to 1

   The threshold in the Liu and Layland's schedulability test is too low as it is only

0.69. If the threshold can be increased, then the range of task sets that do not fail the corresponding RMA schedulability test is expanded. Thus, many task sets may still be schedulable by RMA if they fail the Liu and Layland's schedulability test. Research work along this line has been done [19, 23, 13]. Like the Liu and Layland's schedulability test, these schedulability tests are only sufficient and the time efficiency is pseudo-polynomial.

2. convert task set
   A task set can be converted into another task set under which a schedulability test is easier to perform. One approach is based on the harmonic task set, which is schedulable if and only if its processor utilization is not greater than 1 [72]. Therefore, if a task set can be converted into a schedulable harmonic task set, then the original task set is schedulable. Research work [29, 28] along this line has been done. Though its time efficiency is polynomial, this approach may fail. First, a successful conversion may be unavailable. Second, the conversion may change the schedulability of the original task set as a converted task set may be unschedulable but the original task set is schedulable by RMA.

3. try both approaches
   Lauzac et al [38] proposed a combined approach to improve RMA scheduling. First, a given task set is converted into a new task set such that if the original task set is schedulable, then the converted task set is also schedulable. Then an improved schedulability test works on the converted task set. Park et al [56] considered supporting tasks with unfixed computation time. In Park's research work, a minimum and maximum computation time for a task set are given. Then a better schedulability test works on the task set. However, the time efficiency for both approaches is pseudo-polynomial.

As RMA priority assignment can be used to assign the regular priorities for a task set in FPPT and is used in a known RBA to compute the maximal reserved bandwidth, a simple introduction to one of its sufficient and necessary schedulability tests can help understanding FPPT and RBA.

2.1.1 Sufficient and Necessary RMA Schedulability Test

Lehoczky et al [41] first proposed a sufficient and necessary schedulability test for RMA based on the RMA critical instant. This schedulability test calculates an expected processor utilization

with Formula 2.8, which depends on Formulae 2.3, 2.4, 2.5, 2.6, and 2.7. All of these formulae are presented by Lehoczky et al [41].

$$W_i(t) = \sum_{j=1}^{i} C_j \cdot \left\lceil t/T_j \right\rceil \tag{2.3}$$

$$L_i(t) = W_i(t)/t \tag{2.4}$$

$$L_i = \min_{\{0<t\leq T_i\}} L_i(t) \tag{2.5}$$

$$S_i = \{k \cdot T_j | j = 1, \ldots, i; k = 1, \ldots, \lfloor T_i/T_j \rfloor\} \tag{2.6}$$

$$L_i = \min_{\{t\in S_i\}} L_i(t) \tag{2.7}$$

$$L = \max_{\{1\leq i\leq n\}} L_i \tag{2.8}$$

where $W_i(t)$ gives the cumulative amount of processor time required by those tasks ready over $[0, t]$. $L_i(t)$ is the average processor utilization of those tasks ready over $[0, t]$. Formula 2.5 calculates the minimal value on the continuous range $(0, T_i]$. The numbers defined in Formula 2.6 are called *check points*. Formula 2.7 calculates the same minimal value on discrete values, i.e., check points, over $(0, T_i]$. Formulae 2.6 and 2.7 simplify the calculation of L. A task set is schedulable by RMA if and only if $L \leq 1$. This observation is formally presented in the following theorem named Theorem **ERMA2** [1]:

Theorem **ERMA2** [41][2]: Given periodic tasks $\tau_1, \ldots, \tau_n$,

1. τ_i can be scheduled for all task phasings using RMA if and only if $L_i \leq 1$.
2. The entire task set can be scheduled for all task phasings using RMA if and only if $L \leq 1$.

2.1.2 RMA Robustness

Mok [51] defines the terms PFP (Preemptive Fixed-Priority) and NPFP (Non-Preemptive Fixed-Priority), which correspond to FPP and FPNP, respectively. Mok focuses on the robustness of

[1] "**ERMA**" indicates the Exact version of Rate Monotonic scheduling Algorithm. Theorem **ERMA2** corresponds to Theorem 2 in [41], which is the second version of the same theorem. That is why 2 appears in the name of the theorem.

[2] The index of the first job is 1 in this research work. A minor change in format is made.

RMA priority assignment for FPP and FPNP. That is, given a task set schedulable by a fixed-priority assignment, if the computation time of a task is reduced and the fixed-priority assignment still preserves the schedulability, then the fixed-priority assignment is *robust*. Mok proves that the RMA priority assignment for FPP is robust but it is not robust for FPNP based on the RMA critical instant. For example, given three periodic tasks $\tau_1 = (3, 1)$, $\tau_2 = (6, 1)$, and $\tau_3 = (12, 4)$, they are schedulable by RMA priority assignment for FPP and FPNP based on the RMA critical instant. If the computation time of the second task is reduced to 1, the task set is still schedulable by RMA priority assignment for FPP, but is unschedulable by RMA assignment for FPNP based on the RMA critical instant. The details are as follows. $J_{1,0}$ starts to execute at time 0 and finishes at time 1, followed by $J_{2,0}$ that starts to execute at time 1 and finishes at time 2. After $J_{2,0}$ finishes, $J_{3,0}$ starts to execute. Due to non-preemption, $J_{3,0}$ continues to execute until it finishes at time 6 though $J_{1,1}$ is ready at time 3 with a higher priority than $J_{3,0}$. $J_{1,1}$ can only start to execute at time 6, which is its deadline. Thus, $J_{1,1}$ misses its deadline and the task set is unschedulable.

The robustness of RMA indicates that if a task set is schedulable by RMA, then any subset of the task set is schedulable by RMA. The reason is as follows. As RMA is robust, when a task set is schedulable by RMA, decreasing the computation of any task cannot change the schedulability of the task set. When the computation time of a task is decreased to 0, resulting in that the task is removed from a task set, the remaining tasks are still schedulable by RMA as it is robust.

The robustness of RMA indicates that its schedulability test can be performed incrementally. Given periodic tasks $\tau_i = (T_i, C_i)$ for $1 \leq i \leq n$ and $T_1 \leq T_2 \leq \ldots \leq T_n$, the schedulability test can be performed on τ_1, τ_2, . . ., τ_i for $1 \leq i \leq n$. If tasks τ_1, τ_2, . . ., τ_i are unschedulable by RMA for $1 \leq i \leq n$, then the whole task set is unschedulable and further schedulability testing is unnecessary. Otherwise, the schedulability test can be performed on τ_1, τ_2, . . ., τ_i, τ_{i+1} until either the whole task set is schedulable or τ_1, τ_2, . . ., τ_i, τ_{i+1} are unschedulable. The latter case indicates that the whole task set is unschedulable.

2.2 Fixed-Priority with Preemption Threshold

FPPT was first introduced by Express Logic in the ThreadX real-time operating system [37]. Preemption threshold allows a task to only prevent preemption of tasks up to a specified threshold priority. For example, during the execution of a task, if a ready task has a regular priority higher than the preemption threshold of the running task, then the former can still preempt the execution of the running task; otherwise, it cannot. When the preemption threshold of each task is equal to

its regular priority in a task set, then FPPT degrades to FPP. When all preemption thresholds are the same as the highest regular priority in a task set, then FPPT degrades to FPNP. RMA may be applied to assign the regular priorities for a task set. Unlike RMA, FPPT allows a task to prohibit some preemptions by assigning an appropriate preemption threshold to the task.

2.2.1 Background

The following is work that moves out of general preemption scheduling (FPP) to non-preemption (FPNP) and then restricted preemption (FPPT). When preemption is required, it should be limited [31]. For example, the number of preemptions is minimized, to reduce its cost.

Kweon and Shin [36] guarantee an end-to-end delay by applying a non-preemptive rate-monotonic priority scheduling policy at each network switch on the path in a real-time communication system. However, they did not consider its strict timing analysis. Parks and Lee [57] propose a dynamic, real-time execution model inspired by multi-threaded data-flow architectures by applying non-preemptive RMA. However, the timing analysis is pessimistic. Bate and Burns [12] extend the schedulability analysis of FPP in real-time systems to the FPNP environment.

Fixed-priority with mutual-preemption task groups [22] is used to partition a task set into several subsets. Within a task subset, each task cannot preempt any other task in the same group so the tasks in the same subset can share one execution stack and resources. This issue is significant when the amount of memory in a microprocessor is small such as 512k bytes, and the speed of the processor is slow. Each task has two priorities, called base priority and dispatch priority respectively, where the latter is equal to the highest base priority in the same subset. The base priority can be assigned using RMA or the deadline monotonic algorithm (DMA) [43, 7, 10]. All tasks in the same subset share the same dispatch priority. When a task starts to run for the first time, its running priority is set to its dispatch priority. Only a task from another subset, whose base priority is higher than the dispatch priority of the current running task, can preempt the execution of the latter. The number of subsets is called the number of preemption levels. The algorithm to find the minimal number of preemption levels is complicated. A resource management protocol is still required due to preemption. The schedulability test is also based on a critical instant that is different from the RMA critical instant. The purpose of the research work in [22] is to reduce the number of execution stacks and the number of preemptions so as to save memory and processor time.

The idea of the schedulability test for FPPT in [74] is correct. However, the actual schedu-

lability test is incorrect [59] as it violates one condition of level-i busy period [39] (discussed formally in Section 2.2.2). The preemption threshold assignment algorithm [74] only finds the minimal valid preemption threshold assignment and the algorithm [60] computes the maximal preemption threshold assignment. However, the algorithm to compute the maximal preemption threshold assignment starts from a valid assignment.

2.2.2 FPP Timing Analysis

As the timing analysis for FPNP [57, 12] and FPPT [74] is based on the idea for the timing analysis of FPP [39], rephrasing the research result of the latter can be helpful to understand the former. FPP assumes at time 0, the first critical instant occurs, implying that all tasks are ready at time 0. For example, RMA critical instant follows this assumption.

Lehoczky [39][3] introduces the concept of a level-i busy period when scheduling periodic tasks with arbitrary deadlines and a fixed-priority scheduling algorithm. A level-i busy period is a time interval $[a, b]$ within which jobs of priority i or higher are processed throughout $[a, b]$ but no jobs of priority i or higher are processed in $(a - \epsilon, a)$ or $(b, b + \epsilon)$ for sufficiently small $\epsilon > 0$. Note, during a level-i busy period, no job with priority lower than i can execute, which indicates jobs with priority lower than i can only start to execute at the end of a level-i busy period or before the beginning of a level-i busy period if required. Suppose $[a, b]$ is a level-i busy period, at time b, all jobs with priorities equal to or greater than i ready before b must have finished. Furthermore, there should be no jobs with priorities equal to or greater than i ready at time b. Consider a level-i busy period $[0, b]$ starting from the first critical instant, two conditions must be satisfied.

1. Within $[0, b]$, no task with priority lower than τ_i can execute.
2. At time b, all tasks with priorities equal to or greater than τ_i have finished their executions.

Given periodic tasks τ_1, τ_2, ..., τ_n with arbitrary deadlines, they are schedulable by FPP if and only if [39][4]

$$\max_{1 \le m \le n} \max_{1 \le k \le N_m} W_m(k, (k-1)T_m + D_m) \le 1 \tag{2.9}$$

[3] In [39], there is an error in Example 3. The level-2 busy period should be $[0, 694]$ instead of $[0, 696]$.

[4] The index of the first job is 1 in this research work.

where

$$W_m(k,x) = \min_{t \le x}\Big(\Big(\sum_{j=1}^{m-1} C_j \Big\lceil \frac{t}{T_j} \Big\rceil + kC_m\Big)/t\Big) \tag{2.10}$$

and $N_m = min\{k|W_m(k,kT_m) \le 1\}$. In $W_m(k,(k-1)T_m + D_m)$, m and k stand for $J_{m,k}$, the *kth* job of task τ_m and $(k-1)T_m + D_m$ the deadline of $J_{m,k}$. If task τ_m is schedulable, then there exists a time t that is equal to the required computation time for the first k jobs of task τ_m and all jobs ready over $[0,t]$ with priorities higher than τ_m and $t \le (k-1)T_m + D_m$. $W_m(k,x)$ calculates the minimum processor utilization over $[0,x]$ based on such a t for a given m, k, and x. $W_m(k,(k-1)T_m + D_m) \le 1$ guarantees that $J_{m,q}$ cannot miss its deadline and $W_m(k,kT_m) \le 1$ guarantees the length of the first level-m busy period is reached, no further checking is required. In other words, given tasks τ_1, τ_2, ..., τ_m, the Lehoczky's schedulability test checks all jobs of these tasks within the first level-m busy period starting from time 0. If any job misses its deadline, then the task set is unschedulable. Otherwise, the task set is schedulable. Correspondingly, on the left hand side of Formula 2.9, the inner maximization guarantees that the worst case response time of task τ_m is not greater than its deadline and the outer maximization guarantees that the worst case response time of each task is not greater than its deadline.

Lehoczky also proves that the longest response time for a job of task τ_i occurs during a level-i busy period initiated at the critical instant, $I_1 = \ldots = I_i = 0$, where I_i is the phase of task τ_i (see Theorem 1 in[39]).

Tindell [69] rephrases the Lehoczky's schedulability test. The research work in [69] defines $w_{i,q}$ as the finishing time of $J_{i,q}$, and r_i as the worst case response time of task τ_i. Then[5]

$$w_{i,q} = (q+1)C_i + \sum_{j=1}^{i-1} \Big\lceil \frac{w_{i,q}}{T_j} \Big\rceil C_j \tag{2.11}$$

$$r_i = \max_{q=0,1,2,\ldots} (w_{i,q} - qT_i) \tag{2.12}$$

The above iteration over increasing values of q can stop if $w_{i,q} \le (q+1)T_i$. A task set is schedulable by FPP if and only if $r_i \le D_i$, $1 \le i \le n$. Both timing analyses for FPP are sufficient and necessary.

[5]The index of the first job is 0 in this research work.

2.2.3 Applying Level-i Busy Period in FPNP

George, Rivierre, and Spuri [26] established the computation of the worst case response time of tasks in the context of FPNP. Given tasks τ_1, τ_2, ..., τ_n, which are listed in decreasing order by their fixed priorities, the worst case response time of any task τ_i is calculated by[6]

$$L_i = \max_{j \in lp(i)} \{C_j - 1\} + \sum_{j \in hp(i) \cup \{i\}} \left\lceil \frac{L_i}{T_j} \right\rceil C_j. \tag{2.13}$$

$$w_{i,q} = qC_i + \sum_{j \in hp(i)} \left(1 + \left\lfloor \frac{w_{i,q}}{T_j} \right\rfloor\right) C_j + \max_{k \in lp(i)} \{C_k - 1\}, \tag{2.14}$$

$$r_i = \max_{q=0,1,\ldots,\lfloor L_i/T_i \rfloor} \{w_{i,q} + C_i - qT_i\} \tag{2.15}$$

$hp(i)$ and $lp(i)$ are index sets of tasks with priorities higher and lower than task τ_i, respectively. L_i is the length of the level-i busy period. $w_{i,q}$ is the time when $J_{i,q}$ starts to execute. Equation 2.13 computes the length of the level-i busy period, equation 2.14 the starting time of $J_{i,q}$, and equation 2.15 the worst case response time of task τ_i. The worst case response time is based on the assumption that the clock resolution is 1 and it occurs when a task (if any) τ_j, $j \in lp(i)$ with longest computation time just starts to execute before the critical instant. Then task τ_i can be blocked by τ_j with $C_j - 1$, which corresponds to the first item in equation 2.13. In practice, this minus one only makes sense when the parameters for a task are based on the clock tick. The second item in equation 2.13 corresponds to the amount of time executed by those tasks with priorities equal to or higher than τ_i. George et al [26] claim their schedulability test is sufficient and necessary.

2.2.4 Applying Level-i Busy Period in FPPT

In FPP, when a task enters a system, it must wait if a task with a higher priority is running. During the execution of a task, if another task with a higher priority becomes ready, the running task must be preempted and wait until all other tasks with higher priorities finish. For convenience, this kind of waiting is called *interference* from tasks with higher priorities.

In FPPT, the kind of waiting for a task is also called interference from other tasks with regular priorities higher than the preemption threshold of the task. For a task τ_i in FPPT, in addition to

[6]The index of the first job is 0 in this research work.

the interference caused by other tasks with regular priorities higher than γ_i, τ_i may wait when another task τ_j with a lower regular priority is running. For example, when task τ_i is ready, task τ_j with a lower regular priority is running and τ_i must wait. This delay occurs because the running task τ_j cannot be preempted by task τ_i due to $\pi_i < \pi_j$ but $\pi_i \geq \gamma_j$, so τ_i must wait. This kind of waiting is called *blocking time* from a task with a lower regular priority and task τ_i is called *being blocked* by task τ_j. Wang and Saksena [74] apply the level-i busy period to calculate the worst case response time in FPPT. However, their schedulability test is incorrect [59] as it may not consider all jobs in the first level-i busy period. The reason why all jobs must be checked is the first job of a task in the first level-i busy period may not get the worst case response time for the task. The corrected schedulability test was presented by Regehr [59]. The formulae in [59] are rephrased[7] or simplified, because Regehr's research work considers release jitter [67], which is not covered in this thesis.

$$B_{(\tau_i)} = \max_{j>i, \gamma_j \leq \pi_i < \pi_j} C_j \tag{2.16}$$

$$L_i = B_{(\tau_i)} + \sum_{\forall j, \pi_j \leq \pi_i} \left\lceil \frac{L_i}{T_j} \right\rceil C_j \tag{2.17}$$

$$S_{i,q} = B_{(\tau_i)} + q \cdot C_i + \sum_{\forall j, \pi_j < \pi_i} \left(1 + \left\lfloor \frac{S_{i,q}}{T_j} \right\rfloor\right) \cdot C_j \tag{2.18}$$

$$F_{i,q} = S_{i,q} + C_i + \sum_{\forall j, \pi_j < \gamma_i} \left(\left\lceil \frac{F_{i,q}}{T_j} \right\rceil - \left(1 + \left\lfloor \frac{S_{i,q}}{T_j} \right\rfloor\right)\right) \cdot C_j \tag{2.19}$$

$$R_i = \max_{q=0,1,\ldots,\lfloor L_i/T_i \rfloor} (F_{i,q} - qT_i) \tag{2.20}$$

Formula 2.16 computes the maximal blocking time for a task from any other task with a lower regular priority. If a task cannot be blocked by another task with a lower regular priority, then its maximal blocking time is 0. Note, the blocking time from a task with a lower regular priority was originally introduced in the priority inheritance protocol. In other words, Formula 2.16 is derived from [62].

Formula 2.17 computes the length of the first level-i busy period. The initial value for the recursive calculation is $B_{(\tau_i)} + \sum_{\forall j, \pi_j \leq \pi_i} C_j$ no matter whether $B_{(\tau_i)}$ is equal to or greater than 0.

[7] Rephrasing follows the standard convention that the larger a numeric value, the lower its priority.

If the iterative computation converges, L_i converges to a stable value. If the iterative computation diverges, and L_i increases, then the calculation can stop when $L_i > T_{mc}$, which indicates the task set is unschedulable.

Formula 2.18 computes the starting time of $J_{i,q}$. Before $J_{i,q}$ can start to run, all jobs of τ_1, τ_2, ..., τ_{i-1} ready over $[0, S_{i,q}]$ must finish before $S_{i,q}$, so do the first q jobs of τ_i in addition to the blocking task with a lower regular priority. The initial value for $S_{i,q}$ is $q \cdot C_i + \sum_{\forall j, \pi_j < \pi_i} C_j + B_{(\tau_i)}$.

Formula 2.19 computes the finishing time of $J_{i,q}$. After $J_{i,q}$ starts to execute, its running priority is upgraded to its preemption threshold, and τ_i can only be preempted by any task τ_h ready over $(S_{i,q}, F_{i,q})$ where $\pi_h < \gamma_i$. Hence, $F_{i,q}$ is equal to its starting time plus its computation time and all the computation times of those tasks having regular priorities higher than γ_i with jobs ready over $(S_{i,q}, F_{i,q})$. At time $S_{i,q}$, no job of task τ_j with $\pi_j < \gamma_i$ is ready, hence the number of jobs of task τ_j ready before $S_{i,q}$ is equal to $\lceil \frac{S_{i,q}}{T_j} \rceil$. The initial value for $F_{i,q}$ is $S_{i,q} + C_i$.

Formula 2.20 computes the worst case response time of τ_i. As the response time of $J_{i,q}$ is equal to $F_{i,q} - qT_i$, then the worst case response time of τ_i can be calculated.

For all tasks τ_i, $1 \leq i \leq n$, if $R_i \leq D_i$, then the task set is schedulable by FPPT; otherwise, the task set is unschedulable. In [74], Wang and Saksena present an algorithm to assign preemption thresholds based on the computation of worst case response time.

The reason why all jobs of a task in the first level-i busy period must be checked to compute the worst case response time of the task is that in the first level-i busy period, based on the FPPT critical instant, there may be multiple jobs for each task and the first job of each task may not get the longest response time. Example 3 in [39] presents such a scenario for FPP (pay attention to the footnotes on page 20), one boundary case of FPPT. When performing a schedulability test on tasks $\tau_1 = (70, 70, 26)$ and $\tau_2 = (120, 100, 62)$ with RMA, a kind of FPP, the response time of $J_{2,0}$ is 114 but the response time of $J_{2,4}$ is 118, which shows that task τ_2 gets worse response time at $J_{2,4}$ than at $J_{2,0}$. This example shows that the first job of a task in the first level-2 busy period does not get the longest response time. As FPPT is also based on the level-i busy period to compute the worst case response time of each task, it must also check all jobs of a task in the first level-i busy period. Similarly, FPNP, the other boundary case of FPPT, must perform the same check.

The related research work on FPPT can be summarized by Figure 2.1. The research work in [39] introduced the idea of level-i busy period based on the RMA critical instant, which is rephrased by the research work in [69]. The research work in [37] introduces the idea of pre-

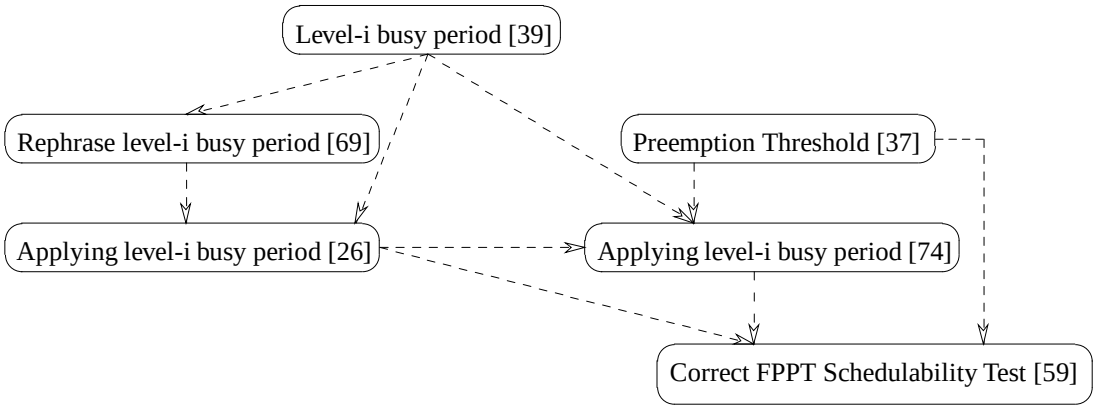


Figure 2.1: Current Research Results

emption threshold. The research work in [26] applied the level-i busy period for FPNP based on a critical instant different from the RMA critical instant and modified one condition of the level-i busy period. In the research work [26], at most one job with a regular priority lower than τ_i can execute within a level-i busy period. The corresponding schedulability test is sufficient and necessary. The research work in [74] applied the idea of the level-i busy period and preemption threshold based on a critical instant similar to [26]. However, its schedulability test is incorrect. The research work in [59] presented a correct schedulability test when the release jitter is also considered, which can be simplified to a correct schedulability test without considering the release jitter.

2.2.5 Cost of Context Switch

In practice, the cost of a context switch for independent periodic tasks is not negligible as a context switch requires extra processor time and memory requirement. Thus, a real-time scheduling algorithm must deal with the effect introduced by context switches. There is a simple rule of thumb [18, 34] to deal with the cost of context switches for periodic tasks. Suppose the cost of a context switch in a system is C_s, then each periodic task is affected by extra time equal to $2C_s$. Thus, the effect of context switches on a periodic task can be included in a schedulability test by increasing the computation time of each periodic task by an amount $2C_s$.

2.3 Scheduling Aperiodic Tasks

Scheduling both periodic tasks and aperiodic tasks can be very important in a real-time system. While periodic tasks have fixed arrival times and hard deadlines, aperiodic tasks have random arrival times and soft deadlines usually. A scheduling algorithm for both types of tasks must guarantee the deadlines of the periodic tasks and should shorten the response time of aperiodic tasks. In general, such a scheduler is based on a well-known algorithm for scheduling periodic tasks such as RMA, EDF, and FPPT to guarantee the deadlines of periodic tasks. The unused time left by periodic tasks is used to schedule aperiodic tasks and attempts are made to use this fixed time resource to shorten the response time of aperiodic tasks. Background server, polling server, bandwidth reservation algorithm [46], and slack stealing algorithm [40] are typical approaches that use the unused time left by periodic tasks to schedule aperiodic tasks. The first three approaches use RMA and the last approach uses EDF to guarantee the deadlines of the periodic tasks. Each of the typical approaches for scheduling aperiodic tasks is outlined as follows.

A background server services aperiodic tasks whenever the processor is not used by periodic tasks. That is, should a job finish early or there is no job to dispatch at this time, an aperiodic task is dispatched until the next periodic job must be scheduled. There is no change to the periodic task set nor the corresponding scheduling algorithm for the periodic tasks. However, when the workload of periodic tasks is heavy, an aperiodic task may encounter a long delay before starting execution and it may be interrupted frequently by periodic tasks during its execution, so its response time may be long.

A polling server is a periodic task scheduled to service aperiodic tasks. The scheduling algorithm of the periodic tasks performs the schedulability test on all periodic tasks including the polling server, i.e., the polling server extends the periodic task set to access the unused time, and hence, a polling server can only be as effective as the underlying periodic scheduler to access the unused time. Furthermore, it cannot take advantage of unused time resulting from a job finishing early because the schedulability test requires a fixed computation time. The processor time assigned to the polling server can then be used to service aperiodic tasks. Typically, a polling server is assigned the highest priority by a priority-based scheduler. For example, the period of a polling server is the shortest if RMA is used to schedule periodic tasks. At the beginning of execution of each instance of a polling server, it checks the aperiodic task queue. If the aperiodic task queue is empty, it stops immediately and it is not made ready to execute and does not check the aperiodic task queue again until its next period. Thus, aperiodic tasks arriving during the middle of one job of the polling server cannot start to execute until the next job of the polling server. If the aperiodic

task queue is not empty, the polling server uses its computation time to service aperiodic tasks until it consumes all of its computation time or all ready aperiodic tasks finish. The weak point of a polling server is that an aperiodic task may have to wait for almost one period of the polling server if it enters the system immediately after the beginning of the execution of one instance of the polling server.

A bandwidth reservation algorithm also creates a periodic task, called periodic server, to service aperiodic tasks. Like a polling server, it extends the periodic task set, so the underlying periodic scheduler must include it in the schedulability test. The computation time of the periodic task is called the bandwidth to be reserved. Typically, the periodic server has the shortest period. Thus, it has the highest priority based on RMA. At the beginning of execution of each instance of the periodic server, it checks the aperiodic task queue. If the aperiodic task queue is empty, it suspends immediately and tries to keep its computation time as long as possible in each period. For example, other ready periodic tasks may start to execute earlier and the computation time of the periodic server is postponed. However, the computation time of the periodic server cannot be accumulated for the next period, and hence, the available computation time of a periodic server decreases to 0 at the end of its period. If the aperiodic task queue is not empty, the periodic server uses its computation time to service aperiodic tasks until it consumes all of its computation time or all ready aperiodic tasks finish. If an aperiodic task enters the system in the middle of one instance of the periodic server, it can start to execute if the computation time of the periodic server is still available. If there is still part of its computation time available after all aperiodic tasks finish, the periodic server still tries to keep its remaining computation time as long as possible. Therefore, it improves the response time of aperiodic task execution by preserving its computation time as late as possible within its period so that an aperiodic task entering the system in the middle of one instance of the periodic server may start its execution with the computation time left. Deferrable server [42], sporadic server [64], and RBA [63] are typical bandwidth reservation scheduling algorithms belonging to this category except that they have different approaches to preserve and consume the computation time of the periodic server.

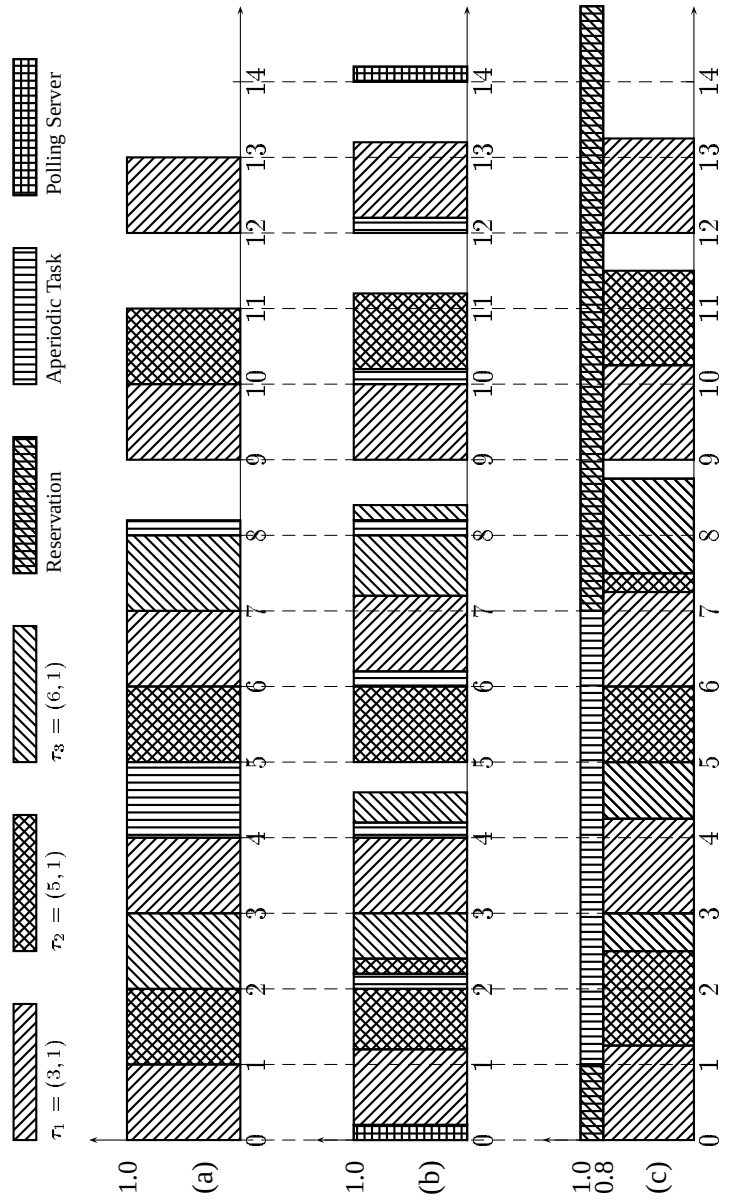
A slack stealing algorithm does not create a periodic task to service aperiodic tasks. First, it uses EDF to perform a schedulability test on the periodic tasks and makes sure all periodic tasks can meet their deadlines. Second, whenever an aperiodic task is ready, it tries to postpone the execution of periodic tasks and service the aperiodic task as long as possible without making any periodic task miss its deadline. Even though the implementation of this dynamic approach is complicated and the run-time cost is expensive, it is optimal with respect to the response time of

aperiodic tasks [40].

Figure 2.2 shows the scheduling results of three periodic tasks scheduled by RMA and one aperiodic task with background server, polling server, and bandwidth reservation algorithms, respectively. The three periodic tasks are $\tau_1 = (3, 1)$, $\tau_2 = (5, 1)$, and $\tau_3 = (6, 1)$. The aperiodic task arrives at time 1 with computation time 1.2. Part (a) is the background server; part (b) is the polling server with period 2 and computation time 0.2; part (c) is the bandwidth reservation algorithm with reserved bandwidth 0.2 in each time unit. The aperiodic task starts to execute at time 4 and finishes at time 8.2 for the background server; 2 and 12.2 for the polling server; 1 and 7 for the bandwidth reservation algorithm. Note, some of the periodic task execution has been subdivided into several pieces due to preemption by RMA. The order of starting time of the aperiodic task from the earliest to the latest is bandwidth reservation algorithm, polling server, and background server; the order of finishing time of the aperiodic task from the earliest to the latest is bandwidth reservation algorithm, background server, and polling server. For polling server and bandwidth reservation algorithm, there is still unused time that can be used to serviced aperiodic tasks to further shorten their response times. In part (b), it can be verified that the finishing time of the aperiodic task is also 8.2 with the addition of a background server. In part (c), the aperiodic task finishes before any unused time is available, which can be managed by a background server. Given that the bandwidth reservation algorithm, in general, gives the better response time for aperiodic tasks, it is selected for further development in this thesis.

2.4 Reservation-Based Algorithm

Among the different approaches for implementing bandwidth reservation algorithm such as deferrable server, sporadic server, and RBA, RBA is one important algorithms to schedule both periodic and aperiodic tasks [63, 46]. Like other bandwidth reservation algorithms, RBA applies RMA to schedule periodic tasks and aperiodic tasks are scheduled by utilizing the processor time unused by periodic tasks. RBA creates a periodic server to service aperiodic tasks. The computation time of the periodic server is called the *reservation size*, denoted as R_s, which is equal to the bandwidth to be reserved in a bandwidth reservation algorithm. RMA is applied to schedule all periodic tasks including the periodic server. The key point in RBA is to maximize the reservation size R while guaranteeing the deadlines of all periodic tasks. In RBA, the expected maximal R_s is called the least upper bound of R_s (R_{lub}). Kang and Yi-Chieh [63] presented an algorithm to calculate R_{lub} and another algorithm to perform the RMA schedulability test with parameter R_s and the period of the periodic server is equal to the unit cycle. The schedulability test checks



(a) background server (b) polling server (c) bandwidth reservation

Figure 2.2: Scheduling Periodic and Aperiodic Tasks

all jobs in the first major cycle, if any job misses its deadline, then the task set is unschedulable; otherwise, the task set is schedulable. This exhaustive computation of checking all jobs in the first major cycle is called *physical scheduling* [43], which is not a good approach. There are two major weak points in the algorithm [63] to compute R_{lub} though the idea of the algorithm is reasonable.

1. The algorithm provides clear steps to compute R_{lub} when the number of periodic tasks is 2. However, when the number of periodic tasks is greater than 2, the authors did not provide clear steps to compute R_{lub}. Essentially, the pseudo-code for the algorithm and its explanation are insufficient to compute R_{lub}.
2. As mentioned by the authors, the R_{lub} derived by the algorithm may not be maximal as only 4 iterations of computation in one major cycle are used; although they claim this R_{lub} is close to 95% of the maximal value.

Therefore, the usefulness of the algorithm is suspect and a new algorithm is required to compute the maximal R_{lub} when RMA is used. As RMA is a special case of FPP, which is a special case of FPPT, the range of schedulable task sets by FPPT is larger than RMA. When RBA is combined with FPPT, it can obtain a not worse reservation size than RMA. Up to now, no research work has been done in this area.

2.5 Current Status in Real-Time Scheduling Tool-Kits

As an exact mathematical model for a real-time system is very difficult to obtain and analyze, simulation is often used to help system designers and programmers understanding a real-time system. A number of real-time scheduling tool-kits have been developed to help real-time system designers and programmers in verification and testing of task sets and scheduling algorithms. Martin Naedele [53] presented a survey of real-time scheduling tool-kits. Table 2.1 substantially extends the survey in Martin Naedele [53] to summarize the current status of real-time scheduling tool-kits based on the features different tool-kits provide. In the table, an empty entry indicates it is unknown or not mentioned by the original authors. Among them, DET/SAT/SIM (Det), PERTS SAT (Perts), and DTRESS/PERTSSim (Dtress) are the most comprehensive.

Most of the tool-kits focus on the performance of the whole system. Some of them also consider the flexibility of changing components in the system such as schedulers, resource management protocols, and system architectures. Based on their functionalities, these scheduling

features\tools	Ast	Det	Perts	Dtress	Saw	After	Brux	Org	Cai	Stress	123	GAST
multi-CPU	Yes			Yes						Yes		Yes
distributed	Yes	Yes		Yes	Yes					Yes		Yes
periodic task		Yes	Yes	Yes		Yes				Yes	Yes	Yes
aperiodic task		Yes	Yes	Yes							Yes	
sporadic task				Yes								
PIP		Yes	No	No		Yes		No			Yes	
PCP		Yes	Yes	Yes		Yes				Yes		
DCP		Yes	No	No								
SRP		Yes	No	No								
SBP		No	Yes	Yes								
RMA	Yes	Yes	Yes	Yes				Yes		Yes	Yes	
DMA	No	Yes	Yes	Yes						Yes		
EDF	Yes	Yes	Yes	Yes				Yes		Yes		
GRMA	Yes	No	No	No								
Cyclic	Yes	No	No	No								
LLFS										Yes		
SG				Yes								
Hard RT		Yes										
Soft RT		Yes										
output analysis		No		Yes								Yes
Gantt-chart	No	Yes		No						Yes		Yes
extensibility		Yes	No	Yes			Yes	Yes				
system model	p.f.		t/r G	t/ G								
available	No	No	Yes	Yes	No	No	No			No		No
continues	No					No				No		No

Ast: Asserts
Det: DET/SAT/SIM
Perts: PERTS SAT
Dtress: DTRESS/PERTSSim
Brux: Tool of the University Libre de Bruxelles
Org: Framework of the Oregon State University
Cai: CAISARTS
123: Scheduler 1-2-3
p.f. : parameter file
t/r G: task/resource Graph
PIP: Priority Inheritance Protocol
PCP: Priority Ceiling Protocol
DCP: Device Control Protocol
SRP: Session Reservation Protocol
SBP: Stack-Based Protocol
GRMA: Generalized RMA
Cyclic: Cyclic Executives
LLFS: Least Laxity First Scheduler
SG: Simulator Generator
Hard RT: Hard Real-Time
Soft RT: Soft Real-Time
continues: continuing development work

Table 2.1: Real-Time Scheduling Tool-Kit Summary

tool-kits are simulators, some of them are simulation languages, and only a few of them are frameworks.

1. A simulator is a simulation program that can be used directly by users. However, all functionality is predefined, so it may not meet all requirements of users. For example, users cannot generate executable code and execute the new code. Jan Jonsson [32] developed a tool-kit called generic allocation and scheduling Tool (GAST) to classify different types of scheduling algorithms, compare the performance of different schedulers, and search for possible scheduling algorithms for a task set with AI techniques to improve the performance of multiprocessor schedulers. However, his multiprocessor scheduler does not take advantage of the multiprocessor system architecture as it still applies a serial algorithm for the schedulability test. Among the tool-kits in the Table 2.1, DET/SAT/SIM, PERTS SAT, DTRESS/PERTSSim, AFTER, Brux, CAISARTS, and Scheduler 1-2-3 are simulators. SAW does not provide a simulation function but provides a system analysis function and system designers can use it interactively to evaluate the designed system. Thus, SAW can be considered to be a simulator, too.

2. STRESS [9] is a simulation language with support for specifying the system architecture, properties of task sets, scheduling algorithms, and resource management protocols. On the one hand, STRESS is a good tool-kit to evaluate scheduling algorithms and resource management protocols, and even help design new scheduling algorithms. On the other hand, STRESS makes some unrealistic assumptions about scheduling algorithms, such as a task starts on each tick of the system clock and the cost of a context switch is zero, which is impractical. Resources are limited to semaphores in STRESS, eliminating many high-level concurrency constructs, such as monitors. STRESS does not consider the relationship among system architectures, properties of task set, scheduling algorithms, and resource management protocols. Among the tool-kits in Table 2.1, Asserts and Brux also provide simulation languages.

3. A framework approach considers the extensibility and potential requirements of users. In a framework, a user can specify a simulation environment, scheduler, resource management protocols, and task set, etc. Then the frame work generates source code based on the specification, compiles the code, and runs the executable code with the task set. Matthew Francis Storch [68] applied the framework approach. His tool-kit also focuses on failure analysis and hierarchical scheduling. Among the tool-kits in the Table 2.1, Framework of the Ore-

gon State University is also based on the framework approach, and it was implemented in C++.

Chapter 3

Fixed-Priority with Preemption Threshold

This chapter examines FPPT to schedule independent periodic task sets. First, the FPPT schedulability test is proved robust under its critical instant. Second, the solution space for a task set schedulable by FPPT is presented, which is delimited by two special preemption threshold assignments. Third, effective algorithms to compute those two special assignments are presented. Finally, the relationship of these algorithms is presented, followed by comparing FPPT with PIP. The first three items are the major contributions of this chapter. No prior research work has been done in exploring the mechanism to generate more valid preemption threshold assignments when a task set is schedulable by FPPT. The reasons to explore the solution space are as follows. First, multiple valid assignments for a task set schedulable by FPPT may be available, providing a real-time system designer with additional flexibility for scheduling periodic tasks, potentially allowing secondary requirements to be satisfied. Second, under the maximal assignment, each individual preemption threshold is maximized, so the number of preemptions is minimized, reducing processor time and memory requirements. Third, if a valid assignment makes each task non-preemptive, i.e., schedulable by FPNP, then resource management protocols may be unnecessary. Even if a resource management protocol is necessary, reducing the number of preemptions potentially reduces the time when the protocol has an effect, which enhances the potential schedulability.

3.1 Relationship among FPNP, FPPT, and FPP

FPPT has two boundary cases. When no preemption is allowed, then the preemption threshold of each task is equal to the highest regular priority in a task set and FPPT degrades to FPNP; when any task can be preempted by other tasks with higher regular priorities, then the preemption threshold of each task is equal to its regular priority in a task set and FPPT degrades to FPP. The relationship among FPNP, FPPT, and FPP is indicated in Figure 3.1. Moving along the spectrum to the left, as the preemption threshold of each task is increasing, task preemptions are decreasing and the boundary case has no preemption, i.e., FPPT degrades to FPNP. Moving along the spectrum to the right, as the preemption threshold of each task is decreasing, task preemptions are increasing and the boundary case is that any task τ_i can preempt the execution of another task τ_j if $\pi_i < \pi_j$, i.e., FPPT degrades to FPP.

Figure 3.1: Relationship among FPNP, FPPT, and FPP

The set of all task sets schedulable by a real-time scheduler RS is denoted as A_{RS}. Given two real-time schedulers RS_1 and RS_2, if $A_{RS_1} \nsubseteq A_{RS_2}$ and $A_{RS_2} \nsubseteq A_{RS_1}$, then RS_1 and RS_2 are called *incomparable*. For example, FPP and FPNP are two typical schedulers that are incomparable. The former maximizes the number of preemptions while the latter eliminates all preemptions. FPPT fills the gap between them.

3.2 FPPT Critical Instant

The correct schedulability test for FPPT is based on the critical instant in [74], where a task's worst case response time occurs when the following three conditions are satisfied.

1. A job of each task with a higher regular priority arrives at the same time (time 0).
2. A task may be blocked by another task with a lower regular priority. That is, the task contributing the maximum blocking time has just started before the first critical instant.
3. Tasks are arriving at their maximum rates.

In FPPT, a task can be blocked by another task with a lower regular priority. Wang and Saksena [74] adopt the same critical instant as [26] except that the blocking time from a task with a

lower regular priority is equal to $\max_{\gamma_j \leq \pi_i < \pi_j} C_j$ instead of $\max_{\gamma_j \leq \pi_i < \pi_j} C_j - 1$. It seems that the latter is better by intuition. However, the latter is useless in practice because the latter assumes that the time unit is a clock tick, such as a number of microseconds. In practice, the time unit for a task is much bigger than the clock tick such as millisecond. Thus, the difference between the results from these two formulae is less than 1/1000, which can be ignored. Furthermore, in order to use the latter formula, all time parameters must use the clock tick as time unit. That is why Formula 2.16 is adopted in the schedulability test.

For convenience, this critical instant is called the *FPPT critical instant*, which is different from the RMA critical instant. One property of the FPPT critical instant is discussed as follows.

The RMA critical instant analysis is one important property in FPP. For FPP, if each job in the first level-i busy period starting from the first critical instant does not miss its deadline, then the whole task set is schedulable by FPP. The RMA critical instant works for all tasks as one scenario makes each task get its worst case response time. Such a critical instant is called a *universal critical instant* for FPP. Furthermore, such a critical instant works for any task set based on FPP. However, this critical instant analysis is not true for FPNP. One counter example is tasks $\tau_1 = (30, 3)$ and $\tau_2 = (50, 42)$, which is shown in Figure 3.2. Suppose each task always executes up to its worst case computation time. Based on FPNP and the RMA critical instant, $J_{1,0}$ starts to run at time 0 and finishes at time 3; $J_{2,0}$ starts to run at time 3 and finishes at time 45. Even if both $J_{1,0}$ and $J_{2,0}$ finish within their deadlines, i.e., all jobs in the first level-i busy period starting from the first critical instant meet their deadlines, these two tasks are unschedulable by FPNP as $J_{1,2}$ misses its deadline at time 90. The reason is non-preemption. When $J_{1,2}$ is ready at time 60, $J_{2,1}$ is running. Though $J_{1,2}$ has higher regular priority than $J_{2,1}$, the former cannot preempt the execution of the latter due to non-preemption.

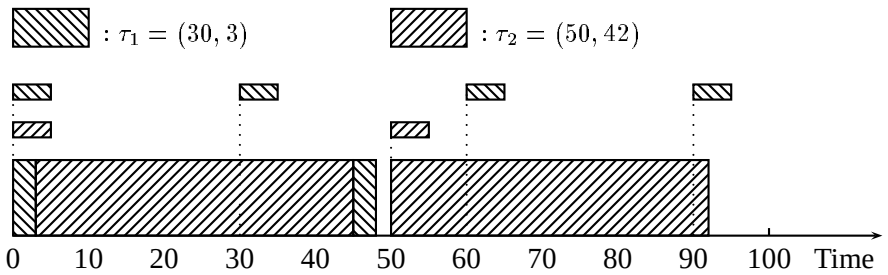


Figure 3.2: Critical Instant for FPP Invalid for FPNP

As mentioned in Section 2.1.2, FPP is robust based on the RMA critical instant. However, FPNP is not robust based on the RMA critical instant. For example, tasks $\tau_1 = (40, 11)$, $\tau_2 = (70, 40)$, and $\tau_3 = (280, 19)$ are schedulable by FPNP based on the RMA critical instant. The scheduling result for all jobs in the first major cycle is shown in Table 3.1. As all jobs in the

Job	Ready Time	Start Time	Finish Time	Response Time	Deadline
$J_{1,0}$	0	0	11	11	40
$J_{2,0}$	0	11	51	40	70
$J_{1,1}$	40	51	62	11	40
$J_{3,0}$	0	62	81	19	280
$J_{1,2}$	80	81	92	11	40
$J_{2,1}$	70	92	132	40	70
$J_{1,3}$	120	132	143	11	40
$J_{2,2}$	140	143	183	40	70
$J_{1,4}$	160	183	194	11	40
$J_{1,5}$	200	200	211	11	40
$J_{2,3}$	210	211	251	40	70
$J_{1,6}$	240	251	262	11	40

Table 3.1: Task Set Schedulable By FPNP under the RMA Critical Instant

first major cycle meet their deadlines, then the task set is schedulable by FPNP under the RMA critical instant.

However, decreasing the computation time of task τ_3 to 14 results in the task set becoming unschedulable by FPNP. The scheduling result remains the same for $J_{1,0}$, $J_{2,0}$, and $J_{1,1}$. However, $J_{3,0}$ finishes at time 76 due to decreased computation time followed by $J_{2,1}$, which finishes at time 116. Then $J_{1,2}$ starts to run at time 116 and misses its deadline. Thus, the task set is unschedulable by FPNP under the RMA critical instant. Hence, FPNP is not robust under the RMA critical instant. At the end of Section 3.4, it is proven that FPPT is robust under its critical instant.

When the computation time of τ_3 is decreased to 0, resulting in that τ_3 is removed from the task set, contrary to the intuition that τ_1 and τ_2 should be schedulable by FPNP under the RMA critical instant as more processor time is available, tasks τ_1 and τ_2 are unschedulable by FPNP under the RMA critical instant. The details are as follows. $J_{1,0}$ starts to run first and finishes at time 11, followed by $J_{2,0}$, which finishes at time 51. $J_{1,1}$ starts to run at time 51 and finishes at time 62. As there are no more ready jobs, the processor can be used by other applications. At time

70, $J_{2,1}$ is ready and starts to run, which finishes at time 110. $J_{1,2}$ ready at time 80 starts to run at time 110 and misses its deadline. Thus, τ_1 and τ_2 are unschedulable by FPNP under the RMA critical instant. Therefore, it is not possible to construct a schedulability test by incrementally testing tasks from the highest to the lowest priority as in RMA.

As the schedulability test for FPNP cannot be based on the RMA critical instant, thus FPNP requires its own critical instant definition [26, 74], hence the schedulability test for FPPT cannot be based on the RMA critical instant because FPNP is a boundary case of FPPT. The new critical instant for FPPT creates an individual worst case scenario for each task in a task set, because there is no universal critical instant available for FPPT (or FPNP) similar to FPP.

Theorem 1

There does not exist a universal critical instant for the whole task set in FPPT except when FPPT degrades to FPP.

Proof: Suppose there exists such a universal critical instant C_{I_u}.

1. When FPPT degrades to FPNP, all tasks are non-preemptive. Given task set $S = \{\tau_1, \tau_2, \ldots, \tau_n\}$, suppose task τ_k is a task with maximum computation time among the tasks with regular priorities lower than τ_1 in a task set. Consider task set $S' = \{\tau_1, \tau_2, \ldots, \tau_k\}$.

   Consider task τ_1 with highest priority in task set S'. It may be blocked by a task with lower priority due to non-preemption, in this case say task τ_k. Thus, the worst case for τ_1 is if it is ready just after τ_k starts to execute. As C_{I_u} is a universal critical instant, then it also includes this scenario for task τ_1. Denote this scenario as C_{I_1}.

   Consider task τ_k in task set S'. At the beginning of the system (at time 0), any job of τ_k must wait until all tasks with higher regular priorities have finished. Thus, the worst case is all tasks with higher regular priorities are ready at time 0 and so is τ_k. Under this scenario, τ_k gets its worst case response time. As C_{I_u} is a universal critical instant, then C_{I_u} must include this scenario for τ_k. Denote this scenario as C_{I_k}.

   Combining C_{I_1} with C_{I_k} can obtain another scenario: tasks τ_1, τ_2, ..., and τ_k are ready at time 0, and one job of τ_k just starts to run. Under this scenario, task τ_k gets a worse response time than in C_{I_k}, which is a contradiction as C_{I_u} is a universal critical instant. Thus, such a C_{I_u} does not exist.

2. When FPPT degrades to neither FPP nor FPNP, there exists at least one task with preemption threshold higher than its regular priority. Given task set $S = \{\tau_1, \tau_2, \ldots, \tau_n\}$ with

regular priority assignment $\pi = \{\pi_1, \pi_2, \ldots, \pi_n\}$ where $\pi_i = i$ and preemption threshold assignment $\gamma = \{\gamma_1, \gamma_2, \ldots, \gamma_n\}$, suppose task τ_k is a task such that $\gamma_k < \pi_k$ and k is minimum. There also exists a task τ_i with $\pi_i = \gamma_k$ because each task has a unique regular priority and the regular priority is between 1 and n inclusively. Consider task set $S' = \{\tau_1, \tau_2, \ldots, \tau_k\}$ with regular priority assignment $\pi' = \{\pi_1, \pi_2, \ldots, \pi_k\}$ and preemption threshold assignment $\gamma' = \{\gamma_1, \gamma_2, \ldots, \gamma_k\}$.

Consider task τ_i with $\pi_i = \gamma_k$ in task set S'. It may be blocked by τ_k due to $\pi_i = \gamma_k$. Then the worst case for τ_i is τ_1, τ_2, . . ., τ_i are ready at the same time just after τ_k starts to execute. As C_{I_u} is a universal critical instant, then it also includes this scenario for task τ_i. Denote this scenario as C_{I_i}.

Consider task τ_k in task set S'. At the beginning of the system (at time 0), any job of τ_k must wait until all tasks with higher regular priorities have finished. Thus, the worst case is all tasks with higher regular priorities are ready at time 0, so is τ_k. Under this scenario, τ_k gets its worst case response time. As C_{I_u} is a universal critical instant, then C_{I_u} must include this scenario for τ_k. Denote this scenario as C_{I_k}.

Combining C_{I_i} with C_{I_k} can obtain another scenario: tasks τ_1, τ_2, . . ., and τ_k are ready at time 0, and one job of τ_k just starts to run. Under this scenario, task τ_k gets a worse response time than in C_{I_k}, which is a contradiction as C_{I_u} is a universal critical instant. Thus, such a C_{I_u} does not exist. ■

Based on Theorem 1, the worst case scenario for each task in FPPT may not occur at the same time except when it degrades to FPP. In other words, under the worst case scenario for a task, it is unnecessary to check whether all other tasks get their worst case scenarios because their worst case scenarios may not exist at the same time.

3.3 Blocking Time from a Task with Lower Regular Priority

Based on the critical instant for FPPT, it is shown that a task can be blocked by at most one other task with a lower regular priority, which is formally described in Theorem 2 and shows that Formula 2.16 is valid.

Theorem 2

A task can be blocked by at most one task with a lower regular priority based on the FPPT critical instant.

Proof: Consider the simple case that a task is blocked by two other tasks with lower regular priorities. Given tasks τ_A, τ_B, and τ_C with $\pi_A < \pi_B$ and $\pi_A < \pi_C$, suppose τ_A can be blocked by τ_B and τ_C, which implies $\gamma_B \leq \pi_A$ and $\gamma_C \leq \pi_A$.

As $\pi_A < \pi_B$ and $\pi_A < \pi_C$, tasks τ_A, τ_B, and τ_C cannot be ready at the same time. In addition, task τ_A can be ready before neither task τ_B nor task τ_C. Otherwise, task τ_A can be blocked by neither task τ_B nor task τ_C. In order to guarantee τ_A is blocked by both tasks τ_B and τ_C, tasks τ_B and τ_C must start to execute for the first time before τ_A is ready so that the running priorities of τ_B and τ_C are upgraded to γ_B and γ_C respectively. Furthermore, if task τ_B (τ_C) starts to run first, then it must be preempted by τ_C (τ_B). Otherwise, τ_A is only blocked by one task. As $\gamma_B \leq \pi_A$ and $\gamma_C \leq \pi_A$, task τ_A cannot preempt their executions and must wait.

1. τ_C runs first. As τ_A is also blocked by τ_B, τ_B preempts the execution of τ_C, after τ_B finishes, τ_C continues its execution because $\gamma_C \leq \pi_A$. After τ_C finishes, τ_A starts its execution. This execution order is indicated in Figure 3.3. In the diagram, t_C, t_B, and t_A are the ready times of τ_C, τ_B, and τ_A respectively. Therefore, $\pi_B < \gamma_C$. However, due to $\gamma_C \leq \pi_A$ and $\pi_A < \pi_B$, $\gamma_C < \pi_B$. As $\pi_B < \gamma_C$, then $\pi_B < \pi_B$, which is a contradiction.

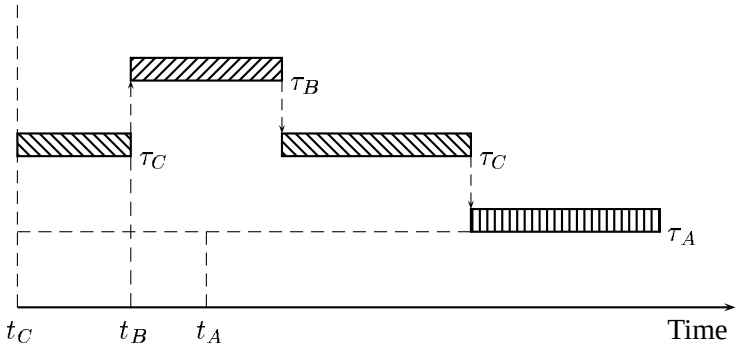


Figure 3.3: Task Execution Order

2. τ_B runs first. Similarly to the first case, $\pi_C < \pi_C$, which is a contradiction.

Therefore, a task can be blocked by at most one task with a lower regular priority. ■

3.4 Robustness of FPPT

As mentioned, FPP is robust under the RMA critical instant while FPNP is not. However, FPPT is also robust under its own critical instant, which is described formally in Theorem 3. This property guarantees that the FPPT schedulability test can be performed incrementally.

Theorem 3
FPPT is robust under the FPPT critical instant.

Proof: Suppose a task set is schedulable by FPPT with regular priority assignment π and preemption threshold assignment γ. The proof shows that the schedulability of any task is unaffected if the computation time of any task τ_i is decreased.

1. For any task τ_h with $\pi_h < \pi_i$. The interference from tasks with regular priorities higher than τ_h is the same. However, $B_{(\tau_h)}$ may be shorter if τ_i was contributing the maximal blocking time originally based on Formula 2.16, which indicates that τ_h may start earlier so its worst case response time cannot be longer. Hence, the schedulability of τ_h is unaffected.

2. For task τ_i itself. The interference from tasks with regular priorities higher than τ_i is the same. $B_{(\tau_i)}$ is also the same based on Formula 2.16. Then the worst case response time of τ_i cannot be longer due to its decreased computation time. Hence, the schedulability of τ_i is unaffected.

3. For any task τ_k with $\pi_k > \pi_i$. $B_{(\tau_k)}$ is the same based on Formula 2.16. However, the interference from tasks with regular priorities higher than τ_k cannot be longer due to the decreased computation time of τ_i as $\pi_i < \pi_k$. Then, the worst case response time of τ_k cannot be longer due to decreasing the computation time of τ_i. Hence, the schedulability of τ_k is unaffected.

Therefore, given a task set schedulable by FPPT, if the computation time of any task τ_i is decreased, it does not affect the schedulability of the task set. Based on the definition of robustness, FPPT is robust under its critical instant. ■

When decreasing the computation time of a task, consider the boundary case such as decreasing it to 0. Thus, the task is removed from a task set. As FPPT is robust under its critical instant, the rest of the tasks are still schedulable. Hence, a subset of a task set schedulable by FPPT is still schedulable by FPPT, which is described in Corollary 1.

Corollary 1
If a task set S is schedulable by FPPT, then task set S' such that $S' \subset S$ is also schedulable. Hence, the schedulability test for FPPT can be performed incrementally.

3.5 Preemption Threshold Assignment

The schedulability test provided in Section 2.2.4 assumes a preemption threshold assignment already exists, which is not true practically. In practice, only the regular priority is defined. For example, the regular priority can be assigned based on the RMA priority assignment. It is reasonable to assume that the regular priorities are predefined for a given task set. The reason is as follows. The algorithm to search for a feasible fixed-priority assignment has been well studied. The research work in [69] presents an effective algorithm to find a feasible fixed-priority assignment for preemptive scheduling. In addition, this algorithm is also applicable to non-preemptive scheduling [26].

To determine whether a task set is schedulable, the key point is to find a valid preemption threshold assignment by using the schedulability test. However, the search space of all possible preemption threshold assignments is $O(n!)$ in the worst case, which indicates that an effective searching algorithm is required. Furthermore, given a task set with predefined regular priority, it may be schedulable under several preemption threshold assignments.

This section proves that when a task set with predefined regular priority is schedulable, any valid preemption threshold assignment must be delimited by two special assignments. When these two special assignments are known, more valid assignments can be generated. Computing these two special assignments is discussed in Section 3.6.

3.5.1 Simple Example

If a task set with predefined regular priority is schedulable by FPPT, there may exist multiple valid preemption threshold assignments making the task set schedulable. For example, given tasks $\tau_1 = (20, 8)$, $\tau_2 = (30, 6)$, $\tau_3 = (50, 10)$, $\tau_4 = (100, 8)$, and $\tau_5 = (300, 12)$ with regular priority assignment $\pi = \{1, 2, 3, 4, 5\}$, they are schedulable with only the following seven preemption

threshold assignments (verified by exhaustive search)

$$\Gamma_1 = \{1,2,3,4,5\}, \quad \Gamma_2 = \{1,1,3,4,5\}, \quad \Gamma_3 = \{1,1,2,4,5\}, \quad \Gamma_4 = \{1,1,1,4,5\},$$
$$\Gamma_5 = \{1,1,1,3,5\}, \quad \Gamma_6 = \{1,1,1,2,5\}, \quad \Gamma_7 = \{1,1,1,1,5\}.$$

For all of these assignments, the preemption thresholds for $\gamma_1 = 1$ and $\gamma_5 = 5$ are the same. Now consider the preemption thresholds for τ_2, τ_3, and τ_4.

Observation: all valid assignments are delimited by Γ_1 and Γ_7. The preemption thresholds of tasks τ_2, τ_3, and τ_4 in Γ_1 are minimal while the preemption thresholds of tasks τ_2, τ_3, and τ_4 in Γ_7 are maximal. In other words, Γ_2, Γ_3, Γ_4, Γ_5, and Γ_6 are logically between them.

Conjecture: For a given task set schedulable by FPPT with predefined regular priority, there are two special valid assignments γ_{min} and γ_{Max}, where $\gamma_{min} = \{\gamma_{m_1}, \gamma_{m_2}, \ldots, \gamma_{m_n}\}$ and $\gamma_{Max} = \{\gamma_{M_1}, \gamma_{M_2}, \ldots, \gamma_{M_n}\}$. In addition, $\gamma_{m_i} \geq \gamma_{M_i}$ for $1 \leq i \leq n$. For any valid assignment $\gamma = \{\gamma_1, \gamma_2, \ldots, \gamma_n\}$, it must satisfy that $\gamma_i \in [\gamma_{M_i}, \gamma_{m_i}]$ for $1 \leq i \leq n$. In addition, an assignment satisfying this condition may be invalid. For example, assignment $\gamma = \{1,2,1,3,5\}$ is invalid as $J_{3,0}$ misses its deadline under this assignment. Furthermore, any other assignments that do not satisfy this condition are invalid. This conjecture is proved correct in the following discussion.

3.5.2 Partial Order Relationship

Before further discussion, a partial order relationship between two assignments is defined.

Definition 1
Given a task set and two valid assignments γ and γ', if all individual preemption thresholds of γ are equal to or greater[1] *than those of γ', i.e., $\gamma_i \geq \gamma'_i$ for $1 \leq i \leq n$, then define $\gamma \preccurlyeq \gamma'$.*

Clearly, $\preccurlyeq$ is a reflexive, anti-symmetric, and transitive relationship. In the above example, $\Gamma_1 \preccurlyeq \Gamma_2 \preccurlyeq \Gamma_3 \preccurlyeq \Gamma_7$ (not a complete list).

Definition 2
Suppose Γ is the group of all valid assignments for a task set. Let $\gamma_{min} \in \Gamma$ satisfy the relationship $\gamma_{min} \preccurlyeq \gamma$ for all $\gamma \in \Gamma$, i.e., each individual preemption threshold of γ_{min} is equal to or

[1] *Note, the larger a numeric value, the lower its priority as mentioned before.*

greater than the corresponding part of any other valid assignment in Γ*, then* γ_{min} *is defined as the minimal assignment in* Γ*.*

Definition 3
Suppose Γ *is the group of all valid assignments for a task set. Let* $\gamma_{Max} \in \Gamma$ *satisfy the relationship* $\gamma \preccurlyeq \gamma_{Max}$ *for all* $\gamma \in \Gamma$*, i.e., each individual preemption threshold of* γ_{Max} *is less than or equal to the corresponding part of any other valid assignment in* Γ*, then* γ_{Max} *is defined as the maximal assignment in* Γ*.*

In the above example, Γ_1 and Γ_7 are the minimal assignment and the maximal assignment, respectively. The purpose for defining γ_{min} and γ_{Max} is they exist and can be calculated effectively when a task set is schedulable by FPPT with predefined regular priority.

3.5.3 Generating Valid Assignments

When multiple valid assignments may exist for a task set schedulable by FPPT with predefined regular priority, an effective mechanism is needed to calculate additional valid assignments. It is proved that for any valid assignment γ, it must satisfy that $\gamma_{min} \preccurlyeq \gamma \preccurlyeq \gamma_{Max}$ (see Theorem 4). In this section, γ_{min} and γ_{Max} are assumed available. In the next section, effective algorithms to compute γ_{min} and γ_{Max} are presented. Before proving the correctness of this necessary condition, four lemmas are proved. Lemma 1, 2, and 3 provide theoretical support to generate another valid assignment based on two known valid assignments that satisfy some special conditions. In the proofs of these lemmas, the following known lemmas, corollary, and theorem in [74] are used[2], which are rephrased first:

Lemma **AFFECT** : Changing the preemption threshold of task τ_i from γ_1 to γ_2 may only affect the worst case response time of task τ_i and those tasks whose regular priorities are between γ_1 and γ_2.

Corollary **AFFECT** : The worst case response time of task τ_i cannot be affected by the preemption threshold assignment of any task τ_j with $\pi_j < \pi_i$.

Theorem **VALID** : Assume a task set of n tasks schedulable with predefined regular priority $\pi = \{\pi_1, \pi_2, \ldots, \pi_n\}$ and preemption threshold assignment $\gamma = \{\gamma_1, \gamma_2, \ldots, \gamma_n\}$. If changing only

[2]Lemma **AFFECT**, Corollary **AFFECT**, Theorem **VALID**, and Lemma **INVALID** correspond to Lemma 5.1, Corollary 5.1, Theorem 5.1,and Lemma 5.2 in [74], respectively.

the preemption threshold of task τ_j from γ_j to γ'_j, where $\gamma'_j > \gamma_j$ makes task τ_j still schedulable, then the whole task set is still schedulable by the preemption threshold assignment $\gamma' = \{\gamma_1, \gamma_2, \ldots, \gamma_{j-1}, \gamma'_j, \gamma_{j+1}, \ldots, \gamma_n\}$.

Lemma **INVALID** : Given a task set of n tasks with predefined regular priority $\pi = \{\pi_1, \pi_2, \ldots, \pi_n\}$, if setting the preemption threshold of task τ_i to the highest regular priority, i.e., $\gamma_i = \pi_1$, still cannot make task τ_i schedulable, then the whole task set is unschedulable.

Lemma 1

Let γ and γ' be two valid assignments such that $\gamma' \preccurlyeq \gamma$. Then assignment $\gamma'' = \{\gamma_1, \gamma_2, \ldots, \gamma_{k-1}, \gamma''_k, \gamma'_{k+1}, \ldots, \gamma'_n\}$ is also valid, where $\gamma''_k \in [\gamma_k, \gamma'_k]$ for $k \in [1, n]$.

Proof: The proof first considers the case when $k = n$ and shows that it is correct. Then it uses the correct result to prove the case when $k = n - 1$ to be correct and so on for the cases when $k = n - 2, n - 3, \ldots, 2, 1$. Note, $\gamma' \preccurlyeq \gamma$ indicates $\gamma_i \leq \gamma'_i$ for $1 \leq i \leq n$.

1. Assignment $\gamma'' = \{\gamma_1, \gamma_2, \ldots, \gamma_{n-1}, \gamma''_n\}$ is valid, where $\gamma''_n \in [\gamma_n, \gamma'_n]$.

   Based on Lemma **AFFECT**, changing γ_n to γ''_n in γ may only affect the worst case response time of task τ_n and those tasks whose regular priorities are between γ''_n and γ_n.

   (a) For any task τ_i with $\pi_i \in [\gamma_n, \gamma''_n]$, consider its worst case response time under γ'' and γ. For both cases, the interference from tasks with higher regular priorities is the same. However, its blocking time from a task with a lower regular priority under γ'' cannot be longer than γ based on Formula 2.16 as $\gamma_n \leq \gamma''_n$. Thus, the response time of task τ_i cannot be longer under γ'' than γ. Hence, the schedulability of τ_i is unaffected.

   (b) Consider the worst case response time of τ_n under γ'' and γ'. For both assignments, the blocking time from a task with a lower regular priority is the same, which is equal to 0. Under γ'', any task τ_j with $\pi_j < \gamma''_n$ can preempt the execution of τ_n; under γ', any task τ_j with $\pi_j < \gamma'_n$ can preempt the execution of τ_n. As $\gamma''_n \leq \gamma'_n$, the number of tasks that can preempt the execution of τ_n under γ'' is less than or equal to that under γ'. Hence, the interference for τ_n from tasks with higher regular priorities under γ'' cannot be worse than that under γ'. As γ' is valid, then τ_n meets its deadline under γ'', too.

   Therefore, γ'' is also valid.

2. Based on step 1, assignment $\gamma_S = \{\gamma_1, \gamma_2, \ldots, \gamma_{n-1}, \gamma'_n\}$ is valid. Now consider assignment $\gamma_T = \{\gamma_1, \gamma_2, \ldots, \gamma_{n-2}, x, \gamma'_n\}$ where $x \in [\gamma_{n-1}, \gamma'_{n-1}]$.

   Based on Lemma **AFFECT**, changing γ_{n-1} to x in γ_S may only affect the worst case response time of task τ_{n-1} and those tasks whose regular priorities are between x and γ_{n-1}.

   (a) For any task τ_i with $\pi_i \in [\gamma_{n-1}, x]$, consider its worst case response time under γ_T and γ_S. For both cases, the interference from tasks with higher regular priorities is the same. However, its blocking time from a task with a lower regular priority under γ_T cannot be longer than γ_S based on Formula 2.16 as $\gamma_{n-1} \leq x$. Thus, the response time of task τ_i cannot be longer under γ_T than γ_S. Hence, the schedulability of τ_i is unaffected.

   (b) Consider the worst case response time of τ_{n-1} under γ_T and γ'. For both assignments, the blocking time from a task with a lower regular priority is the same. Under γ_T, any task τ_j with $\pi_j < x$ can preempt the execution of τ_{n-1}; under γ', any task τ_j with $\pi_j < \gamma'_{n-1}$ can preempt the execution of τ_{n-1}. As $x \leq \gamma'_{n-1}$, the number of tasks that can preempt the execution of τ_{n-1} under γ_T is less than or equal to that under γ'. Hence, the interference for τ_{n-1} from tasks with higher regular priorities under γ_T cannot be worse than that under γ'. As γ' is valid, then τ_{n-1} meets its deadline under γ_T, too.

   Therefore, γ_T is also valid.

3. Using a similar approach in step 2, assignment $\gamma_T = \{\gamma_1, \gamma_2, \ldots, \gamma_{k-1}, x, \gamma'_{k+1}, \ldots, \gamma'_n\}$ can be proved valid, where $x \in [\gamma_k, \gamma'_k]$ when $k = n-2, n-3, \ldots, 2, 1$.

Therefore, Lemma 1 is correct. ■

Lemma 1 guarantees that additional valid assignments can be calculated based on γ_{min} and γ_{Max}.

Lemma 2

Let γ and γ' be two valid assignments. If k is the maximal number such that $\gamma'_k > \gamma_k$, then assignment $\gamma'' = \{\gamma_1, \gamma_2, \ldots, \gamma_{k-1}, \gamma'_k, \gamma_{k+1}, \ldots, \gamma_n\}$ is also valid.

Proof: Based on Lemma **AFFECT**, changing γ_k to γ''_k in γ may only affect the worst case response time of task τ_k and those tasks whose regular priorities are between γ''_k and γ_k.

1. For any task τ_i with $\pi_i \in [\gamma_k, \gamma_k'']$, consider its worst case response time under γ and γ''. For both cases, the interference from tasks with higher regular priorities is the same. However, its blocking time from a task with a lower regular priority cannot be longer under γ'' than γ based on Formula 2.16 as $\gamma_k' > \gamma_k$ and $\gamma_k' = \gamma_k''$. Thus, the response time of task τ_i cannot be longer under γ'' than γ, i.e., the schedulability of τ_i is unaffected.

2. Consider the worst case response time of τ_k under γ' and γ''. For both assignments, the interference from tasks with higher regular priorities is the same. However, the blocking time from a task with a lower regular priority cannot be longer under γ'' than γ'. First, $\gamma_i' \leq \gamma_i''$ for $k < i \leq n$ as k is the maximal number such that $\gamma_k' > \gamma_k$ and $\gamma_i = \gamma_i''$ for $k < i \leq n$. Second, $B(\tau_k'') \leq B(\tau_k')$ based on Formula 2.16 as $\gamma_i = \gamma_i''$ for $k < i \leq n$. Thus, the response time of τ_k cannot be longer under γ'' than γ'. As γ' is valid, τ_k meets its deadline under γ'', too.

Therefore, γ'' is also valid. ■

Lemma 2 provides additional support to generate a new valid assignment based on two valid assignments that satisfy some specific conditions. The new valid assignment is obtained by decreasing the preemption threshold of one task to the preemption threshold in another valid assignment of the same task, which provides the possibility to generate a new valid assignment that may be a candidate for γ_{min}.

Lemma 3

Let γ and γ' be two valid assignments. If $\gamma_i \leq \gamma_i'$ for $1 \leq i \leq n$ except that $\gamma_k' < \gamma_k$, then assignment $\gamma'' = \{\gamma_1, \gamma_2, \ldots, \gamma_{k-1}, \gamma_k', \gamma_{k+1}, \ldots, \gamma_n\}$ is also valid.

Proof: Based on Lemma **AFFECT**, changing γ_k to γ_k' in γ may only affect the worst case response time of task τ_k and those tasks with regular priorities between γ_k and γ_k'.

1. For any task τ_h with $\pi_h \in [\gamma_k', \gamma_k]$, there are two cases.

   (a) τ_h is blocked by τ_k under γ''. In this case, τ_h must be blocked by τ_k under γ' because $\gamma_i'' \leq \gamma_i'$ for $1 \leq i \leq n$, resulting in τ_h getting the same blocking time under γ'' and γ'. However, τ_h's interference is no worse under γ'' than under γ' because $\gamma_h'' \leq \gamma_h'$. Hence, the worst case response time of τ_h under γ'' is not longer than under γ'. As γ' is valid, then the schedulability of τ_h is unaffected under γ''.

(b) τ_h is not blocked by τ_k under γ''. In this case, τ_h gets the same blocking time under γ'' and γ as $\gamma''_i = \gamma_i$ for $1 \leq i \leq n$ except $i \neq k$. Note, τ_h's response time can be affected by only one task with lower regular priority. As τ_h's priority remains the same under γ'' and γ, τ_h gets the same interference under γ'' and γ. Hence, τ_h gets the same worst case response time under γ'' and γ. As γ is valid, then the schedulability of τ_h is unaffected under γ''.

Therefore, the schedulability of τ_h is unaffected under γ''.

2. Consider the worst case response time of τ_k under γ'' and γ. For both assignments, the blocking time from a task with a lower regular priority is the same. However, under γ'', any task τ_j with $\pi_j < \gamma''_k$ can preempt the execution of τ_k; under γ, any task τ_j with $\pi_j < \gamma_k$ can preempt the execution of τ_k. As $\gamma'_k < \gamma_k$ and $\gamma'_k = \gamma''_k$, the number of tasks that can preempt the execution of τ_k under γ'' is less than or equal to that under γ. Hence, the interference for τ_k from tasks with higher regular priorities cannot be worse under γ'' than γ. Thus, the response time of τ_k cannot be longer under γ'' than γ. As γ is valid, τ_k meets its deadline under γ'', too.

Therefore, γ'' is also valid. ■

Lemma 3 provides further support to generate a new valid assignment based on two valid assignments satisfying some specific conditions. The new valid assignment is obtained by increasing the preemption threshold of one task to the preemption threshold in another valid assignment of the same task, which provides the possibility to generate a new valid assignment that may be a candidate for γ_{Max}.

When a task set is schedulable by FPPT with predefined regular priority, and both γ_{min} and γ_{Max} are already known, then all other valid assignments must be delimited by them, which is based on Lemma 1, 2, and 3. At the same time, it is possible to generate additional assignments based on γ_{min} and γ_{Max} directly without calculating the worst case response time for each task based on the schedulability test under the given assignment.

Theorem 4

Given a task set schedulable by FPPT with predefined regular priority, there exist valid assignments γ_{min} and γ_{Max} such that for any valid assignment γ, $\gamma_{min} \preccurlyeq \gamma \preccurlyeq \gamma_{Max}$.

Proof: Let $\gamma_L = \{\gamma_{L_1}, \gamma_{L_2}, \ldots, \gamma_{L_n}\}$ be any valid assignment such that for any other valid assignment γ, $\gamma \preccurlyeq \gamma_L$ is false; let $\gamma_U = \{\gamma_{U_1}, \gamma_{U_2}, \ldots, \gamma_{U_n}\}$ be any valid assignment such that

for any other valid assignment γ, $\gamma_U \preccurlyeq \gamma$ is false. For any assignment $\gamma = \{\gamma_1, \gamma_2, \ldots, \gamma_n\}$, define

$$Y_1 = \{j \mid \forall j, \gamma_j > \gamma_{L_j}\}, \quad Y_2 = \{j \mid \forall j, \gamma_j < \gamma_{U_j}\}, \quad Y_3 = \{j \mid \forall j, \gamma_{L_j} \geq \gamma_j \geq \gamma_{U_j}\}.$$

In other words, Y_1, Y_2, and Y_3 are subsets of $\{1, 2, \ldots, n\}$. Y_1 is the index set such that for any element j in Y_1, its corresponding preemption thresholds γ_j and γ_{L_j} satisfy γ_j is greater than γ_{L_j}. Y_2 is the index set such that for any element j in Y_2, its corresponding preemption thresholds γ_j and γ_{U_j} satisfy γ_j is less than γ_{U_j}. Y_3 is the index set such that for any element j in Y_3, its corresponding preemption thresholds γ_j, γ_{L_j} and γ_{U_j} satisfy γ_j is equal to or greater than γ_{U_j} but less than or equal to γ_{L_j}. Based on the definition of Y_1, Y_2, and Y_3, they form a partition of $\{1, 2, \ldots, n\}$. If γ is valid,

1. Assume $Y_1 \neq \emptyset$
   As $Y_1 \neq \emptyset$, find the maximal $k \in Y_1$. Based on the definition of Y_1, $\gamma_k > \gamma_{L_k}$ and k is maximal. Consider two valid assignments γ and γ_L. Based on Lemma 2, the assignment $\gamma' = \{\gamma_{L_1}, \gamma_{L_2}, \ldots, \gamma_{L_{k-1}}, \gamma_k, \gamma_{L_{k+1}}, \ldots, \gamma_{L_n}\}$ is also valid. Obviously, $\gamma' \preccurlyeq \gamma_L$ and $\gamma' \neq \gamma_L$. Then based on the definition of γ_L, this is a contradiction.

2. Assume $Y_1 = \emptyset$ and $Y_2 \neq \emptyset$

   - $Y_3 = \emptyset$
     It implies that $\gamma_j < \gamma_{U_j}$ for $1 \leq j \leq n$, i.e., $\gamma_U \preccurlyeq \gamma$ and $\gamma_U \neq \gamma$. As γ is also valid, then based on the definition of γ_U, this is also a contradiction.
   - $Y_3 \neq \emptyset$
     As $Y_2 \neq \emptyset$, choose the minimal $k \in Y_2$. As $Y_1 = \emptyset$, $\gamma_L \preccurlyeq \gamma$. Based on Lemma 1, assignment $\gamma'' = \{\gamma_1, \gamma_2, \ldots, \gamma_k, \gamma_{L_{k+1}}, \ldots, \gamma_{L_n}\}$ is valid. As well, $\gamma_{U_i} \leq \gamma_i$ for $1 \leq i \leq k-1$ based on the selection of k; $\gamma_{U_i} \leq \gamma_{L_i}$ for $1 \leq i \leq n$ by definition of γ_U and because $Y_1 = \emptyset$; and $\gamma_k < \gamma_{U_k}$ as $k \in Y_2$. Then, assignment $\gamma' = \{\gamma_{U_1}, \gamma_{U_2}, \ldots, \gamma_{U_{k-1}}, \gamma_k, \gamma_{U_{k+1}}, \ldots, \gamma_{U_n}\}$ is valid based on Lemma 3. Obviously, $\gamma_U \preccurlyeq \gamma'$ and $\gamma' \neq \gamma_U$. Then based on the definition of γ_U, this is a contradiction.

There are 8 possible combinations depending on whether each of Y_1, Y_2, and Y_3 is equal to $\emptyset$ or not, which is shown in Table 3.2. For convenience, in Table 3.2, when Y_i is equal to $\emptyset$, it is represented by 0; when Y_i is not equal to $\emptyset$, it is represented by 1. The values in the first column

are the corresponding decimal values of binary numbers formed by $Y_1Y_2Y_3$ where Y_1 is the most significant bit and Y_3 the least significant bit. The symbols in the last column indicate whether the corresponding combination of Y_1, Y_2, and Y_3 is valid or invalid. Symbols "$\surd$" and "$\times$" indicate "being valid" and "being invalid" respectively.

Value	Y_1	Y_2	Y_3	Status
0	0	0	0	$\times$
1	0	0	1	$\surd$
2	0	1	0	$\times$
3	0	1	1	$\times$
4	1	0	0	$\times$
5	1	0	1	$\times$
6	1	1	0	$\times$
7	1	1	1	$\times$

Table 3.2: Combinations of Y_1, Y_2, and Y_3

As Y_1, Y_2, and Y_3 form a partition of $\{1, 2, \ldots, n\}$, obviously, Y_1, Y_2, and Y_3 cannot be equal to $\emptyset$ simultaneously, whose corresponding decimal value of $Y_1Y_2Y_3$ is equal to 0, which is indicated with "$\times$" in Table 3.2. Case 1 includes the cases $Y_1 \neq \emptyset$ and the values of Y_2 and Y_3 can be either the empty set or the non-empty set. Thus, 4 distinct combinations are included, whose corresponding decimal values of $Y_1Y_2Y_3$ are 4, 5, 6, and 7. As Case 1 is impossible, then 4 distinct combinations are excluded, which are indicated with "$\times$" in Table 3.2. Similarly, 2 distinct combinations are excluded in Case 2, whose corresponding decimal values of $Y_1Y_2Y_3$ are 2 and 3, which are indicated with "$\times$" in Table 3.2.

After eliminating 7 distinct combinations, only the combination $Y_1 = \emptyset$, $Y_2 = \emptyset$, and $Y_3 \neq \emptyset$ holds, whose corresponding decimal value of $Y_1Y_2Y_3$ is 1, which is indicated with "$\surd$" in Table 3.2. Thus, $\gamma_{L_i} \geq \gamma_i \geq \gamma_{U_i}$ for $1 \leq i \leq n$ based on the definition of Y_3, i.e., $\gamma_L \preccurlyeq \gamma$ and $\gamma \preccurlyeq \gamma_U$ based on the definition of $\preccurlyeq$. Therefore, $\gamma_L \preccurlyeq \gamma \preccurlyeq \gamma_U$, $\gamma_L = \gamma_{min}$, and $\gamma_U = \gamma_{Max}$. ■

Note, not all assignments delimited by γ_{min} and γ_{Max} are valid. Consider the example in Section 3.5.1, $\gamma_{min} = \Gamma_1$ and $\gamma_{Max} = \Gamma_7$. Assignment $\{1, 2, 1, 3, 5\}$ is invalid. In fact, there are $(2 - 1 + 1)(3 - 1 + 1)(4 - 1 + 1) = 24$ distinct assignments delimited by γ_{min} and γ_{Max}, and 17 of them are invalid (verified by exhaustive search).

When a task set is unschedulable, there is no assignment able to make the task set schedulable.

Of course, neither γ_{min} nor γ_{Max} exists. Therefore, if an algorithm is guaranteed to compute γ_{min} and it cannot find γ_{min}, then γ_{Max} does not exist either. Being guaranteed is two-fold. First, if the task set is schedulable, the algorithm can find one valid assignment. Second, the assignment is minimal, which is equal to γ_{min}. Similarly, if an algorithm is guaranteed to compute γ_{Max} and it cannot find γ_{Max}, then γ_{min} does not exist either. This property is described formally in Corollary 2.

Corollary 2

If an algorithm is guaranteed to compute $\gamma_{min}(\gamma_{Max})$ and it cannot find $\gamma_{min}(\gamma_{Max})$, then $\gamma_{Max}(\gamma_{min})$ does not exist either.

Proof: Suppose γ_{Max} does exist. Based on Theorem 4, γ_{Max} is valid and makes the task set schedulable. As the algorithm is guaranteed to compute γ_{min} and it cannot find γ_{min}, which indicates that the task set is unschedulable, resulting in a contradiction. Thus, γ_{Max} does not exist. The other case can be proved similarly. ■

3.5.4 Area Delimited by γ_{min} and γ_{Max}

Based on Theorem 4, any valid assignment must be delimited by γ_{min} and γ_{Max}. Based on the assumption $\pi_i = i$ and $\gamma_i \leq \pi_i$ for $1 \leq i \leq n$, all valid assignments can be represented by a 2-dimensional area, where the regular priorities of all tasks are located on the horizontal direction and the preemption thresholds are located on the vertical direction. Given tasks $\tau_1 = (10, 1)$, $\tau_2 = (15, 1)$, $\tau_3 = (40, 4)$, $\tau_4 = (60, 10)$, $\tau_5 = (80, 20)$, $\tau_6 = (100, 15)$, $\tau_7 = (200, 10)$, $\tau_8 = (240, 16)$, the corresponding 2-dimensional area is shown in Figure 3.4.

Figure 3.4 shows two polylines corresponding to the minimal and maximal assignment for a task set with 8 tasks in a 2-dimensional space. Note, tasks are indexed starting from 1 instead of 0. In the diagram, $\gamma_{min} = \{1, 2, 3, 4, 5, 5, 5, 7\}$ is indicated by the upper polyline delimiting the shaded area, while $\gamma_{Max} = \{1, 1, 1, 2, 3, 3, 2, 3\}$ is indicated by the lower polyline delimiting the shaded area. The preemption thresholds for τ_1 is equal to the highest regular priority, which are not shown clearly due to their overlapping the boundary. For any valid assignment, the corresponding polyline must be located within the shaded area delimited by the two polylines for the minimal and maximal assignments (including the boundary) based on Theorem 4. The horizontal dotted line indicates the boundary case FPNP as the preemption threshold of each task is equal to the highest regular priority 1, while the diagonal dotted line indicates the boundary

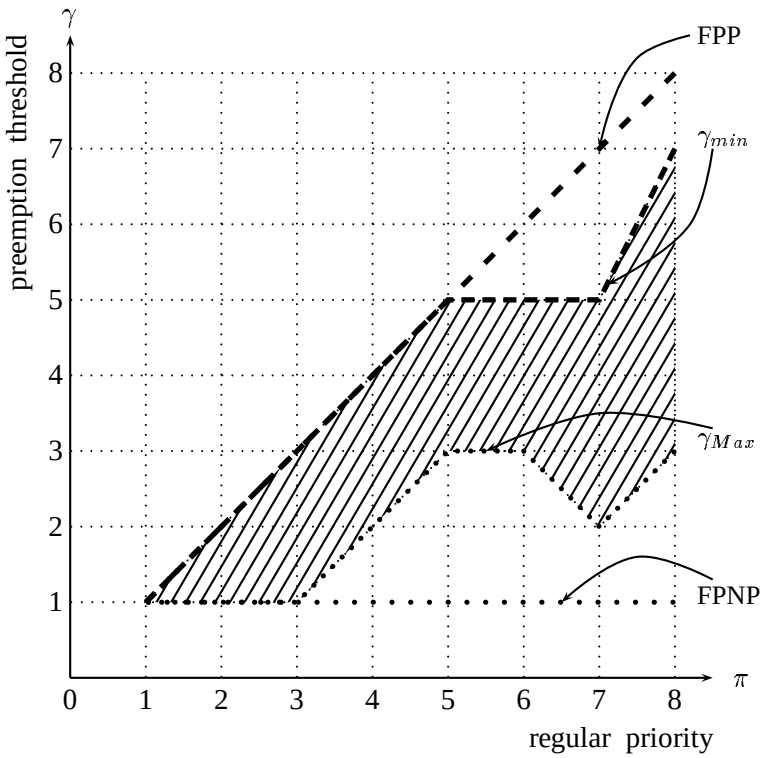


Figure 3.4: All Possible Valid Assignments

case FPP as the preemption threshold of each task is equal to its regular priority. The area above the diagonal dotted line is invalid as the preemption threshold of a task is lower than its regular priority, which is unreasonable (see the top of page 7). The area below the horizontal dotted line is invalid as the highest preemption threshold is equal to the highest regular priority. Note, not each assignment in the shaded area is valid. For example, assignment $\{1, 1, 1, 3, 4, 4, 3, 4\}$ is invalid. In fact, there are 3240 distinct assignments delimited by γ_{min} and γ_{Max}, only 1512 of them are valid (verified by exhaustive search).

3.6 Computing Minimal and Maximal Assignments

In Section 3.5.3, the minimal assignment γ_{min} and the maximal assignment γ_{Max} are assumed to be available. In practice, this assumption is not true. To generate more valid assignments when a task set with predefined regular priority is schedulable by FPPT, the minimal and maximal

assignment must be calculated first.

When a task set with predefined regular priority is schedulable by FPPT, the research work in [74] presented an effective algorithm to find a valid preemption threshold assignment. The authors do not point out that the assignment obtained is minimal. The research work in [60] presented an algorithm to compute the maximal assignment provided that a valid assignment is already known. By starting with a valid assignment, the algorithm calculates γ_{Max} but the authors do not prove formally the result is maximal.

The known algorithm to compute the minimal assignment starts its computation from FPP, one boundary case of FPPT. As FPPT has two boundary cases – FPP and FPNP, it is reasonable to consider computing the minimal assignment from FPNP, the other boundary case of FPPT. This section presents algorithms to compute minimal and maximal assignments effectively. In addition, the known algorithm to compute the maximal assignment starts its computation with a valid assignment. This section presents an algorithm to compute minimal assignment by starting from the maximal assignment. Furthermore, it is shown that the minimal and maximal assignments can be calculated by starting from either boundary case and the time complexity is the same. Finally, the algorithm to compute the maximal assignment can be used to perform a schedulability test for FPPT similar to the algorithm to compute the minimal assignment. The reason to explore effective algorithms to compute the minimal and maximal assignments starting from both boundary cases is whether computing γ_{min} (γ_{Max}) from FPP is more expensive than from FPNP depends on a given task set. Under a multiprocessor architecture, it is reasonable to compute γ_{min} (γ_{Max}) using both approaches and stop when a result is found. Note, a multiprocessor architecture is only used to compute γ_{min} (γ_{Max}), which does not imply that FPPT works on a multiprocessor architecture. Instead, FPPT works on a uniprocessor architecture.

3.6.1 Computing Minimal Assignment from FPP

For convenience, the assignment algorithm in [74] is called *FindMinFromFPP*. The pseudo-code is shown in Figure 3.5 (see Figure 2 in [74] for details). At the beginning, the algorithm sets the initial values for π and γ, which corresponds to lines 1 to 4, where $\pi_i = i$ and $\gamma_i = \pi_i$ for $1 \leq i \leq n$. Then the algorithm tries to calculate the minimal preemption threshold for each task from the task with the lowest regular priority to the task with the highest regular priority, which corresponds to lines 5 to 10. For any task τ_i, whenever it cannot meet its deadline with the current preemption threshold, the algorithm increases the preemption threshold by one. If

the increased preemption threshold is higher than the highest regular priority π_1, then there is no valid preemption assignment that can make τ_i schedulable and the algorithm stops. Otherwise, the algorithm repeats this increment and checking if the current task can meet its deadline. Lines 6 to 9 performs this computation for each task.

```
1   for ( i ← 1 to n )
2     π_i ← i;
3     γ_i ← π_i;
4   endfor
5   for ( i ← n to 1 )
6     while ( WCRT(π_i, γ_i) > D_i)
7       γ_i ← γ_i − 1; // increase the preemption threshold of τ_i by 1
8       if (γ_i < π_1) then return FAIL;
9     endwhile
10  endfor
11  return γ;
```

Figure 3.5: Pseudo-code for *FindMinFromFPP*

Theorem 5

Algorithm FindMinFromFPP is guaranteed to find γ_{min} if γ_{min} exists.

Proof:

1. The searching direction and the starting point are reasonable. First, the starting point is $\gamma_i = \pi_i$ for $1 \leq i \leq n$. When $\gamma_i = \pi_1$, if τ_i cannot meet its deadline, it indicates that the task is unschedulable by any preemption threshold and the algorithm stops based on Lemma **INVALID**. Second, the searching direction is from the task with the lowest regular priority to the task with the highest regular priority. Based on Corollary **AFFECT**, changing the preemption threshold of any task with a higher regular priority cannot affect the worst case response time of any task with a lower regular priority. In addition, when task τ_i is considered, all tasks with regular priorities lower than τ_i can already meet their deadlines. Thus, the validity of γ_i for τ_i cannot affect the validity of all other preemption thresholds of tasks with regular priorities lower than τ_i.

2. Suppose γ_{min} does not exist but the algorithm returns assignment γ. Based on the searching direction, function *WCRT*(π_i, γ_i) is called for each task. As any assignment of a preemption threshold to a task does not affect the validity of tasks with lower regular priorities,

after the algorithm checks the task with highest regular priority, γ is found to be valid. Based on Theorem 4, $\gamma_{min} \preccurlyeq \gamma \preccurlyeq \gamma_{Max}$, indicating that γ_{min} exists, which is a contradiction. Thus, the algorithm must return FAIL when γ_{min} does not exist.

3. Suppose γ_{min} exists but the algorithm returns γ. Similar to the previous step, γ is valid and $\gamma_{min} \preccurlyeq \gamma \preccurlyeq \gamma_{Max}$. Suppose k is maximal such that $\gamma_k < \gamma_{m_k}$. When τ_k is checked, its interference from tasks with higher regular priorities under γ_k is not greater than under γ_{m_k} but with the same blocking time from a task with a lower regular priority. As γ_{min} is valid and the initial preemption threshold for τ_k is π_k, the algorithm finds γ_{m_k} is valid for τ_k and the algorithm stops checking before it reaches γ_k. Then the final preemption threshold for τ_k is not higher than γ_{m_k}, resulting in $\gamma_k \geq \gamma_{m_k}$, which is a contradiction. Thus, no such k exists and $\gamma = \gamma_{min}$.

Therefore, the algorithm returns FAIL when γ_{min} does not exist; otherwise, it returns γ_{min}. ■

Theorem 5 guarantees that given a task set, if algorithm *FindMinFromFPP* finds a valid assignment γ, then any other valid assignment γ' satisfies $\gamma \preccurlyeq \gamma'$. In the pseudo-code, the function *WCRT*(π_i, γ_i) is used to compute the worst case response time of task τ_i based on the regular priority assignment π and the preemption threshold assignment γ. When the function *WCRT*(π_i, γ_i) is called, it applies Formulae 2.16, 2.17, 2.18, 2.19, and 2.20. As Formulae 2.17, 2.18, and 2.19 are solved by iteration, the time complexity for solving these formulae is non-deterministic, resulting in the time complexity for *WCRT*(π_i, γ_i) being non-deterministic, too. For convenience, the time complexity for *WCRT*(π_i, γ_i) is assumed to be $O(r)$, where r is not constant and depends on the task set, regular priority, and preemption threshold. This assumption is different from the research work in [74] where the time complexity for *WCRT*(π_i, γ_i) is assumed to be constant. The worst case for r occurs when all jobs in a major cycle must be checked as each iteration only deals with one job, which is equal to

$$\begin{aligned} \sum_{i=1}^{n} \frac{T_{mc}}{T_i} &= T_{mc} \sum_{i=1}^{n} \frac{1}{T_i} \\ &\leq T_{mc} \cdot n \qquad \text{as } T_i \text{ is an integer} \end{aligned}$$

Thus, $T_{mc} \cdot n$ can be considered to be r in the worst case. Because of the double nested loops in the pseudo-code, where the outer for-loop iterates n times, and the inner while loop iterates i times in the worst case for each task τ_i. Hence, the time complexity of the algorithm is $O(n^2 \cdot r)$ where n is the number of tasks and the complexity is pseudo-polynomial. In this thesis, r stands

for the cost of function $WCRT(\pi_i, \gamma_i)$ in the worst case and $T_{mc} \cdot n$ is the worst case cost, where n is the number of tasks in a periodic task set.

3.6.2 Computing Maximal Assignment from Minimal Assignment

Based on Theorem 4, all valid assignments for a task set schedulable by FPPT with predefined regular priority are delimited by γ_{min} and γ_{Max}. This section presents an algorithm called *FindMaxFromMin* to calculate the maximal assignment for a task set schedulable by FPPT with predefined regular priority by starting with the minimal assignment. *FindMaxFromMin* is similar to the algorithm to compute the maximal assignment presented in [60], which starts its computation with any valid assignment. However, the algorithm in [60] did not formally prove that its result is maximal. In fact, *FindMaxFromMin* can compute the maximal assignment by starting with any valid assignment. Hence, *FindMaxFromMin* rephrases the algorithm in [60].

The pseudo-code for algorithm *FindingMaxFromMin* is shown in Figure 3.6 (see Figure 4 in [60] for details). First, it assumes the task set is schedulable by FPPT and γ_{min} is calculated by using *FindMinFromFPP* (There is another approach to calculate γ_{min} in Section 3.6.4), and uses γ_{min} as the starting point to calculate γ_{Max}, which corresponds to lines 1 to 4. Denote the tentative maximal assignment as γ. Second, starting from the task with highest regular priority to the task with lowest regular priority, for each task τ_i, the algorithm increases the preemption threshold γ_i by one, which corresponds to line 7 in Figure 3.6. The worst case response time of the potentially affected task τ_k is recalculated, where k is equal to the new preemption threshold γ_i. If τ_k misses its deadline where $\pi_k = \gamma_i$, then the increasing the preemption threshold of γ_i is invalid. Then γ_i is reduced to its immediately previous value and the next task is considered, which corresponds to lines 5 to 14 in Figure 3.6. If τ_k still can meet its deadline, the algorithm repeats the increment and checks the worst case response time of the next potentially affected task. For the same reason as the previous algorithm, the time complexity of the algorithm is $O(n^2 \cdot r)$.

Theorem 6

Algorithm FindMaxFromMin calculates the maximal valid assignment.

Proof:

1. The searching direction and the starting point are guaranteed to find a valid assignment. First, starting from γ_{min} indicates each task can meet its deadline at the beginning. Second,

```
1    for ( i ← 1 to n )
2        π_i ← i;
3        γ_i ← γ_{m_i};
4    endfor
5    for (i ← 1 to n)
6        while (γ_i > π_1) // still have space to increase the preemption threshold of τ_i
7            γ_i ← γ_i − 1; // increase the preemption threshold of τ_i by 1
8            k ← γ_i;
9            if (WCRT(π_k, γ_k) > D_k) // affects τ_k
10               γ_i ← γ_i + 1;
11               break;
12           endif;
13       endwhile
14   endfor
15   return γ;
```

Figure 3.6: Pseudo-code for *FindMaxFromMin*

from the task with highest regular priority to the task with lowest regular priority, when task τ_i is considered, those tasks τ_1, τ_2, . . ., τ_{i-1} are already schedulable. When increasing the preemption threshold of τ_i, it can only affect tasks τ_1, τ_2, . . ., τ_{i-1} but it cannot affect tasks τ_{i+1}, τ_{i+2}, . . ., τ_n based on Corollary **AFFECT**. For task τ_i itself, its worst case response time cannot be worse than before as it gets less interference from tasks with higher regular priorities. Then the algorithm checks the affected tasks τ_1, τ_2, . . ., τ_{i-1}. If any of them, say τ_j misses its deadline, it implies that τ_i contributes the maximal blocking time. Note the preemption threshold of τ_j is already maximized. Thus, it cannot be increased any more. The only option is to decrease the preemption threshold of τ_i to its immediately previous value so τ_i no longer blocks τ_j. If all of the tasks are still schedulable, then the new assignment is valid. After all tasks are checked, a valid assignment is obtained.

2. Suppose the algorithm returns γ instead of γ_{Max}. As γ is valid, then $\gamma_{min} \preccurlyeq \gamma \preccurlyeq \gamma_{Max}$ based on Theorem 4. Suppose k is minimal such that $\gamma_k > \gamma_{M_k}$. When τ_k is checked, the algorithm checks γ_{m_k}, $\gamma_{m_k} - 1$, $\gamma_{m_k} - 2$, . . ., γ_{M_k} sequentially if each of them is valid for τ_k. As both γ and γ_{Max} are valid and $\gamma_i = \gamma_{M_i}$ for $1 \leq i \leq k-1$, the algorithm finds that $\{\gamma_{M_1}, \gamma_{M_2}, \ldots, \gamma_{M_{k-1}}, \gamma_{M_k} - j, \gamma_{k+1}, \gamma_{k+2}, \ldots, \gamma_n\}$ is valid for $0 \leq j \leq \gamma_{m_k} - \gamma_{M_k}$ based on Lemma 1. Thus, after the algorithm finds γ_k is valid, it continues to check until γ_{M_k}. Then the final preemption threshold for τ_k cannot be lower than γ_{M_k}, resulting in $\gamma_k \leq \gamma_{M_k}$, which is a contradiction. Thus, no such k exists and $\gamma = \gamma_{Max}$.

Therefore, the algorithm returns γ_{Max}. ■

If γ_{min} is substituted by any other valid assignment, the algorithm still works as the algorithm only needs a valid assignment to start its computation. Based on Corollary **AFFECT**, increasing the preemption threshold of any task τ_i cannot affect the worst case response time of any other task with a lower regular priority lower than τ_i. Thus, starting with any valid assignment, increasing the preemption threshold of any task τ_i cannot affect the schedulability of any other task with regular priority lower than τ_i.

One weak point in algorithm *FindMaxFromMin* is that it depends on γ_{min} or any other valid assignment.

3.6.3 Computing Maximal Assignment from FPP

Based on Theorem 5 and 6, γ_{Max} can be computed by starting from FPP if existing. If γ_{Max} existing, γ_{min} also exists based on Theorem 4. First, *FindMinFromFPP* calculates γ_{min} if existing. Second, if γ_{min} exists and calculated, *FindMaxFromMin* can calculate γ_{Max} by starting from γ_{min}. The sequential combinations of *FindMinFromFPP* and *FindMaxFromMin* form a new algorithm called *FindMaxFromFPP*. For the same reason as the previous algorithm, the time complexity of the algorithm is $O(n^2 \cdot r)$.

3.6.4 Computing Minimal Assignment from FPNP

Corresponding to the algorithm to compute γ_{min} by starting from FPP, a new algorithm to compute γ_{min} by starting from FPNP is presented, called *FindMinFromFPNP*, and its pseudo-code is shown in Figure 3.7.

At the beginning, the algorithm sets the initial values for π and γ, which corresponds to lines 1 to 4, where $\pi_i = i$ and $\gamma_i = \pi_1$ for $1 \leq i \leq n$. Then the algorithm tries to calculate the minimal preemption threshold for each task starting from the task with the lowest regular priority to the task with the highest regular priority, which corresponds to lines 5 to 13. For any task τ_i, the algorithm first checks if its worst case response time is not greater than its deadline when its preemption threshold is highest. If its worst case response time is greater than its deadline, then there is no valid preemption assignment that can make τ_i schedulable and the algorithm stops. Lines 6 and 7 perform this check. If the worst case response time is not greater than its deadline, the preemption threshold of the task is decreased by one; note that this decreasing does

not increase the worst case response time of any task with regular priority higher or lower than τ_i, it may only affect the worst case response time of τ_i. Thus, only the schedulability of τ_i needs to be checked. If it is still schedulable, then repeat the process until either $\gamma_i = \pi_i$ and τ_i is still schedulable, or τ_i is unschedulable. Lines 8 to 11 perform this computation. If τ_i is schedulable and $\pi_i = \gamma_i$, switch to the next task directly as there is no more space to decrease the preemption threshold of the task. If τ_i is unschedulable, increase the preemption threshold of the task by one and switch to the next task. Line 12 performs this check.

```
1   for ( i ← 1 to n )
2      π_i ← i;
3      γ_i ← π_1; // starting from FPNP
4   endfor
5   for ( i ← n to 1 )
6      R_i ← WCRT(π_i, γ_i);
7      if (R_i > D_i) then return FAIL;
8      while ( (R_i ≤ D_i) and (γ_i < π_i) )
9         γ_i ← γ_i + 1;
10        R_i ← WCRT(π_i, γ_i);
11     endwhile
12     if (R_i > D_i) then γ_i ← γ_i − 1;
13  endfor
14  return γ;
```

Figure 3.7: Pseudo-code for *FindMinFromFPNP*

Theorem 7

Algorithm FindMinFromFPNP is guaranteed to find γ_{min} if γ_{min} exists.

Proof:

1. The searching direction and the starting point are reasonable. First, the starting point is $\gamma_i = \pi_1$ for $1 \leq i \leq n$. When $\gamma_i = \pi_1$, if τ_i cannot meet its deadline, it indicates that the task is unschedulable by any preemption threshold and the algorithm stops based on Lemma **INVALID**. Second, the searching direction is from the task with the lowest regular priority to the task with the highest regular priority. Based on Corollary **AFFECT**, changing the preemption threshold of any task with a higher regular priority cannot affect the worst case response time of any task with a lower regular priority. In addition, when task τ_i is considered, all tasks with regular priorities lower than τ_i can already meet their

deadlines. Thus, the validity of γ_i for τ_i cannot affect the validity of all other preemption thresholds of tasks with regular priorities lower than τ_i.

2. Suppose γ_{min} does not exist but the algorithm returns assignment γ. Based on the searching direction, function *WCRT*(π_i, γ_i) is called for each task. As any assignment of a preemption threshold to a task does not affect the validity of tasks with lower regular priorities, after the algorithm checks the task with highest regular priority, γ is found to be valid. Based on Theorem 4, $\gamma_{min} \preccurlyeq \gamma \preccurlyeq \gamma_{Max}$, indicating that γ_{min} exists, which is a contradiction. Thus, the algorithm must return FAIL when γ_{min} does not exist.

3. Suppose γ_{min} exists but the algorithm returns γ. Similar to the previous step, γ is valid and $\gamma_{min} \preccurlyeq \gamma \preccurlyeq \gamma_{Max}$. Suppose k is maximal such that $\gamma_k < \gamma_{m_k}$. When τ_k is checked, its interference from tasks with higher regular priorities under γ_k is not greater than under γ_{m_k} but it has the same blocking time from a task with a lower regular priority. As γ_{min} is valid, the algorithm finds γ_k is valid for τ_k and the algorithm continues to check $\gamma_k + 1$, $\gamma_k + 2$,. . ., $\gamma_{m_k} - 1$, because under any of these preemption thresholds, τ_k gets shorter interference than γ_{m_k} from tasks with higher regular priorities but with the same blocking time. After finding γ_{m_k} is valid, then the final preemption threshold for τ_k is not higher than γ_{m_k}, resulting in $\gamma_k \geq \gamma_{m_k}$, which is a contradiction to the assumption that k is maximal such that $\gamma_k < \gamma_{m_k}$. Thus, no such k exists and $\gamma = \gamma_{min}$.

Therefore, the algorithm returns FAIL when γ_{min} does not exist; otherwise, it returns γ_{min}. ■

Due to the double nested loops in the pseudo-code, where the outer for-loop iterates n times and the inner while-loop iterates i times in the worst case, the time complexity of the algorithm is $O(n^2 \cdot r)$ where n is the number of tasks.

3.6.5 Computing Minimal Assignment from Maximal Assignment

As γ_{Max} can be calculated by starting with γ_{min}, it is reasonable to consider calculating γ_{min} by starting with γ_{Max}. For convenience, such an algorithm is called *FindMinFromMax* and its pseudo-code is shown in Figure 3.8.

At the beginning, lines 1 to 4 set the initial values for π and γ, where $\pi_i = i$ for $1 \leq i \leq n$ and $\gamma = \gamma_{Max}$. Then the algorithm checks each task from the task with lowest regular priority to the task with highest regular priority, which corresponds to lines 5 to 13. For each task τ_i, whenever

its preemption threshold is still equal to or higher than its regular priority, the algorithm decreases the preemption threshold of task τ_i by one. Note, decreasing the preemption threshold of task τ_i cannot affect the worst case response time of any task with regular priorities lower than τ_i based on Corollary **AFFECT**. As those tasks with regular priorities lower than τ_i cannot miss their deadlines, then the decrement cannot affect the schedulability of those tasks with lower regular priorities. In addition, the decrement cannot make the worst case response time of those tasks with regular priorities higher than τ_i longer based on Formula 2.16. However, the worst case response time of τ_i may be longer as it can get more interference from tasks with higher regular priorities. Thus, only the worst case response time of τ_i is checked, which corresponds to lines 8 to 11. If task τ_i misses its deadline, the algorithm increases the preemption threshold by one and switches to the next task.

```
1     for ( i ← 1 to n )
2         π_i ← i;
3         γ_i ← γ_{M_i};
4     endfor
5     for (i ← n to 1)
6         while (γ_i < π_i) // still have space to decrease the preemption threshold of τ_i
7             γ_i ← γ_i + 1; // decrease the preemption threshold of τ_i by 1
8             if (WCRT(π_i, γ_i) > D_i)
9                 γ_i ← γ_i − 1;
10                break;
11            endif;
12        endwhile
13    endfor
14    return γ;
```

Figure 3.8: Pseudo-code for *FindMinFromMax*

Theorem 8

Algorithm FindMinFromMax calculates γ_{min}.

Proof: The proof is similar to the proof of Theorem 7. ■

If γ_{Max} is substituted by any other valid assignment, the algorithm still works as the algorithm only needs a valid assignment to start its computation. Based on Corollary **AFFECT**, decreasing the preemption threshold of any task τ_i cannot affect the worst case response time of any other task with a lower regular priority lower than τ_i. Thus, starting with any valid assignment, decreasing the preemption threshold of any task τ_i cannot affect the schedulability of any

other task with a regular priority lower than τ_i, and it cannot affect the schedulability of any task with a regular priority higher than τ_i. Only the worst case response time of task τ_i is required to be checked. For the same reason as the previous algorithm, the time complexity of the algorithm is $O(n^2 \cdot r)$.

3.6.6 Computing Maximal Assignment from FPNP

Corresponding to the algorithm *FindMinFromFPNP* to compute the minimal assignment from FPNP, it is reasonable to compute the maximal assignment from FPNP, resulting in an algorithm called *FindMaxFromFPNP*, whose pseudo-code is shown in Figure 3.9. At the beginning, the algorithm sets the initial values for the regular priorities and preemption thresholds, which corresponds to lines 1 to 4. Task set S stores a subset of tasks that are already schedulable, and it is initialized to the empty set in line 5. Then from the task with highest regular priority to the task with lowest regular priority, for each task τ_i, it is appended to S, which corresponds to line 7. Appending τ_i requires rechecking the schedulability of τ_1, τ_2, . . ., τ_{i-1}, which corresponds to lines 8 to 10. *WCRT*(π_j, γ_j, S) is similar to *WCRT*(π_j, γ_j), which computes the worst case response time of τ_j in subset S, with the corresponding regular priority and preemption threshold. For each such task τ_j, if it becomes unschedulable, then the preemption threshold of τ_i is set to the regular priority of τ_j plus 1, i.e., set $\pi_j + 1$ to γ_i. After checking all tasks with regular priorities higher than τ_i, τ_i is checked, which corresponds to line 11. The time complexity of algorithm *FindMaxFromFPNP* is $O(n^2 \cdot r)$. Note, this algorithm does not need to start with a valid assignment.

Theorem 9

Algorithm FindMaxFromFPNP is guaranteed to find γ_{Max} if γ_{Max} exists.

Proof: Let $S_k = \{\tau_1, \ldots, \tau_k\}$.

1. The searching direction, the starting point, and the incremental schedulability test for FPPT are guaranteed to find a valid assignment if existing. First, the starting point is $\gamma_i = \pi_1$ for $1 \leq i \leq n$ and the subset of tasks schedulable is the empty set. Second, from the task with highest regular priority to the task with lowest regular priority, when task τ_i is considered, those tasks in S are already schedulable. After τ_i is appended to S, it may affect the schedulability of all other tasks in S, which must be rechecked. For each task τ_j in S (excluding τ_i), they are checked from highest regular priority to lowest priority, to

```
1   for ( i ← 1 to n )
2      π_i ← i;
3      γ_i ← π_1;
4   endfor
5   S ← ∅;
6   for (i ← 1 to n)
7      S ← S ∪ {τ_i};
8      for ( j ← 1 to i − 1 )
9         if (WCRT(π_j, γ_j, S) > D_j) then γ_i ← π_j + 1;
10     endfor
11     if (WCRT(π_i, γ_i, S) > D_i) then return FAILURE;
12  endfor
13  return γ;
```

Figure 3.9: Pseudo-code for *FindMaxFromFPNP*

determine if τ_j is schedulable. If τ_j is unschedulable, it implies thats τ_i contributes the maximal blocking time for τ_j because before τ_i was appended, τ_j was schedulable. Thus, changing the preemption thresholds for tasks with regular priorities lower than τ_j but higher than τ_i cannot affect the schedulability of τ_j. That is why only τ_i needs to be considered. The options are either to decrease the preemption threshold of τ_i to the regular priority of τ_j plus 1 (set $\pi_j + 1$ to γ_i) so τ_i cannot block τ_j or increase the preemption threshold of τ_j. The latter is impossible because the preemption threshold of τ_j is either π_1 or increasing γ_j makes other task(s) with regular priorities higher than π_j unschedulable. When τ_i does not affect any such τ_j, then its schedulability must be checked. When τ_i is unschedulable, then no preemption threshold can make it schedulable as changing the preemption thresholds of any such τ_j cannot help based on searching its preemption threshold is from the highest to the lowest.

2. Suppose γ_{Max} does not exist but the algorithm returns assignment γ. Based on the searching direction, when τ_k is checked in S_k, the algorithm first guarantees that τ_k does not affect the schedulability of all tasks other than τ_k, i.e., τ_k's preemption threshold must not affect the schedulability of tasks with higher regular priorities. After the task with lowest regular priority is checked, the algorithm finds that γ is valid. Based on Theorem 4, $\gamma_{min} \preccurlyeq \gamma \preccurlyeq \gamma_{Max}$, indicating that γ_{Max} exists, which is a contradiction. Thus, the algorithm must return FAIL when γ_{Max} does not exist.

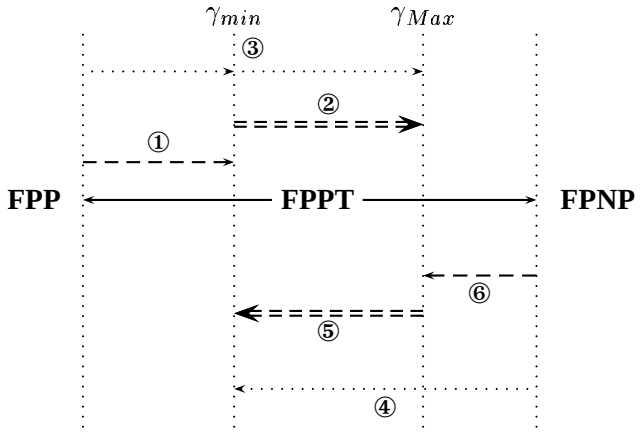
3. Suppose γ_{Max} exists but the algorithm returns γ. Similar to the previous step, γ is valid

and $\gamma_{min} \preccurlyeq \gamma \preccurlyeq \gamma_{Max}$. Suppose k is minimal such that $\gamma_k > \gamma_{M_k}$. Let $\Gamma_k = \{\gamma_{M_1}, \gamma_{M_2}, \ldots, \gamma_{M_k}\}$. Note, Γ_k is valid for S_k as the schedulability test for FPPT is incremental based on Theorem 3 (see Section 3.4). Due to searching from π_1 to π_k, when τ_k is checked in S_k, the algorithm finds that Γ_k is valid for S_k and assigns γ_{M_k} to τ_k as $\gamma_k > \gamma_{M_k}$. Thus, the algorithm does not check γ_k, which is a contradiction. Thus, no such k exists and $\gamma = \gamma_{Max}$.

Therefore, the algorithm returns FAIL when γ_{Max} does not exist; otherwise, it returns γ_{Max}. ■

3.6.7 The Whole Picture

Based on the previous six sections, the relationships among those six algorithms is shown in Figure 3.10.



① : *FindMinFromFPP*
② : *FindMaxFromMin*
③ : *FindMaxFromFPP*
④ : *FindMinFromFPNP*
⑤ : *FindMinFromMax*
⑥ : *FindMaxFromFPNP*

Figure 3.10: Relationships Among Six Algorithms

The diagram shows that FPP and FPNP are two boundary cases of FPPT. When a task set with predefined regular priority is unschedulable by FPPT, neither γ_{min} nor γ_{Max} exists. When a task set with predefined regular priority is schedulable by FPPT, its γ_{min} can be calculated by starting from either FPP or FPNP, which is indicated by algorithms ① and ④ respectively. In addition,

whenever either γ_{min} or γ_{max} is known, the other can be calculated, which is indicated by algorithms ② and ⑤ respectively. Furthermore, no algorithm to compute γ_{Max} by starting directly from FPP is presented. However, an algorithm to compute γ_{Max} by starting indirectly from FPP is presented, which is indicated by algorithm ③. Algorithm ③ is the sequential combinations of algorithms ① and ②. An algorithm to compute γ_{Max} by starting directly from FPNP is presented, which is indicated by algorithm ⑥.

Among the six algorithms, algorithm ① is presented in [74] and algorithm ② is presented in [60]. However, the validity of each algorithm is not formally proved. This thesis provides a formal proof for the validity of each algorithm. Algorithms ④, ⑤, and ⑥ including the formal proofs of their validity are contributions of this thesis.

3.7 Comparing FPPT with PIP

FPPT and PIP seem similar as both FPPT and PIP apply two priorities to schedule a task set. One of the priorities is fixed and predefined. In PIP, the fixed-priority is called the static priority; in FPPT it is called the regular priority. Both PIP and FPPT apply a fixed-priority to compete for the processor. The other priority must be calculated. In PIP, the other priority is called the inherited priority and it is dynamic; in FPPT it is the preemption threshold.

However, PIP and FPPT are different. First, once the preemption threshold of a task is calculated in FPPT, it is fixed during the execution of the task, while the inherited priority of a task in PIP is determined dynamically during its execution. This is due to the property of PIP: if a task τ_H with higher static priority is blocked by another task τ_L with lower static priority because τ_L is using an exclusively-shared resource also needed by τ_H, then τ_L inherits the priority of τ_H, i.e., the inherited priority of τ_L is set to the static priority of τ_H. In other words, τ_L inherits the static priority of τ_H. This inheritance can be transitive if more ready tasks with higher static priorities need to access the same exclusively-shared resource.

Second, the purpose of the dual-priority is different. In PIP, the inherited priority is used to synchronize the concurrent accesses to exclusively-shared resource so that a task with higher static priority can be blocked by at most one task with lower static priority when they access the same exclusively-shared resource. Tasks still compete for the processor with their static priorities. When a task starts to run, its running priority is upgraded to its inherited priority. Once the task with lower static priority releases the exclusively-shared resource, its running priority returns to

its immediately previous inherited priority. Similarly, in FPPT, the regular priority is used to compete for the processor. When a task starts to run, its running priority is upgraded to its preemption threshold. Only those tasks with regular priorities higher than the preemption threshold of the currently running task can preempt the execution of the latter. Thus, FPPT applies dual-priority to guarantee the deadlines of tasks that are independent. If tasks are not independent, then FPPT does not work without an extra resource management protocol.

Third, in PIP the inherited priority guarantees that a task with lower static priority that is already using an exclusively-shared resource cannot be preempted when the same resource is also required by another ready task with higher static priority. (Deadlocks may still occur. Priority ceiling protocol (PCP) guarantees that no deadlock can occur.) In FPPT, the preemption threshold just controls the range of tasks that can preempt the execution of the currently running task.

3.8 Summary

This chapter proves that the FPPT schedulability test is robust under its critical instant. When a task set is schedulable by FPPT with predefined regular priority, there may exist multiple valid assignments, among which there are two special assignments: minimal and maximal. A partial order relationship is defined among these valid assignments. This chapter shows that all valid assignments are delimited by the minimal and maximal assignments. Effective algorithms to compute both the minimal assignment and maximal assignment from FPNP are presented. In addition, it is shown that the minimal (maximal) assignment can be calculated from the maximal (minimal) assignment and the corresponding algorithms are presented. All algorithms are proved correct. It is shown that the maximal assignment can be calculated indirectly from FPP. Furthermore, this chapter formally proves that results of algorithms *FindMinFromFPP* and *FindMaxFromMin* are minimal and maximal, respectively. The relationships among these algorithms indicate that FPPT does generalize both FPP and FPNP and provide more flexibilities for real-time scheduling.

Chapter 4

Reservation-Based Algorithm

This chapter examines RBA to schedule periodic tasks with EDF, RMA, or FPPT while aperiodic tasks are executed with the unused time left by periodic tasks. RBA is selected because it has the potential to minimize the start-time variance among randomly arriving aperiodic tasks while guaranteeing the deadlines of periodic tasks. The main contributions are two algorithms to calculate the maximal size of the reserved bandwidth when RMA and FPPT are used, respectively. Furthermore, the period of the periodic server is one time unit instead of one unit cycle. Hence, within the reservation band, it is possible to start an aperiodic task in one time unit and continue its execution every subsequent time unit, therefore providing immediate response.

4.1 Motivations

Based on the research result in Chapter 3, RMA is only a special case of FPP, which is a special case of FPPT. This property implies that whenever an independent task set is schedulable by RMA, it must also be schedulable by FPP. Of course, it must also be schedulable by FPPT. Thus, the range of schedulable independent task sets of FPPT is larger than that of FPP, which is larger than that of RMA. In addition, an independent periodic task set is schedulable if and only if the processor utilization of the task set is not greater than 1 when EDF is used. As any independent task set with processor utilization greater than 1 cannot be schedulable by any scheduler, then any other schedulers cannot be better than EDF. Thus, the range of schedulable independent task sets of EDF is the largest. In other words, when tasks are independent, EDF is stronger than FPPT and FPPT is stronger than either FPP or FPNP. The range of schedulable independent task sets are shown in Figure 4.1. Note, there exist task sets schedulable by both FPP and FPNP.

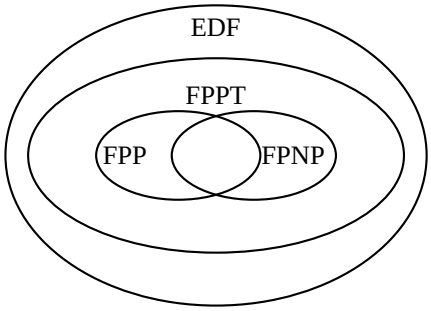


Figure 4.1: Schedulability Relationship among FPP, FPNP, FPPT, and EDF

Given an independent task set with processor utilization U, the maximal reservation in each time unit is $1 - U$. EDF can allow all unused time left by periodic tasks to be reserved evenly in each time unit for servicing aperiodic tasks, resulting in a maximized reservation size. For task sets schedulable by RMA, the reservation size cannot be better than that of EDF, which is explained in detail in Section 4.3. For task sets schedulable by FPPT, it can obtain a reservation size not worse than RMA. Therefore, if tasks are independent and the processor utilization is the only consideration, EDF is the best choice, followed by FPPT and RMA.

However, in practice, conditions can vary. First, EDF is dynamic and has high run-time cost; RMA and FPPT are both static and can perform schedulability tests offline. Second, tasks may not be independent and resource management protocols are necessary and most well-known resource management protocols are based on RMA. Thus, the combinations of RBA with these different schedulers for periodic tasks make sense and have their practical usages. For example, when resource management protocols are necessary, RMA with RBA should be considered. When the processor utilization is the only consideration and the run-time cost can be ignored, EDF with RBA is the best choice. When tasks are independent and the run-time cost cannot be ignored, FPPT with RBA is better than either EDF or RMA as it can get a reservation size not worse than RMA and avoid the high run-time cost of EDF. These are the reasons to explore the combinations of RBA and EDF, RMA, and FPPT, respectively.

4.2 Location of Bandwidth Reserved

With bandwidth reservation, the processor time is categorized into two bands. The first one is the bandwidth reserved. The second one is the time for periodic tasks. Part (a) in Figure 4.2

shows this categorization. In part (a), the locations of the specific time segments for periodic tasks and the bandwidth are unspecified, it only indicates the time ratio between periodic tasks and bandwidth. The specific location of the periodic server depends on the rule to consume the reservation [42]. A rule to consume a reservation specifies how the reservation is used. For example, if the reservation is used by polling, then the reservation is available at the beginning of each cycle of the periodic server. If the reservation is used by a deferrable server, then the reservation can be kept as long as possible in each cycle. Under this consuming rule, the computation time of the periodic server may be sliced over a period. Part (b) in Figure 4.2 shows the location of the reserved bandwidth when polling is used. This chapter does not consider consuming rules. Instead, it focuses on the computation of the maximal reservation size, which can be used by different consuming rules.

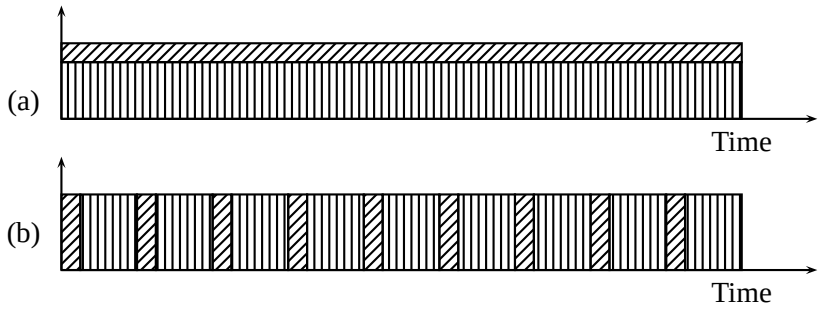


Figure 4.2: Location of Bandwidth Reserved

4.3 RBA with EDF

When EDF is used to schedule periodic tasks in RBA, this type of RBA is called *RBA_EDF* for convenience. The algorithm to compute the maximal size of the reserved bandwidth for *RBA_EDF* is trivial. In *RBA_EDF*, the periodic server is $\tau = (1, 1 - U)$, where U is the processor utilization of an independent task set. The reason is an independent task set is schedulable by EDF if and only if the processor utilization of the task set is not greater than 1 [45].

4.4 RBA with RMA

RBA_EDF can obtain a maximal reservation size when the processor utilization is the only consideration. However, the processor utilization is not always the only consideration in practice. First, periodic tasks may not be independent. For example, they compete for exclusively-shared resources. Shared resources are usually managed by resource management protocols. Many well-known resource management protocols are based on RMA [62] instead of EDF. Second, EDF has a high dynamic scheduling cost, which must be calculated into its scheduling. Thus, EDF is not always an appropriate choice in RBA. When RMA is used, the corresponding RBA is called *RBA_RMA* for convenience.

However, when RMA is used to schedule such a task set, it is schedulable if and only if L is not greater than 1, i.e., $L \leq 1$ (see Theorem ERMA2 in Section 2.1.1). It is shown that $L \geq U$, which is formally described in Lemma 4. Thus, whenever an independent task set is schedulable by RMA, it implies $1 \geq L \geq U$. Hence, the task set must be schedulable by EDF. In addition, given a task set with $U \leq 1$, L can be greater than 1. Thus, the task set may be schedulable by EDF but unschedulable by RMA. For example, given tasks $\tau_0 = (3, 1)$, $\tau_1 = (5, 2)$, $\tau_2 = (12, 3)$, $L = 1.08333$ and $L > 1$; $U = 0.983333$ and $U < 1$. The task set is schedulable by EDF but unschedulable by RMA. Therefore, EDF is more powerful than RMA if the processor utilization is the only consideration.

Lemma 4
Given periodic tasks $\tau_i = (T_i, C_i), 1 \leq i \leq n$, $T_1 \leq T_2 \leq \ldots \leq T_n$, then $L \geq U$.

Proof: Note, $U_i = \sum_{j=1}^{i} \frac{C_j}{T_j}$, which is the processor utilization of tasks $\tau_1, \tau_2, \ldots, \tau_i$ and $U_1 < U_2 < \ldots < U_n = U$.

1. Calculate $L_i(t)$,

$$\begin{aligned} L_i(t) &= \frac{\sum_{j=1}^{i} C_j \cdot \left\lceil \frac{t}{T_j} \right\rceil}{t} && \text{based on Formulae 2.3 and 2.4} \\ &\geq \frac{\sum_{j=1}^{i} C_j \cdot \frac{t}{T_j}}{t} && \text{as } \left\lceil \frac{t}{T_j} \right\rceil \geq \frac{t}{T_j} \\ &= \sum_{j=1}^{i} \frac{C_j}{T_j} \\ &= U_i \end{aligned}$$

2. Calculate L_i,

$$\begin{aligned} L_i &= \min_{\{0<t\leq T_i\}} L_i(t) && \text{based on Formula 2.5} \\ &\geq U_i && \text{as } L_i(t) \geq U_i \text{ in step 1} \end{aligned}$$

3. Calculate L,

$$\begin{aligned} L &= \max_{\{1\leq i\leq n\}} L_i && \text{based on Formula 2.8} \\ &\geq \max_{\{1\leq i\leq n\}} U_i && \text{as } L_i \geq U_i \text{ in step 2} \\ &= U_n && \text{as } U_1 < U_2 < \ldots < U_n \text{ at start of proof} \\ &= U \end{aligned}$$

■

4.4.1 Computing Reserved Bandwidth with RMA

When *RBA_RMA* performs a schedulability test on a periodic task set [63], it takes R_s as a parameter. The schedulability test algorithm can be used to calculate R_{lub} by trial-and-error, which is simpler than the algorithm in [63] to compute R_{lub}. The corresponding pseudo-code in Figure 4.3 performs an iterative approach using the schedulability test from [63] to determine R_{lub}. Note, the tentative R_{lub} acts as the parameter R_s in the schedulability test. If the step size is small

enough, then the result can be maximal. In fact, the trial-and-error approach can calculate R_{lub} quickly if the step size is refined by exponential increments or decrements during the calculation.

```
1   R_lub ← 0.0;
2   stepsize ← 0.01; // assume a stepsize of 0.01
3   if ( !RBA( R_lub) ) then return -1; // the task set is unschedulable by RMA
4   while ( RBA( R_lub ) )
5      R_lub ← R_lub + stepsize;
6   endwhile
7   return (R_lub − stepsize); // reservation size
```

Figure 4.3: Computing R_{lub} with Trial-and-Error

As mentioned, the schedulability test in [63] checks all jobs in one major cycle and performs physical scheduling, which is not a good approach. The next section presents an algorithm to calculate the maximal R_{lub}.

4.4.2 Maximal Reserved Bandwidth with RMA

Based on Theorem **ERMA2**, L_i is the minimal processor utilization to schedule the periodic tasks τ_1, τ_2, ..., τ_i using RMA. Each task τ_j, $1 \leq j \leq i$, cannot miss its deadline over $[0, T_i]$ if and only if $L_j \leq 1$. In each time unit from 0 to T_i, $1 - L_i$ is the amount of time left unused by periodic tasks τ_1, τ_2, ..., τ_i if $L_i < 1$. If this amount of time is used to service aperiodic tasks, it is the bandwidth to be reserved. Furthermore, as L is the minimal value to guarantee the deadlines of periodic tasks, $1 - L$ is the maximal amount of time that can be reserved in each time unit. Hence, the bandwidth to be reserved is maximal while guaranteeing the deadlines of periodic tasks scheduled by RMA. This observation is formally presented in the following theorem.

Theorem 10
Given periodic task set $S = \{\tau_i = (T_i, C_i) : 1 \leq i \leq n\}$, $T_1 \leq T_2 \leq \ldots \leq T_n$, if it is schedulable using RMA, then $1 - L$ is the maximal amount of processor time that can be reserved in each time unit to service aperiodic tasks without missing the deadline of any periodic task.

Proof: The proof is based on creating an extra periodic task $\tau_0 = (T_0, C_0) = (1, 1 - L)$. Consider periodic task set $S' = \{\tau_0\} \cup S$, $T_0 \leq T_1 \leq T_2 \leq \ldots \leq T_n$. Thus, Theorem **ERMA2** can be applied directly on S'.

1. If only task τ_0 is considered, based on Formulae 2.8, 2.3, 2.4, 2.7, $L' = L'_0 = L'_0(T_0) = \frac{W'_0(T_0)}{T_0} = \frac{C_0 \cdot \lceil \frac{T_0}{T_0} \rceil}{T_0} = \frac{C_0}{T_0} = C_0 = 1 - L$. As tasks τ_1, τ_2, ..., τ_n are schedulable by RMA, based on Theorem **ERMA2**, $0 < L \leq 1$, thus, $L' = 1 - L < 1$. Based on Theorem **ERMA2**, the task set composed of only task τ_0 is schedulable by RMA.

2. Consider tasks τ_0, τ_1, τ_2, ..., τ_i for $1 \leq i \leq n$. As tasks τ_1, τ_2, ..., τ_n are schedulable by RMA and RMA is robust, task set $S_i = \{\tau_1, \tau_2, \ldots, \tau_i\}$ is schedulable by RMA for $1 \leq i \leq n$. Based on Theorem **ERMA2**, the corresponding value of L, denoted as L_{S_i}, for S_i is not greater than 1 for $1 \leq i \leq n$. Then $L_{S_i} = \max_{\{1 \leq j \leq i\}} L_j$ for $1 \leq i \leq n$. Obviously, $L_{S_i} \leq L \leq 1$ for $1 \leq i \leq n$.

$$
\begin{aligned}
W'_i(t) &= \sum_{j=0}^{i} C_j \cdot \left\lceil \frac{t}{T_j} \right\rceil \qquad \text{based on Formula 2.3} \\
&= C_0 \cdot \left\lceil \frac{t}{T_0} \right\rceil + \sum_{j=1}^{i} C_j \cdot \left\lceil \frac{t}{T_j} \right\rceil \\
&= C_0 \cdot \left\lceil \frac{t}{1} \right\rceil + W_i(t) \qquad \text{based on Formula 2.3} \\
&= (1 - L) \cdot t + W_i(t)
\end{aligned}
$$

$$
\begin{aligned}
L'_i(t) &= \frac{W'_i(t)}{t} \qquad \text{based on Formula 2.4} \\
&= \frac{(1 - L) \cdot t + W_i(t)}{t} \\
&= \frac{(1 - L) \cdot t}{t} + \frac{W_i(t)}{t} \\
&= (1 - L) + L_i(t) \qquad \text{based on Formula 2.4}
\end{aligned}
$$

$$
\begin{aligned}
L'_i &= \min_{\{0 < t \leq T_i\}} L'_i(t) \qquad \text{based on Formula 2.5} \\
&= \min_{\{0 < t \leq T_i\}} ((1 - L) + L_i(t)) \\
&= (1 - L) + \min_{\{0 < t \leq T_i\}} L_i(t) \\
&= (1 - L) + L_i \qquad \text{based on Formula 2.5}
\end{aligned}
$$

$$
\begin{aligned}
L' &= \max_{\{0 \le j \le i\}} L'_j \qquad \text{based on Formula 2.8} \\
&= \max\Big(L'_0, \max_{\{1 \le j \le i\}} L'_j\Big) \\
&= \max\Big(L'_0(T_0), \max_{\{1 \le j \le i\}} L'_j\Big) \\
&= \max\Big(1 - L, \max_{\{1 \le j \le i\}} ((1 - L) + L_j)\Big) \\
&= \max\Big(1 - L, (1 - L) + \max_{\{1 \le j \le i\}} L_j\Big) \\
&= \max(1 - L, (1 - L) + L_{S_i}) \quad \text{as } L_{S_i} = \max_{\{1 \le j \le i\}} L_j \\
&= \max(1 - L, 1 - L + L_{S_i}) \\
&\le \max(1 - L, 1) \qquad \text{as } L_{S_i} \le L \le 1 \\
&= 1 \qquad \text{as } 0 < L \le 1
\end{aligned}
$$

Based on Theorem **ERMA2**, periodic tasks τ_0, τ_1, τ_2, . . ., τ_i are schedulable by RMA for $1 \le i \le n$. When $i = n$, tasks τ_0, τ_1, τ_2, . . ., τ_n are schedulable by RMA.

If task τ_0 acts as the periodic server to service aperiodic tasks, then in each time unit, $1 - L$ amount of time can be reserved to service aperiodic tasks. As L is the minimal processor utilization required to schedule the periodic task set, $1 - L$ is the maximal amount of processor time to be reserved to service aperiodic tasks in each time unit. Thus, $R_{lub} = 1 - L$ is maximal. ■

The following example demonstrates the application of Theorem 10. Given tasks $\tau_1 = (3, 1.2)$, $\tau_2 = (5, 1.5)$, and $\tau_3 = (6, 0.6)$, $L = 0.9$, this task set is schedulable by RMA based on Theorem **ERMA2**. Based on Theorem 10, $1 - L = 1 - 0.9 = 0.1$ is the maximal amount of time in each time unit that can be reserved to service aperiodic tasks. The period of the periodic server is 1. The scheduling result of *RBA_RMA* with reservation size $R_s = 0.1$ is shown in Figure 4.4. Each time unit has only 0.9 amount of processor time to service periodic tasks. At time 0, $J_{1,0}, J_{2,0}$ and $J_{3,0}$ are ready, $J_{1,0}$ has highest priority and is chosen to run followed by $J_{2,0}$. At time 3, $J_{1,1}$ is ready and it has higher priority than $J_{3,0}$. After $J_{1,1}$ finishes at time 4.4, $J_{3,0}$ starts to run and finishes at time 5. As all jobs at the first critical instance can finish within their deadlines, the tasks are schedulable by *RBA_RMA* with $R_s = 0.1$.

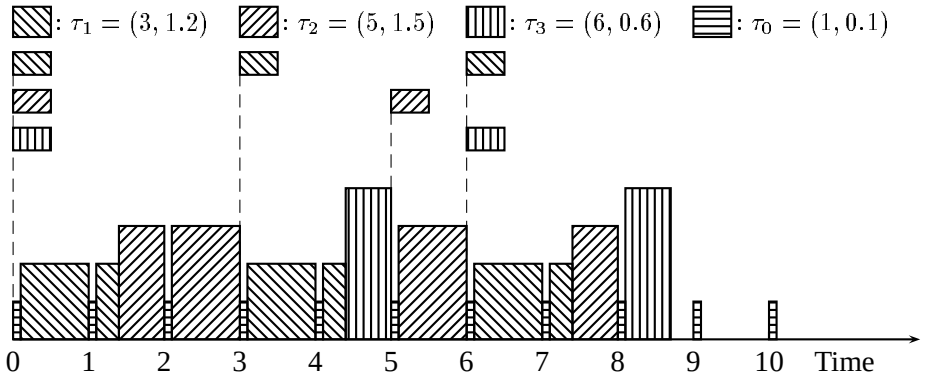


Figure 4.4: Scheduling Result of *RBA_RMA* with Maximal R_{lub}

4.5 RBA with FPPT

In the previous two sections, *RBA_EDF* and *RBA_RMA* are presented. As FPPT generalizes FPP and FPNP, resulting in a larger range of schedulable task sets than both FPP and FPNP, this section examines the combination of FPPT and RBA, called *RBA_FPPT* for convenience. Unlike *RBA_EDF* and *RBA_RMA* where the exact maximal size of the reserved bandwidth can be computed with simple formulae directly, the computation of the maximal size of the reserved bandwidth for *RBA_FPPT* is composed of three steps. First, a schedulability test for FPPT is performed on a given task set. If the task set is unschedulable, then no reservation is available in each time unit. Second, when the task set is schedulable, a trial-and-error algorithm is presented to compute a reservation size when a valid assignment of the original task set is known. Third, starting from the maximal assignment of the original task set, search for a valid assignment under which the maximal reservation size can be calculated. Again, the regular priority of the periodic task set is assumed to be predefined.

4.5.1 Priority Range

Suppose periodic tasks are τ_1, τ_2, ..., τ_n and the periodic server is τ_0, where the period of the periodic server is 1. As the reservation must be guaranteed at each period, the priority range for the periodic server and other periodic tasks are assumed to satisfy the following two conditions.

1. The periodic server has a regular priority higher than any other periodic tasks.

2. Any of the other periodic tasks cannot have a preemption threshold equal to or higher than the regular priority of the periodic server.

The first condition implies that the periodic server cannot get any interference from any other periodic tasks, i.e., it cannot be preempted by any other periodic tasks. In addition, the periodic server also has the highest preemption threshold as the highest preemption threshold is equal to the highest regular priority. The second condition guarantees that the periodic server can preempt the execution of any other periodic tasks.

As the larger a numeric value, the lower the priority it represents, it is reasonable to assign 0 to the regular priority and preemption threshold of the periodic server. The regular priorities and preemption thresholds of periodic tasks are in the range $[1, n]$.

The periodic server τ_0 and the given periodic tasks τ_1, τ_2, . . ., τ_n form a new task set. For convenience, the new task set is called an *extended task set*. Assume a concatenation operator $\|$, which does two things: it merges two sets and keeps the order of the elements. For example, $\{0\} \parallel \{1, 2, 3\} = \{0, 1, 2, 3\}$. As the periodic server has highest regular priority and preemption threshold 0, i.e., $\pi_0 = 0$ and $\gamma_0 = 0$, the regular priority assignment for the extended task set is denoted as $\{\pi_0\} \parallel \pi$, where π is the regular priority assignment for the original task set. Similarly, when an assignment γ for the original task set is given, the corresponding assignment for the extended task set is denoted as $\{\gamma_0\} \parallel \gamma$.

4.5.2 Original Task Set and Extended Task Set

Given a task set with regular priority assignment π, the corresponding extended task set has a regular priority assignment $\{0\} \parallel \pi$ for an arbitrary reservation size. For any valid assignment γ of the original task set, $\{0\} \parallel \gamma$ may not be a valid assignment for the extended task set under the regular priority assignment $\{0\} \parallel \pi$. The reason is the interference from the periodic server can affect all other periodic tasks. Thus, all periodic tasks must be rechecked in the extended task set. Consider the relationship between the valid assignments for the original task set and the valid assignments for the extended task set, which can formally be specified in the following lemma.

Lemma 5

Let Γ_o be the set of valid assignments for the original task set and Γ_e be the set of valid assignments for the extended task set with an arbitrary reservation size for the periodic server. Define $\Gamma = \{\{0\} \parallel \gamma : \forall \gamma \in \Gamma_o\}$. $\Gamma_e \subseteq \Gamma$.

Proof: $\forall \gamma' \in \Gamma_e$, γ' can be denoted as $\{0, \gamma_1, \gamma_2, \ldots, \gamma_n\}$. As γ' is valid for the extended task set, then it must be valid for the extended task set when the reservation size of the periodic server is decreased. The reason is the FPPT schedulability test is robust based on Theorem 3. Thus, if a task set is schedulable by FPPT under some regular priority assignment and preemption threshold assignment, decreasing the computation time of a task cannot affect the schedulability of the task set.

When the reservation size of the periodic server is decreased to 0, the extended task set degrades to the original task set. Thus, the regular priority and preemption threshold for the periodic server can be ignored and $\{\gamma_1, \gamma_2, \ldots, \gamma_n\}$ is valid for the original task set. Note, $\{0, \gamma_1, \gamma_2, \ldots, \gamma_n\} \in \Gamma$ based on the definition of Γ, resulting in $\gamma' \in \Gamma_e$. Therefore, $\Gamma_e \subseteq \Gamma$. ■

Lemma 5 shows that all possible assignments for the extended task set are in Γ and indicates that for any valid assignment $\{0, \gamma_1, \gamma_2, \ldots, \gamma_n\}$ for the extended task set, the assignment $\{\gamma_1, \gamma_2, \ldots, \gamma_n\}$ is valid for the original task set. Given a valid assignment γ in the original task set, $\{0\} \parallel \gamma$ is a candidate valid assignment for the extended task set depending on the reservation size. One exhaustive approach is to check each assignment γ in Γ_o, calculate the maximal reservation size under assignment $\{0\} \parallel \gamma$ in the extended task set, and choose the assignment γ under which the maximal reservation size can be obtained. However, this exhaustive approach based on the original task set is impractical because $|\Gamma_o|$ may be $n!$ in the worst case.

On the other hand, another approach is to perform the computation directly on the extended task set instead of the original task set. As $1 - U$ is the maximal reservation size for any scheduler, one naive approach is to perform a binary search over the range $[0, 1 - U]$ for the maximal reservation size. While this approach is trivial, its practical usage is limited. Such an approach is not discussed further.

4.5.3 Reservation Size under a Valid Assignment

Given a periodic task set with predefined regular priority, a FPPT schedulability test can be performed based on Chapter 3. If the task set is schedulable, there may exist multiple valid assignments. The minimal and maximal assignments can be calculated effectively based on Theorems 5, 7, and 9. As well, additional valid assignments can be calculated. Furthermore, Lemma **Valid** and Lemma 5 provide support to compute the maximal reservation size effectively, i.e, without checking each valid assignment in Γ_o exhaustively. Before further discussion, an algorithm to compute the maximal reservation size in the extended task set is presented when a valid

assignment in the original task set is known.

Suppose a valid assignment for the original task set is γ and the regular priority assignment is π. The corresponding preemption threshold assignment and regular priority assignment for the extended task set are γ' and π', where $\gamma' = \{0\} \parallel \gamma$ and $\pi' = \{0\} \parallel \pi$. Computing a reservation size starts with a small candidate value C_0. The algorithm proceeds in a straightforward manner using trial-and-error and is shown in Figure 4.5. In the extended task set, due to interference from the periodic server, all periodic tasks are affected and their worst case response time must be re-calculated. Whenever all periodic tasks can still meet their deadlines, C_0 is increased by a step size until some task cannot meet its deadline. For example, the step size can be set to 0.01. Then the final result $C_0 - stepsize$ is the reservation size. When some task misses its deadline, denote it as τ_i. Let τ_j be the task contributing the maximal blocking time for τ_i. The algorithm returns $< C_0 - stepsize, \tau_i, \tau_j >$. Note, if more than one task misses its deadline, the algorithm returns the one with the highest regular priority and its corresponding τ_j; τ_j may not exist, e.g., when τ_i is the task with the lowest regular priority or no task exits with preemption threshold equal to or higher than π_i.

```
1     π' ← {0} || π;
2     γ' ← {0} || γ;
3     C0 ← 0;
4     stepsize = 0.01;
5     while (true)
6         C0 ← C0 + stepsize;
7         for (i ← 1 to n)
8             if (WCRT(πi', γi') > Di) then
9                 compute τj such that γj ≤ πi < πj and Cj is maximal;
10                return < C0 − stepsize, τi, τj >;
11            endif
12        endfor
13    endwhile
```

Figure 4.5: Pseudo-code for Computing Reservation Size

The pseudo-code assumes a valid assignment γ from the original task set exists. For convenience, this algorithm is called $ReservationSize(\gamma)$. As the number of iterations of the while-loop is $\frac{1-U}{stepsize}$ in the worst case, the time complexity of this algorithm is $O(\frac{1-U}{stepsize} \cdot n \cdot r)$, which is pseudo-polynomial, where n is the number of tasks and r (see page 55) is the cost of calling function $WCRT(\pi_i', \gamma_i')$.

4.5.4 Computing Maximal Reservation Size with γ_{min} and γ_{Max}

The heuristic for the algorithm to compute the maximal reservation size in the extended task set by checking some valid assignments in the original task set is based on considering the relationship between the reservation sizes and the valid assignments in the original task set.

For the boundary case γ_{min}, a task with a higher regular priority gets minimal blocking time from another task with a lower regular priority, implying that a task with a higher regular priority may potentially get its minimal response time. However, a task with a lower regular priority may potentially get its maximal response time when its preemption threshold is minimal, which indicates that it gets more interference from tasks with higher regular priorities. When a periodic server is appended, a task with lower regular priority is more likely to miss its deadline as it gets more interference from tasks with higher regular priorities. By intuition, under γ_{min}, the reservation size may not be maximal.

For the other boundary case γ_{Max}, a task with a lower regular priority gets minimal interference from the tasks with higher regular priorities while it may get maximal blocking time from another task with lower regular priority. However, if the saved interference from tasks with higher regular priorities is greater than the increased blocking time, then the worst case response time of a task with a lower regular priority may potentially be minimized. At the same time, the worst case response time of a task with a higher regular priority may potentially be maximized. When a periodic server is appended, a task with a higher regular priority is more likely to miss its deadline. By intuition, under γ_{Max}, the reservation size may not be maximal.

For example, given tasks $\tau_1 = (50, 15)$, $\tau_2 = (100, 10)$, $\tau_3 = (150, 15)$, $\tau_4 = (200, 30)$, and $\tau_5 = (250, 45)$, $\gamma_{min} = \{1, 2, 3, 4, 5\}$ and $\gamma_{Max} = \{1, 1, 1, 1, 2\}$, as well $\gamma = \{1, 1, 1, 2, 2\}$ is also valid. Let $\gamma'_{min} = \{0\} \parallel \gamma_{min}$, $\gamma'_{Max} = \{0\} \parallel \gamma_{Max}$, $\gamma' = \{0\} \parallel \gamma$, and $\pi' = \{0\} \parallel \pi$. Based on algorithm $ReservationSize(\gamma)$, the corresponding reservation sizes under assignments γ'_{min}, γ'_{Max}, and γ' are $R'_{min} = 0.07$, $R'_{Max} = 0.1$, and $R' = 0.14$, respectively. This example shows that the maximal reservation may occur between γ_{min} and γ_{Max}.

There may exists multiple valid assignments in the original task set under which the maximal reservation size can be calculated. For the previous example, under assignments $\{1, 2, 3, 3, 4\}$, $\{1, 2, 3, 3, 3\}$, and $\{1, 2, 3, 3, 2\}$, the maximal reservation is also 0.14. Among these valid assignments, the maximal assignment is preferred as it makes the number of preemptions minimal.

As the maximal reservation size may not occur at γ_{min} or γ_{Max}, it seems that there exists

a balance point located between γ_{min} and γ_{Max}, inclusively. The key point in finding such a balance point between γ_{min} and γ_{Max} is to decrease the preemption thresholds of those tasks with lower regular priorities from γ_{Max}. Note, lowering the preemption threshold of a task may have two effects. First, it may potentially get a longer worst case response time for the task itself as it may get more interference from tasks with higher regular priorities. Second, those tasks with regular priorities higher than the task may potentially get a shorter worse case response time as they may get less blocking time from some task with a lower regular priority. Thus, lowering the preemption threshold of a task with a lower regular priority results in the worst case response time of a task with lower regular priority potentially lengthening and the worst case response time of a task with higher regular priority shortening. Consider those tasks in the previous example, their worse case response times under γ_{Max}, γ, γ_{min} are shown in Figure 4.6.

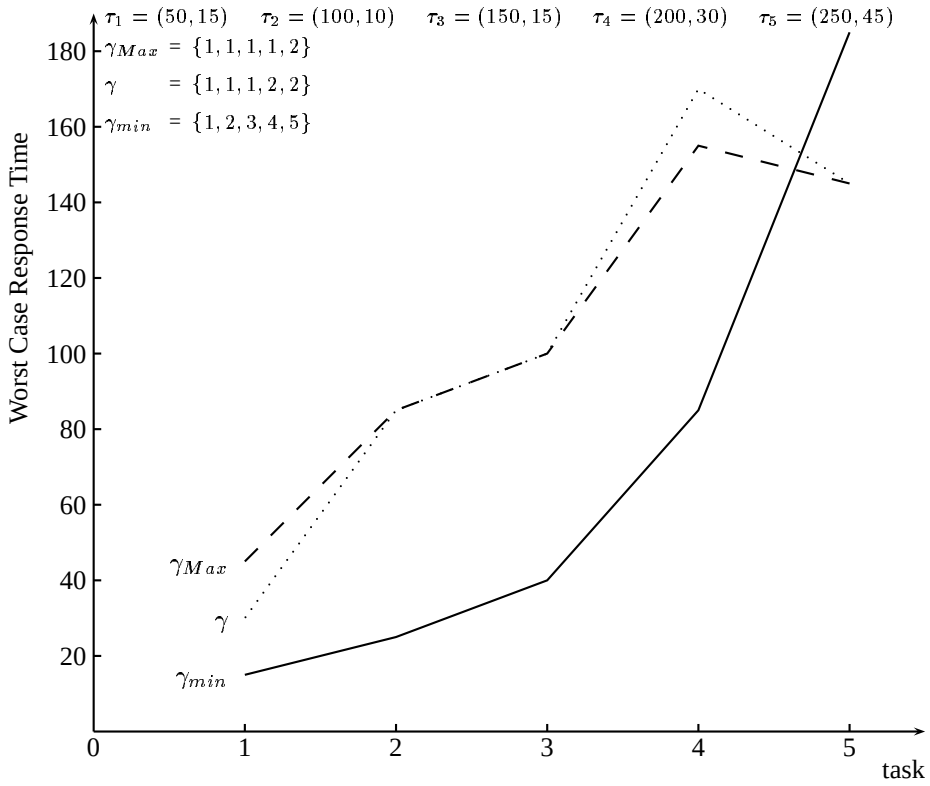


Figure 4.6: Boundary Cases and Balanced Point

In the diagram, the dashed line stands for the worst case response time of all tasks in the task set under assignment γ_{Max}, similarly the dotted line for assignment γ and the solid line for

assignment γ_{min}. The diagram indicates the tasks with higher regular priorities under γ_{min} get shorter worst case response times than γ and γ_{Max}, while the tasks with lower regular priorities under γ_{min} get longer worst case response times than γ and γ_{Max}. Compare γ_{Max} with γ, the latter is obtained by decreasing the preemption threshold of τ_4 in γ_{Max}. The diagram indicates that the worst case response times for τ_1, τ_2, and τ_3 under γ are not worse than under γ_{Max}; the worst case response time of τ_4 and τ_5 under γ are not better than under γ_{Max}.

Computing such a balance point starts from the maximal assignment γ_{Max}. As the preemption threshold of each task under γ_{Max} is already maximized, increasing the preemption threshold of any task is already impossible. Thus, during the computation, it is important to only decrease the preemption threshold of tasks. At the same time, oscillation of preemption threshold of a task up and down is avoided. The computation is based on algorithm $ReservationSize(\gamma)$ and its result $< maxRs, \tau_i, \tau_j >$.

Suppose calling function $ReservationSize(\gamma_{Max})$ returns $< maxRs, \tau_i, \tau_j >$. The result indicates that the maximal reservation size is $maxRs$. If a larger reservation size is used, then τ_i is the task with highest regular priority under γ_{Max} that misses its deadline; and τ_j contributes the maximal blocking time for τ_i. What is the approach to allow τ_i to continue to meet its deadline under a larger reservation size? There are two possible options to allow τ_i to meet its deadline. First, τ_i's preemption threshold could be increased, resulting in less interference from tasks with higher regular priorities. Second, the preemption threshold of some task with a lower regular priority could be decreased, resulting in less blocking time for τ_i. Because the starting assignment is the maximal assignment γ_{Max}, increasing the preemption threshold of any task is impossible. The only option is decreasing the preemption threshold of the task contributing the maximal blocking time. Consider these results for $< maxRs, \tau_i, \tau_j >$.

1. τ_j does not exist. This case is trivial. τ_i cannot be blocked by any task because no τ_j contributes blocking time for τ_i. Hence, the only option to allow it to meet its deadline is to increase its preemption threshold, which is impossible because of γ_{Max}. Thus, no further increment of the preemption threshold for τ_i can be made. Thus, no better result can be obtained and γ_{Max} is the maximal assignment under which the maximal reservation size can be obtained. This property is formally specified in the following lemma.

   Lemma 6

   If calling function $ReservationSize(\gamma_{Max})$ returns $< maxRs, \tau_i, \tau_j >$, where τ_j does not exist, then γ_{Max} is the maximal assignment under which the maximal reservation size can be obtained.

Proof: For any other valid assignment $\gamma = \{\gamma_1, \gamma_2, \ldots, \gamma_n\}$, let Rs be the maximal reservation size under γ. Suppose $Rs > maxRs$. Consider the extended task set under assignments γ and γ_{Max} with reservation size Rs. As γ is valid, based on Theorem 4, $\gamma \preccurlyeq \gamma_{Max}$. Hence, $\gamma_{M_k} \leq \gamma_k$ for $1 \leq k \leq n$. Under both assignments, τ_i cannot be blocked by any task as τ_j does not exist. Furthermore, τ_i cannot get worse interference from tasks $\tau_0, \tau_1, \tau_2, \ldots, \tau_{i-1}$ under γ_{Max} than under γ with reservation size Rs as $\gamma_{M_i} \leq \gamma_i$. Thus, the worst case response time of τ_i under γ_{Max} cannot be worse than under γ with reservation size Rs. This is a contradiction as $Rs > maxRs$ and $maxRs$ is the maximal reservation under γ_{Max}, implying that any reservation larger than $maxRs$ causes τ_i to miss its deadline under γ_{Max}. Thus, the assumption $Rs > maxRs$ is incorrect, resulting in $Rs \leq maxRs$.

Clearly, γ_{Max} is the maximal assignment under which the maximal reservation size can be obtained. ■

2. τ_j exists, but $\gamma_{m_j} \leq \pi_i$, implying that the minimal threshold assignment of task τ_j is not lower than the regular priority of task τ_i. As increasing the preemption threshold of τ_i is impossible because of γ_{Max}, the only option is to decrease the preemption threshold of τ_j so the blocking time for τ_i can be shorter. But this does not work because $\gamma_{m_j} \leq \pi_i$. Hence, τ_j's preemption threshold cannot be lowered sufficiently so it no longer blocks τ_i. Thus, no better result can be obtained and γ_{Max} is the maximal assignment under which the maximal reservation size can be obtained. This property is formally specified in the following lemma.

Lemma 7

If calling function $ReservationSize(\gamma_{Max})$ returns $< maxRs, \tau_i, \tau_j >$, where τ_j exists with $\gamma_{m_j} \leq \pi_i$, then γ_{Max} is the maximal assignment under which the maximal reservation size can be obtained.

Proof: For any other valid assignment $\gamma = \{\gamma_1, \gamma_2, \ldots, \gamma_n\}$, let Rs be the maximal reservation size under γ. Suppose $Rs > maxRs$. Consider the extended task set under assignments γ and γ_{Max} with reservation size Rs.

(a) As γ is valid, based on Theorem 4, $\gamma_{min} \preccurlyeq \gamma \preccurlyeq \gamma_{Max}$, resulting in $\gamma_{M_k} \leq \gamma_k \leq \gamma_{m_k}$ for $1 \leq k \leq n$. Thus, $\gamma_{M_j} \leq \gamma_j \leq \pi_i$ due to $\gamma_{m_j} \leq \pi_i$. Based on Formula 2.16, τ_j can block τ_i under assignments γ_{Max} and γ. As τ_j contributes the maximal blocking time for τ_i under γ_{Max}, it also contributes the maximal blocking time for τ_i under γ

based on Formula 2.16. Hence, under both assignments γ_{Max} and γ, τ_i gets the same maximal blocking time from τ_j.

(b) Now consider the interference for τ_i under γ_{Max} and γ. τ_i cannot get worse interference from tasks τ_0, τ_1, τ_2, ..., τ_{i-1} under γ_{Max} than under γ because τ_i gets its smallest interference from tasks with higher regular priorities under γ_{Max}.

Thus, the worst case response time of τ_i under γ_{Max} cannot be worse than under γ with reservation size Rs. This is a contradiction as $Rs > maxRs$ and $maxRs$ is the maximal reservation under γ_{Max}, implying that any reservation size larger than $maxRs$ causes τ_i to miss its deadline under γ_{Max}. Thus, the assumption $Rs > maxRs$ is incorrect, resulting in $Rs \leq maxRs$.

Clearly, γ_{Max} is the maximal assignment under which the maximal reservation size can be obtained. ■

3. τ_j exists and $\gamma_{m_j} > \pi_i$, implying that the minimal preemption threshold of task τ_j is lower than the regular priority of task τ_i. Let $\gamma = \{\gamma_{M_1}, \gamma_{M_2}, \ldots, \gamma_{M_{j-1}}, \pi_i + 1, \gamma_{M_{j+1}}, \ldots, \gamma_{M_n}\}$. If γ is a valid assignment under which a better reservation size *can be* obtained, then a better choice is found and further checking is required. The following lemma indicates that choosing such a better choice cannot exclude the maximal assignment under which maximal reservation size can be obtained.

Lemma 8

Assume algorithm $ReservationSize(\gamma_{Max})$ *returns* $< maxRs, \tau_i, \tau_j >$, *where* τ_j *exists with* $\gamma_{m_j} > \pi_i$. *Let assignment* $\gamma = \{\gamma_{M_1}, \gamma_{M_2}, \ldots, \gamma_{M_{j-1}}, \pi_i + 1, \gamma_{M_{j+1}}, \ldots, \gamma_{M_n}\}$. *If* γ *is a valid assignment under which a better reservation size can be obtained, then any valid assignment* γ' *under which a better reservation size than* γ *can be obtained, must satisfy* $\gamma' \preccurlyeq \gamma$.

Proof: As γ' is valid, $\gamma' \preccurlyeq \gamma_{Max}$ based on Theorem 4, resulting in $\gamma'_{M_k} \leq \gamma_k$ for $1 \leq k \leq n$. As $\gamma_k = \gamma_{M_k}$ for $1 \leq k \leq n$ but $k \neq j$, hence $\gamma'_k \geq \gamma_k$ for $1 \leq k \leq n$ but $k \neq j$. The remainder of the proof is to show $\gamma'_j \geq \gamma_j = \pi_i + 1$.

Let Rs and Rs' be the maximal reservation sizes under γ and γ', respectively. Then $Rs' > maxRs$ because $Rs' > Rs$ and $Rs > maxRs$. Suppose $\gamma'_j < \gamma_j$, which implies $\gamma'_j < \pi_i + 1$. Consider the worse case response time of τ_i under assignments γ' and γ_{Max} with reservation size Rs'. Under both assignments, τ_i gets the same maximal blocking time

from τ_j because $\gamma_{M_j} \leq \gamma'_j < \pi_i + 1$. As $\gamma_{M_i} \leq \gamma'_i$, τ_i gets not worse interference from tasks with higher regular priorities under γ_{Max} than under γ'. Therefore, under assignment γ_{Max} with reservation size Rs', τ_i can meet its deadline and $Rs' > maxRs$. This is a contradiction as any reservation larger than $maxRs$ causes τ_i to miss its deadline under γ_{Max}. Therefore, $\gamma'_j \geq \gamma_j$, resulting in $\gamma' \preccurlyeq \gamma$. ■

4. τ_j exists and $\gamma_{m_j} > \pi_i$. Let $\gamma = \{\gamma_{M_1}, \gamma_{M_2}, \ldots, \gamma_{M_{j-1}}, \pi_i + 1, \gamma_{M_{j+1}}, \ldots, \gamma_{M_n}\}$. A better reservation size *cannot be* obtained under γ. The reasons why a better reservation size cannot be obtained are as follows.

   (a) γ may be invalid. When γ is invalid, a better reservation size *cannot be* obtained, of course.

   (b) τ_j itself is affected due to a lower preemption threshold in γ than in γ_{Max}, resulting in more interference.

   (c) τ_i is no longer blocked by τ_j, but it may be blocked by another task with the same blocking time as τ_j.

   (d) τ_k would miss its deadline where $i + 1 \leq k \leq j - 1$.

   No matter which case occurs, to get a better reservation size, the preemption thresholds of tasks τ_0, τ_1, τ_2, ..., τ_{i-1} are not required to change based on Lemma **Valid**. In addition, only the preemption thresholds of tasks τ_i, τ_{i+1}, ..., τ_n should be modified. An algorithm called $FindBetter(\gamma, maxRs)$ is used to compute the next assignment under which a better reservation size may be obtained, which is shown in Figure 4.7.

   Algorithm $FindBetter(\gamma, maxRs)$ performs the 4 checks. At beginning, the algorithm sets $\{0\} \parallel \{\gamma_1, \gamma_2, \ldots, \gamma_{i-1}\}$ to γ' and uses a reservation size $maxRs + stepsize$, which is the minimal reservation size better than $maxRs$. The algorithm tries to schedule τ_0, τ_1, τ_2, ..., τ_{i-1}, τ_i with assignment $\gamma' \parallel \{\gamma_{M_i}\}$. If any task τ_h with higher regular priority is unschedulable, then γ'_i is set to $\pi'_h + 1$. After checking all tasks with regular priorities higher than τ_i, if τ_i misses its deadline, then no better reservation size can be made and γ_{Max} is the maximal assignment under which the maximal reservation size can be calculated. The loop is repeated to add the next task until all tasks are checked. If the task set is still schedulable at the end, then the new assignment is returned. Due to the two nested for-loops, the time complexity of the algorithm is $O(n^2 \cdot r)$, where n is the number of tasks and r (see page 55) is the cost of calling function $WCRT(\pi'_h, \gamma'_h, S)$ or $WCRT(\pi'_k, \gamma'_k, S)$.

```
1    stepsize ← 0.01;
2    π' ← {0} ‖ π;
3    γ' ← {0} ‖ {γ_1, γ_2, ..., γ_{i-1}};
4    τ_0 ← (1, Rs + stepsize);
5    S ← {τ_0, τ_1, τ_2, ..., τ_{i-1}};
6    for (k ← i to n) do
7        S ← S ∪ {τ_k};
8        γ' ← γ' ‖ {γ_{M_k}};
9        for (h ← 0 to k − 1)
10           if (WCRT(π'_h, γ'_h, S) > D_h) then γ'_k ← π'_h + 1;
11       endfor
12       if (WCRT(π'_k, γ'_k, S) > D_k) then return < false, γ' >;
13   endfor
14   return < true, γ' >;
```

Figure 4.7: Pseudo-code for FindBetter

Compared to algorithm *FindMaxFromFPNP* in Section 3.6.6 (see Figure 3.9 on page 63), calling function $FindBetter(\gamma, maxRs)$ tries to compute the maximal preemption thresholds for tasks τ_i, τ_{i+1}, ..., τ_n with the constraints that the reservation size is $maxRs + stepsize$ and the preemption thresholds for tasks τ_0, τ_1, τ_2, ..., τ_{i-1} are known. The reason is as follows. When task τ_k is considered for $i \leq k \leq n$, its initial preemption threshold is γ_{M_k}, which is maximal. As a result, task τ_k may block tasks with higher regular priorities. Lines 9 to 11 check each task τ_h with higher regular priority for $0 \leq h \leq k-1$. If τ_h cannot meet its deadline, the reasons is that τ_h is blocked by τ_k as before τ_k is appended, tasks τ_0, τ_1, τ_2, ..., and τ_{k-1} are schedulable. Thus, the preemption threshold of τ_k is adjusted to $\pi_h + 1$. After all tasks with higher regular priorities are checked, if τ_k is schedulable, then tasks τ_0, τ_1, τ_2, ..., and τ_k are schedulable; otherwise, they are unschedulable. Note, during the computation, the preemption threshold of τ_k is never decreased except when it blocks some task that misses its deadline. Thus, the preemption threshold of τ_k is maximal under the constraints.

Assume calling function $ReservationSize(\gamma_{Max})$ returns $< maxRs, \tau_i, \tau_j >$, where τ_j exists with $\gamma_j > \pi_i$. Let $\gamma = \{\gamma_{M_1}, \gamma_{M_2}, ..., \gamma_{M_{j-1}}, \pi_i + 1, \gamma_{M_{j+1}}, ..., \gamma_{M_n}\}$ (γ may be invalid). A better reservation size cannot be obtained under assignment γ.

Lemma 9

If calling function $FindBetter(\gamma, maxRs)$ returns $< false, \gamma' >$, then γ_{Max} is the max-

imal assignment under which the maximal reservation size can be obtained; if calling function $FindBetter(\gamma, maxRs)$ *returns* $< true, \gamma' >$*, and the maximal reservation size under* γ' *is* Rs'*, then for any valid assignment* γ'' *under which the maximal reservation size is* Rs'' *such that* $Rs'' > Rs'$*, it must satisfy* $\gamma'' \preccurlyeq \gamma'$*.*

Proof: Note, if calling function $ReservationSize(\gamma_{Max})$ returns $< maxRs, \tau_i, \tau_j >$, this indicates that tasks τ_0, τ_1, τ_2, ..., τ_{i-1} can meet their deadlines under γ_{Max} with reservation $maxRs + stepsize$, and τ_i is the task with the highest regular priority that misses its deadline under the same conditions.

When function $FindBetter(\gamma, maxRs)$ returns $< false, \gamma' >$, a reservation size of $maxRs + stepsize$ cannot be obtained because any reservation size larger than $maxRs$ causes τ_i to miss its deadline, which is indicated by line 12 in Figure 4.7. As $maxRs + stepsize$ is the minimal value greater than $maxRs$, then $maxRs$ is the maximal reservation size, which can be obtained under γ_{Max}.

When function $FindBetter(\gamma, maxRs)$ returns $< true, \gamma' >$, a reservation size of $maxRs + stepsize$ can be obtained under assignment γ'. As γ'' is valid, then $\gamma'' \preccurlyeq \gamma_{Max}$ based on Theorem 4. Thus, $\gamma''_k \geq \gamma_{M_k}$ for $1 \leq k \leq n$. As $\gamma'_k = \gamma_{M_k}$ for $1 \leq k < i$, then $\gamma''_k \geq \gamma'_k$ for $1 \leq k < i$. The remainder of the proof is to show $\gamma''_k \geq \gamma'_k$ for $i \leq k \leq n$.

Suppose $\gamma''_k < \gamma_k$ for some k such that $i \leq k \leq n$. As γ'_k is the maximal preemption threshold of task τ_k for $i + 1 \leq k \leq n$ with reservation size Rs', increasing the preemption threshold of τ_k can cause some task with a higher regular priority to miss its deadline, resulting in that the corresponding reservation size cannot be better than Rs'. Hence, $Rs'' \leq Rs'$, which is a contradiction as $Rs'' > Rs'$. Therefore, no k exists such that $\gamma''_k < \gamma_k$ for $i \leq k \leq n$.

Based on the previous two steps, $\gamma'' \preccurlyeq \gamma'$. ■

Lemma 9 guarantees that if there is a better choice, it cannot be missed. At the same time, the maximal assignment under which the maximal reservation size can be calculated is not excluded when such a better choice is made.

Lemmas 6, 7, 8, and 9 indicate when the maximal reservation size can be obtained under the maximal assignment γ_{Max} or indicate when further computation is required.

Now consider the case when further computation is required. When $ReservationSize(\gamma_{Max})$ returns $< maxRs, \tau_i, \tau_j >$, based on Lemma 8 and 9, There are two cases when further com-

putation is required. First, τ_i is no longer blocked by τ_j and a better reservation size can be obtained. Under this case, assignment $\gamma = \{\gamma_{M_1}, \gamma_{M_2}, \ldots, \gamma_{M_{j-1}}, \pi_i + 1, \gamma_{M_{j+1}}, \ldots, \gamma_{M_n}\}$ is better than γ_{Max}. Second, τ_i is no longer blocked by τ_j but a better reservation size cannot be obtained. Under this case, the results depends on the result $< flag, \gamma'' >$ from calling function $FindBetter(\gamma)$. $FindBetter(\gamma)$ tries to compute the maximal preemption thresholds for τ_i, $\tau_{i+1}, \ldots, \tau_n$ with the constraints that the preemption thresholds for $\tau_1, \tau_2, \ldots, \tau_{i-1}$ are $\gamma_{M_1}, \gamma_{M_2}, \ldots, \gamma_{M_{i-1}}$ and the reservation size is $maxRs + stepsize$. If such an assignment is unavailable, which is indicated by $flag = false$, then γ_{Max} is the maximal assignment under which the maximal reservation size can be obtained. Otherwise, γ'' is better than γ_{Max}.

To perform further computation, the same computation on γ_{Max} is performed on γ or γ' obtained by Lemmas 8 and 9. The pseudo-code is shown in Figure 4.8. For convenience, this algorithm is called $MaxReservationSize(\gamma)$ and parameter γ is a valid assignment for the original task set. Initially, γ is set to γ_{Max}.

```
1     γ ← γ_Max;
2     < maxRs, τ_i, τ_j > ← ReservationSize(γ);
3     maxRs' ← maxRs;
4     while (true)
5         if (τ_j does not exist) then return < maxRs, γ >;
6         if (γ_{m_j} ≤ π_i) then return < maxRs, γ >;
7         γ' ← {γ_1, γ_2, ..., γ_{j-1}, π_i + 1, γ_{j+1}, ..., γ_n};
8         if (γ' is valid) then < maxRs', τ_i', τ_j' > ← ReservationSize(γ');
9         if (maxRs' > maxRs) then
10            γ ← γ';
11            maxRs ← maxRs';
12            τ_i ← τ_i';
13            τ_j ← τ_j';
14        endif;
15        if (maxRs' ≤ maxRs) then
16            < flag, γ'' > ← FindBetter(γ', maxRs);
17            if (flag = false) return < maxRs, γ >;
18            γ ← γ'';
19            < maxRs, τ_i, τ_j > ← ReservationSize(γ'');
20        endif
21    endwhile
```

Figure 4.8: Pseudo-code for Algorithm Computing Maximal Reservation Size

The correctness of the algorithm is as follows. First, the correctness of the first iteration is guaranteed by Lemmas 6, 7, 8, and 9. From the second iteration, the following lemma guarantees that when the preemption threshold is decreased once, it can no longer be increased during the computation.

Lemma 10
Based on algorithm $MaxReservationSize(\gamma)$, increasing the preemption threshold of any task at any step can never result in a larger reservation size.

Proof: The proof is by induction on each iteration (step) of the while-loop.

1. Base case: k = 1, any increment is impossible as all the preemption thresholds are maximal.
2. Hypothesis: Assume that for $steps \leq k$, only decreasing the preemption threshold results in a larger reservation size.
3. Induction: Consider step $k + 1$. Suppose τ_i is the task with highest regular priority that misses its deadline and τ_j is the task contributing the maximal blocking time for τ_i. There are two options to get a larger reservation size. First, increase the preemption threshold of τ_i. Second, decrease the preemption threshold of τ_j. This proof only shows that increasing the preemption threshold of τ_i is impossible. There are two possible situations when increasing the preemption threshold of τ_i.
   - $\gamma_i = \gamma_{M_i}$, implying that increasing γ_i is impossible.
   - $\gamma_i \neq \gamma_{M_i}$. As $\gamma_i = \gamma_{M_i}$ at the beginning of the algorithm, consider the last time γ_i was decreased. Say it was done because it was blocking task τ_p. Based on the hypothesis, no task's preemption threshold was increased. Thus, τ_p must be of the same or lower priority as when it was being blocked by τ_i and it is schedulable, resulting in τ_p's interference from tasks with higher regular priorities being the same or larger. If γ_i is increased, it contributes the same blocking time to τ_p as before it was decreased, but the reservation size is larger. Clearly, τ_p is unschedulable under the larger reservation size, as its worst case response time is the same or larger but it was the task limiting the reservation size previously. Hence, τ_i's preemption threshold cannot be increased.

   Thus, it is impossible to increase the preemption threshold of τ_i.

Based on induction, Lemma 10 is correct. ■

Lemma 10 indicates that increasing the preemption threshold of a task during the computation is impossible. That is why no preemption threshold increment operation exists in $FindBetter(\gamma)$ and $MaxReservationSize(\gamma)$. Based on Lemma 10, after an iteration, the original assignment γ and the new assignment γ' satisfy $\gamma' \preccurlyeq \gamma$ and a better reservation size can be obtained under γ' than under γ. In other words, the maximal assignment under which the maximal reservation size can be obtained is never excluded and the searching direction is correct. When the algorithm stops, it indicates that no more searching space and no larger reservation size is available.

Suppose when the algorithm stops, the result is $< maxRs, \tau_i, \tau_j >$. Then there are two possibilities.

1. τ_j does not exist, which indicates that no adjustment can be made for τ_i as no job contributes blocking time for τ_i and increasing its preemption threshold is impossible based on Lemma 10.

2. τ_j exists. As $< maxRs, \tau_i, \tau_j >$ is returned, $maxRs + stepsize$ is valid for tasks τ_1, τ_2, ..., τ_{i-1} under γ_1, γ_2, ..., γ_{i-1}. Further changing any of γ_1, γ_2, ..., γ_{i-1} cannot affect the schedulability of τ_i. Assume a larger reservation size is possible. Hence, some adjustment of γ_{i+1}, γ_{i+2}, ..., γ_n must result in $maxRs + stepsize$ being a valid reservation size. As the algorithm to adjust γ_{i+1}, γ_{i+2}, ..., γ_n is based on the algorithm to compute the maximal assignment from FPNP, it is guaranteed to find such an assignment if it exists. As the algorithm fails to find such an assignment, no such assignment exists and the reservation size $maxRs + stepsize$ is invalid. Therefore, $maxRs$ is the maximal reservation size.

The worst case for algorithm $MaxReservationSize(\gamma)$ is each iteration only increases the reservation size with $stepsize$. Note, the upper bound for the maximal reservation size is $1 - U$. As the time complexity for $ReservationSize(\gamma)$ is $O(\frac{1-U}{stepsize} \cdot n \cdot r)$ and the time complexity for $FindBetter(\gamma)$ is $O(n^2 \cdot r)$, then the time complexity for algorithm $MaxReservationSize(\gamma)$ is $O(\frac{1-U}{stepsize} \cdot n \cdot r + n^2 \cdot r)$, assuming that $ReservationSize(\gamma)$ starts with the current maximal reservation size when it is called at each iteration.

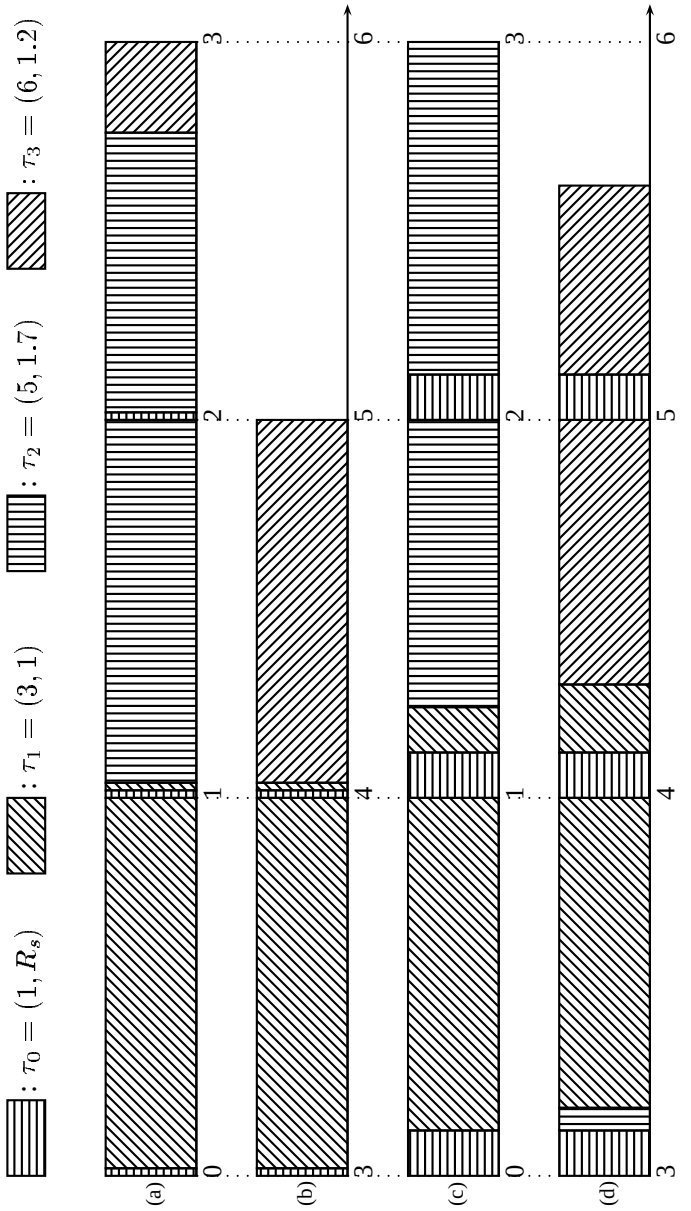
4.6 Comparing *RBA_EDF*, *RBA_RMA*, and *RBA_FPPT*

The section presents a simple example to demonstrate the difference among *RBA_EDF*, *RBA_RMA*, and *RBA_FPPT*. Given tasks $\tau_1 = (3, 1)$, $\tau_2 = (5, 1.7)$, and $\tau_3 = (6, 1.2)$, then $U = 0.873333$,

$L = 0.98$. The bandwidth reserved starts at the beginning of each time unit for convenience and simplification. If *RBA_RMA* is used, then $R_s = 1 - L = 1 - 0.98 = 0.02$. If *RBA_EDF* is used, breaking ties in favor of the task with shorter period, then $R_s = 1 - U = 1 - 0.88 = 0.12$. The scheduling result by *RBA_RMA* is shown by parts (a) and (b) in Figure 4.9, while the corresponding result by *RBA_EDF* is shown by parts (c) and (d) in Figure 4.9. Only the scheduling results of the first critical instance are shown in Figure 4.9 because of the critical zone analysis [45] for RMA and EDF. Thus, the schedulability test stops when those jobs in the first critical instant meet their deadlines.

In part (a) and (b), *RBA_RMA* is used. At the beginning of each time unit, $R_s = 0.02$ is reserved. Thus, only 0.98 is available for periodic tasks. At time 0, $J_{1,0}$, $J_{2,0}$ and $J_{3,0}$ are ready. $J_{1,0}$ has highest priority and runs at time 0.02. $J_{1,0}$ is preempted by the reservation at time 1 and resumes running at time 1.02 and finishes at time 1.04, after which $J_{2,0}$ starts to run. $J_{2,0}$ continues to run until it is preempted by the reservation at time 2, after which it resumes running at time 2.02 and finishes at time 2.76, after which $J_{3,0}$ starts to run. After $J_{3,0}$ is preempted by the reservation at time 3, $J_{1,1}$ is ready. After the reservation, $J_{1,1}$ starts to run at time 3.02 as τ_1 has higher priority than τ_3. After $J_{1,1}$ is preempted by the reservation at time 4, it resumes running at time 4.02 and finishes at time 4.04, after which $J_{3,0}$ resumes running and finishes at time 5. Up to this point, all jobs in the first critical instant have meet their deadlines, implying that they are schedulable by *RBA_RMA* with $R_s = 0.02$.

In part (c) and (d), *RBA_EDF* is used. At the beginning of each time unit, $R_s = 0.12$ is reserved. Thus, only 0.88 is available for periodic tasks. At time 0, $J_{1,0}$, $J_{2,0}$ and $J_{3,0}$ are ready. $J_{1,0}$ has highest priority and runs at time 0.12. $J_{1,0}$ is preempted by the reservation at time 1 and resumes running at time 1.12 and finishes at time 1.24, after which $J_{2,0}$ starts to run. $J_{2,0}$ continues to run until it is preempted by the reservation at time 2, after which it resumes running at time 2.12 until it is preempted by the reservation at time 3. At time 3, $J_{1,1}$ is ready with deadline at time 6. After the reservation, $J_{2,0}$ resumes running at time 3.12 and finishes at time 3.18 because its deadline is 5, which is earlier than that of $J_{1,1}$. $J_{1,1}$ starts to run at time 3.18 and is preempted by the reservation at time 4. After the reservation, $J_{1,1}$ resumes running at time 4.12 and finishes at time 4.3, after which $J_{3,0}$ starts to run. $J_{3,0}$ is preempted by the reservation at time 5 and resumes running at time 5.12 and finishes at time 5.62. Though $J_{2,1}$ is ready at time 5 with deadline 10, $J_{3,0}$ resumes running as its deadline 6 is earlier than 10. Up to this point, all jobs in the first critical instant have meet their deadlines, implying that they are schedulable by *RBA_EDF* with $R_s = 0.12$.



(a) and (b): *RBA_RMA* $R_s = 0.02$ (c) and (d): *RBA_EDF* $R_s = 0.12$

Figure 4.9: Sample Scheduling Results

If *RBA_FPPT* is used, $R_s = 0.1$ and the scheduling stops when all jobs in the first level-i busy period meet their deadlines. The scheduling for *RBA_FPPT* with $R_s = 0.1$ is shown by Figure 4.10 and 4.11, where $\pi' = \{0, 1, 2, 3\}$ and $\gamma' = \{0, 1, 1, 1\}$ for the extended task set. Figure 4.10 shows the scheduling results for the busy periods of level-0, level-1, and level-2. Figure 4.11 shows the scheduling result for level-3 busy period. Note, the processor utilization under FPPT is $0.873333 + 0.1 = 0.973333$, which is less than 1.0, implying that *RBA_FPPT* cannot be better than *RBA_EDF*.

When *RBA_FPPT* is used, a task competes for the processor with its preemption threshold after it is preempted by any other task such that the regular priority of the latter is higher than the preemption threshold of the former. The ties are broken in favor of the preempted task. At the beginning of each time unit, $R_s = 0.10$ is reserved. Thus, only 0.9 is available for periodic tasks. The level-0 busy period is trivial in Figure 4.10. In the level-1 busy period, τ_1 is blocked by τ_2. $J_{2,0}$ just starts to run when the reservation and $J_{1,0}$ are ready at time 0. As the regular priority of the periodic server is higher than the preemption threshold of τ_2, $J_{2,0}$ is preempted by the reservation. After the reservation, as the preemption threshold of τ_2 is equal to the regular priority of τ_1, $J_{2,0}$ resumes running at time 1.1 and finishes at time 1.9 because of breaking ties in favor of the preempted task. $J_{1,0}$ starts to run at time 1.9 and is preempted by the reservation at time 2. $J_{1,0}$ resumes running at time 2.1 and finishes at time 3, when $J_{1,1}$ is ready. After the reservation, $J_{1,1}$ starts to run at time 3.1 and is preempted by the reservation at time 4. $J_{1,1}$ resumes running at time 4.1 and finishes at time 4.2. Up to this point, all jobs in the first level-1 busy period have meet their deadlines, implying that τ_1 is schedulable by *RBA_FPPT* with $R_s = 0.1$.

In the level-2 busy period, τ_2 is blocked by τ_3. $J_{3,0}$ just starts to run when the reservation, $J_{1,0}$, and $J_{2,0}$ are ready at time 0. As the regular priority of the periodic server is higher than the preemption threshold of τ_2, $J_{3,0}$ is preempted by the reservation. After the reservation, as the preemption threshold of τ_3 is equal to the regular priority of τ_1, $J_{3,0}$ resumes running at time 1.1 and finishes at time 1.4 because of breaking ties in favor of the preempted task. $J_{1,0}$ starts to run at time 1.4 and is preempted by the reservation at time 2. $J_{1,0}$ resumes running at time 2.1 and finishes at time 2.5, after which $J_{2,0}$ starts to run until it is preempted at time 3, when $J_{1,1}$ is ready. After the reservation, $J_{2,0}$ resumes running at time 3.1 until it is preempted by the reservation again at time 4. $J_{2,0}$ resumes running at time 4.1 and finishes at time 4.4, after which $J_{1,1}$ starts to run until it is preempted by the reservation at time 5, when $J_{2,1}$ is ready. $J_{1,1}$ resumes running at time 5.1 and finishes at time 5.5, after which $J_{2,1}$ starts to run until it is preempted by the reservation at time 6. $J_{1,2}$ is ready at time 6. After the reservation, $J_{2,1}$ resumes

Figure 4.10: Combination of FPPT and RBA (continued)

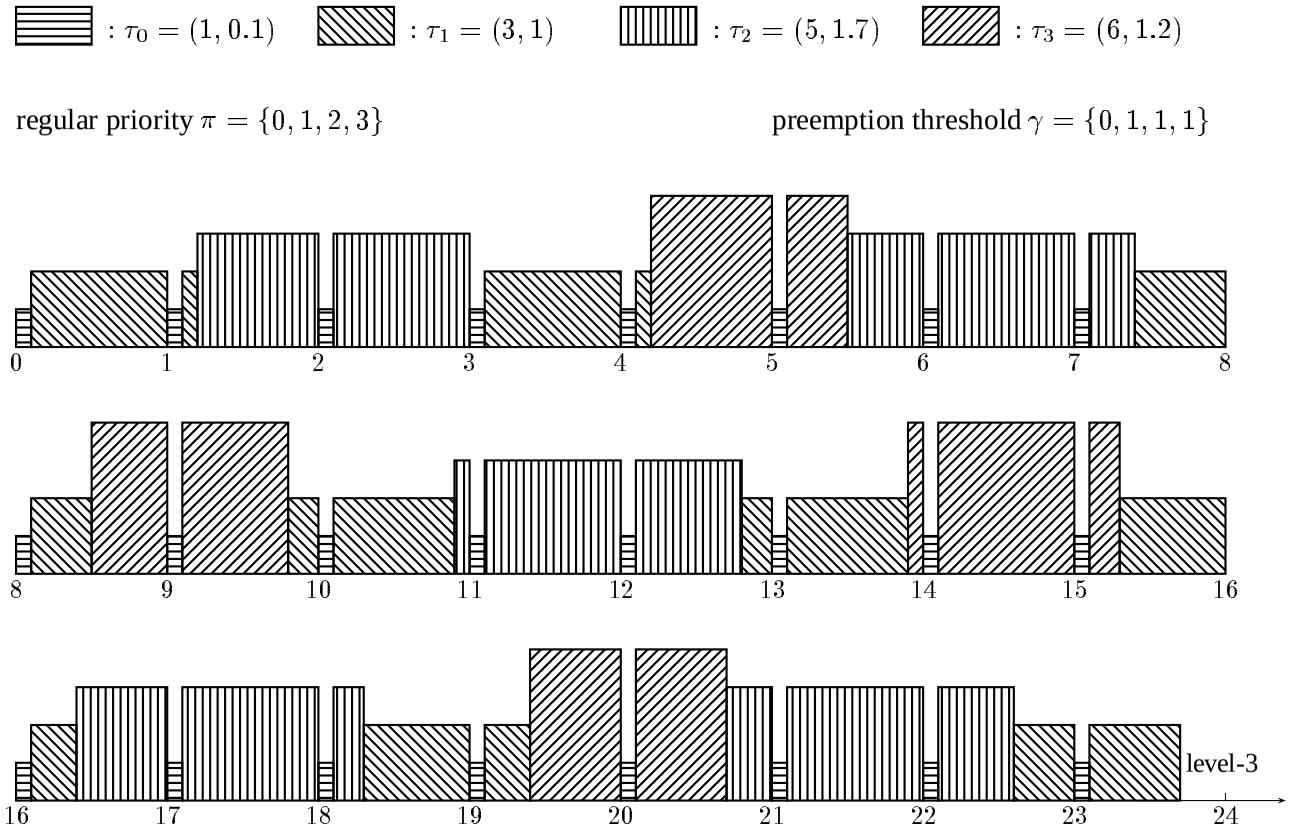


Figure 4.11: Combination of FPPT and RBA

running at time 6.1 until it is preempted by the reservation at time 7. $J_{2,1}$ resumes running at time 7.1 and finishes at time 7.4, after which $J_{1,2}$ starts to run until it is preempted by the reservation at time 8. $J_{1,2}$ resumes running at time 8.1 and finishes at time 8.5. Up to this point, all jobs in the first level-2 busy period have meet their deadlines, implying that τ_2 is schedulable by *RBA_FPPT* with $R_s = 0.1$.

In the level-3 busy period, no task can block τ_3. The scheduling result is separated into 3 segments. Similarly to level-1 and level-2 busy period, it can be verified that all jobs in the first level-3 busy period can meet their deadlines. Thus, τ_3 can be schedulable by *RBA_FPPT* with $R_s = 0.1$. Therefore, the whole task set is schedulable by *RBA_FPPT* with $R_s = 0.1$.

The scheduling results show that not only can FPPT schedule a bigger range of task sets than either FPP or FPNP, it can also provide better response time for aperiodic tasks.

4.7 Further Improvement of RBA

Note, the maximal reservation sizes for *RBA_EDF* and *RBA_RMA* are $1 - U$ and $1 - L$, respectively. As $L \geq U$ based on Lemma 4, the maximal value of *RBA_EDF* is not less than that of *RBA_RMA*. Furthermore, there is no more unused time after reserving a fraction $1 - U$ of processor time in each time unit for *RBA_EDF*. However, since $1 - L \leq 1 - U$, *RBA_RMA* may result in some unused time in certain time units after reserving a fraction $1 - L$ of processor time in each time unit. Such an unused time segment is shown in unit 8 in Figure 4.4. This unused time may not be distributed evenly in each time unit but it is known in advance (statically) and can also be used to run aperiodic tasks in conjunction with the reservation bandwidth. Taking advantage of this extra unused time increases the potential for aperiodic tasks to start and complete their executions earlier. Thus, their response times of aperiodic tasks are shortened..

In one major cycle T_{mc}, the total amount of unused time after reserving $1 - L$ amount of time in each time unit is denoted as t_{unused} and the percentage of extra time to service aperiodic tasks in one major cycle is denoted as p_{ext}, then

$$t_{unused} = T_{mc} - T_{mc} \cdot U - T_{mc} \cdot (1 - L) = T_{mc} \cdot (L - U) \tag{4.1}$$

$$p_{ext} = \frac{t_{unused}}{T_{mc}} = \frac{T_{mc} \cdot (L - U)}{T_{mc}} = (L - U) \tag{4.2}$$

Alternatively speaking, there still exists an extra $(L - U)$ percent of time that can be used to service aperiodic tasks, if required, in one major cycle. For tasks $\tau_1 = (3, 1)$, $\tau_2 = (5, 1.7)$, and

$\tau_3 = (6, 1.2)$, $U = 0.87333$ and $L = 0.98$. Based on Formula 4.1, $t_{unused} = T_{mc} \cdot (L - U) = 30 \cdot (0.98 - 0.87333) \approx 30 \cdot 0.1 = 3$. Based on Formula 4.2, $p_{ext} = (L - U) = (0.9 - 0.873333) \approx 0.10$.

As the reservation size under *RBA_FPPT* cannot be greater than *RBA_EDF*, there may exist extra time unused by the periodic server and periodic tasks. Like *RBA_RMA*, the unused time in each time unit is $1 - U - R_s$, which is known in advance as R_s can be calculated offline. When this value is greater than 0, it indicates that extra time is available to service aperiodic tasks if required.

Finally, there is one remaining source of unused time to service aperiodic tasks: when periodic tasks or aperiodic tasks finish earlier. The unused computation time can be managed by an aperiodic or a background server to execute aperiodic tasks to further improve performance.

4.8 Scaling the Periodic Server

The periodic server with maximal bandwidth in *RBA_EDF* is the periodic task $\tau = (1, 1 - U)$. As scaling the period and computation time of a task with the same scaler does not change the processor utilization of the task, then for any positive integer T, $(T, T(1-U))$ and $(1, 1-U)$ have the same processor utilization. Given a task set, if the periodic server returned by *RBA_EDF* is $(1, 1-U)$, then $(T, T(1-U))$ is also a candidate for the periodic server in *RBA_EDF* as replacing $(1, 1-U)$ with $(T, T(1-U))$ does not change the processor utilization of a task set and EDF only depends on the processor utilization. Therefore, there is more flexibility in choosing a periodic server with EDF.

The periodic server $(1, 1 - U)$ can be generalized to $(T, T(1 - U))$ by scaling the period and computation time with the same scaler when *RBA_EDF* is used. However, when *RBA_RMA* is used, this may not be true. For example, consider three periodic tasks $\tau_1 = (3, 1.2)$, $\tau_2 = (5, 1.5)$, $\tau_3 = (6, 0.6)$, with $U = 1.2/3 + 1.5/5 + 0.6/6 = 0.8$, $L = 0.9$. Based on Theorem 10, the periodic server candidate is $(1, 0.1)$. If it is scaled to $\tau_0 = (2, 0.2)$, then tasks τ_0, τ_1, τ_2, and τ_3 are unschedulable by *RBA_RMA*. This is because scaling a task with the same scaler against period and computation time requires RMA to recalculate the schedulability test based on [41], and the new task set may no longer be schedulable by RMA. On the other hand, consider two periodic tasks $\tau_1 = (6, 2)$ and $\tau_2 = (12, 2)$. Based on Theorem 10, the periodic server candidate is $(1, 0.5)$. If it is scaled to $\tau_0 = (4, 2)$, then tasks τ_0, τ_1, and τ_2 are still schedulable by RMA. In other words, the scaled periodic server is still valid.

Like RMA, scaling may be invalid under *RBA_FPPT*. For example, given tasks $\tau_1 = (3, 1)$, $\tau_2 = (5, 1.7)$, and $\tau_3 = (6, 1.2)$, They are schedulable by *RBA_FPPT* with $R_s = 0.1$, where $\pi' = \{0, 1, 2, 3\}$ and $\gamma' = \{0, 1, 1, 1\}$ for the extended task set. However, if the periodic server $(1, 0.1)$ is scaled to $(2, 0.2)$, then these tasks are unschedulable by *RBA_FPPT* under the same regular priority assignment and preemption threshold assignment.

The purpose of scaling is to provide flexibility for the real-time system designer. In particular, the periodic server has always had period 1, which may be too small and the number of context switches may be too large, resulting in extra processor time, memory, and other system resources. If the period and the computation time of the periodic server can be scaled by the same scaler, the result is a periodic server with a decreased number of context switches. Of course, the scaling cannot be arbitrary. When the period of the periodic server is too large, the periodic server may degrade to a background server, which may worsen the response time of aperiodic tasks. The real-time system designer has the option to choose a reasonable scaler based on both periodic and aperiodic task sets.

4.9 Summary

This chapter points out the maximal reservation size for *RBA_EDF* for an independent task set. For *RBA_RMA*, a trial-and-error algorithm is presented to calculate R_{lub} and Theorem 10 proves how to calculate the maximal reservation size R_{lub} when RMA is used to schedule periodic tasks, which cannot be obtained all the time in [63]. Using *RBA_RMA* results in that all resource management protocols based on RMA can be used. For *RBA_FPPT*, an algorithm is presented to calculate a reservation size when a valid FPPT assignment for a periodic task set is available. When a periodic task set is schedulable by FPPT, γ_{min} and γ_{Max} can be calculated effectively. Starting from them, an algorithm to compute a reservation size is presented. The maximal reservation size calculated by the latter cannot be less than that of the former for the same periodic task set. Though *RBA_FPPT* cannot beat *RBA_EDF* with respect to the reservation size, it does remove the expensive dynamic scheduling cost of EDF. Furthermore, when *RBA_RMA* and *RBA_FPPT* are used, after reserving the maximal amount of time in each time unit, there may exist some unused time in some time units that can be used directly by aperiodic tasks so as to shorten their response times. Finally, periodic tasks and aperiodic tasks may finish earlier, this unused time can be managed by an aperiodic or a background server to further shorten the response time of aperiodic tasks.

Chapter 5

Scheduling Tool-Kit

During the research work of this thesis, it was often necessary to verify some ideas or the validity of an assumption. For example, when a task set with predefined regular priority is schedulable by FPPT, there may exist multiple valid assignments. It turns out to be complex to verify the validity of all possible assignments by hand even with only 5 tasks. A more strict assumption is to assume the task set is harmonic. Even under this strict assumption, the search space is still 5! = 120 in the worst case, which is beyond verifying manually. Thus, a program was required to help perform these computations. This program was extended step by step over the course of the thesis work to become a small real-time scheduling tool-kit, which can verify not only the research work of this thesis but also other previous and future real-time work. Currently, this tool-kit assumes tasks are independent, so resource management protocols are not considered. However, the architecture of the tool-kit does not prohibit supporting resource management protocols.

The motivations to develop this scheduling tool-kit are as follows. First, available tool-kits may not support new scheduling algorithms, which is the thrust of this work. Even if they provided support for new scheduling algorithms, it is often complex to customize the tool-kit, especially to the specific requirements needed in this thesis. Second, available tool-kits focus on some special features of schedulers, which are often not of interest for this thesis. Third, available tool-kits do not provide the necessary output procedure to generate a Gantt-chart step by step even if Gantt-charts are supported. In the research work of this thesis, this procedure is important to trace how a task misses its deadline when it is unschedulable. Based on these reasons, a simple and specific scheduling tool-kit was developed after several separate simulation programs were written.

5.1 Tool-Kit Architecture

The general functional model of the scheduling tool-kit is illustrated in Figure 5.1. The tool-kit has two input parameters: schedulers and task sets. Based on the input parameters, the tool-kit outputs an answer of "yes" or "no" for a schedulability test. If the answer is "yes", the scheduler provides one schedule and system-status information such as the utilization of the system resources. If the answer is "no", the tool-kit should point out which task misses its deadline. This procedure can be repeated as long as the user wants to change the input parameters. Based on this model, the performance of different schedulers and the impact of the task set can be analyzed. Each of these two input parameters, the output information, and the tool-kit are discussed in detail.

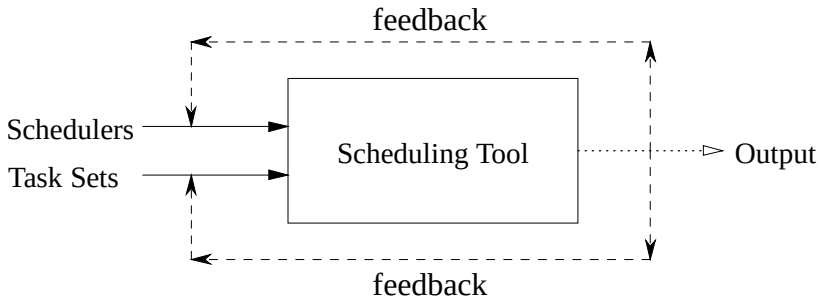


Figure 5.1: Architecture of Scheduling Tool-Kit

5.1.1 Schedulers

Though many real-time schedulers are available [45, 43, 7, 41, 39, 26, 74], there is no general scheduler that works for all task sets. Hence, it is necessary to provide different schedulers in a tool-kit and let the tool-kit and/or user choose the most appropriate schedulers for an application. As multiprocessor architecture and networking are becoming ubiquitous, it is reasonable for a computationally expensive schedulability test algorithm to take advantage of the multiprocessor architecture to improve scheduling performance. For example, based on Section 3.6, when FPPT performs a schedulability test on a given task set, it searches for a valid assignment to make the task set schedulable. It is shown that the minimal assignment is one valid assignment and it can be computed effectively by starting from its two boundary cases – FPP and FPNP, with the same worst case time complexity. However, it is unknown from which boundary case it is faster to compute an assignment for a specific task set. In a uniprocessor environment, two options are to

choose one of them at random or use experience gained from the results of simulations. However, in a multiprocessor environment, the scheduler can create two threads that run on different processors to compute the minimal assignment by starting from FPP and FPNP independently. Whenever either of them finishes, the other can stop and the former returns its result. Thus, the scheduler finishes with the time cost of the faster one.

5.1.2 Task Sets

The tool-kit provides functionality to create both periodic and aperiodic task sets. In the tool-kit, a task set is considered to be fixed and changing a task set during a simulation is not considered.

There are some general assumptions for a periodic task set in a real-time system. Programmers usually know or can determine the following four parameters for each task: period, worst case computation time, deadline, and phase. If the worst case computation time and/or deadline are unavailable, the distributions of the worst case computation time and/or the deadline can be considered. In general, real-time system designers know the functionality of the system, and can compute these parameters and other constraints based on analysis and/or simulation.

This thesis considers aperiodic tasks but not sporadic tasks. Most research work attempts to transfer aperiodic tasks into periodic tasks with worst case estimation for period and computation time, which is not adopted in this thesis. Instead, this thesis is only interested in how to shorten the response time of an aperiodic task. Whenever an aperiodic task is ready, it is assumed its worst case computation time is also known.

5.1.3 Output Information

The output information from the tool-kit should be concise and helpful:

1. The tool-kit must provide a “yes” or “no” answer for the schedulability test based on a task set and a specific scheduler.

2. If the answer to the schedulability test is “yes”, the tool-kit must provide at least one schedule.

3. If the answer to the schedulability test is “no” and the schedulability test is only sufficient, a simple message states the failure of the schedulability test but this does not indicate the

task set is unschedulable; if the schedulability test is sufficient and necessary, the tool-kit should point out which task misses its deadline.

4. A graphical output should be provided such as a colorful Gantt-chart to indicate the simulation results so a user can easily trace a scheduling procedure step by step.

The importance of visualizing the scheduling procedure step by step must be emphasized. If a schedulability test is successful, examining the scheduling steps can identify pessimistic assumptions in the task set or the scheduler. A user can then make fine grain adjustments to achieve higher processor utilization, perform more work, or get better performance for aperiodic tasks. If the schedulability test is unsuccessful, examining the scheduling steps can identify not only the exact location of the failure but also the job leading up to the failure. A user can then look back in time to examine multiple details that ultimately resulted in the unschedulability.

5.2 Tool-Kit Implementation

Several aspects of the tool-kit created to aid in the development of this thesis are presented. The tool-kit is written in C++ and has been designed in an extensible way.

5.2.1 Scheduler

Currently, the tool-kit supports the following real-time scheduling algorithms: RMA, DMA, EDF, FPNP, FPP, and FPPT. As the number of schedulers increases, a mechanism to manage them effectively is required so it is unnecessary to start from scratch when generating a new scheduler, especially as all real-time schedulers share common features. First, all schedulers need a task set on which to perform a schedulability test. Second, they perform a schedulability test on a given task set. Third, they need to report the results of the schedulability test. All of these common features can be represented in an abstract class. Abstract functionality can be represented by pure virtual functions in C++ and each specific scheduler implements the virtual function differently. This structure is shown in Figure 5.2.

In the interface AbstractPeriodicScheduler, there are five pure virtual functions. Function test verifies if the task set is a schedulable, schedulingInformation displays the scheduling result, taskVerify verifies the validity of a given task set, schedulelist returns a schedule if

```
1  #ifndef RT_ABSTRACT_SCHEDULER_H
2  #define RT_ABSTRACT_SCHEDULER_H
3  #include <list>
4  #include <string>
5  #include "common.h" // for Schedule
6  class AbstractPeriodicScheduler {
7    public:
8      AbstractPeriodicScheduler() {};
9      virtual ~AbstractPeriodicScheduler() {};
10     virtual bool test() = 0; // schedulability test
11     virtual void schedulingInformation( std::string sname ) = 0; // show result
12     virtual bool tasksetVerify() = 0;  // verify the validity of a task set
13     virtual std::list<Schedule> schedulelist() = 0; // get one schedule if existing
14 };
15 #endif
```

Figure 5.2: Abstract Periodic Scheduler

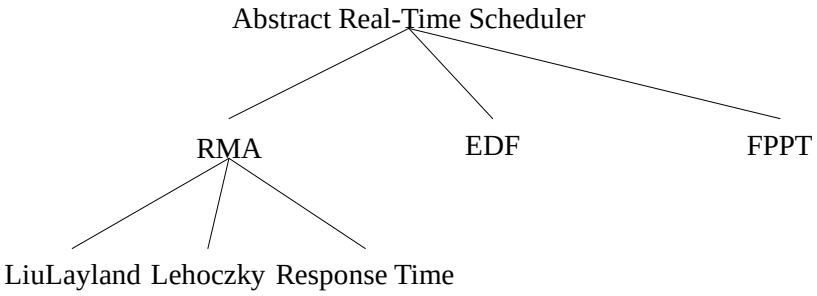


Figure 5.3: Hierarchy of Schedulers

existing. All schedulers can be derived from this abstract scheduler. For example, the partial hierarchical inheritance tree for RMA, EDF, and FPPT is shown in Figure 5.3.

In Figure 5.3, the root of the hierarchical tree is the abstract real-time scheduler, from which RMA, EDF, and FPPT are derived. In addition, RMA can be further subdivided based on the schedulability test algorithms. By default, these schedulers do not consider reservation for aperiodic tasks. When reservation is considered, the reservation size is passed as an explicit parameter to the specific scheduler's constructor to reduce the number of classes. For example, it is unnecessary to derive two classes from EDF, one of which is without reservation and the other with reservation.

The interface for RMA is shown in Figure 5.4. The class RateMonotonicAlgorithm inherits from the abstract interface AbstractPeriodicScheduler. Extra member data and functions are

appended. The second parameter in the constructor has a default value of 0.0, which indicates no reservation is considered. When this parameter is greater than 0.0, it indicates that a non-zero reservation is considered. As there are three schedulability tests in RMA, the pure virtual function test is still not implemented in RMA. When the specific schedulability test is known, test is implemented in a subclass. Like test, schedulingInformation needs to know the name of a specific scheduler to show the scheduling result. Thus, in RMA, it is still not implemented and is still declared as a pure virtual function. Function getFailpoint returns the first job that misses its deadline when a task set cannot be schedulable by RMA.

```
1  #ifndef RateMonotonicAlgorithm_H
2  #define RateMonotonicAlgorithm_H
3  #include "common.h"
4  #include "AbstractPeriodicScheduler.h"
5  #include "RealTimePeriodicTaskSet.h"
6  class RateMonotonicAlgorithm : public AbstractPeriodicScheduler {
7    protected:
8      RealTimePeriodicTaskSet rtpSet; // task set
9      bool schedulable;   // schedulalable or not
10     vector<int> periods;   // periods of tasks
11     vector<double> computations; // computation times of tasks
12     vector<double> deadlines; // deadlines of tasks
13     long long int  cost; // time to perform schedulability test in microsecond
14     double rs; // reservation size
15     FailPoint failpoint; // the failing point when the task set is unschedulable
16   public:
17     RateMonotonicAlgorithm( const RealTimePeriodicTaskSet &taskSet,
18                     double r = 0.0); // default no reservation
19     virtual ~RateMonotonicAlgorithm();
20     long long int getCost() const; // get time to perform schedulability test
21     double getRs() const; // get the reservation size
22     FailPoint getFailpoint(); // get the first task that misses its deadline
23     virtual bool test() = 0;
24     virtual void schedulingInformation( string sname ) = 0;
25     virtual bool tasksetVerify();
26     virtual list<Schedule> schedulelist();
27 };
28 #endif
```

Figure 5.4: Rate Monotonic Scheduler

When a specific schedulability test is known for RMA, a subclass can be derived from RateMonotonicAlgorithm where test and schedulingInformation are overridden. Figure 5.5 shows

the class definitions when the Liu and Layland's schedulability test is used. A private data mem-

```
1   #ifndef RateMonotonicAlgorithmLiuLayland_H
2   #define RateMonotonicAlgorithmLiuLayland_H
3   #include "RateMonotonicAlgorithm.h"
4   class RateMonotonicAlgorithmLiuLayland : public RateMonotonicAlgorithm {
5     private:
6       double Ue; // expected processor utilization
7     public:
8       RateMonotonicAlgorithmLiuLayland( const RealTimePeriodicTaskSet &taskSet,
9                                         double r = 0.0);
10      virtual ~RateMonotonicAlgorithmLiuLayland();
11      virtual bool test(); // Using Liu & Layland's schedulability test
12      virtual void schedulingInformation( string sname );
13  };
14  #endif
```

Figure 5.5: Rate Monotonic Scheduler : LiuLayland

ber stores the expected processor utilization. The two virtual functions test and schedulingInformation are implemented. Note, this schedulability test is only sufficient so schedulingInformation only prints "yes" or "no".

Figure 5.6 shows the class definitions when the Lehoczky's schedulability test presented is used. A data member stores the minimal average processor utilization. The two functions test and schedulingInformation are implemented. In addition, a member function getL is used to get the minimal average processor utilization. Note, this schedulability test is sufficient and necessary so schedulingInformation can print a Gantt-chart.

Figure 5.7 shows the class definitions when the schedulability test based on response time is used. A data member stores the worst case response times of tasks. The two functions test and schedulingInformation are implemented. This schedulability test is sufficient and necessary so schedulingInformation can print a Gantt-chart.

Figure 5.8 shows the class definition for EarliestDeadlineFirst. As no subclass is derived from this scheduler, all pure virtual functions are implemented in this subclass. The two virtual functions test and schedulingInformation are overridden based on EDF. This schedulability test is sufficient and necessary so schedulingInformation can print a Gantt-chart.

Figure 5.9 shows the class definition for FPPT. Extra data members are appended. For example, data members to store the regular priority and preemption threshold assignment, worst case

```
1  #ifndef RateMonotonicAlgorithmLehoczky_H
2  #define RateMonotonicAlgorithmLehoczky_H
3  #include "RateMonotonicAlgorithm.h"
4  class RateMonotonicAlgorithmLehoczky : public RateMonotonicAlgorithm {
5    private:
6      double L;   // minimal average processor untilzation
7      double computeLit(int i, double t);// L for tasks 1,2,...,i at time t
8    public:
9      RateMonotonicAlgorithmLehoczky( const RealTimePeriodicTaskSet &taskSet,
10                                     double r = 0.0 );
11     virtual ~RateMonotonicAlgorithmLehoczky();
12     virtual bool test(); // Using Lehoczky's schedulability test
13     virtual void schedulingInformation( string sname );
14     double getL(); // get the minimal average processor untilzation
15 };
16 #endif
```

Figure 5.6: Rate Monotonic Scheduler : Lehoczky

```
1  #ifndef RateMonotonicAlgorithmResponseTime_H
2  #define RateMonotonicAlgorithmResponseTime_H
3  #include "RateMonotonicAlgorithm.h"
4  class RateMonotonicAlgorithmResponseTime : public RateMonotonicAlgorithm {
5    private:
6      vector<double> R; // response time of J_1,0; J_2,0; ...
7      void computeR();   // compute the worst case response time for each task
8    public:
9      RateMonotonicAlgorithmResponseTime( const RealTimePeriodicTaskSet &taskSet,
10                                        double r = 0.0   );
11     virtual ~RateMonotonicAlgorithmResponseTime();
12     virtual bool test(); // using schedulability test based on response time
13     virtual void schedulingInformation( string sname );
14 };
15 #endif
```

Figure 5.7: Rate Monotonic Scheduler : ResponseTime

```
1  #ifndef EarliestDeadlineFirst_H
2  #define EarliestDeadlineFirst_H
3  #include <string>
4  #include "AbstractPeriodicScheduler.h"
5  #include "RealTimePeriodicTaskSet.h"
6  class EarliestDeadlineFirst : public AbstractPeriodicScheduler {
7    protected:
8      RealTimePeriodicTaskSet rtpSet; // task set
9      bool schedulable; // schedulalable or not
10     vector<int> periods;  // periods of tasks
11     vector<double> computations; // computation times of tasks
12     vector<double> deadlines; // deadlines of tasks
13     long long int  cost; // time to perform schedulability test in microsecond
14     double rs; // reservation size
15     FailPoint failpoint; // the failing point when the task set is unschedulable
16   public:
17     EarliestDeadlineFirst( const RealTimePeriodicTaskSet &taskSet,
18                   double r = 0.0 ); // default no reservation
19     virtual ~EarliestDeadlineFirst();
20     long long int getCost() const; // get time to perform schedulability test
21     double getRs() const;  // get reservation size
22     FailPoint getFailpoint(); // get the first job that misses its deadline
23     virtual bool test();
24     virtual void schedulingInformation( string sname );
25     virtual bool tasksetVerify();
26     virtual list<Schedule> schedulelist();
27 };
28 #endif
```

Figure 5.8: Earliest Deadline First

response times of tasks, the blocking time for each task, and the length of each level-i busy period. In addition, extra functions are also appended. maxFromMin computes the maximal assignment from any valid assignment low; minFromFpp computes the minimal assignment starting from FPP; minFromMax computes the minimal assignment from any valid assignment γ such that $\gamma \neq \gamma_{min}$ but $\gamma_{min} \preccurlyeq \gamma$; maxFromFpnp computes the maximal assignment from FPNP; minFromFpnp computes the minimal assignment from FPNP; i_schedule gets one schedule in the level-i busy period; all_schedule gets one schedule for each level-i busy period. This schedulability test is sufficient and necessary so schedulingInformation can print a Gantt-chart.

```
1  #ifndef FixedPriorityWithPreemptionThreshold_H
2  #define FixedPriorityWithPreemptionThreshold_H
3  #include "AbstractPeriodicScheduler.h"
4  #include "RealTimePeriodicTaskSet.h"
5  class FixedPriorityWithPreemptionThreshold : public AbstractPeriodicScheduler {
6    protected:
7      RealTimePeriodicTaskSet rtpSet;
8      bool schedulable;
9      vector<int> periods; // periods of tasks
10     vector<double> computations; // computation times of tasks
11     vector<double> deadlines; // deadlines of tasks
12     long long int  cost; // time to perform schedulability test in microsecond
13     double rs; // reservation size
14     FailPoint failpoint; // Record the failing point when the task set is unschedulable
15     vector<int> regular; // regular priority
16     vector<int> threshold; // preemption threshold
17     vector<double> level_i_length; // length of level-i busy period
18     vector<double> response; // worst case respone time
19     vector<double> blocking; // blocking time from a task with a lower regular priority
20   public:
21     FixedPriorityWithPreemptionThreshold( const RealTimePeriodicTaskSet &taskSet,
22                                 double r = 0.0 ); // default no reservation
23     virtual ~FixedPriorityWithPreemptionThreshold();
24     long long int getCost() const; // get time to perform schedulability test
25     double getRs() const; // get the reservation size
26     vector<int> maxFromMin( const vector<int>& low); // max assignment from min
27     vector<int> minFromFpp(); // compute minimal assignment from FPP
28     vector<int> minFromMax(const vector<int>& high); // min assignment from max
29     vector<int> maxFromFpnp(); // compute maximal assignment from FPNP
30     vector<int> minFromFpnp(); // compute minimal assignment from FPNP
31     virtual bool test();
32     virtual void schedulingInformation( string sname );
33     virtual bool tasksetVerify();
34     virtual list<Schedule> schedulelist();
35     list<Schedule> i_schedule( int i ); // schedule in one level-i busy period
36     list< list<Schedule> > all_schedule(); // schedule in all level-i busy periods
37 };
38 #endif
```

Figure 5.9: Fixed Priority with Preemption Threshold

5.2.2 Data Structures for Task Sets

A task set is stored in a plain text file. As periodic tasks and aperiodic tasks have different parameters, the structure of a data file to store a periodic task set is different from that for an aperiodic task set.

1. A periodic task set is composed of one or more periodic tasks. The internal structure is: the first value N, is the number of tasks in the set, followed by N 4-tuples, one for each task. Each tuple representing a periodic task contains its period, deadline, worst case computation time, and phase respectively. For example, the following is a file storing 5 periodic tasks.

```
5
2    2    1    0
4    4    1    0
7    7    1    0
14   14   1    0
28   28   1    0
```

   In this example, for each periodic task, its period is equal to its deadline, the worst case computation time is 1, and its phase is 0.

2. An aperiodic task set is composed of one or more aperiodic tasks. The internal structure is: the first value N, is the number of tasks in the set, followed by N 2-tuples, one for each task. Each tuple representing an aperiodic task contains its arrival time and worst case computation time, respectively. For example, the following is a file storing 3 aperiodic tasks.

```
3
10.5    1.5
61.1    3.4
200.9   2.7
```

   The first aperiodic task arrives at time 10.5 with worst case computation time 1.5.

5.2.3 User Interface

The tool-kit is implemented with a graphical user interface (GUI) and the GUI libraries are from Gtkmm [21, 30], which is similar to Motif for X-Windows. The main window of the tool-kit is shown in Figure 5.10. In the main window, there is a menu bar containing one menu item to choose a scheduler. Notice that some schedulers such as RateMonotonicAlgorithm, have an additional selection menu to specify the specific schedulability test. When reservation is considered with RMA, one sufficient and necessary schedulability test is chosen. That is why there is no submenu for menu item RateMonotonicAlgorithmWithReservation.

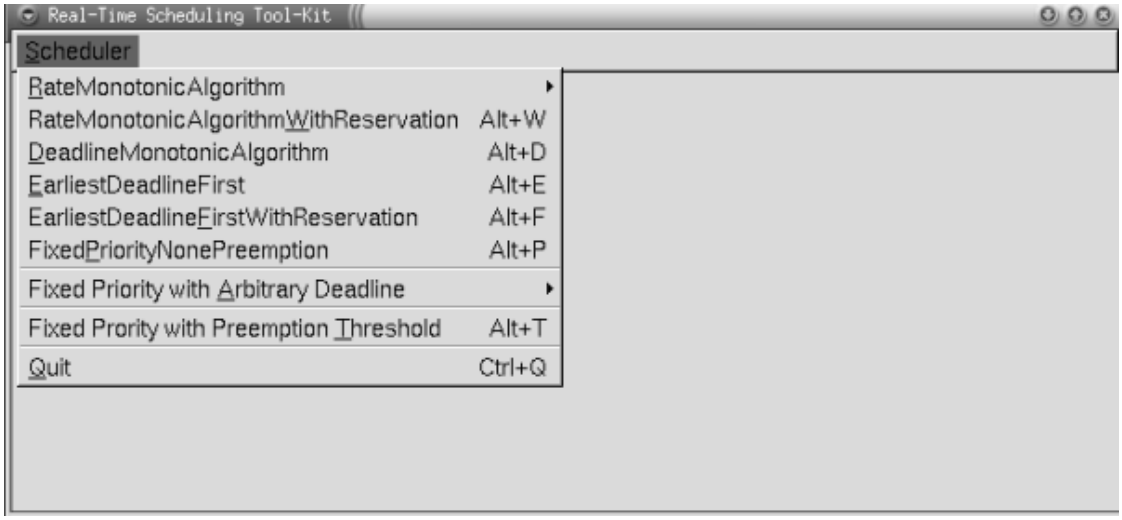


Figure 5.10: Tool-Kit Main Window

After a scheduler is chosen, another window pops up, which is shown in Figure 5.11. There is a menu bar in the pop-up window to perform data file operations such as generating new data files and selecting existing data files.

Below the menu bar, there is a drawing area to show the scheduling results based on the chosen scheduler and data file when available. A schedule is shown in this area based on a time line. Each task is indicated by a distinct color. The schedule is shown up to one major cycle. If a task set stored in the chosen data file is schedulable by the chosen scheduler, one schedule is shown in this area. If the task set is unschedulable, then the scheduling result up to the job missing its deadline is shown. The job missing its deadline is also specified.

At the bottom of the window, are seven buttons to control the schedule. These buttons are

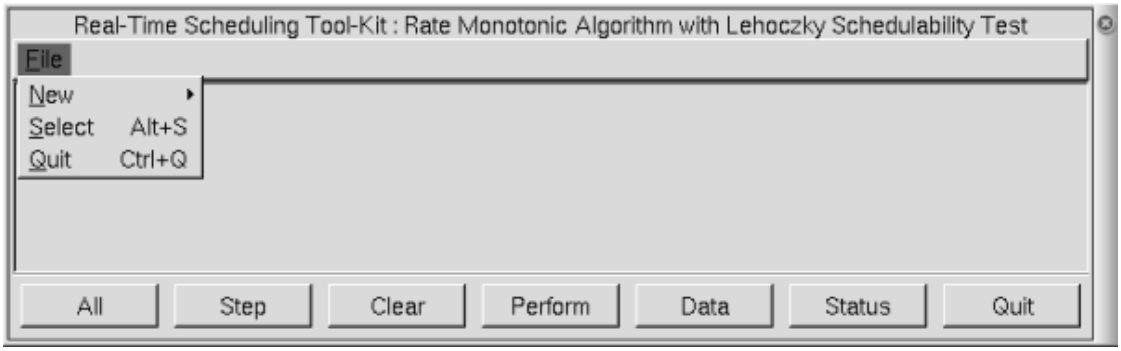


Figure 5.11: Tool-Kit Scheduler Window

explained from left to right based on the chosen scheduler and data file. Clicking the button "All" shows one schedule completely. Clicking the button "Step" moves through a schedule step by step. For example, when RMA is chosen, the scheduling result of each task is appended for each step. When EDF is chosen, the schedule of one job is appended for each step. Clicking the button "Clear" clears the schedule. Clicking the button "Perform" performs a schedulability test. Clicking the button "Data" shows the content of the chosen data file. Clicking the button "Status" shows whether a task set passes the corresponding schedulability test. Clicking the button "Quit" closes the window.

5.3 Case Study

This section traces some examples with the tool-kit to demonstrate the functionality of the tool-kit and explain how to use its graphical user interface. The examples also demonstrate how the tool-kit was used to verify the theoretical results of this thesis.

5.3.1 Case for Different Schedulability Tests for RMA

Consider an example task set that passes the Liu and Layland's schedulability test. Given tasks $\tau_1 = (3, 0.9)$, $\tau_2 = (5, 1.5)$, and $\tau_3 = (6, 0.6)$, then $U = 0.7$, and $L = 0.78$. As $U_e(3) \approx 0.779763$ and $0.779763 > 0.7$, these tasks are schedulable by RMA based on the Liu and Layland's schedulability test. Figure 5.12 shows the scheduling result after scheduling the first task, the second task, and the third task, respectively.

In each picture, the top line is the time line and at each step, another task is added to the time

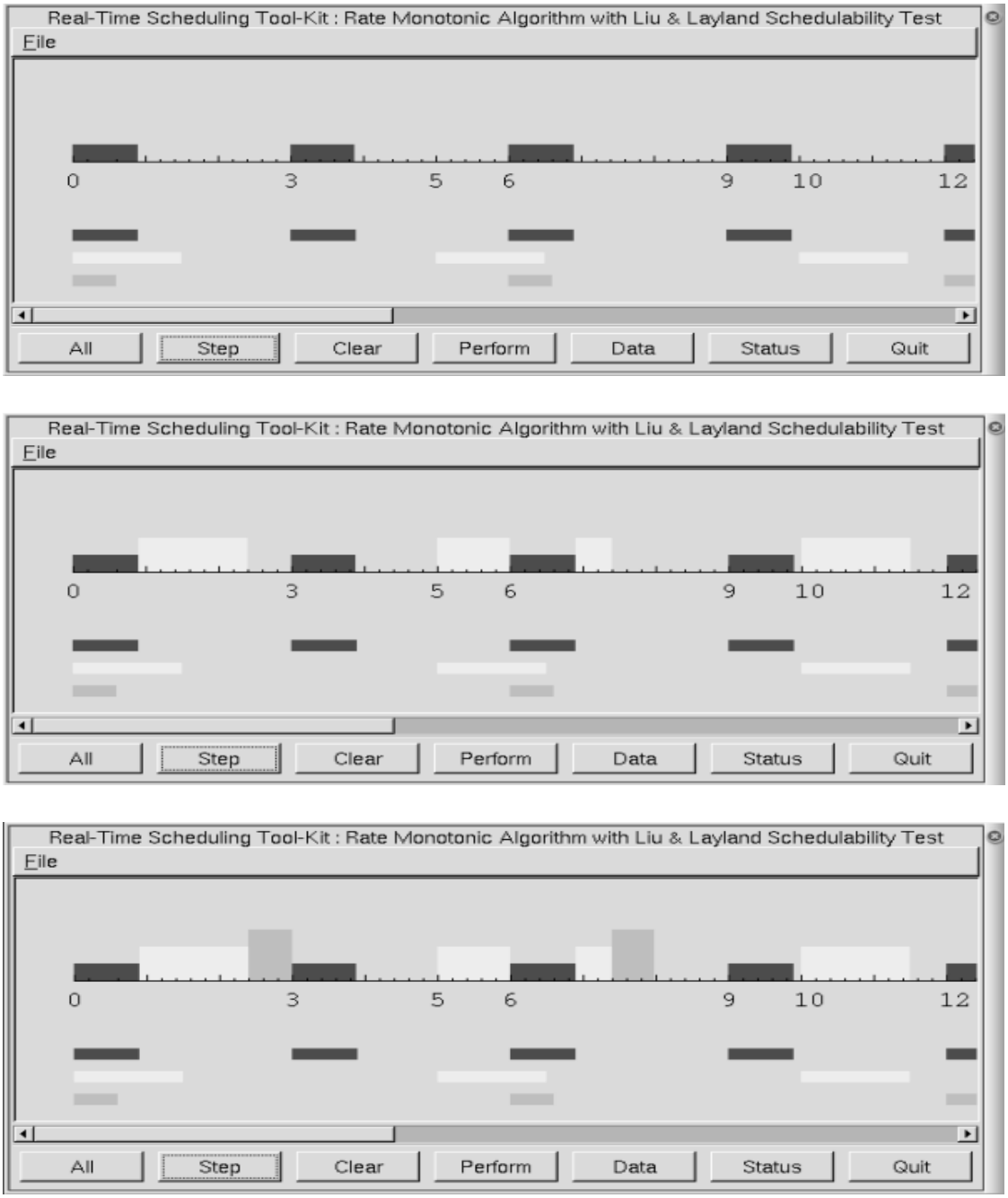


Figure 5.12: RMA Liu and Layland's Schedulability Test Step by Step (step 1, 2, 3)

line. The bottom 3 lines show when each job enters the system. The scroll bar at the bottom allows scrolling along the time line. The different heights of the lines for each task do not represent any value; it only helps visually distinguish among the tasks on the task line.

Now consider an example task set that fails the Liu and Layland's schedulability test but passes the Lehoczky's schedulability test. Given tasks $\tau_1 = (3, 1.2)$, $\tau_2 = (5, 1.5)$, and $\tau_3 = (6, 0.6)$, then $U = 0.8$. These tasks fail the Liu and Layland's schedulability test because $0.8 > 0.779763$. As the Liu and Layland's schedulability test is only sufficient, a task set failing this schedulability test may still be schedulable by RMA. In fact, based on the Lehoczky's schedulability test for RMA, $L = 0.9 < 1$, so the task set is schedulable because $0.9 \leq 1$. The schedule is shown in Figure 5.13.

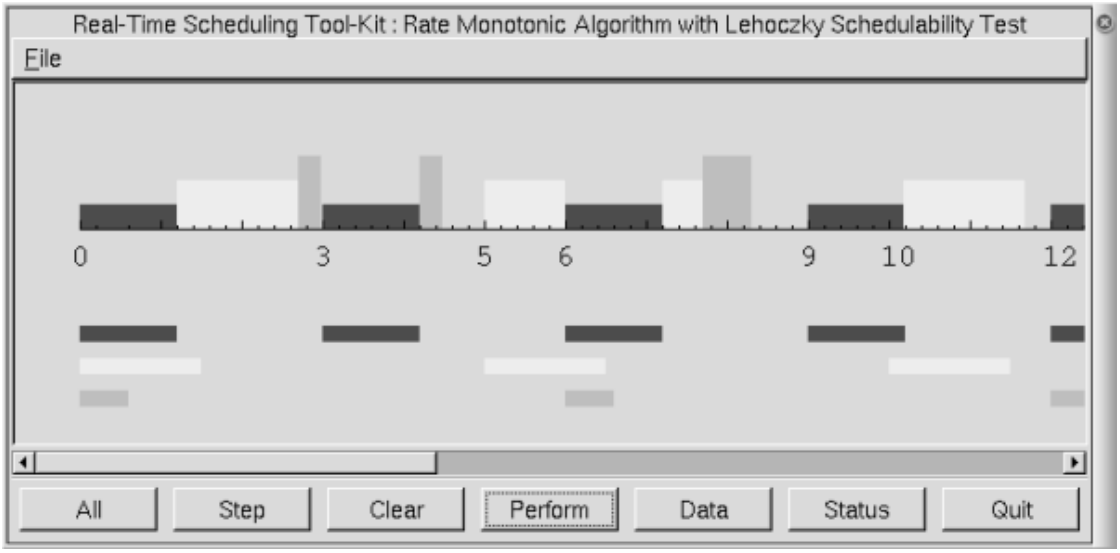


Figure 5.13: RMA Lehoczky Schedulability Test

Now consider an example task set that fails the Lehoczky schedulability test. Given tasks $\tau_1 = (2, 0.2)$, $\tau_2 = (3, 1.2)$, $\tau_3 = (5, 1.5)$, and $\tau_4 = (6, 0.6)$, $L = 1.02 > 1$. The task set is unschedulable by RMA. The scheduling result is shown in Figure 5.14. As shown in the diagram, $J_{4,0}$ misses its deadline of 6, implying that this task set is unschedulable by RMA.

However, this task set is schedulable by FPPT with $\pi = \{1, 2, 3, 4\}$, $\gamma_{min} = \{1, 2, 3, 3\}$, and $\gamma_{Max} = \{1, 1, 1, 1\}$. (see another example in detail in Section 5.3.3)

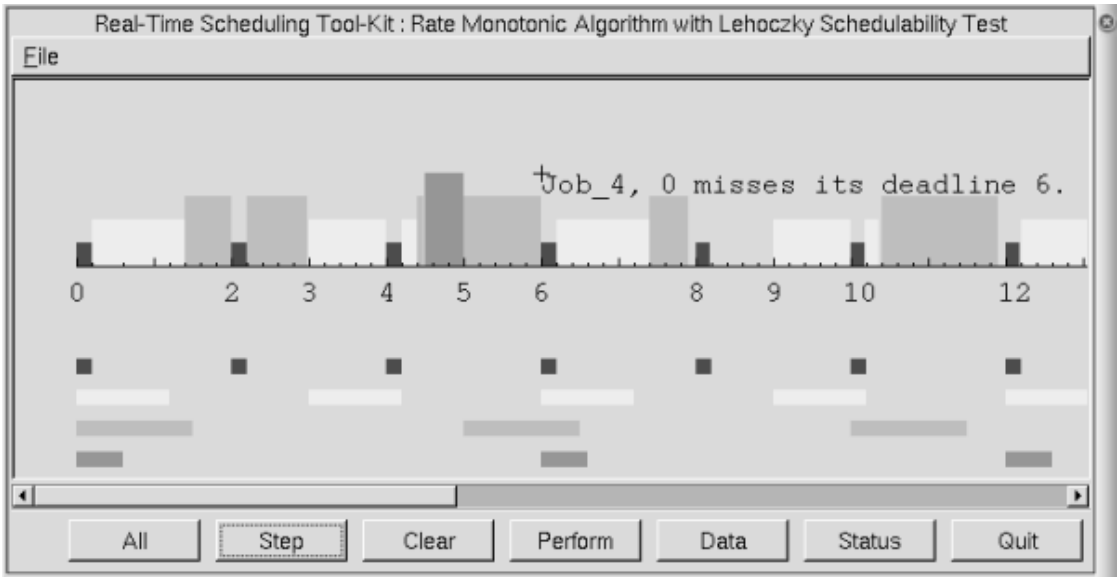


Figure 5.14: Task Set Unschedulable by RMA

5.3.2 Case for *RBA_RMA*

Given tasks $\tau_1 = (3, 1.2)$, $\tau_2 = (5, 1.5)$, and $\tau_3 = (6, 0.6)$, then $U = 0.8$ and $L = 0.9$. Based on *RBA_RMA*, the maximal reservation size is $1 - L = 0.1$. When a scheduler with reservation is selected, clicking the "Perform" button first pops up a dialog window prompting for a reservation size, which is shown on the upper part in Figure 5.15. Note, the maximal reservation size R_s is already calculated by *RBA_RMA*. Any value in the range $[0.0, R_s]$ is a valid reservation size. In Figure 5.15, the slider has been adjusted to the current reservation size of 0.06, which is shown on the top of the slider.

After the reservation is selected, it is used to initialize *RBA_RMA* in order to generate one schedule. The scheduling result is shown on the lower part in Figure 5.15. In the diagram, the reservation is the small black bar located at the beginning of each time unit.

5.3.3 Case for FPPT

Given tasks $\tau_1 = (10, 1)$, $\tau_2 = (15, 1)$, $\tau_3 = (40, 4)$, $\tau_4 = (60, 8)$, $\tau_5 = (80, 25)$, $\tau_6 = (100, 10)$, $\tau_7 = (155, 14)$, and $\tau_8 = (190, 6)$, then $L = 1.0129 > 1$. Based on Theorem **ERMA2**, the task set is unschedulable by RMA. However, it is schedulable by FPPT with $\pi = \{1, 2, 3, 4, 5, 6,$

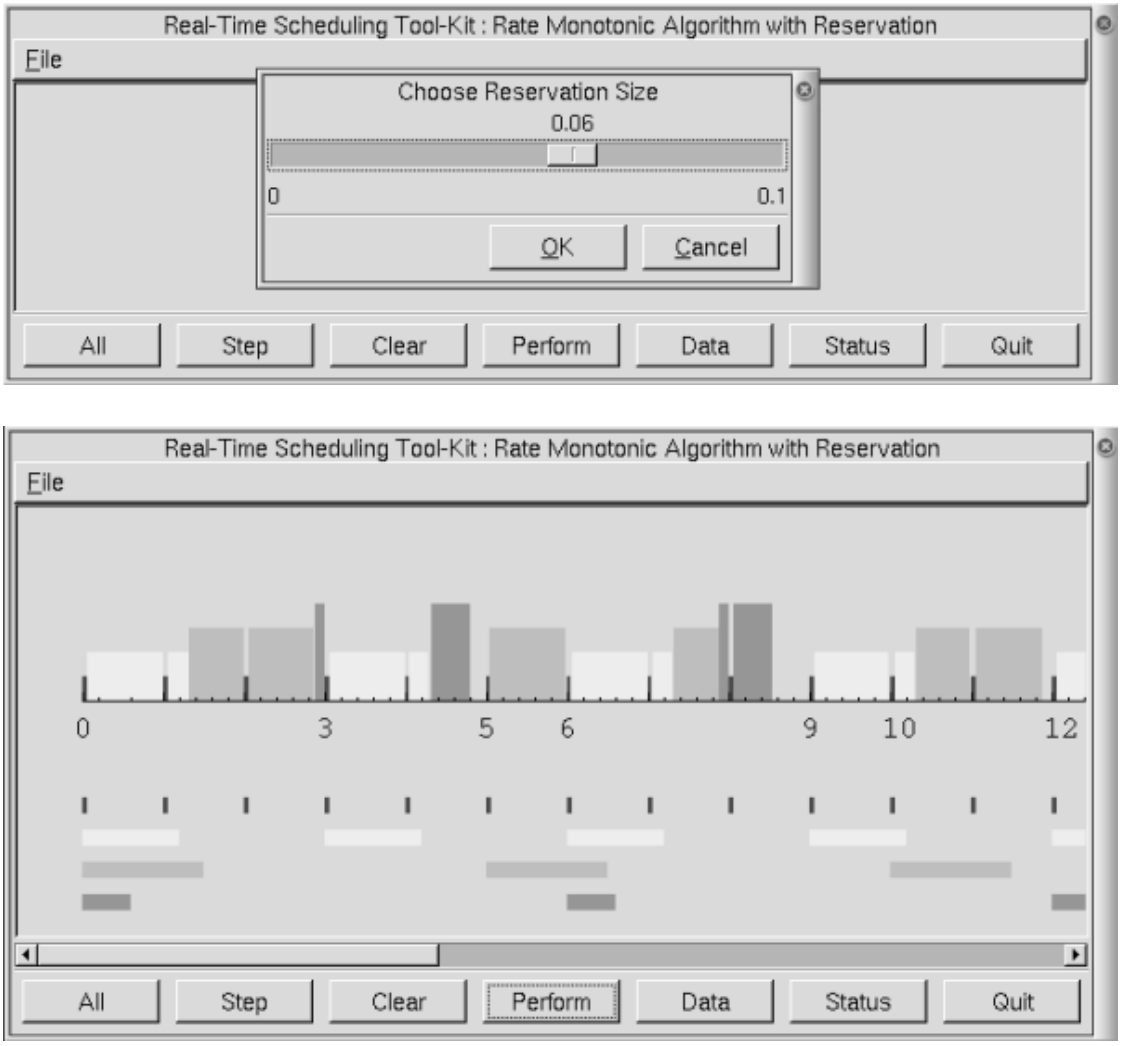


Figure 5.15: Choose Reservation Size for *RBA_RMA*

7, 8}, $\gamma_{min} = \{1, 2, 3, 4, 5, 5, 6, 7\}$, and $\gamma_{Max} = \{1, 1, 1, 1, 3, 2, 3, 1\}$. The minimal and maximal assignments are shown in Figure 5.16. The upper polyline corresponds to the minimal assignment and the lower one corresponds to the maximal assignment.

5.3.4 Case for Minimal and Maximal Assignments in FPPT

Consider the same task set in Section 3.5.4, tasks are $\tau_1 = (10, 1)$, $\tau_2 = (15, 1)$, $\tau_3 = (40, 4)$, $\tau_4 = (60, 10)$, $\tau_5 = (80, 20)$, $\tau_6 = (100, 15)$, $\tau_7 = (200, 10)$, and $\tau_8 = (240, 16)$. $\gamma_{min} = \{1,$

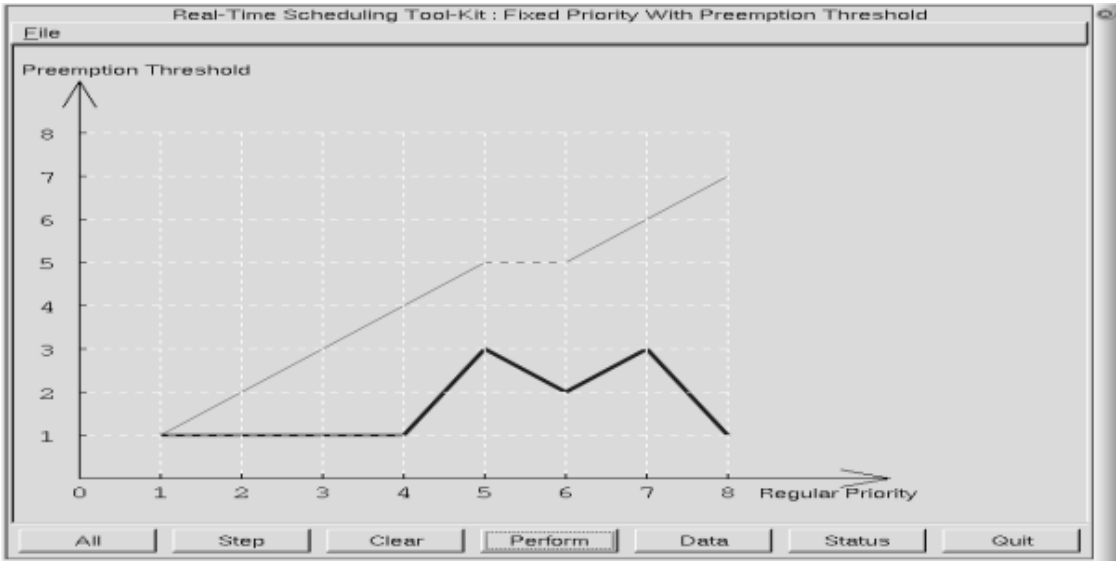


Figure 5.16: Sample Scheduling of FPPT

2, 3, 4, 5, 5, 5, 7$\}$, and $\gamma_{Max} = \{1, 1, 1, 2, 3, 3, 2, 3\}$. Different algorithms presented in Section 3.6 are used to compute the minimal and maximal assignments. The results are shown in Figure 5.17. Clearly, the corresponding results are consistent because the graphs for the minimal (maximal) assignments calculated by the different algorithms are the same. Hence, the minimal (maximal) assignments computed by the different algorithms are the same, which also verifies the correctness of the algorithms presented in Section 3.6.

5.4 Two Interesting Results

In addition to verifying the correctness of the research work in this thesis, the tool-kit was also used to find two interesting results. These results are presented in this section.

5.4.1 Counter Example

Given tasks τ_1, τ_2, ..., τ_n, FPP can perform an incremental schedulability test on the task set because FPP is robust under the RMA critical instant. That is, any subset of a task set must be schedulable by FPP if the whole task set is schedulable by FPP. However, this incremental schedulability test does not hold in FPNP as FPNP is not robust under the RMA critical instant (see Section 2.1.2). One approach to prove FPNP is not robust under the RMA critical instant

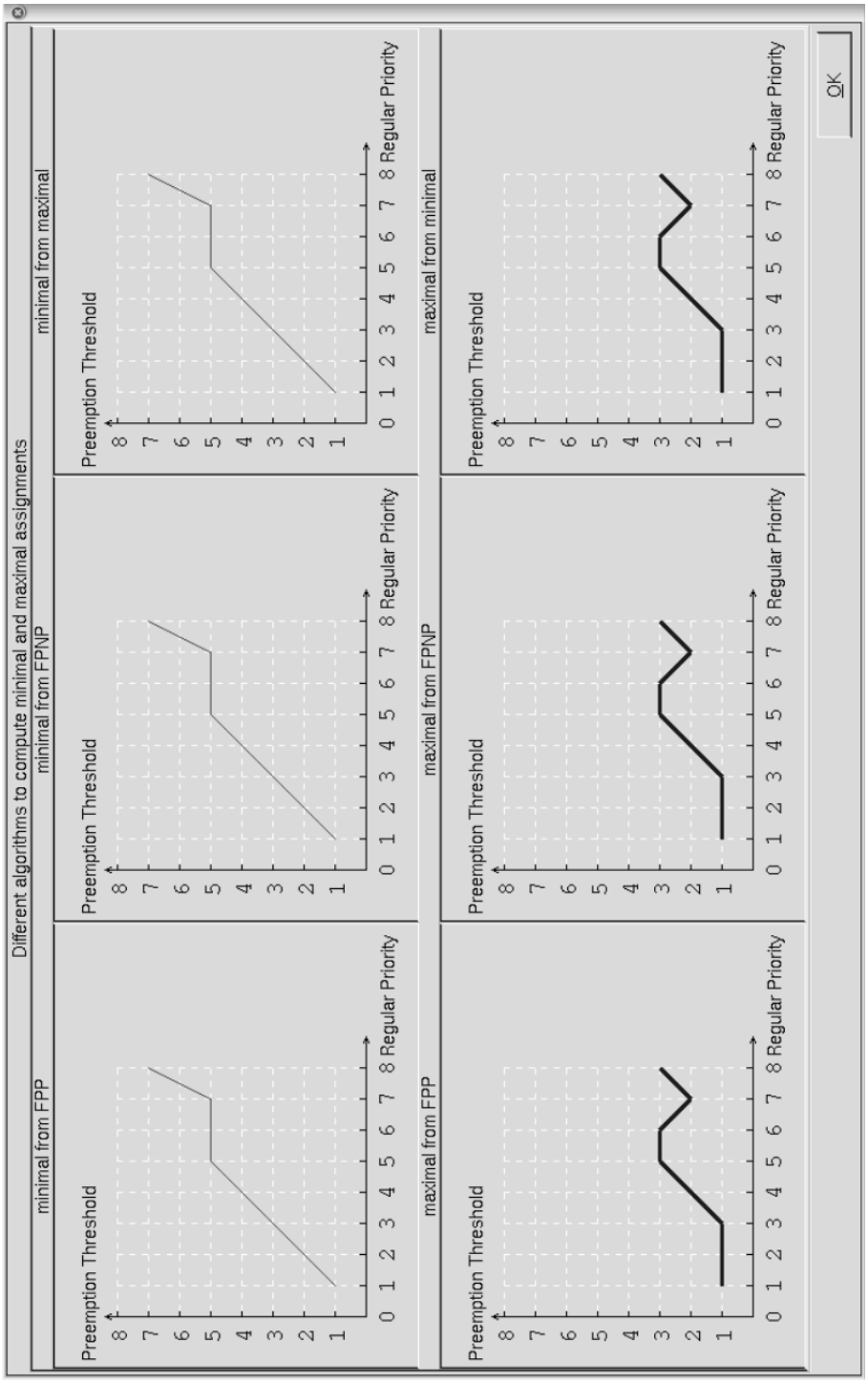


Figure 5.17: Different Algorithms to Compute Minimal and Maximal Assignments in FPPT

is to generate a counter example. The tool-kit provides a convenient and fast mechanism to search for a counter example. Starting with a simple task set, an intuitive adjustments can be made to the task set to construct a counter example in a couple of hours. For example, tasks $\tau_1 = (4, 1.1)$, $\tau_2 = (7, 4)$, and $\tau_3 = (28, 1.8)$ are schedulable by FPNP based on the RMA critical instant. However, tasks τ_1 and τ_2 are unschedulable by FPNP based on the RMA critical instant. The scheduling results in the first major cycle based on the RMA critical instant are shown in Figure 5.18[1].

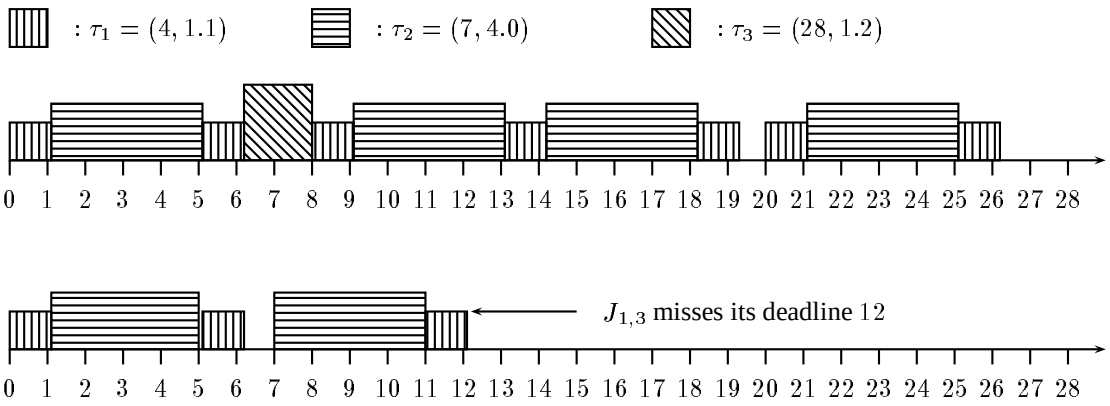


Figure 5.18: FPNP not Robust under RMA Critical Instant

The upper part of Figure 5.18 shows that the task set composed of tasks τ_1, τ_2, and τ_3 is schedulable by FPNP based on the RMA critical instant. The lower part of Figure 5.18 shows that the task set composed of only tasks τ_1 and τ_2 is unschedulable by FPNP based on the RMA critical instant as $J_{1,3}$ misses its deadline at 12. This example indicates that FPNP is not robust under the RMA critical instant because when the computation time of task τ_3 is decreased to 0, resulting in that task τ_3 is removed from the original task set, the new task set is unschedulable. Therefore, the incremental schedulability test does not work on FPNP based on the RMA critical instant.

[1]This figure can be generated by the tool-kit. However, converting to gray scale when the number of colors is over 2 is poor. Hence, this figure is hand generated using patterns instead of gray scale. Figure 5.19 is also hand generated for the same reason.

5.4.2 Semi-Harmonic Periodic Task Sets

An interesting problem is to find task sets with maximal processor utilization, which is equal to 1. For example, a harmonic task set (see page 9) is schedulable by RMA if and only if its processor utilization is not greater than 1 [72]. Given a set of periods that are harmonic, the computation times can be scaled to achieve a processor utilization of 1. Thus, harmonic task sets represent a class of task sets that are schedulable by RMA with processor utilization of 1.

Is it possible for non-harmonic task sets to achieve a processor utilization of 1? First, this section shows that such task sets exist and all such task sets must satisfy a special condition. Second, one related work is rephrased in brief. Third, a special class of such periodic task sets based on perfect numbers [76] is presented, which removes the limitations in the related work and the necessity to start from a task set schedulable by RMA with processor utilization less than 1. Though the form of this special class of periodic task sets is limited, these task sets can be good examples for teaching in real-time scheduling.

Definition 4
Given a task set, if the period of each task is a divisor of the longest period, then the task set is called a semi-harmonic task set.

For example, a task set composed of four tasks with periods 2, 3, 6, and 12 is semi-harmonic. Clearly, if a periodic task set is harmonic, then it must be semi-harmonic. However, if a periodic task set is semi-harmonic, it may not be harmonic.

Theorem 11
Given periodic tasks $\tau_i = (T_i, C_i), T_i$ *is an integer, for* $1 \leq i \leq n, T_1 \leq T_2 \leq \ldots \leq T_n$, $U = \sum_{i=1}^{n} \frac{C_i}{T_i} = 1$*, if they are schedulable by RMA, then* $T_i \mid T_n$ *for* $1 \leq i \leq n$*, i.e., the task set is semi-harmonic.*

Proof: As task τ_n is schedulable, then there must exist a t such that $t = \sum_{j=1}^{n} C_j \lceil \frac{t}{T_j} \rceil$ [15], where

$0 < t \leq T_n$. However,

$$\begin{aligned} t &= \sum_{j=1}^{n} C_j \cdot \lceil \frac{t}{T_j} \rceil \\ &\geq \sum_{j=1}^{n} C_j \cdot \frac{t}{T_j} \\ &= t \sum_{j=1}^{n} \frac{C_j}{T_j} \\ &= t \cdot U \\ &= t \cdot 1 \\ &= t \end{aligned}$$

Note, the equality in $\geq$ holds if and only if t is an integer multiple of T_j for $1 \leq j \leq n$. As the processor utilization of the task set is 1, task τ_n meets its deadline at time T_n. Thus, $t = T_n$, implying that T_n is a multiple of T_j for $1 \leq j \leq n$. Therefore, these tasks are semi-harmonic. ■

Theorem 11 indicates that a task set schedulable by RMA with a processor utilization of 1 must be semi-harmonic.

Lord [48] presents examples of non-harmonic task sets schedulable by RMA with utilization 1. Such a non-harmonic task set can be derived based on the formulae $\tau_i = (T_i, T_i/n)$ and $T_i = 1/((n-i+1)f)$, where n is the number of tasks and f is called a frequency[2]. For example, when $f = 0.01$ and $n = 4$, four tasks are calculated,

$$\tau_1 = (25, 6.25), \qquad \tau_2 = (33.33, 8.33), \qquad \tau_3 = (50, 12.5), \qquad \tau_4 = (100, 25).$$

Such a task set is semi-harmonic but not harmonic with processor utilization 1. Lord's work on non-harmonic task sets does not formally prove that such a task set is schedulable by RMA. Furthermore, the task sets proposed by Lord have two limitations. First, the period of a task may not be an integer. Second, the processor utilization of each task is the same.

It is straightforward to create a semi-harmonic task set that is not harmonic but schedulable by RMA with processor utilization equal to 1. Start with a task set schedulable by RMA, which is not harmonic and whose processor utilization is less than 1. Suppose the processor utilization

[2] In [48], T_i is given as $1/(n-(i-1)f)$, which is incorrect.

is U and the major cycle is T_{mc}. Create another task with period T_{mc} and computation time $T_{mc}(1-U)$. Then the new task set is schedulable by RMA with processor utilization 1 but it is not harmonic. The correctness of the above result is apparent. First, the processor utilization of the task set is $U + T_{mc}(1-U)/T_{mc} = U + 1 - U = 1$. Second, as the appended task has the longest period, based on RMA, its priority is lowest. Hence, appending the new task cannot affect the schedulability of tasks in the original task set. It remains to consider the schedulability of the appended task. As the major cycle of the original task set is T_{mc} and in one major cycle the unused time left by the original task set is $T_{mc}(1-U)$, which is equal to the computation time of the appended task, implying that the appended task can meet its deadline. Thus, it is schedulable. Third, as the original task set is not a harmonic task set, there exists at least one pair of periods T_i and T_j such that $T_i \nmid T_j$, where $i \neq j$ and $T_j > T_i$. This property still holds in the new task set as no task is removed in the new task set, resulting in that the new task set is not harmonic.

However, the limitation is that the starting task set must be schedulable by RMA and not harmonic, requiring a schedulability test be performed on the task set. In fact, there exist many task sets such that the periods of all tasks are integers and the processor utilization of each task is different. For example, periodic tasks $\tau_1 = (2, 1)$, $\tau_2 = (3, 1)$, and $\tau_3 = (6, 1)$ are schedulable by RMA with processor utilization of 1. Note, 6 is the smallest perfect number. Now, consider a more complicated example. Given tasks

$$\tau_1 = (2, 1), \qquad \tau_2 = (4, 1), \qquad \tau_3 = (7, 1), \qquad \tau_4 = (14, 1), \qquad \tau_5 = (28, 1),$$

they are schedulable by RMA with processor utilization $\frac{1}{2} + \frac{1}{4} + \frac{1}{7} + \frac{1}{14} + \frac{1}{28} = 1$. The Gantt-chart of the schedule of all the jobs in the first major cycle by RMA is shown in Figure 5.19. Each task has computation time 1 and 28 is an even perfect number, whose distinct positive divisors greater than 1 are 2, 4, 7, 14, and 28. Perhaps this observation is a coincidence. By using the tool-kit to quickly test a number of similar task sets, it was possible to get positive feedback that the observation might be an actual consequence, and hence, a formal examination might be fruitful.

Conjecture: given a perfect number, if a task set is derived such that each period corresponds to a distinct divisor of the perfect number greater than 1 and the computation time of each task is 1, then such a task set is schedulable by RMA with processor utilization of 1. The remainder of this section examines this form of task sets in detail.

Perfect numbers are used in the following discussion. These numbers do not appear arbitrarily, but rather fell out from the following observation. Given a positive integer n and all of

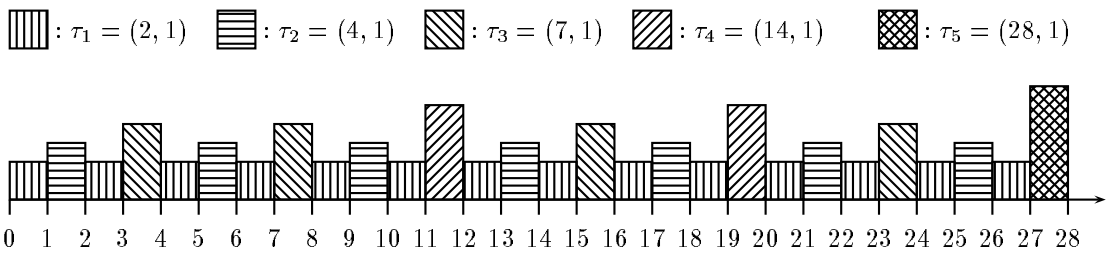


Figure 5.19: A Task Set based on Perfect Number 28

its distinct positive divisors, $d_0, d_1, d_2, \ldots, d_r$, where $1 = d_0 < d_1 < d_2 < \ldots < d_r = n$, if $\sum_{j=0}^{r-1} d_j = n$, then n is called a perfect number. Given a perfect number n, let D be the set of such divisors. Thus, $D = \{d : d|n, 1 \leq d < n\}$. Then, $\sum_{d \in D} d = n$, which can be transformed to $\sum_{d \in D} \frac{d}{n} = 1$, resulting in $\sum_{d \in D} \frac{1}{\frac{n}{d}} = 1$. Interestingly, $\sum_{d \in D} \frac{1}{\frac{n}{d}}$ is similar to the right-hand side of Formula 2.1 to calculate the processor utilization of a task set. This similarity leads to creating a task set $S = \{\tau = (\frac{n}{d}, 1) : d \in D\}$ whose processor utilization is equal to 1. If task set S is schedulable by RMA, then it is a task set that is schedulable by RMA with processor utilization 1. In addition, such a task set is semi-harmonic but not harmonic. As only even perfect numbers are found up to now, only even perfect numbers are considered.

Based on the Euclid-Euler Theorem [76], n is an even perfect number if and only if $n = 2^{p-1}(2^p - 1)$, where $2^p - 1$ is a Mersenne prime [76]. Therefore, an even perfect number must be of the form $2^p(2^{p+1} - 1)$. Hence, all of its distinct positive divisors are 1, 2, ..., 2^p, $2^{p+1} - 1$, $2(2^{p+1} - 1)$, $2^2(2^{p+1} - 1)$, ..., $2^p(2^{p+1} - 1)$, increasingly. In fact, the general denotation of these divisors is as follows:

$$d_i = \begin{cases} 2^i & 0 \leq i \leq p, \\ 2^{i-(p+1)}(2^{p+1} - 1) & p + 1 \leq i \leq 2p + 1. \end{cases} \tag{5.1}$$

One relationship among d_i, d_{2p+1-i}, and n for $\forall$i, $0 \leq i \leq 2p + 1$, is presented formally in the following lemma, which is applied in the proof of Theorem 12.

Lemma 11

Given an even perfect number $n = 2^p(2^{p+1} - 1)$*, all of its distinct positive divisors are* $d_i = 2^i$ *for* $0 \leq i \leq p$ *and* $d_i = 2^{i-(p+1)}(2^{p+1} - 1)$ *for* $p + 1 \leq i \leq 2p + 1$ *based on Formula 5.1. Then* $d_i * d_{2p+1-i} = n$ *for* $\forall i$, $0 \leq i \leq 2p + 1$.

Proof: The key point is to show that when $0 \leq i \leq p$, $p + 1 \leq 2p + 1 - i \leq 2p + 1$ and when $p + 1 \leq i \leq 2p + 1$, $0 \leq 2p + 1 - i \leq p$. Then based on Formula 5.1, $d_i * d_{2p+1-i} = n$ can be verified.

1. $\forall$i, $0 \leq i \leq p$. Then,

$$\begin{aligned} 0 \leq i \leq p &\implies -p \leq -i \leq 0 \qquad \text{multiplying each side with } -1 \\ &\implies 2p + 1 - p \leq 2p + 1 - i \leq 2p + 1 \\ &\qquad \text{adding } 2p + 1 \text{ to each side} \\ &\implies p + 1 \leq 2p + 1 - i \leq 2p + 1 \end{aligned}$$

   Thus,

$$\begin{aligned} d_i * d_{2p+1-i} &= 2^i * 2^{2p+1-i-(p+1)}(2^{p+1} - 1) \qquad \text{based on Formula 5.1} \\ &= 2^p(2^{p+1} - 1) \\ &= n \end{aligned}$$

2. $\forall$i, $p + 1 \leq i \leq 2p + 1$. Then,

$$\begin{aligned} p + 1 \leq i \leq 2p + 1 &\implies -(2p + 1) \leq -i \leq -(p + 1) \\ &\qquad \text{multiplying each side with } -1 \\ &\implies 2p + 1 - (2p + 1) \leq 2p + 1 - i \leq 2p + 1 - (p + 1) \\ &\qquad \text{adding } 2p + 1 \text{ to each side} \\ &\implies 0 \leq 2p + 1 - i \leq p \end{aligned}$$

Thus,

$$\begin{aligned} d_i * d_{2p+1-i} &= 2^{i-(p+1)}(2^{p+1}-1) * 2^{2p+1-i} \qquad \text{based on Formula 5.1} \\ &= 2^p(2^{p+1}-1) \\ &= n \end{aligned}$$

Therefore, $d_i * d_{2p+1-i} = n$ for $\forall$i, $0 \leq i \leq 2p+1$. ■

Given an even perfect number, if all of its distinct divisors greater than 1 are considered to be periods of a periodic task set and the computation time of each task is equal to 1, then such a task set is semi-harmonic but not harmonic, which may be schedulable by RMA with processor utilization 1. This observation is formally specified in Theorem 12. Lemma 12 is proved first, which is used in the proof of Theorem 12.

Lemma 12

Given an even perfect number $n = 2^p(2^{p+1}-1)$, all of its distinct divisors are $d_i = 2^i$ for $0 \leq i \leq p$ and $d_i = 2^{i-(p+1)}(2^{p+1}-1)$ for $p+1 \leq i \leq 2p+1$ based on Formula 5.1. Derive a periodic task set $S_{2p+1} = \{\tau_i = (T_i, C_i) = (d_i, 1) : 1 \leq i \leq 2p+1\}$, then $\sum_{j=1}^{i} \left\lceil \frac{T_i}{T_j} \right\rceil = T_i$ for periodic task set S_{2p+1}, where $p+1 \leq i \leq 2p+1$.

Proof: The proof is composed of three steps.

$$\sum_{j=1}^{i} \left\lceil \frac{T_i}{T_j} \right\rceil = \sum_{j=1}^{p} \left\lceil \frac{T_i}{T_j} \right\rceil + \sum_{j=p+1}^{i} \left\lceil \frac{T_i}{T_j} \right\rceil$$

Let $X = \sum_{j=1}^{p} \left\lceil \frac{T_i}{T_j} \right\rceil$ and $Y = \sum_{j=p+1}^{i} \left\lceil \frac{T_i}{T_j} \right\rceil$. The next two steps calculate X and Y.

1. Calculate X

$$
\begin{aligned}
X &= \sum_{j=1}^{p} \left\lceil \frac{T_i}{T_j} \right\rceil \\
&= \sum_{j=1}^{p} \left\lceil \frac{d_i}{d_j} \right\rceil \qquad \text{as } T_i = d_i \text{ and } T_j = d_j \\
&= \sum_{j=1}^{p} \left\lceil \frac{2^{i-(p+1)}(2^{p+1}-1)}{T_j} \right\rceil \\
&\quad \text{as } p+1 \le i \le 2p+1,\ d_i = 2^{i-(p+1)}(2^{p+1}-1) \\
&\quad \text{based on Formula 5.1} \\
&= \sum_{j=1}^{p} \left\lceil \frac{2^{i-(p+1)}(2^{p+1}-1)}{2^j} \right\rceil \\
&\quad \text{as } 1 \le j \le p,\ d_j = 2^j \text{ based on Formula 5.1} \\
&= \sum_{j=1}^{i-(p+1)} \left\lceil \frac{2^{i-(p+1)}(2^{p+1}-1)}{2^j} \right\rceil \\
&\quad + \sum_{j=i-(p+1)+1}^{p} \left\lceil \frac{2^{i-(p+1)}(2^{p+1}-1)}{2^j} \right\rceil \\
&= \sum_{j=1}^{i-(p+1)} \left\lceil \frac{2^{i-(p+1)}(2^{p+1}-1)}{2^j} \right\rceil + \sum_{j=i-(p+1)+1}^{p} \left\lceil \frac{2^i}{2^j} - \frac{2^{i-(p+1)}}{2^j} \right\rceil
\end{aligned}
$$

Let $A = \sum_{j=1}^{i-(p+1)} \left\lceil \frac{2^{i-(p+1)}(2^{p+1}-1)}{2^j} \right\rceil$ and $B = \sum_{j=i-(p+1)+1}^{p} \left\lceil \frac{2^i}{2^j} - \frac{2^{i-(p+1)}}{2^j} \right\rceil$, then

$$
\begin{aligned}
A &= \sum_{j=1}^{i-(p+1)} \left\lceil \frac{2^{i-(p+1)}(2^{p+1}-1)}{2^j} \right\rceil \\
&= \sum_{j=1}^{i-(p+1)} \frac{2^{i-(p+1)}(2^{p+1}-1)}{2^j} \qquad \text{as } j \le i-(p+1) \\
&= 2^{i-(p+1)}(2^{p+1}-1) \sum_{j=1}^{i-(p+1)} \frac{1}{2^j} \\
&= 2^{i-(p+1)}(2^{p+1}-1)(1-\frac{1}{2^{i-(p+1)}}) \\
&= 2^{i-(p+1)}(2^{p+1}-1) - (2^{p+1}-1) \\
&= T_i - (2^{p+1}-1) \qquad \text{as } p+1 \le i \text{ and } T_i = d_i = 2^{i-(p+1)}(2^{p+1}-1) \\
&= T_i + 1 - 2^{p+1}
\end{aligned}
$$

$$
\begin{aligned}
B &= \sum_{j=i-(p+1)+1}^{p} \left\lceil \frac{2^i}{2^j} - \frac{2^{i-(p+1)}}{2^j} \right\rceil \\
&= \sum_{j=i-(p+1)+1}^{p} \frac{2^i}{2^j} \\
&\quad \text{as } i \ge j \text{ and } j > i-(p+1) \text{ therefore } 0 < \frac{2^{i-(p+1)}}{2^j} < 1 \\
&= 2^i \sum_{j=i-(p+1)+1}^{p} \frac{1}{2^j} \\
&= 2^i \left(\frac{1}{2^{i-(p+1)}} - \frac{1}{2^p} \right) \\
&= 2^{p+1} - 2^{i-p}
\end{aligned}
$$

Thus, $X = A + B = T_i + 1 - 2^{p+1} + 2^{p+1} - 2^{i-p} = T_i + 1 - 2^{i-p}$.

2. Calculate Y

$$\begin{aligned}
Y &= \sum_{j=p+1}^{i} \left\lceil \frac{T_i}{T_j} \right\rceil \\
&= \sum_{j=p+1}^{i} \left\lceil \frac{d_i}{d_j} \right\rceil \qquad \text{as } T_i = d_i \text{ and } T_j = d_j \\
&= \sum_{j=p+1}^{i} \left\lceil \frac{2^{i-(p+1)}(2^{p+1}-1)}{2^{j-(p+1)}(2^{p+1}-1)} \right\rceil \\
&\quad \text{based on Formula 5.1 and } p+1 \le j \le i \le 2p+1 \\
&= \sum_{j=p+1}^{i} \left\lceil \frac{2^i}{2^j} \right\rceil \\
&= \sum_{j=p+1}^{i} \left\lceil 2^{i-j} \right\rceil \\
&= \sum_{j=p+1}^{i} 2^{i-j} \qquad i \text{ and } j \text{ are integers, so is } 2^{i-j} \text{ as } i \ge j \\
&= 2^{i-p} - 1
\end{aligned}$$

3. Calculate $X + Y$

$$\begin{aligned}
\sum_{j=1}^{i} \left\lceil \frac{T_i}{T_j} \right\rceil &= X + Y \\
&= T_i + 1 - 2^{i-p} + 2^{i-p} - 1 \\
&= T_i
\end{aligned}$$

■

Theorem 12

Given an even perfect number $n = 2^p(2^{p+1}-1)$, all of its distinct divisors are $d_i = 2^i$ for $0 \le i \le p$ and $d_i = 2^{i-(p+1)}(2^{p+1}-1)$ for $p+1 \le i \le 2p+1$ based on Formula 5.1. Derive a periodic task set $S_{2p+1} = \{\tau_i = (T_i, C_i) = (d_i, 1) : 1 \le i \le 2p+1\}$, then the periodic task set S_{2p+1} is schedulable by RMA with processor utilization 1. Furthermore, task set S_{2p+1} is semi-harmonic but not harmonic.

Proof: The proof is composed of three steps. The first step is proving $U = 1$ by applying

Lemma 11. The second step is proving that tasks $\tau_1, \tau_2, \ldots, \tau_i$ are harmonic for $1 \leq i \leq p$, which are schedulable by RMA as $L = U_i < U = 1$. The third step is proving tasks $\tau_1, \tau_2, \ldots, \tau_i$ for $p < i \leq 2p+1$ are schedulable through proving $L_i(T_i) = 1$ and $L \leq 1$ for $p < i \leq 2p+1$, which is based on 12; finally, proving $L \leq 1$ when $p < i \leq 2p+1$. Thus, the tasks $\tau_1, \tau_2, \ldots, \tau_i$ for $p < i \leq 2p+1$ are schedulable by RMA based on Theorem **ERMA2**. When $i = 2p+1$, the whole task set is schedulable by RMA and $U = 1$. At the beginning of each step and sub-step, the target of that step or sub-step is specified first, followed by the details.

1. $U = 1$
   $n = \sum_{i=0}^{2p} d_i$ as n is an even perfect number. Divide both sides by n, then

$$
\begin{aligned}
1 &= \sum_{i=0}^{2p} \frac{d_i}{n} \\
&= \sum_{i=0}^{2p} \frac{d_i}{d_i * d_{2p+1-i}} \\
&\quad \text{based on Lemma 11, } d_i * d_{2p+1-i} = n \text{ for } 0 \leq i \leq 2p+1 \\
&= \sum_{i=0}^{2p} \frac{1}{d_{2p+1-i}} \\
&= \frac{1}{d_{2p+1}} + \frac{1}{d_{2p}} + \frac{1}{d_{2p-1}} + \ldots + \frac{1}{d_1} \\
&= \sum_{i=1}^{2p+1} \frac{1}{d_i} \\
&= \sum_{i=1}^{2p+1} \frac{C_i}{T_i} \qquad \text{as } C_i = 1 \text{ and } T_i = d_i \\
&= U \qquad \text{based on Formula 2.1}
\end{aligned}
$$

2. $1 \leq i \leq p$, tasks $\tau_1, \tau_2, \ldots, \tau_i$ are schedulable by RMA.
   Tasks $\tau_1, \tau_2, \ldots, \tau_i$ are harmonic because $T_i = d_i = 2^i$, implying that all periods are equal to the power of 2 increasingly and $U_i < U = 1$. It is known that a harmonic task set is schedulable by RMA if its processor utilization is not greater than 1. Therefore, $\tau_1, \tau_2, \ldots, \tau_i$ are schedulable by RMA.

3. $p+1 \leq i \leq 2p+1$, tasks $\tau_1, \tau_2, \ldots, \tau_i$ are schedulable by RMA.

Consider tasks $\tau_1, \tau_2, \ldots, \tau_p, \tau_{p+1}, \ldots, \tau_i$. Based on step 2, tasks $\tau_1, \tau_2, \ldots, \tau_p$ are already schedulable with $U_p < U = 1$. Thus, it is unnecessary to consider the schedulability of tasks $\tau_1, \tau_2, \ldots, \tau_p$ when considering tasks $\tau_{p+1}, \ldots, \tau_{2p+1}$ as the latter ones have longer periods. Based on RMA, they have lower priorities. Thus, they cannot affect the schedulability of those tasks with shorter periods, which have higher priorities. When $p+1 \leq i \leq 2p+1$, $T_i = d_i = 2^{i-(p+1)}(2^{p+1} - 1)$ based on Formula 5.1.

(a) Calculate $L_i(T_i)$

$$\begin{aligned} L_i(T_i) &= \frac{\sum_{j=1}^{i} C_j \cdot \left\lceil \frac{T_i}{T_j} \right\rceil}{T_i} && \text{based on Formulae 2.4 and 2.3} \\ &= \frac{\sum_{j=1}^{i} \left\lceil \frac{T_i}{T_j} \right\rceil}{T_i} && \text{as } C_j = 1 \\ &= \frac{T_i}{T_i} && \text{based on Lemma 12} \\ &= 1 \end{aligned}$$

(b) Calculate L for tasks $\tau_1, \tau_2, \ldots, \tau_i$

$$\begin{aligned} L_i &= \min_{\{0<t\leq T_i\}} L_i(t) && \text{based on Formula 2.5} \\ &\leq L_i(T_i) \\ &= 1 && \text{by step 3a} \end{aligned}$$

$$\begin{aligned} L &= \max_{\{1\leq j\leq i\}} L_j && \text{based on Formula 2.8} \\ &\leq 1 && \text{by step 2 and 3b} \end{aligned}$$

Based on Theorem **ERMA2**, tasks $\tau_1, \tau_2, \ldots, \tau_i$ are schedulable by RMA.

Therefore, for $1 \leq i \leq 2p+1$, tasks τ_1, τ_2, ..., τ_i are schedulable by RMA. When $i = 2p+1$, task set S_{2p+1} is schedulable by RMA with $U = 1$.

Based on Formula 5.1, $T_p = 2^p$ and $T_{p+1} = 2^{p+1} - 1$ in S_{2p+1}. As $2^p \nmid (2^{p+1} - 1)$, then task set S_{2p+1} is not harmonic. Clearly, $n = T_{2p+1} = 2^p(2^{p+1} - 1)$ and all periods of tasks in S_{2p+1} are divisors of n, resulting in $T_i \mid T_{2p+1}$. Therefore, S_{2p+1} is semi-harmonic. ■

5.5 Summary

This chapter presents a real-time scheduling tool-kit developed during the research work of this thesis. The tool-kit was used to verify the correctness of the research work and find some interesting results. Though the tool-kit is an ad hoc application instead of a framework, it tries to support different schedulers based on the abstraction of a general real-time scheduler as the root in a hierarchy of schedulers. Currently, the tool-kit only supports independent task sets so resource management protocols are not supported. This deficiency is a weak point, as in practice, tasks often compete for exclusively-shared resources in many real-time systems. Furthermore, the tool-kit does not consider system architectures as another parameter. Both of these problems should be further studied.

Chapter 6

Conclusion and Future Work

6.1 Overview

This thesis focuses on two real-time scheduling algorithms. The first scheduling algorithm is FPPT; its schedulability test is proved robust under its critical instant. As well, this thesis explores the solution space when a task set with predefined regular priority is schedulable by FPPT with multiple valid assignments. A necessary condition is presented to describe the solution space based on a partial order relationship. Effective algorithms to compute minimal and maximal assignments are presented and proved correct. The second scheduling algorithm is RBA, in which the maximal size of the reservation bandwidth is calculated. This thesis provides theoretical support to calculate the maximal bandwidth effectively for *RBA_RMA* and *RBA_FPPT*. Finally, a small real-time scheduling tool-kit was developed to verify the correctness of these two algorithms and for use with other real-time scheduling algorithms.

6.2 Contributions

FPPT fills the gap between FPP and FPNP, which are two (typically) incomparable schedulers. This thesis reaffirms that FPPT is a valid form of real-time task scheduling and proves that the FPPT schedulability test is robust under its critical instant.

When a task set is schedulable by FPPT with predefined regular priorities, there may exist multiple valid assignments, which are delimited by the minimal and maximal assignments based on a partial order relationship. Some mechanisms are provided to generate additional valid assignments when two valid assignments satisfying some specific conditions are provided. Any

valid assignment must be located between the minimal and maximal assignments. However, not all assignments between the minimal and maximal assignments are valid.

The known algorithm to compute the minimal assignment starts its computation from FPP and the known algorithm to compute the maximal assignment starts its computation from a valid assignment. However, neither algorithm formally proves the result is minimal or maximal. This thesis provides formal proofs for both algorithms. In addition, algorithms to compute the minimal and maximal assignments starting from FPNP are presented. It is also shown that the minimal assignment can be calculated by starting from any valid assignment such as the maximal assignment. The correctness of these algorithms are also proven.

RBA is a kind of bandwidth reservation algorithm in which a periodic server executes during an evenly reserved bandwidth to shorten the response time of aperiodic tasks executed by the server. The known algorithm to calculate the reservation size for *RBA_RMA* is ineffective. This thesis proves that the maximal reservation size for *RBA_RMA* is equal to $1 - L$ where L is the minimal processor utilization required by the periodic task set when RMA is used. For *RBA_FPPT*, no direct formula is provided. However, an algorithm to search for the maximal assignment under which the maximal reservation size can be calculated is presented.

To verify the research results, one important approach is to write programs to simulate the examined real-time scheduling algorithms. Due to the difficulties in extending existing available tool-kits and the lack of support for FPPT, a small tool-kit was developed to verify the research work and to develop and verify two interesting results.

6.3 Benefits

As FPP and FPNP are both special cases of FPPT, FPPT has a larger range of schedulable task sets than either FPP and FPNP. When a task set is schedulable by FPPT, if the maximal assignment is used, it implies the number of preemptions is minimal, resulting in fewer context switches and less system cost. Under the maximal assignment, a resource management protocol may be unnecessary, resulting in better processor utilization. As well, FPPT has a static schedulability test.

RBA computes the maximal reservation size to service aperiodic tasks while guaranteeing the deadlines of periodic tasks. Though EDF can reserve all unused time left by periodic tasks to service aperiodic tasks evenly, its run-time cost is high due to its dynamic scheduling. When RMA is used to guarantee the deadlines of periodic tasks, an even reservation may be impossible.

When an even reservation is possible, the reservation size cannot be better than that of EDF. However, available resource management protocols based-on RMA can be applied when tasks are not independent. As well, RMA also has a static schedulability test. When FPPT is used and tasks are independent, it can obtain a reservation size not worse than RMA but remove the high run-time cost of EDF.

6.4 Constraints

This thesis does not consider all practical issues. For both scheduling algorithms examined, tasks are assumed to be independent during their execution, which is not always true in practice as tasks may communicate and access exclusively-shared resources.

While all bandwidth reservation algorithms may attempt to utilize any unused time, once that piece of time is smaller than a certain threshold, there is no practical benefit as no useful work can be accomplished by the aperiodic tasks. In fact, RBA increases practical overhead because both periodic and aperiodic tasks are executed in multiple short bursts requiring more context switches, although this can be mitigated by scaling.

6.5 Future Work

Future research work focuses on reducing the constraints. First, the regular priority of a task set is predefined. As each task has two priorities – regular priority and preemption threshold, it is reasonable to adjust both regular priority and preemption threshold for each task at the same time for a schedule search. When both priorities are adjusted, it is possible that the preemption threshold of a task may be lower than its regular priority. Second, given a task set with predefined regular priority, if a valid FPPT assignment does not make each task non-preemptive, then preemptions still exist. If tasks are not independent, then resource management protocols are required. It seems that the PIP protocol may work with FPPT. If PIP does not work, then new resource management protocols are required for FPPT. Third, the situation when the single uniform reserved bandwidth is too small should be considered in RBA. When a uniform reservation band cannot be generated across the entire major cycle, it may still be possible to have multiple larger reservation bands. Fourth, the possibility of creating a genetic scheduling algorithm [20] should be explored, i.e., only provide rules for tasks to compete for the processor and the rules for preemption. During the development of the tool-kit, different scheduling algorithms were implemented. Each algorithm required the same cycle to be repeated: analyzing the rules for a task

to compete for the processor and rules for preemption if allowed. Based on these rules, the worst case response time or expected processor utilization is computed. It seems that it is reasonable to create a genetic scheduling algorithm, so after a user chooses specific rules for a scheduling algorithm, the system can synthesize a scheduling algorithm based on the user's requirement. Of course, such a genetic scheduling algorithm may be implemented by generic programming techniques such as inheritance, template, and virtual function.

Bibliography

[1] P. Ancilotti, G. Buttazzo, M. Di Natale, and M. Bizzarri. A flexible tool kit for the development of real-time applications. In *IEEE Real-Time Technology and Applications Symposium*, pages 260–262, 1996.

[2] P. Ancilotti, G. Buttazzo, M. Di Natale, and M. Spuri. A development environment for hard real-time applications. *International Journal of Software Engineering and Knowledge Engineering*, 6(3):331–354, 1996.

[3] P. Ancilotti, G. Buttazzo, M. Di Natale, and M. Spuri. Design and programming tools for time critical applications. *Real-time Systems Journal*, 14:251–267, 1998.

[4] D. P. Anderson, R. G. Herrtwich, and C. Schaefer. SRP: A Resource Reservation Protocol for Guaranteed Performance Communication in the Internet. Technical Report TR–90–006, International Computer Science Institute, Berkeley, Feb 1990.

[5] B. Andersson, S. Baruah, and J. Jonsson. Static-priority scheduling on multiprocessors. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 193 – 202, Dec 2001.

[6] B. Andersson and J. Jonsson. Some insights on fixed-priority preemptive non-partitioned multiprocessor scheduling. In *Proceedings of the IEEE Real-Time Systems Symposium, Work-in-Progress Session*, pages 53–56, Nov 2000.

[7] N. C. Audsley. Deadline monotonic scheduling. Technical Report YCS 146, Department of Computer Science, University of York, Oct 1990.

[8] N. C. Audsley, A. Burns, M. Richardson, and K. Tindell. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, 1993.

[9] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Stress: a simulator for hard real-time systems. *Software-Practice and Experience*, 24(6):543–564, Jun 1994.

[10] N. C. Audsley, A. Burns, and A. J. Wellings. Deadline monotonic scheduling theory and application. *J. Control Engineering Practice*, pages 71–78, 1993.

[11] T. P. Baker. A stack-based resource allocation policy for realtime processes. In *Real-Time Systems Symposium*, pages 191–200. IEEE Computer Society Press, 1990.

[12] I. Bate and A. Burns. Schedulability analysis of fixed priority real-time systems with offsets. In *Proceedings of the Ninth Euromicro Workshop on Real-Time Systems*, pages 153–160, Jun 1997.

[13] E. Bini and G. C. Buttazzo. The space of rate monotonic schedulability. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium*, pages 169–178, 2002.

[14] N. S. Bowen, C. N. Nikolaou, and A. Ghafoor. Hierarchical workload allocation for distributed systems. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 102–109, 1988.

[15] A. Burns and A. Wellings. *Real-Time System and Programming Languages*, chapter 13, pages 407–418. ADDISON-WESLEY, second edition, 1997.

[16] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages*, chapter 1, pages 4–14. ADDISON-WESLEY, 1997.

[17] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages*, chapter 14, page 482. ADDISON-WESLEY, 1997.

[18] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages*, chapter 16, pages 547–551. ADDISON-WESLEY, 1997.

[19] D. Chen, A. K. Mok, and T. Kuo. Utilization bound re-visited. In *Proceedings of the Sixth International Conference on Real-Time Computing Systems and Applications*, pages 295–302, 1999.

[20] G. Coates, A.H.B. Duffy, R.I. Whitfield, W. Hills, and P. Sen. Real-time co-ordinated scheduling using a genetic algorithm. In *Proceedings of the 5th International Conference on Adaptive Computing in Design and Manufacture (ACDM '02)*, pages 381–392, April 2002.

[21] M. Cumming, B. Rieder, J. M'Sadoques, O. Laursen, G. Ruebsamen, and C. Gustin. *Programming with gtkmm2*. http://www.gtkmm.org, 2002.

[22] R. Davis, N. Merriam, and N. Tracey. How embedded applications using an RTOS can stay within on-chip memory limits. In *Proceedings for the Work in Progress and Industrial Experience Sessions, 12th EuroMicro Conference on Real-Time Systems*, pages 43–50, Jun 2000.

[23] R. Devillers and J. Goossens. Liu and layland's schedulability test revisited. *Information Processing Letters*, 73(5-6):157–161, Mar 2000.

[24] B. Ford and S. Susarla. CPU inheritance scheduling. In *Usenix Association Second Symposium on Operating Systems Design and Implementation (OSDI)*, pages 91–105, 1996.

[25] M. R. Garey and D. S. Johnson. Two-processor scheduling with start-times and deadlines. *sicomp*, 6(3):416–426, Sep 1977.

[26] L. George, N. Rivierre, and M. Spuri. Preemptive and non-preemptive real-time uniprocessor scheduling. Technical Report N° 2966, Institut National de Recherche en Informatique et en Automatique, Sep 1996.

[27] J. B. Goodenough and L. Sha. The priority ceiling protocol: A method for minimizing the blocking of high priority ada tasks. *ACM SIGAda Ada Letters(Special edition: International Workshop on Real-Time Ada Issues)*, VIII(7):20 – 31, Jun 1988.

[28] C.-C. Han. A better polynomial-time schedulability test for real-time multiframe tasks. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 104–113, Dec 1998.

[29] C.-C. Han and H.-Y Tyan. A better polynomial-time schedulability test for real-time fixed-priority scheduling algorithms. In *Proceedings of the 18th IEEE Real-Time Systems Symposium*, pages 36–45, Dec 1997.

[30] http://www.gtkmm.org. *API Reference for gtkmm 2.4*, 2004.

[31] K. Jeffay. Scheduling sporadic tasks with shared resources in hard-real-time systems. In *Proceedings of the 13th IEEE Real-Time Systems Symposium*, pages 89–99, Dec 1992.

[32] J. Jonsson. *The Impact of Application and Architecture Properties on Real-Time Multiprocessor Scheduling*. PhD thesis, CTH, Dept. of Computer Engineering, Computer Architecture Laboratory (CAL), MicroMultiProcessor Group (MMP), 1997.

[33] M. Joseph and P. Pandya. Finding response times in a real-time system. *The Computer Journal*, 29(5):390–395, 1986.

[34] M. H. Klein, T. Ralya, B. Pollak, and R. Obenza. *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*, chapter 7, pages 15–18. Kluwer Academic Publishers, 1 edition, Nov 1993.

[35] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Benjamin/Cummings, Redwood City, CA, 1994.

[36] S. Kweon and K. G. Shin. Providing deterministic delay guarantees in ATM networks. *IEEE/ACM Transactions on Networking*, 6(6):838–850, Dec 1998.

[37] W. Lamie. Preemption-threshold. Express Logic, Inc. White Paper. Available at http://www.expresslogic.com/wppreemption.html, 1997.

[38] S. Lauzac, R. Melhem, and D. Mossé. An efficient RMS admission control and its application to multiprocessor scheduling. In *Parallel Processing Symposium*, pages 511–518, 1998.

[39] J. P. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *Proceedings of the 11th Real-Time Systems Symposium*, pages 201–209, Dec 1990.

[40] J. P. Lehoczky and S. Ramos-Thuel. An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority preemptive systems. In *IEEE Real-Time Systems Symposium*, pages 110–123, 1992.

[41] J. P. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *IEEE Real-Time Systems Symposium*, pages 166–171, Dec 1989.

[42] J. P. Lehoczky, L. Sha, and J. K. Strosnider. Enhanced aperiodic responsiveness in hard real-time environments. In *IEEE Real-Time Systems Symposium*, pages 261–270, 1987.

[43] J. Y-T Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2:237–250, 1982.

[44] T. G. Lewis. *Foundations of Parallel Programming: A Machine-Independent Approach*. IEEE Computer Society Press, 1994.

[45] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, Jan 1973.

[46] J. W. S. Liu. *Real-Time Systems*, chapter 7, pages 190–195. Alan R. Apt, 2000.

[47] J. M. López, J. L. Díaz, and D. F. García. Minimum and maximum utilization bounds for multiprocessor RM scheduling. In *Proceedings of the 13th Euromicro Conference on Real-Time Systems*, pages 67–75, 2001.

[48] C. Lord. Analysis of embedded real-time system, final project report. Technical report, Carnegie Mellon University, 2001.

[49] Y. Manabe and S. Aoyagi. A feasibility decision algorithm for rate monotonic and deadline monotonic scheduling. *Real-Time Systems*, 14(2):171–181, Mar 1998.

[50] C. W. Mercer. An introduction to real-time operating systems: Scheduling theory. Draft, Nov 1992.

[51] A. K. Mok. Tracking real-time systems requirements. Technical report, University of Texas at Austin, Austin, Texas, 78712, USA, 1997.

[52] B. Mukherjee, K. Schwan, and K. Ghosh. A survey of real-time operating systems – preliminary draft. Technical Report GIT-CC-93-18, College of Computing, Georgia Institute of Technology, 1993.

[53] M. Naedele. A survey of real-time scheduling tools. Technical Report 72, ETH Zurich, Computer Engineering and Networks Laboratory, 1998.

[54] G. J. Narlikar and G. E. Blelloch. Pthreads for dynamic and irregular parallelism. In *Proc. SC98: High Performance Networking and Computing*, Orlando, FL, Nov 1998. IEEE.

[55] D. Oh and T. P. Bakker. Utilization bounds for n-processor rate monotone scheduling with static processor assignment. *Real-Time Systems*, 15(2):183–192, Sep 1998.

[56] D. Park, S. Natarajan, A. Kanevsky, and M. J. Kim. A generalized utilization bound test for fixed-priority real-time scheduling. In *Proceedings of the Second International Workshop on Real-Time Computing Systems and Applications*, pages 73–77, Oct 1995.

[57] T. M. Parks and E. A. Lee. Non-preemptive real-time scheduling of dataflow systems. In *Proceedings of International Conference on Acoustics, Speech, and Signal Processing*, volume 5, pages 3235–3238, May 1995.

[58] R. Rajkumar. *Synchronization in Real-Time Systemes A Priority Inheritance Approach*, chapter 2, page 15. Kluwer Academic Publishers, 101 Philip Drive, Assinppi Park, Norwell, Massachusetts 02061 USA, 1991.

[59] J. Regehr. Scheduling tasks with mixed preemption relations for robustness to timing faults. In *23rd IEEE Real-Time Systems Symposium, 2002*, pages 315–326, Dec 2002.

[60] M. Saksena and Y. Wang. Scalable real-time system design using preemption thresholds. In *Proceedings of the 21st IEEE Real-Time Systems Symposium*, pages 25–34, 2000.

[61] L. Sha, R. Rajkumar, and C.-H. Chang. A real-time locking protocol. *IEEE Transactions on Computers*, 40(7):793–800, Jul 1991.

[62] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: an approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, Sep 1990.

[63] K. G. Shin and Y. Chang. A reservation-based algorithm for scheduling both periodic and aperiodic real-time tasks. *IEEE Transactions on Computers*, 44(12):1405–1419, Dec 1995.

[64] B. Sprunt. *Aperiodic Task Scheduling for Real-time Systems*. PhD thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University, Aug 1989.

[65] M. A. Squadrito. Extending the priority ceiling protocol using read/write affected sets. Master's thesis, Department of Computer Science, University of Rhode Island, 1996.

[66] J. A. Stankovic. Misconceptions about real-time computing. *IEEE Computer*, 21(10):10–19, Oct 1988.

[67] J.A. Stankovic, M. Spuri, K. Ramamritham, and G.C. Buttazzo. *Deadline Scheduling for Real-Time Systems EDF and Related Algorithms*, chapter 3, page 51. Kluwer Academic Publishers, 1998.

[68] M. F. Storch. *A Framework for the Simulation of Complex Real-Time Systems*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1997.

[69] K. Tindell, A. Burns, and A. Wellings. An extendible approach for analysing fixed priority hard real-time tasks. *Real-Time Systems*, 6(2):133–152, Mar 1994.

[70] H. M. Tjandrasa, S. Davari, and T. Leibfried. An exact execution time test for fixed priority real -time tasks. In *Work-In-Progress Sessions of The 21st IEEE Real-Time Systems Symposium (RTSSWIP00)*, Nov 2000.

[71] J. Ullman. NP-Complete scheduling problems. *Journal Comp. Syst. Sciences*, 10:384–393, 1975.

[72] A. M. van T. and G. M. Koob, editors. *Foundations of real-time computing: scheduling and resource management*, chapter 1: Fixed Priority Scheduling Theory for Hard Real-Time Systems by J. P. Lehoczky, Liu Sha, J.K. Strosnider and Hide Tokuda, page 8. Kluwer Academic Publishers, Boston, 1991.

[73] S. Wang and G. Färber. On the schedulability analysis for distributed real-time systems. In *Proceedings of Joint 24th IFAC/IFIP Workshop on Real Time Programming and Third International Workshop on Active and Real-Time Database Systems (WRTP'99, ARTDB'99)*, 1999.

[74] Y. Wang and M. Saksena. Scheduling fixed-priority tasks with preemption threshold. In *Proceedings of the Sixth International Conference on Real-Time Computing Systems and Applications*, pages 328–335, 1999.

[75] M. Wu and W. Shu. High-performance incremental scheduling on massively parallel computers — A global approach. In *Supercomputing '95. Proceedings*, pages 1501–1527, 1995.

[76] S. Y. Yan. *Perfect, amicable, and sociable numbers : a computational approach*, chapter 1, pages 10–17. World Scientific Publishing Co. Pte. Ltd., 1996.