Semi-Fixed-Priority Scheduling

A Dissertation Presented
by
Hiroyuki Chishiro

Submitted to
the School of Science for Open and Environmental Systems
in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

at Keio University

March 2012

Acknowledgments

First of all, I would like to express my sincere gratitude to my advisor Prof. Nobuyuki Yamasaki. He has supported my research since I was a bachelor student. I am thankful to Prof. Hiroki Matsutani for his detailed reading. I am proud of obtaining the first Ph.D. in Yamasaki and Matsutani Laboratory. I would like to give my sincere gratitude to Prof. Fumio Teraoka, Prof. Kenji Kono and Prof. Takahiro Yakoh for serving on the dissertation committee. Though they are busy, they have dedicated their time and effort beyond their duty.

I sincerely appreciate the technical support from Dr. Kenji Funaoka and Akira Takeda. Without their support, this dissertation could not have been completed. My special thanks go to Dr. Hidenori Kobayashi for completing this dissertation.

I am deeply grateful to Dai Yamanaka who invited me to the IRL track and field club. Thanks to his invitation, I have enjoyed my runner's life. I would like to thank Toshio Utsunomiya for his friendship and encouragement since I was a high school student.

Finally, I would like to express my heartfelt thanks to my parents for encouraging my life.

Abstract

Semi-Fixed-Priority Scheduling

Hiroyuki Chishiro

Real-time systems have been encountering overloaded conditions in dynamic environments. In order to perform real-time scheduling in such overloaded conditions, an imprecise computation model, which improves the quality of result with timing constraints, has been proposed. In the imprecise computation model, dynamic-priority scheduling algorithms cause high-jitter of the shortest period task. Unfortunately, no fixed-priority scheduling algorithm with low-jitter of the shortest period task can be adapted to the imprecise computation model due to the overrun of the non-critical part.

This dissertation first proposes a concept of semi-fixed-priority scheduling to achieve both low-jitter and high-schedulability. Semi-fixed-priority scheduling schedules the part of each imprecise task by fixed-priority. This dissertation also proposes a novel semi-fixed-priority scheduling algorithm based on the Rate Monotonic (RM) algorithm, called *Rate Monotonic with Wind-up Part (RMWP)*. The schedulability analysis proves that a task set is schedulable by the RMWP algorithm if the task set is schedulable by the RM algorithm. In addition, this dissertation extends the RMWP algorithm for global and partitioned scheduling algorithms on multiprocessors.

This dissertation next presents a real-time operating system for semi-fixed-priority scheduling algorithms, called *RT-Est*. The RT-Est real-time operating system implements two queueing policies for semi-fixed-priority scheduling algorithms, called *hybrid scheduler* and *dual scheduler*. The hybrid scheduler is implemented to achieve semi-fixed-priority scheduling algorithms with low overhead. The dual scheduler is an extension of the hybrid scheduler for global scheduling.

The effectiveness of semi-fixed-priority scheduling is confirmed through both simulation studies and experimental evaluations. This dissertation concludes that semi-fixed-priority scheduling contributes both theory and practice to practical imprecise computation.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

There are many computers in our world. We use many computers in homes, schools, stations, factories and laboratories. Without computers, we cannot lead our lives. There are many types of computers including laptop computers, server machines, mobile phones and robots. Laptop computers and server machines are categorized as general purpose systems, which operate in cyber world. In contrast, mobile phones and robots are categorized as embedded systems. More than 79% of manufactured processors were used for embedded systems in 2006 [1].

Most of embedded systems are usually as same as real-time systems. The primary reason why embedded systems are real-time systems is because they operate in real world. When an event occurs in real world, embedded systems must react to perform proper operations.

Real-time systems are technically characterized by the fact that they require temporal correctness as well as logical correctness. In other words, the correctness of the real-time system depends on both the logical correctness of results and the time when they are produced. More precisely, every real-time computation must complete its execution in an interval of certain length. The beginning of the interval is called *release time* and the end of that is called *deadline*. The deadline in real-time systems has mainly three types depending on what could happen if a deadline miss occurs.

- If the contribution to the system suddenly drops to negative, then the deadline is *hard*.
- If the contribution to the system suddenly drops to 0, then the deadline is *firm*.
- If the contribution to the system does not become 0 suddenly and degrades gradually as the completion time is further delayed, then the deadline is *soft*.

The value of each computation is shown in Figure 1.1.

For example, robots have hard deadlines for their control because a catastrophe may occur if their actuators cannot complete processing sensor data within a certain feedback period.

Multimedia systems often have soft deadlines for image and audio processing, because multimedia systems are able to continue to provide a service to users even if some computations miss their deadlines, though the quality of service may be degraded compared to the case in which all computations meet their deadlines. In order to maintain substantial quality, deadlines should not be missed even in soft real-time systems.

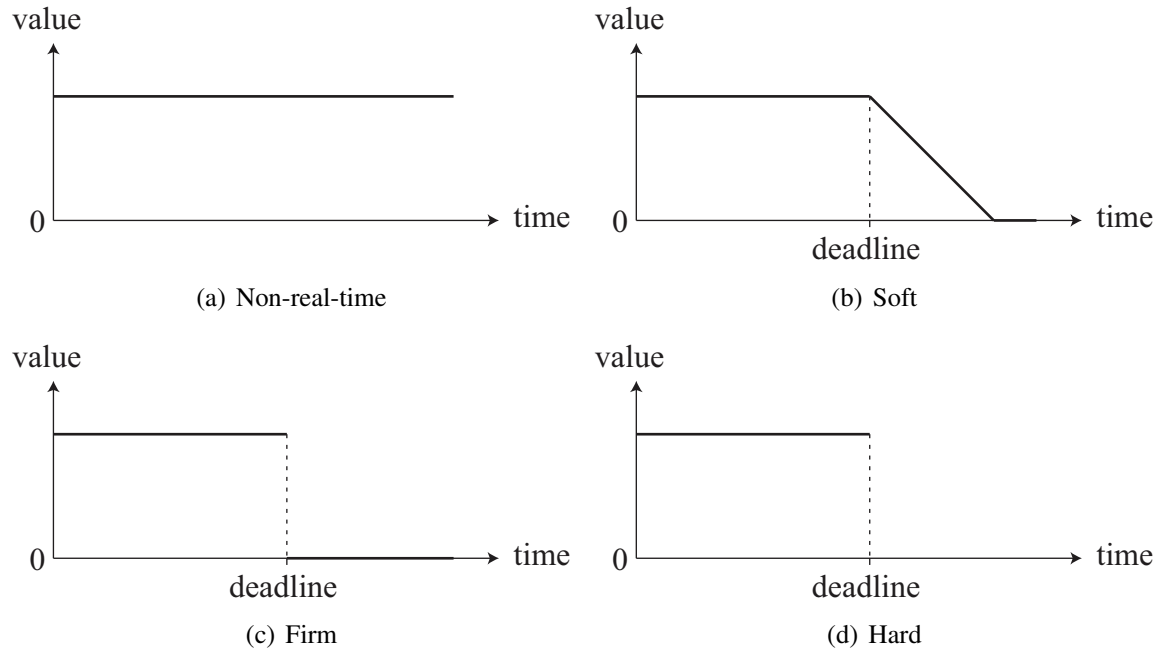


Figure 1.1: Value of each computation

Other multimedia systems including ATM networking systems have firm deadlines because they have included several applications with similar combinations of fine-grained timing requirements (typical of hard deadline) with system service requirements (typical of soft deadline).

Real-time systems require reserving processor time to meet all computations called tasks by their deadlines. The primary solution is the technique to execute tasks in the proper order, called real-time scheduling. The contribution of real-time scheduling is to guarantee completing real-time tasks by their deadlines theoretically.

The history of real-time scheduling has started by Liu and Layland [2] since 1973. They presented the traditional task model called Liu and Layland's model and proposed two representative real-time scheduling algorithms, called Rate Monotonic (RM) and Earliest Deadline First (EDF).

The RM algorithm is a basic fixed-priority scheduling algorithm in real-time scheduling. In the RM algorithm, tasks with shorter periods have higher priorities. Since periods are constant, the RM algorithm is a fixed-priority assignment: priorities are assigned to tasks before execution and do not change over time. They prove that any independent periodic task set can be scheduled without deadline miss by the RM algorithm on uniprocessors if the total processor utilization of the independent periodic task set is lower than or equal to the following equation.

$$n(2^{1/n} - 1),$$

where n is the number of tasks.

In contrast, the EDF algorithm is a basic dynamic-priority scheduling algorithm in real-time scheduling. In the EDF algorithm, tasks with earlier absolute deadlines are executed at higher priorities. Therefore, each job, which is an instance of a task, is generated periodically every release time and has different priority, unlike the RM algorithm. They also prove

that any independent periodic task set can be scheduled without deadline miss by the EDF algorithm on uniprocessors if the total processor utilization of the independent periodic task set is lower than or equal to 1, regardless of the number of tasks.

Ever since these two algorithms are presented, there have been a considerable number of arguments on which algorithm performs better in what conditions, some of which are summarized by Buttazzo [3]. Buttazzo claims that the real advantage of the RM algorithm with respect to the EDF algorithm is its simpler implementation in commercial kernels that do not provide explicit support for timing constraints, such as periods and deadlines. In addition, real-time scheduling should support dynamic real-time environments because we use embedded devices everywhere. Therefore, real-time systems require a higher possibility of being in overloaded conditions.

1.2 Overload in Real-Time Systems

In order to define what an overloaded condition is, the feasibility of a task set must be defined. If a task set is feasible in a real-time scheduling algorithm, it means that no task in the task set violates its timing constraints under any circumstance [4].

The feasibility of a task set based on Liu and Layland's model in the EDF algorithm on uniprocessors can be assessed by simply checking the processor utilization. In the EDF algorithm, the task set is feasible on uniprocessors if the total processor utilization is lower than or equal to 1. Therefore, when the EDF algorithm is used, an overloaded condition is said to occur when the processor utilization is higher than 1. In a more general context, an overloaded condition is said to occur in a system when no task set is schedulable by an optimal scheduling algorithm.

In modern real-time systems, an overloaded condition is much harder to handle, because workloads have become dynamic and the state of the overload is exposed only at run-time. The background of this change is derived from the transition of real-time systems. One of the real-time systems that have typically gone through this transition is a robot system. The robot systems in their early days were used in limited environments that raised at a fixed rate beforehand. For example, the majority of robots were once developed to work in assembly lines of production factories. Since these robots only had to do some routine work, most of their workloads were static. On the other hand, robots have appeared in much wider areas lately. In fact, it has not become very uncommon to see an autonomous mobile robot working in the same environment that we live. Moreover, robots have been sold even in a department store as pets and helpers to people; for example, Roomba [5]. These robots that aim to work in various environments must handle various kinds of events in various dynamic environments. Thus, their workloads have become much more dynamic.

There are two important factors, which afflict the predictability of workloads in dynamic real-time systems.

First of all, the worst case execution time of an application is hard to analyze, since complex hardware components in modern real-time systems decrease the accuracy of the worst case execution time analysis. High performance processors that can meet the requirement of dynamic real-time systems adapt recent complex techniques, such as caches, translation lookaside buffers, speculation and multithreading. These techniques can improve the average case performance at a larger cost in the worst case, leading to a wider gap between the average case execution time and the worst case execution time. Since the worst case rarely happens in practice, the accuracy of the worst case execution time can be either too

pessimistic or unsafe to use in feasibility assessment in practice.

Next, the worst case scenario is hard to identify. The dynamic environments make it almost impossible to make correct assumptions on the incoming events. In addition, even when it is possible, determining every impact of external factors on all scenarios needs extensive testing and simulations. Considering the tight time-to-market schedules in the development of embedded systems, it is not a practical approach. It should be also noted that the commercial-off-the-shelf components used to be shorten the development period could ironically encumber the correct identification of the worst case scenario.

If a real-time system must not fall into an overloaded condition at all, the only possible approach for handling the dynamic workloads would be to continue using the classical theory and reserve enough resources based on the pessimistic worst case execution time on a processor with sufficiently high capability so that the system never becomes overloaded. Unfortunately, no matter how tight the worst case execution time is bounded, this reservation approach is only effective when the degree of variation in the workloads is small. Otherwise, the amount of reserved but unused resources becomes significantly large that more expensive processors with higher processing capability must be used. Moreover, in a dynamic real-time system with computation intensive workloads, it is also possible that there exists no processor that can provide the required performance.

If a real-time system can also work in an overloaded condition, other existing approaches can be used to sustain the system performance to an acceptable level under overload. These approaches can be categorized by the types of the deadline. If some computations have soft deadlines, a transient overload can be resolved by delaying their execution. Gardner and Liu [6] described and evaluated scheduling algorithms that isolate the effect of overruns caused by a computation, the worst case execution time of which is not safe. Buttazzo and Stankovic [7] presented the Robust Earliest Deadline (RED) algorithm. In the RED algorithm, a deadline tolerance of a computation is defined and used in an overloaded condition to check whether each computation can still contribute to the system when its completion must be delayed. If the computation cannot contribute to the system, the algorithm uses important values attached to computations to determine which computation to reject. Buttazzo et al. [8] gave a comparative study on such robust scheduling algorithms. Baruah and Haritsa [9] developed an asymptotically optimal scheduling algorithm called Resistance to Overload By Using Slack Time (ROBUST) that sustains the same level of effective processor utilization both in normal and overloaded conditions. An unfortunate result is that the ROBUST algorithm needs a processor that is twice as fast as the one used in a system without any possibility of undergoing overload.

1.3 Multiprocessor Real-Time Systems

One of the solutions to overcome overloaded conditions is to make use of multiprocessors. Multiprocessors can strongly reserve the spare capacity for tolerating overloaded conditions. Multiprocessor real-time systems require multiprocessor real-time scheduling so that many real-time scheduling algorithms on uniprocessors are extended for multiprocessors. Before the exposition of the problem of real-time scheduling on multiprocessors, this dissertation explains the reason why multiprocessors gather worldwide attention in recent years. The performance of uniprocessors has been improved by shrinking transistor sizes, refining processor architectures and raising processor clock frequencies. The performance improvement of uniprocessors comes from the advancement of Instruction-Level Parallelism (ILP). The

improvement of ILP is difficult due to the interference, which comes from physical factors and the depletion of effective ideas. Raising processor clock frequencies is also difficult due to the overheating problem. The hardware vendors must scope out other sales points against traditional uniprocessors. Therefore, multiprocessors have been widely used in real-time systems. Examples of multiprocessors are Intel's Xeon [10], Sony-IBM-Toshiba's Cell [11] and ARM's MPCore [12].

Multiprocessors also improve the quality of service efficiently to make use of the spare capacity. For example, smartphones including iPhone [13] and Android [14] are used everywhere and we would like to generate the response of the operation at proper timing. If the response time of such operation is longer and longer, we may chafe at smartphones. However, it is also a problem that the response of such operation is too fast to understand what smartphones do. The timing constraints of such actions between the time when we input something and the time when we output operations are important to use smartphones with comfort. The other example is robots including humanoid robots [15], mobile robots [16] and animal robots [17]. These robots require multiprocessors because they detect and avoid objects by image processing with high load. The processing time of the image processing task depends on the dynamic real-time environments so that they always encounter the overloaded conditions. The processing with sensitive timing constraint in autonomous mobile robots is to control actuators. If the interval of getting the information from sensors periodically in dynamic real-time environments is too fluctuated, autonomous mobile robots may crash into objects due to not performing operations with proper timing. The fluctuation of the interval between the previous input operation and the post input operation periodically requires being low as much as possible without timing violations, which is called *jitter*.

1.4 Imprecise Computation

Liu and Layland's model has the dilemma that became apparent in the development of complex real-time systems, the workloads of which vary dynamically and may cause overload only at run-time. The two conflicting demands behind the dilemma are as follows. First of all, developers would like to statically reserve resources as much as possible so that all timing constraints are met even in the worst case. Next, they also would like to leave the same resources unreserved as much as possible so that only a small portion of resources is wasted as spare capacity in the other cases. Stated differently, designing real-time systems for high processor utilization will risk the temporal correctness, whereas designing the systems for perfect temporal correctness will increase the cost or may be impossible. Therefore, there is a crucial dilemma of meeting the timing constraints and the high processor utilization at the same time.

Solving this problematic dilemma requires establishing a good balancing point that guarantees timing constraints of at least important computations and allows dynamic sharing among other computations for efficient use of resources, for example, the quality of service. Thus, finding a solution to the problem enables the system to adjust effective system load to a level dynamically where no critical constraint is violated. The ability to adjust the system load is important especially in real-time systems, since they do not allow manual adjustment of workloads once the system is put to operations. In other words, some forms of automatic adjustment mechanism must be adapted if the system may fall into overloaded conditions.

In order to establish such a good balancing point, this dissertation needs to identify critical and non-critical parts of the system. In other words, this dissertation needs to dis-

criminate parts that must be executed strictly as requested from what does not need to discriminate those. In this research, the discrimination is made within computations as well as among computations. The discrimination among computations means distinguishing soft deadlines from hard deadlines. On the other hand, the discrimination within a computation is accomplished by using the notion of the traditional imprecise computation model [18].

The traditional imprecise computation model aims to obtain the best result with all available resources, exactly by distinguishing a part of a computation that must be always completed from a part that does not have to be completed. The adjustment of load by the traditional imprecise computation model is carried out by terminating computations prematurely to discard non-critical parts. A prematurely terminated computation can only return a partially correct, called *imprecise result*. However, it is often desirable to receive an imprecise result before deadline than to receive a precise result later.

Many real-time applications can be implemented based on the traditional imprecise computation model actually. For example, a wavelet transform used in data compression and signal processing is well suited to imprecise computation. Compression and decompression of images by using the wavelet transform can directly trade-off the quality of images with the processing time. Moreover, the wavelet transform can be also used indirectly to find a balancing point for the trade-off. The data transformed in different resolutions can be used to implement multiple versions with different processing time. For example, in a template matching application, multiple templates can be stored in different resolutions and be chosen dynamically depending on how high the system load is.

Some other examples among diverse applications are scalable multimedia processing and transmission [19, 20, 21, 22], network traffic management [23, 24], decision making under uncertainty [25], anytime learning in evolutionary robotics [26, 27], motion planning and robot control [28, 29], multi-target tracking [30], transactions in real-time databases [31] and fuzzy systems [32]. Unfortunately, despite thus many applications, the traditional imprecise computation model has not been used widely in industry yet. One of the reasons is that the traditional imprecise computation model assumes an impractical assumption, in which the processing time of termination or completion of a non-critical part is 0. However, the processing to terminate or complete a non-critical part is guaranteed by its deadline. For example, the result of object detection by image processing tasks based on imprecise computation must output the actuator by its deadline in robots. Therefore, the traditional imprecise computation model cannot support practical imprecise applications.

1.5 Motivation

This research was motivated by the impracticality of the traditional imprecise computation model. In order to overcome the weakness of the traditional imprecise computation model, Kobayashi and Yamasaki proposed a practical imprecise computation model [33]. This model redefines the practicality and applicability of imprecise computation by allowing more than one critical parts and also more than one non-critical parts in a computation. Moreover, these critical and non-critical parts can be freely interleaved. This model bridges the gap between the existing theoretical models of imprecise computation and the practical characteristics of real world applications. They also proposed an EDF-based dynamic-priority scheduling algorithm in the practical imprecise computation model on uniprocessors, called Mandatory-First with Wind-up Part (M-FWP) [34, 35]. The M-FWP algorithm first supports the practical imprecise computation model so that the contribution of the M-

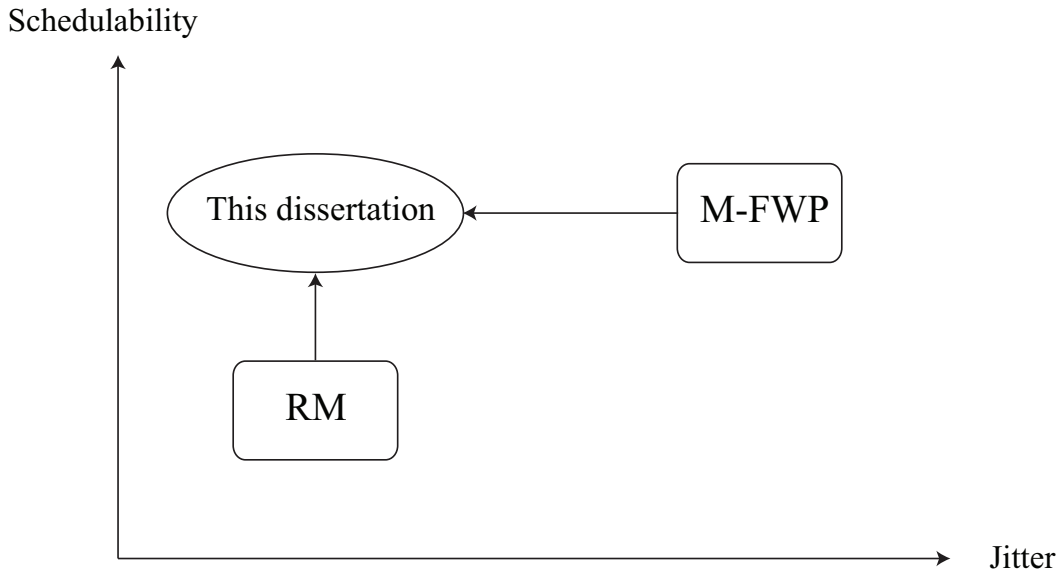


Figure 1.2: Area of this dissertation in real-time scheduling

Table 1.1: Area of this dissertation in system model

Algorithm	Uniprocessor	Multiprocessor	Imprecise
RM [2]	✓	✓	
M-FWP [34, 35]	✓		✓
This dissertation	✓	✓	✓

FWP algorithm is high. However, the M-FWP algorithm is difficult to support multiprocessors. Because the M-FWP algorithm calculates the assignable time of non-critical parts dynamically, the M-FWP algorithm is not practical. In addition, the dynamic calculation of the assignable time of the non-critical parts causes high-jitter of the shortest period task, which usually controls an actuator with jitter-sensitive in autonomous mobile robots.

On the other hand, the RM algorithm supports multiprocessors and achieves low-jitter against the M-FWP algorithm. Unfortunately, the RM algorithm cannot be adapted to the practical imprecise computation model. Because the practical imprecise computation model has more than one non-critical parts which require unknown processor time so that the overrun of the non-critical parts causes the deadline miss of the critical parts in the overloaded conditions. In order to support the practical imprecise computation model on multiprocessors, a new priority assignment policy is required.

Figure 1.2 shows the area of this dissertation in real-time scheduling. The goal of this dissertation in real-time scheduling is to achieve higher schedulability than the RM algorithm and lower jitter than the M-FWP algorithm. If the jitter is lower and lower, real-time applications such as robots can achieve the more precise control.

Table 1.1 shows the area of this dissertation in the system model. The goal of this dissertation in the system model is to support uniprocessors and multiprocessors for practical imprecise computation.

1.6 Research Overview and Contributions

The goal of this research is to support imprecise computation and to improve the quality of service on multiprocessors. In order to achieve this goal, this research considers real-time scheduling in the practical imprecise computation model on both uniprocessors and multiprocessors. The research overview of this dissertation is as follows.

> A new priority assignment policy for practical imprecise computation aspires for both theoretical and practical frontier.

In order to support this research, this dissertation makes the following contributions.

- A new priority assignment policy for practical imprecise computation is proposed to achieve low-jitter and high schedulability. This policy, called *semi-fixed-priority scheduling*, can make use of the techniques of the schedulability analysis for fixed-priority scheduling.

- The proposed algorithms achieve both uniprocessor and multiprocessor real-time scheduling in the practical imprecise computation model. The schedulability analysis shows that semi-fixed-priority scheduling is at least as effective as fixed-priority scheduling.

- A real-time operating system is developed to achieve semi-fixed-priority scheduling with low overhead in the practical imprecise computation model from scratch. Especially, the real-time operating system provides the presented schedulers for semi-fixed-priority scheduling algorithms. The presented schedulers can implement semi-fixed-priority scheduling with reasonable overhead.

- The effectiveness of semi-fixed-priority scheduling is confirmed through both simulation studies and experimental evaluations on uniprocessors and multiprocessors. Semi-fixed-priority scheduling contributes both theory and practice to practical imprecise computation from simulation and experimental results.

The advantage of this approach against other approaches is to try to remove the uncertainty completely, for example, providing a method to analyze tighter worst case execution time. This approach does not depend on a proprietary processor that does not require a rigid analysis of application programs, thereby allowing use of software components, execution paths of which are only determined at run-time.

Finally, the scope of this research is limited to providing a systematic approach for scheduling practical imprecise computations. Especially, it focuses on the practical imprecise computation model, real-time scheduling algorithms and implementation that can be provided by a real-time operating system. The presented approach is targeted to both uniprocessor and multiprocessor systems. A system with more than one processing units must be structured as a group of independent subsystems, each with one processing unit, in order to utilize the result of this research. Moreover, it is not the scope of this research to provide hardware, language and compiler support for imprecise computation. In particular, this research does not focus on extracting appropriate time attributes such as the worst case execution time from programs or developing a method that facilitates its precise analysis.

1.7 Organization

The rest of this dissertation is organized as follows. Chapter 2 summarizes the state of the art techniques related to traditional computation models, multiprocessor real-time scheduling policies, real-time scheduling algorithms and real-time operating systems. Chapter 3 formally defines the practical imprecise computation model. A linear task model is also introduced to theoretically represent applications based on this computation model. Chapter 4 presents a new priority assignment policy for practical imprecise computation, called *semi-fixed-priority scheduling*. In addition, semi-fixed-priority scheduling algorithms for uniprocessor scheduling, multiprocessor partitioned scheduling and multiprocessor global scheduling are proposed. Chapter 5 presents a real-time operating system for semi-fixed-priority scheduling algorithms. The real-time operating system implements the proposed algorithms with low overhead. Chapter 6 evaluates the performance of the proposed algorithms through simulation studies, compared to other algorithms. Chapter 7 evaluates the practicality of the proposed algorithms through experimental evaluations. Chapter 8 concludes this dissertation and suggests the future directions of this research.

Chapter 2

State of the Art

This chapter provides a survey on the state of the art techniques for traditional computation models, multiprocessor real-time scheduling policies, real-time scheduling algorithms and real-time operating systems. First, the survey explains existing task models. Next, multiprocessor real-time scheduling policies and real-time scheduling algorithms in these task models are summarized. In addition, existing real-time operating systems implementing these real-time scheduling algorithms are introduced. The survey ends with summarizing the experimental evaluations of multiprocessor real-time scheduling algorithms.

2.1 Traditional Computation Models

2.1.1 Liu and Layland's Model

Liu and Layland's model is a basic computation model in real-time scheduling [2]. Figure 2.1 shows an example of task execution in Liu and Layland's model. This model assumes that a system has a task set Γ consisted of n periodic tasks τ_i $\{i = 1, 2, ..., n\}$. Each task τ_i is released periodically at every period T_i and consumes its worst case execution time m_i as a mandatory part between the interval. The utilization of task τ_i is $U_i = m_i/T_i$. The system utilization of all tasks is $U_s = \sum_i U_i$. The j^{th} instance of task τ_i is called job $\tau_{i,j}$. A solid up arrow and a solid down arrow represent release time and deadline respectively. Each job consumes the Worst Case Execution Time (WCET) of its mandatory part m_i. The relative deadline D_i of each task τ_i is equal to its period T_i. The absolute deadline $D_{i,j}$ of each task τ_i is $D_{i,j} = r_{i,j} + T_i$, where $r_{i,j}$ is the j^{th} release time of job $\tau_{i,j}$. However, this model does not consider overloaded conditions in dynamic real-time environments. Therefore, a more flexible computation model than Liu and Layland's model is required.

2.1.2 Traditional Imprecise Computation Model

Lin et al. presented the traditional imprecise computation model [18] to consider overloaded conditions. Figure 2.2 shows an example of task execution in the traditional imprecise computation model. The crucial point is that the computation is split into two parts: mandatory part and optional part. A mandatory part as a critical part affects the correctness of the result. On the other hand, an optional part as a non-critical part only affects the quality of result. By restricting the execution of the optional part only after the completion of the mandatory part,

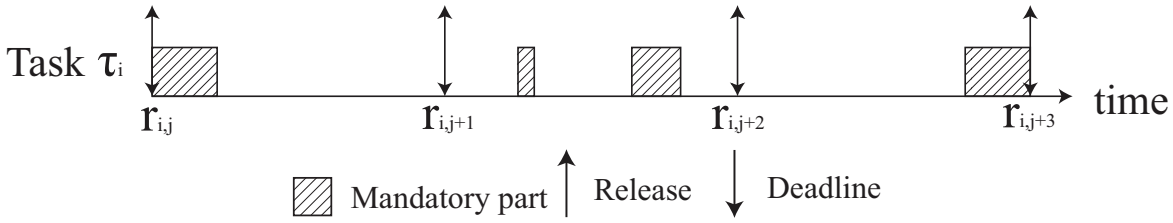


Figure 2.1: Example of task execution in Liu and Layland's model

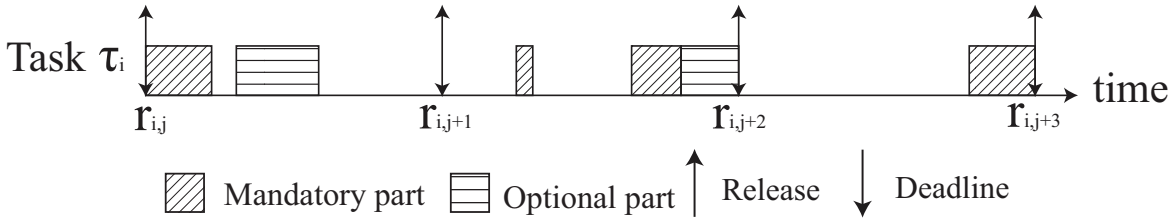


Figure 2.2: Example of task execution in the traditional imprecise computation model

real-time applications based on the traditional imprecise computation model can provide the correct output with lower quality, by terminating the optional part.

Figure 2.3 shows the value of result. The concave function $C_R(t)$ indicates the quality of result and is within the range of [0, 1]. If the concave function $C_R(t)$ is 0, the result does not make sense. In addition, t_m indicates the WCET of mandatory part. After t_m, then $C_R(t)$ is increased if the elapsed time is longer and longer. The interval $t_o - t_m$ indicates the Required Execution Time (RET) of the optional part. Unlike WCET, the RET has unknown upper bound. However, the traditional imprecise computation model does not consider the processing time to terminate or complete optional parts. For example, an object detection task must output the result to the actuator in robots so that the computation to output the result must guarantee the schedulability. In such cases, the traditional imprecise computation model is not practical.

2.1.3 Practical Imprecise Computation Model

Kobayashi and Yamasaki presented the practical imprecise computation model [33]. The practical imprecise computation model has more than one mandatory parts and more than one optional parts in each job. Thanks to the following mandatory parts after executing optional parts, the practical imprecise task guarantees the schedulability of the processing to output the result. Figure 2.4 shows an example of task execution in the practical imprecise computation model. In this example, task τ_i has two mandatory parts and one optional part. Task τ_i executes the first mandatory part, the first optional part and the second mandatory part sequentially. Job $\tau_{i,j+2}$ only executes the first mandatory part and the second mandatory part because job $\tau_{i,j+2}$ discards its optional part. The total WCET of mandatory parts in each job is $m_i = \sum_{l=1}^{n_i^m} m_i^l$, where n_i^m is the number of mandatory parts of task τ_i and m_i^l is the WCET of the l^{th} mandatory part of task τ_i. On the other hand, the total RET of optional parts in each job is $o_i = \sum_{l=1}^{n_i^o} o_i^l$, where n_i^o is the number of optional parts of task τ_i and o_i^l is the RET of the l^{th} mandatory part of task τ_i.

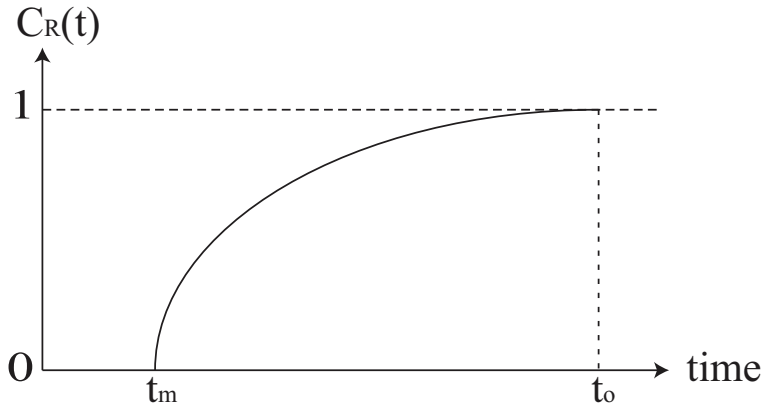


Figure 2.3: Value of result

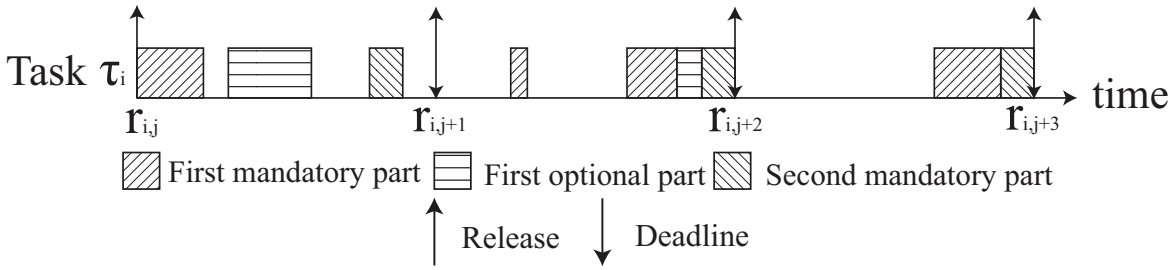


Figure 2.4: Example of task execution in the practical imprecise computation model

The schedulability analysis in the practical imprecise computation model is more difficult than other task models. Because the practical imprecise computation model has more than one mandatory parts so that the schedulability analysis in Liu and Layland's model may not be adapted to that in the practical imprecise computation model.

The practical imprecise computation model is similar to the self-suspension model [36]. The practical imprecise computation model defers the execution of the following mandatory part to execute the optional part. Each practical imprecise task must not miss its deadline by the deferred execution. That is to say, the schedulability of the practical imprecise computation model is higher than or equal to that of Liu and Layland's model. In contrast, the self-suspension model manages the worst case suspension time. Due to suspension, each self-suspension task may miss its deadline. Therefore, the practical imprecise computation model and the self-suspension model is different.

2.1.4 Summary of Traditional Computation Models

In this section, three traditional computation models are introduced. The practical imprecise computation model has the advantage of supporting overloaded conditions compared to Liu and Layland's model and overcomes the implementation problem of the traditional computation model. In addition, the practical imprecise computation has upward compatibility against both Liu and Layland's model and the traditional imprecise computation model. Therefore, the practical imprecise computation model is effective to achieve real-time systems in dynamic environments.

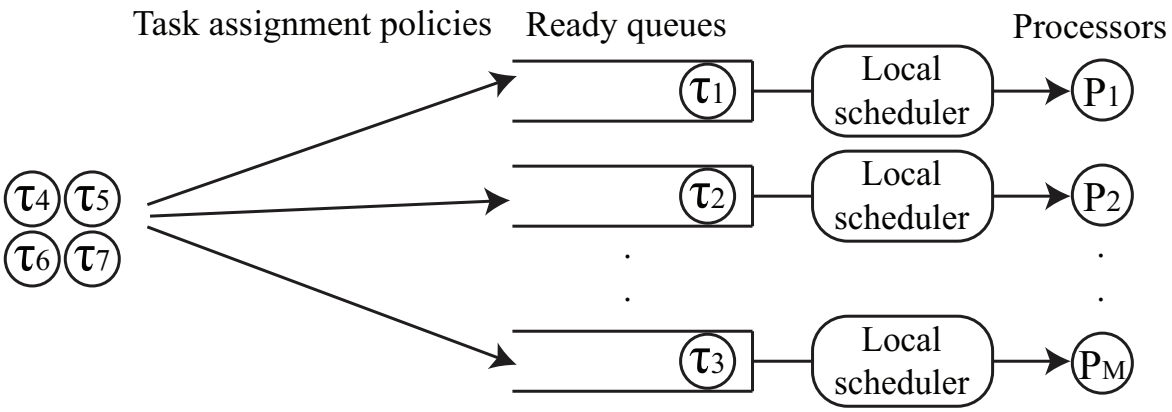


Figure 2.5: Partitioned scheduling

2.2 Multiprocessor Real-Time Scheduling

There are mainly two multiprocessor scheduling policies: *partitioned scheduling* and *global scheduling*. In addition, two hybrid policies with both advantages of partitioned scheduling and global scheduling are presented: *semi-partitioned scheduling* and *cluster scheduling*.

2.2.1 Partitioned Scheduling

Figure 2.5 shows the overview of partitioned scheduling. Partitioned scheduling assigns all tasks to specific processors beforehand. Partitioned scheduling has the following advantages against global scheduling.

- As each task only runs on each processor, then there is no penalty in terms of migration cost. For example, a task preempted by higher priority tasks must have its context restored on other processors. This can result in additional communication loads and cache misses that would not occur in partitioned scheduling.
- Partitioned scheduling has a separate run queue per processor, rather than a single global queue. For large systems, the overhead of manipulating a single global queue can become excessive by memory access, cache and synchronization.

From a practical perspective, the main advantage of multiprocessor partitioned scheduling is that once assigning tasks to processors has been achieved, the techniques of the schedulability analysis on uniprocessors can be applied on each processor.

The main disadvantage of multiprocessor partitioned scheduling is that the task assigning problem is similar with the bin packing problem, which is known to be NP-Hard [37].

In order to solve this problem, several heuristic task assignment policies have been presented; for example, first-fit, next-fit, best-fit and worst-fit and task orderings such as increasing relative deadline and decreasing utilization for task allocation. When a task is partitioned to a processor, the partitioned tasks on the processor are checked to be feasible by the utilization bound of scheduling algorithms or schedulability tests. For example, one task set is feasible by the EDF algorithm on each processor if the following condition is met [2]:

$$U_{ub}^{EDF} = \sum_{i=1}^{n} \frac{m_i}{T_i} \leqq 1, \tag{2.1}$$

where n is the number of tasks, then m_i is the WCET of task τ_i and T_i is the period of task τ_i. In the RM algorithm, one task set is feasible on each processor if the following equation is met [2]:

$$U_{lub}^{RM} \leqq n(2^{1/n} - 1). \tag{2.2}$$

Equation (2.2) is the utilization bound of the RM algorithm. For the more precise partitioned test, Response Time Analysis (RTA) [38] is performed in pseudo-polynomial time and can be analyzed precisely that one task set is feasible or not feasible on uniprocessors. The equation of RTA is

$$R_k = m_k + \sum_{i=1}^{k-1} \left\lceil \frac{R_k}{T_i} \right\rceil m_i, \tag{2.3}$$

where R_k is the worst case response time of task τ_k, then m_k is the WCET of task τ_k and T_i is the period of task τ_i.

Oh and Baker proved that the worst case utilization bound of the Partitioned Rate Monotonic (P-RM) algorithm [39] is

$$U_{wb}^{P-RM} = M(2^{1/2} - 1), \tag{2.4}$$

where M is the number of processors. Lopez et al. proved that the refined worst case utilization bound of the P-RM algorithm [40] is as follows.

$$U_{rwb}^{P-RM} = (M + 1)(1 + 2^{1/(M+1)}) \tag{2.5}$$

On the other hand, the worst case utilization bound of the Partitioned EDF (P-EDF) algorithm [41] is

$$U_{wb}^{P-EDF}(M, \beta) = \frac{\beta M + 1}{\beta + 1}, \tag{2.6}$$

where $\beta = \lfloor 1/U_{max} \rfloor$. If $\beta = 1$ in Equation (2.6), the utilization bound of the P-EDF algorithm is

$$U_{ub}^{P-EDF}(M, 1) = \frac{M + 1}{2}. \tag{2.7}$$

Partitioned scheduling has a critical disadvantage against global scheduling that the worst case utilization bound is potentially limited to at most 50% [42]. Figure 2.6 shows a problem of partitioned scheduling. Suppose that all the tasks are released at the time $t = 0$. Considering that $M + 1$ tasks have the same utilizations $50 + \epsilon\%$. Since an individual processor cannot be utilized over 100%, there is no idle processor to execute task τ_{M+1} when the M tasks are assigned. Letting $\epsilon \to 0$, a task set with a total utilization over $(M + 1)/2$ is never schedulable. They also have online problems that when a new task is submitted to the system at run-time, the tasks may be required to be sorted again to accept the new task. Such repartitioned processing can incur significant run-time overhead. Those drawbacks of the partitioned scheduling lead to revival of the global scheduling approaches to achieve more sophisticated scheduling of periodic tasks on multiprocessors.

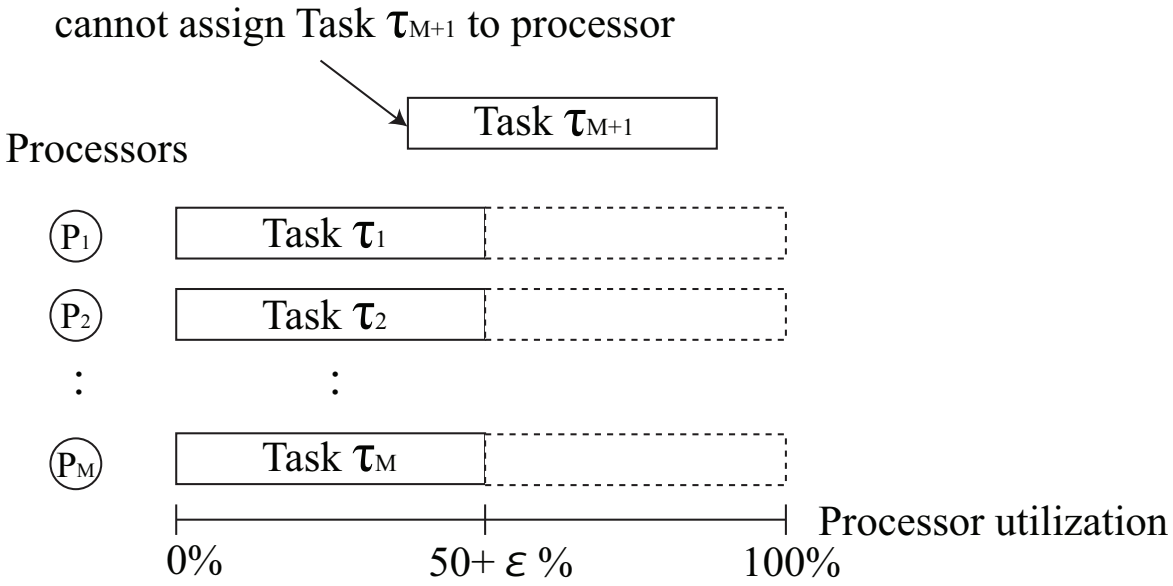


Figure 2.6: Problem of partitioned scheduling

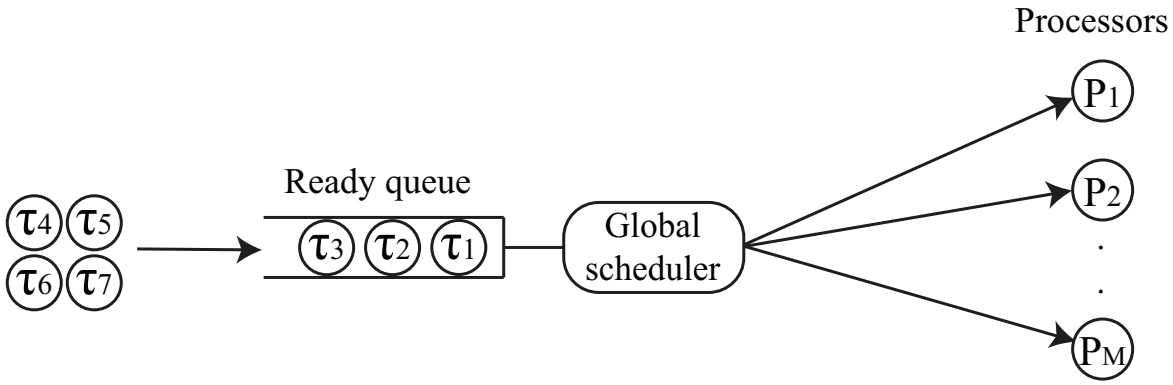


Figure 2.7: Global scheduling

2.2.2 Global Scheduling

Figure 2.7 shows the overview of global scheduling. Global scheduling permits to migrate tasks from one processor to another processor at run-time.

Global scheduling has the following advantages against partitioned scheduling.

- There are fewer context switches and preemptions when global scheduling policies are used, because the scheduler will only preempt a task when there is no idle processor [43].
- Global scheduling is more appropriate for open systems, as there is no need to run load balancing and task allocation algorithms when the task set is changed.

Now this dissertation introduces two types of global scheduling: optimal scheduling and non-optimal scheduling.

2.2.2.1 Optimal Scheduling

Optimal scheduling means that one periodic task set is feasible if the following equation is met:

$$U_s = \sum_{i=1}^{n} \frac{m_i}{T_i} \leqq M, \tag{2.8}$$

where U_s is the system utilization, M is the number of processors, n is the number of tasks, m_i is the WCET of task τ_i and T_i is the period of task τ_i.

Proportionate fair (Pfair) [44] is an optimal dynamic-priority scheduling algorithm and divides the timeline into equal length quanta. Due to dividing the timeline into equal length quanta, many context switches are caused.

In order to reduce the number of context switches in Pfair scheduling, the Early Release fair (ERfair) algorithm [45] is proposed. The ERfair algorithm is a Pfair-based work-conserving scheduling algorithm, if and only if there is no idle processor when at least one ready task exists, to reduce the run-time overhead and average response time. However, Pfair-based scheduling incurs significant overhead compared to other real-time scheduling.

Largest Local Remaining Execution First (LLREF) [46] is also an optimal scheduling algorithm and divides the timeline into sections separated by normal scheduling events, for example, release time and deadline. The LLREF algorithm has usually lower bounded overhead than Pfair scheduling with time quanta. The work-conserving algorithm for the LLREF algorithm [47] is presented to reduce the number of preemptions than the LLREF algorithm.

These approaches have the fairness of task execution, which causes high overhead. On the other hand, the approach of the unfairness based on the EDF algorithm is presented. Unfair EDF (U-EDF) [48] is an EDF-based optimal scheduling algorithm without fairness. The U-EDF algorithm reduces the number of preemptions and migrations than other optimal scheduling.

2.2.2.2 Non-Optimal Scheduling

Non-optimal scheduling is not always true that one periodic task set is feasible if Equation (2.8) is met. Non-optimal scheduling usually extends traditional uniprocessor real-time scheduling algorithms such as the RM and EDF algorithms for global scheduling. These algorithms are usually lower context switch and migration costs than optimal scheduling algorithms. Moreover, the implementation of non-optimal scheduling is usually easier than that of optimal scheduling.

Global RM (G-RM) is a global fixed-priority scheduling algorithm based on the RM algorithm and the utilization bound of the G-RM algorithm [49] is

$$U_{ub}^{G-RM} = \sum_{i=1}^{n} \frac{m_i}{T_i} \leqq \frac{M}{2}(1 - U_{max}) + U_{max}, \tag{2.9}$$

where $U_{max} = \max\{m_i/T_i \mid i = 1, ..., n\}$.

On the other hand, Global EDF (G-EDF) is a global dynamic-priority scheduling algorithm based on the EDF algorithm and the utilization bound of the G-EDF algorithm [50] is

$$U_{ub}^{G-EDF} = M(1 - U_{max}) + U_{max}. \tag{2.10}$$

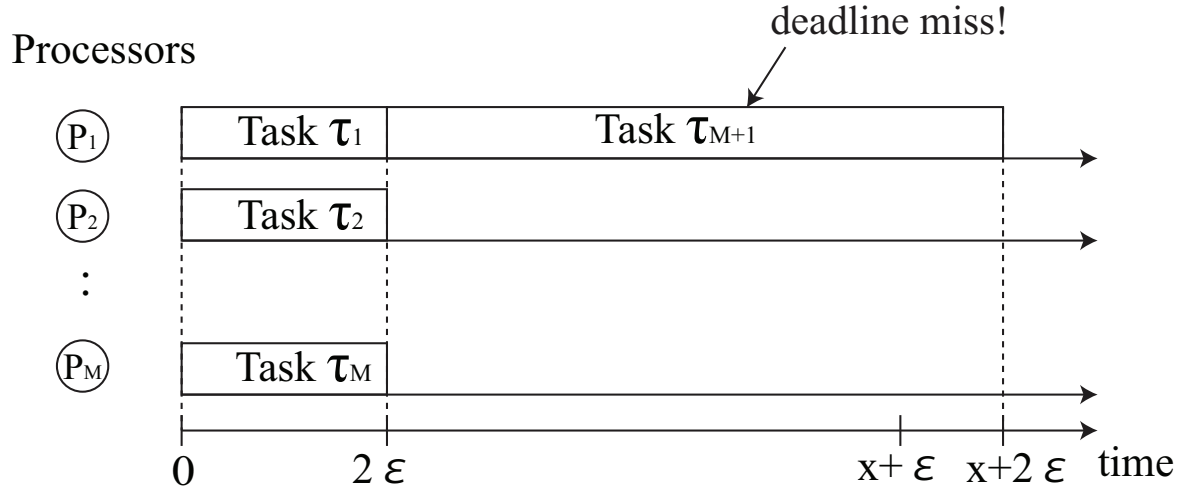


Figure 2.8: Problem of global scheduling

The utilization bounds of the G-RM and G-EDF algorithms depend on U_{max} and are lower and lower if U_{max} is higher and higher. The worst case utilization bound of both the G-RM and G-EDF algorithms is only 1 when $U_{max} = 1$, which does not depend on the number of processors. Therefore, the G-RM and G-EDF algorithms cannot make use of the multiprocessor capacity. Figure 2.8 shows a problem of global scheduling. There are $M + 1$ tasks on M processors according to the G-RM algorithm is given. Tasks τ_1, τ_2, ..., τ_M have the same periods of x and the same execution times of 2ϵ. On the other hand, task τ_{M+1} has a period of $x + \epsilon$ and an execution time of x. Notice that $2\epsilon < x$. Then, suppose that all the tasks are released at the time $t = 0$. The M tasks, periods of which are all x, are dispatched in advance according to the G-RM algorithm. Then, all the M tasks consume 2ϵ time units and complete at the time $t = 2\epsilon$ at the same time. Hence, task τ_{M+1} starts to be executed at the time $t = 2\epsilon$ on a processor. However, task τ_{M+1} cannot avoid missing its deadline because the task completes its execution at the time $t = x + 2\epsilon$ and the deadline of the task is $x + \epsilon$. Letting $\epsilon \to 0$, the M tasks have the zero utilization because their execution times become 0. In addition, task τ_{M+1} has the 100% utilization because its period is equal to its worst case execution time. The total utilization of the tasks becomes 100%, namely the system utilization becomes $1/M \times 100\%$. Therefore, a deadline can be missed even if only the $1/M$ of the system is utilized. The G-EDF algorithm also sees the same phenomenon with the same task set. This phenomenon is often called Dhall's effect [51]. Dhall's effect occurs for the above case in which high utilization tasks exist.

In order to overcome Dhall's effect, the technique of the utilization separation is presented. RM-US[M/(3M-2)] [52] is a G-RM based fixed-priority scheduling algorithm with the technique of the utilization separation and gives the highest priority to task τ_i if $U_i > M/(3M - 2)$. If $U_i \leqq M/(3M - 2)$, then task τ_i has the RM priority. The utilization bound of the RM-US[M/(3M-2)] algorithm is

$$U_{ub}^{RM-US[M(3M-2)]} = \frac{M^2}{3M - 2} \tag{2.11}$$

On the other hand, Earliest Deadline First with Utilization Separation (EDF-US)[M/(2M-1)] [50] is a G-EDF based dynamic-priority scheduling algorithm with the technique of the utilization separation and gives the highest priority to task τ_i if $U_i > M/(2M - 1)$. If

$U_i \leqq M/(2M-1)$, then task τ_i has the EDF priority. The utilization bound of the EDF-US[M/(2M-1)] algorithm is

$$U_{ub}^{EDF-US[M/(2M-1)]} = \frac{M^2}{2M-1}. \tag{2.12}$$

The disadvantage of the technique of the utilization separation is that the EDF-US algorithm does not dominate the G-EDF algorithm. That is to say, there is one task set, which is schedulable by the G-EDF algorithm if the task set is not schedulable by the EDF-US algorithm. In order to overcome the weakness of the technique of the utilization separation, the zero laxity rule is presented.

Earliest Deadline until Zero Laxity (EDZL) [53] is a G-EDF-based dynamic-priority scheduling algorithm with the zero laxity rule. The zero laxity rule means that the task is set to the highest priority if the laxity of task τ_i at the time t, which denotes $l_i(t) = D_i - R_i(t) - t$, is equal to 0. Here, D_i is the deadline of task τ_i and $R_i(t)$ is the remaining execution time of task τ_i. Until when the zero laxity rule occurs, the EDZL algorithm generates the same schedule as the G-EDF algorithm. Thanks to the zero laxity rule, the EDZL algorithm dominates the G-EDF algorithm and achieves higher schedulability than other non-optimal real-time scheduling algorithms with lower number of preemptions compared to optimal scheduling. The utilization bound of the EDZL algorithm [54] is

$$U_{ub}^{EDZL} = M(1 - \frac{1}{e}) \simeq 0.6321M, \tag{2.13}$$

where $e \simeq 2.718$ is the Euler's number.

Earliest Deadline until Critical Laxity (EDCL) [55] is a G-EDF-based dynamic-priority scheduling algorithm and also dominates the G-EDF algorithm like the EDZL algorithm. The EDCL algorithm generates the same schedule as the G-EDF algorithm until the laxity of task τ_i is critical, if its laxity holds the following condition at the time t.

$$l_i(t) < e_{\min}, \tag{2.14}$$

where $e_{\min}$ denotes the minimum remaining execution time of the M jobs that have the earliest deadlines. When Equation (2.14) meets, the laxity is critical. The priority-promotion operation of the critical laxity is aligned with times where some jobs are released or completed so that the scheduler invocation does not occur due to this operation.

2.2.3 Hybrid Scheduling

Global scheduling usually incurs significant overhead depending on the hardware architecture. Because the migrating task can result in cache misses, which causes increased WCET of each task. However, partitioned scheduling suffers from the limitation of schedulability. In order to integrate the advantages of both global scheduling and partitioned scheduling, there are two hybrid approaches: *semi-partitioned scheduling* and *cluster scheduling*.

2.2.3.1 Semi-Partitioned Scheduling

Semi-partitioned scheduling assigns most tasks to specific processors, while a few tasks are migrated to another processor.

EDF-fm (fm denotes that each task is either fixed or migrating) [56] is a semi-partitioned EDF-based dynamic-priority scheduling algorithm and assigns the highest priority to migrating tasks. The non-migrating tasks are scheduled according to the EDF algorithm when no migrating task is ready to be executed.

EDF with task splitting and K processors in a Group (EKG) [57] is an EDF-based dynamic-priority scheduling algorithm with semi-partitioned approach. The utilization bound of the EKG algorithm depends on the parameter k, which is used to control division of tasks into groups of both high and low utilization tasks. The utilization bound of the EKG algorithm is as following equation.

$$U_{ub}^{EKG} = \begin{cases} \frac{k}{k+1} & k < M \\ 1 & k = M \end{cases} \tag{2.15}$$

By Equation (2.15), the utilization of the EKG algorithm is optimal if $k = M$ as well as optimal global scheduling. Moreover, the average number of preemptions per job over the hyperperiod is bounded by $2k$.

Rate Monotonic Deferrable Portion (RMDP) [58] is a semi-partitioned RM-based fixed-priority scheduling algorithm and makes use of the portioned scheduling technique, which classifies each task into a fixed or migrating task. A fixed task is scheduled on the dedicated processor without migrations. A migrating task is migrated between two processors so that the RMDP algorithm makes at most M - 1 migrating tasks on M processors. The utilization bound of the RMDP algorithm is as following equation.

$$U_{ub}^{RMDP} = 0.5 \tag{2.16}$$

Earliest Deadline Deferrable Portion (EDDP) [59] is a semi-partitioned EDF-based dynamic-priority scheduling algorithm and also makes use of the portioned scheduling technique. The EDDP algorithm as well as the RMDP algorithm makes at most M - 1 migrating tasks on M processors. The utilization bound of the EDDP algorithm is as following equation.

$$U_{ub}^{EDDP} = 4\sqrt{2} - 5 \simeq 0.65 \tag{2.17}$$

Deadline Monotonic with Priority Migration (DM-PM) [60] is a semi-partitioned fixed-priority scheduling algorithm and dominates partitioned fixed-priority scheduling. Because the DM-PM algorithm permits to migrate tasks to another processor if they fit on any processor. A migrating task is set to the highest priority, with portions of their execution time assigned to processors.

EDF with Window-constraint Migration (EDF-WM) [61] is a semi-partitioned EDF-based dynamic-priority scheduling algorithm and dominates partitioned dynamic-priority scheduling like the DM-PM algorithm.

Partitioned Deadline-Monotonic Scheduling (PDMS) algorithms by allowing the Highest Priority Task on a processor to be Split (HPTS) across more than one processor (PDMS_HPTS) [62] is presented to improve the schedulability of semi-partitioned scheduling. The least upper bound of the PDMS_HPTS algorithm is as following equation.

$$U_{lub}^{PDMS\_HPTS} \simeq 0.6003 + \frac{0.0928}{M} \tag{2.18}$$

In addition, PDMS_HPTS with allocating tasks to processors in the Decreasing order of Size, called PDMS_HPTS_DS [62] improves the least upper bound until the following equation.

$$U_{lub}^{PDMS\_HPTS\_DS} \simeq 0.6547 \tag{2.19}$$

Guan et al. proposed two semi-partitioned RM-based fixed-priority scheduling algorithms: SPA1 and SPA2 [63]. The utilization bound of the SPA1 and SPA2 algorithms are analyzed with Liu and Layland's utilization bound [2]. The utilization bound of the SPA1 algorithm is

$$U_{ub}^{SPA1} = U_{lub}^{RM} = n(2^{1/n} - 1), \tag{2.20}$$

where n is the number of tasks if the utilization of each task is lower than or equal to $U_{lub}^{RM}/(1 + U_{lub}^{RM})$. By introducing an extra task pre-assigning mechanism, the utilization bound of the SPA2 algorithm is also Equation (2.20) for any task set.

Deadline Partitioned WRAP (DP-WRAP) [64] is a semi-partitioned EKG-based dynamic-priority scheduling algorithm to simplify the EKG algorithm. In the DP-WRAP algorithm, the non-migrating tasks assigned to a given processor are scheduled in the EDF algorithm on each processor instead of McNaughton's wrap around algorithm [65], which reduces the number of context switches.

Reduction to UNiprocessor (RUN) [66] is an optimal semi-partitioned EDF-based dynamic-priority scheduling algorithm, like the EKG algorithm. The RUN algorithm transforms the multiprocessor scheduling problem into an equivalent set of uniprocessor problems. The RUN algorithm significantly outperforms existing optimal scheduling algorithms with an upper bound of $O(log\ M)$ average preemptions per job on M processors and reduces the P-EDF algorithm whenever a proper partitioning is found.

2.2.3.2 Cluster Scheduling

Cluster scheduling permits to migrate tasks from one processor to another processor in each cluster. Cluster scheduling has the advantage of reducing migration overhead, compared to global scheduling, because each cluster usually shares L2 or L3 caches. Therefore, when each task is migrated to another processor, the migrated task can reuse data on L2 or L3 cache.

The utilization bound of the Cluster EDF (C-EDF) algorithm [67] is

$$U_{ub}^{C-EDF}(M_k, k, \beta) = \frac{\beta M_k + 1}{\beta + 1}\, k, \tag{2.21}$$

where $\beta = \lfloor c/U_{max} \rfloor$ and c is the number of processors in each cluster. If each cluster has one processor (i.e., $k = 1$ and $M_k = M$), the utilization bound of the C-EDF algorithm is equal to that of the P-EDF algorithm in Equation (2.6).

2.2.4 Summary of Multiprocessor Real-Time Scheduling

Many multiprocessor real-time scheduling algorithms in Liu and Layland's model have been presented. However, Liu and Layland's model does not support overloaded conditions so that the imprecise computation is required to achieve multiprocessor real-time systems, which run in dynamic environments.

2.3 Real-Time Scheduling for Imprecise Computation

2.3.1 Real-Time Scheduling for Imprecise Computation on Uniprocessors

The Mandatory-First with Earliest Deadline (M-FED) algorithm [68] is an EDF-based dynamic-priority scheduling algorithm in the traditional imprecise computation model on uniprocessors. The M-FED algorithm ranks mandatory parts of tasks by the EDF algorithm and all mandatory parts have higher priorities than all optional parts. The M-FED algorithm is optimal about *competitive ratio* if the value of result created by each task is uniform. The competitive ratio $c.A$ of a scheduling algorithm A is defined to be the worst case ratio of the value returned by an optimal off-line algorithm on the task set Γ to the value returned by A on the same task set. The competitive ratio is defined as follows:

$$C.A = \max_{all\ \tau} \frac{v.OPT(\tau)}{v.A(\tau)}, \tag{2.22}$$

where OPT is an optimal off-line schedule and the value obtained by an algorithm on a task set is defined to be the sum of the values obtained on each task in the task set.

The OPTimization with Least-Utilization (OPT-LU) algorithm [69] is an EDF-based scheduling algorithm in the traditional imprecise computation model on uniprocessors. The goal of the OPT-LU algorithm is to maximize the reward. An instance of the OPT-LU algorithm is specified by the set of non-decreasing concave reward functions of each task $F = \{f_1, f_2, ..., f_n\}$, the set of upper bounds of optional parts $O = \{o_1, o_2, ..., o_n\}$, the rational number b_i, the assignable optional part o_i and the available slack S. The aim of the OPT-LU algorithm is:

$$\text{maximize} \quad \sum_{i=1}^{n} f_i(t_i) \tag{2.23}$$

$$\text{subject to} \quad \sum_{i=1}^{n} b_i t_i = S \tag{2.24}$$

$$t_i \leqq o_i \; i = 1, 2, ..., n$$

$$0 \leqq t_i \; i = 1, 2, ..., n,$$

where $0 < S < \sum_{i=1}^{n} b_i o_i$, then b_i is the rational number and t_i is the assignable time of optional part. However, the OPT-LU algorithm assumes that the WCET of each optional part can be analyzed.

The Mandatory-First with Wind-up Part (M-FWP) algorithm [34, 35] is an EDF-based dynamic-priority scheduling algorithm in the practical imprecise computation model on uniprocessors. The M-FWP algorithm schedules each practical imprecise task by the following parameters.

- $l_{i,j}(t)$: the sum of the remaining execution time including all mandatory parts of job $\tau_{i,j}$ at the time t
- $R_{i,j}(t)$: the assignable time of job $\tau_{i,j}$ at the time t
- $S_{i,j}(t)$: the assignable time of the optional part of job $\tau_{i,j}$ at the time t

- $X_{i,j}(t)$: the reserved time of executing the optional part of job $\tau_{i,j}$

Since there is a case that each task completes its optional part to consume all the assignable time of the optional part, the following equation always holds:

$$S_{i,j}(t) = R_{i,j}(t) - l_{i,j}(t). \tag{2.25}$$

In addition, the following groups are defined.

- $\Gamma_h(\tau_{i,j})$: the group of all ready jobs which have higher priority than job $\tau_{i,j}$
- $\Gamma_{hp}(\tau_{i,j})$: the group of jobs that satisfy $r_{k,l} + T_k < D_{i,j}$
- $\Gamma_{hpe}(\tau_{i,j})$: the group of jobs in $\Gamma_{hp}(\tau_{i,j})$ that satisfy $((D_{i,j} - r_{k,l}) \bmod T_k) < D_k$

Then, the amount of idle processor time allocated to the optional part of job $\tau_{i,j}$ is estimated as

$$\begin{aligned} S_{i,j}(t) = \min(D_{i,j} - t - l_{i,j}(t) - E_{i,j}(t) - F_{i,j}(t) - \min(G_{i,j}(t), H_{i,j}(t)), \\ S^n_{i,j}(t) - X^n_{i,j}(t)), \end{aligned} \tag{2.26}$$

where $S^n_{i,j}(t)$ is the remaining optional computation time of the next job in the optional ready queue at the time t and $X^n_{i,j}(t)$ is the amount of its optional computation time reserved. In addition, $E_{i,j}(t)$, $F_{i,j}(t)$, $G_{i,j}(t)$ and $H_{i,j}(t)$ are as follows.

$$E_{i,j}(t) = \sum_{\tau_{k,l} \in \Gamma_h(\tau_{i,j})} R_{k,l}(t) \tag{2.27}$$

$$F_{i,j}(t) = \sum_{\tau_{k,l} \in \Gamma_{hp}(\tau_{i,j})} \left(1 + \left\lfloor \frac{D_{i,j} - (r_{k,l} + T_k) - D_k}{T_k} \right\rfloor\right) m_i \tag{2.28}$$

$$G_{i,j}(t) = \sum_{\tau_{k,l} \in \Gamma_{hpe}(\tau_{i,j})} \min(m_i + w_i, (D_{i,j} - r_{k,l}) \bmod T_k) \tag{2.29}$$

$$H_{i,j}(t) = \max((D_{i,j} - r_{k,l}) \bmod T_k \mid \tau_{k,l} \in \Gamma_{hpe}(\tau_{i,j})) \tag{2.30}$$

The M-FWP algorithm calculates the assignable time of the optional part by Equation (2.26). If there is the assignable time of the optional part, execute the optional part of each task. Otherwise, the M-FWP algorithm discards the optional part of each task and executes its following mandatory part.

The Slack Stealer for Optional Parts (SS-OP) algorithm [70] is also an EDF-based dynamic-priority scheduling algorithm in the practical imprecise computation model on uniprocessors. In order to maximize the quality of result, the SS-OP algorithm distributes the assignable time of the optional part, the upper bound of which is

$$U_s = 1 - \sum_{i=1}^{n} \frac{m_i}{T_i}. \tag{2.31}$$

The job which has the least absolute deadline $D_{i,j}$ from current time t consumes the assignable time of the optional part in $[t, D_{i,j})$. In $[t, D_{i,j})$, the assignable time of the optional part is

Table 2.1: Overview of real-time scheduling for imprecise computation

Algorithm	Practical	Uniprocessor	Multiprocessor
M-FED		✓	
OPT-LU		✓	
M-FWP	✓	✓	
SS-OP	✓	✓	

$(D_{i,j} - t)U_s$. The SS-OP algorithm assigns the slack time S_i to job $\tau_{i,j}$ to maximize the following equation.

$$\text{maximize} \quad \sum_{i=1}^{n} \frac{f_i(S_i)}{T_i} \tag{2.32}$$

$$\text{subject to} \quad \sum_{i=1}^{n} \frac{S_i}{\min(T_i, D_i)} \leqq U_s \tag{2.33}$$

$$\forall i, 0 \leqq S_i \leqq o_i,$$

where f_i is the value function of task τ_i and o_i is the WCET of the optional part of task τ_i. However, like the OPT-LU algorithm, the SS-OP algorithm requires the known WCET of each optional part. The WCET of each optional part may be unknown because real-time applications run in dynamic environments where the overloaded conditions occur suddenly. Therefore, the SS-OP algorithm cannot be adapted to such situations.

2.3.2 Real-Time Scheduling for Imprecise Computation on Multiprocessors

The research of real-time scheduling in the traditional imprecise computation model on multiprocessors is introduced.

Khemka et al. discuss the problem of multiprocessor real-time scheduling for imprecise computations, as a network flow problem [71]. Yun et al. propose a heuristic scheduling algorithm for imprecise computation with 0/1 constraint on multiprocessors [72]. Stavrinides and Karatza evaluate the performance of dynamic-priority scheduling in distributed real-time systems [73, 74]. However, these approaches for multiprocessor or distributed systems do not analyze the schedulability. Therefore, real-time scheduling algorithms in the traditional imprecise computation model do not support multiprocessor real-time scheduling.

2.3.3 Summary of Real-Time Scheduling for Imprecise Computation

Table 2.1 shows the overview of real-time scheduling for imprecise computation. The M-FED and OPT-LU algorithms in the traditional imprecise computation model are impractical because the traditional imprecise computation model assumes that there is no time terminating or completing optional parts. In order to overcome the weakness of the traditional imprecise computation model, the practical imprecise computation model is presented. In the practical imprecise computation model, the M-FED and OPT-LU algorithms do not support multiprocessor real-time scheduling. In contrast, the M-FWP and SS-OP algorithms in the practical imprecise computation model are adapted to real-time applications based on

imprecise computation, thanks to following mandatory parts after executing optional parts. Unfortunately, the M-FWP and SS-OP algorithms also do not support multiprocessor real-time scheduling, like the M-FED and OPT-LU algorithms.

2.4 Real-Time Operating Systems

There are two approaches to implement real-time operating systems. One approach is to extend general purpose operating systems for real-time processing. The other approach is to implement real-time operating systems from scratch for specific real-time applications. Now this dissertation introduces real-time operating systems based on the two approaches.

2.4.1 Real-Time Extensions of General Purpose Operating Systems

Real-time extensions of general purpose operating systems have the following advantages against proprietary real-time operating systems.

- Developers can make use of many Application Program Interfaces (APIs), libraries and device drivers to develop real-time applications on general purpose operating systems.
- Many application users make use of real-time applications easily because they always use general purpose operating systems.

Real-time extensions of Linux [75] are major approaches in real-time systems. Now this dissertation introduces several real-time extensions of Linux.

The RT-Linux real-time operating system [76] is a real-time extension of Linux for hard real-time systems. The RT-Linux real-time operating system provides the capability of running special real-time tasks and interrupt handlers on the same machine as standard Linux.

The RTAI real-time operating system [77] is also a real-time extension of Linux. The RTAI real-time operating system has supported the Adaptive Domain Environment for Operating Systems (ADEOS) nano-kernel as an alternative for RTAI's core to get rid of the old kernel patch.

The Xenomai real-time operating system [78] supports to execute real-time RTAI tasks in user space. The Xenomai real-time operating system brings the concept of virtualization one step further: like RTAI, it uses the ADEOS nano-kernel to provide the interrupt virtualization. However, it also permits to execute a real-time task in user space.

The MontaVista Linux real-time operating system [79] meets embedded developers where they are in the development cycle with a complete embedded Linux distribution and developer tools for a faster time-to-development.

The PREEMPT_RT real-time operating system [80] is implemented as a kernel patch to make Linux more predictable and deterministic. This is done through several optimizations. The kernel patch makes almost all kernel code preemptive except the most critical kernel routines for reducing the maximum latency.

The Linux/RK real-time operating system [81] has been directly modified to introduce real-time features of Linux. The Linux/RK real-time operating system provides resource reservations directly to user processes. The use of this mechanism is transparent. Therefore, it is possible to assign a reservation to a legacy Linux application and to access a specific API to take advantage of the reservations and the quality of service management.

The KURT-Linux real-time operating system [82] satisfies the constraints of the firm real-time applications in Linux. By running the hardware timer as an aperiodic device, the KURT-Linux real-time operating system has increased the temporal resolution of the system without significantly increasing the overhead of the software clock, called UTIME.

The ART-Linux real-time operating system [83] is also one of hard real-time extensions in Linux. The features of the ART-Linux real-time operating system are to execute hard real-time tasks in user level and not to require special device drivers.

The RED-Linux real-time operating system [84] implements a general scheduling framework, which divides the system scheduler into two components: dispatcher and allocator. The dispatcher provides the mechanism of real-time scheduling and resides in the kernel space. The allocator is used to define the scheduling policy and implemented as a user space function. This framework allows users to implement application-specific schedulers in the user space which is easy to program and to debug.

The Linux-SRT real-time operating system [85] is a real-time version of Linux enhanced with support for predictable scheduling and quality of service management. It is binary compatible with standard Linux: existing applications can benefit from quality of service without being modified in any way. Processor and disk bandwidth are scheduled, and scheduling policies are propagated to servers. Automated control and management features simplify the use of advanced features.

The LITMUS^{RT} real-time operating system [86] is a soft real-time extension of Linux and supports multiprocessor real-time scheduling. The LITMUS^{RT} real-time operating system provides a useful experimental platform for applied real-time systems research.

The Redline real-time operating system [87] brings the first-class support for interactive applications in Linux. Redline relies on lightweight specifications, which gives an estimate of the resources required by an application over any period in which they are active for their responsiveness.

The SCHED_DEADLINE real-time operating system [88, 89] is a kernel patch of a new scheduling class in Linux so that normal tasks can still behave as when the kernel patch is not applied. Eventually, this kernel patch might be merged inside Linux mainline.

The AQuoSA real-time operating system [90] features a flexible, portable, lightweight and open architecture for supporting soft real-time applications with facilities related to timing guarantees and quality of service, on the top of a general-purpose operating system as Linux.

The AIRS real-time operating system [91] is aimed at supporting systems that run multiple interactive real-time applications such as H264 movie to improve the quality of service of the overall systems.

The ChronOS real-time operating system [92] is a best-effort real-time multiprocessor Linux kernel and addresses the intersection of three problem spaces: (i) operating system support for obtaining best-effort timing assurances; (ii) real-time Linux kernel augmented with the PREEMPT_RT patch and (iii) operating system support for multiprocessor real-time scheduling.

On the other hand, other real-time extensions of general purpose operating systems are introduced.

The RT-Mach real-time operating system [93] is developed as a real-time version of the Mach operating system [94] and supports predictable real-time computing environments. Unfortunately, the development of the RT-Mach real-time operating system was stopped because the porting costs from the Mach operating system to the RT-Mach real-time operating system and the implementation costs of new device drivers are too much [95].

The Windows CE real-time operating system [96] is a real-time extension of Windows [97] designed for a wide range of small-footprint consumer and enterprise devices and is optimized for devices that have minimum storage.

The RT-UNIX real-time operating system [98] is a real-time extension of the UNIX operating system [99] to achieve real-time operation in the UNIX environments.

2.4.2 Proprietary Real-Time Operating Systems

Proprietary real-time operating systems have the following advantages against real-time extensions of general purpose operating systems.

- Real-time application-specific implementation can be achieved without no-use codes. Because general purpose operating systems implement various codes which does not use real-time applications. Due to the no-use codes, the reliability of real-time applications may be degraded.

- Optimal design and implementation can be achieved because proprietary real-time operating systems have been developed from scratch.

- Developers do not need to consider the application compatibility unlike general purpose operating systems.

2.4.2.1 Standards of Real-Time Operating Systems

The role of standards in real-time operating systems is very important as it provides portability of applications from one platform to another platform. In addition, standards allow the possibility of having several kernel providers for a single application to promote competition among vendors and to increase quality. Current real-time operating system standards mostly specify portability at the source code level, requiring the application developer to recompile the application for every different platform.

The RT-POSIX standard [100] is a real-time extension of the POSIX standard [101], which is the portability of applications at the source code level.

- Minimal real-time systems profile (PSE51) is intended for small embedded systems so that most of the complexity of general purpose operating systems is eliminated. The unit of concurrency is the thread and processes are not supported. Input and output operations are possible through predefined device files. However, there is not a complete file system. The PSE51 profile can be implemented with a few thousand lines of code and memory footprints in the tens of kilobytes range.

- Real-time controller profile (PSE52) is similar to the PSE51 profile, with the addition of a file system in which regular files can be created, read or written. It is intended for systems like a robot controller, which may need support for a simplified file system.

- Dedicated real-time system profile (PSE53) is intended for large embedded systems (e.g., avionics) and extends the PSE52 profile with the support for multiple processes that operate with protection boundaries.

- Multi-purpose real-time system profile (PSE54) is intended for general purpose systems running applications with real-time and non-real-time requirements. It requires most of the POSIX functionality for general purpose systems and most real-time services.

The HeartOS real-time operating system [102] is fast, light and well featured for most for small to medium embedded applications including safety-critical applications. The HeartOS real-time operating system is compliant with the PSE51 profile. The SHaRK real-time operating system [103] has a dynamically configurable module to support development and test of new scheduling algorithms and synchronization protocols. The SHaRK real-time operating system is compliant with the PSE52 profile. The MaRTE OS real-time operating system [104] follows the PSE51 profile. The services in the MaRTE OS real-time operating system have a time-bounded response so that hard real-time requirements can be supported.

The μITRON [105] specification is specified as a real-time operating system for improving reliability and reusability of embedded and real-time systems. Real-time operating systems in the ITRON specification have been applied over a large range of embedded application domains.

- audio-visual equipment (TVs, digital cameras and audio components)
- home appliances (microwave ovens, rice cookers, air-conditioners and washing machines)
- personal information appliances (PDAs, personal organizers and car navigation systems)
- entertainment (game gears and electronic musical instruments)
- PC peripherals (printers, scanners, disk drives and CD-ROM drives)
- office equipment (copies, FAX machines and word processors)
- communication equipment (phone answering machines, ISDN telephones, cellular phones, ATM switches, wireless systems and satellites)
- transportation (automobiles)
- industrial control (plant control and industrial robots)
- others (elevators, vending machines, medical equipment and data terminals)

Examples of real-time operating systems in the μITRON specification are TOPPERS/JSP [106], HOS [107], T-Kernel [108], Nucleus [109] and TNKernel [110].

The Offene Systeme und deren schnittstellen fur die Elektronik im Kraftfahrzeug/Vehicle Distributed eXecutive (OSEK/VDX) specification [111] aims at the definition of an industrial standard for an open-ended architecture for distributed control units in vehicles. The objective of the OSEK/VDX specification is to describe an environment that supports efficient utilization of resources for automotive application software. This standard can be viewed as an API for real-time operating systems integrated on a network management system, which describes the characteristics of a distributed environment that can be used for developing automotive applications. Examples of real-time operating systems in the OSEK/VDX

specification are TOPPERS/ATK [112], Erika Enterprise [113], FreeOSEK [114], PICOS18 [115] and Trampoline [116].

The AUTomotive Open System ARchitecture OS (AUTOSAR OS) specification [117] extends the OSEK/VDX specification. The AUTOSAR OS specification reserves execution time, release time, interrupt disable time and resource time. Currently, the TOPPERS/ATK real-time operating system is extended for the AUTOSAR OS specification.

The Avionics Application Standard Software Interface 653 (ARINC 653) specification [118] is specified for avionics real-time systems that specifies how to host multiple applications on the same hardware. In order to decouple the real-time operating system from the application software, ARINC 653 defines an API called APplication/EXecutive (APEX). The goal of the APEX API is to allow analyzable safety critical real-time applications to be implemented, certified and executed. Several critical real-time systems have been successfully built and certified using APEX, including some critical components for the Boeing 777 aircraft. An example of real-time operating system in the ARINC 635 specification is INTEGRITY-178B [119].

Now this dissertation introduces real-time operating systems, which supports multiple specifications.

The eCos real-time operating system [120] is intended for embedded applications. The highly configurable nature of the eCos real-time operating system allows the operating system to be customized to precise application requirements, delivering the best possible runtime performance and an optimized hardware resource footprint. The eCos real-time operating system has compatibility layers and APIs for the PSE51 profile, the PSE52 profile and the μITRON specification.

The RTEMS real-time operating system [121] supports many open APIs and interface standards including the PSE52 profile and the μITRON specification.

The LynxOS real-time operating system [122] is the superior foundation for sophisticated real-time systems and supports the PSE53 specification, the ARINC 653 specification and Linux applications simultaneously in a single partition.

2.4.2.2 Proprietary Implementations of Real-Time Operating Systems for Research

Many researchers have developed real-time operating systems from scratch to evaluate real-time scheduling algorithms and synchronization protocols in real-time systems. Now some of real-time operating systems are introduced.

The Alpha real-time operating system [123] is a novel nonproprietary operating system for large, complex and distributed real-time systems. Examples include combat platform and battle management, factory automation and telecommunications.

The ARX/ULTRA real-time operating system [124] employs user level threads for scheduling, communication and multithreading. The goal of the ARX/ULTRA real-time operating system is to provide (i) flexible and predictable scheduling services; (ii) effective management of kernel-level thread blocking; (iii) efficient handling of scheduling events including timer interrupts.

The CHAOS real-time operating system [125] offers kernel-level primitives that allow high-performance and large-scale real-time software to be programmed as a system of interacting objects.

The CHIMERA II real-time operating system [126] is designed to reduce the development time by providing a convenient software interface between the hardware and the user.

The Contiki real-time operating system [127] supports highly portable multitasking and is developed for use on a number of memory-constrained networked systems ranging from small to medium embedded systems on microcontrollers including sensor network systems.

The EMERALDS real-time operating system [128] is designed for small to medium size embedded systems. The EMERALDS real-time operating system uses the novel approach of mapping the kernel into each user-level address space with full memory protection. Therefore, system calls do not need context switches unless a user-level server is involved.

The EOS real-time operating system [129] is intended to be compact enough to be used in embedded systems and provides standard capabilities such as message handling, memory sharing, device management and direct support for error tracking and recovery.

The EROS real-time operating system [130] combines an unusual collection of facilities into a single package, hopefully in a novel way. Each of these facilities is essential to providing scalable reliability. Currently, the EROS real-time operating system is merged into the CapROS real-time operating system [131].

The Fiasco real-time operating system [132] is used to construct flexible systems. The Fiasco real-time operating system is not only both suitable for big and complex systems but also for small and embedded applications.

The HARTIK real-time operating system [133] is designed to provide facilities for programming robot tasks with explicit timing constraints and predictable execution. The HARTIK real-time operating system is used as a platform for programming predictable real-time tasks in robotics applications, where control tasks and sensor acquisition processes have to be performed at different rates.

The HARTOS real-time operating system [134] is to design and implement an experimental distributed real-time system. An important feature of the HARTOS real-time operating system is the use of an intelligent network processor to handle many of the functions relating to communications.

The hthreads real-time operating system [135] is a hardware/software co-design of a multithreaded kernel and an integral part of the hybrid thread programming model being developed for hybrid systems, which are comprised of both software resident and hardware resident concurrently executing threads.

The Mars real-time operating system [136] guarantees a deterministic system behavior. The goal of the Mars real-time operating system is to administer resources (processor, memory and bus) and to hide all hardware details from the tasks.

The MARUTI real-time operating system [137] is designed to support real-time applications on a variety of hardware systems. The MARUTI real-time operating system supports guaranteed-service scheduling, in which jobs that are accepted by the system are verified to satisfy timing constraints.

The MERT real-time operating system [138] is built on top of a kernel, which provides the basic services such as memory management, process scheduling, and trap handling needed to build various operating system environments.

The Nano-RK real-time operating system [139] has multi-hop networking support for use in wireless sensor networks. The Nano-RK real-time operating system supports fixed-priority preemptive multitasking for ensuring that task deadlines are met, along with support for processor and network as well as sensor and actuator reservations.

The Quest real-time operating system [140] supports virtual real-time scheduling on x86 multiprocessors. The Quest real-time operating system develops a relatively small kernel for research and educational purposes avoiding complexities of existing open source systems like Linux.

The S.O.O.S. real-time operating system [141] implements servers as a resource reservation mechanism, which offers the ability to execute tasks with soft real-time requirements.

The Spring real-time operating system [142] first considers supporting multiprocessors. The goal of the Spring real-time operating system extends beyond stand-alone multiprocessor systems and encompasses distributed systems composed of several multiprocessing nodes and tasks with synchronization requirements.

The Timix real-time operating system [143] is developed to support multisensor robot systems. Three features of the Timix real-time operating system are as follows: (i) it is possible to estimate the amount of time that a real-time process is executed since the execution times of system calls are bounded; (ii) a hierarchy of communication methods differing in synchronization, overheads and bandwidth requirements is provided to allow the programmer to choose the one most applicable to a particular application, depending on the real-time requirements; (iii) new devices, which are directly controlled by application processes, can be integrated into the system without changing the kernel.

The YARTOS real-time operating system [144] provides guaranteed response times to tasks. The YARTOS real-time operating system is distinguished by the programming model and by the usage of a novel processor scheduling and resource allocation policy.

2.4.2.3 Proprietary Implementations of Real-Time Operating Systems for Commercial Use

Some real-time operating systems are developed as commercial products. Now real-time operating systems for commercial use are introduced.

The Abassi real-time operating system [145] is configured with many features unmatched in the industry. These features including robustness and code savings are intelligent starvation protection, dynamic tracking and hybrid interrupt stack.

The AMX real-time operating system [146] meets the critical needs of the most challenging real-time applications and remains simple and easy to use. In addition, the AMX real-time operating system is compact and modular as can readily be seen from the memory size measurements.

The AVIX real-time operating system [147] is very fast, offers unprecedented interrupt handling capabilities, consumes little memory, makes application development manageable and introduces real-time insight in the application dynamics.

The ChorusOS real-time operating system [148] is very finely tuned to meet the requirements of a given application or environment. The core executive component is always present in an instance of the ChorusOS real-time operating system. Optional components provide scheduling, memory management, time management, inter-thread communication and inter-process communication.

The CMX real-time operating system [149] is designed for microprocessors, microcomputers and DSPs. The CMX real-time operating system supports nested interrupts, extremely fast context switches and very low interrupt latencies.

The DioneOS real-time operating system [150] is designated for microcontrollers. The main goal of the DioneOS real-time operating system is to improve performance. Due to this requirement, many parts of the code have been optimized.

The embOS real-time operating system [151] is designed to be used as foundation for the development of embedded real-time applications. The internal structure of the embOS real-time operating system has been optimized in a variety of applications with different customers, to fit the needs of different industries.

The Fusion real-time operating system [152] is designed and optimized for networking and media-centric processors. The Fusion real-time operating system is targeted at media applications for DSPs. Since then it has evolved into a more general real-time operating system, it has maintained all of the strengths that made it successful for DSP applications. These strengths include small footprint, objects for streaming data, very low processor overhead, stack sharing capabilities and fully integrated interrupt controls.

The iRMX real-time operating system [153] has been proven in thousands of demanding real-time applications worldwide and is highly configurable from a small-footprint and kernel-only solution to a full-service.

The MicroC/OS-II real-time operating system [154] is a highly portable, scalable and preemptive real-time multitasking kernel for microprocessors and microcontrollers.

The MQX real-time operating system [155] is a multitasking kernel with pre-emptive scheduling, fast interrupt response, extensive inter-process communication, synchronization facilities and a file system.

The OSE real-time operating system [156] is widely used in the automotive industry and the communications industry. OSE processes can be either static or dynamic; that is, created at compile-time or at run-time. Five different types of processes are supported: interrupt process, timer interrupt process, prioritized process, background process and phantom process. There are different scheduling principles for different processes: priority-based, cyclic and round-robin. The interrupt processes and the prioritized processes are scheduled according to their priorities, while timer interrupt processes are triggered cyclically. The background processes are scheduled in a round-robin fashion. The phantom processes are not scheduled at all and are used as signal redirectors.

The PikeOS real-time operating system [157] is a virtualization platform allowing to run several applications in different virtual machine together on a single hardware platform. For those applications having hard real-time requirements, the scheduling mechanism of the PikeOS real-time operating system ensures temporal and spatial deterministic.

The Portos real-time operating system [158] does not use tasks: the priority levels are directly assigned to functions. The scheduler of the Portos real-time operating system is very small and fast to save context switches and stack space.

The Q-Kernel real-time operating system [159] implements the unique micro kernel segmented interrupt architecture, called Dual-Mode. The Dual-Mode combines the traditional thread-based kernel architecture for real-time control with specialized fibers for high dataflow operations.

The QNX Neutrino real-time operating system [160] is a full-featured and robust kernel that scales down to meet the constrained resource requirements of real-time embedded systems. The micro-kernel design and modular architecture in the QUX Neutrino real-time operating system enable customers to create highly optimized and reliable systems with low total cost of ownership.

The QP real-time operating system [161] supports a highly portable, event-driven and real-time framework for concurrent execution of state machines specifically designed for real-time and embedded systems.

The ReaGOS real-time operating system [162] is a preemptive single-stack kernel. The main advantages of the ReaGOS real-time operating system are fast scheduling and small footprint. The kernel is generated automatically from a high level graphical description of the application so that much of run-time setup and lookup can be avoided.

The rt-kernel real-time operating system [163] is designed to meet the hard real-time requirements. The rt-kernel real-time operating system is well suited for automotive power

train control applications or real-time audio and video streaming applications.

The RTX real-time operating system [164] reduces the product cost of the computer platform and the operational costs of producing and supporting product, shortens the cycle time of getting new generations of products to market and increases market share by opening new markets not previously reachable and doing so profitably.

The RTXC Quadros real-time operating system [165] has scalability and extensive protocol stacks and middleware. In addition, the RTXC Quadros real-time operating system supports not only integrated and tested platforms but also design and configuration tools.

The Salvo real-time operating system [166] is designed expressly for very low-cost embedded systems with severely limited program and data memory. The Salvo real-time operating system is highly configurable and scalable with a full set of run-time features including priority-based scheduling, cooperative multitasking, event services, real-time delays and elapsed-time services.

The SCIOPTA real-time operating system [167] is specifically designed to provide excellent real-time performance and to use small memory. Internal data structures, memory management, inter-process communication and time management are highly optimized.

The Sirius real-time operating system [168] is deterministic, predictable, safe and reliable. Therefore, the Sirius real-time operating system can be successfully applied in a broad range of embedded applications in medical, telecommunication, robotics, automotive, aerospace and space industries.

The SMX real-time operating system [169] is characterized by small footprint, high performance, ease of use and integration with popular development tool suites.

The Talon DSP real-time operating system [170] meets the requirements of sophisticated embedded DSP applications providing predictable, deterministic and hard real-time behavior.

The TargetOS real-time operating system [171] is fast, small and preemptive. In order to accelerate time-to-market, the TargetOS real-time operating system is integrated with development tools and commercial-off-the-shelf board support packages.

The ThreadX real-time operating system [172] is a multitasking kernel with preemptive scheduling, fast interrupt response, memory management, inter-thread communication, mutual exclusion, event notification and thread synchronization features.

The μTasker real-time operating system [173] specifically targets smaller single-chip Internet-enabled embedded processors (with internal Ethernet controller and memory). It is small footprint but still offers comfortable development and powerful features as typically required in control type applications.

The μ-velOSity real-time operating system [174] allows developers to configure and build the microkernel to meet the specific and unique system design requirements while only including what services are absolutely needed.

The VxWorks real-time operating system [175] addresses the overhead of context switch by saving only the register windows that are actually in use by a task being switched. When the context of each task is restored, only a single register window must be restored. The other register windows are restored later at the appropriate returns from subroutine.

2.4.2.4 Proprietary Implementations of Real-Time Operating Systems for Open Source

Other real-time operating systems are usually open source. Examples of real-time operating systems for open source are Atomthreads [176], BeRTOS [177], Brazilian Real-Time Operating System [178], ChibiOS/RT [179], cocoOS [180], Embox [181], Femto OS [182],

FreeRTOS [183], FunkOS [184], Helium [185], iRTOS [186], Milos [187], OSA [188], picoOS [189], Phoenix [190], Prex [191], Real-Time Thread [192], scmRTOS [193], Small Devices Portable Operating System [194], uOS [195] and uSmartx [196].

2.4.2.5 Proprietary Implementations of Real-Time Operating Systems for Imprecise Computation

There are three ways for implementing imprecise computations. They are all suggested by Lin et al. [18].

The milestone method is used for imprecise computations with monotone optional parts. The method saves intermediate results, which the optional part is executed so that the best result is immediately available when terminating optional parts. The points to save the results are called milestones or checkpoints. The disadvantage of the milestone function is to waste the processor time between the milestone time and the termination time.

The sieve function method is used for imprecise computations, which have optional parts with 0/1 constraints. The method requires that the underlying operating system be capable of telling applications the amount of available computation time.

The multi-version method is used for imprecise computations with multiple versions. The method is implemented in programming languages [197] and is not implemented in real-time operating systems.

The Concord system presented by Lin et al. [18] supports both the milestone and sieve function methods in a client and server architecture. The Concord system provides both a programming tool so that a user can design imprecise computations and a run-time support so that imprecise results can be reliably and meaningfully returned. Such a system would give users much greater flexibility in implementing their applications. Hull et al. [198] develop the imprecise computation server, which implements monotone imprecise computations in as much as the same structure as the Concord system. Unfortunately, the Concord system in the imprecise computation model cannot guarantee the schedulability to output the results. Therefore, a real-time operating system in the practical imprecise computation model is required.

The RT-Frontier real-time operating system [70] supports the practical imprecise computation model and implements EDF-based dynamic-priority scheduling algorithms in the practical imprecise computation model such as the M-FWP and SS-OP algorithms. However, the M-FWP and SS-OP algorithms calculate the assignable time of the optional part dynamically, which causes high overhead [33].

2.4.3 Summary of Real-Time Operating Systems

Many real-time operating systems support Liu and Layland's model. However, these real-time operating systems do not support overloaded conditions. Therefore, a few real-time operating systems are developed to support the traditional imprecise computation model or the practical imprecise computation model. Unfortunately, there is no real-time operating system, which supports both imprecise computation and multiprocessor real-time scheduling.

2.5 Experimental Evaluations of Multiprocessor Real-Time Scheduling

In this section, several experimental evaluations of multiprocessor real-time scheduling in Liu and Layland's model are introduced.

Now this dissertation introduces experimental evaluations of multiprocessor real-time scheduling in the LITMUS^{RT} real-time operating system. There are several implementations of global scheduling. Brandenburg and Anderson implement several global schedulers on Sun's Niagara platform [199]. The experimental results show that a combination of a parallel heap, event-driven scheduling, and dedicated interrupt handling performed best for most workloads. Bastoni et al. evaluate the G-EDF, P-EDF and C-EDF algorithms [200] on a 24-core Intel system. The experimental results show that the overhead of scheduler in the P-EDF algorithm is approximately $10\mu s$ regardless of the number of tasks. In contrast, the overhead of scheduler in the C-EDF algorithm is slightly higher than that in the P-EDF algorithm. However, the overhead of scheduler in the G-EDF algorithm is 150-200μs for 100-350 tasks and dramatically higher than the overheads of schedulers in the P-EDF and C-EDF algorithms. In addition, semi-partitioned EDF-based dynamic-priority scheduling is often higher schedulability than other scheduling policies [201].

On the other hand, Lelli et al. present an efficient implementation of the G-EDF scheduler in Linux kernel [202]. The implementation is scalable and the overhead of scheduler in the G-EDF algorithm is very close to that of `SCHED_FIFO`. From these results, the G-EDF algorithm has comparable overhead of scheduler to the P-EDF algorithm on a 48-core AMD system.

From the above experimental results, the overhead of global scheduler depends on both hardware platform and software implementation so that the practicality of global scheduling is still an open problem.

The ChronOS real-time operating system also evaluates the overhead of the G-EDF algorithm. The average overhead of the G-EDF scheduler for 20 tasks is $15\mu s$ and the average migration overhead is $8\mu s$ [92]. The overhead of the Chron OS real-time operating system becomes high.

These Linux-based evaluation approaches suffer from the overhead including scheduler, migrations and context switches. Therefore, experimental evaluations of multiprocessor real-time scheduling in proprietary real-time operating systems are required achieving low overhead.

2.6 Summary of State of the Art

This chapter first gives three task models and introduces the advantage of the practical imprecise computation model against other task models. This chapter next shows multiprocessor real-time scheduling policies and real-time scheduling algorithms on multiprocessors. In contrast, existing real-time scheduling algorithms do not support the practical imprecise computation model. The existing real-time operating systems support these task models and implement these real-time scheduling algorithms. However, existing real-time scheduling algorithms in the practical imprecise computation model do not support multiprocessor real-time scheduling so that the effectiveness of the practical imprecise computation model on multiprocessors is unknown. They calculate the assignable time of the optional part dynamically. Unfortunately, the analysis of the assignable time of the optional part is too complex

on multiprocessors. In addition, they suffer from the overhead of such processing [33]. This dissertation claims that dynamic-priority scheduling algorithms in the practical imprecise computation model are difficult to support multiprocessors and a new priority assignment policy for practical imprecise computation model is required.

Chapter 3

System Model

This chapter defines the system model in this dissertation. Especially, this dissertation focuses on a practical imprecise computation model [33]. The practical imprecise computation model that can be applied to a diverse range of real-time applications and form the basis of diverse real-time systems is required for the imprecise computation platform. As pointed out in Chapter 2, the traditional imprecise computation model lacks the ability to incorporate compensation or recovery operations properly needed after the termination of an optional part to maintain the context of the computation in a safe stable state. Moreover, other computation models do not allow the mixture of mandatory parts and optional parts, which practical real-time applications require.

3.1 Wind-up Operation

In the traditional imprecise computation model [18], the most restrictive one is that no consideration is given on how practical imprecise computations can be terminated. The traditional imprecise computation model has no place for compensation or recovery operations needed on the termination of optional parts. Thus, as long as these computation models are assumed, even a monotone imprecise computation cannot be implemented properly as requested in practice.

Consider the following scenario to depict the problem. In a remote sensing application, there exists an imprecise computation on one node that periodically monitors the environment and sends the up-to-date data to other nodes. The role of the imprecise computation is to read sensor values, to process these raw values and to send these processed data to the other nodes. Suppose that this computation is based on the traditional imprecise computation model so that its mandatory part entirely precedes its optional part. In addition, the reward function of this computation is linearly monotone. Also suppose that the imprecise computation has the highest priority in the interval $[t, D)$, where t is the time and D is the deadline of this computation. Consider that its optional part is too long to complete before D in overloaded conditions.

Figure 3.1 shows the case with the milestone method. The imprecise computation starts to execute its mandatory part from the time t. After it has completed its mandatory part, it goes on to execute its optional part. When executing the optional part, the operating system regularly places checkpoints. At each checkpoint, the intermediate imprecise results are sent out to the destination nodes, since it will be too late to send out the result after the computation is terminated at its deadline. Thus, when the size of the data becomes large, the

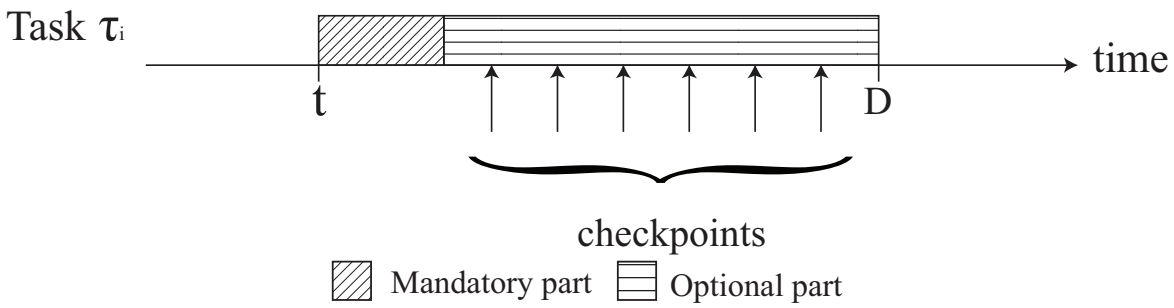


Figure 3.1: Inadequacy of milestone methods

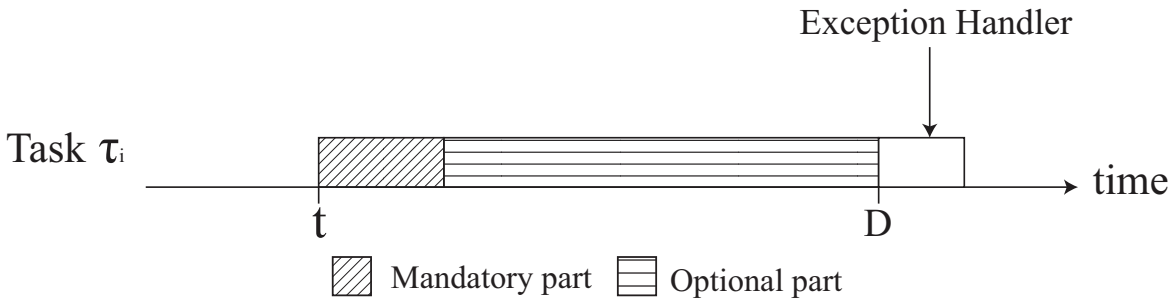


Figure 3.2: Infeasible schedule created by exception handling

overhead of the checkpoint can become large so that the performance of the computation itself is degraded. On the other hand, when the deadline expires, thanks to the checkpoints, the best imprecise result is already available in the remote nodes. However, the terminated computation and the resources that the computation used may not be in a stable state, since the computation could have been terminated at an arbitrary point. For example, the computation may not have released all the acquired locks for shared data. It is also possible that the next instance of this computation uses the progress achieved by this terminated instance as its start point. In that case, it is problematic to leave the internal data in an inconsistent state. Therefore, the milestone method is not enough to support correct imprecise computation if applications are based on the traditional imprecise computation model.

An exception handling mechanism may be superior to the milestone method in that there is no overhead at checkpoints. Unfortunately, the exception handling mechanism can lead to a deadline miss in the worst case. Suppose the same scenario of the previous example. An exception handler is invoked at the deadline instead of regularly placing checkpoints in Figure 3.2. The role of the exception handler is to send out the result to other nodes and to put all the shared data back to a consistent state. However, there are two critical problems. First of all, the exception handler is only invoked on a deadline miss. This means that the operations performed by the exception handler violate the timing constraints of the terminated imprecise computation. In particular, the result can only be sent after the original deadline, which means that the result can only arrive at the destination nodes too late. Next, if the handler is executed at the highest priority, the subsequent computations are delayed by its execution. The execution of the exception handler could lead to another deadline miss.

Considering these cases, it must be concluded that there is a mismatch between the traditional imprecise computation model and practical real-time applications. Especially, practi-

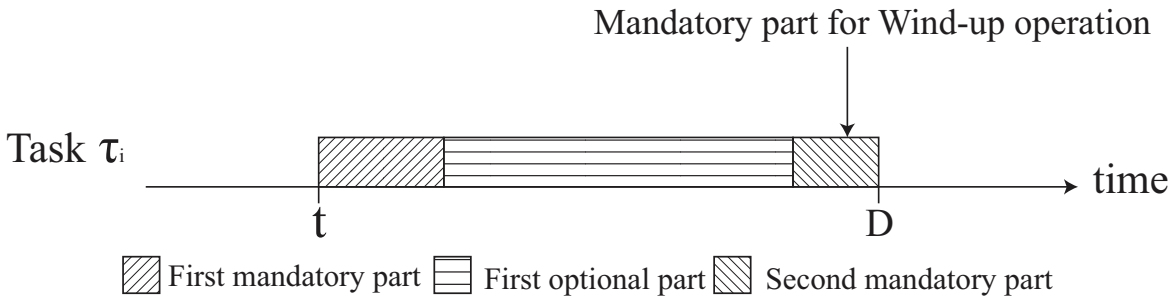


Figure 3.3: Feasible schedule in the practical imprecise computation model

cal real-time applications that allow termination in its optional part require compensation or recovery operations to allow safe termination, while the traditional imprecise computation model allows no operation for that.

This dissertation calls these compensation operations for terminated optional parts *wind-up operations*. Examples of wind-up operations are

- to return the result of computation to other computations on the same node and to those on remote nodes;
- to notify other parts of system of its premature termination;
- to release acquired locks;
- to maintain the consistency of private and shared data by completing the portions that were not executed or by rolling back prematurely terminated operations;
- to store the reward of terminated computation to provide a feedback to the scheduler to optimize its behavior dynamically.

3.2 Computation Model

The basic idea behind the practical imprecise computation model to solve the issues arising from the mismatch is to let applications have the wind-up operations included within themselves so that if appropriately modeled and scheduled, the timing constraints of wind-up operations are met without a special supporting mechanism. The major characteristics in the practical imprecise computation model that make different from the traditional imprecise computation model are as follows.

- One computation can have more than one mandatory parts and more than one optional parts.
- The mandatory and optional parts of the imprecise computation can be executed in an interleaving manner.

Using this practical imprecise computation model, the previous example can be solved by having two mandatory parts and one optional part as shown in Figure 3.3. In this case, task τ_i has two mandatory parts and one mandatory part. The first mandatory part and the first optional part is the same as those in the traditional imprecise computation model and

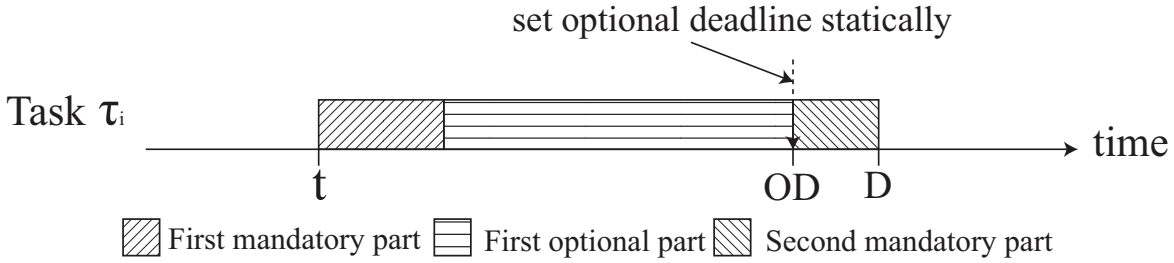


Figure 3.4: Feasible schdule with optional deadline in the practical imprecise computation model

the second mandatory part contains all the wind-up operations needed by this computation. The second mandatory part is scheduled before the deadline without termination, since its characteristics are the same as the mandatory part in the traditional imprecise computation model, except that it can follow an optional part.

An example of the practical imprecise task with two mandatory parts and one optional part is a visual-based feedback control task in autonomous mobile robots. Each part in the practical imprecise task executes the following operation.

- First mandatory part: inputs the image data from cameras.
- First optional part: analyzes objects from the image data.
- Second mandatory part: outputs the proper operation to the actuator for avoiding objects.

3.3 Optional Deadline

The M-FWP algorithm is difficult to support multiprocessors because the assignable time of the optional part is calculated dynamically. If the assignable time of the optional part is overestimated, the following mandatory part may miss a deadline due to the overrun of the optional part. Though the RM algorithm cannot be adapted to the practical imprecise computation model, a fixed-priority based assignment policy is a better choice to support multiprocessors. In order not to overrun optional parts and to start wind-up operations, each practical imprecise task has its *optional deadline*.

Figure 3.4 shows a feasible schedule with an optional deadline *OD* in the practical imprecise computation model. A dotted down arrow represents an optional deadline. An optional deadline is a time when an optional part is terminated and the following mandatory part as a wind-up operation is released. All mandatory parts except the first mandatory part are ready to be executed after each optional deadline and can be completed if each mandatory part is completed by each optional deadline. If each task executes the following mandatory part after its optional deadline, the task may miss its deadline. Each optional deadline is set to the time as late as possible to expand the executable range of each optional part. Not all mandatory parts of each practical imprecise task must miss its deadline if there is an idle processor or lower priority tasks are executed between the time when the mandatory part is completed and the time when the following mandatory part is released.

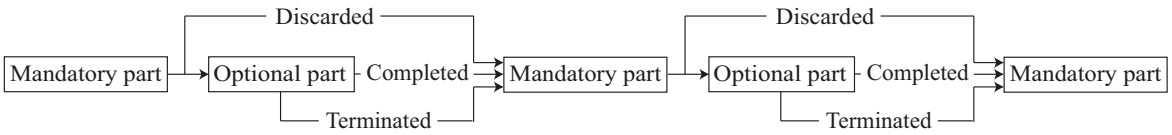


Figure 3.5: Linear task in the practical imprecise computation model

3.4 Linear Task Model

All tasks are expressed by a linear task model. A linear task does not have any branching point in its logical structure when a mandatory part or an optional part is considered as a single basic block. A branching point is defined as a mandatory or an optional part that has more than one successors that can be selectively executed. An example of a linear task in the practical imprecise computation model is shown in Figure 3.5. This linear task has three mandatory parts and two optional parts. The purpose of using this linear task model is to enable the feasibility assessment of given task sets in a practical overhead. It must be noted that this linear constraint does not necessarily forbid a computation to have multiple execution paths. Multiple execution paths within a part are allowed as long as there is only one path between all consecutive parts.

An example of the practical imprecise task with three mandatory parts and two optional parts is a lip reading task by a sensor fusion of a camera and a microphone [203]. Each part in the practical imprecise task executes the following operation.

- First mandatory part: inputs the visual information from a camera. There are some metrics to read the lip: the width of the lip, the height of the lip and the variation of the height of the lip. In the first mandatory part, one of the metrics is executed to generate the result with low quality.
- First optional part: performs other metrics except the metric performed in the first mandatory part. More metrics are performed in the first optional part, the result has higher quality.
- Second mandatory part: inputs the auditory information from a microphone. There are also some metrics to analyze what a person speaks: log power spectrum and LPC-derived mel-cepstrum. In the second mandatory part, one of the metrics is also executed to generate the result with low quality.
- Second optional part: also performs other metrics except the metric performed in the second mandatory part.
- Third mandatory part: performs the fusion of the visual and auditory information by neural network [204] or hidden markov model [205].

This dissertation assumes that the system has M identical processors and a task set Γ consisted of n periodic tasks with implicit deadlines. Task τ_i is represented as the following tuple $(T_i, D_i, OD_i, m_i, o_i)$: where T_i is the period, D_i is the relative deadline, OD_i is the relative optional deadline, m_i is the total WCET of the mandatory part and o_i is the total RET of the optional part. The total WCET of mandatory parts of task τ_i is $m_i = \sum_{l=1}^{n_i^m} m_i^l$, where n_i^m is the number of mandatory parts and m_i^l is the WCET of the l^{th} mandatory part. On the other hand, the total RET of optional parts is $o_i = \sum_{l=1}^{n_i^o} o_i^l$, where n_i^o is the number of optional parts

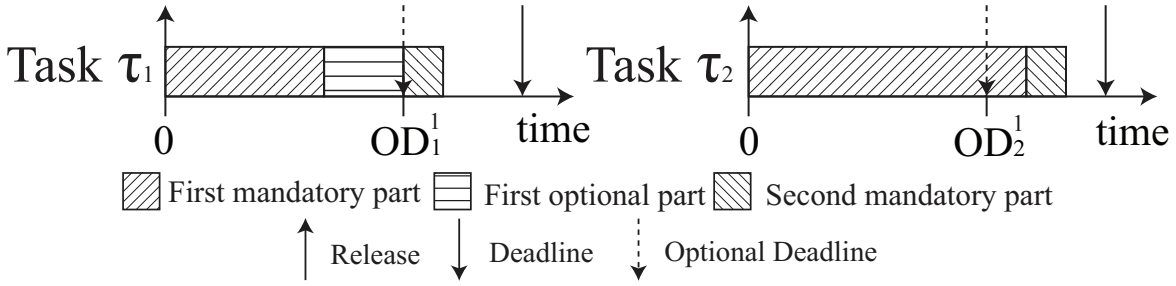


Figure 3.6: Behavior of optional deadline

and o_i^l is the RET of the l^{th} mandatory part. For example, as shown in Figure 3.5, task τ_i has three mandatory parts and two optional parts so that $n_i^m = 3$ and $n_i^o = 2$. The RET of each optional part tends to be underestimated or overestimated from time to time because this dissertation assumes that target real-time applications run in dynamic environments. The relative optional deadline of task τ_i is $OD_i = \sum_{l=1}^{n_i^o} OD_i^l$. An optional deadline performs to terminate an optional part so that the number of optional deadlines is equal to that of optional parts. The relative deadline D_i of each task τ_i is equal to its period T_i. The j^{th} instance of task τ_i is called job $\tau_{i,j}$. The utilization of each periodic task is defined as $U_i = m_i/T_i$. The reason why U_i does not include o_i is because the optional part of task τ_i is a non-real-time part so that completing it is not relevant to scheduling the task set successfully. Hence, the system utilization within n tasks can be defined as $U_s = \sum_{i=1}^{n} U_i/M$. All tasks are ordered by increasing their periods and task τ_1 has the shortest period.

In order to simplify the schedulability analysis, the following hypotheses are assumed on the tasks.

- All tasks in the task set Γ are independent. That is to say, there is no precedence relation and no resource constraint between practical imprecise tasks. However, there are precedence relations in each practical imprecise task, as shown in Figure 3.5.
- No task can suspend itself, for example on I/O operations.
- All tasks are released as soon as they arrive.
- All overheads in the kernel including context switches and migrations are assumed to be 0.
- Each job of the task is not allowed to be executed simultaneously.

Figure 3.6 shows a behavior of an optional deadline. Each task has two mandatory parts and one optional part. In this behavior, task τ_1 can execute the first mandatory part after the first optional deadline OD_1^1 and task τ_2 cannot execute the first optional part after the first optional deadline OD_2^1. Task τ_1 completes the first mandatory part by the first optional deadline OD_1^1 and executes the first optional part until the first optional deadline OD_1^1. After the first optional deadline OD_1^1, then task τ_1 executes the second mandatory part. In contrast, task τ_2 does not complete the first mandatory part by the first optional deadline OD_2^1. When task τ_2 completes the first mandatory part, task τ_2 executes the second mandatory part and does not execute the first optional part.

In addition, this dissertation defines the following symbols as follows.

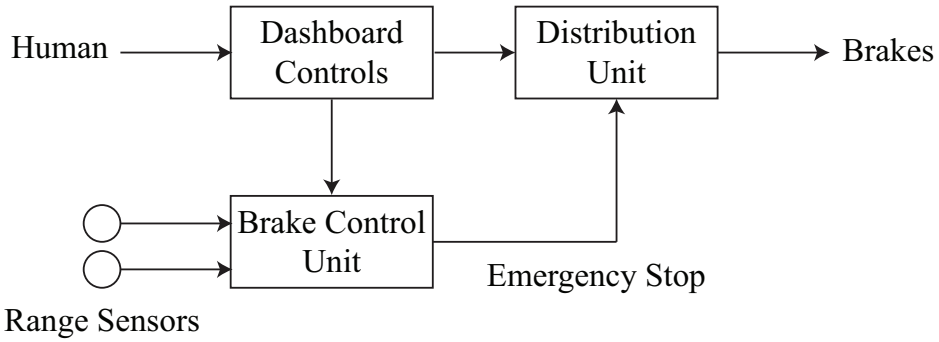


Figure 3.7: Block diagram of an automatic braking system

- $o^l_{i,j}$: the actual case RET of the l^{th} optional part of job $\tau_{i,j}$
- $r_{i,j}$: the release time of job $\tau_{i,j}$
- $s^l_{i,j}$: the start time of the l^{th} mandatory part of job $\tau_{i,j}$
- $R_i(t)$: the remaining execution time of task τ_i at the time t
- H_k: the hyperperiod of the k^{th} task set, which is the minimum interval of time after which the schedule repeats itself. If the hyperperiod H_k is the length of such an interval, then the schedule in $[0, H)$ is the same as that in $[xK, (x + 1)K)$ for any integer $x > 0$. For a set of periodic tasks synchronously activated at the time $t = 0$, the hyperperiod H_k is given by the least common multiple of the periods $H_k = lcm(T_1, T_2, ..., T_n)$.

3.5 Jitter

Real-time applications running in dynamic environments such as autonomous mobile robots require getting the information of the environments from sensors periodically so that the interval of each input processing is constant as much as possible without missing a deadline.

In order to explain the importance of each input timing, this dissertation introduces a wheel-vehicle by Buttazzo [206]. The wheel-vehicle is equipped with range sensors and must operate in a certain environment running within a maximum given speed. The wheel-vehicle could be a completely autonomous system such as an autonomous mobile robot or a partially autonomous system driven by a human, such as a car or a train having an automatic braking system for stopping motion in emergency situations.

In order to simplify discussion and reduce the number of controlled variables, a wheel-vehicle is considered like a train, which moves along a straight line and suppose that there an automatic braking system able to detect obstacles in front of the wheel-vehicle must be designed and the brakes must be controlled to avoid collisions.

Figure 3.7 shows the block diagram of the automatic braking system. The Brake Control Unit (BCU) is responsible for acquiring a pair of range sensors, computing the distance of the obstacle, reading the state variables of the wheel-vehicle from instruments on the dashboard and deciding whether an emergency stop must be superimposed. Given the criticality of the braking action, this task must be periodically executed on the BCU. Let T be its period.

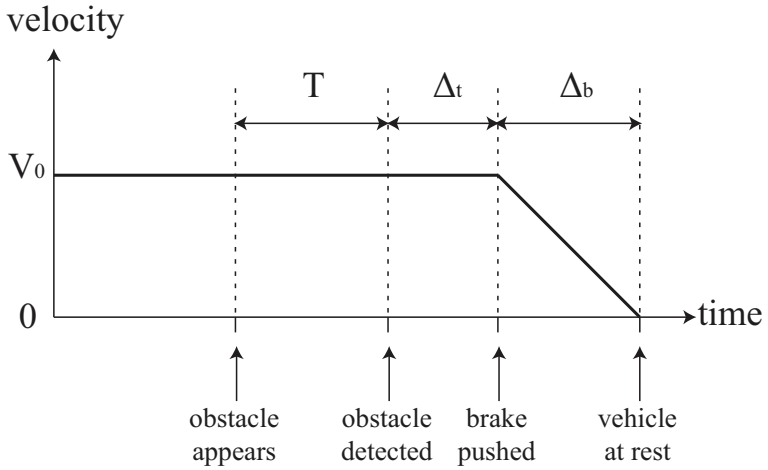


Figure 3.8: Velocity during brake

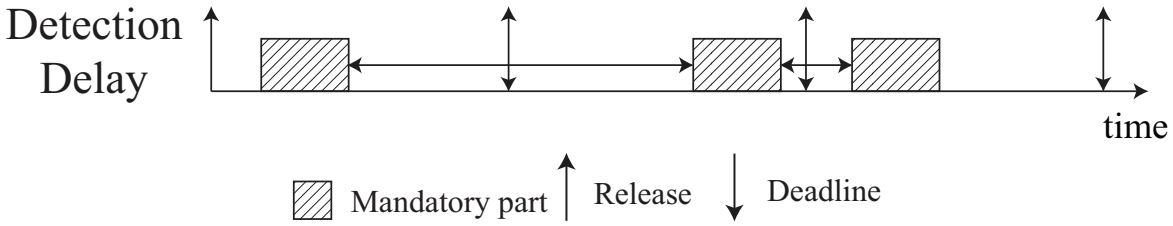


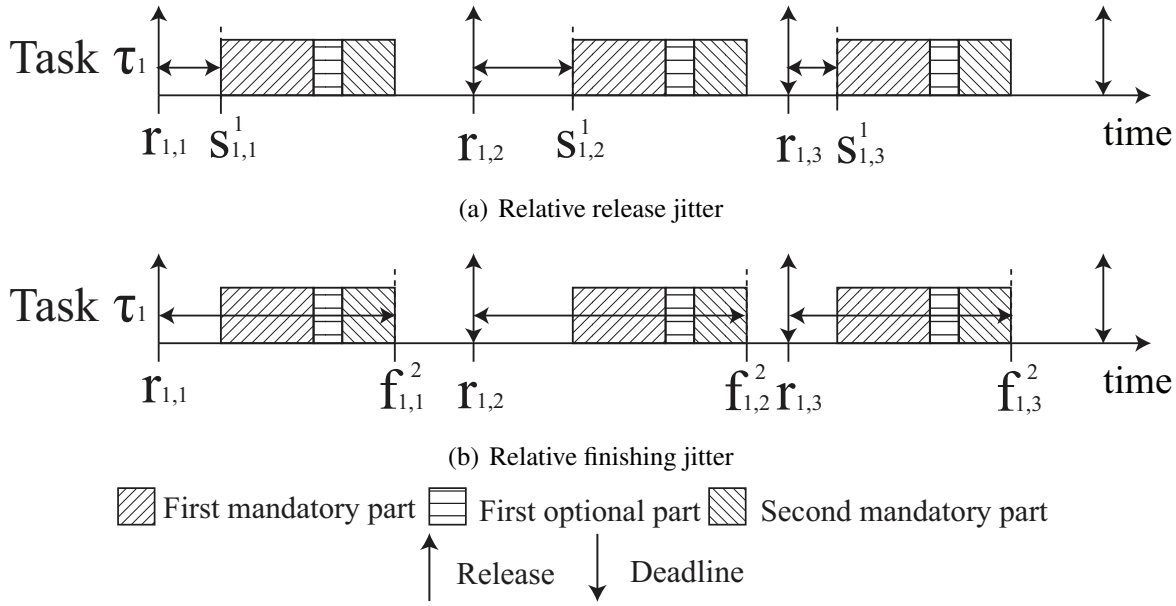
Figure 3.9: Detection delay

In order to determine a safe value for T, several factors must be considered. In particular, the system must ensure that the maximum latency from the time when an obstacle appears and the time when the wheel-vehicle reaches a complete stop is less than the time to impact. Equivalently, the distance of the obstacle from the wheel-vehicle must always be greater than the minimum space needed for a complete stop. Figure 3.8 shows the velocity of the wheel-vehicle as a function of time when an emergency stop is performed. The initial velocity at the time 0 is V_0.

Three time intervals must be taken in to account to compute the worst case latency.

- The detection delay, which is the interval between the time when an obstacle appears and the time when the obstacle is detected by the BCU.
- The transmission delay Δ_t, which is the interval between the time at which the stop command is activated by the BCU and the time at which the command starts to be actuated by the brakes.
- The braking delay Δ_b, which is the interval needed for a complete stop.

This dissertation focuses on the detection delay. Figure 3.9 shows the execution of the detection delay task. The interval of each detection delay is at most $2T$. If the interval of each detection delay is fluctuated, the obstacle detection task may miss obstacles and robots may crash into obstacles. In order to achieve precise control in such wheel-vehicle, the interval of each task must be constant without timing constraints. In addition, there are



(a) Relative release jitter

(b) Relative finishing jitter

Figure 3.10: Jitter

many metrics to measure the stability of the interval, called *jitter*. One of the examples to measure the jitter is the differential of the relative start and finishing time from the release time of each job. For example, the first, second and third differentials can be used to measure the jitter. However, these differentials are difficult to measure jitter of each task. In order to simplify the measure of the jitter, this dissertation defines jitter as both Relative Release Jitter (RRJ) and Relative Finishing Jitter (RFJ) of each practical imprecise task.

RRJ is defined as the maximum deviation of the start time of two consecutive jobs:

$$RRJ_i = \max_j |(s^1_{i,j+1} - r_{i,j+1}) - (s^1_{i,j} - r_{i,j})|, \tag{3.1}$$

where $s^1_{i,j}$ is the start time of the first mandatory part of job $\tau_{i,j}$ and $r_{i,j}$ is the release time of job $\tau_{i,j}$.

RFJ is defined as the maximum deviation of the finishing time of two consecutive jobs:

$$RFJ_i = \max_j |(f^{n^m_i}_{i,j+1} - r_{i,j+1}) - (f^{n^m_i}_{i,j} - r_{i,j})|, \tag{3.2}$$

where $f^{n^m_i}_{i,j}$ is the finishing time of the last mandatory part of job $\tau_{i,j}$, then n^m_i is the number of mandatory parts of task τ_i and $r_{i,j}$ is the release time of job $\tau_{i,j}$.

Figure 3.10 shows the RRJ and RFJ of task τ_1 with two mandatory parts and one optional part. In Figure 3.10(a), the RRJ of task τ_1 is the maximum of $|(s^1_{1,2} - r_{1,2}) - (s^1_{1,1} - r_{1,1})|$ and $|(s^1_{1,3} - r_{1,3}) - (s^1_{1,2} - r_{1,2})|$. In Figure 3.10(b), the RFJ of task τ_1 is the maximum of $|(f^2_{1,2} - r_{1,2}) - (f^2_{1,1} - r_{1,1})|$ and $|(f^2_{1,3} - r_{1,3}) - (f^2_{1,2} - r_{1,2})|$.

Chapter 4

Semi-Fixed-Priority Scheduling

This dissertation proposes a new priority assignment policy for practical imprecise computation, called *semi-fixed-priority scheduling*. This chapter starts describing the basic strategy of semi-fixed-priority scheduling. Then, three semi-fixed-priority scheduling algorithms for uniprocessor scheduling, multiprocessor global scheduling and multiprocessor partitioned scheduling are proposed.

4.1 Basic Strategy

Semi-fixed-priority scheduling is defined as part-level fixed-priority scheduling. That is to say, semi-fixed-priority scheduling fixes the priority of each part in the practical imprecise task and changes the priority of each practical imprecise task only in the two cases: (i) when the practical imprecise task completes its mandatory part and executes its following optional part; (ii) when the practical imprecise task terminates or completes its optional part and executes the following mandatory part. Also, semi-fixed-priority scheduling splits one practical imprecise task into several general tasks in Liu and Layland's model. The split general tasks have same periods and same or different release times, are not allowed to be executed simultaneously and are scheduled by fixed-priority as shown in Figure 4.1. In this case, the practical imprecise task has two mandatory parts. Tasks τ_i^1 and τ_i^2 are represented as the first mandatory part and the second mandatory part of task τ_i. The release times of tasks τ_i^1 and τ_i^2 are 0 and OD_i^1 respectively. When there is no task which is ready to execute its mandatory parts, each task executes its optional parts.

Figure 4.2 shows the difference between general scheduling in Liu and Layland's model and semi-fixed-priority scheduling in this dissertation's model. In general scheduling, when task τ_i is released at the time 0, then the remaining execution time $R_i(t)$ is set to $m_i^1 + m_i^2$ and monotonically decreasing until the remaining execution time $R_i(t)$ becomes 0 at the time $m_i^1 + m_i^2$. In semi-fixed-priority scheduling, when task τ_i is released at the time 0, then $R_i(t)$ is set to m_i and monotonically decreasing until the remaining execution time $R_i(t)$ becomes 0 at the time m_i^1. When the remaining execution time $R_i(t)$ is 0 at the time m_i^1, then task τ_i sleeps until the optional deadline OD_i^1. When task τ_i is released at the optional deadline OD_i^1, then the remaining execution time $R_i(t)$ is set to m_i^2 and monotonically decreasing until the remaining execution time $R_i(t)$ becomes 0 at the time $OD_i^1 + m_i^2$. If task τ_i does not complete its mandatory part by the optional deadline OD_i^1, then the remaining execution time $R_i(t)$ is set to m_i^2 at the time when task τ_i completes its mandatory part. In both cases, task τ_i completes the second mandatory part by the deadline D_i.

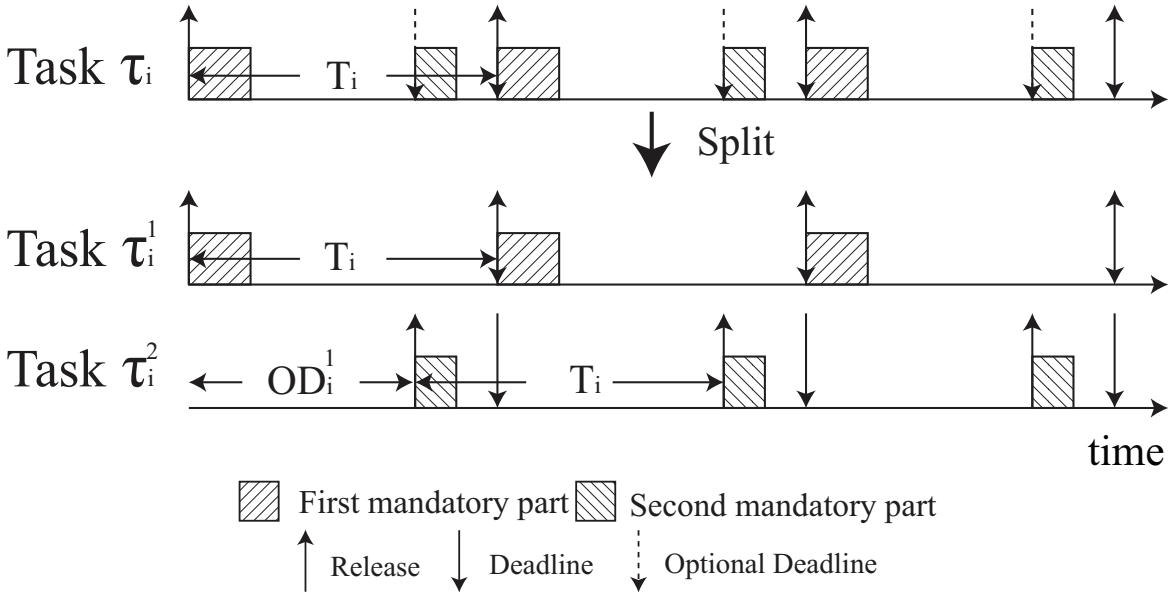


Figure 4.1: Split one practical imprecise task into two general tasks

This dissertation first proposes a semi-fixed-priority scheduling algorithm on uniprocessors. This dissertation next extends the semi-fixed-priority scheduling algorithm for global and partitioned scheduling on multiprocessors.

4.2 The RMWP Algorithm

The Rate Monotonic with Wind-up Part (RMWP) algorithm is one of semi-fixed-priority scheduling algorithms in the practical imprecise computation model on uniprocessors. As shown in Figure 4.3, the RMWP algorithm manages three task queues: Real-Time Queue (RTQ), Non-Real-Time Queue (NRTQ) and Sleep Queue (SQ). The RTQ holds tasks, which are ready to execute their mandatory parts in the RM order. One task is not allowed to execute their mandatory parts simultaneously. The NRTQ holds tasks, which are ready to execute their optional parts in the RM order. Every task in the RTQ has higher priority than that in the NRTQ. The SQ holds tasks which complete their optional parts by their optional deadlines or their last mandatory parts by their deadlines. Figure 4.4 shows the RMWP algorithm. The RMWP algorithm executes each scheduling event when the following conditions are met: (1) task τ_i becomes ready at its release time; (2) task τ_i completes its mandatory part; (3) task τ_i completes its optional part; (4) the l^{th} optional deadline OD_i^l expires; (5) there are one or multiple tasks in the RTQ; (6) there is no task in the RTQ and there are one or multiples tasks in the NRTQ. This dissertation next describes how to calculate the optional deadline and analyze the schedulability of the RMWP algorithm.

4.2.1 Optional Deadline of the RMWP Algorithm

An optional deadline is a time when optional parts are terminated and the following mandatory parts are released. The following mandatory parts are ready to be executed after the optional deadlines and can be completed if each mandatory part is completed by the optional deadline. Each optional deadline is set to the time as late as possible to expand the

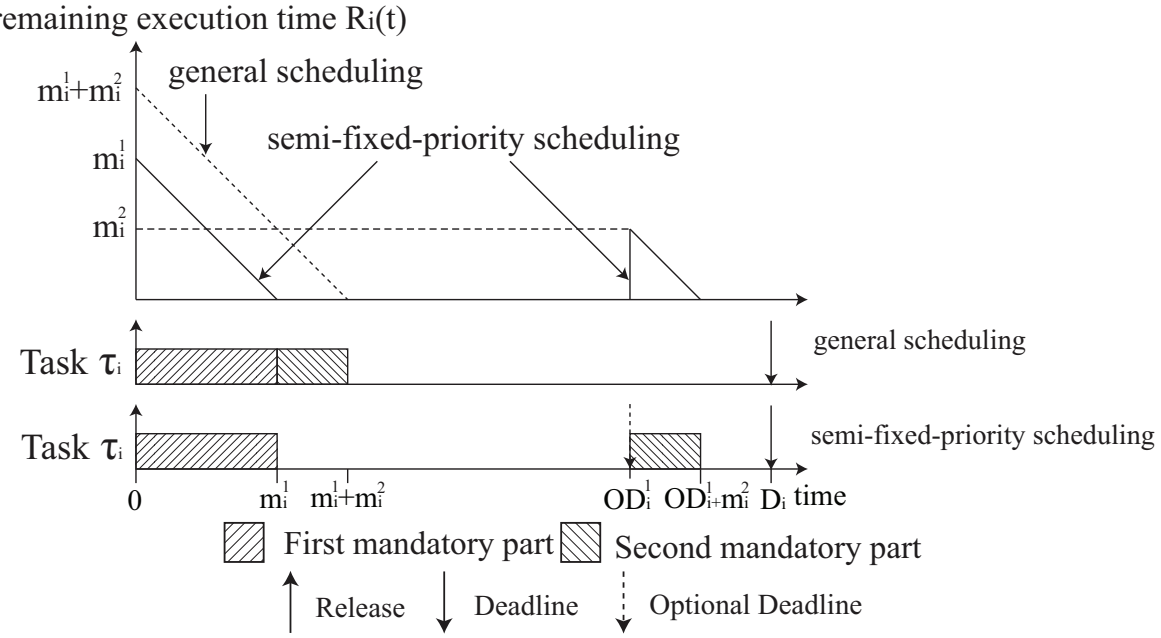


Figure 4.2: General scheduling and semi-fixed-priority scheduling

executable range of each optional part. The following mandatory part of each task must not miss the deadline if there is an idle processor or lower priority tasks are executed between the time when the mandatory part is completed and the following mandatory part is released. In order to calculate the optional deadline, this dissertation first analyzes the worst case interference time $I_k^i (i < k)$ which is the upper bound when task τ_k is interfered by higher priority tasks τ_i.

Theorem 1 (Worst case interference time by higher priority tasks in the RMWP algorithm)**.** *The worst case interference time $I_k^i (i < k)$ which is the upper bound when task τ_k is interfered by higher priority tasks τ_i in the RMWP algorithm is*

$$I_k^i = \left\lceil \frac{T_k}{T_i} \right\rceil m_i. \tag{4.1}$$

Proof. If all optional deadlines of each task are equal to 0, as shown in Figure 4.5, the interference time of task τ_k interfered by higher priority tasks $\tau_i (i < k)$ is equal to Equation (4.1), regardless of the number of mandatory parts. Moreover, there is no case that the interference time of task τ_k interfered by higher priority tasks τ_i is more than Equation (4.1). □

By Theorem 1, this dissertation next calculates the relative optional deadlines of each task in the RMWP algorithm.

Theorem 2 (Optional deadline in the RMWP algorithm)**.** *The l^{th} relative optional deadline OD_k^l of task τ_k in the RMWP algorithm is*

$$OD_k^l = \begin{cases} \max(0, D_k - m_k^{n_k^m} - \sum_{i<k} I_k^i) & (l = n_k^o) \\ \max(0, OD_k^{l+1} - m_k^{l+1} - o_k^{l+1}) & (l < n_k^o). \end{cases} \tag{4.2}$$

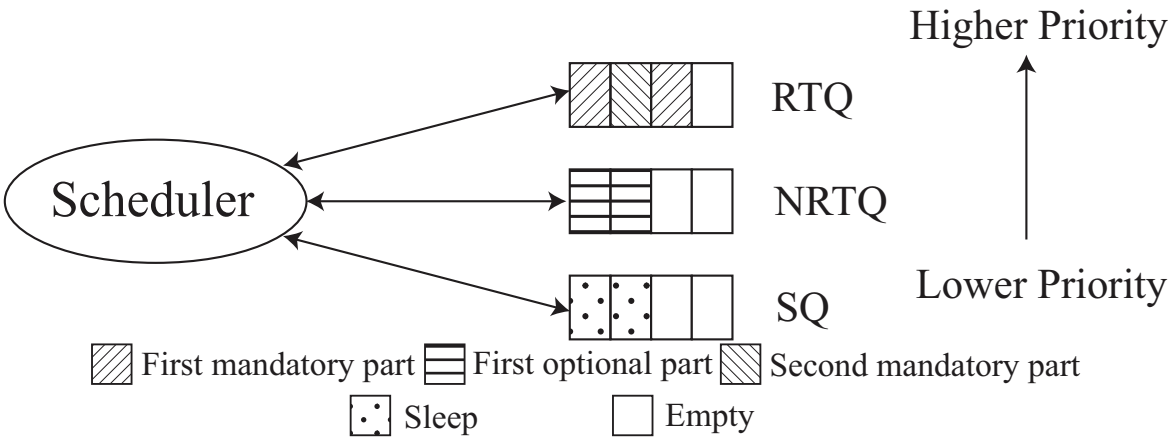


Figure 4.3: Task queue

Table 4.1: Task set A

Task	T_i	D_i	OD_i^1	m_i^1	m_i^2	o_i^1
τ_1	10	10	7	3	3	1
τ_2	15	15	1	3	2	1

Proof. It is clear that if task τ_k completes its mandatory part by its optional deadline by Theorem 1, task τ_k completes the following mandatory part by its following optional deadline or deadline. □

The RMWP algorithm can calculate optional deadlines by Theorem 2. In contrast, in the M-FWP algorithm, analyzing that what job maximizes the worst case interference time I_k^i is too complex due to dynamic-priority scheduling. Therefore, in order to calculate the optional deadline OD_k easily, the RMWP algorithm is a semi-fixed-priority scheduling algorithm and only considers tasks, which have higher priority than task τ_k. The RMWP algorithm can delay the release time of the following mandatory part until the time when the following mandatory part does not miss the deadline if the mandatory part is completed by the optional deadline.

Figure 4.6 shows an example of schedule generated by the RMWP and RM algorithms. Table 4.1 shows the task set scheduled by the RMWP and RM algorithms in Figures 4.6(a) and 4.6(b) respectively. Each practical imprecise task has two mandatory parts and one optional part. Each optional deadline is calculated by Theorem 2. This example shows that there is at least one task set, which is schedulable by the RMWP algorithm and is not schedulable by the RM algorithm. Moreover, in the RMWP algorithm, job $\tau_{1,1}$ and $\tau_{1,2}$ can execute its optional part in $[14, 15)$ and $[26, 27)$ respectively.

4.2.2 Schedulability Analysis of the RMWP Algorithm

Now this dissertation analyzes the schedulability of the RMWP algorithm.

Theorem 3 (The RMWP algorithm is at least as effective as the RM algorithm)**.** *One task set is schedulable by the RMWP algorithm if the task set is schedulable by the RM algorithm.*

1. When task τ_i becomes ready at its release time, set the remaining execution time $R_i(t)$ to m_i^1, dequeue task τ_i from the SQ and enqueue task τ_i to the RTQ. If task τ_i has the highest priority in the RTQ, preempt the current task.
2. When task τ_i completes the l^{th} mandatory part:
   (a) If the l^{th} mandatory part is the last mandatory part in each job, dequeue task τ_i from the RTQ and enqueue task τ_i to the SQ.
   (b) If the l^{th} optional deadline OD_i^l expired, set the remaining execution time $R_i(t)$ to m_i^{l+1}.
   (c) Otherwise set $R_i(t)$ to the RET of the optional part o_i^l, dequeue task τ_i from the RTQ and enqueue task τ_i to the NRTQ. If there are one or multiple tasks in the RTQ or the NRTQ which have higher priority than task τ_i, preempt task τ_i.
3. When task τ_i completes its optional part, dequeue task τ_i from the NRTQ and enqueue task τ_i to the SQ.
4. When the l^{th} optional deadline OD_i^l expires:
   (a) If task τ_i is in the RTQ and does not complete the l^{th} mandatory part, do nothing.
   (b) If task τ_i is in the NRTQ, terminate and dequeue task τ_i from the NRTQ, set the remaining execution time $R_i(t)$ to m_i^l and enqueue task τ_i to the RTQ. If task τ_i has the highest priority in the RTQ, preempt the current task.
   (c) If task τ_i is in the SQ, dequeue task τ_i from the SQ, set the remaining execution time $R_i(t)$ to m_i^l and enqueue task τ_i to the RTQ.
5. When there are one or multiple tasks in the RTQ, perform the RM algorithm in the RTQ.
6. When there is no task in the RTQ and there are one or multiples tasks in the NRTQ, perform the RM algorithm in the NRTQ.

Figure 4.4: The RMWP algorithm

Proof. This proof is shown by contraposition. This dissertation shows that if one task set is not schedulable by the RMWP algorithm, the task set is not schedulable by the RM algorithm. By Theorem 2, it is clear that task τ_i completes its all mandatory parts by its deadline if task τ_i completes its all mandatory parts except the first mandatory part by its all optional deadlines respectively. Task τ_i misses its deadline only if task τ_i executes its mandatory parts after its optional deadlines. In this case, task τ_i executes its all mandatory parts continuously without executing its all optional parts. In the RM algorithm, task τ_i also misses its deadline because of executing its all mandatory parts continuously. Hence, this theorem holds. □

Theorem 4 (Least upper bound of the RMWP algorithm)**.** *For a set of n tasks with semi-fixed-priority assignment, the least upper bound of the RMWP algorithm on uniprocessors is* $U_{lub} = n(2^{1/n} - 1)$.

Proof. The RMWP algorithm is at least as effective as the RM algorithm by Theorem 3 and generates the same schedule as the RM algorithm in the case of worst case interference time by Theorem 1. Therefore, the least upper bound of the RMWP algorithm on uniprocessors is equal to that of the RM algorithm [2]. □

Theorem 4 shows the least upper bound of the RMWP algorithm for n tasks. Thanks to Theorem 4, the least upper bound of the RMWP algorithm for two tasks on uniprocessors is not so important so that this theorem is shown in Appendix A.

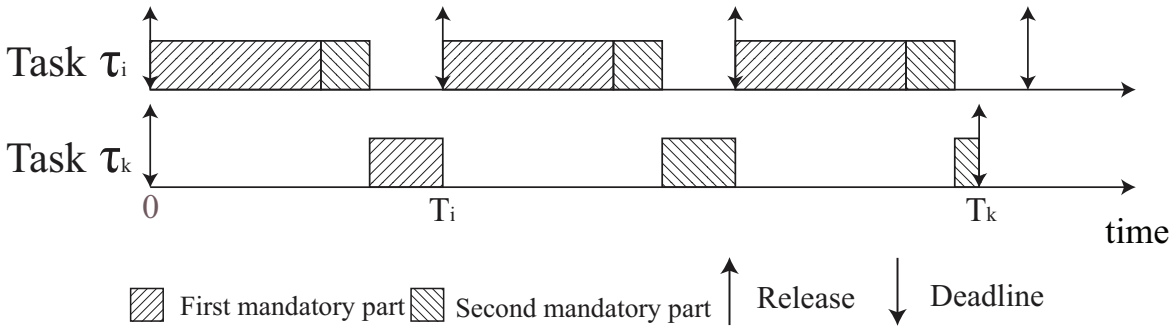


Figure 4.5: Case of worst case interference time

4.3 The G-RMWP Algorithm

The Global Rate Monotonic with Wind-up Part (G-RMWP) algorithm is based on and extends the RMWP algorithm for global scheduling on multiprocessors. Figure 4.7 shows the overall of the G-RMWP algorithm. Like the RMWP algorithm, the G-RMWP algorithm executes each scheduling event when the following conditions are met: (1) task τ_i becomes ready at its release time; (2) task τ_i completes its mandatory part; (3) task τ_i completes its optional part; (4) the l^{th} optional deadline OD_i^l expires; (5) there are one or multiple tasks in the RTQ; (6) the number of tasks in the RTQ is less than that of processors and there are one or multiples tasks in the NRTQ. The primary difference between the G-RMWP and RMWP algorithms is events (5) and (6), which assign ready tasks to multiprocessors in the G-RM order. This dissertation next shows how to calculate optional deadlines in the G-RMWP algorithm.

4.3.1 Optional Deadline of the G-RMWP Algorithm

First this dissertation calculates the worst case interference time of each practical imprecise task in the G-RMWP algorithm using RTA [38] for the G-RM algorithm. Next this dissertation calculates the relative optional deadlines of each task in the G-RMWP algorithm.

Bertogna and Cirinei showed RTA with slack for the G-RM algorithm [207]. This RTA can be expressed in the following fixed-point iteration on the upper bound R_k^{ub}.

$$R_k^{ub} \leftarrow m_k + \left\lfloor \frac{1}{M} \sum_{i<k} I_i^{CI}(R_k^{ub}) \right\rfloor, \tag{4.3}$$

where $I_i^{CI}(R_k^{ub})$ is the worst case interference time due to task τ_i within the worst case response time of task τ_k given by:

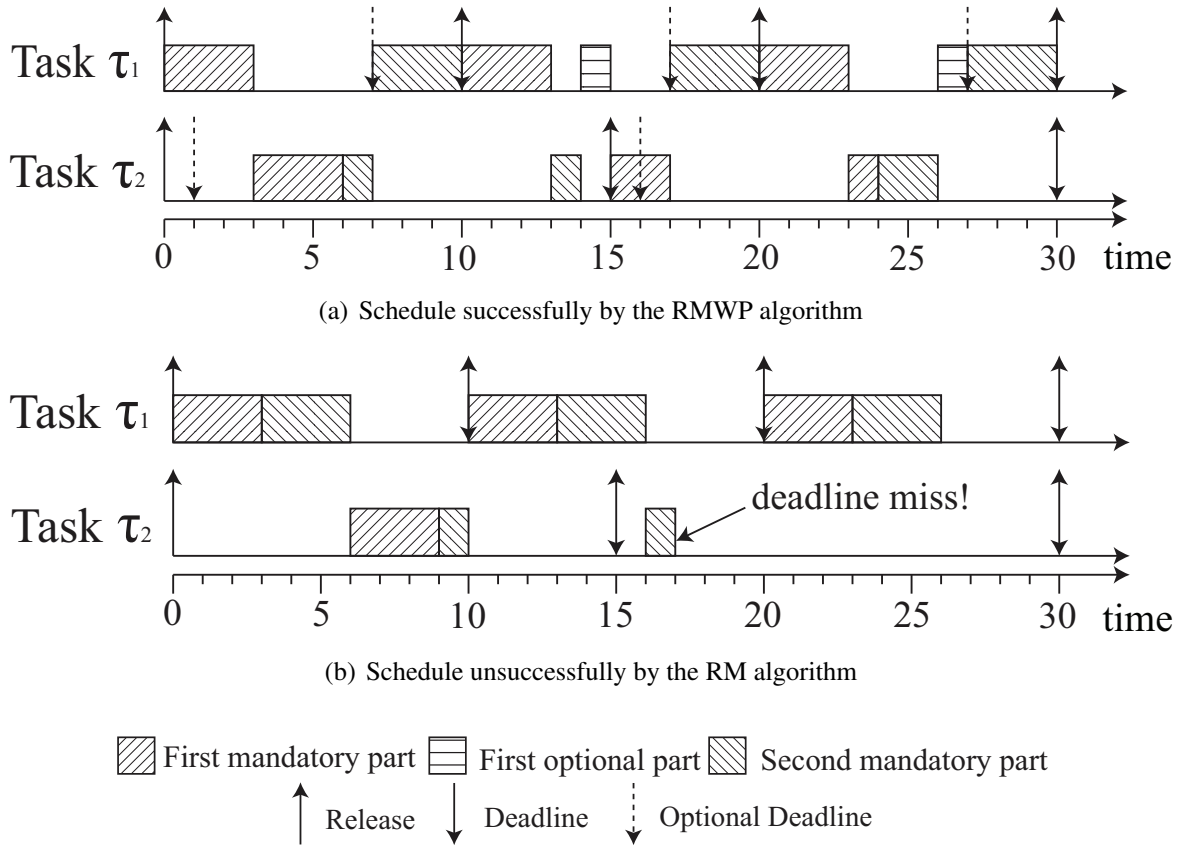
$$I_i^{CI}(R_k^{ub}) = \min(W_i^{CI}(R_k^{ub}), R_k^{ub} - m_k + 1), \tag{4.4}$$

where $W_i^{CI}(R_k^{ub})$ is the worst case workload of task τ_i in an interval of the length L given by:

$$W_i^{CI}(L) = N_i^{CI}(L)m_i + \min(m_i, L + R_i^{ub} - m_i - N_i^{CI}(L)T_i), \tag{4.5}$$

where $N_i^{CI}(L)$ is the maximum number of jobs of task τ_i that contributes all of their execution time in the interval given by:

$$N_i^{CI}(L) = \left\lfloor \frac{L + R_i^{ub} - m_i}{T_i} \right\rfloor. \tag{4.6}$$



(a) Schedule successfully by the RMWP algorithm

(b) Schedule unsuccessfully by the RM algorithm

Figure 4.6: Example of schedule generated by the RMWP and RM algorithms

Guan et al. improved the precision of RTA for the G-RM algorithm against Equation (4.3) [208] using Baruah's window analysis framework [209]. They showed that the refined worst case interference time $I_i^{NC}(R_k^{ub})$ is

$$I_i^{NC}(R_k^{ub}) = \min(W_i^{NC}(R_k^{ub}), R_k^{ub} - m_k + 1), \tag{4.7}$$

where

$$W_i^{NC}(L) = N_i^{NC}(L)m_i + \min(m_i, L - N_i^{NC}(L)T_i) \tag{4.8}$$

$$N_i^{NC}(L) = \left\lfloor \frac{L}{T_i} \right\rfloor. \tag{4.9}$$

The difference between $I_i^{CI}(R_k^{ub})$ and $I_i^{NC}(R_k^{ub})$ is

$$I_i^{DIFF}(R_k^{ub}) = I_i^{CI}(R_k^{ub}) - I_i^{NC}(R_k^{ub}). \tag{4.10}$$

Using this result, the refined RTA for the G-RM algorithm can be expressed in the following fixed-point iteration on the refined upper bound R_k^{ub}:

$$R_k^{ub} \leftarrow m_k + \hat{I}_k, \tag{4.11}$$

1. When task τ_i becomes ready at its release time, set the remaining execution time $R_i(t)$ to m_i^1, dequeue task τ_i from the SQ and enqueue task τ_i to the RTQ. If task τ_i has higher priority than the running task with the lowest priority, preempt this running task.
2. When task τ_i completes the l^{th} mandatory part:
   (a) If the l^{th} mandatory part is the last mandatory part in each job, dequeue task τ_i from the RTQ and enqueue task τ_i to the SQ.
   (b) If the l^{th} optional deadline OD_i^l expired, set the remaining execution time $R_i(t)$ to m_i^{l+1}.
   (c) Otherwise set $R_i(t)$ to the RET of the optional part o_i^l, dequeue task τ_i from the RTQ and enqueue task τ_i to the NRTQ. If there are one or multiple tasks in the RTQ or the NRTQ which have higher priority than task τ_i, preempt task τ_i.
3. When task τ_i completes its optional part, dequeue task τ_i from the NRTQ and enqueue task τ_i to the SQ.
4. When the l^{th} optional deadline OD_i^l expires:
   (a) If task τ_i is in the RTQ and does not complete the l^{th} mandatory part, do nothing.
   (b) If task τ_i is in the NRTQ, terminate and dequeue task τ_i from the NRTQ, set the remaining execution time $R_i(t)$ to m_i^l and enqueue task τ_i to the RTQ. If task τ_i has higher priority than the running task with the lowest priority, preempt this running task.
   (c) If task τ_i is in the SQ, dequeue task τ_i from the SQ, set the remaining execution time $R_i(t)$ to m_i^l and enqueue task τ_i to the RTQ.
5. When there are one or multiple tasks in the RTQ, perform the G-RM algorithm in the RTQ.
6. When the number of tasks in the RTQ is less than that of processors and there are one or multiples tasks in the NRTQ, perform the G-RM algorithm in the NRTQ on the remaining processors.

Figure 4.7: The G-RMWP algorithm

where

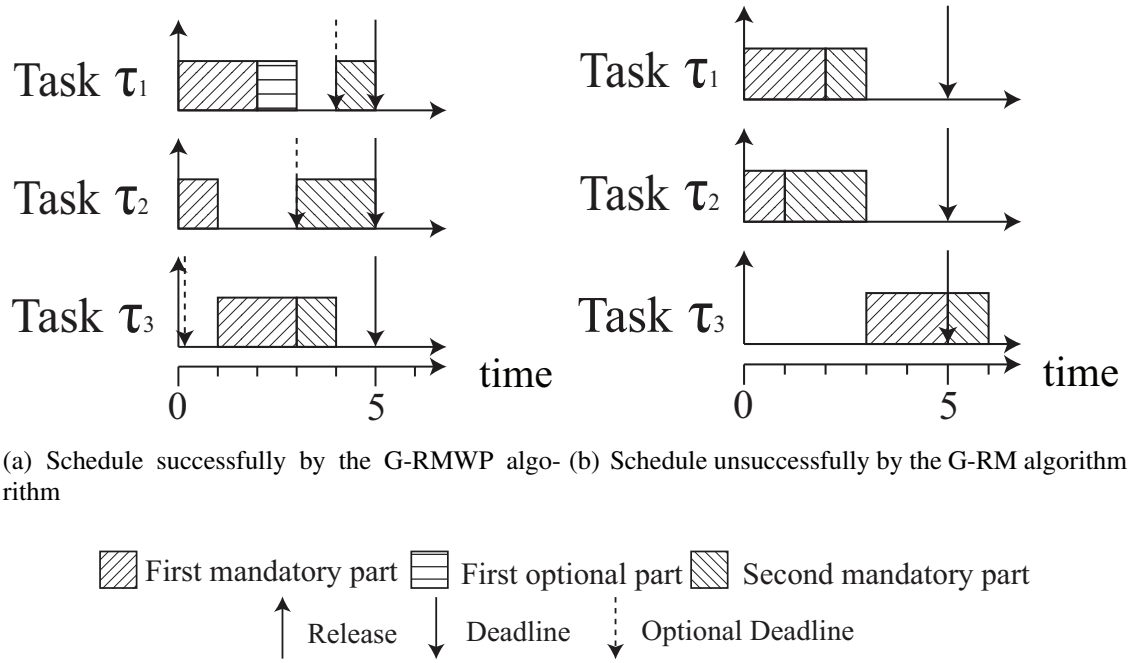
$$\hat{I}_k = \left\lfloor \frac{1}{M}\left(\sum_{i<k} I_i^{NC}(R_k^{ub}) + \sum_{i<\max(k,M-1)} I_i^{DIFF}(R_k^{ub}) \right) \right\rfloor. \tag{4.12}$$

Now this dissertation shows the worst case interference time of task τ_i in the G-RMWP algorithm by Equation (4.11).

Theorem 5 (Worst case interference time in the G-RMWP algorithm)**.** *The worst case interference time of task τ_k by higher priority tasks $\tau_i (i < k)$ in the G-RMWP algorithm is $\hat{I}_k$ by Equation (4.11).*

Proof. The worst case interference time of task τ_k by higher priority tasks $\tau_i (i < k)$ occurs if all optional deadlines of each task are equal to 0. In this case, the G-RMWP algorithm generates the same schedule as the G-RM algorithm. Moreover, there is no case that the worst case interference time of task τ_k interfered by higher priority tasks τ_i is more than $\hat{I}_k$ by Equation (4.11). □

In addition, this theorem explains that the technique for fixed-priority scheduling can be adapted to semi-fixed-priority scheduling. Next this dissertation calculates the relative optional deadlines of each task in the G-RMWP algorithm.



(a) Schedule successfully by the G-RMWP algorithm

(b) Schedule unsuccessfully by the G-RM algorithm

Figure 4.8: Example of schedule generated by the G-RMWP and G-RM algorithms on two processors

Theorem 6 (Optional deadline in the G-RMWP algorithm)**.** *The l^{th} relative optional deadline OD_k of task τ_k in the G-RMWP algorithm is the following equation.*

$$OD_k^l = \begin{cases} \max(0, D_k - m_k^{n_k^m}) & (k \leqq M \text{ and } l = n_k^o) \\ \max(0, D_k - m_k^{n_k^m} - \hat{I}_k) & (k > M \text{ and } l = n_k^o) \\ \max(0, OD_k^{l+1} - m_k^{l+1} - o_k^{l+1}) & (l < n_k^o) \end{cases} \tag{4.13}$$

Proof. If $k \leqq M$ and $l = n_k^o$, it is clear that the following mandatory part of task τ_k does not miss its deadline if task τ_k completes its mandatory part at the l^{th} relative optional deadline OD_k^l. If $k > M$ and $l = n_k^o$, the worst case interference time of task τ_k by higher priority tasks $\tau_i (i < k)$ is at most $\hat{I}_k$ by Theorem 5. In a similar way, if $l < n_k^o$, it is clear that task τ_k completes its following mandatory parts by its following optional deadlines or deadline if task τ_k completes its mandatory parts by its optional deadlines by Theorem 5. Hence, this theorem holds. □

The approach of calculating the relative optional deadlines of each task by Theorem 6 makes use of the worst case interference time of the G-RM algorithm by Equation (4.11), which is pessimistic. If the worst case interference time of the G-RM algorithm is more precise, the relative optional deadlines of each task in the G-RMWP algorithm can be set to the later value to expand the executable range of each optional part.

Figure 4.8 shows an example of schedule generated by the G-RMWP and G-RM algorithms on two processors. Table 4.2 shows the task set scheduled by the G-RMWP and G-RM algorithms in Figures 4.8(a) and 4.8(b) respectively. Each practical imprecise task has two mandatory parts and one optional part. Each relative optional deadline is calculated by Theorem 6. This example shows that there is at least one task set, which is schedulable

Table 4.2: Task set B

Task	T_i	D_i	OD_i^1	m_i^1	m_i^2	o_i^1
τ_1	5	5	4	2	1	1
τ_2	5	5	3	1	2	0
τ_3	5	5	0	2	1	0

by the G-RMWP algorithm and is not schedulable by the G-RM algorithm. Moreover, in the G-RMWP algorithm, job $\tau_{1,1}$ executes its optional part in $[2, 3)$.

4.3.2 Schedulability Analysis of the G-RMWP Algorithm

Now this dissertation analyzes the schedulability of the G-RMWP algorithm. First this dissertation proves that the G-RMWP algorithm is at least as effective as the G-RM algorithm, which is the similar approach to the RMWP algorithm.

Theorem 7 (The G-RMWP algorithm is at least as effective as the G-RM algorithm)**.** *One task set is schedulable by the G-RMWP algorithm if the task set is schedulable by the G-RM algorithm.*

Proof. This proof is shown by contraposition as well as that in Theorem 3. This dissertation shows that if one task set is not schedulable by the RMWP algorithm, the task set is not schedulable by the RM algorithm. By Theorem 6, it is clear that task τ_i completes its all mandatory parts by its deadline if task τ_i completes its all mandatory parts except the first mandatory part by its all optional deadlines respectively. Task τ_i misses its deadline only if task τ_i executes its mandatory part after its optional deadlines. In this case, task τ_i executes its all mandatory parts continuously without executing its all optional parts. In the G-RM algorithm, task τ_i also misses its deadline because of executing its all mandatory parts continuously. Hence, this theorem holds. □

By Theorem 7, this dissertation next shows the least upper bound of the G-RMWP algorithm.

Theorem 8 (Least upper bound of the G-RMWP algorithm)**.** *The least upper bound of the G-RMWP algorithm on multiprocessors is*

$$U_{lub} = \frac{M}{2}(1 - U_{max}) + U_{max}, \tag{4.14}$$

where M is the number of processors and $U_{max} = \max\{U_i \mid i = 1, 2, 3, ..., n\}$.

Proof. The G-RMWP algorithm is at least as effective as the G-RM algorithm by Theorem 7 and generates the same schedule as the G-RM algorithm if all relative optional deadlines of each task are equal to 0 as shown in Theorem 5. Hence, the least upper bound of the G-RMWP algorithm is equal to that of the G-RM algorithm [210]. □

By Theorems 7 and 8, the schedulability of the G-RMWP algorithm is higher than or equal to that of the G-RM algorithm. In addition, by Figure 4.8, the G-RMWP algorithm outperforms the G-RM algorithm from the aspects of both schedulability and imprecise computation.

```
j ← 1;
for i ← 1 to n do
    partitioned = FALSE;
    for k ← j to M + j − 1 do
        q ← (k − 1) mod M + 1;
        if task τ_i satisfies Theorem 9 on processor P_q then
            assign task τ_i to processor P_q;
            partitioned = TRUE;
            break;
        end
    end
    if partitioned == FALSE then
        return UNSCHEDULABLE;
    end
    j ← q mod M + 1;
end
return SCHEDULABLE;
```

Figure 4.9: Next-fit task assignment algorithm for the P-RMWP algorithm

4.4 The P-RMWP Algorithm

The Partitioned Rate Monotonic with Wind-up Part (P-RMWP) algorithm assigns all tasks to specific processors with bin packing heuristics. Assigning tasks on each processor are executed in the RMWP order. Now this dissertation analyzes the schedulability of the P-RMWP algorithm.

Theorem 9 (Schedulability analysis of the P-RMWP algorithm)**.** *The P-RMWP algorithm makes use of the two following equations as schedulability tests.*

$$U_{lub} = n(2^{1/n} - 1) \quad \geqq \quad U_{total} \tag{4.15}$$

$$R_k = m_k + \sum_{i=1}^{k-1} \left\lceil \frac{R_k}{T_i} \right\rceil m_i \quad \leqq \quad D_k, \tag{4.16}$$

where n is the number of tasks on each processor and U_{total} is each total utilization of assigning tasks to each processor.

Proof. By Theorem 3, it is clear that this theorem is true. □

By Theorem 9, the P-RMWP algorithm can use bin packing heuristics for the RM algorithm such as first-fit [211], next-fit [51] and best-fit [211] with task orderings such as increasing relative deadline and decreasing utilization [212] for task allocation.

Figure 4.9 shows the next-fit task assignment algorithm for the P-RMWP algorithm. Here, the processor P with the processor id q is represented as P_q. The P-RMWP algorithm uses schedulability test in Equation (4.15) or (4.16), as well as the P-RM algorithm. Therefore, the P-RMWP algorithm has as same schedulability as the P-RM algorithm.

4.5 Summary of Semi-Fixed-Priority Scheduling

This chapter first described the basic strategy of semi-fixed-priority scheduling. Next semi-fixed-priority scheduling algorithms for uniprocessor scheduling, multiprocessor global schedul-

ing and multiprocessor partitioned multiprocessor, called the RMWP, G-RMWP and P-RMWP algorithms respectively are proposed. The important characteristics of the proposed algorithms are as follows.

- The RMWP algorithm is at least as effective as the RM algorithm. That is to say, one task set is schedulable by the RMWP algorithm if the task set is schedulable by the RM algorithm. The least upper bound of the RMWP algorithm is equal to that of the RM algorithm $U_{lub} = n(2^{1/n} - 1)$, where n is the number of tasks.
- The G-RMWP algorithm is also at least as effective as the G-RM algorithm. The least upper bound of the RMWP algorithm is equal to that of the RM algorithm $U_{lub} = M(1 - U_{max})/2 + U_{max}$, where M is the number of processors and $U_{max} = \max\{U_i \mid i = 1, 2, 3, ..., n\}$.
- The P-RMWP algorithm makes use of the schedulability test for the RM algorithm with the least upper bound or RTA when tasks are assigned to processors. Therefore, the P-RMWP algorithm has as same schedulability as the P-RM algorithm.
- The advantage of semi-fixed-priority scheduling in the practical imprecise computation model against existing real-time scheduling in Liu and Layland's model is to execute optional parts for improving the quality of result without overrun of the optional part, thanks to the optional deadline.

Chapter 5

RT-Est Real-Time Operating System

This chapter describes the implementation of three semi-fixed-priority scheduling algorithms in the RT-Est real-time operating system, which is developed from scratch. The goals of the RT-Est real-time operating system are as follows. The first goal is to evaluate semi-fixed-priority scheduling algorithms compared to other real-time scheduling algorithms. Because there is no real-time operating system implementing semi-fixed-priority scheduling algorithms. The second goal is to make use of semi-fixed-priority scheduling algorithms in real-time systems such as autonomous mobile robots, which require low-jitter and high schedulability. The RT-Est real-time operating system is implemented on x86 multiprocessors.

The RT-Est real-time operating system must manage the system time so that the scheduler can release jobs at specified times. Moreover, the RT-Est real-time operating system must also manage the amount of time spent in both mandatory and optional parts of real-time jobs so that an overrun of an optional part is detected at the time when optional computation time is exhausted at the optional deadline. Otherwise, delayed termination of optional parts can cause a deadline miss in the schedule.

Terminating an optional part of each job and making the job resume in the following mandatory part require that the context of every imprecise thread be managed by the kernel. This issue is not trivial when the practical imprecise computation model is deployed, since the kernel must prepare the context of the thread for a mandatory part that might not have been executed at all.

Another one of the required features in the RT-Est real-time operating system is that the kernel should have the ability to allow users to easily change configurations for different situations. This is accomplished by supporting many modules and by extending the kernel for some new real-time scheduling algorithms and the portability to some new hardware platforms. This requires that the kernel clearly distinguish the part that is dependent to the real-time scheduling algorithm and to the hardware used from the other independent parts.

In addition, the RT-Est real-time operating system has enough time predictability to utilize the capability of imprecise computation to cope with uncertain workloads. To this end, the RT-Est real-time operating system is designed to be preemptive. This means that all the system calls including those required to implement the real-time scheduling algorithms are preemptive whenever possible. All the threads inside the kernel are scheduled preemptively with other application threads.

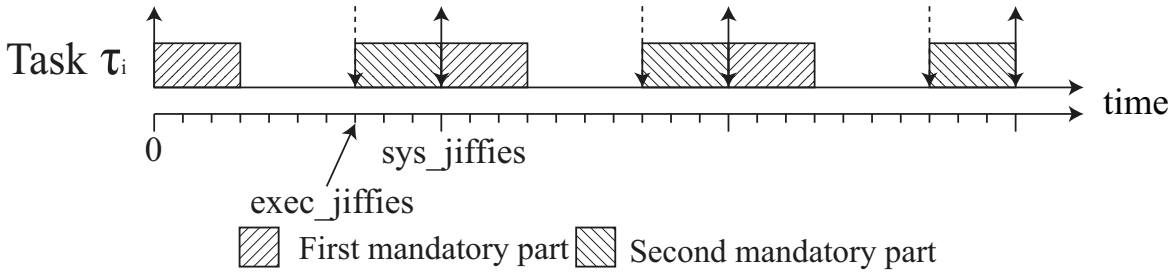


Figure 5.1: `sys_jiffies` and `exec_jiffies`

5.1 System Time Management

The RT-Est real-time operating system supports the high resolution timer, which usually performs to terminate optional parts at optional deadlines. The time when the timer interrupt of the high resolution timer occurs is between ticks. The execution time of each task between timer interrupts should be managed. In order to manage the execution time of each task, the RT-Est real-time operating system manages both system time and execution time, called `sys_jiffies` and `exec_jiffies` respectively.

`sys_jiffies` is incremented every periodic timer interrupt and `exec_jiffies` is set to the relative elapsed time to `sys_jiffies`, as shown in Figure 5.1. In this example, task τ_1 has two mandatory parts. The interrupt of each high resolution timer can occur per `exec_jiffies`. In this example, the unit of `sys_jiffies` is 10 times as much as that of `exec_jiffies`. In addition, the relative length of `exec_jiffies` compared to `exec_jiffies` can be defined by users.

5.2 Thread Management

Figure 5.2 shows the state machine of each thread in the RT-Est real-time operating system. When a thread is created in `create` function, its initial state is `UNADMITTED`. Then, it becomes `READY` in `activate` function. If the thread is assigned to the processor, its state becomes `RUNNING`. When the thread finishes its execution, the completion of each job is informed to the scheduler in `end_job` function. This function neither takes an argument nor returns a value and it succeeds always. If the aperiodic thread finishes its execution, its state is put back to `UNADMITTED`. If the periodic thread finishes its execution, its state is put to `NON_INT_SLEEP` unless `suspend` function was issued prior to `end_job` function. If the suspension of each periodic thread was requested in `suspend` function, the state of the thread is changed to `UNADMITTED`. Thus, `suspend` function does not directly change the state of a real-time thread. By comparison, `suspend` function issued for a non-real-time thread puts the state of the thread to `UNADMITTED`. A thread in the `NON_INT_SLEEP` state can only be woken up by the scheduler and only when its next release time is reached. A similar state is `INT_SLEEP`. This state is reached when a non-real-time thread issues `sleep` function, which takes the length to sleep as its argument. The thread in this state is woken up and put to the `READY` state either when its requested wake-up time is reached or when another thread issues `wake_up` function. `wake_up` function takes a pointer to the thread control block of the requested thread as its argument. The last remaining state `BLOCKED` can only be reached when a non-real-time thread is blocked on requesting a resource in `sem_down` function. In the RT-Est real-time operating system, a resource requested by a non-real-time thread should

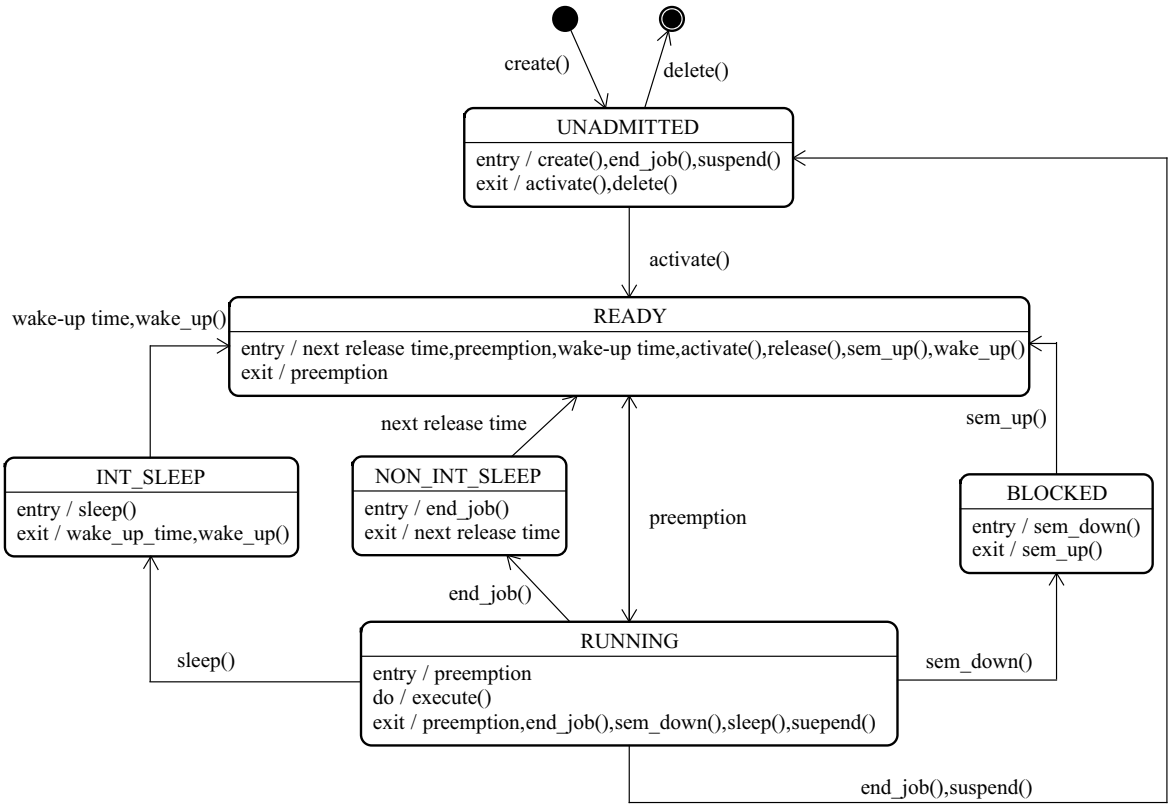


Figure 5.2: State machine of thread

never be requested by a real-time thread. The thread in the `BLOCKED` state is put back to the `READY` state when the blocking thread releases the resource in `sem_up` function. These two functions used to acquire and to release a resource take a pointer to the semaphore that is guarding the requested resource.

There are two ways an application can be implemented. One way is to implement it as a user level task. A user level task receives services from the server threads via port-based communication or by entering supervisor mode via system calls. The second way is to implement it as one of the server threads to build in the application as part of the kernel. The first way has an advantage that the application program can be dynamically loaded and tested without having the kernel reloaded to the target board every time, which makes the cross development of applications easier and quicker. On the other hand, the second way allows an application with very fine timing constraints to be implemented. For example, software that controls the body of a robot may be critical enough to reside in the kernel space to access peripheral I/O devices directly. Therefore, the RT-Est real-time operating system supports the second way.

5.3 Ultra Configurable Module

This dissertation introduces the ultra configurable module in the RT-Est real-time operating system. Thanks to the ultra configurable module, users can change various configurations easily. The supported module is as follows.

- Real-time scheduling

- Synchronization protocol
- Processor environment
- Task set
- Architecture

The real-time scheduling module selects real-time scheduling algorithms including the RM [2], RM-US [52] EDF [2], EDF-US [213], RMWP and M-FWP algorithms [34, 35]. The synchronization protocol module selects synchronization protocols including Priority Ceiling Protocol (PCP) [214] and Stack Resource Policy (SRP) [215].

The processor environment module selects supported processor environments including uniprocessor environment, multiprocessor global environment and multiprocessor partitioned environment with bin packing heuristics such as first-fit, next-fit, best-fit and worst-fit.

The task set module has the task set type, the processor type and the task sorting policy. The supported task set types are general task sets and harmonic task sets. A general task set indicates that there is not relevant to the period of each task. In contrast, a harmonic task set indicates that the period of each task is equal to the integral multiple of that of the task with the shorter period. The supported task sorting policies are increasing period, increasing relative deadline and decreasing utilization.

The architecture module selects architectures to execute various processors including x86, SH and SIM. SIM is one of the architecture modules for simulating real-time scheduling in the RT-Est real-time operating system. Unlike User-Mode Linux [216], SIM does not measure the actual processor time consumed by tasks. Like Real-Time system SIMulator (RTSIM) [217], SIM executes simulations of real-time scheduling algorithms theoretically. SIM can make use of implementations of architecture independent parts to those of other architectures, unlike RTSIM. Therefore, SIM is an effective technique for not only measuring the performance of real-time scheduling algorithms but also improving the efficiency of development.

5.4 Implementation of Scheduler

In this section, this dissertation presents two schedulers for semi-fixed-priority scheduling, called *hybrid scheduler* and *dual scheduler*.

5.4.1 Hybrid Scheduler

The hybrid scheduler manages not only practical imprecise tasks in this dissertation's model but also general tasks in Liu and Layland's model [2] if $n_i^m = 1$ and $n_i^o = 0$.

Figure 5.3 shows the hybrid scheduler with N priority levels. In this dissertation, the number of priority levels is $N = 512$ because Liu claims that 256 priority levels are sufficient even for the most complex rate-monotonically scheduled systems [218]. Like Linux, a smaller number indicates higher priority. In addition, the sum of priority levels including both real-time and non-real-time parts in the practical imprecise tasks is 512 and the highest priority level is 0. The priority range of the RTQ is $[0, N/2 - 1]$ and that of the NRTQ is $[N/2, N - 1]$. The hybrid scheduler manages each task queue by each double circular linked list in the FIFO order.

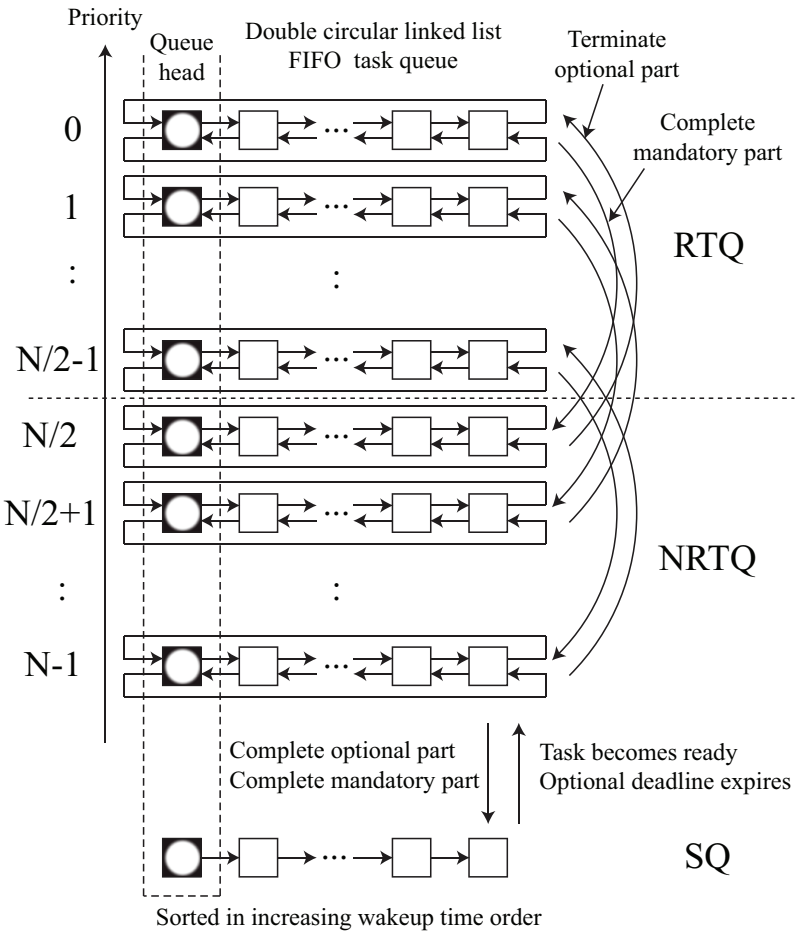


Figure 5.3: Hybrid scheduler

Now this dissertation explains the behavior of the hybrid scheduler. For example, task τ_i, the priority of which is 0, is created and enqueued to the SQ. When task τ_i is released, task τ_i is dequeued from the SQ, is ready to execute its mandatory part and is enqueued to the RTQ. When task τ_i completes its mandatory part, task τ_i is dequeued from the RTQ, sets its priority to $N/2$ and is enqueued to the NRTQ. When task τ_i terminates its optional part, task τ_i is dequeued from the NRTQ, sets its priority to 0, is ready to execute its following mandatory part and is enqueued to the RTQ. When task τ_i completes its optional part or mandatory part until its optional deadline, task τ_i is dequeued from the RTQ or the NRTQ and is enqueued to the SQ.

5.4.2 Dual Scheduler

The dual scheduler is an extension of the hybrid scheduler for global scheduling. First of all, this dissertation explains how to implement the G-RMWP algorithm. Figure 5.4 shows the dual scheduler. A running queue manages running tasks, the number of which is at most that of processors and a ready queue manages ready tasks. Each queue has N double circular linked lists, where N is the number of priority levels. In addition, the implementation of the dual scheduler can be adapted to global fixed-priority scheduling algorithms such as the G-RM algorithm in Liu and Layland's model if $n_i^m = 1$ and $n_i^o = 0$.

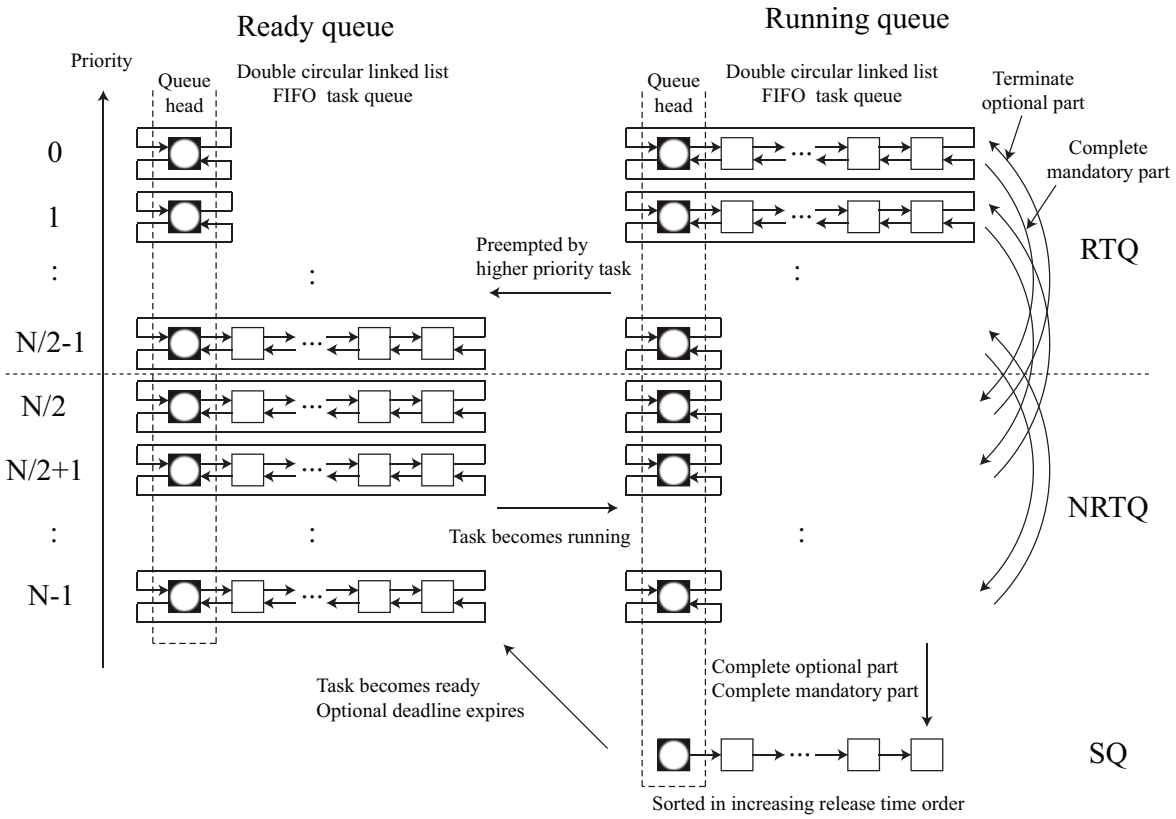


Figure 5.4: Dual scheduler

5.5 Imprecise Computation

Figure 5.5 shows the pseudo code of the practical imprecise computation model. This practical imprecise task has two mandatory parts and one optional part. First each task saves its context including general purpose registers and the program counter in `save_context` function. Because each task resumes its context in the timer interrupt routine for terminating the optional part at the optional deadline. If `save_context` function is called via the timer interrupt routine to terminate the optional part at the optional deadline, `save_context` function returns `MANDATORY2`. Otherwise `save_context` function returns `MANDATORY`. Next each task executes the first mandatory part in `exec_mandatory` function. After completing its mandatory part, each task calls `end_mandatory` function. If the return value of `end_madnatory` function is `DISCARD`, each task discards its optional part and executes the following mandatory part in `exec_mandatory2` function. Otherwise each task executes the first optional part in `exec_optional` function. If each task completes its optional part, each task calls `end_optional` function and executes the following mandatory part in `exec_mandatory2` function. If each task terminates its optional part at its optional deadline, the scheduler resumes its context in the timer interrupt routine and calls `save_context` function. In this case, the return value of `save_context` function is `MANDATORY2` so that the resumed task executes the second mandatory part in `exec_mandatory2` function. After completing the second mandatory part, each task calls `end_job` function to complete its job.

This dissertation now introduces three functions to implement the practical imprecise computation model: `end_mandatory`, `end_optional` and `terminate_optional`.

```
part = save_context(); /* return value: MANDATORY,MANDATORY2,... */
switch (part) {
case MANDATORY:
  /* execute first mandatory part */
  exec_mandatory();
  res = end_mandatory();
  /* pass through */
  /* execute first optional part */
  if (res != DISCARD) {
    exec_optional();
    end_optional();
  }
  /* pass through */
case MANDATORY2:
  /* execute second mandatory part */
  exec_mandatory2();
}
end_job();
```

Figure 5.5: Pseudo code of the practical imprecise computation model

Figure 5.6 shows `end_mandatory` function for the G-RMWP algorithm, which is called when each task completes its mandatory part. If the optional deadline of the current task does not expire at the current time, then the current task starts to execute its optional part. The current task is dequeued from the running queue, decreases its priority and changes its part from mandatory part to optional part. Next the scheduler finds the ready task which has the highest priority in the ready queue. If the ready task has higher priority than the current task, preempt the current task. The current task is enqueued to the ready queue. After that, the ready task is dequeued from the ready queue and enqueued to the running queue. Otherwise the current task is enqueued to the running queue, again. If the optional deadline of the current task does not expire at the current time, then the current task starts to execute its following mandatory part.

On the other hand, the implementation of `end_mandatory` function in the P-RMWP algorithm is different from the enqueue and dequeue operations. Because the P-RMWP algorithm schedules assigned tasks on each processor so that enqueue and dequeue operations are performed in the same ready queue.

Figure 5.7 shows `end_optional` function for the G-RMWP algorithm, which is called when each task completes its optional part. The current task changes its part from optional part to mandatory part, sleeps until its optional deadline and increases its priority. If the current time is the optional deadline of the task, execute its following mandatory part. When each task sleeps until its optional deadline or deadline, the task is managed by the single SQ, regardless of multiprocessor real-time scheduling policies. Therefore, the implementation of `end_optional` function in the P-RMWP algorithm is equal to that in the G-RMWP algorithm.

Figure 5.8 shows `terminate_optional` function for the G-RMWP algorithm, which is called when each task terminates its optional part at its optional deadline. First the scheduler gets the task at the head of the optional deadline queue, which sorts tasks by increasing

```
if currentTime < currentTask.optionalDeadline then
    DequeueTask (runningQueue, currentTask);
    DecreasePriority (currentTask);
    ChangePart (OPTIONAL);
    readyTask ← GetHighestPriorityTask (readyQueue);
    if readyTask has higher priority than currentTask then
        // preempt currentTask
        EnqueueTask (currentTask, readyQueue);
        DequeueTask (readyTask, readyQueue);
        EnqueueTask (readyTask, runningQueue);
    end
    else
        // not preempt currentTask
        EnqueueTask (currentTask, runningQueue);
    end
end
else
    SleepUntil (currentTask.release + currentTask.period, currentTask);
end
```

Figure 5.6: `end_mandatory` function for the G-RMWP algorithm

```
ChangePart (MANDATORY);
SleepUntil (currentTask.optionalDeadline, currentTask);
IncreasePriority (currentTask);
```

Figure 5.7: `end_optional` function for the G-RMWP algorithm

absolute optional deadline. Next the scheduler checks if the absolute optional deadline of the task expires. If this is true, execute the following operations. If the task is ready to execute or executing its optional part, then the task is dequeued from the optional deadline queue. If the task is running, then the task is dequeued from the running queue. Otherwise the task is dequeued from the ready queue. After that, the task increases its priority and changes its part from optional part to mandatory part. Next the scheduler gets an idle processor ID. If there is an idle processor, then the task is enqueued to the running queue and assigned to the processor. Otherwise the scheduler gets the running task which has the lowest priority in the running queue. If the task has higher priority than the running task, then preempt the running task. The scheduler gets the processor ID of the running task. Next the running task is dequeued from the running queue and enqueued to the ready queue. After that, the task is enqueued to the running queue and assigned to the processor. Otherwise the task is enqueued to the ready queue. Finally the scheduler gets the next task and checks if the absolute optional deadline of the next task expires, again.

On the other hand, the implementation of `terminate_optional` function in the P-RMWP algorithm is different from the G-RMWP algorithm. The P-RMWP algorithm does not need to find the lowest priority task in running tasks. Because the P-RMWP algorithm manages each task queue on each processor. By the same reason in `end_mandatory` function, the P-RMWP algorithm schedules assigning tasks on each processor so that the enqueue and dequeue operations are performed in the same ready queue.

```
task ← GetHeadOfQueue (optionalDeadlineQueue);
while task.optionalDeadline ≦ currentTime do
    if task is ready to execute or executing its optional part then
        DequeueOptionalDeadlineQueue (task);
        if task is running then
            DequeueTask (task, runningQueue);
        end
        else
            DequeueTask (task, readyQueue);
        end
        IncreasePriority (task);
        ChangePart (MANDATORY);
        idleProcessorId ← GetIdleProcessorId ();
        if there is an idle processor then
            EnqueueTask (task, runningQueue, idleProcessorId);
        end
        else
            runningTask ← GetLowestPriorityTask (runningQueue);
            if task has higher priority than runningTask then
                // preempt runningTask
                processorId ← GetProcessorId (runningTask);
                DequeueTask (runningTask, runningQueue);
                EnqueueTask (runningTask, readyQueue);
                EnqueueTask (task, runningQueue, processorId);
            end
            else
                // not preempt runningTask
                EnqueueTask (task, readyQueue);
            end
        end
    end
    task ← GetHeadOfQueue (optionalDeadlineQueue);
end
```

Figure 5.8: `terminate_optional` function for the G-RMWP algorithm

5.6 Architecure Dependent Implementation on x86 Multiprocessors

The RT-Est real-time operating system supports x86 multiprocessors, which have Advanced Programmable Interrupt Controller (APIC). Each APIC has a local APIC timer which generates a local APIC timer interrupt on each processor. Now this dissertation explains x86-specific implementation on multiprocessors.

When booting the system on x86 multiprocessors, the Boot Strap Processor (BSP) is booted. Next BSP initializes IRQ and sends startup inter-processor interrupts to Application Processors (APs), which are remaining processors except BSP. Then all APs are booted. When all processors including BSP and APs are ready to start each scheduler, then each processor generates each local APIC timer interrupt and starts to execute tasks. The IRQ ID of each local APIC timer interrupt on each processor is different because the interrupt

```
irq = do_LAPIC_IRQ();
if (irq != NO_IRQ) {
  goto end;
}
irq = do_Master8259_IRQ();
if (irq != NO_IRQ) {
  goto end;
}
irq = do_Slave8259_IRQ();
if (irq != NO_IRQ) {
  goto end;
}
end:
do_IRQ_vector(irq);
```

Figure 5.9: Pseudo code of the interrupt handler

handler should be executed simultaneously to reduce the overhead. In addition, the interrupt of local APIC is handled before that of Intel 8259, which is one of the PIC. Figure 5.9 shows the pseudo code of the interrupt handler in the RT-Est real-time operating system. First `do_LAPIC_IRQ` function is called to check whether any local APIC interrupt occurs. If the return value of `do_LAPIC_IRQ` function is `NO_IRQ`, no local APIC interrupt occurs. Otherwise `do_IRQ_vector` function is called without checking interrupts of Master and Slave 8259s. In a similar way, `do_Master8259_IRQ` and `do_Slave8259_IRQ` functions are called to check whether other interrupts occur.

5.7 Summary of RT-Est Real-Time Operating System

This chapter presents the RT-Est real-time operating system for semi-fixed-priority scheduling algorithms. This chapter first introduces some features including time management, thread management and ultra configurable module. The RT-Est real-time operating system implements the hybrid scheduler for the RMWP and P-RMWP algorithms and the dual scheduler for the G-RMWP algorithm. This chapter next presents how to implement the practical imprecise computation model and functions: `end_mandatory`, `end_optional` and `terminate_optional`. Finally, the architecture dependent implementation using APIC on x86 multiprocessors is described.

Chapter 6

Simulation Studies

This chapter evaluates the proposed algorithms through simulation. The simulation studies are running at the SIM architecture in the RT-Est real-time operating system. The source codes of architecture independent parts used at the SIM architecture are common so that these codes are reusable for other architectures such as x86.

In this simulation, there is no overhead such as context switches, task queueing, interrupt handler and migrations. Because this simulation focuses on the theoretical capacity of evaluated algorithms. Each practical imprecise task has two mandatory parts and one optional part. This dissertation uses the Mersenne Twister [219] for uniform pseudo random number generator.

The performance metrics are defined as the following equations.

$$\text{Success Ratio} = \frac{\text{\# of successfully scheduled task sets}}{\text{\# of scheduled task sets}} \tag{6.1}$$

$$\text{Reward Ratio} = \frac{T_i}{n \cdot \text{simulation length}} \sum_j \frac{o_{i,j}}{o_i} \tag{6.2}$$

$$\text{Switch Ratio} = \frac{\text{\# of context switches}}{M \cdot \text{simulation length}} \tag{6.3}$$

$$\text{RRJ Ratio} = \frac{1}{n} \sum_i \frac{RRJ_i}{T_i} \tag{6.4}$$

$$\text{RFJ Ratio} = \frac{1}{n} \sum_i \frac{RFJ_i}{T_i} \tag{6.5}$$

$$\text{Migration Ratio} = \frac{\text{\# of migrations}}{M \cdot \text{simulation length}} \tag{6.6}$$

The success ratio is the value of the number of successfully scheduled task sets divided by the number of scheduled task sets in each system utilization. If the success ratio is equal to 1, all task sets are schedulable. If the success ratio of each system utilization is lower than 1, the results of the system utilization evaluated by other performance metrics are omitted.

The reward ratio is the sum of the average execution ratio between the actual RET of optional part $o_{i,j}$ of job $\tau_{i,j}$ and the RET of optional part o_i of task τ_i normalized by the period T_i and by the inverses of both the number of tasks and the simulation length. If the reward ratio is equal to 1, all tasks complete their all optional parts.

The switch ratio is the sum of the number of context switches normalized by the inverses of both the number of processors and the simulation length.

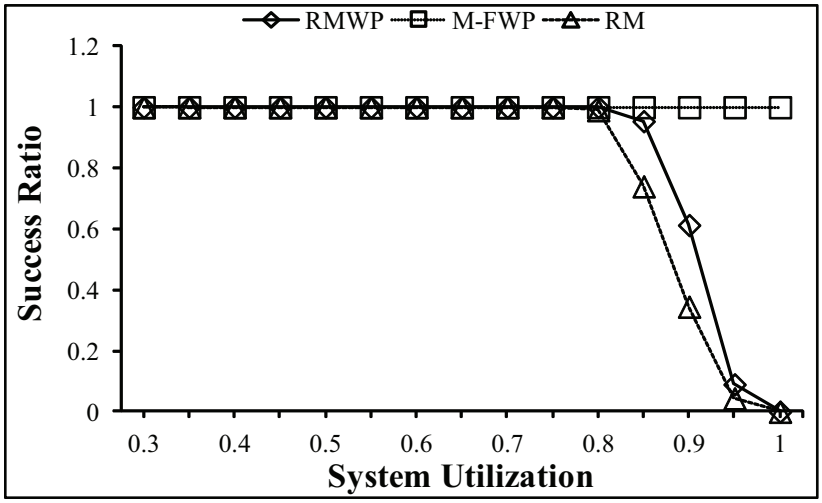


Figure 6.1: Success ratio on uniprocessors

The RRJ ratio is the sum of the RRJ of task τ_i in Equation (3.1) normalized by the inverses of both the period T_i of task τ_i and the number of tasks. Also, the RFJ ratio is the sum of the RFJ of task τ_i in Equation (3.2) normalized by the inverses of both the period T_i of task τ_i and the number of tasks. If the RRJ and RFJ ratios are equal to 0, the jitters of all tasks are equal to 0.

The migration ratio is the sum of the number of migrations normalized by the inverses of both the number of processors M and the simulation length. In addition, the migration ratio is only used for global scheduling on multiprocessors. If the migration ratio is equal to 0, there is no migrating task.

6.1 Simulation Studies on Uniprocessors

6.1.1 Simulation Setups on Uniprocessors

The simulation studies on uniprocessors use $1,000$ task sets in each system utilization and evaluate the RMWP, RM and M-FWP algorithms. The period T_i of each task τ_i is selected within $[100, 200, 300, ..., 3000]$. Each system utilization U_i is selected from $[0.02, 0.03, 0.04, ..., 0.25]$. The total system utilization U_s is selected from $[0.3, 0.35, 0.4, ..., 1.0]$. The simulation length of the k^{th} task set is H_k, which is the hyperperiod of the k^{th} task set.

The utilization of the optional part $o_{i,j}/T_i$ is within the range of $[o_i - 0.05, o_i + 0.05]$, where o_i is selected within $[0.1, 0.2, 0.3]$, represented such as the RMWP-10, RMWP-20 and RMWP-30 algorithms or the M-FWP-10, M-FWP-20 and M-FWP-30 algorithms respectively, computed at every task release. If the utilization of the optional part $o_{i,j}/T_i$ is always equal to 0, the result is represented as the RMWP or M-FWP algorithm.

The results of the RMWP-10, RMWP-20 and RMWP-30 algorithms in the success and RRJ ratios are equal to the result of the RMWP algorithm and those of the M-FWP-10, M-FWP-20 and M-FWP-30 algorithms in the success ratio are equal to the result of the M-FWP algorithm so that they are omitted.

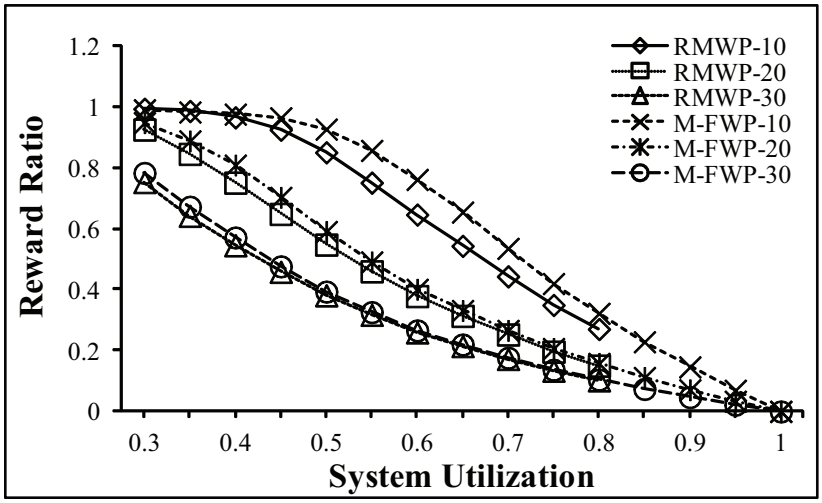


Figure 6.2: Reward ratio on uniprocessors

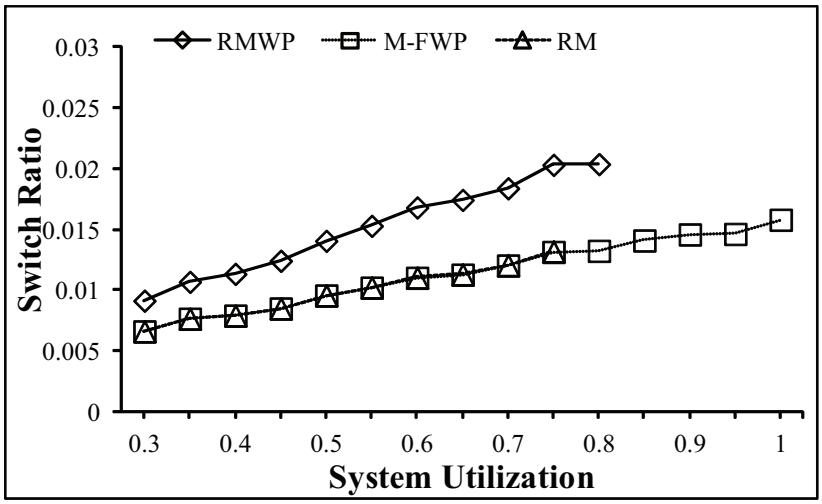


Figure 6.3: Switch ratio on uniprocessors

6.1.2 Simulation Results on Uniprocessors

Figure 6.1 shows the simulation result of the success ratio on uniprocessors. The success ratio of the M-FWP algorithm is always 1. The success ratio of the RMWP algorithm is degraded when the system utilization is higher than 0.8. In contrast, the success ratio of the RM algorithm is degraded when the system utilization is higher than 0.75. In addition, the success ratio of the RMWP algorithm is always higher than or equal to that of the RM algorithm by Theorem 3.

Figure 6.2 shows the simulation result of the reward ratio on uniprocessors. The M-FWP-10, M-FWP-20 and M-FWP-30 algorithms outperform the RMWP-10, RMWP-20 and RMWP-30 algorithms respectively because the RMWP algorithm sets each optional deadline statically. In contrast, the M-FWP algorithm calculates the assignable time of each optional part dynamically so that the analysis of the assignable time of the optional part is more precise.

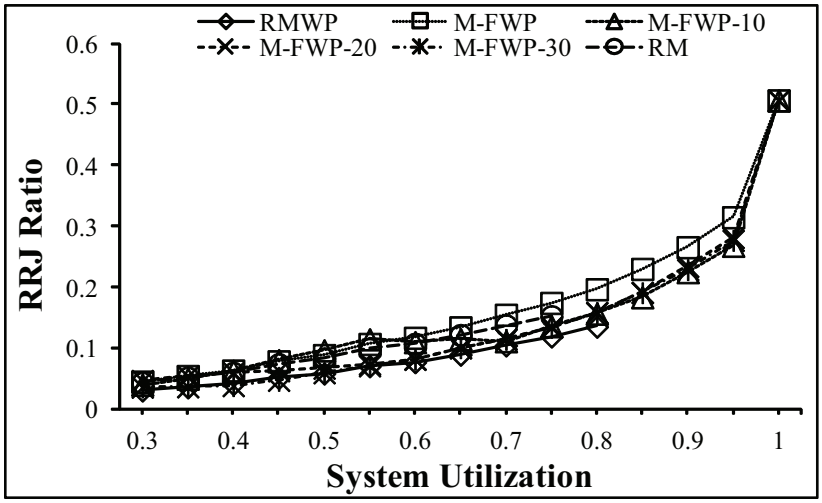


Figure 6.4: RRJ ratio on uniprocessors

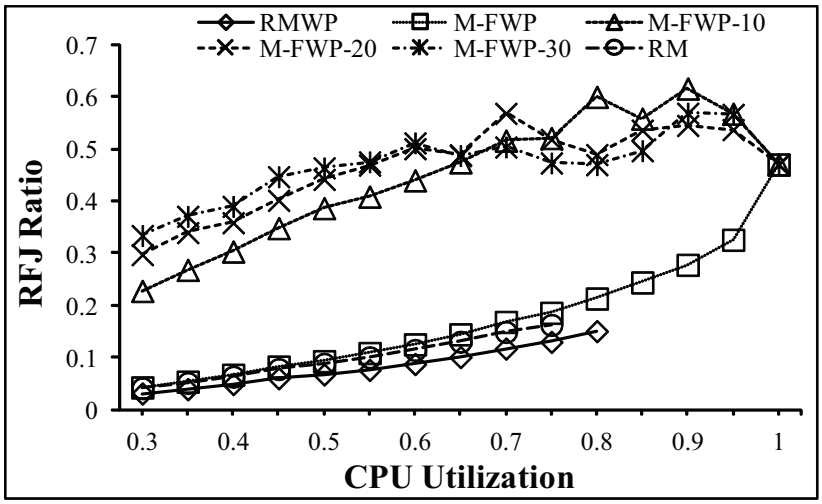


Figure 6.5: RFJ ratio on uniprocessors

Figure 6.3 shows the simulation result of the switch ratio on uniprocessors. The switch ratio of the RMWP algorithm is higher than that of the RM algorithm because the RMWP algorithm splits one practical imprecise task into multiple Liu and Layland's tasks, the number of which is equal to that of mandatory parts. Therefore, the frequency of context switches in the RMWP algorithm is higher than that in the RM algorithm. The switch ratio of the RMWP algorithm is also higher than that of the M-FWP algorithm because when the optional part of each task is completed by each optional deadline, the task sleeps until the optional deadline in the RMWP algorithm. In addition, the switch ratio of the M-FWP algorithm is approximately equal to that of the RM algorithm. If the utilization of the optional part is 0 in the M-FWP algorithm, the M-FWP algorithm generates the same schedule as the EDF algorithm, which has approximately as same switch ratio as the RM algorithm.

Figure 6.4 shows the simulation result of the RRJ ratio on uniprocessors. The RMWP algorithm can reduce the RRJ ratio, regardless of the assignable time of each optional part thanks to the optional deadline, unlike the M-FWP-20, M-FWP-40 and M-FWP-60 algo-

rithms. The RMWP algorithm has the lowest RRJ ratio in all evaluated algorithms.

Figure 6.5 shows the simulation result of the RFJ ratio on uniprocessors. The RFJ ratio of the M-FWP-10, M-FWP-20 and M-FWP-30 algorithms are dramatically the different results to the RFJ ratio of the M-FWP algorithm because they calculate the assignable time of each optional part when each mandatory part is completed. If the RET of each optional part is more than 0, the optional part is ready to be executed. Otherwise the following mandatory part is ready to be executed immediately or the task finishes its all mandatory parts. Therefore, the RFJ ratios of the M-FWP-10, M-FWP-20 and M-FWP-30 algorithm are dramatically higher than the RFJ ratio of the M-FWP algorithm. On the other hand, the RFJ ratio of the RMWP algorithm is also lowest in all evaluated algorithms, thanks to the optional deadline.

From the simulation results of the RRJ and RFJ ratios, the RMWP algorithm is the lowest jitter in all evaluated algorithms.

6.2 Simulation Studies on Multiprocessors

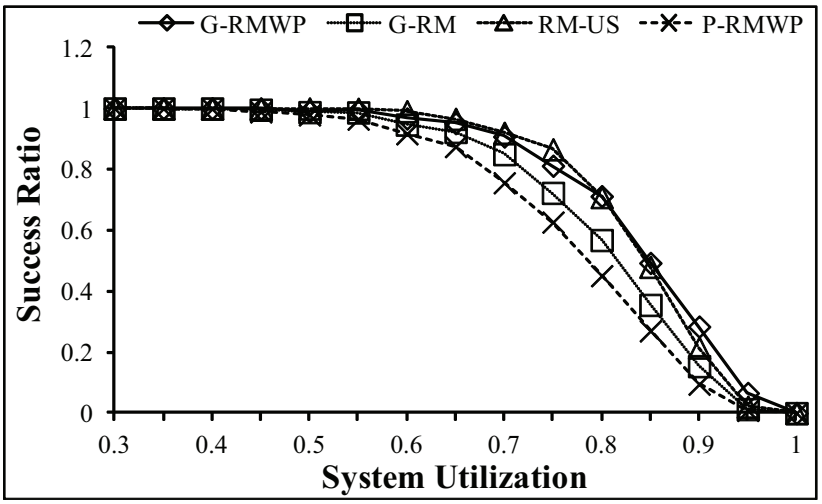
6.2.1 Simulation Setups on Multiprocessors

The simulation studies on multiprocessors use $1,000$ task sets in each system utilization and evaluate the G-RMWP, G-RM, RM-US[M/(3M-2)], P-RMWP and P-RM algorithms. In simulation environments, the number of processors M is selected within $[4, 8, 16]$. Each system utilization U_i is selected within $[0.02, 0.03, 0.04, ..., U_{max}]$, where U_{max} is selected within $[0.1, 0.5, 1.0]$. The period T_i of each task τ_i is selected within $[100, 200, 300, ..., 3000]$. The utilization of optional part $o_{i,j}/T_i$ is within the range of $[o_i - 0.05, o_i + 0.05]$, where o_i is selected within $[0.2, 0.4, 0.6]$, represented such as the G-RMWP-20, G-RMWP-40 and G-RMWP-60 algorithms or the P-RMWP-20, P-RMWP-40 or P-RMWP-60 algorithms respectively, computed at every task release. If the utilization of the optional part $o_{i,j}/T_i$ is always equal to 0, the result is represented as the G-RMWP or P-RMWP algorithm respectively. The total system utilization U_s is selected from $[0.3, 0.35, 0.4, ..., 1.0]$. The simulation length of the k^{th} task set is H_k, which is the hyperperiod of the k^{th} task set.

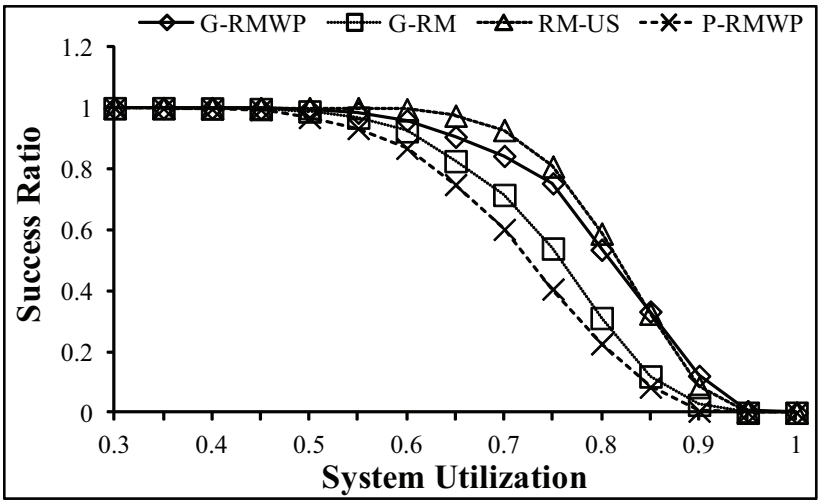
The P-RMWP and P-RM algorithms use the same schedulability test by the next-fit heuristic in Equation (4.16) so that the success ratio of the P-RMWP algorithm is the same as that of the P-RM algorithm and the result of the P-RM algorithm is omitted. The results of the G-RMWP-20, G-RMWP-40 and G-RMWP-60 algorithms in the success and RRJ ratios are equal to the result of the G-RMWP algorithm so that they are omitted.

6.2.2 Simulation Results on Multiprocessors

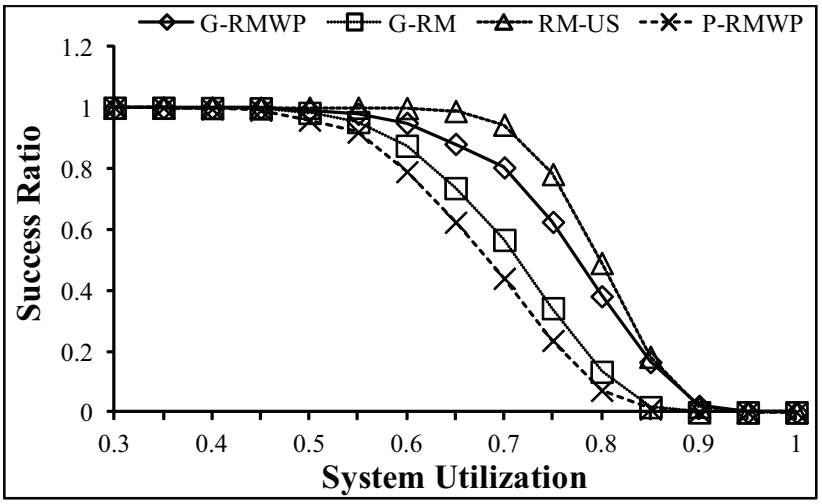
Figures 6.6, 6.7 and 6.8 show the success ratios when $U_{max} = 1.0$, $U_{max} = 0.5$ and $U_{max} = 0.1$ respectively. In all results, the success ratio of the G-RMWP algorithm is higher than or equal to that of the G-RM algorithm by Theorem 7. In Figure 6.6, the RM-US algorithm has the highest success ratio in all evaluated algorithms. In contrast, in Figure 6.7, the G-RMWP algorithm has the highest success ratio in all evaluated algorithms. The G-RM algorithm outperforms the RM-US algorithm in Figure 6.7 because the occurrence frequency of Dhall's effect [51] is less than that of avoiding the deadline miss of each task thanks to the technique of the utilization separation. In Figure 6.8, the G-RMWP, G-RM and RM-US algorithms have the approximately same success ratios. In addition, the success ratio of the



(a) M = 4

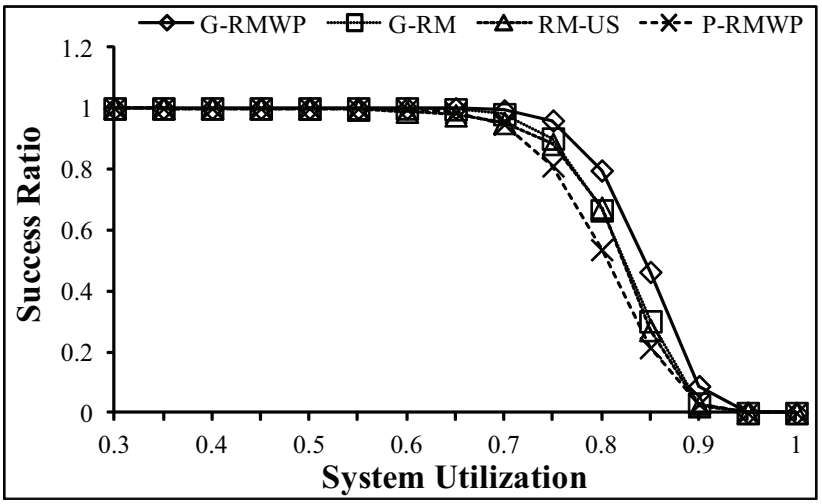


(b) M = 8

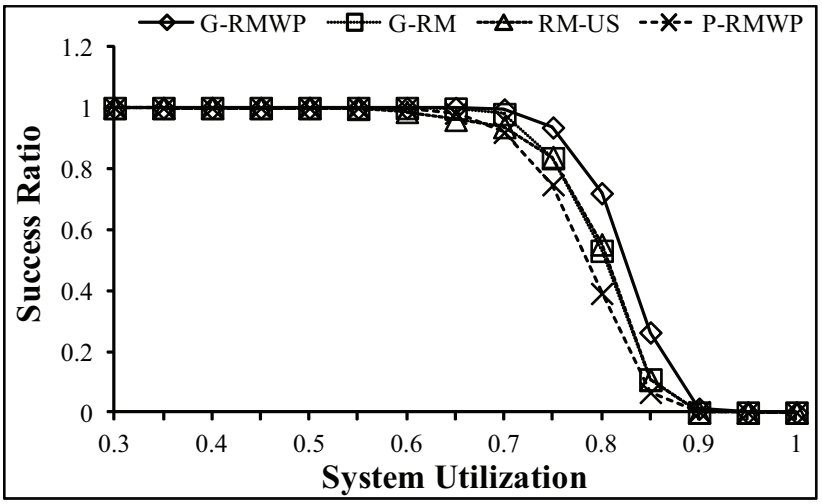


(c) M = 16

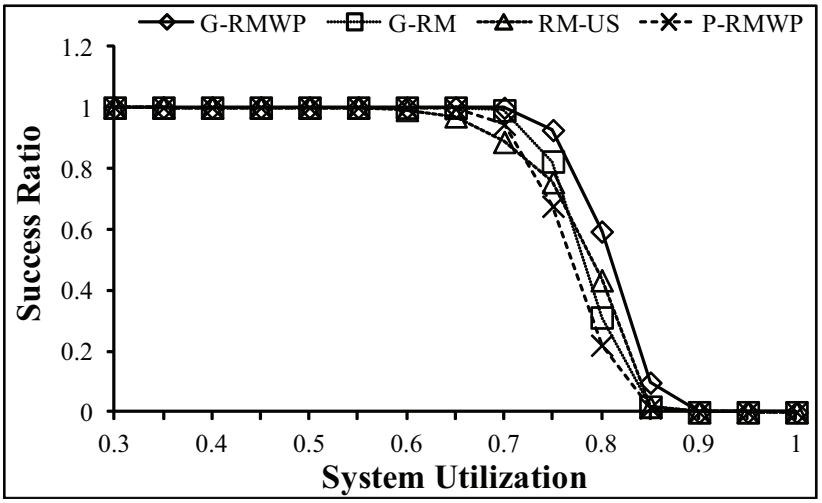
Figure 6.6: Success ratio on multiprocessors when U_{max} = 1.0



(a) M = 4

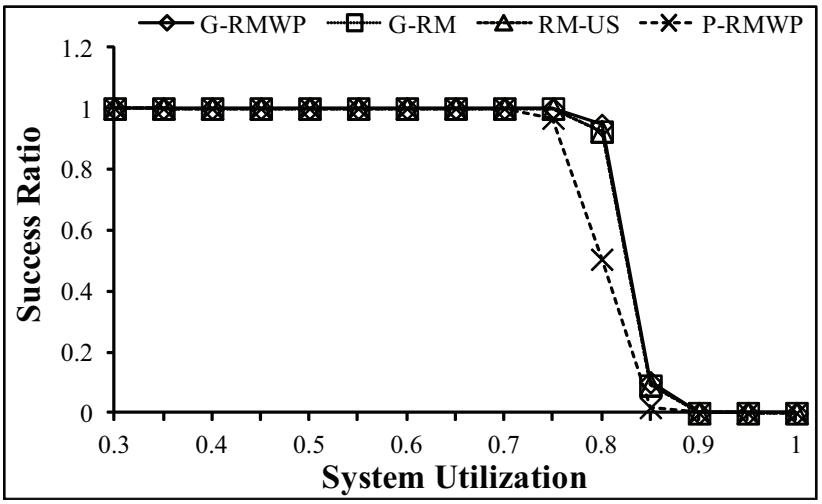


(b) M = 8

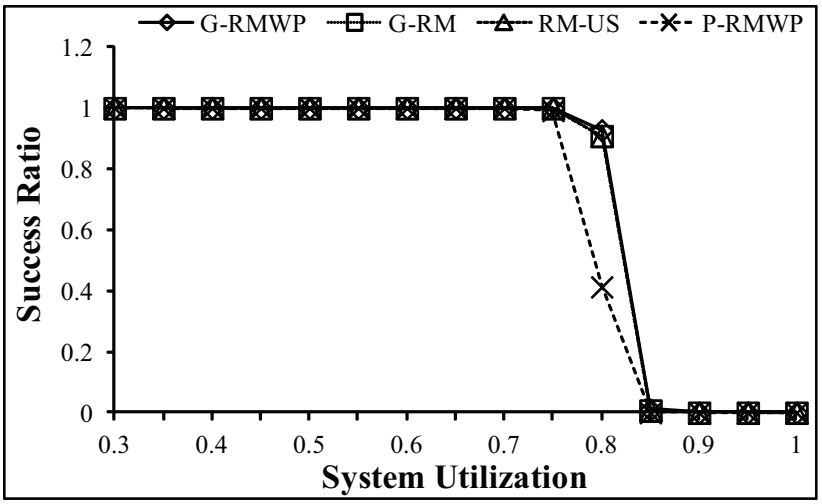


(c) M = 16

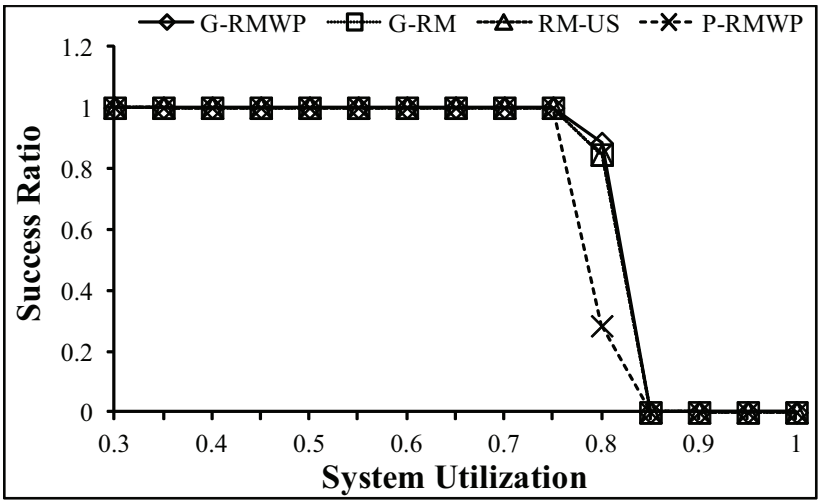
Figure 6.7: Success ratio on multiprocessors when U_{max} = 0.5



(a) M = 4

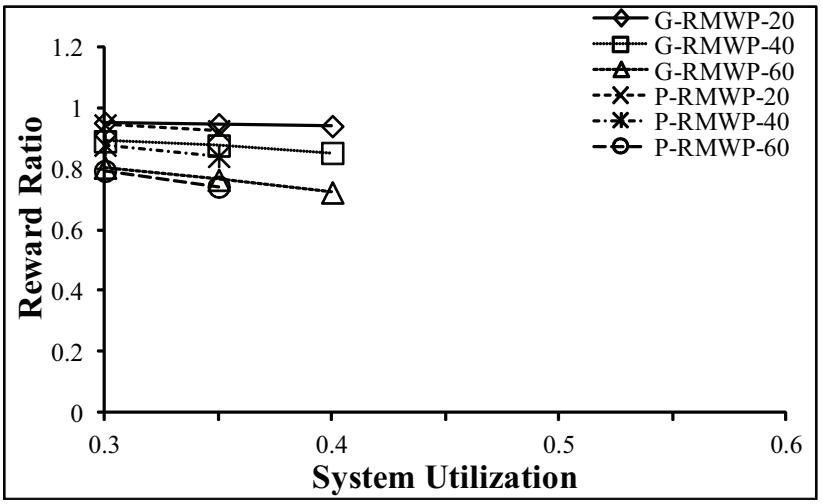


(b) M = 8

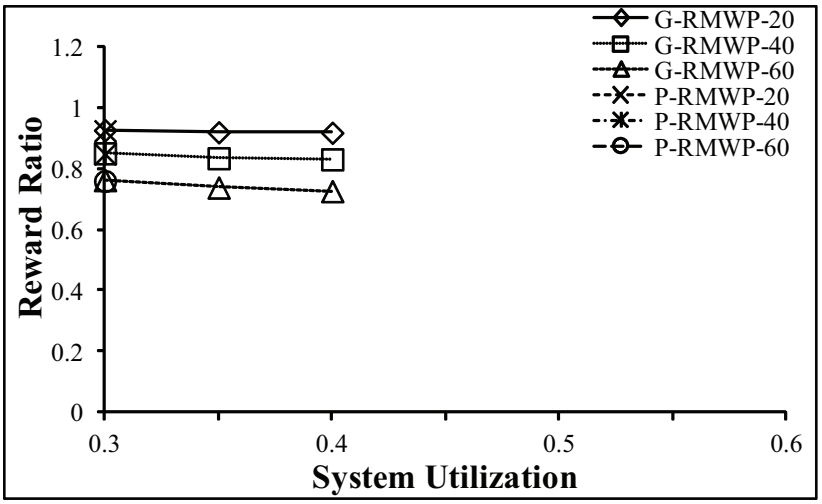


(c) M = 16

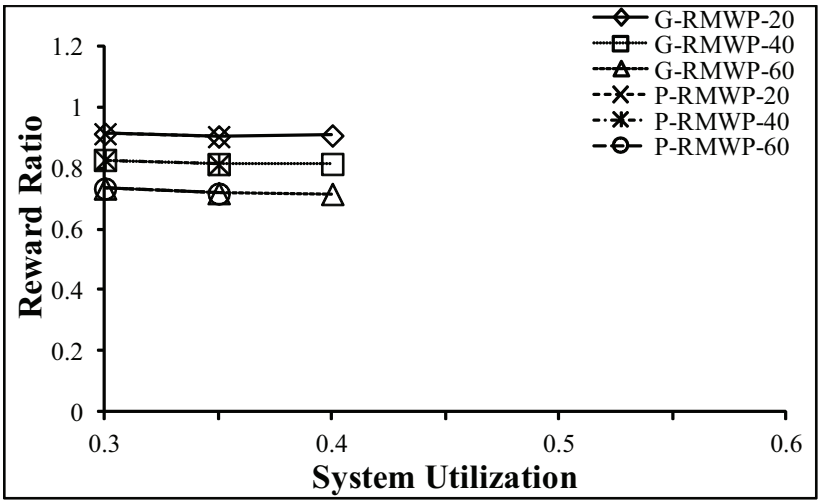
Figure 6.8: Success ratio on multiprocessors when U_{max} = 0.1



(a) M = 4

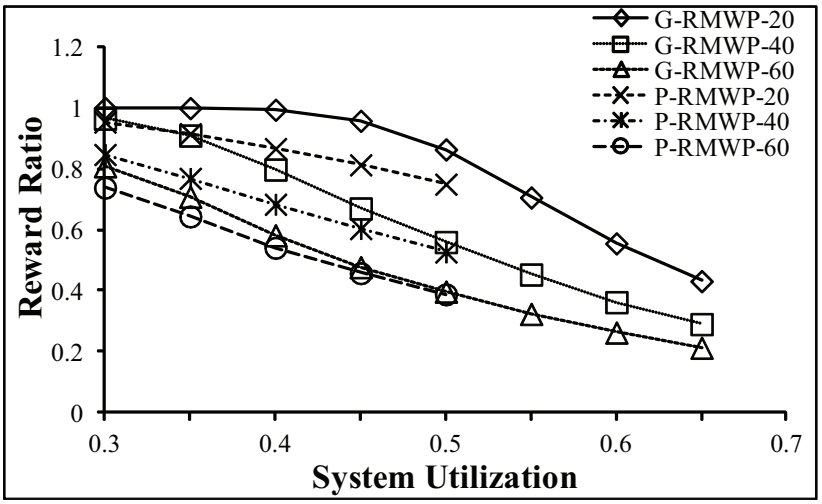


(b) M = 8

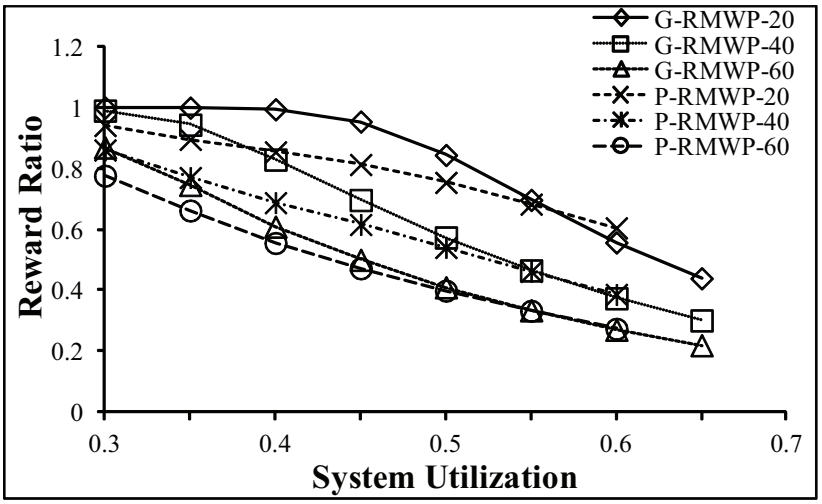


(c) M = 16

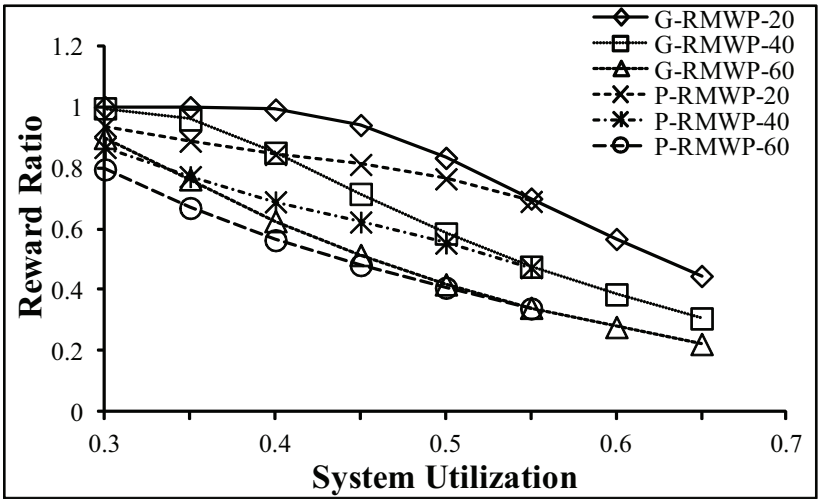
Figure 6.9: Reward ratio on multiprocessors when U_{max} = 1.0



(a) M = 4

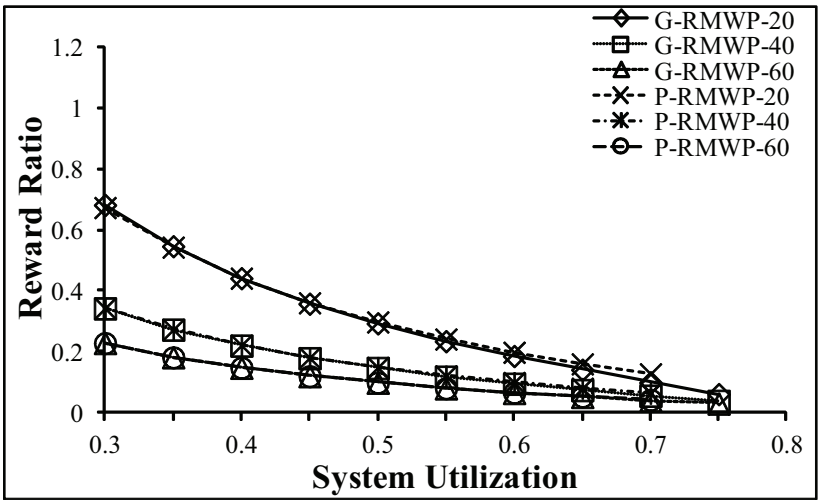


(b) M = 8

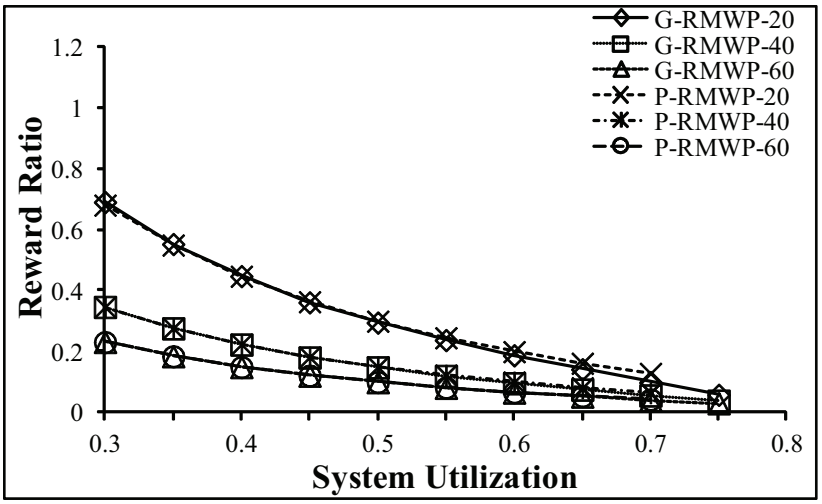


(c) M = 16

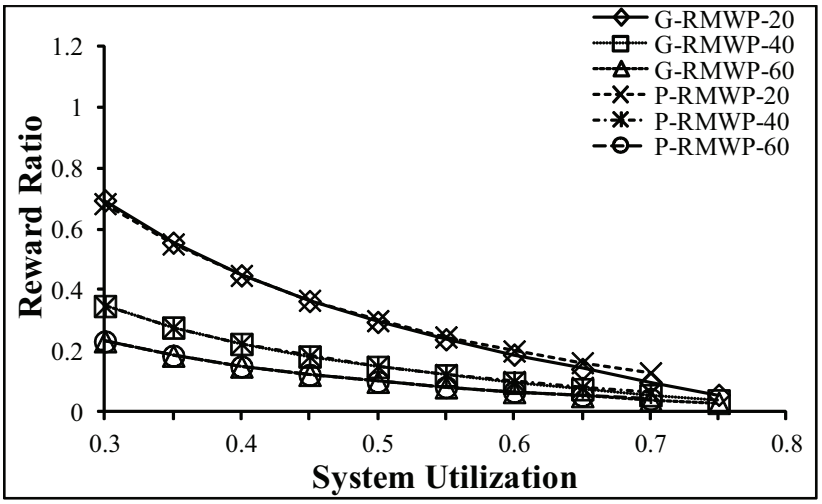
Figure 6.10: Reward ratio on multiprocessors when U_{max} = 0.5



(a) M = 4

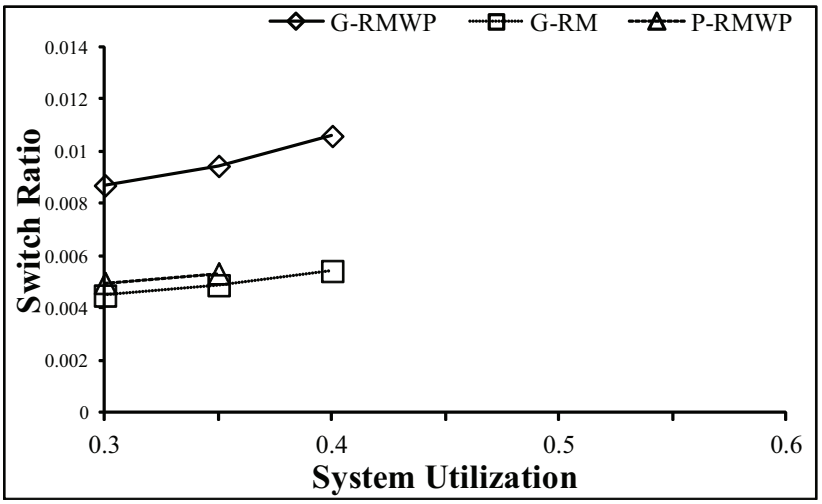


(b) M = 8

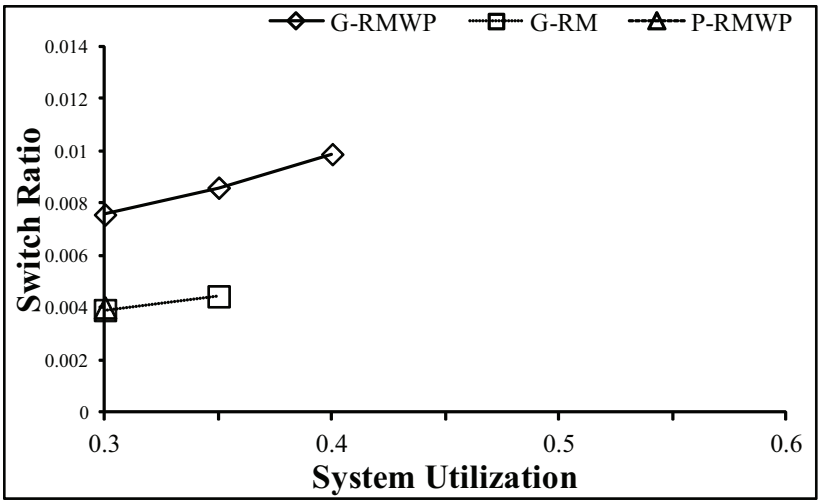


(c) M = 16

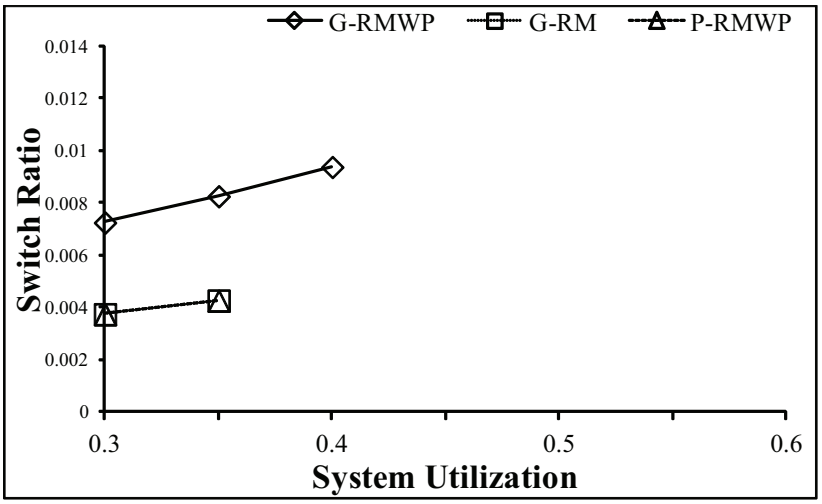
Figure 6.11: Reward ratio on multiprocessors when U_{max} = 0.1



(a) M = 4

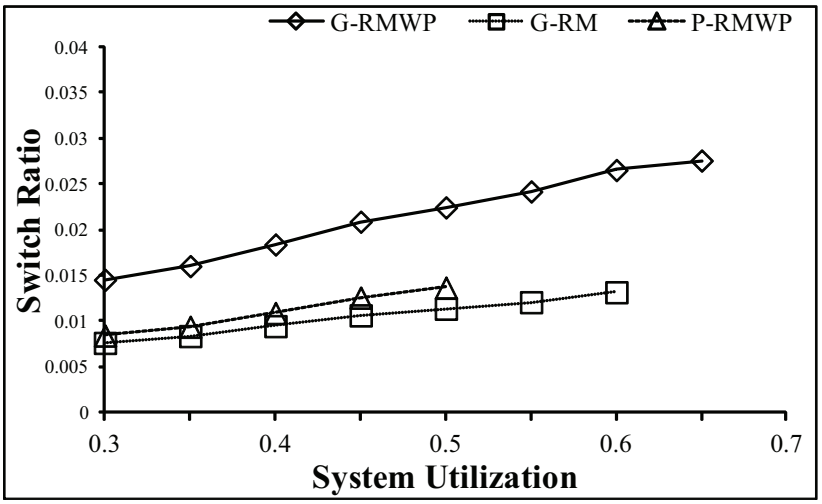


(b) M = 8

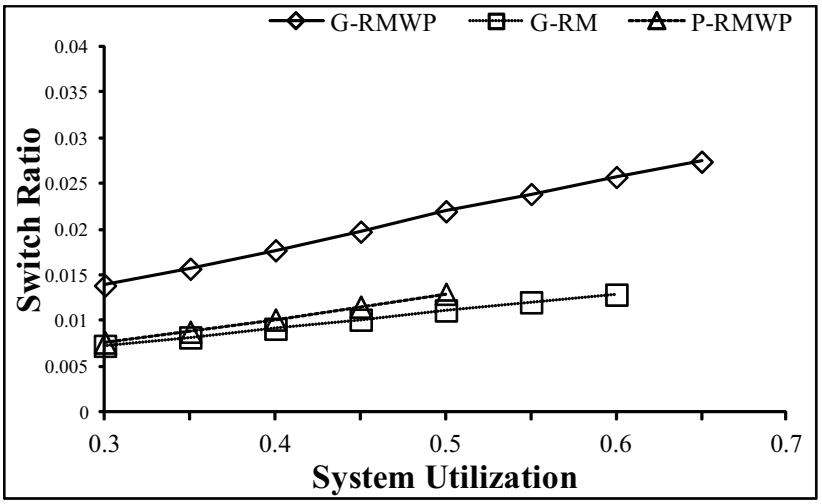


(c) M = 16

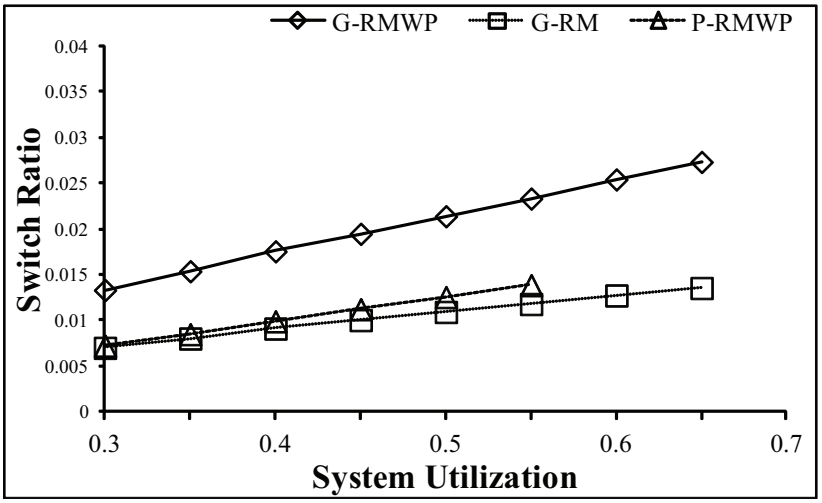
Figure 6.12: Switch ratio on multiprocessors when U_{max} = 1.0



(a) M = 4

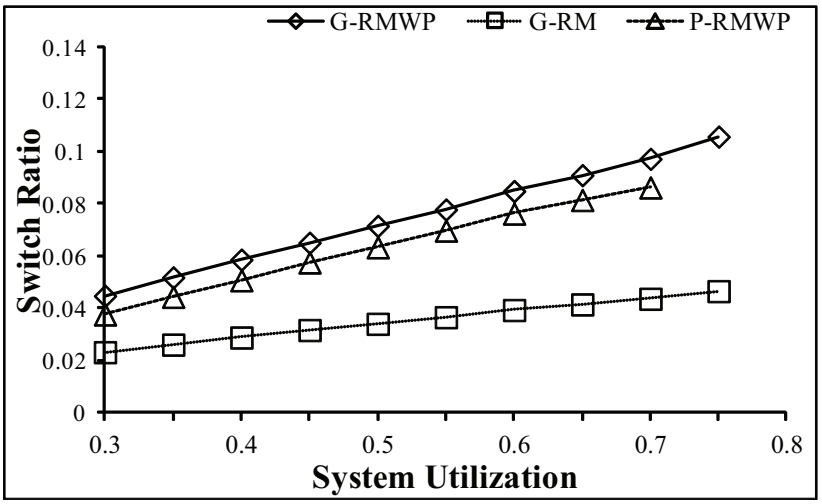


(b) M = 8

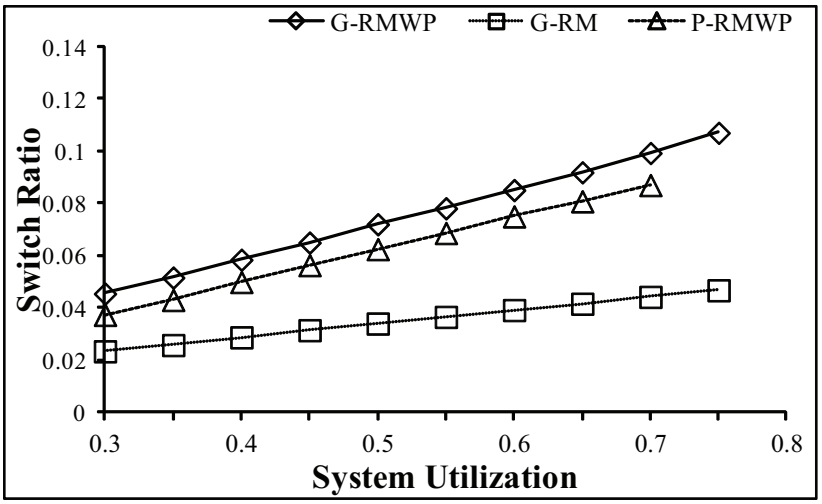


(c) M = 16

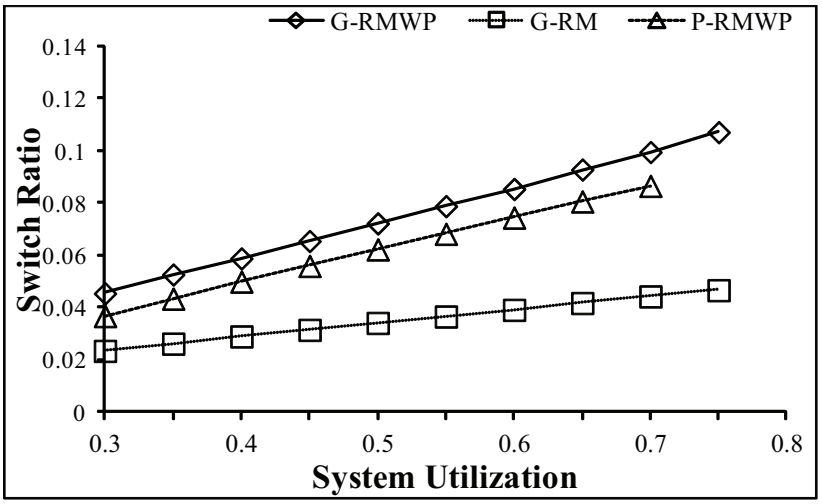
Figure 6.13: Switch ratio on multiprocessors when U_{max} = 0.5



(a) M = 4

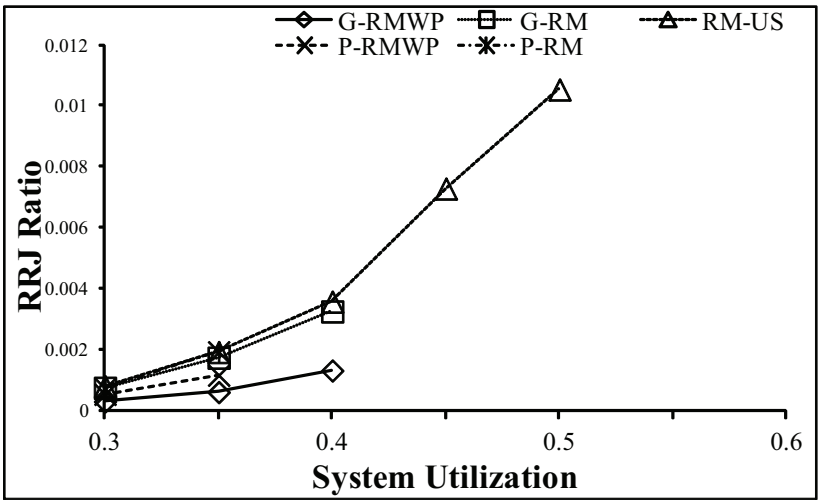


(b) M = 8

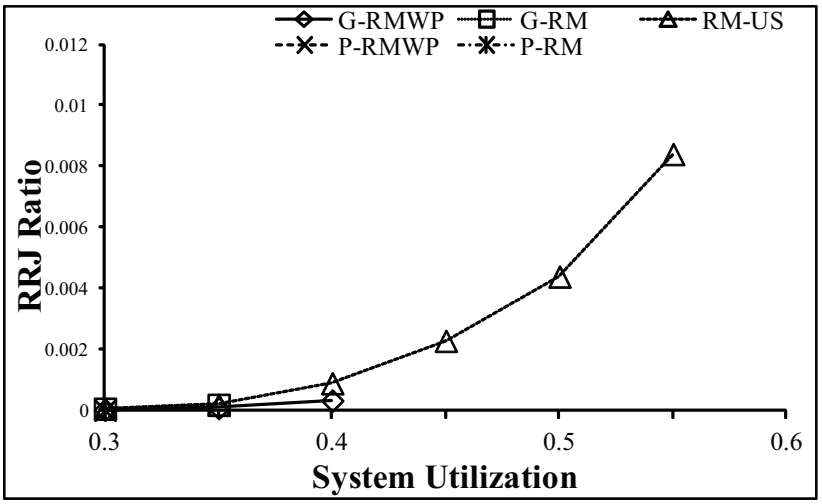


(c) M = 16

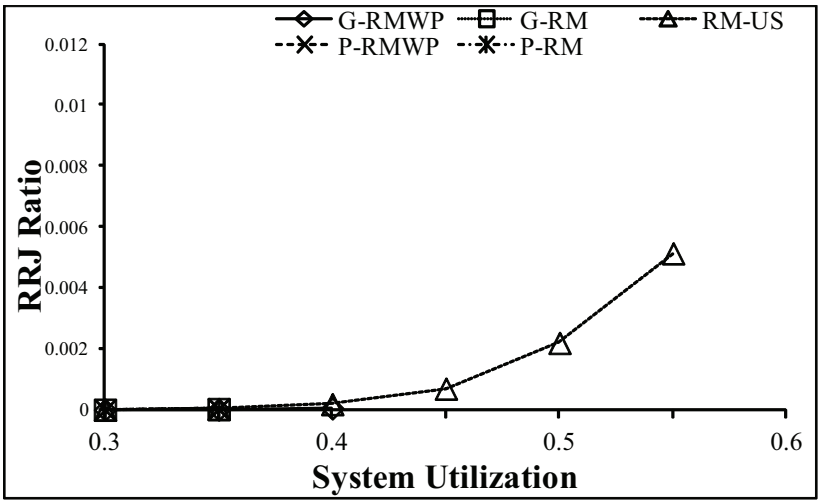
Figure 6.14: Switch ratio on multiprocessors when U_{max} = 0.1



(a) M = 4

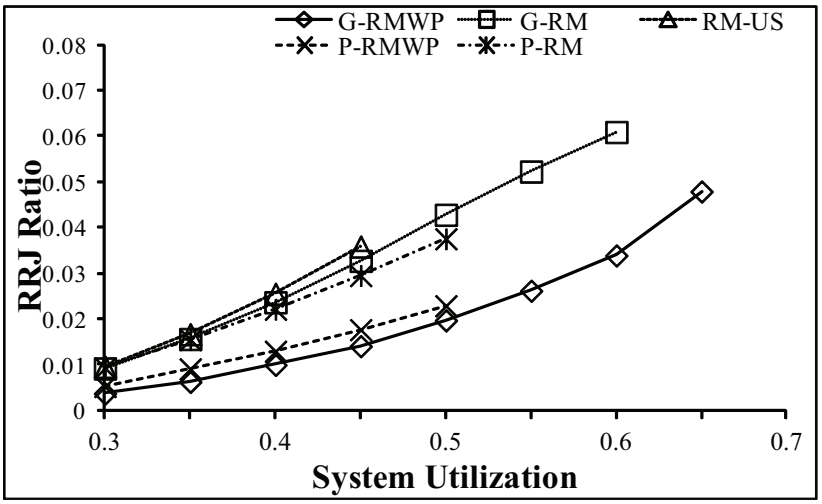


(b) M = 8

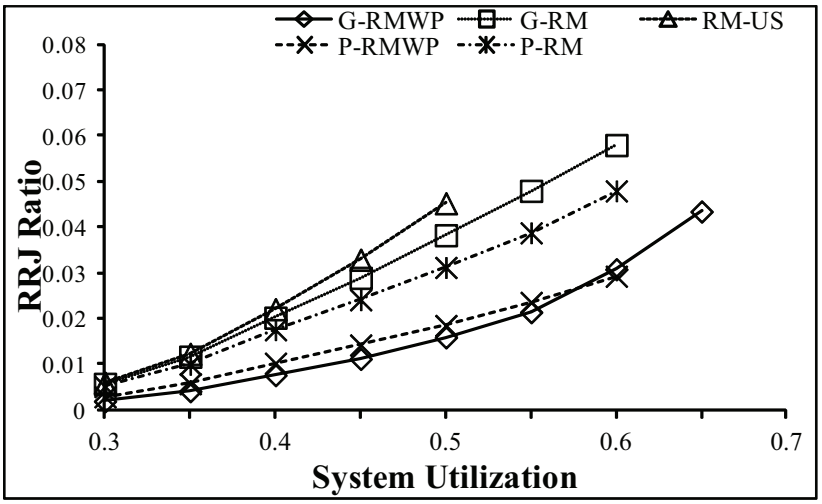


(c) M = 16

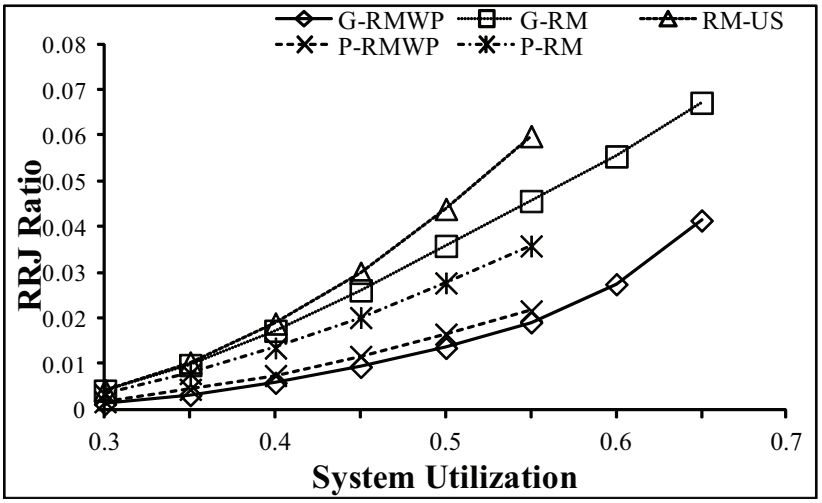
Figure 6.15: RRJ ratio on multiprocessors when U_{max} = 1.0



(a) M = 4

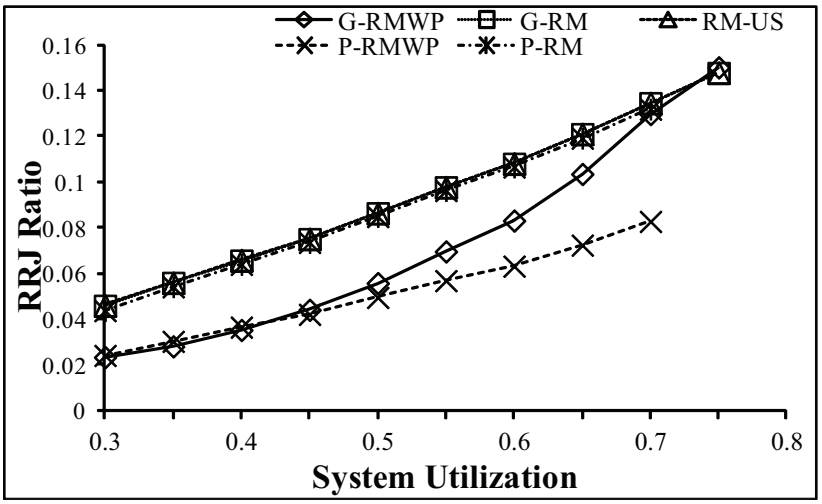


(b) M = 8

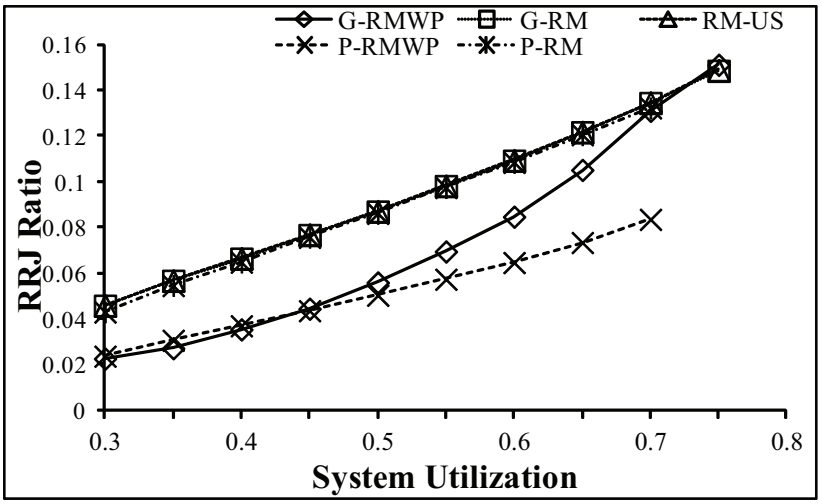


(c) M = 16

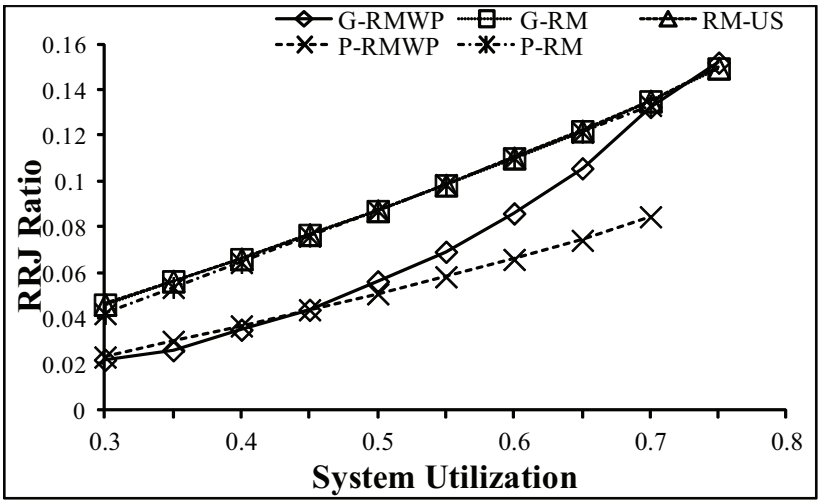
Figure 6.16: RRJ ratio on multiprocessors when U_{max} = 0.5



(a) M = 4

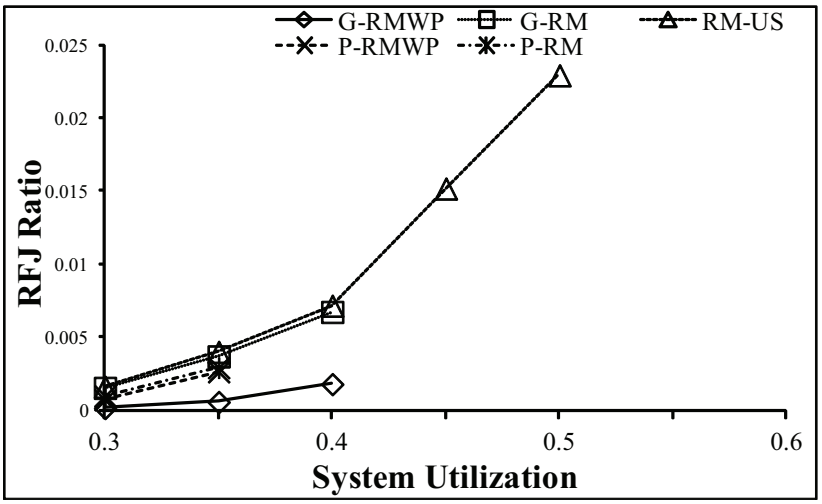


(b) M = 8

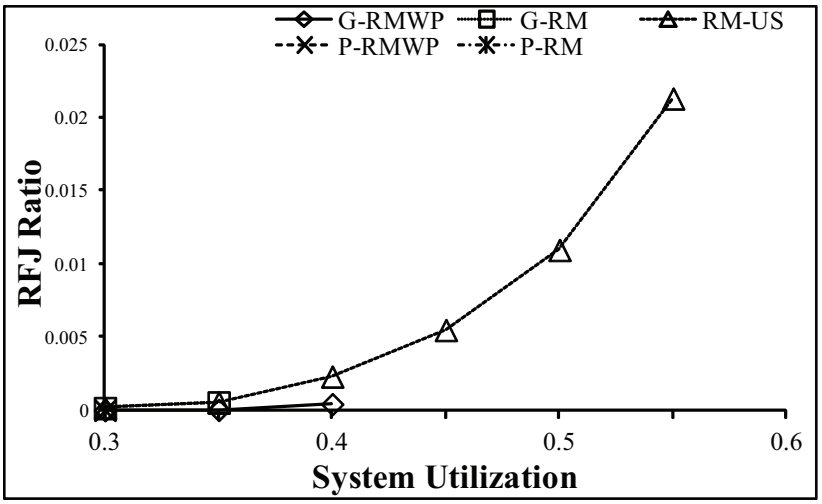


(c) M = 16

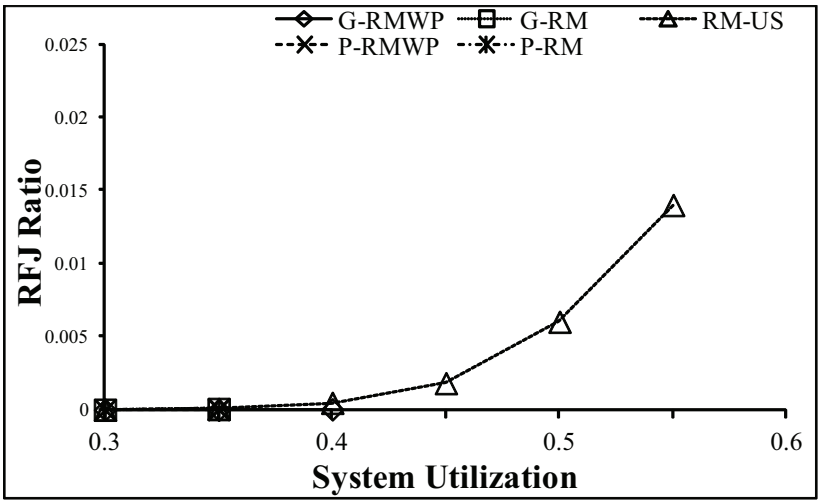
Figure 6.17: RRJ ratio on multiprocessors when U_{max} = 0.1



(a) M = 4

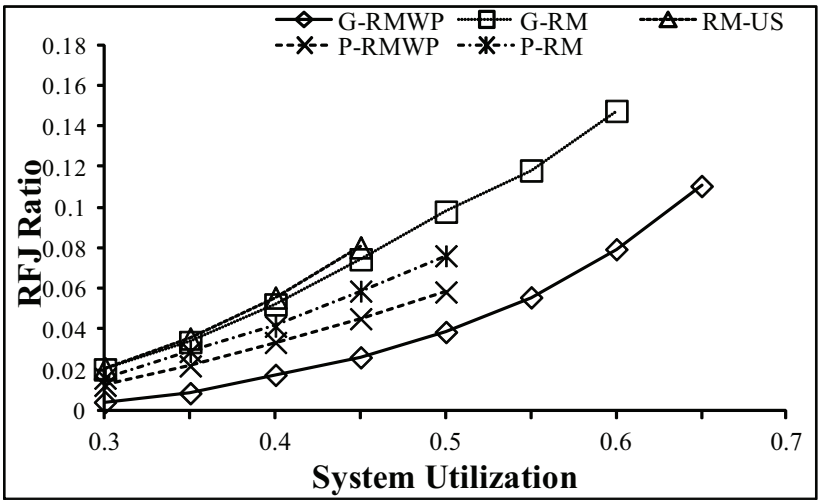


(b) M = 8

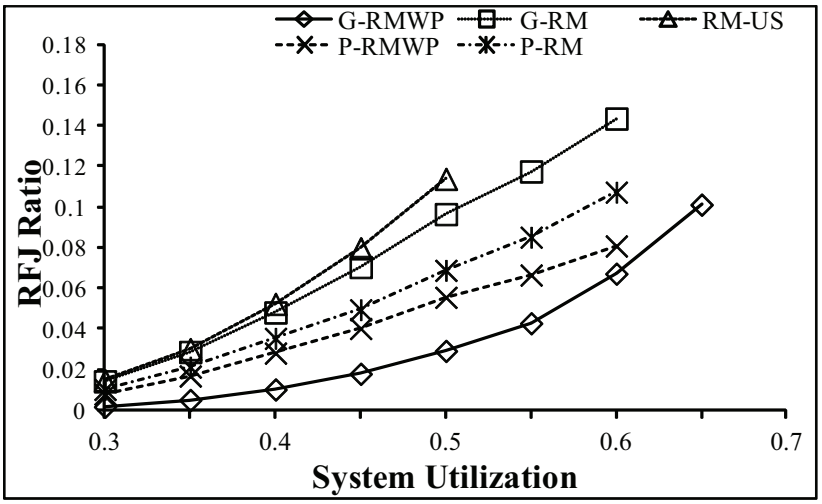


(c) M = 16

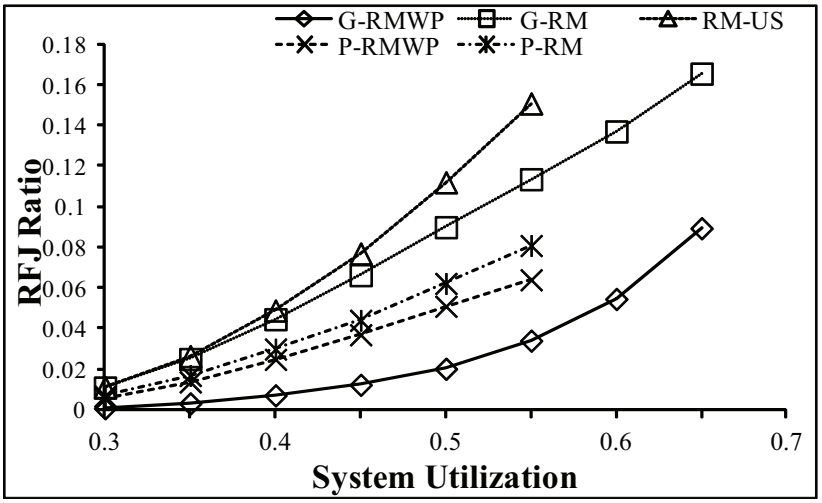
Figure 6.18: RFJ ratio on multiprocessors when U_{max} = 1.0



(a) M = 4

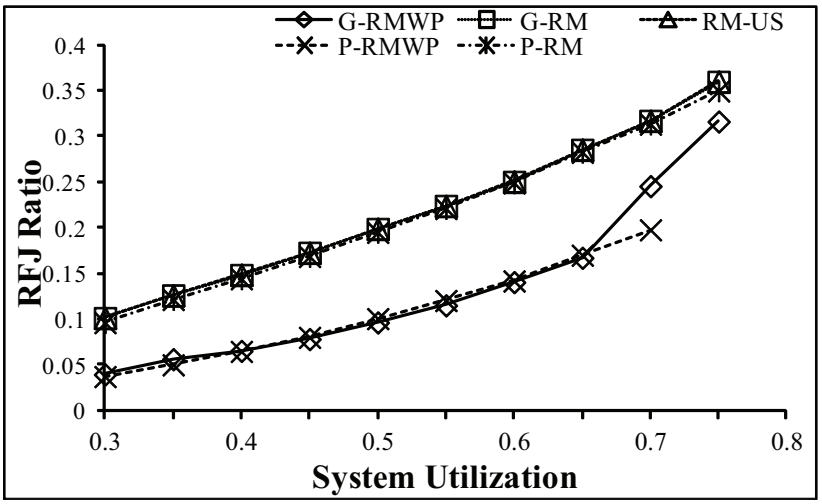


(b) M = 8

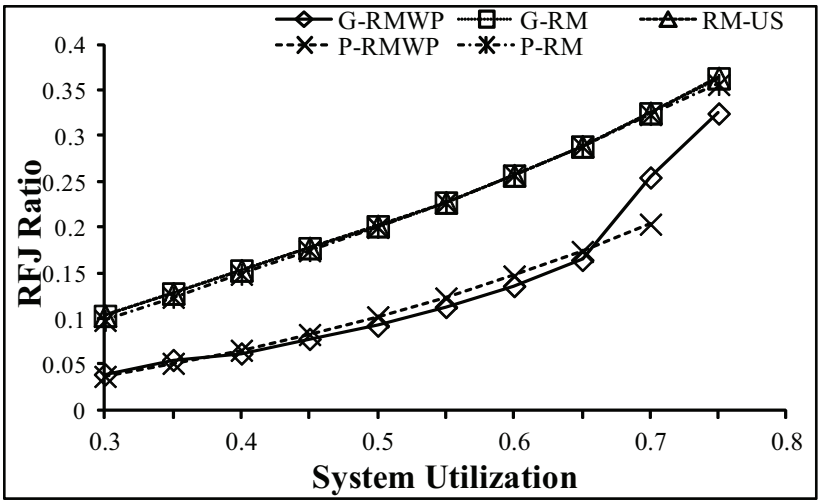


(c) M = 16

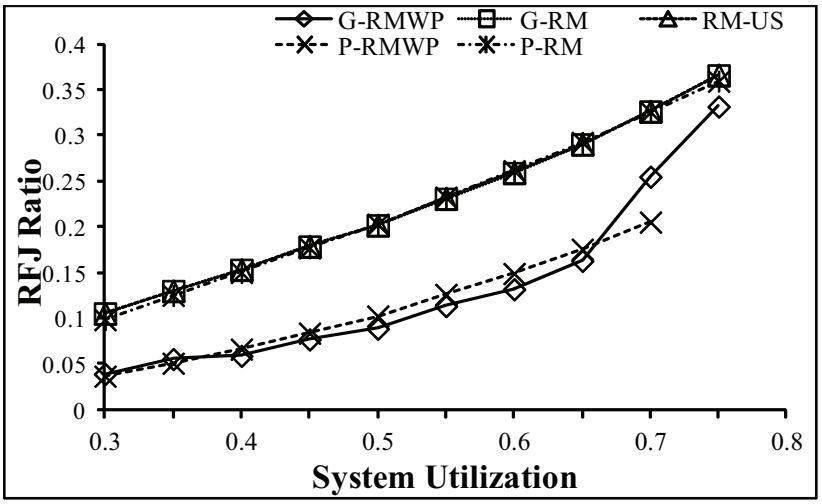
Figure 6.19: RFJ ratio on multiprocessors when U_{max} = 0.5



(a) M = 4

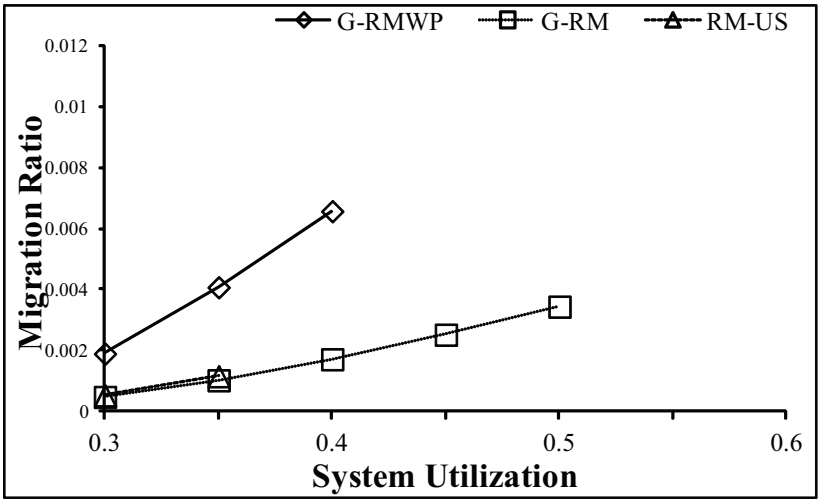


(b) M = 8

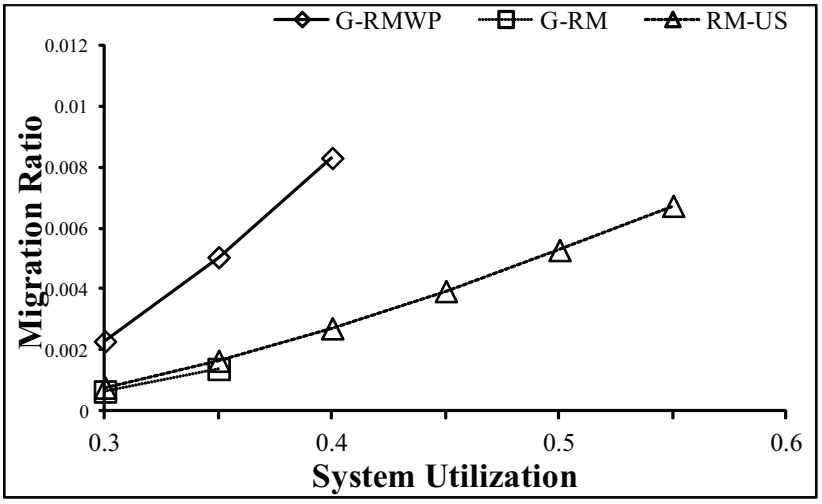


(c) M = 16

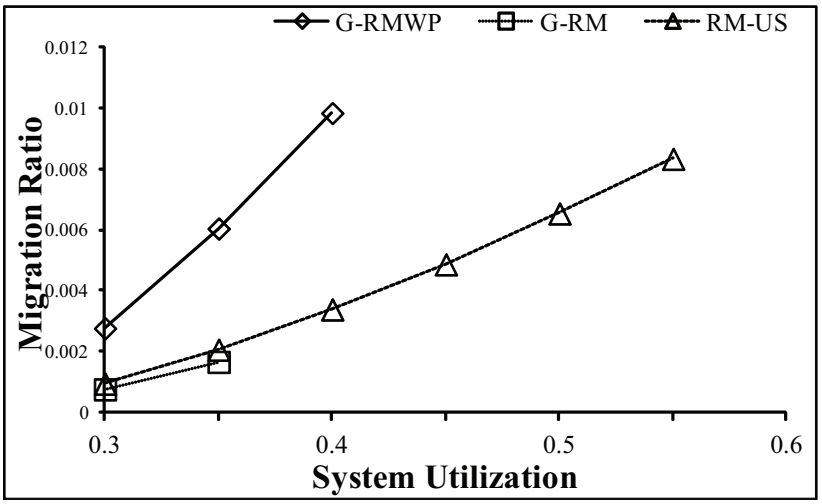
Figure 6.20: RFJ ratio on multiprocessors when U_{max} = 0.1



(a) M = 4

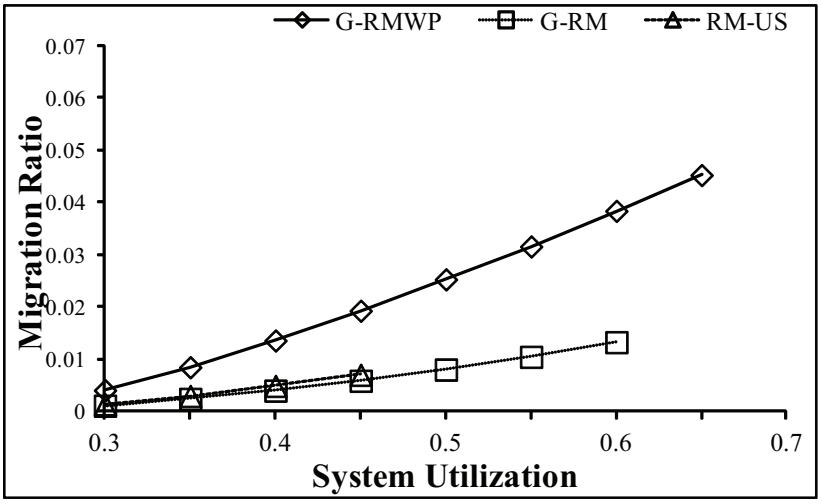


(b) M = 8

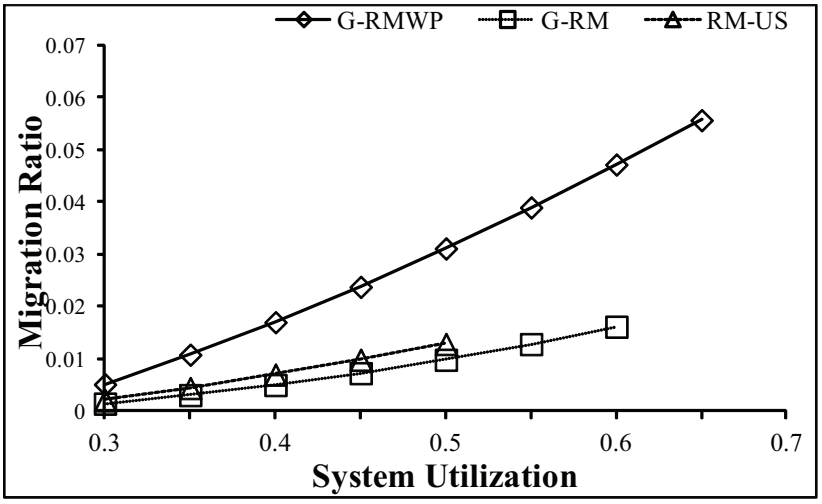


(c) M = 16

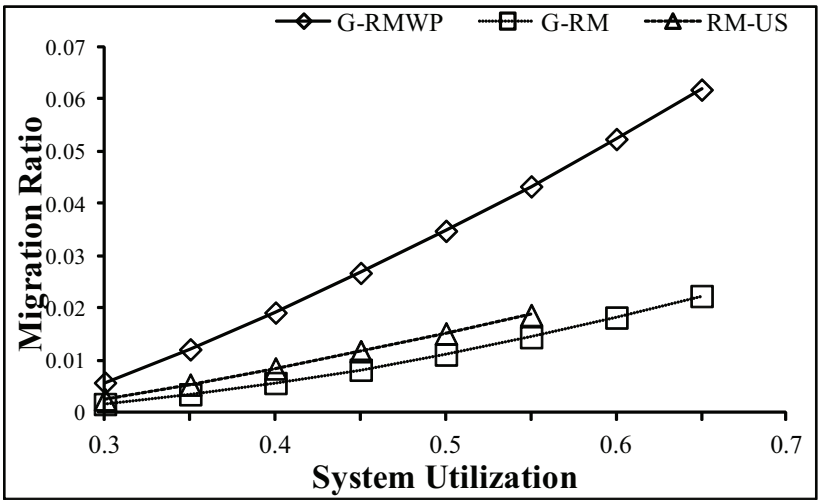
Figure 6.21: Migration ratio on multiprocessors when U_{max} = 1.0



(a) M = 4

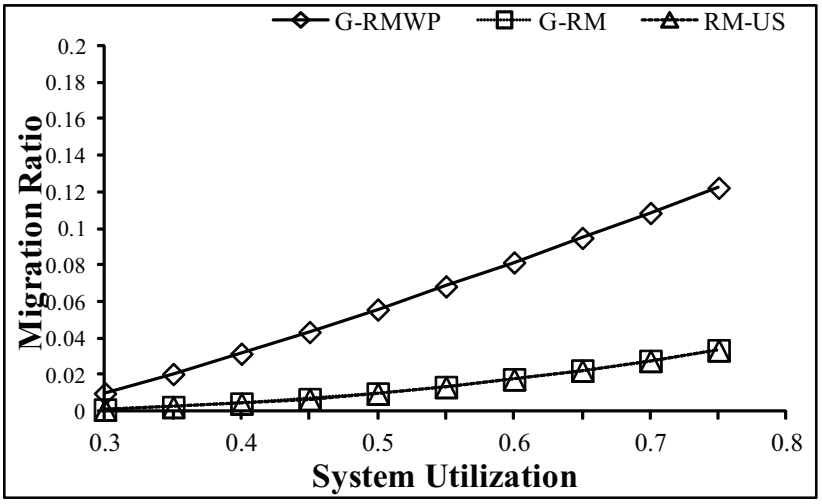


(b) M = 8

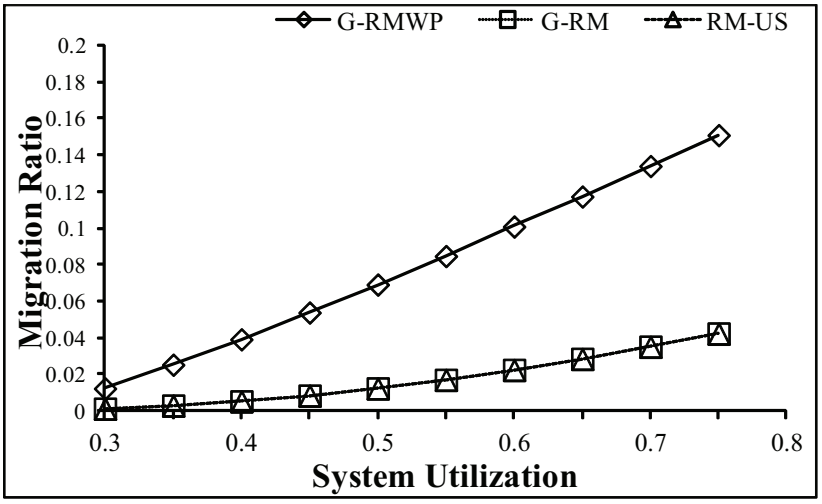


(c) M = 16

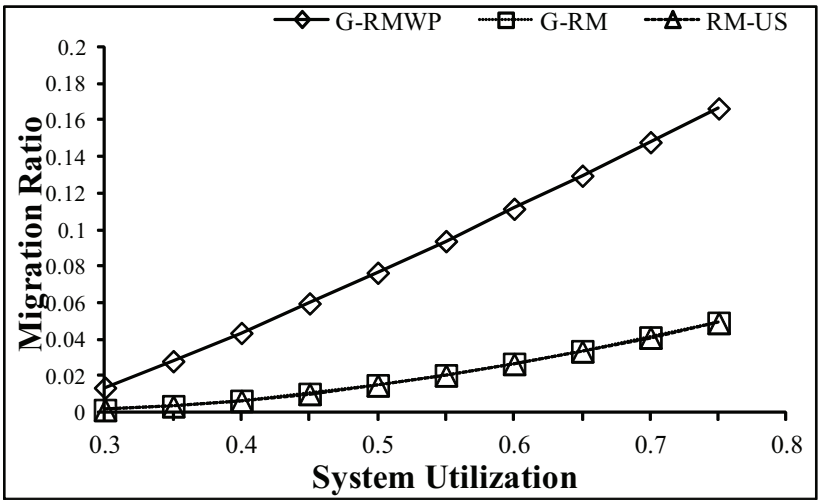
Figure 6.22: Migration ratio on multiprocessors when U_{max} = 0.5



(a) M = 4



(b) M = 8



(c) M = 16

Figure 6.23: Migration ratio on multiprocessors when U_{max} = 0.1

G-RM algorithm is equal to that of the RM-US algorithm because the G-RM algorithm generates the same schedule as the RM-US algorithm when $U_{max} = 0.1 < M/(3M - 2)$. On the other hand, the P-RMWP algorithm has the lowest success ratio in all evaluated algorithms.

Figures 6.9, 6.10 and 6.11 show the reward ratios when $U_{max} = 1.0$, $U_{max} = 0.5$ and $U_{max} = 0.1$ respectively. When the utilization of each optional part is higher and higher, the reward ratio is lower and lower. Interestingly, when the system utilization is lower than 0.5, the reward ratios of the G-RMWP-20, G-RMWP-40 and G-RMWP-60 algorithms in Figure 6.10 are higher than those in Figure 6.9 respectively. If U_{max} is too high like Figure 6.9, the degradation of the reward ratio occurs because the interval to execute each optional part is decreased. If the utilization of the task is 1, then the task cannot execute their optional parts. In Figure 6.9, the G-RMWP algorithm has higher reward ratio than the P-RMWP algorithm. In contrast, in Figure 6.10, the P-RMWP-20, P-RMWP-40 and P-RMWP-60 algorithms have higher reward ratio than the G-RMWP-20, G-RMWP-40 and G-RMWP-60 algorithms respectively when the system utilization is higher than 0.55. In Figure 6.11, the reward ratios of the P-RMWP-20, P-RMWP-40 and P-RMWP-60 algorithms are slightly higher than those of the G-RMWP-20, G-RMWP-40 and G-RMWP-60 algorithms respectively.

Figures 6.12, 6.13 and 6.14 show the switch ratios when $U_{max} = 1.0$, $U_{max} = 0.5$ and $U_{max} = 0.1$ respectively. In all results, the switch ratio of the G-RM algorithm is approximately equal to the switch ratios of the RM-US and P-RM algorithms so that the all results of the RM-US and P-RM algorithms are omitted. The switch ratio of the G-RMWP algorithm has approximately twice as much as that of the G-RM algorithm, which is the similar trend of the RMWP and RM algorithms on uniprocessors, as shown in Figure 6.3. Interestingly, in Figure 6.12, the switch ratio of the P-RMWP algorithm is approximately equal to that of the G-RM algorithm. In contrast, in Figure 6.13, the switch ratio of the P-RMWP algorithm is slightly higher than those of the G-RM algorithm. In Figure 6.14, the switch ratio of the P-RMWP algorithm is also higher than those of the G-RM algorithm, like Figure 6.13. If U_{max} is higher and higher, the switch ratio of the P-RMWP algorithm is gradually equal to that of the G-RM algorithm.

Figures 6.15, 6.16 and 6.17 show the RRJ ratios when $U_{max} = 1.0$, $U_{max} = 0.5$ and $U_{max} = 0.1$ respectively. In Figures 6.15 and 6.16, the RRJ ratio of the G-RMWP algorithm is lowest in all evaluated algorithms. In Figure 6.17, the RRJ ratio of the P-RMWP algorithm is lowest in all evaluated algorithms. That is to say, the G-RMWP and P-RMWP algorithms have lower RRJ ratio than the G-RM, RM-US and P-RM algorithms. The RM-US algorithm has dramatically higher RRJ ratio than other algorithms in Figures 6.15 and 6.16 due to the technique of the utilization separation. This explains that the technique of the utilization separation has the disadvantage of high RRJ ratio.

Figures 6.18, 6.19 and 6.20 show the RFJ ratios when $U_{max} = 1.0$, $U_{max} = 0.5$ and $U_{max} = 0.1$ respectively. The simulation results of the RFJ ratios are similar to those of the RRJ ratios, as shown in Figures 6.15, 6.16 and 6.17. That is to say, in Figures 6.18 and 6.19, the RFJ ratio of the G-RMWP algorithm is lowest and that of the RM-US algorithm is highest in all evaluated algorithms. In addition, in Figure 6.20, the RFJ ratio of the P-RMWP algorithm is lowest in all evaluated algorithms.

From the simulation results of the RRJ and RFJ ratios, the G-RMWP and P-RMWP algorithms achieve low-jitter in all evaluated algorithms. In addition, the G-RMWP algorithm has lower jitter than the P-RMWP algorithm when $U_{max} = 1.0$ and $U_{max} = 0.5$.

Figures 6.21, 6.22 and 6.23 show the migration ratios when $U_{max} = 1.0$, $U_{max} = 0.5$ and $U_{max} = 0.1$ respectively. In all results, the G-RMWP algorithm has higher migration ratio than the G-RM and RM-US algorithms. In Figures 6.21 and 6.22, the migration ratio of the

RM-US algorithm is approximately equal to that of the G-RM algorithm. That is to say, the technique of the utilization separation does not generate the additional migration cost.

6.3 Discussion of Simulation Studies

Considering all simulation results, this dissertation discusses which algorithm is well suited to practical imprecise computation.

From the simulation results on uniprocessors, the RM algorithm does not consider the remaining time to improve the quality of service. In dynamic real-time environments, the WCET of each task usually tends to be overestimated so that the RM algorithm is not well suited to practical imprecise computation. In the RMWP and M-FWP algorithms, this dissertation discusses the trade-offs between reward ratios and jitter to achieve practical imprecise computation. The M-FWP algorithm is well suited to applications requiring high schedulability with non-jitter sensitive tasks because the M-FWP algorithm has higher success ratio than the RMWP algorithm. In contrast, the RMWP algorithm is well suited to dynamic real-time systems because not only is the jitter of the RMWP algorithm not affected by the execution time of each optional part, but also the RMWP algorithm has lower jitter than the M-FWP algorithm. The goal of this dissertation is to achieve real-time scheduling with low-jitter and high schedulability so that the RMWP algorithm is well suited to practical imprecise computation on uniprocessors.

From the simulation results on multiprocessors, the success ratio of the RM-US algorithm is higher than that of the G-RM algorithm when $U_{max} = 1.0$ and lower than that of the G-RM algorithm when $U_{max} = 0.5$. In contrast, the success ratio of the G-RMWP algorithm is higher than or equal to that of the G-RM algorithm by Theorem 7. The success ratio of the P-RMWP algorithm is equal to that of the P-RM algorithm by Theorem 3. The disadvantage of the G-RMWP algorithm is that the G-RMWP algorithm has the highest switch ratio in all evaluated algorithms. In contrast, the switch ratio of the P-RMWP algorithm is approximately as same as that of the G-RM algorithm when $U_{max} = 1.0$. The G-RMWP and P-RMWP algorithms have the novel advantage against the G-RM, RM-US and P-RM algorithms to achieve practical imprecise computation on multiprocessors. In addition, the G-RMWP and P-RMWP algorithms have lower jitter than the G-RM, RM-US and P-RM algorithms. Especially, the jitter of the RM-US algorithm is dramatically high. Therefore, the G-RMWP and P-RMWP algorithms are cost-effective real-time scheduling for practical imprecise computation on multiprocessors.

From the simulation results on uniprocessors and multiprocessors, semi-fixed-priority scheduling is effective for practical imprecise computation in dynamic environments.

6.4 Summary of Simulation Studies

This chapter performs simulation studies on uniprocessors and multiprocessors. The success ratios of semi-fixed-priority scheduling algorithms are higher than or equal to those of fixed-priority scheduling algorithms. The reward ratios of semi-fixed-priority scheduling algorithms are lower than those of dynamic-priority scheduling algorithms. The switch ratios of semi-fixed-priority scheduling algorithms are higher than those of other real-time scheduling algorithms. The jitters of semi-fixed-priority scheduling algorithms are lower than other algorithms and do not depend on the assignable time of the optional part, thanks

to the optional deadline. The migration ratios of semi-fixed-priority scheduling algorithms are higher than those of fixed-priority scheduling algorithms. Considering all simulation results, semi-fixed-priority scheduling algorithms can achieve practical imprecise computation on uniprocessors and multiprocessors. Therefore, semi-fixed-priority scheduling can be strongly adapted to overloaded conditions in dynamic environments.

Chapter 7

Experimental Evaluations

This chapter evaluates the proposed algorithms through experimental evaluations in the RT-Est real-time operating system on an x86 uniprocessor and an x86 multiprocessor. Each practical imprecise task has two mandatory parts and one optional part. The system has the Corei5 750 2.66GHz quad core processor and 2GB DDR3SDRAM 1,333MHz. In all measurements, the RT-Est real-time operating system is compiled with `gcc` version 4.6.1 with the second level of optimization (-O2). The metrics in the experimental evaluations are the following overheads.

- `end_mandatory` function: is called if each task completes its mandatory parts. The detail implementation of `end_mandatory` function is shown in Figure 5.6.
- `end_optional` function: is called if each task completes its optional parts. The detail implementation of `end_optional` function is shown in Figure 5.7.
- `terminate_optional` function: is called if each task terminates its optional parts. The detail implementation of `terminate_optional` function is shown in Figure 5.8.
- scheduler: finds the running task which has the highest priority by the hybrid scheduler as shown in Figure 5.3 or the dual scheduler as shown in Figure 5.4.
- overall: includes the above overheads and the overhead of the interrupt handler. That is to say, the overall overhead is between the time when the interrupt handler is started and the time when the interrupt handler is finished.

Each overhead is measured by ReaD Time Stamp Counter (RDTSC) instruction on x86 processors. In addition, the experimental evaluations do not use cache to measure the worst case overhead. The RDTSC instruction is used at the start and the finishing points to measure each overhead.

In addition, the experimental evaluations measure the jitter of each task, which is called RRJ and RFJ in Equations (3.1) and (3.2) respectively. The RRJ and RFJ ratios are defined as the following equations.

$$\text{RRJ Ratio} = \frac{1}{n}\sum_{i}\frac{RRJ_i}{T_i} \tag{7.1}$$

$$\text{RFJ Ratio} = \frac{1}{n}\sum_{i}\frac{RFJ_i}{T_i} \tag{7.2}$$

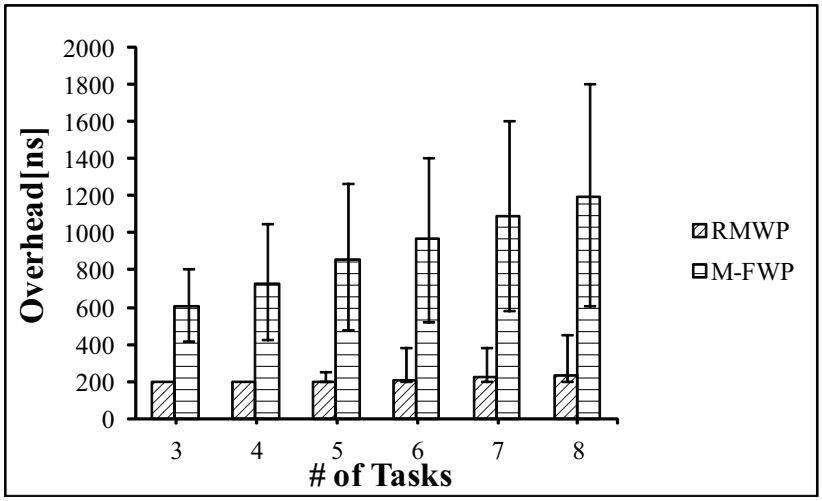


Figure 7.1: Overhead of `end_mandatory` function on an x86 uniprocessor

7.1 Experimental Evaluations on an x86 Uniprocessor

7.1.1 Experimental Setups on an x86 Uniprocessor

The experimental evaluations use only one core on the Corei5 750 processor. In addition, the experimental evaluations use $1,000$ task sets in each system utilization and evaluate the RMWP, M-FWP and RM algorithms. The period T_i of each task τ_i is selected within $[1ms, 2ms, 3ms, ..., 30ms]$. Each utilization U_i is selected from $[0.02, 0.03, 0.04, ..., 0.25]$ and splits U_i into two utilizations which are assigned to m_i^1 and m_i^2 respectively. The utilization of optional part $o_{i,j}/T_i$ is within the range of $[0, 0.3]$. The system utilization U_s is selected from $[0.3, 0.35, 0.4, ..., 0.8]$. The execution time of the k^{th} task set is H_k, which is the hyperperiod of the k^{th} task set.

7.1.2 Experimental Results on an x86 Uniprocessor

Figure 7.1 shows the overhead of `end_mandatory` function on an x86 uniprocessor. The overhead of the RMWP algorithm is approximately constant and low. On the other hand, the overhead of the M-FWP algorithm is dramatically higher than that of the RMWP algorithm because the M-FWP algorithm calculates the assignable time of the optional part dynamically. The overheads of the RMWP algorithm for 3 and 4 tasks are approximately same so that the length of each error bar is very short.

Figure 7.2 shows the overhead of `end_optional` function on an x86 uniprocessor. Unlike Figure 7.1, the overhead of the RMWP algorithm is higher than that of the M-FWP algorithm. In `end_optional` function, the RMWP algorithm enqueues the task to the SQ and the M-FWP algorithm starts to execute its following mandatory part without queueing operations in the RTQ and NRTQ. That is to say, when the task completes its optional part in the M-FWP algorithm, the task executes its following mandatory part immediately.

Figure 7.3 shows the overhead of `terminate_optional` function on an x86 uniprocessor. The average overhead of the RMWP algorithm is lower than that of the M-FWP algorithm. However, the maximum overhead of the RMWP algorithm is usually higher than that of the M-FWP algorithm. The RMWP algorithm checks whether the current time is the

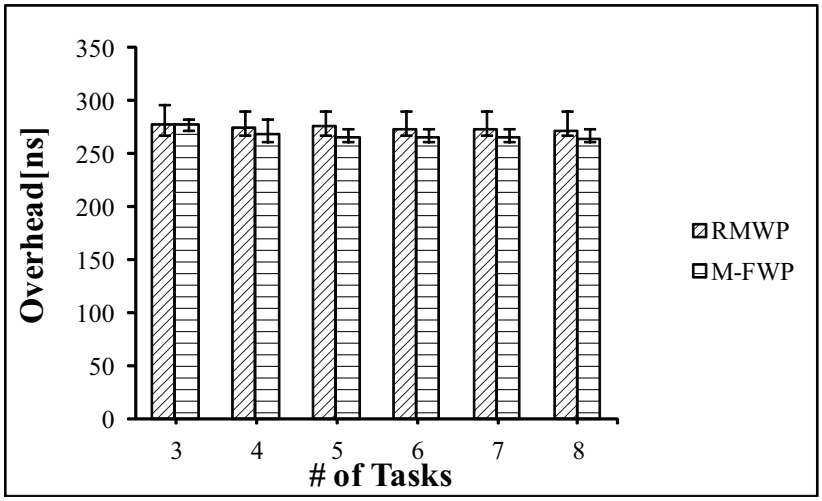


Figure 7.2: Overhead of `end_optional` function on an x86 uniprocessor

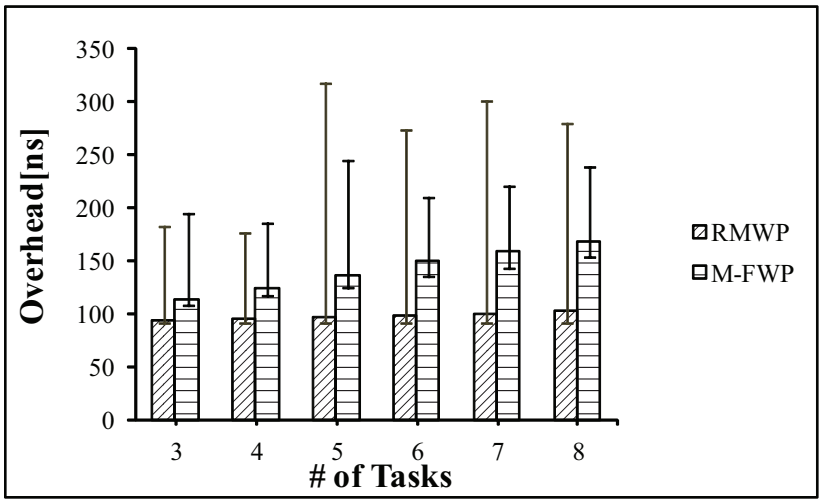


Figure 7.3: Overhead of `terminate_optional` function on an x86 uniprocessor

optional deadline of each task in `terminate_optional` function. If many tasks have the same optional deadlines, the maximum overhead of the RMWP algorithm becomes high. In contrast, the M-FWP algorithm does not need the optional deadline because the M-FWP algorithm calculates the assignable time of the optional part dynamically in `end_mandatory` function, which causes high overhead as shown in Figure 7.1. If the assignable time of the optional part becomes 0, the M-FWP algorithm terminates optional parts so that more than one tasks are not terminated at the same time and the maximum overhead of the M-FWP algorithm becomes low.

Figure 7.4 shows the overhead of scheduler on an x86 uniprocessor. The overhead of the RM algorithm is lower than the overheads of the RMWP and M-FWP algorithms because the RM algorithm does not perform imprecise computation. The overhead of the RMWP algorithm is higher than that of the M-FWP algorithm. This result explain that the overhead of scheduler in the M-FWP algorithm is not so high though the overhead of `end_mandatory` function in the M-FWP algorithm is dramatically higher than that in the RMWP algorithm.

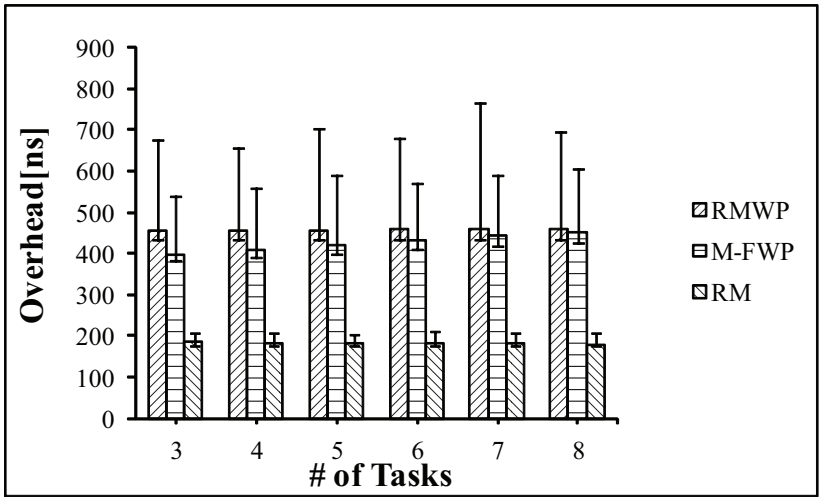


Figure 7.4: Overhead of scheduler on an x86 uniprocessor

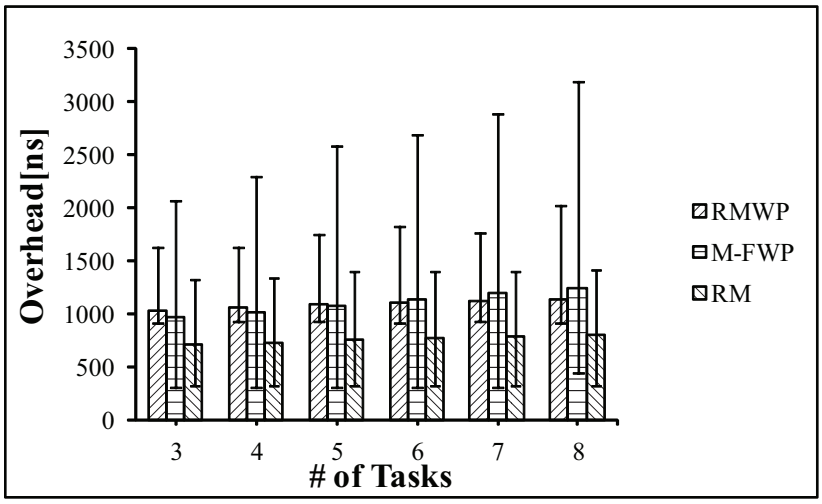


Figure 7.5: Overall overhead on an x86 uniprocessor

Figure 7.5 shows the overall overhead on an x86 uniprocessor. Like Figure 7.4, the overhead of the RM algorithm is also lower than the overheads of the RMWP and M-FWP algorithms. The average overhead of the RMWP algorithm is approximately as same as that of the M-FWP algorithm. On the other hand, the maximum overhead of the RMWP algorithm is dramatically lower than that of the M-FWP algorithm due to the overhead of `end_mandatory` function as shown in Figure 7.1.

Figure 7.6 shows the RRJ ratio on an x86 uniprocessor. The RRJ ratio of the RMWP algorithm is lowest in all evaluated algorithms. The RRJ ratio of the RM algorithm is higher than that of the RMWP algorithm. Especially, the RRJ ratio of the M-FWP algorithm is dramatically high for 7 and 8 tasks because the M-FWP algorithm has high RRJ ratio in high system utilization like the simulation result, as shown in Figure 6.4.

Figure 7.7 shows the RFJ ratio on an x86 uniprocessor. Like Figure 7.6, the RFJ ratio of the RMWP algorithm is lowest in all evaluated algorithms. The RFJ ratio of the RM algorithm is also higher than that of the RMWP algorithm. The RFJ ratio of the M-FWP

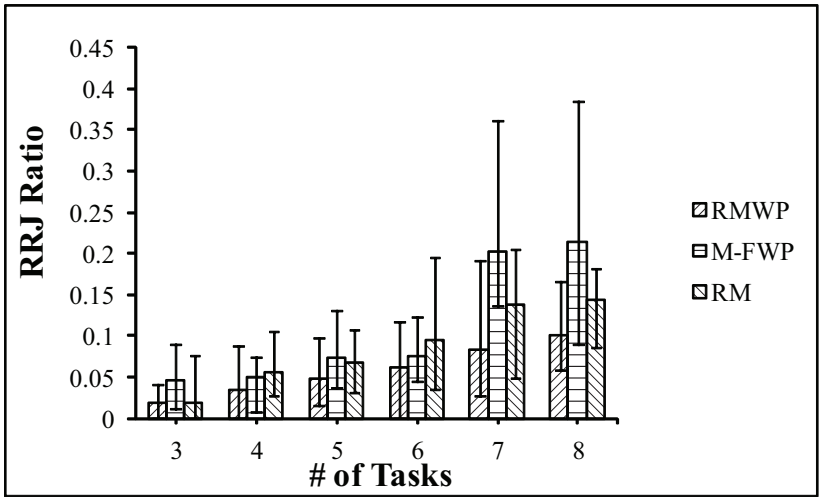


Figure 7.6: RRJ ratio on an x86 uniprocessor

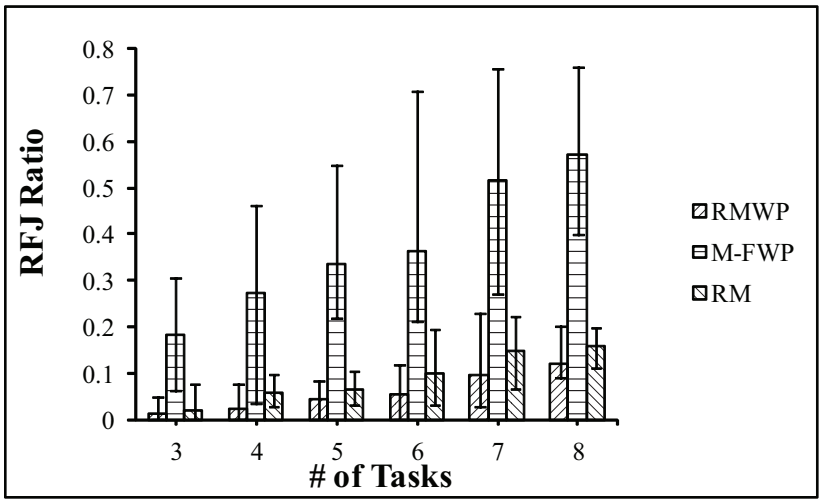


Figure 7.7: RFJ ratio on an x86 uniprocessor

algorithm is dramatically high, regardless of the number of tasks like the simulation result, as shown in Figure 6.5.

From the experimental results of the RRJ and RFJ ratios on an x86 uniprocessor, the RMWP algorithm has low-jitter in all evaluated algorithms.

7.2 Experimental Evaluations on an x86 Multiprocessor

7.2.1 Experimental Setups on an x86 Multiprocessor

The experimental evaluations use $1,000$ task sets in each system utilization and evaluate the G-RMWP, P-RMWP, G-RM and P-RM algorithms. Each U_i is selected within $[0.02, 0.03, 0.04, ..., 0.1]$ and splits U_i into two utilizations which are assigned to m_i^1 and m_i^2 respectively. The period T_i of each task τ_i is selected within $[1ms, 2ms, 3ms, ..., 30ms]$.

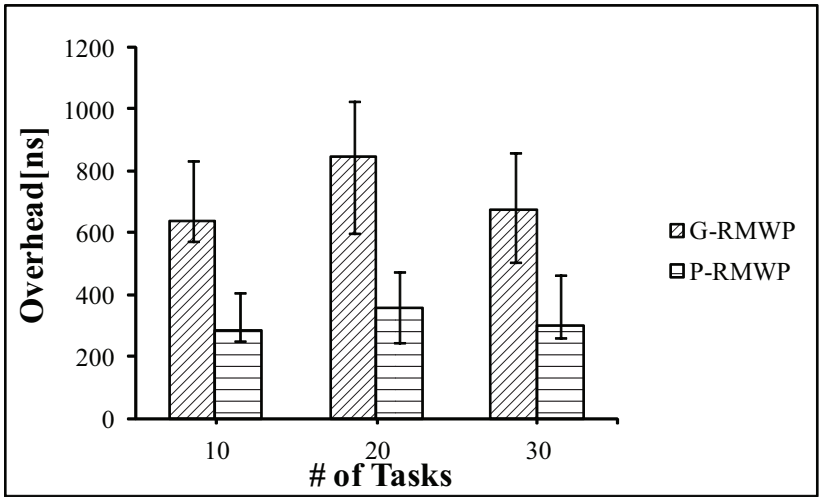


Figure 7.8: Overhead of `end_mandatory` function on an x86 multiprocessor

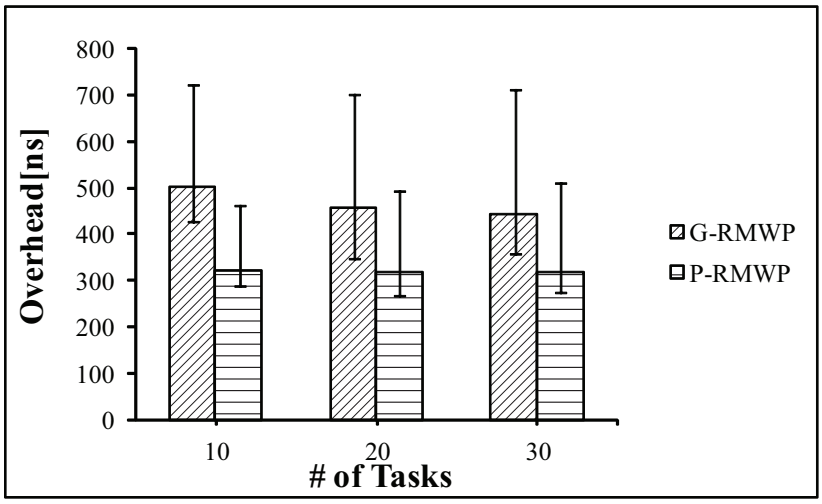


Figure 7.9: Overhead of `end_optional` function on an x86 multiprocessor

The utilization of optional part $o_{i,j}/T_i$ is within the range of $[0, 0.3]$. The system utilization U_s is selected from $[0.15, 0.2, 0.25, ..., 0.6]$. The execution length of the k^{th} task set is H_k, which is the hyperperiod of the k^{th} task set. The task assignment algorithm for the P-RMWP and P-RM algorithms is the next-fit heuristic to even the number of tasks on each core.

7.2.2 Experimental Results on an x86 Multiprocessor

Figure 7.8 shows the overhead of `end_mandatory` function. The overhead of the G-RMWP algorithm is higher than that of the P-RMWP algorithm. In the P-RMWP algorithm, the enqueue and dequeue operations are performed in each ready queue on each core. In contrast, in the G-RMWP algorithm, the enqueue and dequeue operations are performed in a single global queue so that the overhead of the G-RMWP algorithm becomes high.

Figure 7.9 shows the overheads of `end_optional` functions. Like Figure 7.8, the G-RMWP algorithm has higher average and maximum overheads than the P-RMWP algo-

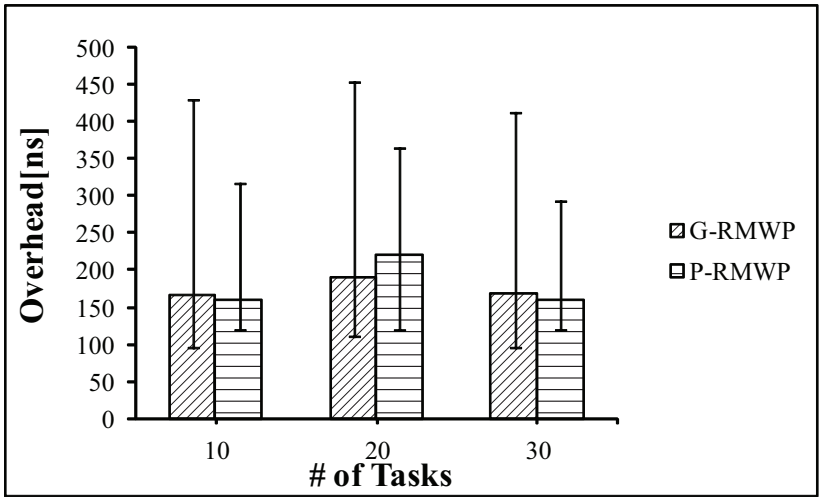


Figure 7.10: Overhead of `terminate_optional` function on an x86 multiprocessor

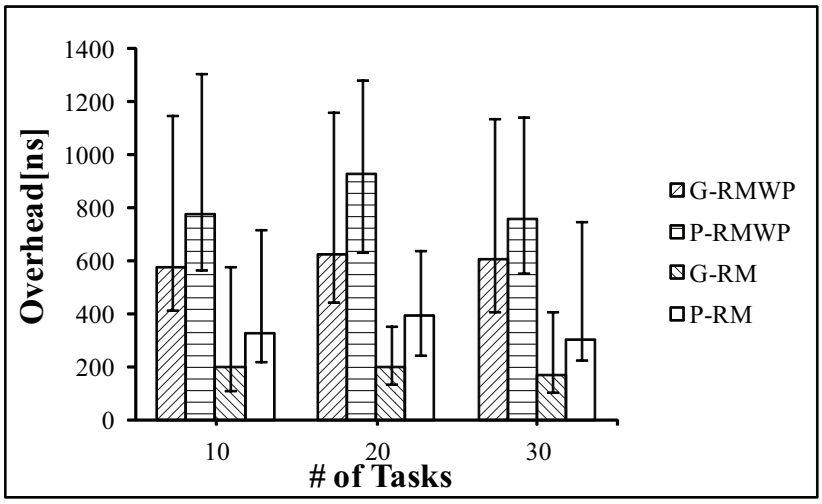


Figure 7.11: Overhead of scheduler on an x86 multiprocessor

rithm. The overhead of `end_optional` function is usually lower than that of `end_mandatory` function. Because the number of enqueue and dequeue operations in `end_mandatory` function is more than that in `end_optional` function.

Figure 7.10 shows the overhead of `terminate_optional` function. The overhead of `terminate_optional` function is lower than the overheads of `end_mandatory` and `end_optional` functions because `terminate_optional` function performs the enqueue and dequeue operations only if there is a task, the optional deadline of which expires at the current time. Otherwise, `terminate_optional` function does not perform the enqueue and dequeue operations.

Figure 7.11 shows the overhead of scheduler. Interestingly, the overhead of scheduler in the G-RMWP algorithm is lower than that in the P-RMWP algorithm. Similarly, the overhead of the G-RM algorithm is also lower than that of the P-RM algorithm. As a result, global scheduling has lower overhead of scheduler than partitioned scheduling, thanks to the dual scheduler.

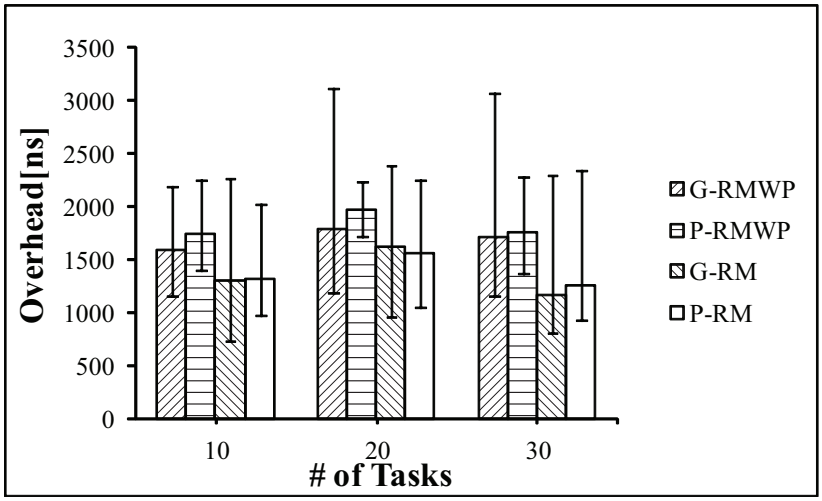


Figure 7.12: Overall overhead on an x86 multiprocessor

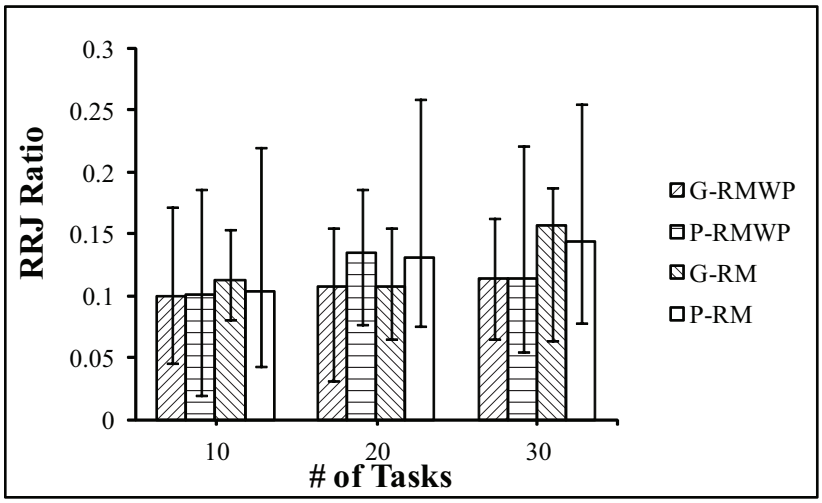


Figure 7.13: RRJ ratio on an x86 multiprocessor

Figure 7.12 shows the overall overhead. Like Figure 7.11, the overall overhead is approximately constant regardless of the number of tasks. That is to say, the dual scheduler is scalable and achieves low overhead. The overall overhead in the G-RMWP and P-RMWP algorithms are approximately 2,000-3,000 *ns* which is very lower than Linux-based experimental evaluations on multiprocessors, as previously described in Section 2.5.

Figure 7.13 shows the RRJ ratio on an x86 multiprocessor. The RRJ ratios of the G-RMWP and P-RMWP algorithms are lower than those of the G-RM and P-RM algorithms respectively, like Figure 7.6. Especially, the maximum RRJ ratio of the P-RM algorithm is highest in all evaluated algorithm.

Figure 7.14 shows the RFJ ratio on an x86 multiprocessor. The RFJ ratios of the G-RMWP and P-RMWP algorithms are approximately as same as those of the G-RM and P-RM algorithms. Because the G-RMWP and P-RMWP algorithms perform the additional operations in `end_mandatory`, `end_optional` and `terminate_optional` functions so that the finishing time of each task tends to be fluctuated.

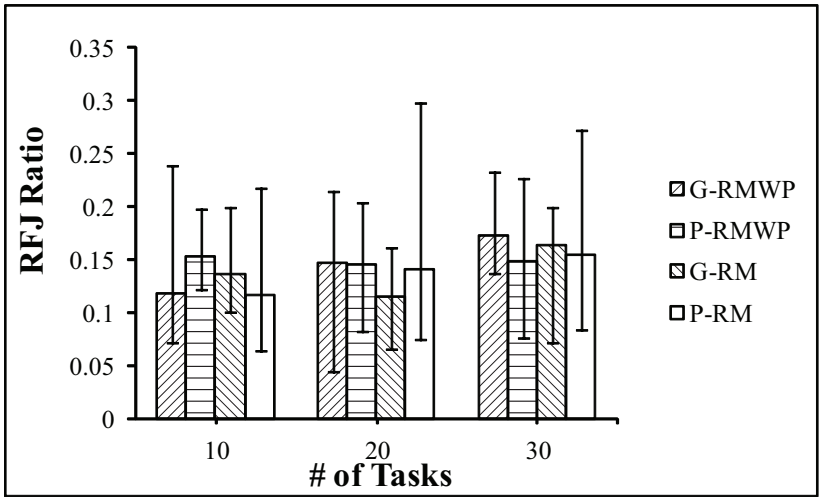


Figure 7.14: RFJ ratio on an x86 multiprocessor

From the experimental results of the RRJ and RFJ ratios on an x86 multiprocessor, the G-RMWP and P-RMWP algorithms have lower jitter than or approximately as same jitter as the G-RM and P-RM algorithms.

7.3 Discussion of Experimental Evaluations

Considering all results of experimental evaluations, this dissertation discusses which algorithm is well suited to practical imprecise computation.

From the experimental results on an x86 uniprocessor, the RM algorithm has the lowest overhead in all evaluated algorithms. However, the RM algorithm cannot make use of the remaining time to improve the reward. The M-FWP algorithm has higher overall overhead than the RMWP algorithm. The RMWP algorithm has approximately constant overall overhead regardless of the number of tasks like the RM algorithm and can make use of remaining time to improve reward, unlike the RM algorithm. In addition, the RMWP algorithm has the lowest jitter in all evaluated algorithms. Therefore, the RMWP algorithm is well suited to practical imprecise computation.

From the experimental results on an x86 multiprocessor, semi-fixed-priority scheduling has comparable overhead to fixed-priority scheduling. The G-RMWP algorithm has lower overhead than the P-RMWP algorithm, thanks to the dual scheduler. In addition, the G-RMWP and P-RMWP algorithms have lower jitter than or approximately as same jitter as the G-RM and P-RM algorithms. Therefore, the G-RMWP and P-RMWP algorithms are well suited to practical imprecise computation.

7.4 Comparison of Simulation and Experimental Results

The simulation results show that the RMWP algorithm has both lower jitter and higher schedulability than the RM algorithm on uniprocessors. The G-RMWP and P-RMWP algorithms also have lower jitter than and at least as same schedulability as the G-RM and P-RM algorithms respectively on multiprocessors. The experimental results show that the RMWP, G-RMWP and P-RMWP algorithms have slightly higher overhead than the RM, G-RMWP

and P-RMWP algorithms respectively. In addition, the jitters of the RMWP, G-RMWP and P-RMWP algorithms are lower than or approximately as same as those of the RM, G-RM and P-RM algorithms respectively. In contrast, the M-FWP algorithm has higher overhead than the RMWP algorithm on uniprocessors. Unfortunately, the M-FWP algorithm does not support multiprocessor real-time scheduling. Therefore, semi-fixed-priority scheduling is an effective technique for practical imprecise computation.

7.5 Summary of Experimental Evaluations

This chapter performs experimental evaluations in the RT-Est real-time operating system on an x86 uniprocessor and an x86 multiprocessor. The experimental results show that semi-fixed-priority scheduling is a cost-effective technique for practical imprecise computation. The additional overhead of semi-fixed-priority scheduling is tiny compared to that of fixed-priority scheduling. From both simulation and experimental results, semi-fixed-priority scheduling achieves practical imprecise computation and has comparable overhead to fixed-priority scheduling.

Chapter 8

Conclusions

This dissertation presented the research of real-time scheduling and real-time operating system in the practical imprecise computation model on both uniprocessors and multiprocessors. The target computation model called the practical imprecise computation model has more than one mandatory parts and more than one optional parts without causing a critical timing violation.

8.1 Summary of Contributions

The contributions of this dissertation cover both theoretical and practical aspects in real-time systems. Each of the contributions leads to eviction of resource overprovision and contributes to the development of cost-effective real-time systems that can work in overloaded conditions and in various dynamic environments.

The theoretical contributions are to present semi-fixed-priority scheduling on uniprocessors and multiprocessors. The first theoretical contribution is to analyze the schedulability of semi-fixed-priority scheduling which is higher than or equal to that of fixed-priority scheduling. The second theoretical contribution is to define the optional deadline of each practical imprecise task. Thanks to the optional deadline, semi-fixed-priority scheduling avoids the deadline miss of the following mandatory parts due to the overrun of the optional parts. Note that mandatory parts in a practical imprecise task can be regarded as tasks with same or different release times in Liu and Layland's model. Therefore, semi-fixed-priority scheduling can schedule a task set, which does not have the time when all tasks are released at the same time. The third theoretical contribution is to support practical imprecise computation on uniprocessors and multiprocessors, thanks to semi-fixed-priority scheduling.

The effectiveness of the proposed algorithms is shown through simulation studies. The simulation results on uniprocessors show that the RMWP algorithm has higher success ratio than the RM algorithm and lower jitter than both the RM and M-FWP algorithms. The jitter of the M-FWP algorithm depends on the assignable time of the optional part. In contrast, the jitter of the RMWP algorithm does not depend on the assignable time of the optional part by the optional deadline. Therefore, the RFJ ratio of the RMWP algorithm is dramatically higher than that of the M-FWP algorithm. The simulation results on multiprocessors show that the G-RMWP algorithm has higher success ratio than the G-RM and RM-US algorithms. In addition, the P-RMWP algorithm has same success ratio as the P-RM algorithm because the P-RMWP algorithm can schedule a task set, which is schedulable by the P-RM algorithm. Therefore, the P-RMWP algorithm can make use of the schedulability test

Table 8.1: Overview of this dissertation

Model	Uniprocessor	Multiprocessor	Imprecise
Liu and Layland [2]	✓	✓	
Imprecise Computation [18, 33]	✓		✓
This dissertation	✓	✓	✓

for the P-RM algorithm. The G-RMWP and P-RMWP algorithms have lower jitter than the G-RM and RM-US algorithms.

The practical contributions are to present the RT-Est real-time operating system for semi-fixed-priority scheduling algorithms. The first practical contribution is to implement the hybrid scheduler for uniprocessor and multiprocessor partitioned scheduling. The second practical contribution is to implement the dual scheduler for multiprocessor global scheduling. The experimental results on uniprocessors show that the RMWP algorithm has comparable overhead to the RM algorithm. The maximum overhead of the M-FWP algorithm has dramatically higher than that of the RMWP algorithm because the M-FWP algorithm calculates the assignable time of the optional part dynamically, which consumes much time. In contrast, the RMWP algorithm does not need to calculate the assignable time of the optional part dynamically, thanks to the optional deadline, which is calculated statically. The jitter of the RMWP algorithm is lowest in all evaluated algorithms. The experimental results on multiprocessors show that the G-RMWP and P-RMWP algorithms have comparable overhead to the G-RM and P-RM algorithms. In addition, the jitters of the G-RMWP and P-RMWP algorithms are lower than or approximately as same as those of the G-RM and P-RM algorithms.

This dissertation concludes that semi-fixed-priority scheduling is the best choice for practical imprecise computation on uniprocessors and multiprocessors. This dissertation believes that many researchers use and refine semi-fixed-priority scheduling. Table 8.1 shows the overview of this dissertation. This dissertation puts all checkmarks.

8.2 Future Directions

This dissertation gives several directions toward the future work as follows.

- A more practical schedulability analysis for semi-fixed-priority scheduling will be desired to fill the gap between ideal simulation results and experimental evaluations. For example, schedulability tests include the overhead of context switches and migrations.
- Distributed real-time systems will be supported by semi-fixed-priority scheduling. The worst case arrival time of each packet is analyzed for remote sensing applications based on imprecise computation.
- Real-time synchronization protocols such as PCP [214] and SRP [215] will be supported in semi-fixed-priority scheduling. In order to manage multiple I/Os in robots, semi-fixed-priority scheduling is required to meet the deadline of each task with shared resources.
- Both the hybrid scheduler and the dual scheduler will be evaluated to verify the scalability on many-core systems such as Intel's Single-Chip Cloud Computer [220]. The

other evaluation approach is to use full system simulation platforms such as Simics [221]. Also, other queueing policies such as binomial heap queue [222] will be compared to both hybrid and dual schedulers.

- Real-time applications will be adapted to the practical imprecise computation model, as described in Chapter 3. Especially, this dissertation will integrate semi-fixed-priority scheduling algorithms to the sensor fusion of visual and auditory information [203].

Bibliography

[1] P. Marwedel. *Embedded System Design*. Springer, 2006.

[2] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of the ACM*, Vol. 20, No. 1, pp. 46–61, 1973.

[3] G. C. Buttazzo. Rate Monotonic vs. EDF: Judgment Day. *Real-Time Systems*, Vol. 29, No. 1, pp. 5–26, 2005.

[4] J. A. Stankovic, K. Ramamritham, and M. Spuri. *Deadline Scheduling for Real-Time Systems: EDF and Related Algorithms*. Kluwer Academic Publishers, 1998.

[5] iRobot Corporation. Roomba. `http://store.irobot.com/home/index.jsp`.

[6] M. K. Gardner and J. W.-S. Liu. Performance of Algorithms for Scheduling Real-Time Systems with Overrun and Overload. In *Proceedings of the 11th Euromicro Conference on Real-Time Systems*, pp. 287–296, June 1999.

[7] G. C. Buttazzo and J. A. Stankovic. RED: Robust Earliest Deadline Scheduling. In *Proceedings of 3rd International Workshop on Responsive Computing Systems*, pp. 100–111, September 1993.

[8] G. C. Buttazzo, M. Spuri, and F. Sensini. Value vs. Deadline Scheduling in Overload Conditions. In *Proceedings of the 16th IEEE Real-time Systems Symposium*, pp. 90–99, December 1995.

[9] S. K. Baruah and J. R. Haritsa. Scheduling for Overload in Real-Time Systems. *IEEE Transactions on Computers*, Vol. 46, No. 9, pp. 1034–1039, 1997.

[10] Intel Corporation. Intel Xeon processor E7 family. `http://www.intel.com/content/www/us/en/processors/xeon/xeon-processor-e7-family.html`.

[11] Sony Computer Entertainment Inc. Cell Broadband Engine. `http://cell.scei.co.jp/`.

[12] ARM Ltd. ARM11 Processor Family. `http://www.arm.com/products/processors/classic/arm11/index.php`.

[13] Apple Inc. iPhone. `http://www.apple.com/iphone/`.

[14] Google Inc. Android. `http://www.android.com/`.

[15] F. Kanehiro, H. Hirukawa, and S. Kajita. OpenHRP: Open Architecture Humanoid Robotics Platform. *The International Journal of Robotics Research*, Vol. 23, No. 2, pp. 155–165, 2004.

[16] Future Robotics Technology Center. HallucII. `http://furo.org/ja/robot/halluc2/index.html`.

[17] Boston Dynamics. BigDog. `http://www.bostondynamics.com/robot_bigdog.html`.

[18] K. Lin, S. Natarajan, and J. Liu. Imprecise Results: Utilizing Partial Computations in Real-Time Systems. In *Proceedings of the 8th IEEE Real-Time Systems Symposium*, pp. 210–217, December 1987.

[19] W. Feng and J. W.-S. Liu. An Extended Imprecise Computation Model for Time-Constrained Speech Processing and Generation. In *Proceedings of the IEEE Workshop on Real-Time Applications*, pp. 76–80, May 1993.

[20] X. Huang and A. M. K. Cheng. Applying Imprecise Algorithms to Real-Time Image and Video Transmission. In *Proceedings of the 1st IEEE Real-Time Technology and Applications Symposium*, pp. 96–101, May 1995.

[21] X. Chen and A. M. K. Cheng. An Imprecise Algorithm for Real-Time Compressed Image and Video Transmission. In *Proceedings of 6th International Conference on Computer Communications and Networks*, pp. 390–397, September 1997.

[22] W. Tan and A. Zakhor. Real-Time Internet Video Using Error Resilient Scalable Compression and TCP-Friendly Transport Protocol. *IEEE Transactions on Multimedia*, Vol. 1, No. 2, pp. 172–186, 1999.

[23] V. Millan-Lopez, W. Feng, and J. W.-S. Liu. Using the Imprecise-Computation Technique for Congestion Control on a Real-Time Traffic Switching Element. In *Proceedings of the International Conference on Parallel and Distributed Systems*, pp. 202–208, December 1994.

[24] W. Feng and J. W.-S. Liu. Performance of a Congestion Control Scheme on an ATM Switch. In *Proceedings of the International Conference on Networks*, pp. 225–228, January 1996.

[25] M. C. Horsch and D. Poole. An Anytime Algorithm for Decision Making under Uncertainty. In *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence*, pp. 246–255, July 1998.

[26] G. B. Parker and J. W. Mills. Adaptive Hexapod Gait Control Using Anytime Learning with Fitness Biasing. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 519–524, July 1999.

[27] G. B. Parker. Punctuated Anytime Learning for Hexapod Gait Generation. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and System*, pp. 2664–2671, October 2002.

[28] K. Fujisawa, S. Hayakawa, T. Aoki, T. Suzuki, and S. Okuma. Real Time Motion Planning for Autonomous Mobile Robot using Framework of Anytime Algorithm. In *Proceedings of the 1999 IEEE International Conference on Robotics and Automation*, pp. 1347–1352, May 1999.

[29] S. Zilberstein and S. J. Russel. Anytime Sensing, Planning and Action: A Practical Model for Robot Control. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, pp. 1402–1407, August 1993.

[30] I. J. Cox, M. L. Miller, R. Danchick, and G. E. Newnam. A Comparison of Two Algorithms for Determining Ranked Assignments with Application to Multi-Target Tracking and Motion Correspondence. *IEEE Transactions on Aerospace and Electronic Systems*, Vol. 33, No. 1, pp. 295–301, 1997.

[31] S. V. Vrbsky and J. W.-S. Liu. APPROXIMATE - A Query Processor that Produces Monotonically Improving Approximate Answers. *IEEE Transactions on Knowledge and Data Engineering*, Vol. 5, No. 6, pp. 1056–1068, 1993.

[32] O. Takács and A. R. Várkonyi-Kóczy. Iterative-type Evaluation of PSGS Fuzzy Systems for Anytime Use. In *Proceedings of IEEE Instrumentation and Measurement Technology Conference*, pp. 233–238, May 2002.

[33] H. Kobayashi. *REAL-TIME SCHEDULING OF PRACTICAL IMPRECISE TASKS UNDER TRANSIENT AND PERSISTENT OVERLOAD*. PhD thesis, Keio University, March 2006.

[34] H. Kobayashi and N. Yamasaki. An Integrated Approach for Implementing Imprecise Computations. *IEICE Transactions on Information and Systems*, Vol. 86, No. 10, pp. 2040–2048, 2003.

[35] H. Kobayashi, N. Yamasaki, and Y. Anzai. Scheduling Imprecise Computations with Wind-up Parts. In *Proceedings of the 18th International Conference on Computers and Their Applications*, pp. 232–235, March 2003.

[36] R. R. Rajkumar. Dealing With Suspending Periodic Tasks. Technical report, IBM Thomas J. Watson Research Center Yorktown Heights, July 1991.

[37] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of Np-Completeness*. W. H. Freeman, 1979.

[38] N. C. Audsley, A. Burns, M. F. Richardson, K. Tindell, and A. J. Wellings. Applying New Scheduling Theory to Static Priority Pre-emptive Scheduling. *Software Engineering Journal*, Vol. 8, No. 5, pp. 284–292, 1993.

[39] T. P. Baker D. Oh. Utilization Bounds for N-Processor Rate Monotone Scheduling with Static Processor Assignment. *Real-Time Systems*, Vol. 15, No. 2, pp. 183–192, 1998.

[40] J. M. López, J. L. Díaz, and D. F. García. Minimum and Maximum Utilization Bounds for Multiprocessor Rate Monotonic Scheduling. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 15, No. 7, pp. 642–653, 2004.

[41] J. M. Lopez, M. Garcia, J. L. Diaz, and D. F. Garcia. Worst-Case Utilization Bound for EDF Scheduling on Real-Time Multiprocessor Systems. In *Proceedings of the 12th Euromicro Conference on Real-Time Systems*, pp. 25–33, June 2000.

[42] B. Andersson and J. Jonsson. The Utilization Bounds of Partitioned and Pfair Static-Priority Scheduling on Multiprocessors are 50%. In *Proceedings of the 15th Euromicro Conference on Real-Time Systems*, pp. 33–40, July 2003.

[43] B. Andersson and J. Jonsson. Fixed-Priority Preemptive Multiprocessor Scheduling: To Partition or not to Partition. In *Proceedings of the Seventh International Conference on Real-Time Computing Systems and Applications*, pp. 337–346, December 2000.

[44] S. K. Baruah, C. G. Plaxton N. K. Cohen, and D. A. Varvel. Proportionate Progress: A Notion of Fairness in Resource Allocation. *Algorithmica*, Vol. 15, No. 6, pp. 600–625, 1996.

[45] J. H. Anderson and A. Srinivasan. Early-Release Fair Scheduling. In *Proceedings of the 12th Euromicro Conference on Real-Time Systems*, pp. 35–43, June 2000.

[46] H. Cho, B. Ravindran, and E. D. Jensen. An Optimal Real-Time Scheduling Algorithm for Multiprocessors. In *Proceedings of the 27th IEEE Real-Time Systems Symposium*, pp. 101–110, December 2006.

[47] K. Funaoka, S. Kato, and N. Yamasaki. Work-Conserving Optimal Real-Time Scheduling on Multiprocessors. In *Proceedings of the 20th Euromicro Conference on Real-Time Systems*, pp. 13–22, July 2008.

[48] G. Nelissen, V. Berten, J. Goossens, and D. Milojevic. Reducing Preemptions and Migrations in Real-Time Multiprocessor Scheduling Algorithms by Releasing the Fairness. In *Proceedings of the 17th International Conference on Embedded and Real-Time Computing Systems and Applications*, pp. 15–24, August 2011.

[49] T. P. Baker. Multiprocessor EDF and Deadline Monotonic Schedulability Analysis. In *Proceedings of the 24th IEEE Real-Time Systems Symposium*, pp. 120–129, December 2003.

[50] J. Goossens, S. Funk, and S. Baruah. Priority-Driven Scheduling of Periodic Task Systems on Multiprocessors. *Real-Time Systems*, Vol. 25, No. 2-3, pp. 187–205, 2003.

[51] S. K. Dhall and C. L. Liu. On a Real-Time Scheduling Problem. *Operations Research*, Vol. 26, No. 1, pp. 127–140, 1978.

[52] B. Andersson, S. K. Baruah, and J. Jonsson. Static-Priority Scheduling on Multiprocessors. In *Proceedings of the 22th IEEE Real-Time Systems Symposium*, pp. 193–202, December 2001.

[53] S. K. Lee. On-line Multiprocessor Scheduling Algorithms for Real-Time Tasks. In *Proceedings the IEEE Region 10's Ninth Annual International Conference*, pp. 607–611, August 1994.

[54] Y.-H. Chao, S.-S. Lin, and K.-J. Lin. Schedulability issues for EDZL scheduling on real-time multiprocessor systems. *Information Processing Letters*, Vol. 107, No. 5, pp. 158–164, August 2008.

[55] S. Kato and N. Yamasaki. Global EDF-based Scheduling with Efficient Priority Promotion. In *Proceedings of the 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pp. 197–206, August 2008.

[56] J. H. Anderson, V. Bud, and U. C. Devi. An EDF-based Scheduling Algorithm for Multiprocessor Soft Real-Time Systems. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems*, pp. 199–208, July 2005.

[57] B. Andersson and E. Tovar. Multiprocessor Scheduling with Few Preemptions. In *Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pp. 322–334, August 2006.

[58] S. Kato and N. Yamasaki. Portioned Static-Priority Scheduling on Multiprocessors. In *Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium*, May 2008.

[59] S. Kato and N. Yamasaki. Portioned EDF-based Scheduling on Multiprocessors. In *Proceedings of the 8th ACM international conference on Embedded Software*, pp. 139–148, October 2008.

[60] S. Kato and N. Yamasaki. Semi-Partitioned Fixed-Priority Scheduling on Multiprocessors. In *Proceedings of the 15th IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 23–32, April 2009.

[61] S. Kato and N. Yamasaki. Semi-Partitioned Scheduling of Sporadic Task Systems on Multiprocessors. In *Proceedings of the 21st Euromicro Conference on Real-Time Systems*, pp. 249–258, July 2009.

[62] K. Lakshmanan, R. R. Rajkumar, and J. P. Lehoczky. Partitioned Fixed-Priority Preemptive Scheduling for Multi-Core Processors. In *Proceedings of the 21st Euromicro Conference on Real-Time Systems*, pp. 239–248, July 2009.

[63] N. Guan, M. Stigge, W. Yi, and G. Yu. Fixed-Priority Multiprocessor Scheduling with Liu and Layland's Utilization Bound. In *Proceedings of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 165–174, April 2010.

[64] G. Levin, S. Funk, C. Sadowski, I. Pye, and S. Brandt. DP-FAIR : A Simple Model for Understanding Optimal Multiprocessor Scheduling. In *Proceedings of the 21st Euromicro Conference on Real-Time Systems*, pp. 3–13, July 2010.

[65] R. McNaughton. Scheduling with Deadlines and Loss Functions. *Management Science*, Vol. 6, No. 1, pp. 1–12, 1959.

[66] P. Regnier, G. Lima, E. Massa, and G. Levinand S. Brandt. RUN: Optimal Multiprocessor Real-Time Scheduling via Reduction to Uniprocessor. In *Proceedings of the 32th IEEE Real-Time Systems Symposium*, pp. 104–115, November 2011.

[67] X. Qi, D. Zhu, and H. Aydin. A Study of Utilization Bound and Run-Time Overhead for Cluster Scheduling in Multiprocessor Real-Time Systems. In *Proceedings of the 16th International Conference on Embedded and Real-Time Computing Systems and Applications*, pp. 3–12, August 2010.

[68] S. K. Baruah and M. E. Hickey. Competitive On-line Scheduling of Imprecise Computations. *IEEE Transactions on Computers*, Vol. 47, No. 9, pp. 1027–1033, 1998.

[69] H. Aydin, R. Melhem, D. Mosse, and P. Mejfa-Alvarez. Optimal Reward-Based Scheduling of Periodic Real-Time Tasks. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*, pp. 79–89, December 1999.

[70] H. Kobayashi and N. Yamasaki. RT-Frontier: A Real-Time Operating System for Practical Imprecise Computation. In *Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 255–264, May 2004.

[71] A. Khemka, R. K. Shyamasundar, and K. V. Subrahmanyam. Multiprocessors Scheduling for Imprecise Computations in a Hard Real-Time Environment. In *Proceedings of the Seventh International Parallel Processing Symposium*, pp. 374–378, April 1993.

[72] K. Yun, K. Song, K. Choi, G. Jung, S. Park, M. Hong, and D. Choi. A Heuristic Scheduling Algorithm of Imprecise Multiprocessor System with 0/1 Constraint. In *Proceedings of the Third International Workshop on Real-Time Computing Systems Application*, pp. 307–313, October 1996.

[73] G. L. Stavrinides and H. D. Karatza. Fault-tolerant Gang Scheduling in Distributed Real-time Systems Utilizing Imprecise Computations. *Simulation*, Vol. 85, No. 8, pp. 525–536, 2009.

[74] G. L. Stavrinides and H. D. Karatza. Scheduling multiple task graphs with end-to-end deadlines in distributed real-time systems utilizing imprecise computations. *Journal of Systems and Software*, Vol. 83, No. 6, pp. 1004–1014, 2010.

[75] The Linux Kernel Organization Inc. Linux. `http://www.kernel.org/`.

[76] V. Yodaiken and M. Barabanov. A Real-Time Linux. In *Proceedings of the Linux Applications Development and Deployment Conference*, January 1997.

[77] Department of Aerospace Engineering of Politecnico di Milano. Real-Time Application Interface. `http://www.rtai.org`.

[78] Xenomai. `http://www.xenomai.org/`.

[79] MontaVista Software Inc. MontaVista Linux. `http://www.mvista.com/product_detail_mvl6.php`.

[80] I. Molnar. CONFIG PREEMPT RT Patch. `https://rt.wiki.kernel.org/articles/c/o/n/CONFIG_PREEMPT_RT_Patch_79df.html`.

[81] S. Oikawa and R. R. Rajkumar. Portable RK: A Portable Resource Kernel for Guaranteed and Enforced Timing Behavior. In *Proceedings of the Fifth IEEE Real-Time Technology and Applications Symposium*, pp. 111–120, June 1999.

[82] B. Srinivasan, S. Pather, R. Hill, F. Ansari, and D. Niehaus. A Firm Real-Time System Implementation Using Commercial Off-The-Shelf Hardware and Free Software. In *Proceedings of the Fourth IEEE Real-Time Technology and Applications Symposium*, pp. 112–119, June 1998.

[83] Y. Ishiwata and T. Matsui. A Real-Time Operating System that can Share Device Drivers with General Purpose OS. *IEICE Technical Report*, pp. 41–48, 1998.

[84] Y.-C. Wang and K.-J. Lin. Implementing a general real-time scheduling framework in the RED-Linux real-time kernel. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*, pp. 246–255, December 1999.

[85] S. Childs and D. Ingram. The Linux-SRT integrated multimedia operating system: bringing QoS to the desktop. In *Proceedings of the Seventh IEEE Real-Time Technology and Applications Symposium*, pp. 135–140, May 2001.

[86] J. M. Calandrino, H. Leontyev, A. Block, U. C. Devi, and J. H. Anderson. LITMUS^{RT}: A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers. In *Proceedings of the 27th IEEE Real-Time Systems Symposium*, pp. 111–123, December 2006.

[87] T. Yang, T. Liu, E. D. Berger, S. F. Kaplan, J. Eliot, and B. Moss. Redline: First Class Support for Interactivity in Commodity Operating Systems. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, pp. 73–86, December 2008.

[88] D. Faggioli, M. Trimarchi, and F. Checconi. An Implementation of the Earliest Deadline First Algorithm in Linux. In *Proceedings of the 24th Annual ACM Symposium on Applied Computing*, pp. 1984–1989, March 2009.

[89] D. Faggioli, M. Trimarchi, F. Checconi, and C. Scordino. An EDF Scheduling Class for the Linux Kernel. In *Proceedings of the Real-Time Linux Workshop*, pp. 16–23, September 2009.

[90] L. Palopoli, T. Cucinotta, L. Marzario, and G. Lipari. AQuoSA-adaptive quality of service architecture. *Software-Practice and Experience*, Vol. 39, No. 1, pp. 1–31, 2009.

[91] S. Kato, R. R. Rajkumar, and Y. Ishikawa. AIRS: Supporting Interactive Real-Time Applications on Multicore Platforms. In *Proceedings of the 22nd Euromicro Conference on Real-Time Systems*, pp. 47–56, July 2010.

[92] M. Dellinger, P. Garyali, and B. Ravindran. ChronOS Linux: A Best-Effort Real-Time Multiprocessor Linux Kernel. In *Proceedings of the ACM Design and Automation Conference*, pp. 474–479, July 2011.

[93] H. Tokuda, T. Nakajima, and P. Rao. Real-Time Mach: Towards a Predictable Real-Time System. In *Proceedings of USENIX Mach Workshop*, pp. 73–82, October 1990.

[94] R. Rashid, D. Julin, D. Orr, R. Sanzi, R. Baron, A. Forin, D. Golub, and M. Jones. Mach: A System Software Kernel. In *Proceedings of the 1989 IEEE International Conference, COMPCON*, pp. 176–178, February 1989.

[95] T. Nakajima and S. Oikawa. Experiences with Building Real-Time Mach Operating Systems. *Computer Software*, Vol. 23, No. 1, pp. 24–44, 2006 (in Japanese).

[96] Microsoft Corporation. Windows CE. http://www.microsoft.com/windowsembedded/en-us/windows-embedded.aspx.

[97] Microsoft Corporation. Windows. http://windows.microsoft.com/en-US/windows/home.

[98] B. Furht, D. Grostick, D. Gluch, G. Rabbat, J. Parker, and M. McRoberts. *REAL-TIME UNIX SYSTEMS: Design and Applications Guide*. Springer, 1st edition, 1990.

[99] The Open Group. UNIX. http://www.unix.org/.

[100] POSIX. *1003.13-2003 IEEE Standard for Information Technology - Standardized Application Environment Profile (AEP) - POSIX Realtime and Embedded Application Support*.

[101] POSIX. *IEEE Std 1003.1, 2004 Edition*. http://www.unix.org/version3/ieee_std.html.

[102] HeartOS. http://www.ddci.com/products_heartos.php.

[103] P. Gai, L. Abeni, M. Giorgi, and G. C. Buttazzo. A New Kernel Approach for Modular Real-Time Systems Development. In *Proceedings of the Euromicro Conference on Real-Time Systems*, pp. 199–206, June 2001.

[104] University of Cantabria. MaRTE. http://marte.unican.es/.

[105] H. Takada. *μITRON4.0 Specification*. http://www.ertl.jp/ITRON/SPEC/FILE/mitron-400e.pdf, 1999.

[106] TOPPERS Project. TOPPERS/JSP Kernel. http://www.toppers.jp/en/jsp-kernel.html.

[107] Hyper Operating System. http://sourceforge.jp/projects/hos/.

[108] T-Engine Forum Japan. T-Kernel. http://www.t-engine.org/what-is-t-kernel/t-kernel.

[109] Mentor Graphics Inc. Nucleus. http://www.mentor.com/embedded-software/nucleus/.

[110] Y. Tiomkin. TNKernel. http://www.tnkernel.com/.

[111] OSEK VDX Portal. http://www.osek-vdx.org/.

[112] TOPPERS Project. TOPPERS/ATK Kernel. http://www.toppers.jp/atk1.html.

[113] Evidence Srl and the Real-Time Systems Laboratory. Erika Enterprise. http://erika.tuxfamily.org/.

[114] FreeOSEK. http://opensek.sourceforge.net/.

[115] PICOS18. http://www.picos18.com/index_us.htm.

[116] Real-Time Systems Group in Nantes Communications and Cybernetics Research Institute. Trampoline. http://trampoline.rts-software.org/.

[117] *AUTomotive Open System ARchitecture*. http://www.autosar.org/.

[118] *AVIONICS APPLICATION SOFTWARE STANDARD INTERFACE*. http://www.arinc.com/.

[119] Green Hills Software Inc. INTEGRITY-178B. http://www.ghs.com/products/safety_critical/integrity-do-178b.html.

[120] eCos. http://ecos.sourceware.org/.

[121] OAR Corporation. RTEMS Operating System | Real-Time and Real Free. http://www.rtems.com/.

[122] LynxWorks Inc. LynxRTOS. http://www.lynuxworks.com/rtos/.

[123] E. D. Jensen and J. D. Northcutt. Alpha: a nonproprietary OS for large, complex, distributed real-time systems. In *Proceedings of the IEEE Workshop on Experimental Distributed Systems*, pp. 35–41, October 1990.

[124] S. Hong, Y. Seo, and J. Park. ARX/ULTRA: A New Real-Time Kernel Architecture for Supporting User-Level Threads. Technical Report SNU-EE-TR1997-3, School of Electrical Engineering, Seoul National University, 2000.

[125] K. Schwan, P. Gopinath, and W. Bo. CHAOS-Kernel Support for Objects in the Real-Time Domain. *IEEE Transactions on Computers*, Vol. 36, No. 8, 1987.

[126] D. B. Stewart, D. E. Schmitz, and P. K. Khosla. Implementing Real-Time Robotic Systems using CHIMERA II. In *Proceedings of 1990 IEEE International Conference on Robotics and Automation, Cincinnatti*, pp. 598–603, May 1990.

[127] Contiki. http://www.contiki-os.org/.

[128] K. Zuberi and K. G. Shin. EMERALDS: A Microkernel for Embedded Real-Time Systems. In *Proceedings of the 2nd IEEE Real-Time Technology and Applications Symposium*, pp. 241–249, June 1996.

[129] D. Langan. EOS: an object-oriented operating system for embedded real-time applications. In *Proceedings of the 1993 ACM conference on Computer science*, pp. 60–65, February 1993.

[130] J. S. Shapiro and J. Adams. Design Evolution of the EROS Single-Level Store. In *Proceedings of the 2002 USENIX Annual Technical Conference*, pp. 59–72, June 2002.

[131] Strawberry Development Group. CapROS. http://www.capros.org/.

[132] Dresden University of Technology. Fiasco. http://os.inf.tu-dresden.de/fiasco/.

[133] G. C. Buttazzo. HARTIK: A real-time kernel for robotics applications. In *Proceedings of the 14th IEEE Real-Time Systems Symposium*, pp. 201–205, December 1993.

[134] D. D. Kandlur, D. L. Kiskis, and K. G. Shin. HARTOS: a distributed real-time operating system. *ACM SIGOPS Operating Systems Review*, Vol. 23, No. 3, pp. 72–89, 1989.

[135] D. Andrews, W. Peck, J. Agron, K. Preston, E. Komp, M. Finley, and R. Sass. hthreads: A Hardware/Software Co-Designed Multithreaded RTOS Kernel. In *Proceedings of the 10th IEEE International Conference on Emerging Technologies and Factory Automation*, pp. 331–338, September 2005.

[136] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, and R. Zainlinger. Distributed Fault-Tolerant Real-Time Systems: The Mars Approach. *IEEE Micro*, Vol. 9, No. 1, pp. 25–40, 1989.

[137] S. Levi, S. K. Tripathi, S. D. Carson, and A. K. Agrawala. The MARUTI hard real-time operating system. *ACM SIGOPS Operating Systems Review*, Vol. 23, No. 3, pp. 90–105, 1989.

[138] D. L. Bayer and H. Lycklama. MERT - a multi-environment real-time operating system. In *Proceedings of the fifth ACM symposium on Operating systems principles*, pp. 33–42, November 1975.

[139] A. Eswaran, A. Rowe, and R. R. Rajkumar. Nano-RK: an Energy-aware Resource-centric RTOS for Sensor Networks. In *Proceedings of the 26th IEEE International Real-Time Systems Symposium*, pp. 256–265, December 2005.

[140] M. Danish, Y. Li, and R. West. Virtual-CPU Scheduling in the Quest Operating System. In *Proceedings of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 169–179, April 2011.

[141] D. R. Donari, L. Ordinez, R. Santos, and J. Orozco. Real-Time Server Oriented Operating System for Embedded Applications. `http://www.ingelec.uns.edu.ar/rts/soos/soos_2008.pdf`.

[142] J. Stankovic and K. Ramamritham. The Spring kernel: A new paradigm for real-time systems. *IEEE Software*, Vol. 8, No. 3, 1991.

[143] I. Lee and R. King. Timix: a distributed real-time kernel for multi-sensor robots. In *Proceedings of the 1988 IEEE International Conference on Robotics and Automation*, pp. 1587–1589, April 1988.

[144] K. Jeffay, D. L. Stone, and D. E. Poirier. YARTOS Kernel support for efficient, predictable real-time systems. *Real-Time Systems Newsletter*, Vol. 7, No. 4, pp. 8–13, 1991.

[145] Code Time Technologies Inc. Abassi. `http://www.code-time.com/products.html`.

[146] KADAK Products Ltd. AMX. `http://www.kadak.com/rtos/rtos.htm`.

[147] AVIX-RT. AVIX. http://www.avix-rt.com/.

[148] Oracle Corporation. ChorusOS. http://www.oracle.com/technetwork/documentation/legacy-op-sys-193044.html#os.

[149] CMX Systems Inc. CMX. http://www.cmx.com/rtos.htm.

[150] ELESOFTROM company. DioneOS. http://www.elesoftrom.com.pl/en/os/.

[151] SEGGER Microcontroller Systems. embOS. http://www.segger.com/embos.html.

[152] Fusion RTOS. http://www.unicoi.com/product_suite_pages/fusion_rtos_product_suite.htm.

[153] TenAsys Corporation. iRMX. http://www.tenasys.com/products/irmx.php.

[154] J. J. Labrosse. *MicroC/OS-II: The Real Time Kernel.* Newnes, 2nd edition, 2002.

[155] Freescale Semiconductor Inc. MQX. http://www.freescale.com/webapp/sps/site/homepage.jsp?code=MQX_HOME.

[156] Enea. OSE. http://www.enea.com/software/products/rtos/ose/.

[157] SYSGO Inc. PikeOS. http://www.sysgo.com/products/pikeos-rtos-and-virtualization-concept/.

[158] Softwave Wireless. Portos. http://www.portos.org/.

[159] QuasarSoft Ltd. Q-Kernel. http://www.quasarsoft.com/products.html.

[160] D. Hildebrand. An Architectural Overview of QNX. In *Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures*, pp. 113–126, April 1992.

[161] Quantum Leaps Inc. QP. http://www.state-machine.com/qp/.

[162] OBP Research Oy. ReaGOS. http://www.obp.fi/2009/products/reagos/.

[163] rt-labs Aktiebolag. rt-kernel. http://www.rt-labs.com/rt-kernel_overview.shtml.

[164] IntervalZero Inc. RTX. http://www.intervalzero.com/.

[165] Quadros Systems Inc. RTXC Quadros. http://www.quadros.com/products/operating-systems.

[166] Pumpkin Inc. Salvo. http://www.pumpkininc.com/.

[167] SCIOPTA Systems AG. SCIOPTA. http://www.sciopta.com/.

[168] SpaceShadow Company. Sirius. http://www.spaceshadow.com/products.php.

[169] Micro Digital Inc. SMX. http://www.smxrtos.com/.

[170] EWA Technologies Inc. Talon. http://www.blackhawk-dsp.com/products/Talon.aspx.

[171] Blunk Microsystems Company. TargetOS. http://www.blunkmicro.com/os.htm.

[172] Express Logic Inc. ThreadX. http://rtos.com/products/threadx/.

[173] M. J. Butcher Consulting. μTasker. http://www.utasker.com/.

[174] Green Hills Software. μ-velOSity. http://www.ghs.com/products/micro_velosity.html.

[175] J. Fiddler, Eric Stromberg, and D. N. Wilner. Software considerations for real-time RISC. In *Compcon Spring 90 Digest of Papers: Thirty-Fifth IEEE Computer Society International Conference*, pp. 274–277, February 1990.

[176] Atomthreads. http://atomthreads.com/.

[177] BeRTOS. http://www.bertos.org/.

[178] BRTOS. http://code.google.com/p/brtos/.

[179] ChibiOS/RT. http://www.chibios.org/.

[180] cocoOS. http://www.cocoos.net/.

[181] Hardware Engineering Department of Lanit-Tercom Inc. and Saint-Petersburg State University. Embox. http://code.google.com/p/embox/.

[182] Femto OS. http://www.femtoos.org/.

[183] FreeRTOS. http://www.freertos.org/.

[184] FunkOS. http://funkos.sourceforge.net/.

[185] Helium. http://helium.sourceforge.net/.

[186] iRTOS. http://irtos.sourceforge.net/.

[187] Milos. Milos RTOS. http://www.milos.it/.

[188] OSA. http://wiki.pic24.ru/doku.php/en/osa/ref/intro.

[189] D. Kuschel and S. Moczarski. picoOS. http://picoos.sourceforge.net/.

[190] Phoenix. http://www.phoenix-rtos.org/.

[191] K. Ohtani. Prex. http://prex.sourceforge.net/.

[192] RT-Thread. http://code.google.com/p/rt-thread/.

[193] scmRTOS Team. scmRTOS. http://scmrtos.sourceforge.net/.

[194] SDPOS Team. SDPOS. http://www.sdpos.org/.

[195] uOS. http://code.google.com/p/uos-embedded/wiki/about.

[196] uSmartx. http://usmartx.sourceforge.net/.

[197] C. Marlin, W. Zhao, G. Doherty, and A. Bohonis. GARTL: A Real-time Programming Language Based on Multi-version Computation. In *Proceedings of International Conference on Computer Languages*, pp. 107–115, March 1990.

[198] D. Hull, W. Feng, and J. W.-S. Liu. Enhancing the Performance and Dependability of Real-Time Systems. In *Proceedings of the IEEE International Computer Performance and Dependability Symposium*, pp. 174–182, April 1995.

[199] B. B. Brandenburg and J. H. Anderson. On the Implementation of Global Real-Time Schedulers. In *Proceedings of the 30th IEEE Real-Time Systems Symposium*, pp. 214–224, December 2009.

[200] A. Bastoni, B. B. Brandenburg, and J. H. Anderson. An Empirical Comparison of Global, Partitioned, and Clustered Multiprocessor Real-Time Schedulers. In *Proceedings of the 31th IEEE Real-Time Systems Symposium*, pp. 14–24, December 2010.

[201] A. Bastoni, B. B. Brandenburg, and J. H. Anderson. Is Semi-Partitioned Scheduling Practical? In *Proceedings of the 23rd Euromicro Conference on Real-Time Systems*, pp. 125–135, July 2011.

[202] J. Lelli, G. Lipari, D. Faggioli, and T. Cucinotta. An efficient and scalable implementation of global EDF in Linux. In *Proceedings of the 7th Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pp. 6–15, July 2011.

[203] K. Takahashi. Sensing System Integrating Audio and Visual Information : Concrete Examples of Sensor Fusion Systems. *Journal of the Institute of Electronics, Information, and Communication Engineers*, Vol. 79, No. 2, pp. 155–161, 1996 (in Japanese).

[204] P. Kritsada and K. O.-Yang. Sensor Fusion by Neural Network and Wavelet Analysis for Drill-Wear Monitoring. *Journal of Solid Mechanics and Materials Engineering*, Vol. 4, No. 6, pp. 749–760, 2010.

[205] A. Shintani, A. Ogihara, Y. Yamaguchi, Y. Hayashi, and K. Fukunaga. Speech Recognition Using HMM Based on Fusion of Visual and Auditory Information. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, Vol. 77, No. 11, pp. 1875–1878, 1994.

[206] G. C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Springer, 3rd edition, 2010.

[207] M. Bertogna and M. Cirinei. Response-Time Analysis for Globally Scheduled Symmetric Multiprocessor Platforms. In *Proceedings of the 28th IEEE Real-Time Systems Symposium*, pp. 149–158, December 2007.

[208] N. Guan, M. Stigge, W. Yi, and G. Yu. New Response Time Bounds for Fixed Priority Multiprocessor Scheduling. In *Proceedings of the 30th IEEE Real-Time Systems Symposium*, pp. 387–397, December 2009.

[209] S. K. Baruah. Techniques for Multiprocessor Global Schedulability Analysis. In *Proceedings of the 28th IEEE Real-Time Systems Symposium*, pp. 119–128, December 2007.

[210] T. P. Baker. An Analysis of Fixed-Priority Schedulability on a Multiprocessor. *Real-Time Systems*, Vol. 32, No. 1-2, pp. 49–71, 2006.

[211] Y. Oh and S. H. Son. Tight Performance Bounds of Heuristics for a Real-Time Scheduling Problem. Technical Report CS-93-24, Department of Computer Science, University Of Virginia, 1993.

[212] Y. Oh and S. H. Son. Fixed Priority Scheduling of Periodic Tasks on Multiprocessor Systems. Technical Report CS-95-16, Department of Computer Science, University Of Virginia, 1995.

[213] A. Srinivasan and S. K. Baruah. Deadline-based Scheduling of Peroidic Task Systems on Multiprocessors. *Information Processing Letters*, Vol. 84, No. 2, pp. 93–98, October 2002.

[214] L. Sha, R. R. Rajkumar, and J. P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Transactions on Computers*, pp. 1175–1185, 1990.

[215] T. P. Baker. Stack-Based Scheduling of Realtime Processes. *The Journal of Real-Time Systems*, pp. 67–99, 1991.

[216] J. Dike. *User Mode Linux*. Prentice Hall, 2006.

[217] L. Palopoli, G. Lipari, G. Lamastra, L. Abeni, G. Bolognini, and P. Ancilotti. An object oriented tool for simulating distributed real-time control systems. *Software - Practice and Experience*, Vol. 32, No. 9, pp. 907–932, 2002.

[218] J. W.-S. Liu. *Real-Time Systems*. Prentice Hall, 1st edition, 2000.

[219] M. Matsumoto and T. Nishimura. Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator. *ACM Transactions on Modeling and Computer Simulations*, Vol. 8, No. 1, pp. 3–30, 1998.

[220] Intel Corporation. SCC External Architecture Specification (EAS), July 2010.

[221] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A Full System Simulation Platform. *Computer*, Vol. 35, No. 2, pp. 50–58, 2002.

[222] J. Vuillemin. A Data Structure for Manipulating Priority Queues. *Communications of the ACM*, Vol. 21, pp. 309–315, April 1978.

Appendix A

Schedulability Analysis of the RMWP Algorithm for Two Tasks

This appendix analyzes the least upper bound of the RMWP algorithm for two tasks on uniprocessors in Subsection 4.2.2.

Theorem 10 (Least upper bound of the RMWP algorithm for two tasks)**.** *For a set of two tasks with semi-fixed-priority assignment, the least upper bound of the RMWP algorithm on uniprocessors is* $U_{lub} = 2(2^{1/2} - 1)$.

Proof. As before $F = \lfloor T_2/T_1 \rfloor$ be the number of periods of task τ_1 entirely contained in T_2. Without loss of generality, the computation time $\sum_{L=1}^{n_2^m} m_2^L$ is adjusted to fully utilize the processors.

Case 1: As shown in Figure A.1, when job $\tau_{2,2}$ is released, task τ_1 executes the l^{th} mandatory part and there is no idle processor time between mandatory parts of task τ_1. In this case, the equation $T_2 - T_1 F \leqq \sum_{L=1}^{n_1^m} m_1^L$ is met so that the maximum of all mandatory parts of task τ_2 is

$$\sum_{L=1}^{n_2^m} m_2^L = (T_1 - \sum_{L=1}^{n_1^m} m_1^L)F. \tag{A.1}$$

The corresponding upper bound U_{ub} is

$$\begin{aligned} U_{ub} &= \frac{\sum_{L=1}^{n_1^m} m_1^L}{T_1} + \frac{\sum_{L=1}^{n_2^m} m_2^L}{T_2} = \frac{\sum_{L=1}^{n_1^m} m_1^L}{T_1} + \frac{(T_1 - \sum_{L=1}^{n_1^m} m_1^L)F}{T_2} \\ &= \frac{T_1}{T_2}F + \frac{\sum_{L=1}^{n_1^m} m_1^L}{T_1} - \frac{\sum_{L=1}^{n_1^m} m_1^L}{T_2}F = \frac{T_1}{T_2}F + \frac{\sum_{L=1}^{n_1^m} m_1^L}{T_2}\left[\frac{T_2}{T_1} - F\right]. \end{aligned} \tag{A.2}$$

Since the quantity in square brackets is positive, the corresponding upper bound U_{ub} is monotonically increasing in $\sum_{L=1}^{n_1^m} m_1^L$. Being $T_2 - T_1 F \leqq \sum_{L=1}^{n_1^m} m_1^L$, the minimum of the corresponding upper bound U_{ub} occurs for

$$\sum_{L=1}^{n_1^m} m_1^L = T_2 - T_1 F. \tag{A.3}$$

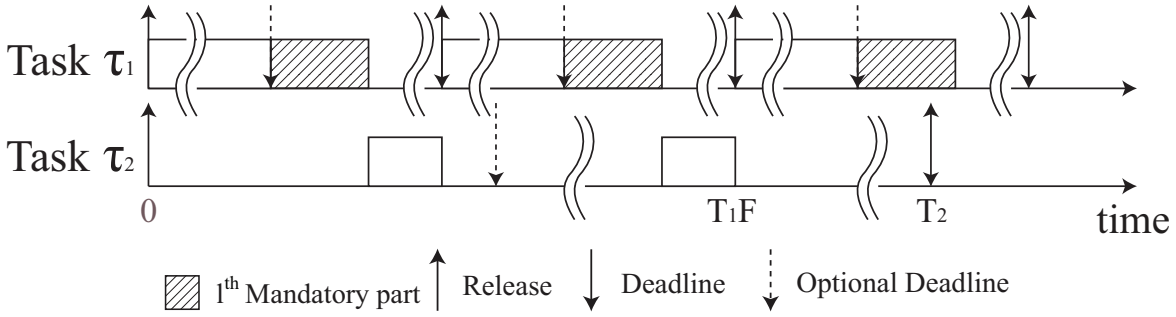


Figure A.1: Case 1

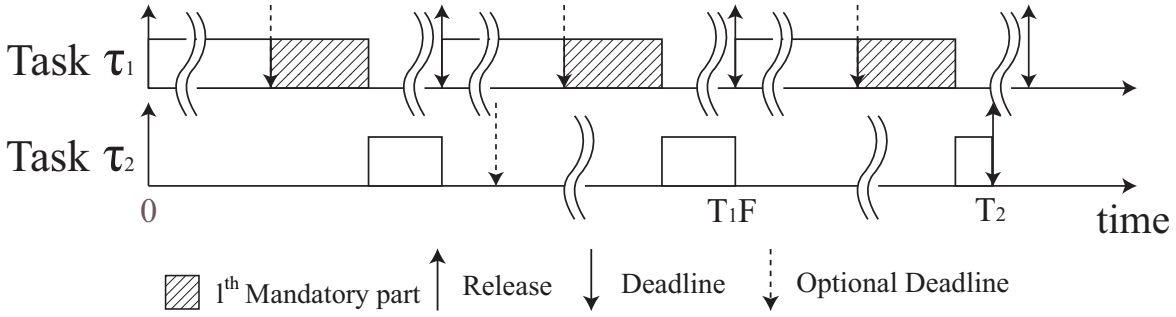


Figure A.2: Case 2

Case 2: As shown in Figure A.2, when job $\tau_{2,2}$ is released, task τ_1 does not complete the l^{th} mandatory part by the l^{th} optional deadline and there is no idle processor time between mandatory parts of task τ_1. In this case, the equation $\sum_{L=1}^{n_1^m} m_1^L \leqq T_2 - T_1F$ is met so that the maximum of all mandatory parts of task τ_2 is

$$\sum_{L=1}^{n_2^m} m_2^L = T_2 - (F+1)\sum_{L=1}^{n_1^m} m_1^L. \tag{A.4}$$

The corresponding upper bound U_{ub} is

$$\begin{aligned} U_{ub} &= \frac{\sum_{L=1}^{n_1^m} m_1^L}{T_1} + \frac{\sum_{L=1}^{n_2^m} m_2^L}{T_2} = \frac{\sum_{L=1}^{n_1^m} m_1^L}{T_1} + \frac{T_2 - (F+1)\sum_{L=1}^{n_1^m} m_1^L}{T_2} \\ &= 1 + \frac{\sum_{L=1}^{n_1^m} m_1^L}{T_1} - \frac{(F+1)\sum_{L=1}^{n_1^m} m_1^L}{T_2} = 1 + \frac{\sum_{L=1}^{n_1^m} m_1^L}{T_2}\left[\frac{T_2}{T_1} - (F+1)\right]. \end{aligned} \tag{A.5}$$

Since the quantity in square brackets is negative, the corresponding upper bound U_{ub} is monotonically increasing in $\sum_{L=1}^{n_1^m} m_1^L$. Being $\sum_{L=1}^{n_1^m} m_1^L \leqq T_2 - T_1F$, the minimum of the corresponding upper bound U_{ub} occurs for

$$\sum_{L=1}^{n_1^m} m_1^L = T_2 - T_1F. \tag{A.6}$$

Case 3: As shown in Figure A.3, when job $\tau_{2,2}$ is released, task τ_1 executes the l^{th} mandatory part and there is processor time between mandatory parts of task τ_1. In this case,

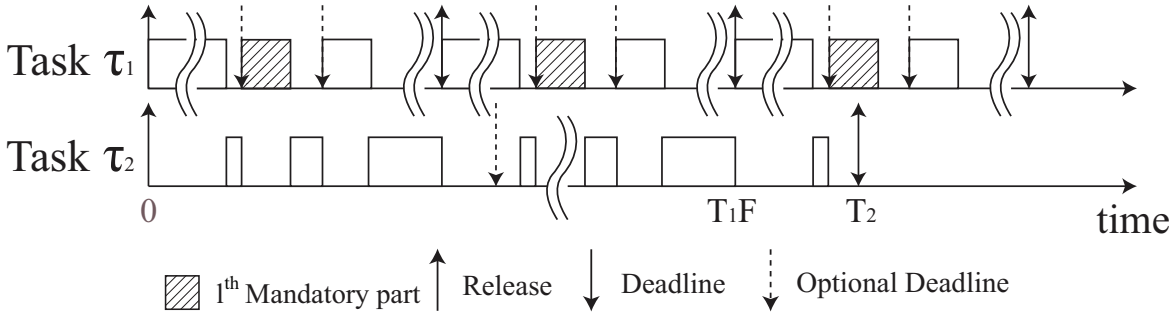


Figure A.3: Case 3

the equations $OD_1^l \leqq T_2 - T_1F \leqq OD_1^l + m_1^l$ and $0 \leqq \sum_{L=l+1}^{n_i^m} m_1^L \leqq T_1 - OD_1^{l+1}$ are met so that the maximum of all mandatory parts of task τ_2 is

$$\sum_{L=1}^{n_2^m} m_2^L = (T_1 - \sum_{L=1}^{n_1^m} m_1^L)F + OD_1^l - \sum_{L=1}^{l-1} m_1^L. \tag{A.7}$$

The corresponding upper bound U_{ub} is

$$\begin{aligned} U_{ub} &= \frac{\sum_{L=1}^{n_1^m} m_1^L}{T_1} + \frac{\sum_{L=1}^{n_2^m} m_2^L}{T_2} = \frac{\sum_{L=1}^{n_1^m} m_1^L}{T_1} + \frac{(T_1 - \sum_{L=1}^{n_1^m} m_1^L)F + OD_1^l - \sum_{L=1}^{l-1} m_1^L}{T_2} \\ &= \frac{T_1}{T_2}F + \frac{\sum_{L=1}^{n_1^m} m_1^L}{T_2}\left[\frac{T_2}{T_1} - F\right] + \frac{OD_1^l - \sum_{L=1}^{l-1} m_1^L}{T_2} \\ &= \frac{T_1}{T_2}F + \frac{\sum_{L=1}^{n_1^m} m_1^L}{T_2}\left[\frac{T_2}{T_1} - F\right] + \frac{OD_1^l + m_1^l - \sum_{L=1}^{l} m_1^L}{T_2}. \end{aligned} \tag{A.8}$$

Since the quantity in square brackets is positive, the corresponding upper bound U_{ub} is monotonically increasing in $\sum_{L=1}^{n_1^m} m_1^L$. In addition, $OD_1^l + m_1^l - \sum_{L=1}^{l} m_1^L$ is positive so that the minimum of that equation occurs if $OD_1^l + m_1^l - \sum_{L=1}^{l} m_1^L = 0$. That is to say, $OD_1^l + m_1^l = \sum_{L=1}^{l} m_1^L$. Being $OD_1^l \leqq T_2 - T_1F \leqq OD_1^l + m_1^l$ and $0 \leqq \sum_{L=l+1}^{n_i^m} m_1^L \leqq T_1 - OD_1^{l+1}$, the minimum of the corresponding upper bound U_{ub} occurs for

$$\sum_{L=1}^{n_1^m} m_1^L = \sum_{L=1}^{l} m_1^L + \sum_{L=l+1}^{n_i^m} m_1^L = (T_2 - T_1F) + 0 = T_2 - T_1F. \tag{A.9}$$

Case 4: As shown in Figure A.4, when job $\tau_{2,2}$ is released, task τ_1 completes the l^{th} mandatory part by the l^{th} optional deadline and does not start to execute the $l+1^{th}$ mandatory part. In addition, there is processor time between mandatory parts of task τ_1. In this case, the equations $0 \leqq \sum_{L=1}^{l} m_1^L \leqq T_2 - T_1F \leqq OD_1^l$ and $0 \leqq \sum_{L=l+1}^{n_i^m} m_1^L \leqq T_1 - OD_1^{l+1}$ are met so that the maximum of all mandatory parts of task τ_2 is

$$\sum_{L=1}^{n_2^m} m_2^L = T_2 - [(F+1)\sum_{L=1}^{l} m_1^L + F\sum_{L=l+1}^{n_i^m} m_1^L]. \tag{A.10}$$

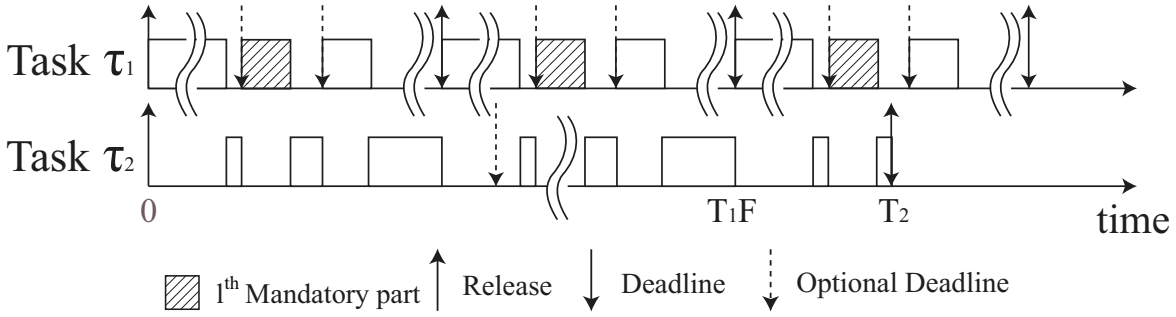


Figure A.4: Case 4

The corresponding upper bound U_{ub} is

$$\begin{aligned} U_{ub} &= \frac{\sum_{L=1}^{n_1^m} m_1^L}{T_1} + \frac{\sum_{L=1}^{n_2^m} m_2^L}{T_2} = \frac{\sum_{L=1}^{n_1^m} m_1^L}{T_1} + \frac{T_2 - [(F+1)\sum_{L=1}^{l} m_1^L + F\sum_{L=l+1}^{n_i^m} m_1^L]}{T_2} \\ &= \left(\frac{1}{T_1} - \frac{F}{T_2}\right)\sum_{L=l+1}^{n_i^m} m_1^L + 1 + \frac{\sum_{L=1}^{l} m_1^L}{T_1} - \frac{(F+1)\sum_{L=1}^{l} m_1^L}{T_2}. \end{aligned} \tag{A.11}$$

After that, the corresponding upper bound U_{ub} is partially differentiated by $\sum_{L=l+1}^{n_i^m} m_1^L$.

$$\frac{\partial U_{ub}}{\partial \sum_{L=l+1}^{n_i^m} m_1^L} = \frac{1}{T_1} - \frac{F}{T_2} = \frac{1}{T_2}\left[\frac{T_2}{T_1} - F\right] \tag{A.12}$$

Since the quantity in square brackets is positive, the corresponding upper bound U_{ub} is monotonically increasing. In this case, the minimum of the corresponding upper bound U_{ub} is as the following equation if $\sum_{L=l+1}^{n_i^m} m_1^L = 0$.

$$U_{ub} = 1 + \frac{\sum_{L=1}^{l} m_1^L}{T_1} - \frac{(F+1)\sum_{L=1}^{l} m_1^L}{T_2} = 1 + \frac{\sum_{L=1}^{l} m_1^L}{T_2}\left[\frac{T_2}{T_1} - (F+1)\right] \tag{A.13}$$

Since the quantity in square brackets is negative, the corresponding upper bound U_{ub} is monotonically decreasing in $\sum_{L=1}^{l} m_1^L$. Being $0 \leqq \sum_{L=1}^{l} m_1^L \leqq T_2 - T_1F \leqq OD_1^l$ and $0 \leqq \sum_{L=l+1}^{n_i^m} m_1^L \leqq T_1 - OD_1^{l+1}$, the minimum of the corresponding upper bound U_{ub} occurs for

$$\sum_{L=1}^{n_1^m} m_1^L = \sum_{L=1}^{l} m_1^L + \sum_{L=l+1}^{n_i^m} m_1^L = (T_2 - T_1F) + 0 = T_2 - T_1F. \tag{A.14}$$

The upper bound of the corresponding upper bound U_{ub} occurs if $\sum_{L=1}^{n_1^m} m_1^L = T_2 - T_1F$ in all cases. This value is equal to that of Theorem 3 in [2]. Hence, the least upper bound of the RMWP algorithm for two tasks on uniprocessors is

$$U_{lub} = 2(2^{1/2} - 1). \tag{A.15}$$

□

List of Papers

Articles on Periodicals

- Hiroyuki Chishiro, Akira Takeda, Kenji Funaoka and Nobuyuki Yamasaki. Real-time Scheduling Based on Rate Monotonic for Extended Imprecise Tasks. *IPSJ Journal*, Vol. 52, No. 8, pp. 2365–2377, August 2011 (in Japanese).

- Hiroyuki Chishiro and Nobuyuki Yamasaki. RT-Est: Real-time Operating System for Semi-fixed-priority Scheduling. *IPSJ Transactions on Advanced Computing Systems*, Vol. 4, No.1, pp. 53–68, February 2011 (in Japanese).

- Hiroyuki Chishiro and Nobuyuki Yamasaki. Semi-fixed-priority Scheduling on Prioritized SMT Processor. *IPSJ Journal*, Vol. 51, No. 12, pp. 2227–2237, December 2010 (in Japanese).

Articles on International Conference Proceedings

- Hiroyuki Chishiro and Nobuyuki Yamasaki. Experimental Evaluation of Global and Partitioned Semi-Fixed-Priority Scheduling Algorithms on Multicore Systems. In *Proceedings of the 15th IEEE International Symposium on Object / Component / Service-Oriented Real-Time Distributed Computing*, April 2012. (To appear)

- Hiroyuki Chishiro and Nobuyuki Yamasaki. RT-Est: Real-Time Operating System for Semi-Fixed-Priority Scheduling Algorithms. In *Proceedings of the 2011 International Symposium on Embedded and Pervasive Systems*, pp. 358–365, October 2011.

- Hiroyuki Chishiro and Nobuyuki Yamasaki. Global Semi-fixed-priority Scheduling on Multiprocessors. In *Proceedings of the 17th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pp. 218–223, August 2011.

- Hiroyuki Chishiro and Nobuyuki Yamasaki. Performance Evaluation of Semi-Fixed-Priority Scheduling on Prioritized SMT Processors. In *Proceedings of the 10th IASTED International Conference on Parallel and Distributed Computing and Networks*, pp. 75–82, February 2011.

- Hiroyuki Chishiro, Akira Takeda, Kenji Funaoka and Nobuyuki Yamasaki. Semi-Fixed-Priority Scheduling: New Priority Assignment Policy for Practical Imprecise Computation. In *Proceedings of the 16th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pp. 339–348, August 2010.

- Hiroyuki Chishiro, Yuji Fujita, Akira Takeda, Yuta Kojima, Kenji Funaoka, Shinpei Kato and Nobuyuki Yamasaki. Extended RT-Component Framework for RT-Middleware. In *Proceedings of the 12th IEEE International Symposium on Object / Component / Service-Oriented Real-Time Distributed Computing*, pp. 161–168, March 2009.