REAL-TIME SCHEDULING OF PRACTICAL IMPRECISE TASKS UNDER TRANSIENT AND PERSISTENT OVERLOAD

HIDENORI KOBAYASHI

March 2006

ACKNOWLEDGMENT

During my stay here at Keio, I have been fortunate to receive every kind of support and assistance from many people in all endeavors.

First, I would like to express my sincere gratitude to my advisor Prof. Nobuyuki Yamasaki. He has patiently guided me from the beginning of the research up to the completion of this dissertation. He has always been helpful when I truly needed a hand. I am proud of being one of his first advisees to obtain a Ph.D. I wish to extend my gratitude to Prof. Fumio Teraoka, Prof. Kenji Kono, and Prof. Takahiro Yakoh for serving on my dissertation committee. They have dedicated their time and energy far beyond their duty. Their constructive criticism and invaluable advice have greatly improved this dissertation. I am grateful to my first advisor Prof. Yuichiro Anzai who helped me take the very first step toward this point. He never failed to encourage me.

I enjoyed every technical and non-technical discussion with all my past and current fellow colleagues. Special thanks go to Tsutomu Itou and Izumi Takata for their friendship. Without their unfailing support, this research could not have been completed. Thanks also to the members of the KODAMA tennis club for being there whenever I am stuck and needed to get away from research.

Finally, I would like to express my heartfelt thanks to my parents for their continuous encouragement and support. No words can adequately express my gratitude to them.

ABSTRACT

**REAL-TIME SCHEDULING OF
PRACTICAL IMPRECISE TASKS UNDER
TRANSIENT AND PERSISTENT OVERLOAD**

HIDENORI KOBAYASHI

The real-time computing community has developed scheduling techniques that overly reserve resources for the worst case scenario. Adopting these techniques in a dynamic real-time system leads to low effective processor utilization. The research presented herein considers both theoretical and practical aspects of imprecise computing to enable construction of theoretically overloaded real-time systems. The thesis established in the research is that systematic real-time scheduling of practical imprecise tasks evicts overprovisioning of resources and increases the effective processor utilization without causing a critical timing violation.

The dissertation first presents a practical imprecise computation model that enables compensation for terminated optional parts. Computations based on this model are represented by a linear task model to assess their feasibility in a polynomial time. Two scheduling algorithms are developed for two different types of overload. The M-FWP algorithm is targeted at transient overload caused by activation of aperiodic tasks. It maximizes the acceptance ratio of aperiodic tasks and resolves the overload in a short period of time. The SS-OP algorithm is targeted at persistent overload caused by activation of periodic tasks. It maximizes Quality-of-Service (QoS) of the whole task set and avoids drastic changes in the QoS of each periodic task. Both algorithms allow tasks to dynamically reserve computation time to prevent premature termination of optional parts with 0/1 constraints. The effectiveness of the scheduling algorithms is studied by two sets of simulations that model systems of transient and persistent overload. Moreover, the practicality of the whole approach is validated by developing an embedded real-time operating system and by implementing required supporting mechanisms for practical imprecise tasks as well as the presented scheduling algorithms. The dissertation concludes that the presented approach serves the purpose of evicting unnecessary reservation of computation time and contributes to the development of cost effective real-time systems.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Numerous computers support our life. We use desktop computers and workstations at home, office, and school. We also know that, while browsing the Internet, there are many server machines behind the Web pages. Yet, in contrary to our intuitive expectation, the number of processors used for those computers altogether only comprises less than 1% of processors in the world. All the remaining processors are in embedded computer systems that we use everyday without ever noticing or recognizing that there are processors inside them.

Most of the embedded systems are at the same time real-time systems. The primary reason why embedded systems are real-time systems is that they operate in our real world. Since events in the real world occur in the real time, embedded systems must react to stimuli with constraints of the real time.

The 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 a real-time system depends both on 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 that interval is called release time and the end is called deadline. The deadlines have three types depending on what could happen if a deadline is missed: if a catastrophe occurs on a deadline miss, then the deadline is hard; if the contribution to the system suddenly becomes zero but no catastrophe occurs, then the deadline is firm; and otherwise if the contribution to the system gradually degrades as the completion time is further delayed, then the deadline is soft.

A great amount of research on real-time computing has its basis on the computation model presented by Liu and Layland [1], which is referred to as *classical periodic computation model* in this dissertation. The classical periodic computation model allows only periodic computations to have hard deadline. The periodic computation is a form of computation that requests exactly one execution in a fixed interval called *period*. Other assumptions made about the classical periodic computation model are: relative deadline and period are equal; all computations are independent; no resource or

precedence constraint exists; the execution time of a computation is fixed; no computation suspends itself; and a computation can be preempted at any moment, which means that a computation can be interrupted immediately upon a request to execute another computation and then be resumed some time after that.

Liu and Layland also gave theoretical analyses of two scheduling algorithms that form the basis of almost all the theoretical work concerning real-time scheduling. These algorithms are called Rate Monotonic (RM) and Eearliest Deadline First (EDF). The RM algorithm schedules the computation with the shortest period first among all ready computations. Since the period of a computation is fixed in the classical periodic computation model, the priority assignment in the RM algorithm is static. On the other hand, the EDF algorithm adopts dynamic priority assignment and schedules the computation with the earliest absolute deadline first.

Ever since these two algorithms were presented, there have been a considerable number of arguments on which algorithm performs better in what conditions, some of which are summarized by Buttazzo [2]. The present consensus in the real-time computing community is that the RM algorithm can be implemented in a smaller overhead than the EDF algorithm, while the EDF algorithm can achieve higher processor utilization in many cases. Generally, higher processor utilization is desirable, because it implies that a larger amount of work is done in the same amount of time if the same hardware is used. Thus, high processor utilization is considered as an advantage in embedded systems where resources are much limited than those of desktop computers. Stated the other way around, the cost of the systems can be lowered by using cheaper processors that just meet the processing capability required to handle the workloads. One drawback of increasing the processor utilization is that the systems also have a higher possibility of being in overloaded conditions.

1.1 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 under a scheduling algorithm, it means that no task in the set violates its timing constraints under any circumstance [3]. A similar notion is the schedulability of a task set. A task set is only schedulable if it can be feasibly scheduled under at least one scheduling algorithm.

The feasibility of a task set based on the classical periodic computation model under the EDF algorithm can be assessed by simply checking the processor utilization. Under the EDF algorithm, the task set is feasible if and only if the processor utilization is less than or equal to one. Therefore, when the EDF algorithm is used, an overloaded condition, or simply overload, is said to occur when the processor utilization goes over one. In a more general context, overload is said to occur in a system when its task set is no longer feasible by an optimal scheduling algorithm.

The overload caused by tasks based on the classical periodic computation model were relatively easy to resolve because the workloads were static, which means that no

task with timing constraints is activated (or suspended) dynamically and none of their time attributes is changed at all. Another factor that facilitated the overload resolution was that, at the time this simple computation model was dominantly used, real-time system developers were able to identify the worst case scenario and estimate the worst case execution time tight and safe, meaning that the estimated amount is equal to or only slightly larger than the amount actually required, since they used simple hardware and software that were all developed and well analyzed by themselves. Consequently, overloaded conditions were easily resolved before the developed system was put to use, for example, by modifying the parameters or mixtures of tasks to demand less processing time, or alternatively by replacing the underlying hardware to supply higher processing capability.

In modern real-time systems, however, an overloaded condition is much harder to handle, because workloads have become dynamic and the state of overload is exposed only at run-time. In the background of this change is 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 only fixed stimuli at a rate known 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 works, their workloads were mostly static. On the other hand, robots have emerged to much wider areas lately. In fact, it has become not very uncommon to see an autonomous robot working in the same environment that we live. Moreover, robots are being sold even in a department store as pets and helpers to people. 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 three important factors that afflict the predictability of workloads in dynamic real-time systems.

- Applications in dynamic real-time systems need much more complex computation models than the classical periodic computation model to express more complicated timing characteristics. These computation models allow mixture of hard periodic real-time computations and soft aperiodic real-time computations. Some of them also allow aperiodic computations with firm deadline, which are called *sporadic computation*. A sporadic computation is characterized by its maximum arrival rate. In other words, its two consecutive activations are separated by at least an interval of certain length, which is called the *minimal interarrival time*. This assumption on the maximum rate allows scheduling algorithms to guarantee the timing constraints of sporadic computations that arrive dynamically at unknown times. There are also computation models that allow firm periodic computations to be activated and suspended dynamically. Furthermore, some models even allow periods to be modified dynamically. These extensions make the system workloads to change dynamically in a way hard to predict.

- The worst case execution time of an application is hard to estimate, since complex hardware used in modern real-time systems degrades the accuracy of estimated worst case execution time. High performance processors that can meet the requirement of dynamic real-time systems adopt 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 estimated worst case execution time can be either too pessimistic or unsafe to use in feasibility assessment in practice.

- The worst case scenario is hard to identify. The dynamic nature of environments makes it almost impossible to make correct assumptions on the incoming stimuli. And 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 also be noted that the commercial-off-the-shelf (COTS) components used to 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 only proves successful 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. Obviously, expensive processors are not desired in embedded system development, because they lowers competitiveness of the product by pushing the total cost higher. 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 type of deadline. If some computations have soft deadline, transient overload can be resolved by delaying their execution. Gardner and Liu [4] described and evaluated scheduling algorithms that isolate the effect of overruns caused by computations whose worst case execution time is not estimated safe. Buttazzo and Stankovic [5] presented the RED scheduling algorithm. In that algorithm, deadline tolerance of a computation is defined and used in overload to check whether each computation can still contribute to the system when its completion must be delayed. If not, the algorithm uses importance values attached to computations

to determine which computation to actually reject. Buttazzo et al. [6] gave a comparative study on such robust scheduling algorithms. Baruah and Haritsa [7] developed an asymptotically optimal scheduling algorithm called ROBUST that sustain the same level of effective processor utilization both in normal and overloaded conditions. An unfortunate result is that the ROBUST scheduling algorithm needs a processor that is twice as fast as the one used in a system without any possibility of undergoing overload. Thus, the solution may only be acceptable in a mission critical system.

1.2 Motivation

This research was motivated by the dilemma that became apparent in the development of complex real-time systems whose workloads vary dynamically and may cause overload only at run-time. The two conflicting demands behind the dilemma are that developers want to statically reserve resources as much as possible so that all timing constraints are met even in the worst case, while they also want to leave the same resources unreserved as much as possible so that only a small portion of resources are wasted as spare capacity in the other cases. Stated differently, designing real-time systems for high utilization will risk the temporal correctness, whereas designing the systems for perfect temporal correctness will increase the cost. Therefore, there is a crucial dilemma of meeting the timing constraints and fully utilizing the processor 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. Thus, finding a solution to the problem enables the system to dynamically adjust effective system load to a level where no critical constraint is violated. The ability to adjust the system load is important especially in embedded real-time systems, since they do not allow manual adjustment of workloads once the system is put to operation. In other words, some form of automatic adjustment mechanism must be adopted if the system may fall into overloaded conditions.

In order to establish such a good balancing point, we need to identify the critical and non-critical part of the system. In other words, we need to discriminate portions that must be strictly executed as requested from what does not need to. In this research, the discrimination is made within computations as well as among computations. The discrimination among computations means distinguishing firm and soft deadlines from hard deadlines, which has already been described before. On the other hand, the discrimination within a computation is accomplished by using the notion of imprecise computation.

The imprecise computation aims to obtain the best result with all available resources, exactly by distinguishing a portion of a computation that must always be completed from a portion that does not. The adjustment of load by imprecise computation is carried out by terminating computations prematurely to discard non-critical portions. A prematurely terminated computation can only return a partially correct, imprecise re-

sult. However, it is often desirable to receive an imprecise result before deadline than to receive a precise result later.

Many real-time applications can actually be implemented based on the imprecise computation model. For example, wavelet transform used in data compression and signal processing is well suited for 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 also be 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 [8, 9, 10, 11], network traffic management [12, 13], decision making under uncertainty [14], anytime learning in evolutionary robotics [15, 16], motion planning and robot control [17, 18], multi-target tracking [19], transactions in real-time databases [20], and fuzzy systems [21].

Unfortunately, despite thus many applications, the imprecise computation technique has not yet been used widely in industry. The primary factor that is making the hurdle higher is that, to the best of our knowledge, there is no research that develops imprecise computation model and scheduling algorithms with the purpose of providing a uniform imprecise computation platform in an integrated manner. Rather existing research remained in theoretical work or merely carried out a case study of imprecise computation by using ad hoc implementation methods.

1.3 Research Overview and Contributions

The goal of this research is to achieve high effective processor utilization without causing a critical timing violation. Attaining this goal allows building cost-effective real-time systems that can always guarantee timing constraints of critical computations and degrade the overall performance gracefully under overload. Therefore, it enables a real-time system to work under different conditions and in various environments without any modification.

To achieve this goal, this research takes a systematic approach that considers both theoretical and practical aspects of imprecise computation system. In particular, the subjects that this research focuses on are a practical imprecise computation model, scheduling algorithms including resource access protocols, and operating system support for scheduling and implementing practical imprecise computations.

The thesis established by this research is:

> Systematic real-time scheduling of practical imprecise tasks evicts overprovisioning of resources and increases the effective processor utilization with-

out causing a critical timing violation.

In support of this thesis, the research makes the following contributions.

- A practical imprecise computation model is developed to represent and implement real-world imprecise computations. This model redefines the practicality and applicability of imprecise computation by allowing more than one mandatory parts and also more than one optional parts in a computation. Moreover, these mandatory parts and optional parts can be freely interleaved. This bridges the gap between the existing theoretical models of imprecise computation and the practical characteristics of real-world applications.

- Two scheduling algorithms are developed to schedule computations based on the practical imprecise computation model. These algorithms work under two different types of overload, namely transient overload and persistent overload. In this dissertation, these two overloaded conditions are not distinguished by their length, although the transient overload tend to last for a relatively shorter period of time and the persistent overload for a longer period. What defines the overload as transient is the fact that the total system load is raised by activation of an aperiodic computation. By comparison, the persistent overload is defined by the fact that the total system load is raised by activation of a periodic computation.

  - The scheduling algorithm for the transient overload improves the acceptance ratio of aperiodic computations and focuses on overload resolution in a short period of time. This objective is similar to maximizing the competitive ratio. The competitive ratio is one of the important metrics for on-line scheduling algorithms for transient overload and refers to the ability of keeping the performance level competitive to that achieved by a clairvoyant scheduling algorithm. In the context of this research, this means that as many firm aperiodic computations are feasibly scheduled as possible.
  - The scheduling algorithm for the persistent overload improves the overall quality of service (QoS) by dynamically determining distribution of computation time among all optional parts of periodic computations. In contrast to the case of transient overload, it is more important to control the QoS than to maximize the competitiveness, because a longer period of overload is asserted to be worth the computationally expensive algorithms that maximizes the QoS. Another desirable property of the scheduling algorithm is that the QoS level achieved by each computation is kept at a stable level.

  Both algorithms allow resource sharing between periodic and aperiodic imprecise computations with bounded blocking time. Moreover, they ensure that no termination occurs during resource accesses in optional parts. Furthermore, these algorithms allow computations to dynamically reserve computation time for optional parts and also to cancel the reservation. The effectiveness of these schedul-

ing algorithms are validated through simulation studies of transient and persistent overload.

- Supporting schemes for implementing practical imprecise computations, as well as the two presented scheduling algorithms, are designed and implemented in a real-time embedded operating system completely developed from scratch. By actually implementing the supporting mechanisms, this research identifies and considers implementation issues concerning the practical imprecise computation model and the scheduling algorithms in a integrated manner. It also demonstrates that the presented approach is practicable to achieve the goal of this research.

The advantages of this systematic scheduling approach over other approaches that try to remove the uncertainty completely, for example by providing a method to estimate tighter worst case execution time, are: this approach does not depend on the types of a processor used; it does not require a rigid analysis of application programs, thereby allowing use of software components whose execution paths are only determined at run-time; and it can tolerate both transient and persistent overload.

The advantages over the existing support provided for imprecise computation are: it allows applications to be terminated in a less demanding manner; it allows use of different scheduling algorithms for different types of overload without having applications recompiled; and applications can be implemented in the same manner irrespective of whether the system is configured for competitiveness or QoS.

Finally, the scope of this research is limited to providing a systematic approach for scheduling practical imprecise computations. Specifically, it focuses on a practical imprecise computation model, scheduling algorithms including resource access control methods, and implementation and run-time support that can be provided by a real-time embedded operating system. The presented approach is targeted to a uni-processor system. 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. Hence, no computation is allowed to migrate from one node to another node. 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 estimation.

1.4 Organization

The rest of the dissertation is organized as follows. Chapter 2 summarizes the state of the art technologies related to imprecise computation. The technologies are summarized in terms of imprecise computation models, scheduling algorithms, resource access protocols, and operating system support for imprecise computation. Chapter 3 provides overview of the target imprecise computation platform and the guidelines that drive this

research. It also describes the specific assumptions of the platform. Chapter 4 describes the need for a more flexible imprecise computation model and formally defines the practical imprecise computation model. A linear task model is also presented to theoretically represent applications based on this computation model and is used to describe scheduling algorithms in the following chapters. Chapter 5 describes the scheduling algorithm developed for transient overload. It first describes the basic strategy and the flow of handling requests and then describes the scheduling algorithm with theoretical validations. Chapter 6 describes the other scheduling algorithm developed for persistent overload, in the same organization as that of Chapter 5. Chapter 7 presents the performance of the two proposed scheduling algorithms through simulation studies and compares them. Chapter 8 presents the implementation issues of these scheduling algorithms and presents their solutions. It also describes the structure of a scheduler that can implement both scheduling algorithms described in this dissertation, along with the implementation of mechanisms required for practical imprecise computations. Moreover, it presents numerical characteristics of the implemented kernel to show the practicality of the whole approach. Chapter 9 concludes the 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 technologies for imprecise computation. The survey is performed from the viewpoint of supporting imprecise computation. The survey begins with imprecise computation models that play a vital role on how imprecise computation is performed. The survey of both optimal and heuristic scheduling algorithms for imprecise computations are given next to establish the context of this research. The resource access control protocols are also summarized, since it is often required to consider processor scheduling and resource scheduling together at the same time. The survey ends with summarizing operating system support to assess the practicability of previous approaches.

2.1 Imprecise Computation Models

The design of imprecise computation models forms the basis of imprecise computation and has the largest impact on how overall imprecise computation is accomplished. Scheduling algorithms and operating system support are often deeply dependent on the imprecise computation model.

In the literature, the term imprecise computation model in its narrower sense refers to computation models that have one or more optional portions. Stated alternatively, a mandatory portion and an optional portion can be clearly distinguished within a computation. In a broader sense, on the other hand, this term refers to any computation models that can trade off the quality of results with an amount of a resource. Hence, we categorize the imprecise computation models into two classes. The first class is limited to the computation models that have a distinct optional portion within a computation and the second class includes the others.

Imprecise computation models with distinct optional parts The first class of imprecise computation models that have distinct optional parts are summarized in Table 2.1.

Among them, the simplest model is formalized by Lin et al. [22] where one computation is split into two parts: a mandatory part and an optional part. The mandatory

Table 2.1: Imprecise computation models with distinct optional parts

Model	Mandatory part	Metrics
Basic imprecise computation model [22]	Exists	Error
Extended imprecise computation model [8, 23]	Exists	Last output error
IRIS model [24, 25]	None	Reward
Models in anytime algorithms [26, 27]	Exists/None	Quality of solution

part is responsible for providing logically correct result (or service) that has the lowest acceptable quality, while the optional part is responsible for enhancing the quality of results produced by the mandatory part. The optional part can be terminated prematurely at any moment before completion and still provides useful approximate results. The quality of results in this model is expressed using the notion of error. The error is said to be one if an optional part is not executed at all. The error usually decreases as the optional part is executed and finally reaches zero when the optional part is completed. If the error of results never increases by allocating more time, the computation is said to be *monotone*. This simple model is referred to as *basic imprecise computation model* in this dissertation,

A slight extension of the basic imprecise computation model is also presented by Lin et al. [22] where an optional part has a *0/1 constraint*. An optional part is said to have a 0/1 constraint if its premature termination yields no meaningful result. Such optional parts with a 0/1 constraint should be either executed to completion or entirely discarded.

Feng and Liu [8, 23] extended the basic imprecise computation model to take into consideration the effect of error propagation. The extended imprecise computation model focuses on the case where there is an end-to-end deadline for a group of dependent imprecise computations, each based on the basic imprecise computation model. This group of dependent imprecise computations is referred to as composite task. If one of the computations within a composite task is terminated prematurely, its imprecise result will be used as input to the subsequent computation. Consequently, the execution time of this subsequent imprecise computation needs longer time to produce a precise result than otherwise. According to the extended model, this extension of execution time due to imprecise input occurs on both the mandatory part and the optional part, and the characteristics of their extension are given by two functions called mandatory extension function and optional extension function, respectively. The performance metrics used for the extended computation model is the output error of the last component computation within each composite task.

In all these computation models, there is an implicit precedence constraint between the mandatory part and optional part of a computation, because the optional part can only execute after the mandatory part produced the result of the lowest but acceptable quality. In fact, almost all the research that uses these computation models implicitly or explicitly assumes that there exists a precedence constraint.

The IRIS (Increasing Reward Increasing Service) Model [24, 25] is the same as the

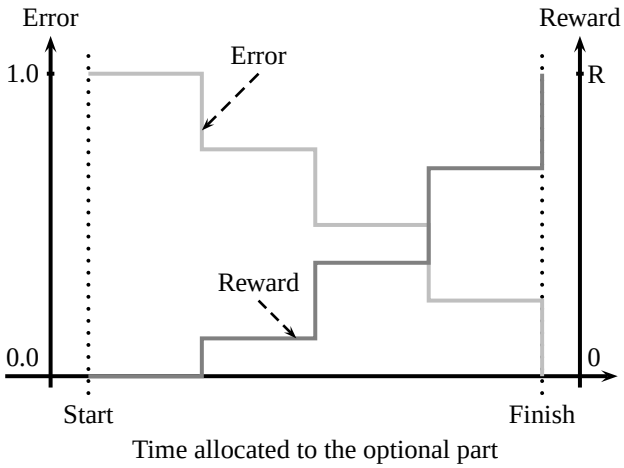


Figure 2.1: Relationship between error and reward

basic imprecise computation model except for two points: it has no mandatory part and it uses the notion of reward to express the quality of results. The reward expresses the degree of contribution. The reward of a task is larger when the error of its result is smaller and its relative importance is higher.

The relationship between the error and reward of an imprecise computation is illustrated in Figure 2.1. The reward is increased at the same time the error is decreased. However, the amounts of their improvement do not necessarily coincide. This is exactly where the difference between the error and the reward lies. While the error merely means the precision of a result, the reward includes its impact to the whole system as a degree of contribution. Thus, an identical amount of improvement in precision may have different level of impact on the overall performance. Hence, it can be said that the notion of reward is more general and more widely applicable than the notion of error.

The computation models adopted in anytime algorithms developed for real-time artificial intelligence also allow computation to be terminated prematurely. Note that the term anytime algorithm does not refer to a certain algorithm. It refers to a group of algorithms that can return meaningful result at any time [26]. The anytime algorithms were later categorized by Russell and Zilberstein [27] into two groups depending on the computation model used therein. The *interruptible algorithms* assume a computation model that solely consists of optional parts, while the *contract algorithms* assume the same computation model as the basic imprecise computation model with a precedence constraint. The contract algorithms need to complete its mandatory portion before executing its optional portion that can be interrupted at any moment. The performance of anytime algorithms is measured by the quality of solutions for the targeted problems. Thus, the computation models assumed in these algorithms do not have a fixed performance criteria.

Table 2.2: Imprecise computation models without distinct optional parts

Model	Load adjustment by
Skip-over computation model [28]	Skipping instances
(m,k)-firm deadline model [29]	Skipping instances
Window-constrained model [30]	Skipping instances
Elastic computation model [31]	Changing periods
Multiple versions model [32, 33]	Selecting one from multiple different versions

Imprecise computation models without distinct optional parts The second class of imprecise computation resolves the trade-off between the performance and the resource consumption without termination. These computation models are summarized in Table 2.2.

Some of the computation models take the strategy of executing only the instances that are critical. Computations found in telecommunication and video transmission can tolerate occasional skips, for instance, due to loss of packets. This robustness can be used to decrease the level of load. Specifically, by intentionally skipping some of the instances at a certain rate, the overloaded conditions are resolved at the cost of performance degradation.

The *skip-over* computation model is presented by Koren and Shasha [28]. In the model, a computation with the skip parameter of s can tolerate every one deadline miss out of s consecutive executions. In other words, after missing a deadline, the computation must at least meet next $s - 1$ deadlines.

Hamadaoui and Ramanathan [29] presented the *(m,k)-firm deadline* model. A computation based on their model must meet every m out of k deadlines. Thus, the computation only experiences a dynamic failure, if less than m deadlines are met in k consecutive invocations.

The *window-constrained* model presented by West and Poellabauer [30] has a more stringent constraint. The model is similar to but different from the (m,k)-firm deadline model in that instances of a computation are rigidly grouped. Specifically, if the constraint is specified as x/y, then at least x of them must meet their deadlines in every fixed window of y consecutive deadlines.

An approach that has virtually the same effect as but different from these computation models is presented by Buttazzo et al. [31] as *elastic computation* model. The elastic computation model characterizes periodic applications whose period can be changed. Such computations are found in control applications [34, 35, 36] as well as in multimedia applications. Changing a period influences both the load factor and the quality. For example, making the period of surveillance longer will decrease the load but at the same time degrades the quality. One of the advantages of this model over the computation models that exploit skips is that it has finer control over the load level. For example, a period can be increased by a factor of one and a half in the elastic model, while it is only possible to skip one out of two instances in the computation model with skips, which is

equal to increasing the period by a factor of two.

Multiple versions that have different execution time and different expected performance for a single function of the system can also serve as a means to trade off the quality of results for execution time. This is known as Design-to-Time approach [32] in real-time artificial intelligence and can be seen as an extension of N-version software [37] in fault tolerant computing. A further extension is given by Musliner et al. [33] which trades off execution time to the completeness of responses. These multiple versions are similar to having many optional parts with a 0/1 constraint but different in that shorter computations used to build up multiple versions can become a mandatory part when longer versions are not feasible.

In summary, there are two classes of imprecise computation models and both have their advantages and disadvantages.

The imprecise computation models that belong to the first class are potentially more suited to increasing utilization of resources, because the system that deploys these models has finer control over the system load. The effective load of the system can be adjusted continuously by these computation models since optional parts can be terminated at arbitrary points, while that by the models that exploit skips can only be adjusted discontinuously. Thus, the continuous load adjustment by the computation models in the first class has an advantage in increasing the processor utilization. In addition, these computation models can decrease the load in a relatively short period of time after the overload is detected. Hence, it is also easier to use these computation models than the computation models in the second class. The weak point of these computation models in the first class is that there is an unrealistic assumption. In particular, they have no place for compensation or recovery operations after termination of an optional part, because they assume that the optional parts can be terminated at no cost. However, real applications require extra work for termination. For example, if a database transaction based on the basic imprecise computation model is terminated prematurely, the database system has to roll back only half completed operations so that data consistency is preserved. Therefore, there is a mismatch between the computation model and the characteristics of real applications.

The imprecise computation models that belong to the second class do not have the unrealistic assumption made in the first class. Another advantage of this class is that legacy applications can be used with little modification. Moreover, these computation models rarely need special mechanisms for implementation. Their drawback is that they still see a computation as a complete black box and requires the worst case execution time of the whole computation to estimate the system load. Consequently, if the given worst case execution time is loose, these computation models may unnecessarily underutilize the processor. Moreover, the computation models that use multiple versions have another drawback that they are too expensive in practice both in terms of memory requirement and processing time. Therefore, multiple versions may not be suitable for most embedded systems, except for those that require fault tolerance severely.

By comparing these two different classes from the perspective of achieving the goal of this research, we conclude that an imprecise computation model similar to those in the first class is more suitable to form the basis of controlling the system load, provided that terminations are carried out in a more realistic manner. Hence, the rest of this chapter gives a survey of the state-of-the-art imprecise computation technologies only in relation to the first class of the imprecise computation models.

2.2 Scheduling Algorithms

A very large body of work developed scheduling algorithms for imprecise computations. Scheduling algorithms play a vital role in almost all real-time systems including imprecise computation systems. The overall performance of the whole system depends on how well the scheduling algorithm performs to meet the objective of the system.

This dissertation classifies the scheduling algorithms into two groups:

- a group of off-line and on-line optimal scheduling algorithms, and
- a group of on-line heuristic scheduling algorithms.

It is important to note the difference between off-line scheduling and on-line scheduling here, since the difference of these two types are often mixed up with that of static scheduling algorithms and dynamic scheduling algorithms. According to Stankovic et al. [38], a static scheduling algorithm has complete knowledge of given task sets and their constraints, while a dynamic scheduling algorithm may not know new task activations that may arrive in the future. By comparison, off-line scheduling is analysis or construction of schedule that is performed before the system is put to operation, while on-line scheduling is that performed during the system operation. It should also be noted that off-line heuristic scheduling problems are of little practical interest in the context of scheduling imprecise computation.

Optimal scheduling algorithms Optimal scheduling algorithms create feasible schedules that minimize or maximize a certain criterion. These groups of scheduling algorithms are summarized in Table 2.3.

Shih et al. [39] developed three algorithms for off-line scheduling of aperiodic imprecise computations to minimize the total error. The total error is minimized when the total amount of the unfinished portions of optional parts is minimized. The ED algorithm finds feasible schedules with the minimum total error when all tasks have only optional parts and have an identical weight. Under the ED algorithm, tasks are scheduled in the EDF order and are terminated at their deadlines even if they are not completed at that time. The complexity of the ED algorithm is $O(n \log n)$ where n is the number of tasks. Although the ED algorithm itself may be of little practical interest, it provides the basis of many other optimal scheduling algorithms. The algorithm F schedules imprecise computations with mandatory parts to obtain a schedule with the

Table 2.3: Optimal scheduling algorithms developed for imprecise computation model with distinct optional parts

Type	Algorithm	Workload	Objective
Off-line	ED, F, LWF [39]	Aperiodic	Minimize total error
	G [40]	Aperiodic	Minimize maximum error
	Binary Partition [41]	Aperiodic	Minimize total error
	LDF, DFS [39]	Aperiodic (0/1)	Minimize total error
	OPT-LU [42]	Periodic	Maximize average reward
On-line	NORA, ORA, OAR [43]	Aperiodic (FMC)	Minimize total error
	M-FED [44]	Aperiodic (WFMC)	Maximize competitiveness

minimum total error. The algorithm F uses the ED algorithm to test whether the given task set can be scheduled and to create the basic schedule. The complexity of the algorithm F is also $O(n \log n)$. This algorithm F is in turn used in the algorithm LWF repeatedly to minimize the total error when tasks have different weights. The complexity of the LWF algorithm is $O(n^2)$. Shih and Liu [40] developed another scheduling algorithm called algorithm G that uses the algorithm F to minimize the maximum error. The complexity of the algorithm G is $O(n^2)$. Later, Shih et al. [41] improved the LWF algorithm to decrease the time complexity to $O(n \log^2 n)$.

The case where optional parts have 0/1 constraints is considered by Shih et al. [39]. They showed that finding an optimal schedule that satisfies the 0/1 constraints and minimizes the total error is NP-complete. However, they also found that a special case of that problem where all optional parts have an identical execution time can be solved. They presented two optimal scheduling algorithms called the LDF algorithm and the DFS algorithms. The LDF algorithm can only schedule imprecise computations with the same release time in $O(n \log n)$ time complexity, while the DFS algorithm can schedule imprecise computations with arbitrary release time in $O(n^2)$ time complexity. These algorithms, again, use the ED algorithm to test the feasibility of the given task set and to create a basis for the final schedule.

For periodic computations, an optimal off-line algorithm called OPT-LU is developed by Aydin et al. [42] to maximize the average reward. Interestingly, it was shown that there exists an optimal schedule where the computation time allocated to optional parts of a task is constant in every period. The significance of the result is that it simplifies the optimization problem remarkably because otherwise a potentially exponential number of unknown variables exist in the optimization problem. In fact, owing to this property of optimal solutions, the complexity of the OPT-LU algorithm is reduced to $O(n^2 \log n)$. The advantage of the OPT-LU algorithm is that is makes the performance of imprecise computations more predictable. Since the computation time that can be allocated to each optional part is determined off-line as a fixed value, system developers can predict the performance of each application beforehand and also make appropriate modifications to maximize the performance. However, the optimality of this identical

optional computation time and thus that of the OPT-LU algorithm no longer holds, if relative deadline and period are not equal, if priorities are assigned statically, or if reward functions are not concave. Moreover, it was shown that the optimization problem becomes NP-Hard when reward functions are convex.

The downside of these off-line scheduling algorithms is that they lack flexibility to integrate scheduling of dynamically activated computations. On the other hand, this research uses imprecise computation to cope with overload triggered by dynamic activation of both aperiodic and periodic computations. Thus, off-line scheduling algorithms are not suitable for scheduling overload of imprecise computations targeted in this research.

The optimal on-line scheduling algorithms need assumptions on workload. It is unfortunately known that there exists no on-line optimal scheduling algorithm that always finds, for any system of on-line imprecise tasks, a feasible schedule with the minimum total error whenever the system has feasible schedules [43].

An assumption made by Shih and Liu [43] is called *feasible mandatory constraint* (FMC). The feasible mandatory constraint is satisfied if at the arrival time of every new task, its mandatory parts, together with the yet to be completed mandatory parts of previously arrived tasks, can always be feasibly scheduled to complete by their deadlines. Three optimal scheduling algorithms were presented for four different types of workload that satisfy the feasible mandatory constraint. The four workloads are characterized by the existence of off-line computations and the release time of on-line computations. All three scheduling algorithms are slack stealing algorithms that execute optional parts whenever mandatory parts have slack. The NORA algorithm schedules computations to maximize the total error when there is no off-line computation and all on-line computations are ready upon arrival. The ORA algorithm maximizes the total error when there are off-line computations and all on-line computations are ready upon arrival. The OAR algorithm creates an optimal schedule when on-line computations become ready at arbitrary times, regardless of whether off-line computations exist or not. The complexity of the NORA algorithm and the ORA algorithm is $O(n \log n)$ and that of the OAR algorithm is $O(n \log^2 n)$.

Baruah and Hickey [44] presented a more realistic assumption on workloads called *weak feasible mandatory constraint* (WFMC). The weak feasible mandatory constraint is satisfied when the mandatory parts of all tasks in the task set can be scheduled to complete before their deadlines by an optimal off-line scheduling algorithm. The algorithm M-FED is also presented for imprecise computations of uniform density. The algorithm M-FED ranks all mandatory parts higher than all optional parts and, within each group of parts, assigns higher priority to one with earlier deadline. This strategy to assign higher priority to all mandatory parts than to all optional parts was originally presented by Chung et al. [45]. However, the algorithms developed by Chung et al. that follow this mandatory-first strategy were not optimal because they were targeted for periodic computations; it is the OPT-LU algorithm that is optimal for scheduling periodic computations. By contrast, the algorithm M-FED is optimal in the sense that it reaches the

Table 2.4: Heuristic scheduling algorithms developed for imprecise computation model with distinct optional parts

Type	Algorithm	Workload	Objective
On-line	Algorithms for IRIS [25]	Aperiodic	Maximize average reward
	OR-ULD [46]	Aperiodic (0/1)	Maximize average reward
	INCA [47]	Periodic (0/1)	Maximize average reward

theoretical limit of competitiveness for on-line scheduling of aperiodic computations. The complexity of the algorithm M-FED is the same as that of the EDF algorithm.

These on-line optimal scheduling algorithms assumes only aperiodic computations. Thus, they consider no relationship between computations. On the other hand, many real-time applications can be modeled as periodic computations. A periodic computation may not tolerate acceptance of one instance and rejection of another, since the whole series of instances may altogether provide the service of the periodic computation. Moreover, an algorithm that schedules overload should take advantage of the periodicity of real-time computations for efficiency, since by doing so it becomes possible to predict the future arrivals of periodic computations.

Heuristic scheduling algorithms Heuristic scheduling algorithms are summarized in Table 2.4. The reason to use heuristic algorithms is that either the whole or a part of the optimization problem in concern is too difficult to solve in practice. Stated in another way, developers must search for a good balancing point between the quality of schedules and the cost of scheduling.

Dey et al. [25] presented three heuristic scheduling algorithms for on-line scheduling of aperiodic workloads. The computations assumed therein is based on the IRIS model. These computations have a concave reward function and can be executed arbitrarily long until their deadline is reached. Two of the algorithms take a two-level approach. The top-level algorithm solves the static problem of maximizing reward. The lower-level algorithms actually schedule computations using the results of the top-level algorithm. The third algorithm is a greedy algorithm. It emulates the *balanced reward processor sharing policy* that is idealized and cannot be implemented in practice. Their simulation study show that the two-level scheduling approach that uses the EDF algorithm as the lower-level scheduling algorithm performs well in terms of reward and the number of context switches. The run time complexity of that algorithm is $O(n^2)$. Although the algorithms are all designed for the IRIS model, a method for converting imprecise computations into computations based on the IRIS model is also presented.

As stated earlier, the problem of optimal scheduling imprecise computations whose optional parts have a 0/1 constraint is intractable. However, many real-world applications have this constraint. Hansson et al. [46] presented using the OR-ULD algorithm to schedule such imprecise computations. The OR-ULD algorithm was originally developed in [48] for scheduling real-time transactions. When the OR-ULD algorithm is ap-

plied to the imprecise computation model, a mandatory part is modeled as a hard transaction and an optional part is modeled as a firm non-critical transaction. The precedence constraint between the mandatory part and the optional part is enforced by assigning a virtual deadline to the mandatory part. When the admission controller of the system spots overload at run-time, the OR-ULD algorithm tries to find and discard one or more of transactions whose utility loss density is the smallest among the active transactions until the overload is resolved. The utility loss density in the context of the imprecise computation model represents the weighted error. The complexity of the OR-ULD algorithm is $O(n \log n)$.

Mejía-Alvarez et al. [47] presented the INCA server that incrementally searches feasible solutions to maximize reward. What is interesting about the approach is that not only the computations but the scheduling algorithm itself is based on the imprecise computation model. When a system falls into overload by dynamic activation of a periodic computation or it becomes underutilized by dynamic suspension of a periodic computation, the INCA server iteratively executes an approximate algorithm to select a set of optional part to execute to maximize the reward. In the case of overload, the first step of the INCA server disables some optional parts to eliminate the overload, and then the subsequent steps improve the solution. The complexity of the k-th step in the algorithm is $O(n^{k+1})$. The INCA server is stopped when there is no more slack left in the system or the k-th step did not improve the schedule created by the $(k-1)$-th step. It is also stopped and restarted when another activation or suspension request arrives the system.

These heuristic scheduling algorithms are developed out of theoretical interest to solve one specific intractable scheduling problem of imprecise computations. What is desired to achieve the goal of this research is an algorithm that handles both periodic and aperiodic imprecise computations. In particular, a scheduling algorithm that can handle dynamic activation of both periodic and aperiodic computation is desired to adopt imprecise computation in a wide variety of dynamic real-time systems.

In summary, there is a huge body of work on scheduling of imprecise computations, since the scheduling algorithms have the largest impact on the overall performance. Many optimal scheduling algorithms were developed for maximization of reward or for similar objectives, while many optimization problems were proven as intractable. The optimal scheduling algorithms are differently characterized by their criteria. Among them, the mandatory-first strategy is competitive, while slack stealing is more suitable for the purpose of maximizing reward. These scheduling algorithms are dependent on the computation model assumed therein. An optimal scheduling algorithm often becomes non-optimal when one of its assumptions is relaxed or a slightly different computation model is used. The heuristic scheduling algorithms tend to be more robust to such changes. However, they are only suboptimal for most of the cases.

Despite the large number of scheduling algorithms, further development of scheduling algorithms is still necessary if the underlying computation model is changed. And

in that case, the time complexity of the algorithm must always be considered from the beginning of its design so that it can be practically used under overload.

2.3 Resource Access Protocols

It is not too much to say that there exists no resource access protocol that is developed specifically for imprecise computation. Instead of developing a new protocol, the resource contention problem has been solved by either:

- using resource access protocols that were not developed for imprecise computations; or
- creating a schedule that has no resource contention.

The simplest approach for establishing mutual exclusion without explicitly creating a schedule that has no resource contention is to make all resource accesses non-preemptive. This is a plausible approach when the duration of resource access is short. Otherwise, the system schedulability of the system decreases due to blocking time, since a computation with lower priority can block any computation with higher priority.

More sophisticated resource access protocols allow a computation with lower priority to inherit either directly or indirectly the higher priority of the blocked computation, so that the priority inversion phenomenon is rectified. The common assumptions here are that the shared resources are implemented by means of critical sections and that critical sections are *properly nested* [49]. Two critical sections are said to be properly nested when one of them is wholly contained in the other one or there is no overlapping between them. Likewise, n critical sections $(z_1, \ldots, z_n)$ are properly nested only when the critical section z_i is wholly contained in the outer critical section z_{i+1} or there is no overlapping between z_i and z_{i+1} for $i = 1, \ldots, n-1$.

Sha et al. [49] presented the Priority Inheritance Protocol and the Priority Ceiling Protocol. The Priority Inheritance Protocol can bound the priority inversion phenomenon, but it has two weak points. First, a *chained blocking* occurs. A high priority computation that shares m resources with low priority computations may be blocked at most m times for the duration of each critical section. Thus, the total blocking time is not minimized. Second, a deadlock can occur, which must be avoided in mission critical real-time systems. The Priority Ceiling Protocol overcomes these problems by assigning each resource a static *priority ceiling*. A priority ceiling of a resource is defined as the highest priority of the computations that may access the resource. At run-time, a computation can access the resource only when its priority is higher than any priority ceiling of resources that are used at that instant. Using that protocol, a computation can be blocked for at most the duration of one critical section. This upper bound on the blocking time guaranteed by the Priority Ceiling Protocol is the smallest possible in practice, because providing an even smaller upper bound requires eliminating all the

possible priority inversion phenomena, which is impossible in systems with complicated task sets.

One of the limitations of the Priority Ceiling Protocol was that the system must use a fixed priority scheduling algorithm, since priority ceilings were calculated statically. This limitation was later relaxed by Chen and Lin [50] which extended the protocol for deadline scheduling algorithms and resources with multiple instances. This dynamic version is called Dynamic Priority Ceiling Protocol.

Another limitation of the Priority Ceiling Protocol is that it considers only periodic computations. By contrast, the Dynamic Deadline Modification protocol presented by Jeffay [51] is designed for sporadic computations. The protocol uses two notions of deadline. A *contending deadline* is given by adding the release time and the minimum interarrival time of a computation. The contending deadline is used for scheduling computations that have not started execution. On the other hand, an *execution deadline* is given by the minimum of the contending deadline and the time given by adding the start time and the minimum interarrival time of computations that requests the same resource. The execution deadline is used for computations that have started but have not completed its execution. The limitation of the Dynamic Deadline Modification protocol is that no access to resources should be nested.

A simpler and more general protocol is the Stack Resource Policy protocol developed by Baker [52]. The protocol uses static *preemption levels*, instead of modifying deadlines dynamically, and allows nested accesses to resources with multiple instances. Under the Stack Resource Policy protocol, every computation is assigned a preemption level usually calculated by the inverse of its relative deadline. Moreover, every resource is assigned a dynamic *ceiling* that is the maximum of zero and the preemption levels of all computations that may be blocked by accessing the resource. Furthermore, there is a system wide ceiling called *current ceiling* or *system ceiling* which is the maximum of all ceilings. Using these notions, the Stack Resource Policy protocol restricts a computation to start its execution only when its priority is the highest among all ready computations and its preemption level is higher than the current ceiling. The maximum blocking time of a computation under the Stack Resource Policy protocol is limited to the maximum duration of one outermost critical section. Moreover, the Stack Resource Policy protocol, as well as the Dynamic Deadline Modification protocol, allows computations to share their run-time stack, which is greatly desirable for embedded systems. Another desirable feature of the Stack Resource Policy protocol is that its implementation cost and run-time overhead is small compared to other protocols that ensure the same blocking time. The application of the Stack Resource Policy protocol to soft aperiodic tasks and its schedulability analysis is later provided by Lipari and Buttazzo [53].

Unfortunately, all these protocols, including the ones that make resource access non-preemptive, are only applicable to the second class of imprecise computation models that do not allow termination of optional parts. A problematic situation can occur when these protocols are simply applied to the first class of imprecise computation models. Suppose if a resource access is terminated in an optional part that is holding resources.

Then, it is most likely that the consistency of resources is broken or, in the case of accessing hardware resources, the state of resources becomes unstable. It could also lead a deadlock if a previously acquired lock was not released due to the termination.

Some of the concurrency protocols developed for real-time database systems aggressively aborts transactions. The motivation for aborting transaction is that duration of resource accesses tends to be relatively long in database systems. It is even probable that a transaction begins with locking a resource and ends with releasing that. Huang et al. [54] evaluated the strategy of aborting transactions. They showed that a strategy of conditional aborting where transaction is aborted depending on how close a lower priority transaction is to its completion performs better than the priority inheritance or simple priority abort strategies. Kuo et al. [55] presented aborting lower priority transactions based on the result of schedulability analysis. The strategy is similar to the notion of imprecise computation in that a transaction is only aborted on overload. However, the cost of rolling back a premature transaction is too expensive in embedded real-time systems. Moreover, if a critical section corresponds to an actual hardware resource, it may not be possible to stop and terminate the access at the requested instant.

An alternative approach is to create a schedule that have no resource contention at all. The most extreme of this approach was adopted in the MARS project [56] where a static schedule was created off-line. The obvious drawback of creating a static schedule is its inflexibility to dynamic workloads. Since the motivation for using an imprecise computation model is to cope with dynamic workloads in various environments, this approach is not suitable for imprecise computation. More dynamic approach is the planning scheduler of the Spring project [57]. The planning scheduler attempts to construct a feasible schedule dynamically by adding one computation at a time to the previously constructed feasible schedule. Since the scheduling problem in general is NP-complete, heuristic algorithms were used [58]. The Spring project also developed a custom co-processor to speed up scheduling [59]. However, the same strategy is not always feasible in commercial real-time systems where cost is an important factor.

In summary, there is very little work on resource sharing among imprecise computations. The most annoying problem here is that optional parts can be terminated at any instant. In particular, delaying the termination of an optional part that holds a resource can cause a deadline miss in the worst case, while terminating the optional part can risk the consistency of resources. Furthermore, accesses to hardware resources often cannot be interrupted before completion.

2.4 Operating System Support

There are three ways of implementing imprecise computations. They are all suggested by Lin et al. [22].

The *milestone* method is used for imprecise computations with monotone optional parts. The method saves intermediate results while an optional part is being executed so that the best result is immediately available on termination. The points where the

results are saved are called milestones or checkpoints. Analyses on where to place the checkpoints is found in [60, 61]. Since the method assumes that checkpoints are performed by an operating system, it requires that the operating system be aware that it is an optional part that is being executed.

The *sieve function* method is used for imprecise computations that 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. Thus, the implementation of the method depends heavily on what scheduling algorithm is used by the operating system.

The last method is used for imprecise computations with multiple versions and thus called *multi-version* method. For this method, techniques developed for fault tolerant computing, for example [62, 63], can be used. For imprecise computations implemented by this method, the support is often provided in programming languages [64, 65, 66] and not in operating systems.

There are a few operating system that actually provides support for these methods to implement imprecise computations.

The Concord system presented by Lin et al. [22] supports both the milestone method and the sieve function method in a client server structure. On the client side, there are a caller and a handler, and on the server side, there are a supervisor and a callee. An imprecise computation is started when the caller requests a service through the *impresult* primitive. In response to the request, the system sets up a dedicated supervisor and passes the arguments of the impresult call to the supervisor. The supervisor is an active entity controlling the execution of the callee that implements the requested service based on the basic imprecise computation model. In addition to the request itself, the supervisor receives a handler. This handler acts as a flag that indicates whether intermediate results should be saved at all. After the supervisor is set up, the callee is started under the control of the supervisor. If the callee has a monotone optional part, it uses the *impreturn* primitive to send imprecise results to the supervisor as it produces them. If the callee has sieve functions, the callee itself asks the system scheduler whether the sieve function can be executed before the deadline. In both cases, if the callee completes its execution before the deadline, then the precise result is sent back to the caller via the supervisor. Otherwise, the callee is terminated on deadline and the imprecise result is returned to the caller after the handler is invoked. The role of the handler here is to make the imprecise result meaningful to the caller.

Hull et al. [67] developed the Imprecise Computation Server that provided support for monotone imprecise computations in as much the same structure as the Concord system. The Imprecise Computation Server was implemented on top of the RT-Mach operating system [68]. The drawback of these checkpointing mechanisms is that the performance of the system can be degraded substantially, if overheads of an checkpoint is not negligibly small.

In the ARTS system [69], a real-time object is defined with a *time fence*, which can be used to implement imprecise computations. The time fence specifies that a certain

operation be executed within a predefined time. If, at run time, this constraint cannot be met, a handler is invoked. The handler should be implemented by the developer of the object to maintain its consistency on termination. Using this mechanism, imprecise computations can also be implemented effectively by making the handler to return the best result on termination.

The RT-Mach operating system provides a similar mechanism [70]. A timing exception is handled by a dedicated thread whose priority can be set irrespective of the original priority of the faulting thread. The dedicated thread, however, must be created with a fixed priority before the the main thread runs.

The Chimera operating system developed at Carnegie Mellon University has a sophisticated way of handling exceptions [71]. When a timing error is detected, an exception handler is executed in the same context as the original computation. When the original computation holds a resource at the time of a deadline miss, the invocation of the handler is delayed to maintain the data consistency. Moreover, programmers can select from three possibilities on what to do after the handler is finished: whether the original computation be restarted, continued, or exited immediately.

One of the major limitations of using these timing exception handling mechanisms is that they are only applicable to imprecise computations whose entire mandatory part precedes their optional part. This limitation is also present in the milestone method.

Marty and Stankovic [72] developed a kernel level thread package for the Spring real-time system. The thread package has new semantics centered around the guarantee of child threads. If the execution of a child thread is necessary for a parent thread to work correctly, the child is given a *Dependent* guarantee. Otherwise if the execution of the child thread is only desirable, the child is given an *Independent* guarantee. The schedulability for a thread group with a Dependent guarantee is analyzed at the time the parent thread is created, while that for a thread group with an Independent guarantee is analyzed at the time the child threads are spawned. Using these two different semantics, imprecise computations can be implemented in all three ways. A mandatory part can be implemented as the parent thread and as a group of child threads with a Dependent guarantee, whereas an optional part can be implemented as a group of child threads with an Independent guarantee. The thread package allows safe termination of computations by terminating threads only at logical boundaries, which enables implementation of sieve function methods. Furthermore, the multi-version method can naturally be conducted using multiple threads.

In summary, despite the fact that there are operating system supports for all three implementation methods, operating system support for imprecise computation is another area that is not explored in depth. In fact, there is only a very little research that actually implements supporting mechanisms for imprecise computation. Moreover, every one of the existing approach has its drawback. The checkpointing mechanism may incur a large overhead, whereas the exception handling mechanism cannot complete recovery operations before the original deadline. The powerful semantics provided by the Spring thread package requires a powerful planning scheduler that cannot be implemented in

many embedded real-time systems.

2.5 Summary

This chapter gave a survey of the state of the art technologies for imprecise computation. There is a large body of work on both optimal and heuristic scheduling algorithms. In contrast, there is relatively little work on imprecise computation models with distinct optional parts, resource access protocols, and operating system support. This fact points out that the existing work on imprecise computation merely remained in the theoretical domain or served as a case study to show the effectiveness of imprecise computation and that there was no work that provided an integrated platform.

In the absence of such integrating work, unfortunately almost all work on imprecise computation is misled by two invalid assumptions:

- an optional part can be terminated at any point without the need for compensation; and
- a mandatory part always precedes an optional part in an imprecise computation.

As for the first assumption, the developers of scheduling algorithms may argue that it is the work of operating system developers that should adequately maintain the execution state of imprecise computations and let them do recovery operations after termination. Our opinion differs from this. We think such care must be taken from the beginning of the design for efficient imprecise computation. In other words, the computation models and the scheduling algorithms must also consider the need to execute recovery operations, so that operations of arbitrary length can be supported efficiently.

As for the second assumption, some argue that applications with complex structures nonetheless can be implemented by composing them with a group of simple mandatory and optional parts with precedence constraints. However, this increases the scheduling overheads. Moreover, the precedence constraints between them can make the scheduling algorithm complicated. Thus, it is desirable to hide complex structures within an scheduling entity.

The following challenging demands concerning practical imprecise computation are still not met by the state of the art imprecise computing technologies:

- Application designers need a more flexible imprecise computation model to freely mix mandatory parts, monotone optional parts, and optional parts with 0/1 constraints. This means that execution of a mandatory part should be allowed after an optional part. They also need a less demanding way to terminate optional parts. Specifically, recovery operations as compensation for terminated optional parts must be included in the computation model. Moreover, the compensation operation must be scheduled before the original deadline. It is also important that the compensation mechanism is not too much dependent on the operating system for

better portability. These together can eliminate the two invalid assumptions made in the literature.

- Application designers need a mechanism to dynamically obtain information on the amount of computation time allocated to optional part, so that they can program their applications to adopt to the system load and to the resource availability. This allows an application to adjust its behavior for different situations without manual intervention. Since general interactions between the scheduler and applications cannot be accomplished by the operating system alone, scheduling algorithms must also be designed to handle such requests dynamically.

- Practical system designers need different scheduling algorithms for different types of overload. However, there is little work that focuses on difference between transient overload and persistent overload. Under the transient overload, it is more desired to resolve the overload in little overhead than to use a computationally expensive algorithm to maximize the QoS. On the other hand, under the persistent overload, it is the exact opposite: maximizing the QoS is more desired than coping with the overload in little overhead. In addition, scheduling algorithms for transient overload should be capable of handling dynamic activation of aperiodic computations, while the scheduling algorithm for persistent overload should be capable of handling that of periodic computations, because these activations are closely related to the nature of respective types of overload.

CHAPTER 3

OVERVIEW OF TARGET IMPRECISE COMPUTATION PLATFORM

This chapter describes the overview of the target imprecise computation platform that serves as the reference system model of this research. The chapter first describes the design philosophy of the platform. Then, it describes the system model and the assumptions made there.

3.1 Design Philosophy

The design philosophy of the target imprecise computation platform is to maximize the performance while achieving the maximum tolerance against uncertainty. The tolerance against uncertainty per se is an important factor, as the goal of this research includes enabling construction of a cost-effective real-time system that can run in various conditions without manual interventions. However, the constructed system would become of little practical interest, if its performance is considerably lowered by enhancing its tolerance against uncertain workloads. Therefore, we focus on the performance of the system in addition to achieving the goal in the literal sense.

As was stated previously, the performance criterion is different under different types of overload. We aim to minimize the number of rejected aperiodic computations under transient overload and to maximize the QoS under persistent overload. In the context of this research, the QoS is only said to be high when the reward of a computation stays at a high level without undergoing a radical change in few periods, because users do not want the performance level of an application to change drastically. In addition to the above, the performance of aperiodic computations is considered in terms of timeliness. The timeliness for aperiodic computations is an uncommon criterion in the literature. The real-time computing community has only focused on shortening the response time

of aperiodic computations. By comparison, this dissertation assumes that an aperiodic computation has a *desired response time* and that its contribution to the system does not degrade if it completes in the desired response time. This assumption is based on an observation that applications in the real world need only to respond in a certain length of time. For example, an aperiodic computation that interacts with human users does not need to respond as quickly as possible, because the users would not be able to recognize the difference of one micro second in the response times.

Before going any further to derive specific guidelines for this research, it must be pointed out first that regarding the imprecise computation model as an almighty tool that can handle all the uncertainties in the system and that can evict all the needs for theoretical analyses is certainly a naive misconception. For example, a state of unbounded priority inversion can still lead to a deadline miss even if all the computations are imprecise. Moreover, it is quite clear that the ability of imprecise computations to handle the affect of uncertainty is not unlimited and depends on a specific set of computations generated from given applications. Specifically, this ability is small if the ratio of optional part to the whole computation is small. With this in mind, our first step toward the goal should be to discriminate the parts where rigid real-time analysis is required from those where uncertainty is allowed by using the imprecise computation model.

In principle, we allow uncertainty to exist only in the application workloads; all the other parts should behave predictably, so that the capability of imprecise computations to handle uncertainty is all dedicated to provide the maximum flexibility for applications. In other words, the imprecise computation platform must support imprecise computation in a predictable manner.

The specific guidelines for the imprecise computation platform targeted in this research are listed in the following.

- Imprecise computation models with distinct optional parts should be deployed in the imprecise computation platform. As was stated in Chapter 2, these computation models are more suited to sustain the system utilization high and to control the load level. Moreover, the imprecise computation platform should accept these computation models if and only if they allow application developers to specify what actions should be taken if optional parts are terminated prematurely. Otherwise, the platform must provide mechanisms dedicated to handling this impracticality. However, in that case, the platform may not be general enough to support a wide range of applications.

- The imprecise computation platform should provide only mechanisms needed to implement imprecise computation. In other words, the platform itself should not be responsible for providing one or more dedicated services for just one specific computation. For example, the platform should not be responsible of rolling back the effect of prematurely terminated operations or of returning imprecise results to specific client applications. Otherwise, the platform would be bloated up by such dedicated services, since imprecise computations are diverse in practice.

- Use of general programming languages and compilers should be allowed in development of applications, so that not all software components must be developed within the community. On the other hand, it should be legitimately requested that each computation be attached with their meta data that express time characteristics so that at least the feasibility of the worst case scenario can be rigidly analyzed. Although allowing different languages is not directly related to the performance of an imprecise computation system, it is more than desirable in many practical situations to shorten the time-to-market and to lower the production cost. Another reason to avoid assumption on programming languages is that language support for imprecise computation almost always requires operating system support. In fact, a programming language can be of little help if the amount of available time can only be determined dynamically. On the other hand, there is little significance in using imprecise computation model if the amount of available time can be predicted off-line.

- Different scheduling algorithms should be used for systems with different types of workloads, since no scheduling algorithm fits all situations well. For instance, it is preferred to execute aperiodic computations before optional parts to enhance responsiveness of the system in some cases, and vice versa in other cases. In this research, we distinguish the difference between transient overload and persistent overload and develop two scheduling algorithms for the respective cases. We assert that there is no use in dynamically switching between these scheduling algorithms, since it requires that the characteristics of the future workloads be predicted. This requirement is unlikely to be met in systems working in dynamic environments targeted in this research. Moreover, supporting two scheduling algorithms can take up a larger portion of memory than supporting only one scheduling algorithm. On the other hand, system developers should be able to change scheduling algorithms easily in the development phase. This means that a uniform interface between the scheduler and applications must be designed to support all common features of the scheduling algorithms so that everything under the interface remains transparent to applications.

- It should be the last choice to increase the number of scheduling entities. Keeping the number of entities small works in favor of keeping run-time scheduling overheads small. Thus, it is desirable to hide complex structures of application programs within computations. Taking a step further, a precedence constraint between computations should also be hidden by merging these computations into one whenever possible. If it is impossible to do so, deadline modification techniques [73] should also be considered as an alternative to eliminate the constraint.

- Each computation should be protected from one another. That is, a malfunctioning computation should not affect the execution of other computations in any way. Thus, it is important to completely isolate any possible side effect of termination

from other computations. Specifically, an additional operation needed at termination should not delay other computations in the system.

- The imprecise computation platform should facilitate extension and evolution of systems. These capabilities are desirable also for embedded systems. Intuitively, embedded systems do not change once it is constructed and put to operation. However, if a whole series of an embedded product is considered, newer models may be based on older ones. Thus, the imprecise computation platform should be general enough to accept changes in the applications and hardware. Moreover, there should be no implicit information flow between the platform and the applications, because such tacit understandings may not hold with the new applications.
- Any modules of the imprecise computation platform should be implemented to allow preemption unless there is a serious reason not to do so. Allowing preemption not only improves the predictability but also makes theoretical analysis simpler. Moreover, preemptive systems are generally more efficient than non-preemptive systems. Thus, it leads to higher effective processor utilization.

3.2 System Model

The system model assumed in this research is illustrated in Figure 3.1.

Every application given to the imprecise computation platform must provide both a program and a set of meta data. The program is what defines the logical structure and behavior of the applications. The meta data is what defines other attributes of the computations and must include all the attributes required for their timely execution. In particular, the meta data is a set of information on the type, time attributes, and timing constraints of computations. The type of a computation indicates whether it is precise or imprecise. The imprecise computations are confined to those that have one or more distinct optional parts within themselves. The time attributes represent the time characteristics that are inherent in the computations. These are the execution time of the mandatory and optional parts, the periodicity of the computations, and the reward which is expressed as a function of time given to the optional parts. The timing constraints are artificial restrictions on execution of computations such as the release time and the deadline.

We assume that all applications are stored in memory before the system starts, meaning that their executable images are loaded into memory before operation. Thus, the imprecise computation platform does not allow computations to be migrated from one node to another node. Hence, it is actually the activation and suspension requests that line up in the First-In-First-Out (FIFO) request queue in Figure 3.1.

The load hypothesis of the system in concern is different from both the feasible mandatory constraint and the weak feasible mandatory constraint, in that not all computations are regarded to have uniform characteristics and that only a part of computations are assumed as always feasible.

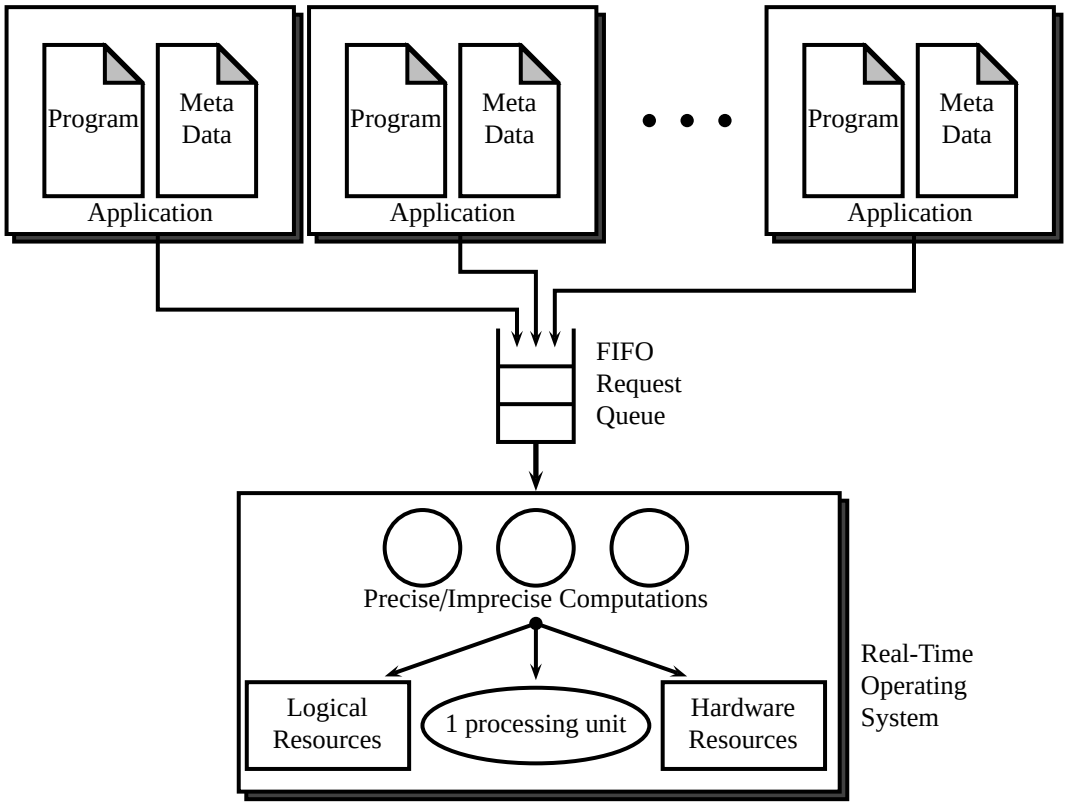


Figure 3.1: System model

Firstly, the periodicity of computations is considered. In particular, the workload of the system consists of periodic and aperiodic computations that are defined as follows. If a computation is periodic, its instances are invoked only at a regular rate and must be completed before deadlines that also come at a regular rate. The period of a periodic computation does not necessarily equal to the relative deadline. On the other hand, if a computation is aperiodic, there is no rule concerning its invocation. This research does not distinguish sporadic computation from periodic computations, because the worst case execution requirement of a sporadic computation can be expressed by that of a periodic computation. Moreover, guaranteeing the timing constraints of sporadic computations can lead to lower processor utilization, since their actual execution time requirement can be lower than its worst case.

Secondly, the criticalness of deadline is considered. The type of deadline can be one of hard, firm, or soft. As was stated, the difference between them are distinguished the best by considering the consequence of a deadline miss or a rejection. These types of deadline can be combined with the periodicity of computations only in a limited manner: the hard deadline can be associated with only periodic computations, the soft deadline can be associated with only aperiodic computations, and the firm deadline can

be associated with both types of computations.

Finally, these computations have different restrictions on their activation requests. All hard critical periodic computations must exist in the system statically. Thus, they can neither be activated nor suspended dynamically. A firm or soft computations can be dynamically requested regardless of its periodicity. In other words, it is only the hard periodic computation that are assumed as always feasible; any other computations can be rejected in the target imprecise computation platform.

The imprecise computation platform receives all the activation and suspension requests in the coming order and generates precise or imprecise computations by consulting their meta data, if and only if the request does not jeopardize the feasibility of already accepted computations. The imprecise computation platform schedules all accepted computations for one processor in the system. It also schedules requests for resources from these computations. The resources handled by the imprecise computation platform are both hardware and logical resources. Although a processor can also be seen a resource shared among all computations, it is not included in the resources for clarity.

There is no assumption made on the type of the processor adopted in the platform. Since the imprecise computation platform assumes the existence of only one processor, the platform is obviously targeted for a uni-processor system. This assumption does not necessarily forbid the computations on a processor to communicate other computations on remote nodes. Our view of network is that it is only one kind of hardware resources. If egress and ingress links are available on a specific hardware platform, then a computation is allowed to use them to communicate with computations on other nodes. It is only that that the imprecise computation platform is not responsible for meeting the timing constraints of any messages that go over the network. Nevertheless, if computations related to the processing of messages are given appropriate timing constraints that can be expressed in the framework of this research and that the flight time of messages are bounded, it is still possible to compose a larger system by uniting the imprecise computation platforms.

3.3 Summary

This chapter provided the overview of the target imprecise computation platform. We described the design philosophy and the guidelines that led us through the research. The design of the imprecise computation platform is centered around three different performance criteria. It aims to resolve overload in a short period of time, to achieve high QoS of periodic computations, and to execute aperiodic computations before their desired response time. The system model assumed in this research has only one processing unit, and the imprecise computations supported here are limited to those with distinct optional parts. Moreover, development of a new imprecise computation model is required to improve the practicality of the platform. Furthermore, two scheduling algorithms for the imprecise computations must be designed to handle different types

of overload. In designing the scheduling algorithms, it must be taken into account that the operating system designed for this imprecise computation platform provides only the mechanisms needed for imprecise computation. The actual policy is left out to the application developers so that the system is flexible to changes.

CHAPTER 4

PRACTICAL IMPRECISE COMPUTATION MODEL

A 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 was pointed out in Chapter 2, the existing imprecise computation models lack the ability to properly incorporate compensation or recovery operations needed after the termination of an optional part to maintain the context of the computation in a safe stable state. Moreover, the existing models do not allow mixture of mandatory parts and optional parts in the way practical applications require.

This chapter presents an imprecise computation model that solves these problems. This computation model is referred to as the *practical imprecise computation model* in this dissertation, in order to distinguish it from the imprecise computations models that have been used in the literature. The basic imprecise computation model and its derivatives are referred to as the *classical imprecise computation models*.

4.1 The Wind-up Operation

Among all the shortcomings of the classical imprecise computation models, the most restrictive one is that no consideration is given on how practical imprecise computations can be terminated. The classical imprecise computation models have no place for compensation or recovery operations needed on 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 so that the data is meaningful to human operators, and to send these processed data to the other nodes. Suppose that this computation is based on the basic imprecise computation model, so

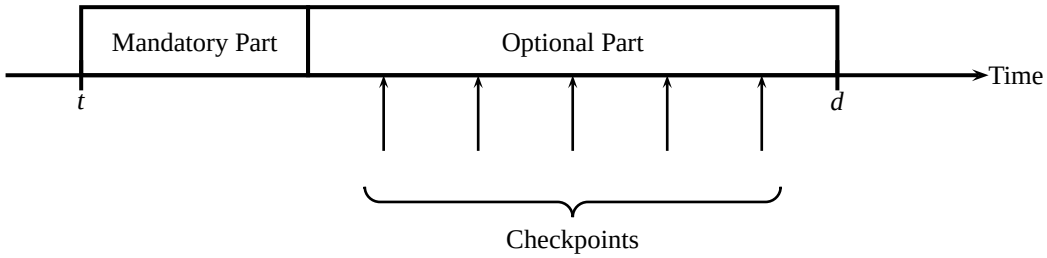


Figure 4.1: Inadequacy of milestone methods

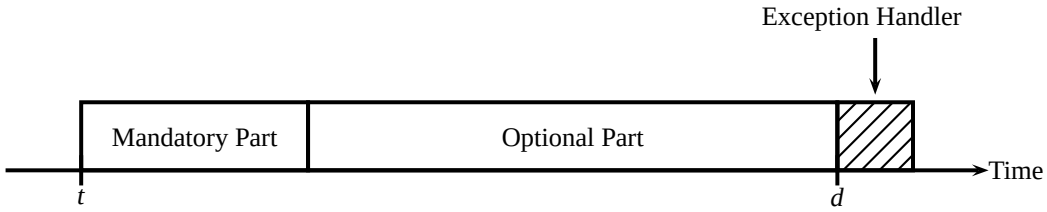


Figure 4.2: Infeasible schedule created by exception handling

that its mandatory part entirely precedes its optional part, and that the reward function of this computation is linearly monotone. Then, this computation should be executed as long as time permits to enhance the quality of the data sent. Also suppose that the imprecise computation has the highest priority in an interval $[t, d)$ where d is the absolute deadline of this computation, but that its optional part is too long to complete before d.

Figure 4.1 shows the case with the milestone method. The imprecise computation starts the execution of its mandatory part from time t. After it has completed its mandatory part, it goes on to execute its optional part. While the optional part is being executed, 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 final result after the computation is terminated at its deadline. Thus, when the size of the data becomes large, the overhead of a checkpoint can become so large that the performance of the computation itself is degraded. On the other hand, when the deadline is reached, thanks to the checkpoints, the best imprecise result is already available in the remote nodes. However, the terminated computation and the resources it 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, and in that case, it is problematic to leave the internal data in an inconsistent state. Therefore, the milestone method by itself is not enough to support correct imprecise computation if applications are based on the basic imprecise computation model.

The situation is little improved when an exception handling mechanism is implemented in the system. This mechanism may be superior to the checkpointing mech-

anism in that no overhead is associated with checkpoints. However, it can lead to a deadline miss in the worst case. Suppose the same scenario of the previous example, but that an exception handler is invoked at the deadline instead of regularly placing checkpoints. 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, the exception handler is only invoked on a deadline miss. This means that the operations performed by the exception handler violates 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 the destination nodes too late. Second, if the handler is executed at the highest priority, the subsequent computations are delayed by its execution. Thus, 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 classical imprecise computation models and practical real-time applications. Specifically, practical applications that allow termination in its optional part require compensation or recovery operations to allow safe termination, while the classical imprecise computation models allow no operation for that.

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

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

4.2 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 of the practical imprecise computation model that make the model different from the classical imprecise computation models are:

- one computation can have more than one mandatory parts and more than one optional parts; and

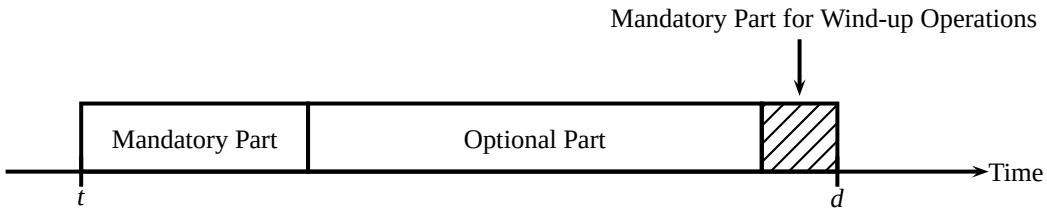


Figure 4.3: Feasible schedule for practical imprecise computation model

- the mandatory parts and the optional parts of an 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 as in Figure 4.3. The first mandatory part and the optional part is the same as those in the previous model, and the second mandatory part contains all the wind-up operations needed by this computation. The second part is scheduled before the deadline without termination, since its characteristics are the same as the mandatory part in the basic imprecise computation model, except that it can follow an optional part.

In this simple example, having another mandatory part for wind-up operations may seem equivalent to scheduling an exception handler before the deadline, by reserving the worst case execution time of the wind-up operations and putting that time aside so that the optional part can not consume that. However, a remarkable difference exits in more general cases. Using the practical imprecise computation model, the application programmers can place wind-up operations by putting mandatory parts after all optional parts that may be terminated without affecting the other parts of the system. Hence, a mandatory part that follows an optional part should only include the wind-up operations required for its preceding optional part. By comparison, when the exception handling mechanism is used, one handler must be programmed to include all wind-up operations, or the registered handler must be dynamically changed by the application, since the number of handlers that can be registered with the operating system is usually limited. However, including all the wind-up operations in one exception handler increases the overhead of compensation. Moreover, dynamic registration of an exception handler makes the timing analysis difficult and error prone.

An alternative to having multiple mandatory parts as in the practical imprecise computation model is to put the wind-up operations in the beginning of the mandatory part, so that it will be executed when its another instance is invoked in the next period. However, this approach is only applicable to a periodic imprecise computation and when the role of the wind-up operation is only to recover a consistent state. Obviously, the same approach does not work for an aperiodic imprecise computation, since there is no next mandatory part. Moreover, every wind-up operation is executed too late, if the wind-up operations have the same deadline as that of the original computation.

One of the advantages of having the wind-up operations in another mandatory part

is that the wind-up operation can be explicitly scheduled so that it is executed to meet the original timing constraints. This is significantly different from invoking exception handlers without a proper feasibility analysis. Moreover, there is no overhead associated with saving intermediate results as in the case of the milestone method, because the wind-up operations included in the mandatory part take care of returning the best result by themselves. Hence, this approach does not have any drawback in the checkpointing mechanisms nor that in the exception handling mechanisms. Furthermore, there is another advantage that the constructed application software is less dependent on special facilities provided by underlying operating systems. An application that contains the wind-up operations in its computation can be executed on an operating system that does not have ability to perform checkpoints nor to invoke an exception handler on timing faults. By contrast, an application that requires the checkpointing mechanism in an operating system cannot be executed on another operating system without the same mechanism.

One of the possible drawbacks of having the wind-up operations programmed within applications is that if each imprecise computation included its own wind-up operations, the size of the application can become larger. Even worse, more than one applications may have the same wind-up operation differently programmed within their mandatory parts. This will not only increase the cost of systems by requiring larger amount of memory, but also degrades the performance by lowering the cache hit rate. Our counterargument is that such code blow-ups can be avoided by putting the common cases together in a software library. The use of the library can evict most of the redundant codes, leaving only the special cases specifically programmed in each application. And it rarely happens that there are more than one applications that require the same special wind-up operations. Another possible drawback is that the scheduling of computations become more complicated. This dissertation shows that it is still possible to develop a practical scheduling algorithm by actually presenting two scheduling algorithms in Chapter 5 and Chapter 6.

Finally, this research regards a precise computation as only a special case of the practical imprecise computation model, since an imprecise computation without an optional part is the same as a precise computation. Moreover, the practical imprecise computation model does not force existence of mandatory part after every optional part, because a wind-up operation would be of little use if a computation has an optional part with a 0/1 constraint and the computation is only known to run on an operating system that can either allocate enough processing time for the optional part or no time at all. This allows application developers to optimize the computations according to the specific scheduling algorithm used in a specific system.

4.3 Linear Task Model

The term *computation* has been used up to this point to broadly refer to any activities. Hereafter, this dissertation uses the term *task* to refer to computations based on the

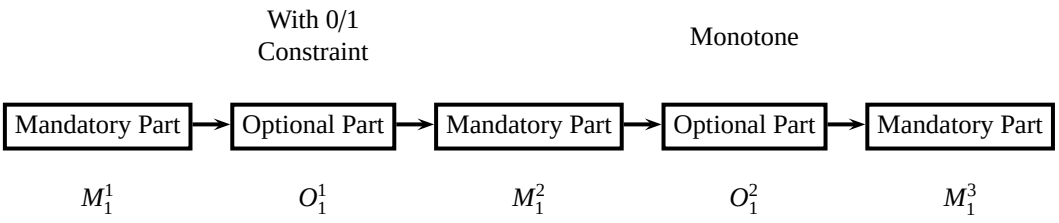


Figure 4.4: Linear task

practical imprecise computation model. In addition, if it is necessary to distinguish a task with optional part from that does not, a task with at least one optional part is called *imprecise task*, while a task without an optional part is called *precise task*.

All tasks are expressed by a linear task model. A linear task is one that 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 optional part that has more than one successors that can be selectively executed. An example of a linear task is shown in Figure 4.4. This task has two optional parts. One with 0/1 constraint follows the first mandatory part and the other that is monotone follows the second mandatory part. The linear task model forbids the case where one part and its successor are of the same type. In that case, there is no need to separately consider the two parts, since the task model divides the original computation only logically.

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 linearity constraint, however, 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.

For a linear task τ_i, we denote the number of mandatory parts and optional parts as n_i^M and n_i^O, respectively. The mandatory part that is executed j-th from the beginning is denoted as M_i^j. Similarly, the optional part that is executed j-th from the beginning is denoted as O_i^j. For example, if the task shown in Figure 4.4 is named as τ_1, then $n_1^M = 3$ and $n_1^O = 2$. Moreover, the optional part with the 0/1 constraint is denoted as O_1^1 and the one that is monotone is denoted as O_1^2.

The time attributes and timing constraints of a task τ_i are characterized by the parameters summarized in Table 4.1. The descriptions of the parameters are provided in the same table. A period is only associated with a periodic task and is larger than or equal to its relative deadline. If the task τ_i does not have any optional parts, n_i^O and all o_i^j are considered as zero. The term *desired response time* is used for an aperiodic task instead of the term relative deadline to avoid any confusion, because a scheduling algorithm is allowed to set a deadline different from the desired response time for soft aperiodic tasks.

The task τ_i is assumed to have a reward function $f_i(t)$ that takes the amount of computation time given to its optional parts as input and returns the total reward accrued.

Table 4.1: Time attributes and timing constraints of task τ_i

Parameter	Description
a_i	Latest activation time
T_i	Period
m_i^j	Worst case execution time of the j-th mandatory part M_i^j
o_i^j	Worst case execution time of the j-th optional part O_i^j
D_i	Relative deadline or desired response time
B_i	Upper bound on blocking time

The reward function is expressed in the form of a concave multi-segmented line where its j-th segment is denoted as φ_i^j. The task model does not restrict that each segment correspond to one optional part. However, it may be more natural to define a segment that corresponds to one or more optional parts than to define a segment with an arbitrary length. The reward function for the task τ_i is denoted as

$$f_i(\boldsymbol{t}_i) = \sum_{j=1}^{K} \frac{t_i^j}{L_i^j} q_i^j, \tag{4.1}$$

where $\boldsymbol{t}_i = \{t_i^1, t_i^2, \ldots, t_i^K\}$, K is the number of segments, L_i^j is the execution time needed to complete the j-th segment φ_i^j, and q_i^j is the maximal reward obtained by completing φ_i^j.

Since the multi-segmented line is concave, the following condition must always hold:

$$\frac{q_i^j}{L_i^j} > \frac{q_i^k}{L_i^k}, \tag{4.2}$$

only if $j < k$. The equality does not hold because then there is no need to consider the difference between them. In other words, if they are equal, they can be put together to form one single segment.

In addition to the parameters shown in Table 4.1, this dissertation uses the following notations for the sake of conciseness. Firstly, m_i is used to mean the sum of the worst case execution time of all the mandatory parts of the task τ_i. Thus,

$$m_i = \sum_{j=1}^{n_i^M} m_i^j. \tag{4.3}$$

Likewise, o_i is used to refer to the sum of the worst case execution time of all the optional parts of the task τ_i, which is given as follows:

$$o_i = \sum_{j=1}^{n_i^O} o_i^j. \tag{4.4}$$

Finally, C_i is used to mean the sum of the above. Thus,

$$C_i = m_i + o_i. \tag{4.5}$$

An instance or logical execution of a task is called a *job*. Every job has a release time and a deadline. The release time and absolute deadline assigned to a job of τ_i is denoted as r_i and d_i, respectively. It is assumed that a job arrives the system on its release time and only leaves the system on its deadline, unless it is explicitly deleted by the scheduler. The release time and deadline of a periodic job are predetermined by the period and the relative deadline. In particular, the j-th job from the activation time a_i has release time of $a_i + (j-1)T_i$ and deadline of $a_i + (j-1)T_i + D_i$. On the other hand, the release time and deadline of an aperiodic job is dynamically assigned by the scheduler based on the activation time and the desired response time, respectively. We denote the job of the task τ_i as J_i when the system time is in the interval $[r_i, d_i)$.

The difference between the task and its job is summarized in the following. It is a task that is activated or requested. If a task is active, then one or more of its jobs are released. It is the job to which the scheduler allocates computation time.

4.4 Conversion Method for Complex Computations

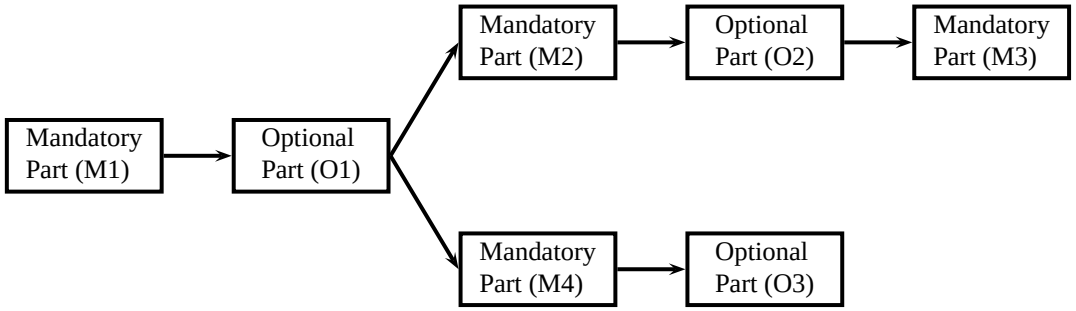
The structure of a task does not always have to correspond to the logical structure of the actual computation as long as the task model represents the execution requirement of computation correctly, since the reason to use the linear task model is to facilitate feasibility assessment. Thus, the represented requirement should always be no less than the actual amount. With this in mind, this research proposes that a computation that does not have a linear structure be converted to fit the linear task model by the following steps.

1. Pick a branching point whose successors are all in linear forms. Let these series of parts in the linear form be called *linear branches*.

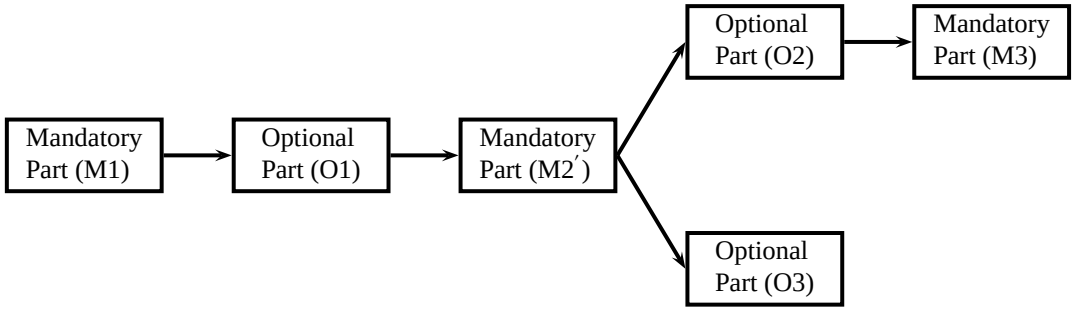
2. Remove the parts nearest to the branching point from the linear branches and merge them to create a single part of the same type. Attach this new part to the branching point as its successor. Let timing characteristics of this new part, and its reward function if it is an optional part, be the same as those of the one with the largest worst case execution time among all the merged parts.

3. Repeat the above step for the subsequent parts of the branches until all branches are merged. If there is no more part left in a branch in this process, stop merging from that branch.

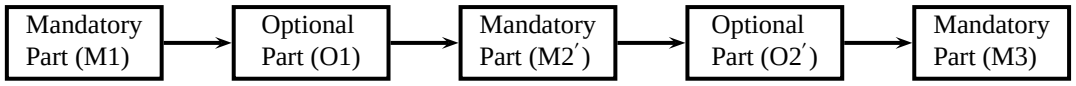
4. Repeat the above steps until there is no branching point in the whole structure of the computation.



(a) Original form



(b) Intermediate form (after M2 and M4 are merged)



(c) Linear form (after O2 and O3 are merged)

Figure 4.5: Conversion example

An example of the conversion is illustrated in Figure 4.5. The original form has one branching point. The first step is to pick the optional part O1 as the branching point, since its two successors are in a linear form. The next step is to merge these linear branches. The mandatory parts M2 and M4 are merged first and are replaced with another mandatory part M2′. Then the optional parts O2 and O4 are merged and replaced with another optional part O2′. At this point, there is no more parts to merge and the whole computation is in a linear form. Hence, the conversion is completed.

One drawback of this conversion is that worst case execution time and reward function are not of the original computation anymore. Thus, schedules created by scheduling algorithms that use these pieces of information to optimize the QoS only be suboptimal. However, this issue is not further considered in this dissertation, since the estimation of precise reward function is not the scope of this research.

4.5 Uniform Scheduling Interface

We now present the design of a uniform scheduling interface between the scheduler and application tasks. The primary objective of defining a uniform scheduling interface is to allow a specific implementation of the imprecise computation platform to switch scheduling algorithms off-line without incurring any changes to applications. It is not intended to enhance the portability of an application, though it may enhance the portability of an application between different implementations of the imprecise computation platform. Moreover, for the same reason, names of the primitives that consist the interface does not necessary have to coincide with that of actual primitives in specific implementations of the imprecise computation platform.

The fundamental question here is why does the scheduler and application tasks ever need to interact. The need for the interaction arises from the fact that not all the scheduling events are detectable solely by the scheduler. This is due to the other guidelines. In order not to increase the number of scheduling entities, the linear task model used in this research represents one computation by only one task. Moreover, the imprecise computation platform does not assume that there is special meta data given to indicate which part of the program belongs to a mandatory part or an optional part. Therefore, the scheduler does not know when tasks are executing their mandatory part and when they are executing their optional part. However, this identification is important in scheduling imprecise tasks, since the scheduler of the imprecise computation platform has to terminate optional parts under overload to control the level of load while it must not terminate any mandatory part. On the other hand, tasks need to know the amount of time that can be allocated to their optional part if they aim to dynamically adjust their behavior. For example, a task may not want to execution its optional part with a 0/1 constraint at all when there is not enough time to complete that. However, this information is only available to the scheduler.

The uniform scheduling interface consists of the following primitives.

- Identification of optional parts:

  Two primitives identify the beginning and end of each optional part. Although the complementary primitives that identify the beginning and end of each mandatory part work just in the same manner, it is the optional part that is identified in this research to support precise tasks efficiently. Thus, the following primitives are defined:

  - `begin_optional` to indicate the beginning of an optional part; and
  - `end_optional` to indicate the end of an optional part.

  It must be noted that these primitives do not restrict an imprecise computation to start execution from a mandatory part.

- Termination Check:

The following primitive checks whether the preceding optional part was terminated:

- `is_terminated` to return `true` if the preceding optional part was terminated.

This primitive allows application programmers to skip wind-up operations that are not required if its preceding optional part has been executed to completion.

- Scheduling of jobs:

  The following primitives related to scheduling requests change the state of a job:

  - `activate` to activate a task to release a new job;
  - `end_job` to indicate completion of a job;
  - `suspend` to suspend a task to stop releasing a new job;

  These primitives provide the minimal facility required for scheduling of real-time jobs. Developers of real-time operating systems are encouraged to add other primitives like sleep and wake-up, even if they cannot be used for scheduling real-time jobs.

- Dynamic reservation of computation time and its cancellation:

  Since scheduling algorithms may not allocate a fixed amount of computation time to optional parts of each job, the following four primitives are defined to enable an imprecise task to dynamically reserve an amount of computation time for its optional parts and to cancel the reservation:

  - `reserve_time` to reserve an amount of computation time that can be allocated to the requesting job;
  - `reserve_all_time` to reserve all the computation time that can be allocated to the requesting job at the time of the request;
  - `fix_reservation` to adjust the amount of computation time reserved; and
  - `cancel_all_reservation` to free all the previously reserved computation time.

  The amount of computation time that can be reserved by a job should be no more than that allocated by the scheduler. There is no primitive provided for a job to request the scheduler to increase the upper limit on the computation time allocated to the optional parts. This decision is left to the scheduling algorithm deployed, because a job is not capable of observing the whole system state and thus it cannot decide appropriately which job should be given more computation time. The cancellation of

reservation is important on the imprecise computation platform, since the dynamic reservation must be made based on the worst case execution time. In other words, it is very likely that some amount of reserved time is left unused when the operation which the time was reserved for is completed. Canceling the excessive amount of reservation enables other jobs to claim this time and thus increases the effective processor utilization.

- Acquisition and release of resources:

  The imprecise computation platform also defines primitives for acquiring and releasing resources, because a scheduling algorithm and a resource access protocol are closely related. The following primitives are defined:

  - `acquire_resource` to acquire an indispensable resource required for further execution;
  - `try_acquire_resource` to acquire a supplementary resource desired but not required for further execution; and
  - `release_resource` to release a previously acquired resource.

4.6 Definitions

We here define terms that are used differently in the literature or can have more than one meanings if the practical imprecise computation model is deployed.

First of all, an amount of time is defined in two terms with respect to a job.

Definition 1. *The* execution time *of a job is the time that is actually consumed before its completion.*

Definition 2. *The* computation time *of a job is the time that is allocated to it.*

The meanings of these terms are further limited by adding *mandatory* or *optional* before them. For example, mandatory execution time refers to the time actually consumed by all mandatory parts of a job, and optional computation time refers to the time allocated solely to optional parts of a job. If neither *mandatory* nor *optional* precedes the terms, their definitions do not consider the difference. For example, by the term execution time, with no preceding *mandatory* or *optional*, it simply refers to the total time consumed by all mandatory parts and optional parts of a job. This discrimination applies to all other terms used in the literature where appropriate, including the notions of loading factor, processor utilization, processor bandwidth, and processor demand.

If a job is ready, it means that the job may either execute its mandatory part or optional part. Thus, this dissertation uses the following expressions to discriminate the two cases.

Definition 3. *A job is* ready for mandatory part *if it is ready and will execute its mandatory part if the processor is allocated.*

Definition 4. *A job is* ready for optional part *if it is ready and will execute its optional part if the processor is allocated.*

The terms slack and idle time are also redefined to avoid ambiguity that arises in the context of scheduling imprecise tasks.

Definition 5. *The* slack *in the system is the time for which an arbitrary mandatory part can be delayed without making the schedule infeasible.*

Definition 6. *The* idle time *in the system is the time left in the schedule after all mandatory parts are scheduled.*

Since this research does not assume that the given worst case execution time and the actual execution time are equal, the processor utilization is discriminated strictly by defining the following two terms.

Definition 7. *The* nominal processor utilization *or simply the* processor utilization *is defined for a group of periodic task* Γ_p *as:*

$$U = \sum_{\tau_i \in \Gamma_p} \frac{C_i}{T_i}. \tag{4.6}$$

Definition 8. *The* effective processor utilization *is the ratio of an amount of time consumed by jobs to the length of the interval in concern.*

Finally, we define two types of overloaded conditions. Since this research assumes a uni-processor system, the optimal scheduling algorithm in the following definitions can be replace by the EDF algorithm, which is optimal in the sense that it always creates a feasible schedule if the given task set is feasible.

Definition 9. *The* transient overload *is said to occur when all periodic tasks in the given task set can be feasibly scheduled by an optimal scheduling algorithm when their execution time and given worst case execution time are equal, but cannot be scheduled with dynamically activated aperiodic tasks.*

Definition 10. *The* persistent overload *is said to occur when all activated periodic tasks in the given task set cannot be feasibly scheduled by an optimal scheduling algorithm when their execution time and given worst case execution time are equal.*

4.7 Summary

This chapter presented a practical imprecise computation model. Its characteristics are summarized as follows.

- A task based on the practical imprecise computation model can have any number of mandatory parts and optional parts. A task may have only one mandatory part and one optional part as in the classical imprecise computation models, but it may also have two or more mandatory parts and optional parts. Moreover, it need not have either a mandatory part nor an optional part.
- The mandatory and optional parts of a task must be executed in an interleaving manner.
- A mandatory part and an optional part have the same characteristics and constraints as those in the classical imprecise computation models, except that a mandatory part may folow an optional part. A mandatory part must always be executed to completion, while an optional part can be skipped or prematurely terminated.
- An optional part can have a 0/1 constraint or be monotone.

Practical imprecise computations are expressed using the linear task model so that the feasibility analysis can be performed in a practical overhead. For computations that do not have a linear structure, a conversion method is also presented. The drawback of the conversion may be that a reward function must be supplied to reflect the characteristics of the converted computation. However, it is not the focus of this research to correctly determine characteristics of applications, and the following arguments assumes that all computations used in this research fit the linear task model.

The downside of the practical imprecise computation model is that existing scheduling algorithms that assume just one mandatory part and one optional part with a precedence constraint cannot be used to schedule linear imprecise tasks. Moreover, a scheduling algorithm that can handle the imprecise tasks may become more complicated due to more than one precedence constraints between mandatory parts and optional parts.

The imprecise computation platform defines a uniform scheduling interface to enable switching scheduling algorithms without accompanying any change in applications. The interface consists of primitives required for identification of optional parts, scheduling of jobs, dynamic reservation of computation time and its cancellation, and acquisition and release of resources.

The definitions of terms used in the following chapters were also given. These definitions were necessary to evict ambiguity from the description of the developed scheduling algorithms.

CHAPTER 5

SCHEDULING UNDER TRANSIENT OVERLOAD

The primary focus under transient overload is at scheduling aperiodic tasks, since the transient overload in this research is triggered by activation of aperiodic tasks. In other words, it is assumed that the activation of periodic tasks seldom puts the system into overload, although the system targeted in this research also allows activation and suspension of periodic tasks.

The objective of scheduling under transient overload is to resolve the overload in as much short time as possible. The transient overload, by definition, is resolved when the aperiodic jobs are completed. An important point to consider is that the scheduling algorithm for transient overload should also minimize the number of rejected activation requests for aperiodic tasks. Otherwise, it leaves a kind of paradox, similar to the one pointed out by Baruah [44], that a scheduling algorithm that denies all the aperiodic requests achieves the shortest overloaded interval. Thus, the objective includes the maximization of the acceptance ratio of activated aperiodic tasks that can be executed before their desired response time in addition to resolving the transient overload in a short period of time.

This chapter begins with describing the basic strategy for scheduling under transient overload and the flow of handling arriving requests in the system. Then, the *Mandatory First with Wind-up Part* (M-FWP) scheduling algorithm developed for transient overload is described and compared with other related scheduling algorithms.

5.1 Basic Strategy

The motivation for allowing transient overload is to increase the average effective processor utilization. Hence, the M-FWP algorithm follows the EDF scheduling strategy to fully utilizing the processor. The EDF scheduling strategy is also simple and easy to implement. By contrast, the RM algorithm has lower run-time overhead than the EDF algorithm, but it cannot fully utilize the processor in the general cases.

Table 5.1: Task set A

Task	a_i	n_i^M	n_i^O	m_i^1	o_i^1	D_i	T_i
τ_1	0	1	1	2	2	8	8
τ_2	1	1	1	2	2	9	9
τ_3	5	1	0	2	0	5	-

The M-FWP scheduling algorithm schedules all jobs preemptively to assess the feasibility of given task sets on-line at a practical cost. By comparison, if jobs are scheduled non-preemptively, the scheduling algorithm must place idle time in the schedule to avoid priority inversion. Since such scheduling problems are NP-hard in the strong sense, the non-preemptive scheduling approach is not practical.

Before going any further, it must be reminded that a real-time scheduling algorithm should ensure real-time execution of hard real-time tasks and that of firm tasks once it has accepted them. For this, the worst case execution time of the mandatory parts of active periodic tasks are reserved. On the other hand, in order to increase the effective processor utilization, the execution time of optional part and that of aperiodic tasks are not reserved. They are executed in the idle time of the schedule.

We can now set a strategy to achieve the objective of minimizing the length of overloaded intervals. The strategy is to execute all ready mandatory parts prior to all ready optional parts, and to schedule any aperiodic jobs by terminating or discarding optional parts. This is similar to the mandatory first strategy used to achieve competitive performance in the literature [44]. The difference is that, in the linear task model, an optional part can also precede a mandatory part. Hence, the mandatory first strategy in the M-FWP algorithm means that if a job is ready for mandatory part, it is executed prior to any jobs ready for optional part at the same moment. This strategy is enforced irrespective of whether jobs are periodic or aperiodic. Thus, a mandatory part of an aperiodic job is scheduled before all optional parts of any jobs. We note that it is indeed desirable to allocate execution time with preference to aperiodic jobs than to optional parts, since the transient overload can be resolved by completing aperiodic jobs.

The reason why the mandatory first strategy works in favor of shortening overloaded interval can be seen by comparing the schedules in Figure 5.1 and Figure 5.2. Each figure contains a schedule created for the task set shown in Table 5.1. The task set only contains two periodic tasks τ_1 and τ_2 and one aperiodic task τ_3. The total utilization of the periodic tasks are one, and thus only the periodic tasks can be feasibly scheduled by the EDF algorithm without terminating optional parts. In other words, activation of the aperiodic task puts the system into transient overload.

Figure 5.1 shows a schedule created under the EDF strategy. Since the job J_2 has the earliest deadline at five, the aperiodic job J_3 cannot be executed until the completion of J_2. Consequently, J_3 only completes at ten. This completion time can be made as early as eight, if the entire optional part of the task τ_2 can be discarded. By comparison, in Figure 5.2, which shows a schedule created under the EDF with the mandatory first

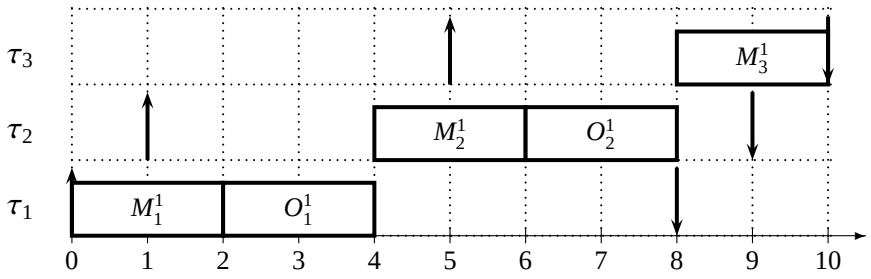


Figure 5.1: Schedule created under EDF

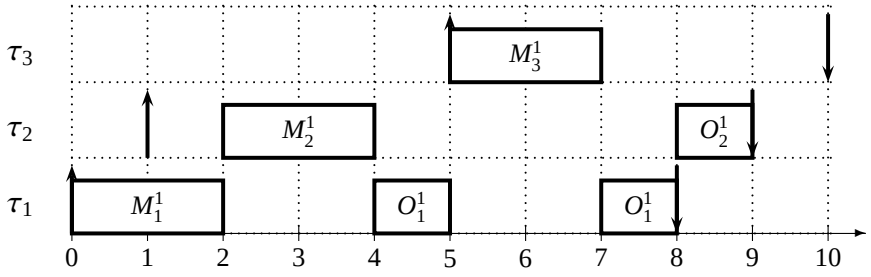


Figure 5.2: Schedule created under EDF with mandatory first strategy

strategy, the aperiodic job J_3 is able to complete at seven. In other words, the overload is resolved at seven, by discarding a half of O_2^1. This completion time is earlier than eight achieved in the case of EDF without the mandatory first strategy.

5.2 Request Handling Flow

Figure 5.3 shows how incoming requests are handled. The entities concerned with the handling of requests are the *admission controller* and the *scheduler*.

The admission controller checks the feasibility of the new task set when activation requests of firm periodic or aperiodic tasks arrive the system. The activation of a firm aperiodic task is rejected if its job cannot be completed by its desired response time. When an activation request for a firm periodic task cannot be accepted, there are two cases for this. Either it can not be feasibly scheduled at the requested moment due to active aperiodic tasks or it can not be feasibly scheduled since the processor utilization of all mandatory parts of periodic tasks becomes too high by this activation. In the first case, the M-FWP algorithm postpones the actual release of the first job until the transient overload is resolved, and in the second case, it rejects the request. Note that the postponed requests can also be withdrawn before it is actually accepted. In this research, we assert that a soft aperiodic task must always be completed. In other words, soft aperiodic tasks need to be executed to completion even if they can be completed only in an interval longer than its desired response time.

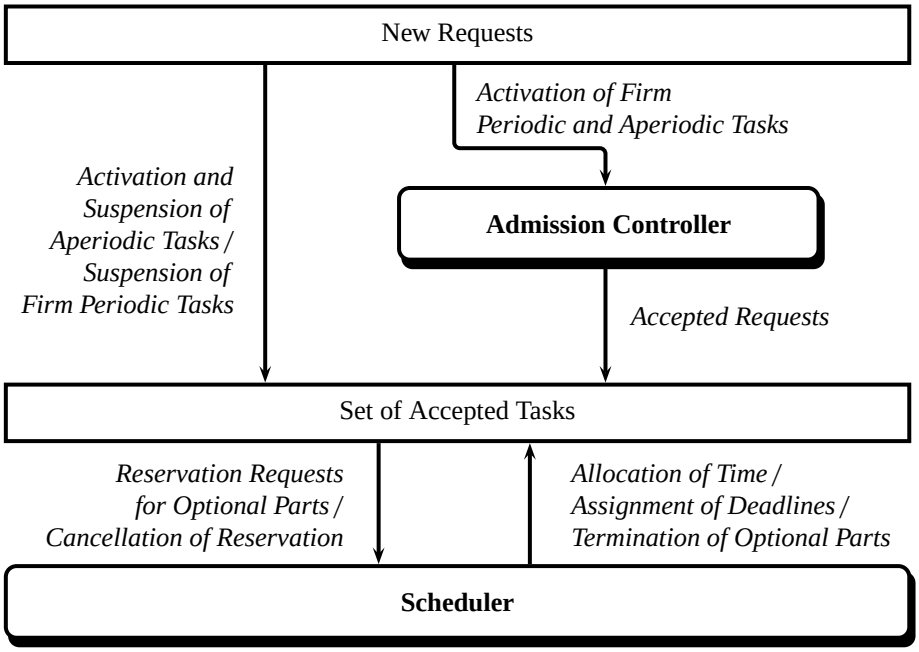


Figure 5.3: Request handling flow under transient overload

All jobs in the system are scheduled by the scheduler. The roles of the scheduler are to dynamically allocate computation time to these jobs, to assign an appropriate deadline to soft aperiodic jobs, and to terminate optional parts when the system is under overload. Furthermore, the scheduler handles the requests from jobs to reserve computation time for their own optional parts and to cancel the reservation.

5.3 The Mandatory-First with Wind-up Part Scheduling Algorithm

The M-FWP algorithm reserves all the necessary time for executing mandatory parts of active periodic tasks but does not reserve any computation time for optional parts nor for aperiodic tasks. Thus, the optional parts of periodic jobs and all aperiodic jobs need otherwise idle time in the system to execute. This, together with the mandatory first strategy, leads to a static priority assignment over EDF scheduling.

To enforce the mandatory first strategy, the M-FWP algorithm assumes that ready jobs are maintained by three ready queues shown in Figure 5.4. Each queue holds jobs in the EDF order. These ready queues are called periodic mandatory ready queue (PMQ), aperiodic mandatory ready queue (AMQ), and optional ready queue (OQ). The PMQ holds periodic jobs that are ready for mandatory part, and similarly the AMQ holds aperiodic jobs that are ready for mandatory part. Ties in the PMQ and AMQ are broken in favor of shorter relative deadline. The OQ holds all jobs that are ready for optional part and ties in the OQ are broken in the FIFO order. Thus, if there are two jobs with

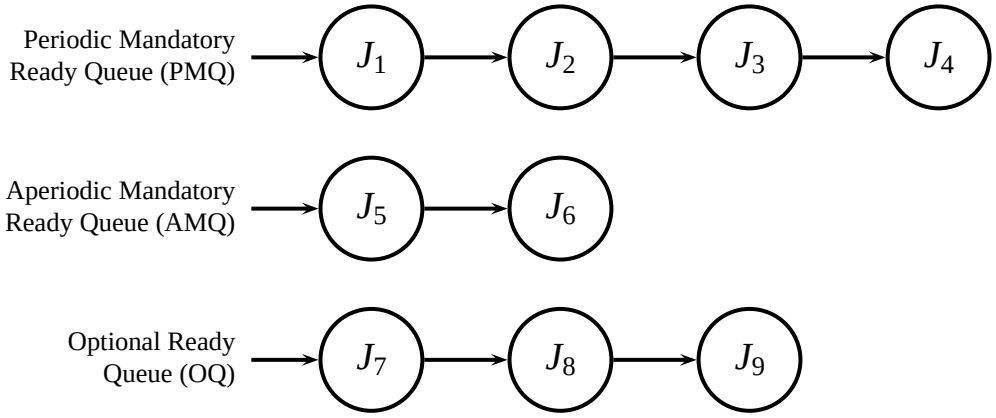


Figure 5.4: Three ready queues for maintaining ready jobs

the same deadline, the job already in the OQ has higher priority than the one that will be newly put to the OQ. These queues are searched in a fixed order for a ready job. The PMQ is searched first, then the AMQ, and then the OQ. Thus, an aperiodic job can only be given the processor when no periodic job is ready for mandatory part, and a job ready for optional part can only be given the processor when neither periodic nor aperiodic job is ready for mandatory part. Consequently, a job in Figure 5.4 is considered eligible in the increasing order of their subscript. This order, which determines the priority of jobs, however, is not fixed and changes, for example, when a job finishes a mandatory part and becomes ready for optional part.

Resource sharing under M-FWP scheduling is accomplished by temporarily making the running job non-preemptive. Specifically, the M-FWP scheduler does not allow preemption while a job is accessing a resource.

In the following description of the M-FWP algorithm, it is assumed that the run-time scheduler is capable of maintaining the following pieces of information:

- $\ell_i^m(t)$: the execution time of the job J_i required at time t to complete all the remaining portions of its mandatory parts;
- $R_i(t)$: the computation time that is allocated to the job J_i at time t;
- $S_i(t)$: the computation time that can be used for the optional parts of the job J_i at and after time t; and
- $X_i(t)$: the computation time that is dynamically reserved by the job J_i for its optional part.

Note that the reserved optional computation time is only consumed in the optional part before any amount of allocated idle time is consumed. Moreover, under a correct scheduling algorithm, these variables always satisfy the following condition:

$$S_i(t) = R_i(t) - \ell_i^m(t). \tag{5.1}$$

5.3.1 Scheduling a Set of Static Tasks

First, we describe the M-FWP algorithm for a set of static tasks where neither activation nor suspension of a task occurs.

The scheduling events in this static system are:

- release of a new job,
- completion of a mandatory part,
- completion of an optional part,
- overrun of an optional part,
- arrival of a reservation request for an optional part, and
- arrival of a cancellation request for previously reserved computation time.

Among these, the completion of a mandatory or optional part includes completion of a job if there is no more part left to execute. Moreover, the last five events conform to the uniform scheduling interface.

In the following, operations taken for each of these events are described.

Release of a New Job When a new job is released, the M-FWP scheduler allocates an amount of computation time depending on whether the job begins from a mandatory part or an optional part. If the job begins from a mandatory part, the scheduler allocates all the computation time reserved for it. Thus, for the newly released job J_i, $R_i(t)$ is set to m_i. On the other hand, if the job begins from an optional part, the scheduler allocates an amount of idle time to the job as optional computation time. It must be noted that, under the M-FWP algorithm, the idle time is allocated dynamically at every time of a job requests execution of its optional part.

Following the definition of idle time, the M-FWP algorithm calculates the amount of idle time by considering all possible execution of mandatory parts in the interval $[r_i, d_i)$. Specifically, the amount of idle time allocated to the job J_i is calculated by considering all jobs that have higher priority and that may have higher priority in the future before the job completes its optional part.

The M-FWP algorithm defines the following groups with respect to the job J_i that requests an amount of idle time.

- $\Gamma_h(J_i)$: a group of all ready jobs that have higher priority than J_i.
- $\Gamma_{hp}(J_i)$: a group of periodic jobs that satisfy the following condition:

$$r_j + T_j < d_i. \tag{5.2}$$

- $\Gamma_{hpe}(J_i)$: a group of periodic jobs J_j in $\Gamma_{hp}(J_i)$ that satisfy the following condition:

$$((d_i - r_j) \bmod T_j) < D_j. \tag{5.3}$$

Then, the amount of idle time allocated to the optional part of job J_i is estimated as

$$S_i(t) = \min\{d_i - t - \ell_i^m(t) - E_i(t) - F_i(t) - \min\{G_i(t), H_i(t)\}, S_i^n(t) - X_i^n(t)\} \tag{5.4}$$

where

$$E_i(t) = \sum_{J_j \in \Gamma_h(J_i)} R_j(t), \tag{5.5}$$

$$F_i(t) = \sum_{J_k \in \Gamma_{hp}(J_i)} \left(1 + \left\lfloor \frac{d_i - (r_k + T_k) - D_k}{T_k} \right\rfloor\right) m_i, \tag{5.6}$$

$$G_i(t) = \sum_{J_\ell \in \Gamma_{hpe}(J_i)} \min\{m_i, (d_i - r_\ell) \bmod T_\ell\}, \tag{5.7}$$

$$H_i(t) = \max\left\{(d_i - r_\ell) \bmod T_\ell \mid J_\ell \in \Gamma_{hpe}(J_i)\right\}, \tag{5.8}$$

and $S_i^n(t)$ and $X_i^n(t)$ are the remaining optional computation time of the next job in the OQ at time t and the amount of its optional computation time reserved, respectively. In other words, the job J_i^n has the highest priority among jobs that have lower priority than J_i. Moreover, the above equations assume $\max\{\emptyset\} = \min\{\emptyset\} = 0$.

The rationale behind Equation (5.4) is as follows. First, it is clear that, by neglecting the interference from all jobs of other active tasks in the system, the upper bound on the amount of idle time the job J_i can spend for executing its optional part before its deadline is $d_i - t - \ell_i^m(t)$. Then, from this upper bound, the amount of interference that J_i suffers from other jobs are subtracted. The function $E_i(t)$ gives the sum of the remaining computation time allocated to ready jobs with priority higher than J_i. The functions $F_i(t)$, $G_i(t)$, and $H_i(t)$ together give the upper bound on the sum of the mandatory execution time of jobs that are released later than r_i but may have higher priority than J_i. The function $F_i(t)$ accounts for the interference from these jobs with deadline before d_i. The interference from jobs with release time before d_i but deadline after d_i is estimated as $\min\{G_i(t), H_i(t)\}$. The function $G_i(t)$ estimates the total of mandatory execution time that may be claimed by J_ℓ in the interval $[r_\ell + T_\ell, d_i)$, whereas the function $H_i(t)$ provides its upper bound. The upper bound $H_i(t)$ must be considered, because the function $G_i(t)$ provides only an estimation. Finally, if there is a job J_i^n that has just lower priority than J_i and the result of this calculation is larger than the amount of the remaining optional computation time allocated to J_i^n, the final optional computation time allocated to J_i must be decreased to $S_i^n(t)$. Otherwise, the job J_i^n can miss its deadline, since J_i^n can only tolerate $S_i^n(t)$ units of interference while it is in the OQ.

There are two things that must be noted for the estimation of the idle time.

First, the reason why the job J_i^n with later deadline can only be allocated smaller amount of idle time than the job J_i is that the amount of idle time is not precisely

calculated by Equation (5.4). The estimation in the calculation of the amount of idle time is inevitable because its precise calculation requires actually creating a schedule for a hyperperiod to find out whether the mandatory parts of jobs in $\Gamma_{hpe}(J_i)$ really needs to be executed before J_i completes its optional part. Unfortunately, examining a schedule of a hyperperiod length is of exponential complexity, and thus not practical for on-line scheduling.

Second, the calculation of idle time does not need to take into account the blocking time B_i, since a job is never delayed by requesting a resource once it has started to execute, owing to the M-FWP algorithm that achieves the mutual exclusion of accesses to shared resources by forbidding preemption while the running job is accessing a resource.

When the optional computation time for the newly release job J_i is calculated as non-zero, J_i is put to the OQ. Otherwise, its first optional part is discarded, and if there is a mandatory part after that, J_i is put to the PMQ. If, in the first case, there exists the job J_i^n in the OQ, the amount of the time allocated to the optional part of J_i^n is decreased, since Equation (5.4) does not take into account any execution of optional parts by jobs other than those in the OQ at the time the optional computation time was allocated. Thus, the scheduler subtracts $S_i(t)$ from $S_i^n(t)$ when putting the task τ_i to the OQ.

Completion of a Mandatory Part When a job J_i completes its mandatory part and becomes ready for optional part, the scheduler allocates optional computation time to J_i in a similar manner to that taken when a job that begins with an optional part is released. One of the differences is that reserved optional computation time is taken into account, since the M-FWP algorithm allows a job to dynamically reserve an amount of time for its optional part after its release. Another difference is that any unused portion of the time previously reserved for the completed mandatory part is also reclaimed and added to the optional computation time of J_i. This dissertation calls it *intra-task reclaiming*. The intra-task reclaiming under the M-FWP algorithm is accomplished by updating $\ell_i^m(t)$. Thus, the optional computation time of J_i is given by the following:

$$S_i(t) = X_i(t) + \min\{d_i - t - \ell_i^m(t) - X_i(t) - E_i(t) - F_i(t) - \min\{G_i(t), H_i(t)\}, S_i^n(t) - X_i^n(t)\}. \quad (5.9)$$

If $S_i(t) > 0$, then the job is removed from the PMQ and put to the OQ. Otherwise, no optional computation time can be allocated to J_i, and the optional part is discarded. Then, if J_i has another mandatory part to execute, the scheduler keeps the job in the PMQ as it was, and otherwise, the scheduler removes the job from the PMQ.

Completion of an Optional Part When a job J_i completes its optional part and becomes ready for mandatory part again, the scheduler moves the job from the OQ to the PMQ. If at this moment the job still holds unused idle time, then that idle time is passed to the job that is now at the head of the OQ. If there is no job left in the OQ, the unused idle

time is freed so that it can be claimed afterward when a job becomes ready for optional part.

On the other hand, if the job J_i is finished with the completion of the optional part, the scheduler removes J_i from the OQ and allows other ready jobs in the OQ to reclaim the unused time in the same manner as above.

Overrun of an Optional Part When a job ready for optional part has consumed all the optional computation time but still not completed its optional part, an overrun is said to occur. When a job overruns, the scheduler terminates the optional part of the overrunning job and removes the job from the OQ. Then, if the job has a mandatory part to execute after the terminated optional part, the scheduler puts the job to the PMQ.

Reservation of Time for an Optional Part When a job J_i requests reservation for its optional part, the scheduler checks whether the requested amount of time can be allocated to the optional part of J_i.

If J_i requests reservation of optional computation time for δ in a mandatory part, the scheduler checks if the following condition holds:

$$\delta \leq \min\left\{d_i - t - \ell_i^m(t) - X_i(t) - E_i(t) - F_i(t) - \min\{G_i(t), H_i(t)\}, S_i^n(t) - X_i^n(t)\right\}. \quad (5.10)$$

If the condition holds, the request is accepted and the reservation is made by increasing $X_i(t)$ by the requested amount. Otherwise the request is rejected. Note that $S_i(t)$ must be recalculated afterward when the mandatory part is completed, since the amount of available idle time may have become different from the time the reservation was made in the mandatory part.

If the reservation was requested in an optional part, the scheduler checks if the following condition holds:

$$\delta \leq S_i(t) - X_i(t). \quad (5.11)$$

If the condition holds, $X_i(t)$ is increased by the requested amount δ. Otherwise, the request is rejected.

Cancellation of Time Reservation When cancellation of all previously made reservation is requested by a job J_i, the M-FWP scheduler sets $X_i(t)$ to zero without an admission control. On the other hand, when only partial cancellation of reservation is requested by J_i so that the amount of computation time that remains reserved is $\hat{\delta}$, the scheduler checks the following condition:

$$\hat{\delta} \leq X_i(t). \quad (5.12)$$

If the condition holds, then the request is accepted and the amount of reserved time is decreased to $\hat{\delta}$. Otherwise the request is rejected.

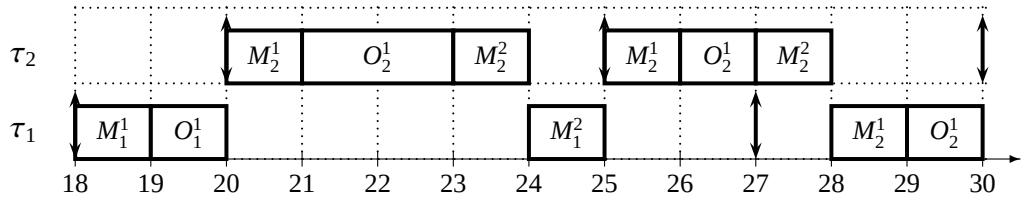


Figure 5.5: M-FWP Schedule of two static periodic tasks in Table 5.2

Table 5.2: Task set B

Task	a_i	n_i^M	n_i^O	m_i^1	m_i^2	o_i^1	D_i	T_i
τ_1	0	2	1	1	1	2	9	9
τ_2	0	2	1	1	1	2	5	5

In both cases of cancellation requests, no other operation is needed, since this is enough to allow other jobs to claim the idle time once reserved for J_i.

An example of M-FWP schedule is shown in Figure 5.5. In the example, two static periodic tasks τ_1 and τ_2 are scheduled in the interval $[18, 30]$. The attributes of these tasks are summarized in Table 5.2.

At 18, τ_1 releases a new job J_1, and it starts executing its first mandatory part, since the latest job of τ_2 has finished before 18 and its next job is not released yet. At 19, the job J_1 completes its first mandatory part. At this time, the scheduler calculates the optional computation time for its optional part O_1^1. The amount of idle time calculated by Equation (5.4) is three. Similarly, when the job J_2 completes its first mandatory part at 21, the scheduler calculates the amount of idle time for O_2^1. In this second case, the first argument of the minimum operator in Equation (5.4) gives three. However, the job J_1 is still in the OQ. Moreover, J_1 has a later deadline than J_2 and has remaining optional computation time of two. Thus, the second argument of the minimum operator becomes two, and thus the amount of time allocated to J_2 is decreased to two. Hence, the scheduler sets $S_2(t)$ to two and decreases the amount of idle time allocated to J_1 to zero.

Finally, when a static task set is concerned, a task set is feasible if and only if the mandatory parts of all tasks can be feasibly scheduled by the EDF algorithm. The feasibility of the task set under the EDF algorithm when relative deadline can be smaller than period is found in [74]. Based on that analysis, the correctness of the M-FWP algorithm for the static task set is stated in the following theorem.

Theorem 1. *A set of static periodic tasks is feasibly scheduled by the M-FWP scheduling algorithm if and only if all mandatory parts of the tasks are feasibly scheduled by the EDF scheduling algorithm.*

Proof. It is sufficient to show that the amount of idle time calculated by Equation (5.4) is no larger than the amount of idle time that actually exists in a feasible schedule, because

if no idle time is available the schedule created by the M-FWP algorithm is equivalent to that by the EDF algorithm.

Suppose that in any interval $[t_1, t_2)$, the sum of computation time allocated to all jobs that have release time no earlier than t_1 and deadline no later than t_2 is less than $t_2 - t_1$. Then, the sum of allocated time can only increase by allocating optional computation time to a job J_i that newly requests execution of its optional part. In particular, only two cases need to be considered: when there is no ready job in the OQ or when the deadline of J_i is the latest among all jobs in the OQ at the time the job J_i claims idle time.

Assume that a deadline miss occurs at time $\hat{t}$ by allocation of idle time $S_i(t)$ to the job J_i at time t. Thus, $t < \hat{t}$. Moreover, assume that the deadline miss occurs even if no other jobs execute their optional parts at all after time t. Thus, we assume that all other optional parts are discarded, without loss of generality. Then, in the interval $[t, d_i)$, the priority of J_i remains the lowest. Consequently, the execution of the optional part by J_i is delayed until all other jobs complete their execution and it must be J_i that misses the deadline, because, by assumption, there was no deadline miss without the allocation of $S_i(t)$. Thus, $\hat{t}$ and d_i are the same time.

Since a deadline miss is caused by an overflow in $[t, d_i)$, the following must hold:

$$d_i - t < Y, \tag{5.13}$$

where Y is the processor demand of all jobs in $[t, d_i)$.

On the other hand, since the optional computation time allocated to J_i is larger than zero, the following must hold:

$$d_i - t > \ell_i^m(t) + X_i(t) + S_i(t) + E_i(t) + F_i(t) + \min\{G_i(t), H_i(t)\}. \tag{5.14}$$

In the above equation, the upper bound on the total processor demand of all jobs in $[t, d_i)$ appears on the right side. Thus,

$$d_i - t > Y. \tag{5.15}$$

This contradicts with Equation (5.13). Hence the theorem follows. □

5.3.2 Dynamic Activation of Aperiodic Tasks

When activation of a firm aperiodic task τ_ν is requested, the admission controller tests if scheduling its job J_ν with deadline $a_\nu + D_\nu$ does not cause a deadline miss. The test is, however, postponed until the completion of resource access, if the job running at this arrival time was accessing a resource, since it is not possible to preempt the running job under the M-FWP algorithm.

Since an aperiodic job cannot delay any periodic job ready for mandatory part, the admission controller only has to test whether any aperiodic jobs with later deadline and any periodic jobs ready for optional part miss their deadlines. This can be tested either directly by considering the interference of this aperiodic job onto other ready jobs, or

indirectly by calculating the amount of idle time that can be allocated to J_ν. The M-FWP algorithm chooses the latter, since the result of the calculation can be used to allocate optional computation time when the accepted task releases a job that begins from an optional part.

The scheduler calculates the amount of idle time that can be allocated to the requested aperiodic job in a manner similar to the calculation of optional computation time for a periodic job in a static task set. There are three differences. First, this calculation does not need to consider the optional computation time allocated to jobs in the OQ, because an aperiodic job ready for mandatory part has higher priority than jobs ready for optional part. Second, the remaining mandatory execution time of jobs in the OQ must be considered. This is because the optional parts of jobs in the OQ may have to be terminated due to this activation, and in that case, these jobs may have higher priority than the accepted job. Third, the idle time already allocated to accepted soft aperiodic jobs may also be used for the requested firm aperiodic job.

The admission controller first calculates the minimum amount of idle time that can be allocated to J_ν by pretending that J_ν has the latest deadline among all aperiodic jobs ready for mandatory part. In other words, the amount of idle time is calculated as if J_ν is put to the tail of the AMQ when it is accepted. Supposing that $\Gamma_{\ell o}$ is a group of jobs in the OQ, then the amount of the idle time that can be used for J_ν is calculated as

$$I_\nu(t) = a_\nu + D_\nu - t - m_\nu - \hat{E}_\nu(t) - F_\nu(t) - \min\{G_\nu(t), H_\nu(t)\}, \tag{5.16}$$

where

$$\hat{E}_\nu(t) = E_\nu(t) + \sum_{J_i \in \Gamma_{\ell o}} (\ell_i^m(t) + X_i(t)). \tag{5.17}$$

If $I_\nu(t) \geq m_i$, then the acceptance test is passed and the aperiodic task τ_ν is activated. The release time and deadline of its job J_ν are set to a_ν and $a_\nu + D_\nu$, respectively. If J_ν starts from a mandatory part, the scheduler sets $R_\nu(t)$ to m_ν and $S_\nu(t)$ to 0, and put J_ν to the AMQ. Thus, optional computation time is not allocated at this moment. On the other hand, if the job starts from an optional part, the scheduler sets $R_\nu(t)$ to $I_\nu(t)$ and $S_\nu(t)$ to $I_\nu(t) - m_\nu$, and then puts J_i to the OQ. Then, in either case, the scheduler decreases the optional computation time allocated to jobs in the OQ that has a lower priority that J_ν, until the sum of the deallocated amount reaches $R_\nu(t)$ or there is no more job with lower priority. No amount of dynamically reserved computation time, however, is deallocated. As a result of this deallocation, if the optional computation time of a job becomes zero, its currently executed optional part is terminated. The terminated job is removed from the OQ, and put back to the PMQ or to the AMQ if there is still a mandatory part left to be executed.

If $I_i(t) < m_i$, then the admission controller goes on to test the following condition:

$$I_\nu(t) + \sum_{J_i \in \Gamma_{sae}} (R_i(t) - X_i(t)) \geq m_i, \tag{5.18}$$

where Γ_{sae} is a group of ready soft aperiodic jobs whose deadline is earlier than or equal to $a_\nu + D_\nu$. If the condition does not hold, the activation request for a firm aperiodic task is rejected, because there is not enough amount of the time available to the requested task. Otherwise, it is possible to accept the activation request by deallocating idle time allocated to soft aperiodic jobs. Thus, in this case, first the optional computation time except the reserved amount of all jobs in the OQ is deallocated. Then, all the computation time except the reserved amount of the jobs in Γ_{sae} are deallocated until the sum of all the deallocated time from jobs in the OQ and from those in the AMQ reaches m_ν or there is no more job left to deallocate from. After that, all jobs in the OQ are put back to the PMQ or the AMQ if there is still a mandatory part to be executed.

An activation request for a soft aperiodic task, on the other hand, is always accepted regardless of whether enough amount of idle time is available in the desired response time or not. Specifically, the scheduler first sets r_ν to a_ν and d_ν to $a_\nu + D_\nu$. Then, if $I_i(t) > m_\nu$, the scheduler releases a job in the same manner as in the case of releasing a firm aperiodic task. And otherwise, it sets $R_\nu(t)$ and $S_\nu(t)$ to $I_\nu(t)$ and 0, respectively. When J_ν overruns before its completion, the scheduler again calculates the amount of time available and increases the deadline of J_ν by D_ν.

Finally, when an aperiodic job is completed, all the amount of unused time it holds is reclaimed by other ready jobs. The reclaiming procedure here is the same as that taken for the completion of a periodic job. That is, the unused time is reclaimed by the job at the head of the OQ.

5.3.3 Dynamic Activation of Periodic Tasks

When activation of a firm periodic task is requested, the admission controller performs an acceptance test to check whether accepting the activation request does not leads to an infeasible task set. This test is again postponed until the completion of the resource access if the running job was accessing a resource.

From Theorem 1, the necessary and sufficient condition for accepting the request is the same as that of the EDF algorithm. The admission controller tests the sufficient condition for feasibly scheduling all mandatory parts of already accepted periodic tasks with that of the newly activated task. The necessary condition is not tested, because the test can take a pseudo-polynomial time unless the relative deadline and period of all periodic tasks are equal.

Suppose that Γ_p is a group of all active periodic tasks and the requested task τ_ν. Then, by renaming the tasks in Γ_p so that $D_i < D_j$ only if $i < j$, the acceptance test can be performed by checking the following condition:

$$\forall \tau_i \in \Gamma_p, \quad \sum_{j=1}^{i} \frac{m_j}{T_j} + \frac{1}{D_i} \sum_{j=1}^{i} \frac{T_j - D_j}{T_j} m_j + \frac{B_i}{D_i} \leq 1. \tag{5.19}$$

If Equation (5.19) is satisfied for all τ_i, the acceptance test is passed. Then, the scheduler goes on to determine if it is possible to release the first job of the requested

task at the same time as the arrival of activation request. If it is impossible, the scheduler determines the earliest time the requested task can actually release its first job. The reason why it is not always possible to release the first job immediately is that jobs in the AMQ and the OQ can miss their deadline when the amount of idle time available to these jobs decreases unexpectedly due to this activation. Thus, the scheduler checks whether the following condition holds:

$$\sum_{\tau_i \in \Gamma_{sae}} (R_i(t) - X_i(t)) + \sum_{\tau_i \in \Gamma_{oe}} (S_i(t) - X_i(t)) \leq m_\nu, \tag{5.20}$$

where Γ_{oe} is a group of jobs in the OQ with deadline earlier than or equal to $a_\nu + D_\nu$.

If Equation (5.20) holds, then the scheduler immediately releases the first job of the activated task at this time and decreases the amount of idle time allocated to the jobs in Γ_{sae} and Γ_{oe}. Specifically, the scheduler first deallocates the optional computation time from jobs in the OQ from its head, and then decreases the computation time allocated to soft aperiodic jobs in the AMQ, until the sum of the deallocated time equals m_ν. Note that no amount of reserved time is deallocated here. If, due to this deallocation, the amount of the idle time allocated to a job in the OQ has become zero, its optional part is terminated and put back to the PMQ or the AMQ, depending on its type, if it still has a mandatory part that needs to be executed.

On the other hand, if Equation (5.20) does not hold, the scheduler sets the release time of the first job to the latest deadline of all jobs in the AMQ and in the OQ. In this way, the amount of idle time allocated to jobs in the AMQ or the OQ does not decrease, and thus no deadline miss occurs.

5.3.4 Dynamic Suspension of Periodic Tasks

The M-FWP algorithm allows a periodic task to be suspended only at the end of its period. When a periodic task is suspended, the utilization of the system decreases. Thus, there is no need for admission control. Since the amount of idle time in the system increases by this suspension, it is desired that this newly available time be added to the optional computation time of jobs in the OQ.

When a periodic task τ_s is suspended, the amount of computation time that newly becomes available to a job J_i with lower priority is given as

$$\hat{S}_i(t) = \left(1 + \left\lfloor \frac{d_i - (r_s + T_s) - D_s}{T_s} \right\rfloor\right) m_s - \sum_{J_j \in \Gamma_{oe}} \hat{S}_j, \tag{5.21}$$

where Γ_{oe} is a group of jobs in the OQ that have earlier deadline than J_i. Thus, the scheduler increases the optional computation time of the job J_i in the OQ as follows:

$$S_i = S_i + \hat{S}_i(t). \tag{5.22}$$

Then, if there is a job J_i in the AMQ that has later deadline than any job in the OQ, the computation time of J_i is also increased by

$$\hat{S}_i(t) = \left(1 + \left\lfloor \frac{d_i - (r_s + T_s) - D_s}{T_s} \right\rfloor\right) m_s - \sum_{J_j \in \Gamma_{aoe}} \hat{S}_j, \tag{5.23}$$

where Γ_{aoe} is a group of jobs in the AMQ and the OQ that have earlier deadline than J_i.

5.3.5 Resource Access Control

First, it is important to note that the notion of quasi-normality [73] does not hold under the mandatory first strategy, since the deadline of a job is not the only criteria for scheduling jobs. Therefore, sophisticated protocols like Stack Resource Policy [52] cannot be used for the M-FWP algorithm.

The M-FWP algorithm forbids preemption of a job accessing a resource to accomplish correct resource sharing. Thus, the maximum blocking time of a job can be obtained as the maximum of access duration of any shared resources.

An important point that must be noted here is that when the imprecise computation model is deployed, the scheduler has to ensure that no job accessing a resource overruns. If an overrun occurs, there are only two choices, both undesirable. Either to terminate the optional part to risk the consistency of shared data or to continue to execute the overrunning job until it releases the resource to risk the temporal correctness.

To avoid overrun while during a resource access, the M-FWP scheduler allows a job J_i in the OQ to gain access to a resource only when its remaining optional computation time is larger than or equal to the worst case access duration of the requested resource. This is enough to avoid the problematic situation, because the the job accessing the resource is never preempted, even when a job with higher priority is released during the access.

5.4 Comparison with Related Work

Some of the scheduling algorithms in the literature that takes the mandatory first strategy are M-FED [44] and scheduling algorithms presented by Chung et al. [45]. The mandatory first strategy, if used to schedule aperiodic jobs, can achieve the theoretical upper bound on the competitiveness of an on-line scheduling algorithm. Moreover, the strategy is easy to implement, since there is no need to calculate the amount of time that can be allocated to optional parts. However, the mandatory first strategy in the literature has assumed that there is only one mandatory part per task. Thus, if these algorithms are used in practice to schedule imprecise tasks that require compensation for termination, the maximum execution time of all wind-up operations must be reserved after each optional part that may be discarded or terminated. Since this is an on-line scheduling strategy, it is difficult to determine which optional part would be terminated, especially

when the actual execution time of a task is different from the worst case execution time. Moreover, by rigidly following the original mandatory first strategy, it is impossible to find the latest time an optional part should be terminated to execute the wind-up operations before deadline. Hence, these algorithms are of only theoretical interests.

In contrast, the M-FWP algorithm explicitly targets tasks that have more than one mandatory parts. This allows the scheduling algorithm to be implemented without modifications for practical applications. The downside of this extension is that the mandatory first strategy is not competitive anymore, since the scheduling algorithm must follow the precedence constraints to execute optional parts before completion of all mandatory parts. The remedy is that the M-FWP algorithm considers the periodicity and criticalness of tasks. In other words, by discriminating periodic tasks from aperiodic tasks, the scheduling algorithm knows about the future arrival of periodic jobs. Thus, it is only jobs of dynamically activated aperiodic tasks and yet not accepted periodic tasks that may be rejected to degrade the competitiveness. Moreover, since the timing constraints of these tasks are only firm or soft, a catastrophe never occurs under the M-FWP algorithm. We assert that this discrimination is enough to make the M-FWP algorithm practical in many situations, although theoretically it is not competitive at all. By comparison, the original mandatory first strategy cannot guarantee the deadline of a hard real-time job even if it is competitive.

5.5 Summary

This chapter described the basic strategy for scheduling jobs and the request handling flow under transient overload. It then gave a theoretical description of the M-FWP scheduling algorithm. The important characteristics of the M-FWP algorithms are:

- all mandatory parts are scheduled before any optional parts as long as they belong to different jobs, so that a large amount of idle time is left in the schedule to resolve transient overload that may occur in the future;
- aperiodic jobs are scheduled prior to optional part to resolve the transient overload in a short period of time;
- the remaining optional computation time allocated to a job that requests a resource in an optional part must be larger than the worst case access duration of the requested resource to avoid any termination of optional part while it is accessing a resource; and
- a job accessing a resource is executed in a non-preemptive manner.

CHAPTER 6

SCHEDULING UNDER PERSISTENT OVERLOAD

This chapter focuses on scheduling under persistent overload. The persistent overload is triggered by activation of periodic tasks. Thus, it is literally not possible to resolve the overload until one or more periodic tasks are suspended.

The objective of scheduling under the persistent overload is to maximize the average QoS of the system achieved by active periodic tasks. The QoS of a task based on the linear task model is expressed as the reward that accrued by executing optional parts.

As in the previous chapter, this chapter first describes the basic strategies taken to schedule tasks and to control the QoS under persistent overload, and describes how dynamically arriving requests are handled in the system. Then, it describes the *Slack Stealer for Optional Parts* (SS-OP) scheduling algorithm developed for persistent overload and compares the presented algorithm with related work.

6.1 Basic Strategy

The final goal that must be achieved by allowing persistent overload is the full utilization of the processor. Since this goal is common to scheduling both under transient overload and under persistent overload, the bottom line scheduling strategy is also common. The SS-OP algorithm developed for persistent overload is based on the preemptive EDF scheduling algorithm and only reserves the worst case execution time of all mandatory parts of active periodic tasks.

The strategy taken to maximize the QoS is to dynamically determine a sub-optimal distribution of slack among optional parts. An amount of computation time allocated to optional parts and that allocated to jobs of accepted aperiodic tasks are the slack in the system, rather than the idle time. There are two reasons to use slack. The first reason is that the QoS of the system is known to become higher if optional part can consume slack. In fact, it was proven by Aydin et al. [42] that the mandatory first strategy is far from optimal for maximizing the QoS. The second reason is that it is vital to use slack

for controlling the QoS of each task. Controlling the QoS of each task is as important as maximizing the QoS of the system as a whole, because users would not want the QoS level of an application to vary unexpectedly up and down. For example, it is usually better to keep presenting moderate quality of video frames than switching back and forth between frames with the highest quality and with the lowest quality. Or, it is desirable to sustain the accuracy of control to some moderate level than to have it occasionally deteriorated from the highest level. Thus, it is crucially important to keep allocating a constant amount of computation time to optional parts. And from this aspect, the slack in the system is far easier to allocate in a controlled manner than idle time. It is known to be impractical, if not impossible, to control the amount of the idle time allocated to optional parts of each task, since it is almost impossible to place an idle interval in a desired moment without risking the feasibility.

The drawback of determining slack distribution for QoS control is often its optimization overhead. If the overhead of slack distribution is large, the amount of time that can be allocated to optional parts decreases. This can affect the performance especially under persistent overload.

To suppress the overhead of slack distribution, the SS-OP algorithm only updates the distribution when there is a change in the active periodic task set. Moreover, it distributes slack with respect to task. In other words, optional computation time is allocated per task instead of per job. This has three advantages. First, the optimization problem itself is simplified. Thus, the overhead of the distribution is decreased. Second, the QoS of a task is kept at a certain level with little fluctuation, since constant optional computation time is allocated to all jobs of a periodic task. And third, the overhead of the scheduler after the slack distribution is small, compared to determining the optional computation time each time a job requests execution of an optional part. One disadvantage is that a portion of slack may be left unclaimed by any periodic job. However, the unclaimed portion can still be used for aperiodic jobs. Moreover, real-world applications allow time attributes of tasks to be manipulated off-line, so that very little portion of slack is left unclaimed.

The QoS of a periodic task τ_i in this research is expressed in terms of reward. Supposing that jobs of a periodic task are allocated the same amount of slack for their optional parts, the definition that we use to express the average QoS of a periodic task τ_i is

$$Q_i = \frac{f_i(s_i)}{T_i}, \tag{6.1}$$

where s_i is the amount of slack equally allocated to every job of τ_i. Using this definition, the average QoS of the system is defined as

$$Q = \sum_{\tau_i \in \Gamma_p} Q_i. \tag{6.2}$$

where Γ_p is a group of all periodic tasks.

Table 6.1: Task set C

Task	a_i	n_i^M	n_i^O	m_i^1	o_i^1	D_i	T_i
τ_1	0	1	1	1	3	5	5
τ_2	0	1	1	2	6	10	10

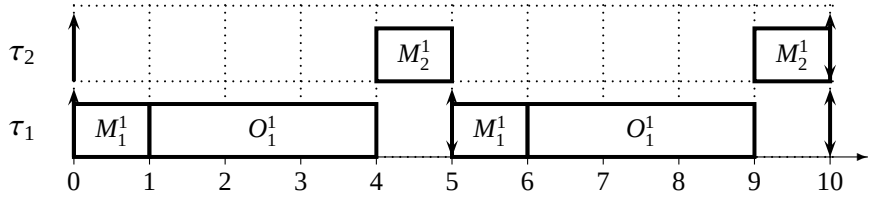


Figure 6.1: Schedule created using Equation (6.1)

By contrast, the average QoS of a periodic task in the literature [25, 42] under the same supposition is defined as

$$Q_i^{'} = f_i(s_i), \tag{6.3}$$

and the average QoS of the system is defined as

$$Q^{'} = \sum_{\tau_i \in \Gamma_p} Q_i^{'}. \tag{6.4}$$

The difference between Equation (6.1) and Equation (6.3) is important, because it determines how the optimization problem is formulated, which in turn determines how slack is distributed. In the former case, the average QoS is considered with respect to the length of the interval in concern, whereas in the latter case it is considered with respect to the number of jobs separately for each task. In other words, the former is more suited to estimate the overall QoS of the whole system, whereas the latter is more suited to estimate the QoS achieved by each task. Since we are not interested in the QoS achieved by an individual task under persistent overload, we conclude that it is more relevant to use the former definition.

To observe the difference more clearly, consider the following example. There are two periodic tasks τ_1 and τ_2 whose time attributes are shown in Table 6.1. The tasks τ_1

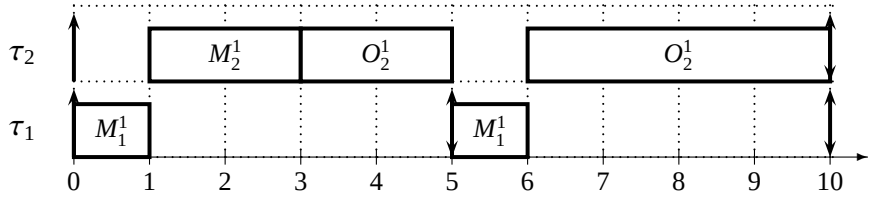


Figure 6.2: Schedule created using Equation (6.3)

and τ_2 have only one segment and their reward functions are defined as $f_1(t) = 1.1t$ and $f_2(t) = t$, respectively. Clearly, there exist six units of slack in the interval $[0, 10)$. Since the hyperperiod of these two tasks are 10, the total reward in this interval is considered here as a measurement of the QoS achieved by the system. Using the definition in Equation (6.1), the scheduling algorithm presented in this chapter allocates slack in the non-increasing order of q_i^1/L_i^1 in this case. Thus, all slack in the system is allocated to τ_1 and, as Figure 6.1 shows, two optional parts of τ_1 is completed in the schedule. On the other hand, if the definition in Equation (6.3) is used, the slack is all allocated to τ_2, because, according to Aydin et al. [42], if there is only one segment for each task and the relative deadline and period of each task are equal, the optimal method for distributing slack among optional parts is to allocate slack in the non-increasing order of $q_i^1 T_i/L_i^1$. Thus, in this latter case, as Figure 6.2 shows, only one optional part of τ_2 is completed in the schedule.

Examining the schedules with respect to reward, the total reward achieved in Figure 6.1 is 6.6, while that in Figure 6.2 is 6. Since the total reward is higher if τ_1 is allocated slack in preference to τ_2, the average reward calculated by Equation (6.1) is also higher. Thus, it is the definition given as Equation (6.1) that can maximize the total reward in the system.

There is one more important observation to make in the schedules. In Figure 6.1, the slack is allocated to tasks whose jobs contribute more to the system. On the other hand, in Figure 6.2, the slack is allocated to tasks whose jobs altogether consume lesser amount of slack to maximize the reward of a single task. Thus, using Equation (6.3), the slack tends to be allocated to tasks with longer period. One of the consequences is that the reward accrued in the system does not increase for a longer period of time if the optional parts have 0/1 constraints. For example, the total reward in Figure 6.2 does not increase until 10, while the total reward in Figure 6.1 increases twice in the schedule at four and nine. Thus, the latter definition is not suited if the system tolerates only a gradual change in the overall QoS. A more serious consequence is that the amount of slack that cannot be allocated to periodic tasks can become larger, if a task should be allocated the same amount of optional computation time for every one of its jobs. In the case of tasks shown in Table 6.1, τ_2 needs one unit of slack in the hyperperiod to increase the optional computation time and τ_1 two units. Now, Suppose that there are only five units of slack in the system and that $O_1^1 = 2$ and $O_2^1 = 4$. Then, using our definition, every job of τ_1 is allocated two units of optional computation time and that of τ_2 one unit. On the other hand, using the definition in the literature, every job of τ_2 is allocated four units of optional computation time and none for those of τ_1, which leaves one unit of slack unused every hyperperiod. Hence, our definition of QoS can lead to higher effective processor utilization.

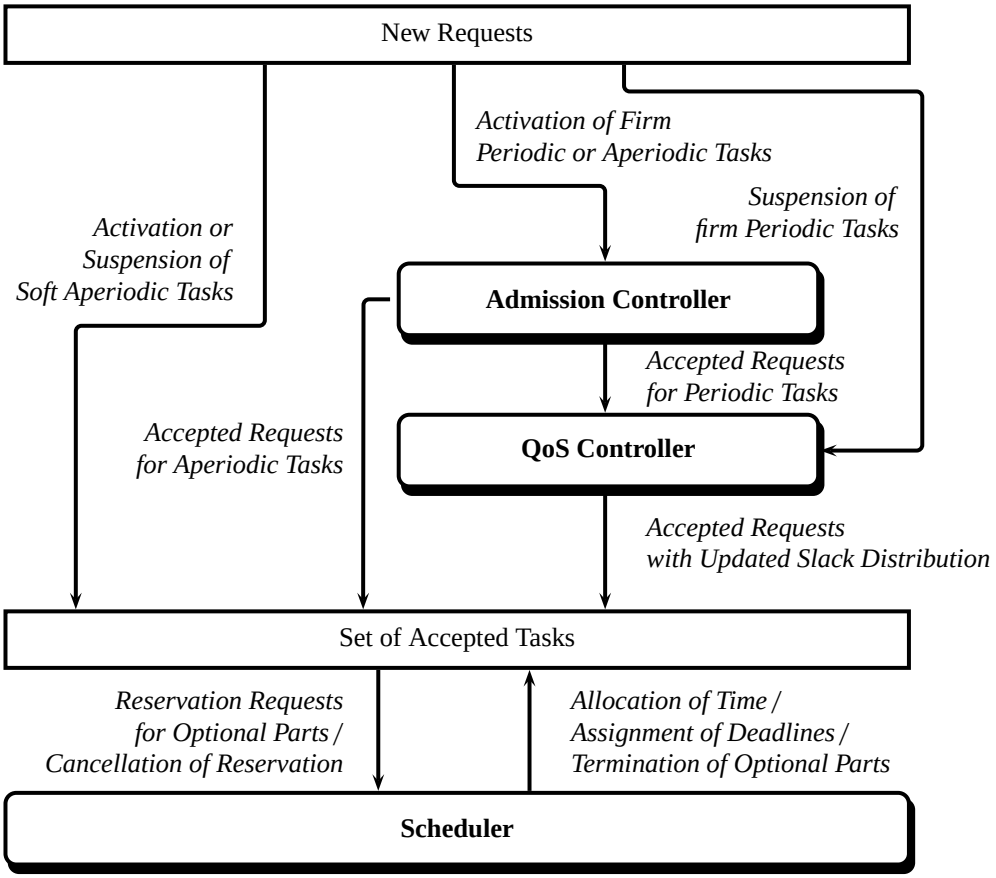


Figure 6.3: Request handling flow under persistent overload

6.2 Request Handling Flow

Figure 6.3 shows how the scheduling algorithm designed for persistent overload handles incoming requests. The entities concerned with processing of requests are the *admission controller*, the *QoS controller*, and the *scheduler*.

The role of the admission controller is to check whether accepting an activation request of a firm task does not jeopardize the guaranteed feasibility of already accepted tasks. The actions taken if an activation request cannot be accepted at the time of the arrival are similar to those in the M-FWP algorithm. One difference is that the release time of the first job of an acceptable periodic task is almost always postponed to a later time than the arrival time of the request. The activation of soft aperiodic tasks, on the other hand, directly changes the set of accepted tasks, because its execution can be delayed after its deadline if its timely execution causes a hard or firm deadline to be missed.

The QoS controller is responsible for distributing the slack that can be used for

executing optional parts. The accepted activation requests and suspension requests of periodic tasks go through the QoS controller, since these requests cause a change in the set of active periodic tasks.

All jobs of the accepted tasks are actually scheduled by the scheduler. The SS-OP scheduler also handles requests for resources, but in different manner from the M-FWP scheduler. The other roles of the scheduler are to allocate processor time to jobs, to assign an appropriate deadline to soft aperiodic jobs, to terminate optional parts on its overrun, to reserve computation time for optional parts, and to cancel the reservation.

6.3 The Slack Stealer for Optional Part Scheduling Algorithm

The SS-OP algorithm schedules jobs in an integrated manner to maximize the average QoS of the system and to minimize blocking time due to resource sharing. The SS-OP algorithm is conceptually similar to the EDF algorithm with the Stack Resource Policy (SRP) protocol in that the schedules created by these algorithms are equivalent if systems are not overloaded and no aperiodic job is requested.

The SS-OP algorithm assumes that all ready jobs are maintained in a single ready queue. The priority of a job in the ready queue is determined by its deadline. Specifically, a job with earlier deadline has higher priority. Ties between jobs with the same deadline are broken in favor of shorter relative deadline. The SS-OP algorithm assumes that the mutual exclusion of resource accesses are achieved by means of semaphores. The preemption level of a job J_i is calculated as $1/D_i$. The resource ceilings and the system ceiling are defined exactly as in the SRP protocol by considering all tasks in the system, including those that are activated only on-line.

The important differences exist in the following points. First, the amounts of optional computation time of periodic jobs are determined by the QoS controller before each job is released, so that the overall QoS is maximized under persistent overload. Second, the interference by dynamically arriving aperiodic jobs in the optional computation time of periodic jobs is limited to a level determined off-line. And third, additional resource access control is performed, since the original SRP protocol cannot be deployed when a job accessing a resource may overrun in its optional part.

The variables that the SS-OP scheduler needs to maintain are $R_i(t)$, $S_i(t)$, and $X_i(t)$, whose definitions are the same as those in the M-FWP algorithm.

6.3.1 Slack Distribution for a Set of Static Tasks

The distribution of slack among a set of static periodic tasks has two steps. The first step is to calculate the amount of slack in the system, and the second step is to actually distribute the slack among optional parts.

The amount of slack is calculated using the notion of processor demand. Suppose that there are N periodic tasks $\tau_1, \ldots, \tau_N$ and that $D_i < D_j$ holds only if $i < j$. Then,

based on the results of the feasibility analysis presented by Devi [75], the processor bandwidth of the slack is calculated as follows:

$$u_S = \min\left\{1 - \sum_{j=1}^{i} \frac{m_j}{T_j} - \frac{1}{D_i}\sum_{j=1}^{i} \frac{T_j - \min\{T_j, D_j\}}{T_j} m_j - \frac{B_i}{D_i} \mid 1 \le i \le N\right\}. \tag{6.5}$$

The processor bandwidth u_S given by Equation (6.5) can be just an underestimation of the actual amount of slack in the system. However, a precise calculation of slack by the processor demand approach is not appropriate for on-line analysis, due to its pseudo-polynomial time complexity in the general case.

When the relative deadline and period of all periodic tasks are equal, however, the processor bandwidth u_S can be calculated more precisely as follows:

$$u_S = \min\left\{1 - \sum_{j=1}^{i} \frac{m_j}{T_j} - \frac{B_i}{T_i} \mid 1 \le i \le N\right\}. \tag{6.6}$$

Moreover, in a special case where the relative deadline and period of all periodic tasks are equal and no job is ever blocked, the processor bandwidth u_S can be calculated precisely, and in this case, Equation (6.5) is simplified as follows:

$$u_S = 1 - \sum_{j=1}^{N} \frac{m_j}{T_j}. \tag{6.7}$$

The SS-OP algorithm requires that u_S be larger than zero to allocate non-zero optional computation time. For the moment, this dissertation assumes that u_S is kept positive, for instance by adjusting parameters of periodic tasks off-line.

The next step is to distribute the slack. The optimal distribution of slack can be theoretically achieved by solving the following optimization problem:

$$maximize \sum_{i=1}^{N} \frac{f_i(s_i)}{T_i} \tag{6.8}$$

$$subject\ to \sum_{i=1}^{N} \frac{s_i}{\min\{T_i, D_i\}} \le u_S, \tag{6.9}$$

$$\forall i, 0 \le s_i \le o_i, \tag{6.10}$$

where s_i is the total amount of slack allocated to τ_i. However, in practice, the unit of execution time that can be allocated to a job on an operating system is not infinitely small. Thus, the above optimization problem must be solved as an integer programming problem. Unfortunately, an integer programming problem is intractable. Thus, a sub-optimal solution to the problem must be sought. The SS-OP algorithm solves the optimization problem assuming that it is a linear programming problem, and then round

1: $s_i := 0$ for $0 \leq i \leq N$;
2: Construct a set $\Theta = \{\varphi_1^1, \varphi_2^1, \ldots \varphi_N^1\}$;
3: **while** $\Theta \neq \emptyset$ or $u_S \neq 0$ **do**
4: Take out one segment φ_i^j with the largest contribution rate $q_i^j \min\{T_i, D_i\}/L_i^j T_i$ from Θ;
5: **if** $L_i^j / \min\{T_i, D_i\} \leq u_S$ **then**
6: $s_i := s_i + L_i^j$;
7: $u_S := u_S - L_i^j / \min\{T_i, D_i\}$;
8: **if** there exists φ_i^{j+1} **then**
9: Add the segment φ_i^{j+1} to Θ;
10: **end if**
11: **else**
12: $s_i := s_i + \lfloor u_S \min\{T_i, D_i\} \rfloor$;
13: Take out all the segments that belong to tasks with period shorter than τ_i form Θ;
14: **end if**
15: **end while**

Figure 6.4: Pseudo-code of sub-optimal slack distribution algorithm

down the amount of slack allocated to each task to the smallest unit that can be managed by the real-time operating system used for implementation.

To solve the optimization problem, the SS-OP algorithm allocates slack in the non-decreasing order of contribution rate. The contribution rate of a periodic task τ_i is defined as $q_i^j \min\{T_i, D_i\}/L_i^j T_i$. To see why this maximizes the average reward, first consider the simplest case where each reward function has only one segment. Suppose the amount of slack allocated to the optional parts of τ_i is increased by one, then the average reward of the system increases by $q_i^1/L_i^1 T_i$. However, by doing so, the processor bandwidth of yet unallocated slack decreases by $1/\min\{T_i, D_i\}$. Thus, the contribution rate of this task can be calculated as $q_i^1 \min\{T_i, D_i\}/L_i^1 T_i$. Now, if two tasks with different contribution rate are considered, the total reward of the system never decreases by allocating slack in favor of the one with larger contribution rate. Therefore, the optimal strategy for solving this linear programming problem is to allocate an amount of time in the non-decreasing order of the contribution rate.

This strategy does not need to be modified even when a reward function has more than one segments. Since the reward function $f_i(t_i)$ with multiple linear segments has a concave form, it must always be $q_i^j/L_i^j > q_i^k/L_i^k$ only if $j < k$. This means that as long as slack is allocated to the segments of a task in the order of their execution, that is φ_i^j before φ_i^{j+1}, the strategy is still optimal.

The pseudo-code of the algorithm for slack distribution algorithm is shown in Figure 6.4. The algorithm continues allocating slack to a segment until all the segments are considered or there is no more amount of slack left for allocation. In every loop, one segment φ_i^j that has the largest contribution rate in the set Θ is taken out and is consid-

ered for slack allocation. If there is enough amount of slack left to meet the maximum requirement of the segment, the allocated amount of the slack is equal to L_i^j. Otherwise, the maximum amount of the slack left at that point is allocated to the segment without violating the constraint that every job of the task is given the same optional computation time. The amount of slack in this latter case is rounded down to an integer number by supposing that the unit time is one. Moreover, at this moment, all segments that belong to the tasks with period shorter than or equal to τ_i are removed from the set Θ. None of these segments can be allocated an amount of slack without violating the above constraint, because the amount of the yet unallocated slack is not enough to increase their optional computation time by one unit. The time complexity of this algorithm is $O(KN \log N)$, because the above loop is iterated at the maximum of KN times.

6.3.2 Scheduling a Set of Static Tasks

The scheduling events in the static system are the same as those in the M-FWP algorithm for transient overload. However, operations for each event are much simpler. Once the distribution of slack is determined, the scheduler only has to allocate the processor to jobs in the eligible order determined by the SRP protocol and to make sure that no job overruns in its optional part.

When a job J_i is released, the scheduler allocates computation time to J_i. The amount of computation time given is the sum of the worst case execution time of all mandatory parts and the amount of slack allocated by the QoS controller. The scheduler also checks whether J_i has the earliest deadline among ready jobs and whether its preemption level is higher than the system ceiling. If it is the case, preemption occurs and the processor is allocated to J_i.

When a job completes a mandatory part and becomes ready for optional part, the scheduler performs intra-task reclaiming. The scheduler adds any unused amount of time reserved for the completed mandatory part to the optional computation time. On the other hand, when the job finishes its execution with the completion of this mandatory part, any unused time is reclaimed by the next running job. The scheduler chooses the next running job according to the SRP protocol. In particular, if a job J_i has the earliest deadline, by excluding the job that has just completed its execution, and has preemption level higher than the system ceiling, J_i is given the processor. Otherwise, the job that has increased the system ceiling to the current level is given the processor.

When a job completes an optional part and becomes ready for mandatory part, there is nothing the scheduler has to do. On the other hand, if the job is finished with the completion of the optional part, the scheduler removes the job from the ready queue and adds any unused time to the next running job, which is chosen according to the SRP protocol.

When a job overruns in an optional part, the scheduler terminates the optional part. If the job is then ready for mandatory part, the processor is given to the same job. Otherwise, the processor is given to the next job again chosen according to the SRP

protocol.

When reservation of optional computation time is requested by a job J_i, the scheduler checks whether the requested amount δ is larger than or equal to $S_i(t) - X_i(t)$. If it is, the request is rejected. Otherwise, $X_i(t)$ is increased by δ.

Finally, when cancellation of all previously made time reservation is requested by a job J_i, the scheduler sets $X_i(t)$ to zero without any acceptance test. On the other hand, when only part of reservation is canceled, the scheduler checks whether the amount of time $\hat{\delta}$ that should remain reserved is less than or equal to $X_i(t)$. If not, the request is rejected, and otherwise $X_i(t)$ is decreased to $\hat{\delta}$.

The feasibility of the static task set is guaranteed if and only if the mandatory parts of all tasks can be feasibly scheduled by the EDF algorithm with the SRP protocol. The SS-OP algorithm, however, does not test the necessary and sufficient condition, because it is not practical to test this feasibility condition on-line, which is required when periodic tasks are dynamically activated.

The sufficient condition for SS-OP algorithm to create a feasible schedule is

$$u_S \geq 0. \tag{6.11}$$

The correctness of this sufficient condition is stated in the following theorems.

Theorem 2. *A set of static periodic tasks $\tau_1, \tau_2, \ldots, \tau_N$ whose period is less than or equal to its relative deadline is feasibly scheduled by the SS-OP scheduling algorithm only if the following condition holds:*

$$\forall i, 1 \leq i \leq N, \quad \sum_{j=1}^{i} \frac{m_j}{T_j} + \frac{1}{D_i} \sum_{j=1}^{i} \frac{T_j - D_j}{T_j} m_j + \frac{B_i}{D_i} + u_S \leq 1. \tag{6.12}$$

Proof. If the processor bandwidth of the slack in the system u_S is zero, then the condition shown in Equation (6.12) is the same as the sufficient feasibility condition shown by Devi [75].

If $u_S > 0$, then it is clear that the optional processor demand of all jobs is less than or equal to u_S. Thus, the following must hold for $i = 1, \ldots, N$:

$$u_S \geq \sum_{j=1}^{i} \frac{s_j}{D_j} \tag{6.13}$$

$$= \sum_{j=1}^{i} \frac{s_j}{D_j} \left(1 - \frac{D_j}{D_i}\right) + \frac{1}{D_i} \sum_{j=1}^{i} s_j \tag{6.14}$$

$$\geq \sum_{j=1}^{i} \frac{s_j}{T_j} + \frac{1}{D_i} \sum_{j=1}^{i} \left(\frac{T_j - D_j}{T_j}\right) s_j. \tag{6.15}$$

Combining this inequality with Equation (6.12), we have the following:

$$\forall i, 1 \leq i \leq N, \quad \sum_{j=1}^{i} \frac{m_j + s_j}{T_j} + \frac{1}{D_i} \sum_{j=1}^{i} \frac{T_j - D_j}{T_j}(m_j + s_j) + \frac{B_i}{D_i} \leq 1. \tag{6.16}$$

By assuming that the execution time of a periodic job J_i is $m_i + s_i$, Equation (6.16) is again equivalent to the above sufficient feasibility condition shown by Devi.

Hence, the static periodic task set is feasible under the SS-OP scheduling algorithm only if Equation (6.12) holds. □

Theorem 3. *A set of static periodic tasks $\tau_1, \tau_2, \ldots, \tau_N$ with period equal to relative deadline is feasibly scheduled by the SS-OP scheduling algorithm only if the following condition holds:*

$$\forall i, 1 \leq i \leq N, \quad \sum_{j=1}^{i} \frac{m_j}{T_j} + \frac{B_i}{T_i} + u_S \leq 1. \tag{6.17}$$

Proof. By substituting D_i and D_j with T_i and T_j, respectively, in Equation (6.12), we have Equation (6.17). Hence, the theorem follows. □

In the following, task sets that contains a periodic task whose relative deadline is smaller than or equal to its period are considered, since the above theorems show that the system with periodic tasks all having the same relative deadline as their period is only a special case.

6.3.3 Dynamic Activation of Aperiodic Tasks

When activation of a firm aperiodic task τ_ν is requested, the admission controller checks its feasibility.

The admission controller first determines the amount of time that can be allocated to a job of the activated task without decreasing the optional computation time of a periodic job, which is only possible when the system is not overloaded. Suppose that there are N active periodic tasks $\tau_1, \tau_2, \ldots, \tau_N$ in the system. Then, the computation time that can be allocated to the requested job J_ν without decreasing the optional computation time of periodic jobs is given as

$$A_\nu(t) = (t + D_\nu - t_{LA})\hat{u}_A \tag{6.18}$$

where t_{LA} is the latest deadline of all aperiodic jobs, t is the time the activation request is handled, which is a_ν in this case, and

$$\hat{u}_A = u_S - \sum_{i=1}^{N} \frac{s_i}{D_i}. \tag{6.19}$$

If $A_\nu(a_\nu) \geq C_\nu$, then the request is fully accepted. That is, all the mandatory parts and optional parts of J_ν can be executed in its desired response time without affecting the QoS of periodic jobs. In this case, the scheduler sets release time r_ν to a_ν, deadline d_ν to $a_\nu + D_\nu$, and computation time $R_\nu(t)$ to C_i. Otherwise, the aperiodic job can only be completed by deallocating some amount of slack from periodic jobs.

Deallocating optional computation time from a ready periodic job results in QoS degradation of the system. Thus, the SS-OP algorithm defines the maximum processor bandwidth that can be used for aperiodic jobs as

$$u_A = \max\left\{\frac{m_i}{D_i} \mid J_i \in \Gamma_{ap}\right\}, \tag{6.20}$$

where Γ_{ap} is a group of all accepted aperiodic jobs. Moreover, it does not allow deallocation from periodic jobs that have executed at least a portion of their optional part, those that have reserved non-zero amount of slack, nor those that have their optional computation time already decreased by previous activation of an aperiodic task. This dissertation denotes the latest deadline of these periodic jobs as t_{LP}. Also, if there is a new distribution of slack due to activation or suspension of a periodic task scheduled in the future, the SS-OP algorithm also takes that time into consideration. This dissertation here supposes that this next distribution time is given as t_d. Then, the activation request for a task τ_ν is accepted only if the following condition is met:

$$\max\{t_{LP}, t_d\} + \frac{m_\nu - A_\nu(a_\nu)}{\min\{u_A, u_S - \hat{u}_A\}} \leq a_\nu + D_\nu. \tag{6.21}$$

If the activation request is accepted, the scheduler releases the job J_ν by setting release time r_ν to a_ν and deadline d_ν to $a_\nu + D_\nu$. Moreover, since the maximum amount of time that can be deallocated from periodic jobs is given as

$$P_\nu = \max\{0, d_\nu - \max\{t_{LP}, t_d\}\} \times \min\{u_A, u_S - \hat{u}_A\}, \tag{6.22}$$

the computation time allocated to J_ν is given as follows:

$$R_\nu(t) = \min\{m_\nu + o_\nu, A_\nu(t) + P_\nu\}. \tag{6.23}$$

The periodic jobs whose optional computation time must be decreased by this allocation of slack to the accepted aperiodic job are confined to those with deadline later than or equal to $\max\{t_{LP}, t_d\}$. The deallocation of slack is carried out by considering these jobs in the non-decreasing order of its deadline. Since the maximum amount of slack that should remain allocated to the periodic job J_i is given as

$$S_i(t) = \frac{u_S - \hat{u}_A - \min\{u_A, u_S - \hat{u}_A\}}{u_S - \hat{u}_A} s_i, \tag{6.24}$$

any excessive amount of slack is deallocated until the sum of the deallocated amount equals $R_\nu(t) - A_\nu(t)$. This deallocation is also applied to all jobs that are released after

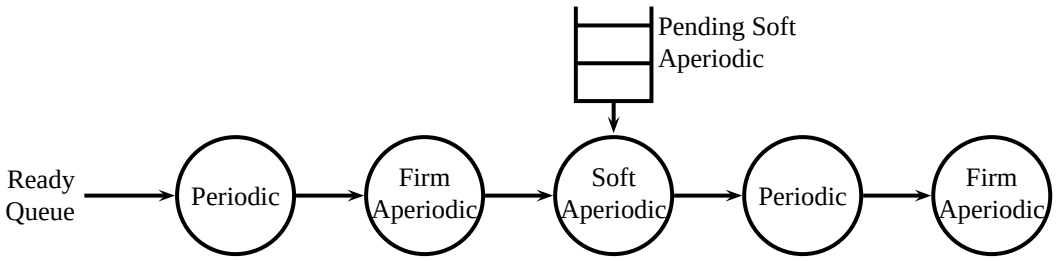


Figure 6.5: Ready queue with aperiodic jobs

this activation time and have deadline earlier than d_ν. For all other jobs, their optional computation time is unaffected and remains the same as the amount calculated by the QoS controller.

By comparison, activation requests for soft aperiodic tasks are handled differently. A soft aperiodic task is activated without admission control. However, its actual execution is controlled by the SS-OP scheduler. The SS-OP scheduler allows at most one ready soft aperiodic job to exist in the system at any moment for the purpose of decreasing the response time of soft aperiodic jobs and increasing the acceptance ratio of activation requests for firm aperiodic tasks. Figure 6.5 illustrates the ready queue with aperiodic jobs. Notice that there is only soft aperiodic job, while there are two firm aperiodic jobs. The pending requests for soft aperiodic tasks τ_ν are enqueued in the non-decreasing order of their original requested deadline $a_\nu + D_\nu$. There is no corresponding job for these pending requests, until the request is actually handled by the scheduler when the only soft aperiodic job in the system is completed.

When the request is actually handled at time t, the scheduler releases the soft aperiodic job J_ν by setting release time r_ν to t.

If $A_\nu(t) \geq C_\nu$, then J_ν is given the computation time of C_i and the deadline of $t + D_\nu$. Otherwise, the computation time is allocated by Equation (6.23) and the deadline is set as follows:

$$d_\nu = \max\left\{t + D_\nu,\ \max\{t_{LP}, t_d\} + \frac{m_\nu - A_\nu(t)}{\min\{u_A, u_S - \hat{u}_A\}}\right\}. \tag{6.25}$$

Then, in this latter case, the scheduler decreases the optional computation time of periodic jobs. The condition for decreasing the optional computation time of periodic jobs and the deallocated amount are the same as those in the case of releasing a firm aperiodic job.

6.3.4 Dynamic Activation of Periodic Tasks

When activation of a periodic task τ_ν is requested, the admission controller first checks the feasibility of a group of tasks Γ_p that consists of all active periodic tasks and the requested task τ_ν. The feasibility test checks whether the processor bandwidth of slack

still remains non-negative after accepting the activation request. Thus, by renaming the tasks in Γ_p so that $D_i < D_j$ only if $i < j$, the acceptance test can be performed by checking the following condition:

$$\forall \tau_i \in \Gamma_p, \quad \sum_{j=1}^{i} \frac{m_j}{T_j} + \frac{1}{D_i} \sum_{j=1}^{i} \frac{T_j - D_j}{T_j} m_j + \frac{B_i}{D_i} \leq 1. \tag{6.26}$$

If Equation (6.26) is satisfied, the request is accepted and the QoS controller is invoked. Otherwise, the request is rejected.

When the QoS controller is invoked, it redistribute the slack among active periodic tasks according to the algorithm shown in Figure 6.4. Then, the scheduler determines the time to actually release the first job of the activated task and to follow the new distribution of slack. The SS-OP scheduler considers the time given by $\max\{t_{LA}, t_d, t_{LP}\}$ as the time to release the first job and to follow the new distribution. Thus, any jobs released at or later than this time is given an amount of slack the QoS controller has newly calculated.

6.3.5 Dynamic Suspension of Periodic Tasks

As Figure 6.3 shows, the suspension of a periodic task τ_i does not need admission control and is directly handled by the QoS controller. The suspension request is handled at the time a job is finished and the QoS controller and the scheduler handle the request just in the same manner as the activation of a periodic task. The QoS controller redistribute the slack among the new set of active periodic tasks using the algorithm shown in Figure 6.4. And then, the scheduler begins to follow this new distribution from the time given by $\max\{t_{LA}, t_d, t_{LP}\}$.

6.3.6 Resource Access Control

The SS-OP algorithm schedules jobs in the same order as the EDF algorithm with the SRP protocol. When preemption of only precise jobs is controlled by the SRP protocol, no access control is required each time a resource is actually requested and released when the system is not overloaded. However, since there is no guarantee that an imprecise job is not terminated while it is accessing a resource, additional resource access control is required when a resource is requested in an optional part.

The SS-OP scheduler denies acquisition requests for a resource from any optional part that may not be able to execute until they release the acquired resource. As in the case of the M-FWP algorithm, this can be ensured by accepting the requests only from jobs that are executing a mandatory part or that are executing an optional part but have optional computation time larger than the worst case access duration of the resource. Since the amount of slack allocated to a job that has once started execution of optional part never decreases under the SS-OP algorithm, this is enough to ensure that no optional part overruns inside a critical section.

The properties of the SRP protocol is preserved by adding the completion check to the original SRP protocol. This following theorem states the equivalence on the maximum blocking time.

Theorem 4. *The maximum blocking time of a job under the SS-OP algorithm is equal to that under the SRP protocol.*

Proof. The definition of the preemption level used in the SS-OP algorithm fulfills the conditions of the preemption level in the SRP protocol. Moreover, all jobs are released in the same order as the EDF algorithm with the SRP protocol. Furthermore, all the acquired resources are always released. Hence, by Baker [52], the theorem follows. □

From Theorem 4, a job J_i scheduled by the SS-OP algorithm is blocked at most once before it is given the processor, and its maximum blocking time B_i is bounded by the longest duration of access to resources shared with jobs that can block J_i.

6.4 Comparison with Related Work

The SS-OP algorithm can be compared with related work with respect to the way it determines the amount of slack in the system and the way it distributes the slack to maximize the QoS of the system.

Under the SS-OP algorithm, the amount of the slack is determined based on the feasibility analysis presented by Devi [75]. The analysis has a linear time complexity, and gives a bound tighter than that by the density test but looser than that by the processor demand test with pseudo-polynomial time complexity. There is also a tighter test with polynomial time complexity presented by Albers and Slomka [76]. These tests have a trade-off: a higher complexity can provide a tighter bound. Since the SS-OP algorithm is an on-line scheduling algorithm, a test with larger complexity than a linear time can degrade the practicality. However, we note that it is possible to replace the feasibility analysis in the SS-OP algorithm to meet the requirements of specific systems. For example, it is possible to use the density test if the relative deadline and period of all periodic tasks are almost the same. On the other hand, a more complicated test may be used when the interval between two consecutive activation or suspension requests are relatively long enough to pay for the large overhead.

The distribution of slack to maximize the QoS of the system is one of the well studied topics in imprecise task scheduling. The amount of slack that can be allocated to optional parts in these algorithms are determined before jobs actually begin executing the optional parts. Thus, it is possible to extend these existing algorithm to schedule imprecise tasks with wind-up operations. Among them, the OPT-LU algorithm [42] is known to be optimal for periodic imprecise tasks with concave reward functions. There are other heuristic on-line scheduling algorithms that can handle aperiodic tasks and 0/1 constraints. The largest and most significant difference between these algorithms and

the SS-OP algorithm is in the definition of the average QoS. As was shown in the beginning of this chapter, these definitions could lead to completely different distribution. Moreover, the existing methods do not allow periodic tasks to have relative deadline different from period.

Finally, the SS-OP algorithm determines the deadline of aperiodic jobs in a similar manner to Total Bandwidth Server (TBS) [77]. In the simplest case, the processor bandwidth of the server is equivalent to u_A. However, in comparison to TBS that needs to reserve the server bandwidth, the processor bandwidth u_A is not reserved for aperiodic jobs. While the dynamic allocation leads to rejection of aperiodic requests, it increases the effective processor utilization when there is no aperiodic jobs. Moreover, the SS-OP algorithm can use u_A to control the affect of aperiodic jobs on periodic jobs. By setting u_A to a larger value than what Equation (6.20) gives, larger amounts of optional parts can be executed by aperiodic jobs, at the cost of further degrading the QoS of periodic jobs. Moreover, blocking time due to aperiodic jobs is also limited to the minimum amount, since aperiodic jobs are never given relative deadline shorter than their desired response time. By contrast, the server mechanisms that minimizes the response time of aperiodic jobs as much as possible, for example TB* with the SRP protocol [78], suffers from longer blocking time, since it must be regarded that all aperiodic jobs may block periodic jobs.

6.5 Summary

This chapter first described the basic strategies for scheduling jobs and controlling QoS under persistent overload. It then described the request handling flow and gave the theoretical description of the SS-OP scheduling algorithm. The important characteristics of the SS-OP scheduling algorithms are:

- the slack in the system is distributed to optional parts of active periodic tasks to achieve the sub-optimal performance in terms of the average QoS;
- the definition of the QoS is different from the literature to maximize the average QoS of the whole system;
- the amount of slack allocated to jobs of a periodic task does not change unless an aperiodic or periodic task is activated or a periodic task is suspended, so that the QoS of each periodic task is stable and predictable;
- the maximum processor bandwidth used for aperiodic jobs are limited to u_A that can be given by the system designer to control the level of QoS degradation suffered by periodic jobs;
- unlike most of the existing server mechanisms for scheduling soft aperiodic jobs, no amount of time is dedicatedly reserved for aperiodic jobs to increase the effective processor utilization; and

- a job is only allowed to access resources in its optional part when its remaining optional computation time is larger than the worst case access duration to ensure that no job overruns while holding a resource.

CHAPTER 7

SIMULATION STUDIES OF THE PROPOSED ALGORITHMS

This chapter presents results of simulations conducted to estimate the performance of the proposed scheduling algorithms. The simulations were carried out by running an embedded real-time operating system called *RT-Frontier*, developed in this research, in the simulator mode. The source codes of the scheduler used in the simulator mode is common to that in a kernel configured for a real processor. However, no overhead is accounted for in the simulator and all the processor time can be used for application workloads.

The simulations characterize two types of overloads targeted in this research, namely the transient overload and the persistent overload. Imprecise tasks used in the simulations, both periodic and aperiodic, have two mandatory parts and one optional part. They start from one of their mandatory parts and end with the other mandatory part. The M-FWP algorithm and the SS-OP algorithm were tested under both types of overload to compare their performance. Moreover, the performance of the EDF algorithm is also shown to provide a baseline for comparison. In the simulations for the EDF algorithm, no optional part is distinguished within a task. Thus, the worst case execution time of a task under the EDF algorithm was set to the total of the worst case execution time of its mandatory parts and that of its optional part. Under the EDF algorithm, the Total Bandwidth Server (TBS) [77] is used to serve activated aperiodic tasks. The size of TBS, which is the processor bandwidth given to the server, is set to the maximum that can be claimed without putting the system into overload.

7.1 Performance under Transient Overload

The performance metrics of scheduling under the transient overload is the average acceptance ratio of dynamic activation requests for firm aperiodic tasks, because higher acceptance ratio means higher ability to resolve overload.

Table 7.1: Simulation parameters for transient overload

Parameters	Values
Simulation length	10000000
Number of periodic tasks	10
Type of periodic deadline	Hard
Processor utilization (U)	0.8, 0.9, 1.0
Period (T_i)	[1000, 50000]
Worst case execution time (C_i)	$0.1U$
Ratio of mandatory parts for periodic tasks (m_i/C_i)	0.7
Accuracy of worst case execution time (AET/WCET)	0.2, 0.5, 0.8, 0.9, 1.0
Number of aperiodic tasks	10
Type of aperiodic deadline	Soft
Mean interarrival time of aperiodic requests (t_A)	1000
Mean worst case execution time of aperiodic jobs (t_C)	100, 200, 300, 400, 500
Ratio of mandatory parts for aperiodic tasks (m_i/C_i)	0.8
Size of TBS	$1 - U$

7.1.1 Experimental Setup

The transient overload in this research occurs only when activation of an aperiodic task is requested. Thus, workloads that simulate the transient overload consists of static periodic tasks and dynamic aperiodic tasks.

The workloads for simulating transient overload are generated using parameters shown in Table 7.1. Each workload includes ten periodic tasks. The periodic tasks have hard deadline, and their period is equal to their relative deadline. The periods are randomly chosen from a uniform distribution whose range is shown in the table. The ratio of the actual execution time (AET) to the worst case execution time of a periodic task was changed for each task set to estimate the performance of the algorithms in realistic conditions. This chapter refers to this ratio as accuracy of worst case execution time. For each set of periodic tasks, ten aperiodic task sets were combined. Each aperiodic task set consists of one thousand imprecise aperiodic tasks. The arrival pattern of activation requests for the aperiodic tasks was modeled as a Poisson process and its sum of worst case execution time was modeled as an exponential distribution. Their mean interarrival time and the mean worst case execution time are denoted as t_A and t_C, respectively. The desired response time of each aperiodic task was determined so that the loading factor of aperiodic tasks can be estimated roughly as t_C/t_A.

7.1.2 Results

Figure 7.1, Figure 7.2, and Figure 7.3 show the average acceptance ratio achieved when the actual execution time and the worst case execution time are equal.

Figure 7.1 shows the average acceptance ratio when the processor utilization of all

periodic tasks is 0.8. The system is seldom overloaded when the aperiodic load (t_C/t_A) is 0.1 and begins to fall into overload as the aperiodic load becomes higher. The EDF algorithm that uses TBS shows the highest acceptance ratio when the aperiodic load is 0.1 and 0.2. However, the acceptance ratio steeply drops to 0 with higher aperiodic load, meaning that all activation requests are rejected when the loading factor of the requested aperiodic task is higher than the processor bandwidth allocated to TBS. A similar result is obtained under the SS-OP algorithm. However, the acceptance ratio is lower than 0.7, which are lower than that achieved by the EDF algorithm. On the other hand, when the aperiodic load is 0.1 and 0.2, the acceptance ratio of the M-FWP algorithm is lower than 0.5, which is the lowest ratio among three algorithms. However, the ratio only gracefully decreases under higher aperiodic load and is higher than 0.2 with the highest aperiodic load generated.

Figure 7.2 shows the acceptance ratio when the processor utilization of all periodic tasks are 0.9. The system falls into transient overload in all combinations, despite that the average system load given as the sum of the processor utilization of periodic tasks and the average aperiodic load is still 1 when the aperiodic load is 0.1. The average acceptance ratio under each scheduling algorithm in this case shows the same tendency as that observed in Figure 7.1. The EDF algorithm with TBS shows the highest acceptance ratio when the aperiodic load is 0.1 and drops to 0 under higher load. The SS-OP algorithm shows similar results, but the ratio is lower than the EDF algorithm when the aperiodic load is 0.1. The acceptance ratio of the M-FWP algorithm is lower than the other two algorithms when the aperiodic load is 0.1. However, the ratio only gradually decreases under the M-FWP algorithm and becomes the highest among all algorithms with the aperiodic load higher than 0.1. By comparing Figure 7.2 with Figure 7.1 when their average system loads are the same, it can also be observed that the acceptance ratio is higher when the processor utilization of periodic tasks is lower. For example, Figure 7.1 shows that when the aperiodic load is 0.2, the ratio achieved under the M-FWP algorithm is about 0.4, while Figure 7.2 shows that when the aperiodic load is 0.1, the ratio is smaller than 0.4.

Figure 7.3 shows the acceptance ratio when the processor utilization of all periodic tasks are 1. This condition does not appropriately characterize the transient overload, since the system load is always higher than or equal to 1. Moreover, since the processor utilization is 1, TBS cannot be used for the EDF algorithm. Results obtained under the presented scheduling algorithms in this case again show the same tendency. The acceptance ratio achieved under the M-FWP algorithm gracefully decreases as the aperiodic load becomes higher, while that achieved the under the SS-OP algorithm are almost 0 for all cases of aperiodic load.

Figure 7.4, Figure 7.5, Figure 7.6, Figure 7.7, Figure 7.8, and Figure 7.9 show the average acceptance ratio achieved when the actual execution time and the worst case execution time are different. In these simulations, the scheduler is given the worst case execution time as the requirement for a periodic task and is not aware that actual jobs have shorter execution time.

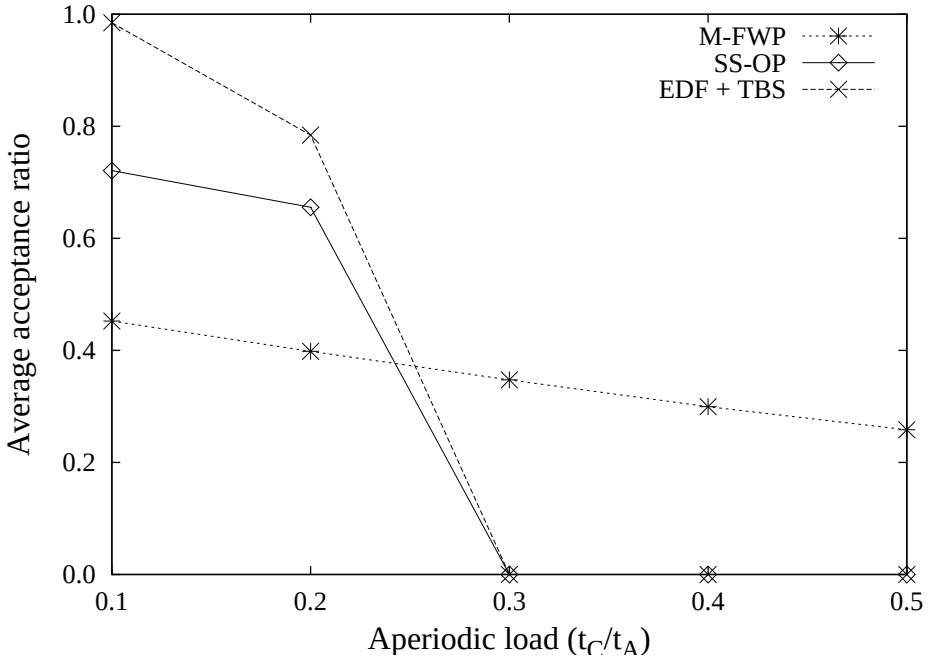


Figure 7.1: Average acceptance ratio (U = 0.8, AET/WCET = 1.0)

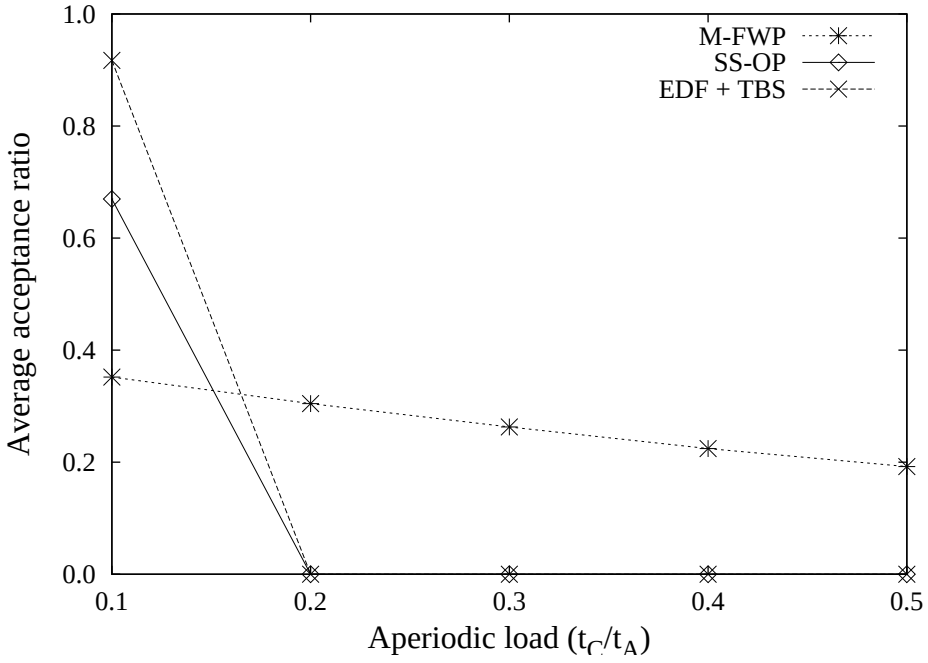


Figure 7.2: Average acceptance ratio (U = 0.9, AET/WCET = 1.0)

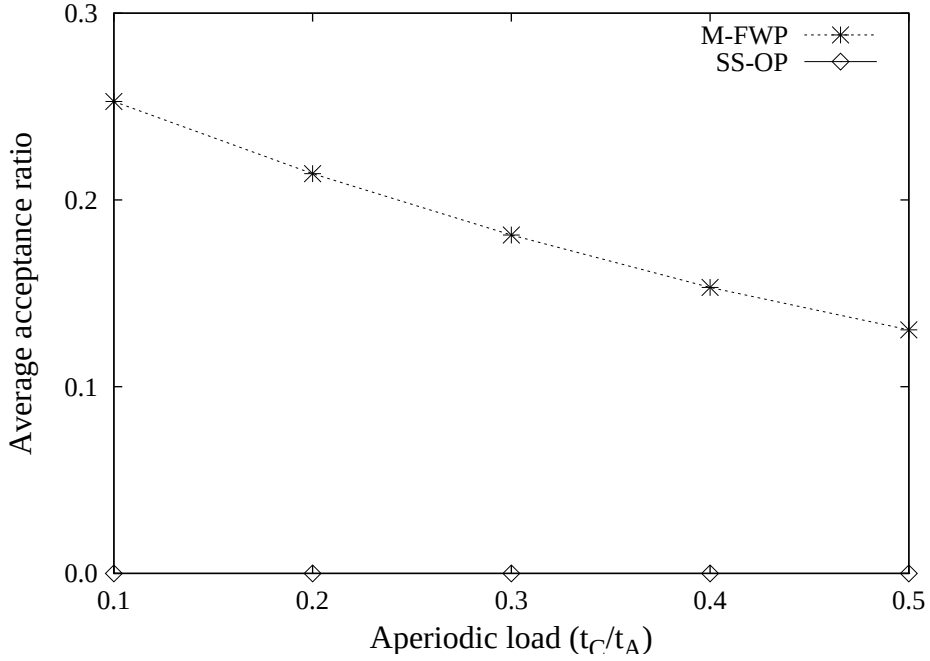


Figure 7.3: Average acceptance ratio (U = 1.0, AET/WCET = 1.0)

Figure 7.4 and Figure 7.5 show the acceptance ratio achieved when the nominal processor utilization U is 0.8 and the accuracy of the worst case execution time is less than 1. The results of simulations conducted when the accuracy of the worst case execution time is 0.8 and 0.9 are shown in Figure 7.4, and the results when the accuracy of the worst case execution time is 0.5 and 0.2 are shown in Figure 7.5. The figures only show one result for the SS-OP algorithm and the EDF algorithm, because the average acceptance ratios achieved under the SS-OP algorithm and EDF algorithm did not change even when the accuracy of the worst case execution time is less than 1. As both Figure 7.4 and Figure 7.5 show, the acceptance ratio achieved under the M-FWP algorithm is higher when the difference between the actual execution time and the worst case execution time is larger. When the aperiodic load is 0.1, the ratio achieved under the M-FWP algorithm is larger than 0.5 in these cases. By comparison, as Figure 7.1 shows, when the accuracy of the worst case execution time is 1, the ratio achieved is lower than 0.5. Moreover, Figure 7.5 shows that when the aperiodic load is 0.2, the acceptance ratio achieved under the M-FWP algorithm is higher than that achieved under the EDF algorithm. Furthermore, the ratio achieved under the M-FWP algorithm only decreases gracefully when the aperiodic load is higher than 0.2 to put the system in heavier overload.

Figure 7.6 and Figure 7.7 show the acceptance ratio achieved when the nominal processor utilization U is 0.9. The results show similar tendency to those shown in Figure 7.4 and Figure 7.5. When the aperiodic load is 0.1, the difference between the

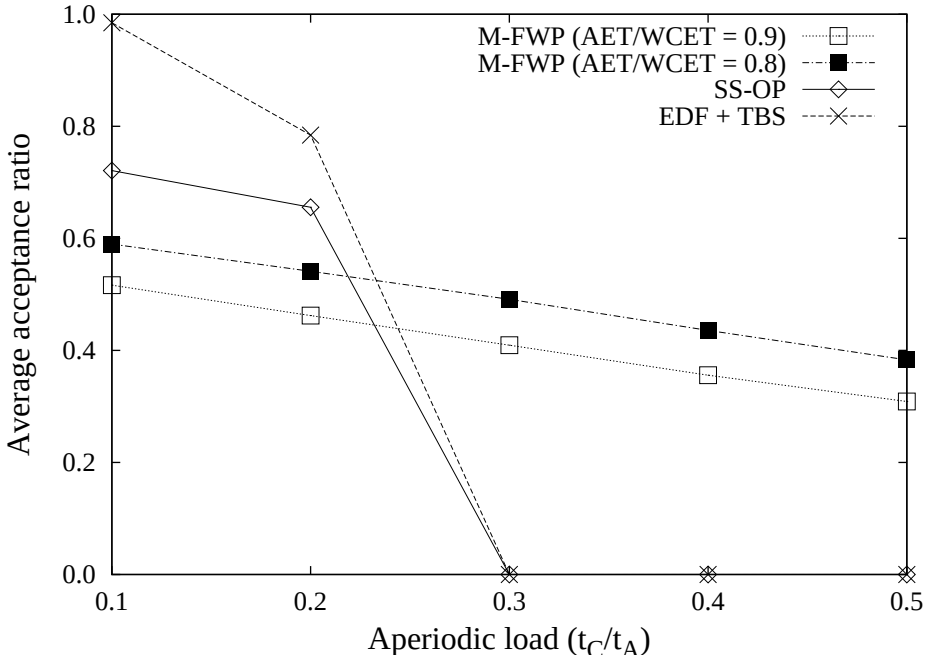


Figure 7.4: Average acceptance ratio (U = 0.8; AET/WCET = 0.8, 0.9)

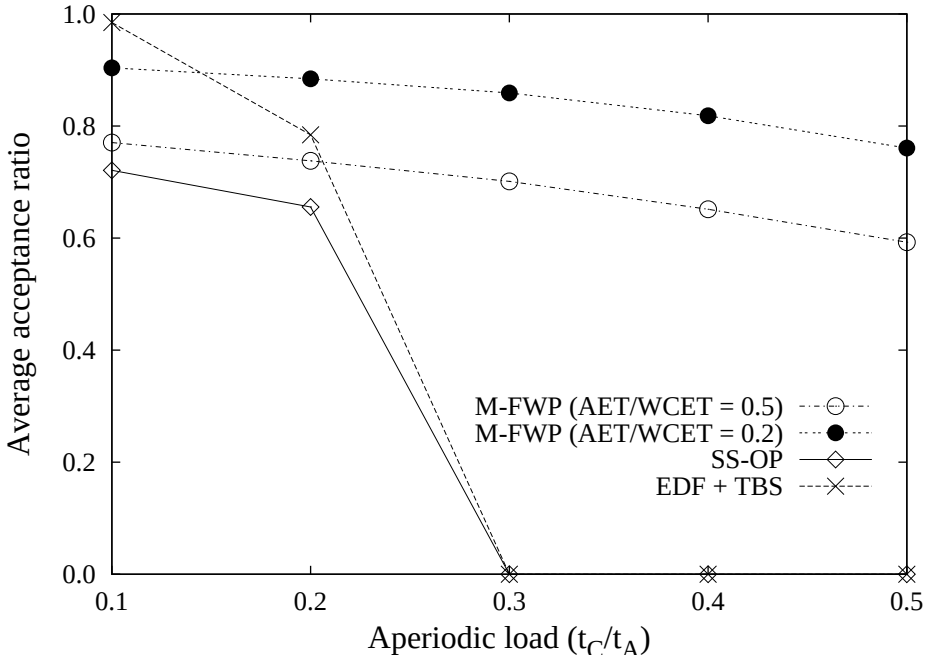


Figure 7.5: Average acceptance ratio (U = 0.8; AET/WCET = 0.5, 0.2)

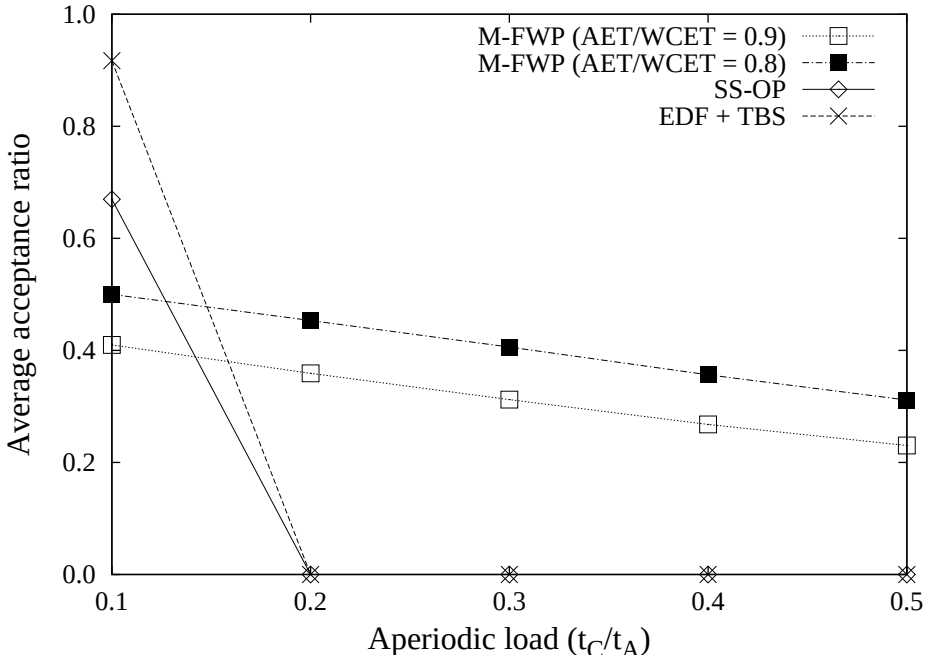


Figure 7.6: Average acceptance ratio (U = 0.9; AET/WCET = 0.8, 0.9)

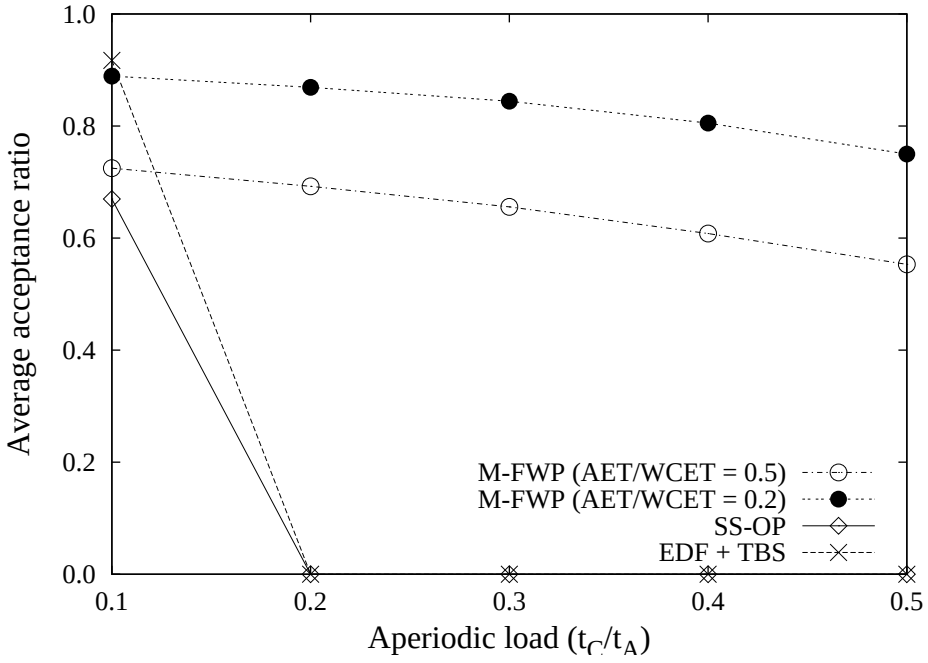


Figure 7.7: Average acceptance ratio (U = 0.9; AET/WCET = 0.5, 0.2)

acceptance ratio achieved under the EDF algorithm and that under the M-FWP algorithm decreases as the accuracy of the worst case execution time becomes lower. When the aperiodic load is higher than 0.1, it is only the M-FWP algorithm that shows graceful degradation in the acceptance ratio to sustain significant acceptance ratios.

Figure 7.8 and Figure 7.9 show the acceptance ratio when the nominal processor utilization U is 1. The acceptance ratio for the EDF algorithm is not shown in this case, since the size of TBS has to be set to zero and thus the EDF algorithm can no longer use TBS to accept aperiodic tasks. The ratio achieved by the M-FWP increases again as the accuracy of the worst case execution time becomes lower. Moreover, the ratio shows graceful degradation when the aperiodic load increases.

7.1.3 Discussion

The EDF algorithm with TBS can theoretically accept all requests when the system is not overloaded. Thus, in Figure 7.1, about 2% of the requests can be said to put the system into overload when aperiodic load is 0.1, and about 20% of the requests must have put the system to overload when aperiodic load is 0.2. Moreover, when aperiodic load is larger than or equal to 0.3, all the aperiodic request lead to overloaded conditions. Similarly, in Figure 7.2, it can be observed that about 10% of the activation requests put the system into overload when aperiodic load is 0.1.

The observed average acceptance ratio is higher when the nominal processor utilization is lower even if the average system loads are the same in the compared cases, because periodic tasks with hard deadline always exist in the system. In other words, their jobs were never rejected in the simulations. By comparison, if activation requests for aperiodic tasks were rejected, the effective system load did not increase. Thus, there remains potentially a larger amount of time for aperiodic jobs when the nominal processor utilization is lower.

Comparing the two presented algorithm with the EDF algorithm when the average system load is less than or equal to 1, the acceptance ratio achieved under the presented scheduling algorithms is lower than that under the EDF algorithm. The EDF algorithm with TBS only rejects an activation request when the request puts the system into overload. By comparison, the acceptance ratio achieved by the M-FWP algorithm was lower than 0.5 in all three cases of different workloads. The SS-OP algorithm achieved higher ratio than the M-FWP algorithm, but still the ratio is lower than that achieved under the EDF algorithm.

The M-FWP algorithm and the SS-OP algorithm have different reasons behind these low acceptance ratios. The primary reason why the acceptance ratio achieved under the M-FWP algorithm is lower than that achieved under the EDF algorithm is that all periodic jobs ready for mandatory part have higher priority than the requested aperiodic job. In other words, the activation request can only be accepted if at least all the periodic jobs ready for mandatory part can complete their mandatory parts in the desired response time of the requested aperiodic task. Another reason is that the amount of idle time

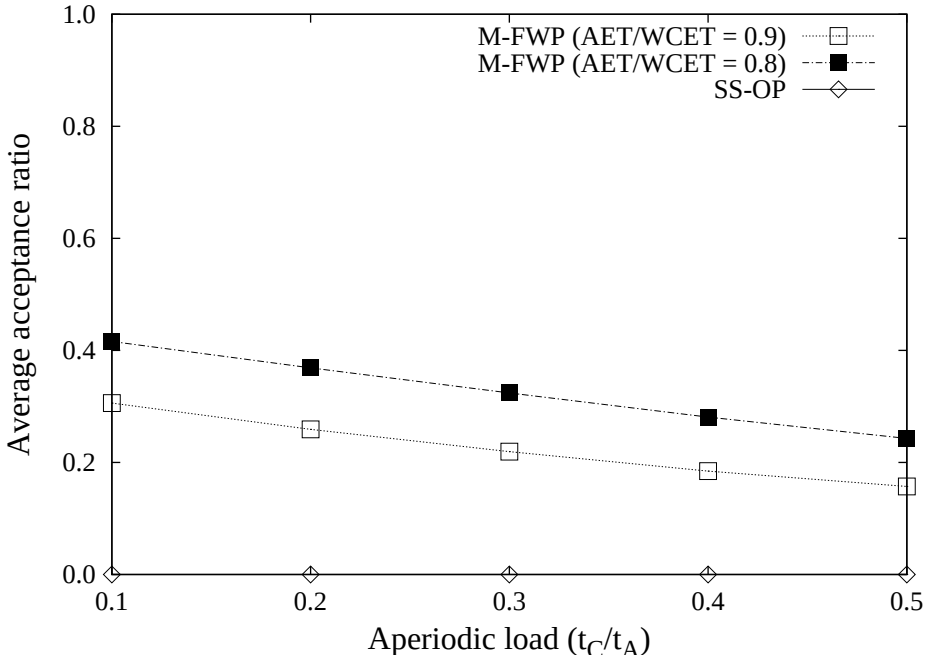


Figure 7.8: Average acceptance ratio (U = 1.0; AET/WCET = 0.8, 0.9)

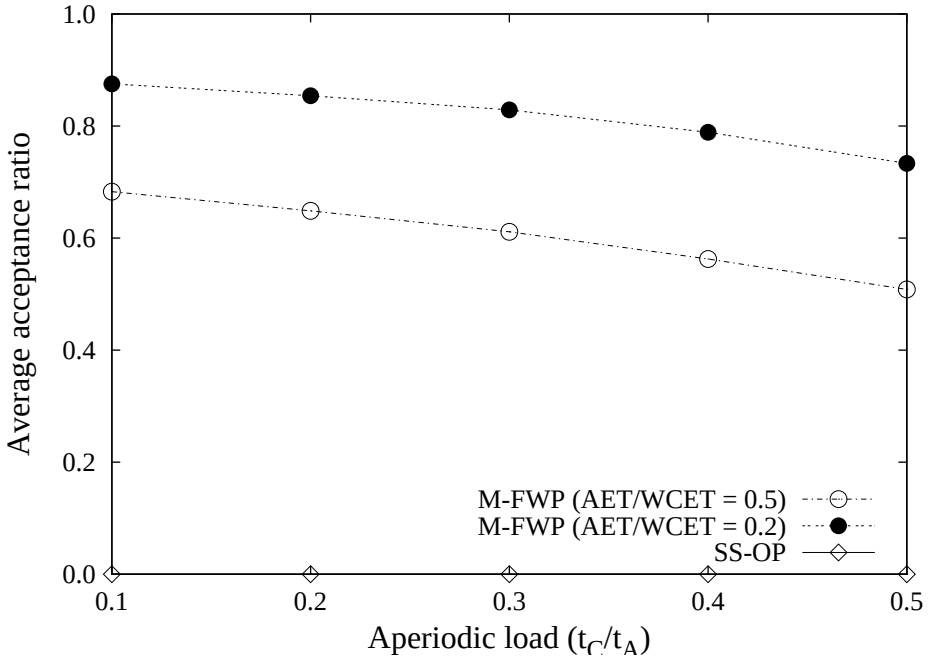


Figure 7.9: Average acceptance ratio (U = 1.0; AET/WCET = 0.5, 0.2)

calculated by the M-FWP algorithm is only an underestimation. In particular, owing to the mandatory first strategy, the time demanded by higher priority jobs are not precisely calculated. This estimation, however, is necessary to suppress the time complexity of the M-FWP algorithm and to make it practicable. On the other hand, the reason why the acceptance ratio achieved under the SS-OP algorithm is lower than that achieved under the EDF algorithm is that the SS-OP algorithm does not consider the slack already allocated to periodic jobs whose optional part has been executed as available to aperiodic tasks.

When aperiodic load becomes higher to put the system in transient overload more frequently, the acceptance ratio achieved under the M-FWP algorithm becomes higher than that achieved under the other two algorithm. Under the M-FWP algorithm, the ratio shows only gradual degradation, because all the idle time allocated to already accepted tasks can be used to accept new requests. This is an advantage compared with the EDF algorithm that can not accept any request when the aperiodic load is higher than or equal to 0.2. On the other hand, the acceptance ratio achieved under the SS-OP algorithm was very close to zero. By examining the cases of the rejection, we found out that this was due to the fact that there were almost always periodic jobs that are executing or that have executed their optional part. Since deallocation of slack from these tasks is not allowed under SS-OP algorithm, it resulted in lower acceptance ratios.

The advantage of the M-FWP algorithm over the other two algorithms becomes more prominent when the actual execution time of periodic tasks are smaller than the given worst case execution time. The acceptance ratio under the M-FWP algorithm increases, while the other two algorithms show no significant difference. The reason why the M-FWP algorithm could use the time left unused by periodic jobs is that the algorithm calculates an amount of idle time directly by considering the interference from other jobs with higher priority. Thus, if periodic jobs completed earlier than expected, the amount of interference always becomes smaller. On the other hand, the SS-OP algorithm uses the notion of processor bandwidth to calculate the amount of slack. Moreover, the computation time left unused by periodic jobs is reclaimed by other periodic jobs for higher QoS. Thus, the amount of slack that can be allocated to jobs of requested aperiodic tasks does not increase. The EDF algorithm with TBS has different reason for the unchanged acceptance ratio. Under the EDF algorithm the processor bandwidth that can be used for aperiodic jobs is statically determined from the nominal processor utilization. Thus, there is no mechanism to reclaim unused computation time for accepting new aperiodic jobs.

In summary, the M-FWP algorithm designed to handle transient overload has higher tolerance to overload caused by aperiodic tasks. Its downside is that the acceptance ratio is lower than necessary when the system is not overloaded. However, its effect is mitigated when the accuracy of the worst case execution time is low.

Table 7.2: Simulation parameters for persistent overload

Parameters	Values
Simulation length	10000000
Number of periodic tasks	10
Type of periodic deadline	Hard
System load (U)	0.8, 0.9, 1.0, 1.1, 1,2, 1.3, 1.4
Period (T_i)	[1000, 50000]
Worst case execution time (C_i)	$0.1U$
Ratio of mandatory parts (m_i/C_i)	0.7
Accuracy of worst case execution time (AET/WCET)	0.2, 0.5, 0.8, 0.9, 1.0

7.2 Performance under Persistent Overload

This research uses three performance metrics to evaluate the scheduling algorithms under the persistent overload.

One of the performance metrics under the persistent overload is the average QoS of the whole system, which is estimated by the average reward accrued in the system. Since the EDF algorithm cannot be used for systems in persistent overload, the average reward obtained by solving a linear programming problem is also considered to show where the optimal line is. Note that since the optimization problems were not characterized as an integer programming problem, which is generally intractable, the calculated average can be larger than the maximum reward that can be obtained in practice. In the following, the results obtained by solving the problem off-line is referred to as *Optimal* results.

The second metrics is the deviation of reward accrued by each task. Besides maximizing the average reward in the system, the reward of each task should be kept stable. In other words, the QoS of a task should not be very high at one moment and then very low at the next moment.

The third metrics is the effective processor utilization achieved. The effective processor utilization is only measured under persistent overload, because it is not adequate to assess the effective processor utilization of a system that undergoes transient overload by measuring the effective processor utilization only under transient overload. On the other hand, the load level of a system that undergoes persistent overload can appropriately be assessed, because persistent overload is more stable than transient overload.

7.2.1 Experimental Setup

The persistent overload in this research occurs due to activation of periodic tasks. Thus, the simulations for this condition use workloads that consist of only periodic tasks. All periodic tasks in the simulations are static, since we are interested in the performance under the persistent overload that usually lasts longer than the transient overload.

In the simulations, workloads were generated using parameters shown in Table 7.2.

Ten periodic task sets were generated for every one of the system loads shown in the table. The definition of the system load in this simulation study is the same as the nominal processor utilization, since there is no aperiodic task. The relative deadline and period of periodic tasks were set to equal, so that the effect of loose worst case execution time on the effective processor utilization can be easily observed. For each case where the actual execution time differ from the worst case execution time, the Optimal result for a task set is calculated using the actual execution time.

For the presented algorithms, the simulations were run with and without the intra-task reclaiming. As was stated, the intra-task reclaiming in this dissertation refers to reclaiming of any unused time in the preceding mandatory part by another optional part of the same task. Since the intra-task reclaiming can only be performed by distinguishing optional parts within a task, the differences in these cases can be used to evaluate another advantage of deploying imprecise computation.

7.2.2 Results

The average rewards accrued in the system are shown in Figure 7.10, Figure 7.11, Figure 7.12, Figure 7.13, and Figure 7.14. In these figures, the results are normalized by the Optimal results calculated off-line. The average reward of 1 means that the results is optimal, which means that the average overall QoS is maximized. The results for the EDF algorithm is only shown when the average load is less than or equal to 1. Moreover, results obtained using the proposed scheduling algorithms without the intra-task reclaiming ability are only shown when the actual execution time differ from the worst case execution time, since otherwise no time can be reclaimed by the intra-task reclaiming ability. The results obtained without the intra-task reclaiming are marked as (w/o intra reclaiming) in the figures.

Figure 7.10 shows the normalized average reward accrued when the actual execution time is the same as the worst case execution time, which means that the accuracy of the worst case execution time is 1. The reward accrued under the EDF algorithm is 1 when the system load is less than or equal to 1. The reward accrued under the M-FWP algorithm is lower than 1 even when the system is not overloaded, and when the system load is 1, the reward accrued is about 0.6. When the system is overloaded, the M-FWP algorithm sustains the normalized average reward between 0.65 and 0.45. The SS-OP algorithm which solves the integer linear programming problem on-line to distribute slack among optional parts of the periodic tasks in a sub-optimal manner keeps the normalized ratio equal to or closed to 1.

Figure 7.11 shows the normalized average reward accrued when the accuracy of the worst case execution time is 0.9. No change is observed for the results under the EDF algorithm. It is still the optimal algorithm when the system load is less than or equal to 1, but it cannot be used with higher load. The M-FWP algorithm, both with and without the intra-task reclaiming, shows gradual decrease, and the reward accrued is higher than that observed in Figure 7.10. Moreover, comparing the results obtained with and

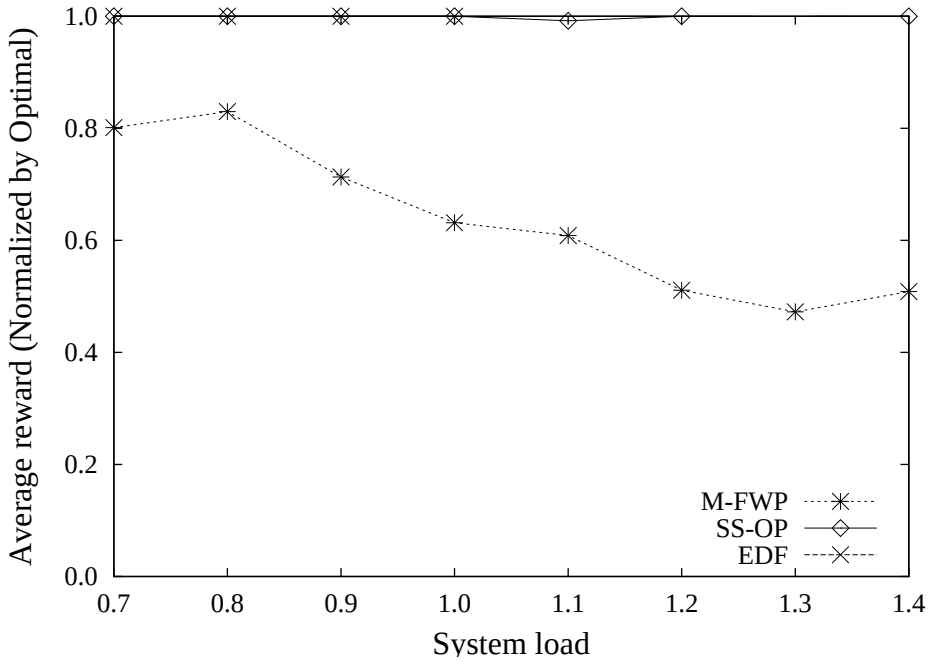


Figure 7.10: Average reward accrued (AET/WCET = 1.0)

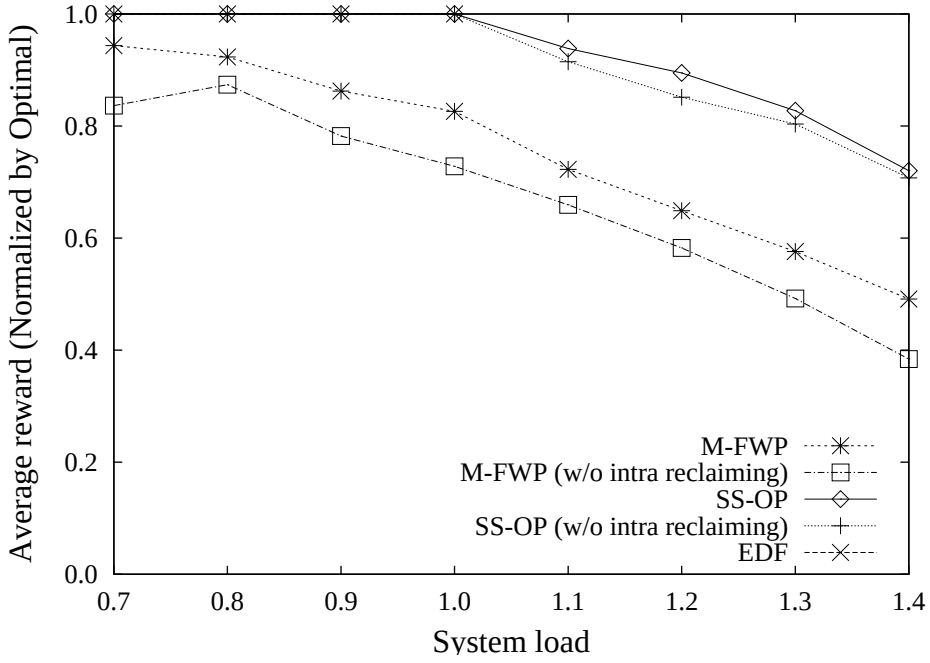


Figure 7.11: Average reward accrued (AET/WCET = 0.9)

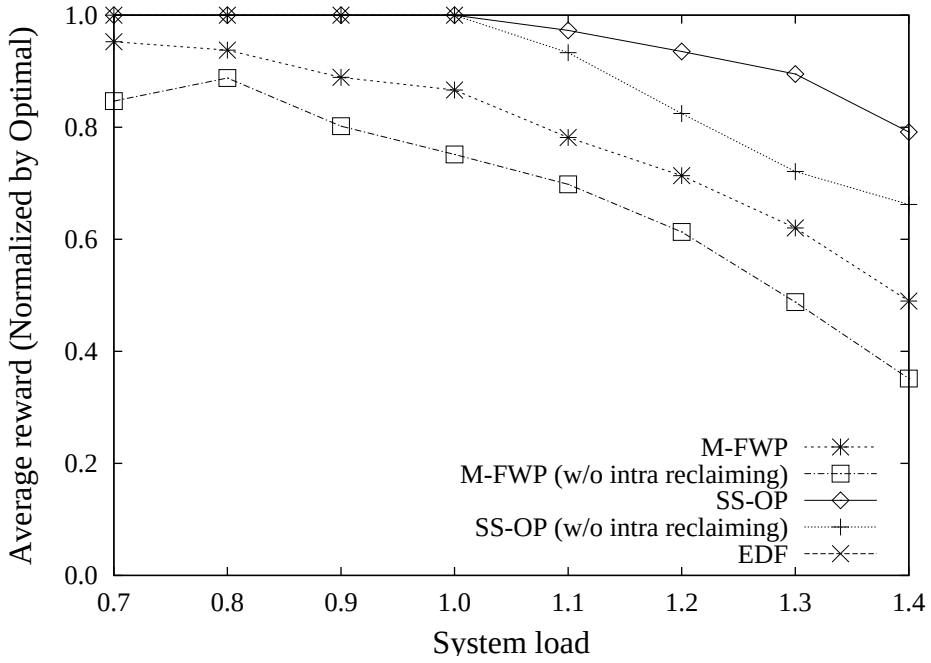


Figure 7.12: Average reward accrued (AET/WCET = 0.8)

without the intra-task reclaiming, the reward accrued with the intra-task reclaiming is always higher by about 10%. By comparison, the reward accrued under the SS-OP algorithm when the system is overloaded dropped to as much as 70% of the Optimal results. However, the reward under the SS-OP algorithm is still higher than that achieved under the M-FWP algorithm and consequently the reward accrued by reclaiming unused time accounts for a smaller ratio under the SS-OP algorithm. Thus, the difference between the reward accrued with and without the intra-task reclaiming is smaller compared with the difference under the M-FWP algorithm.

Figure 7.12, Figure 7.13, and Figure 7.14 show the normalized average reward accrued when the accuracy of the worst case execution time is 0.8, 0.5, and 0.2, respectively. Results obtained in these cases exhibit a similar tendency. The reward accrued under the M-FWP algorithm becomes closer to the Optimal results as the accuracy of the worst case execution time becomes lower, while that accrued under the SS-OP algorithm is higher than 0.8 in most of the cases but not identically close to the Optimal results. Moreover, the results obtained with the intra-task reclaiming is higher than that obtained without the intra-task reclaiming by 10 to 20%. This shows the effectiveness of the intra-task reclaiming ability when the accuracy of the worst case execution time is low. The difference tends to get larger under the M-FWP algorithm as the accuracy of the worst case execution time becomes lower, while the difference under the SS-OP algorithm increases when Figure 7.12 and Figure 7.13 are compared but decreases when Figure 7.13 and Figure 7.14 are compared.

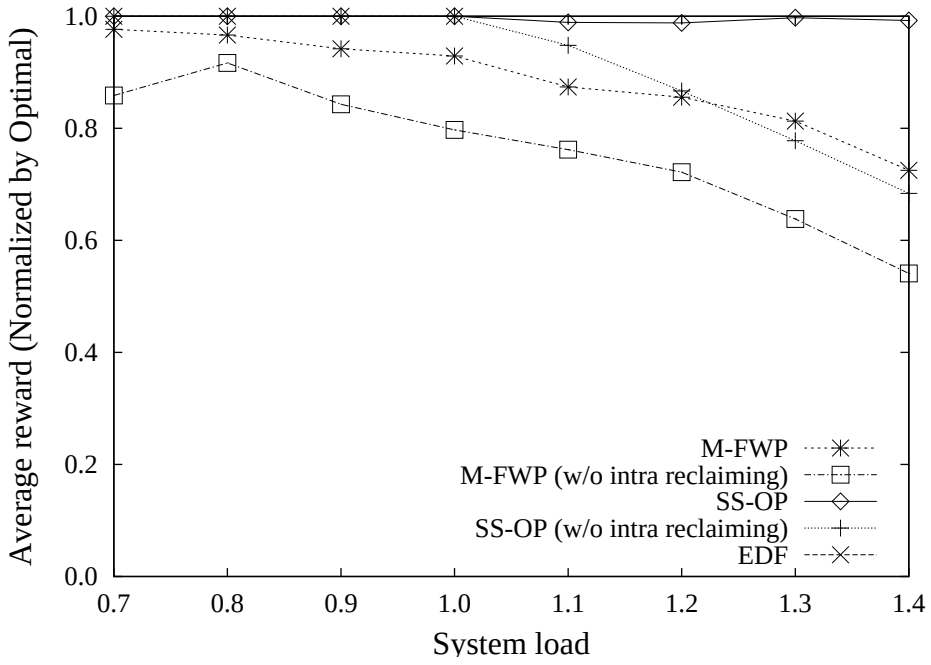


Figure 7.13: Average reward accrued (AET/WCET = 0.5)

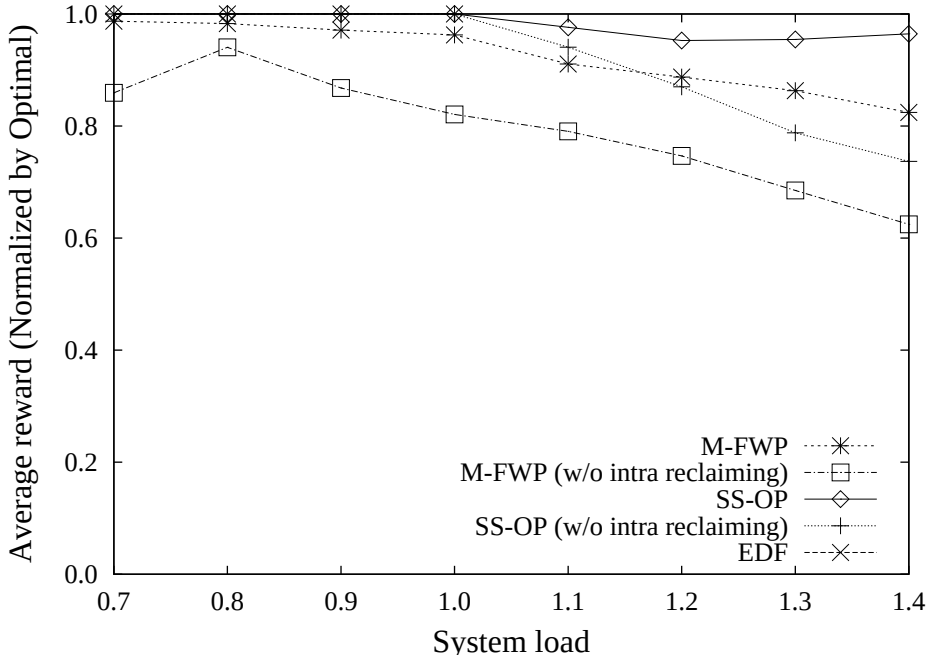


Figure 7.14: Average reward accrued (AET/WCET = 0.2)

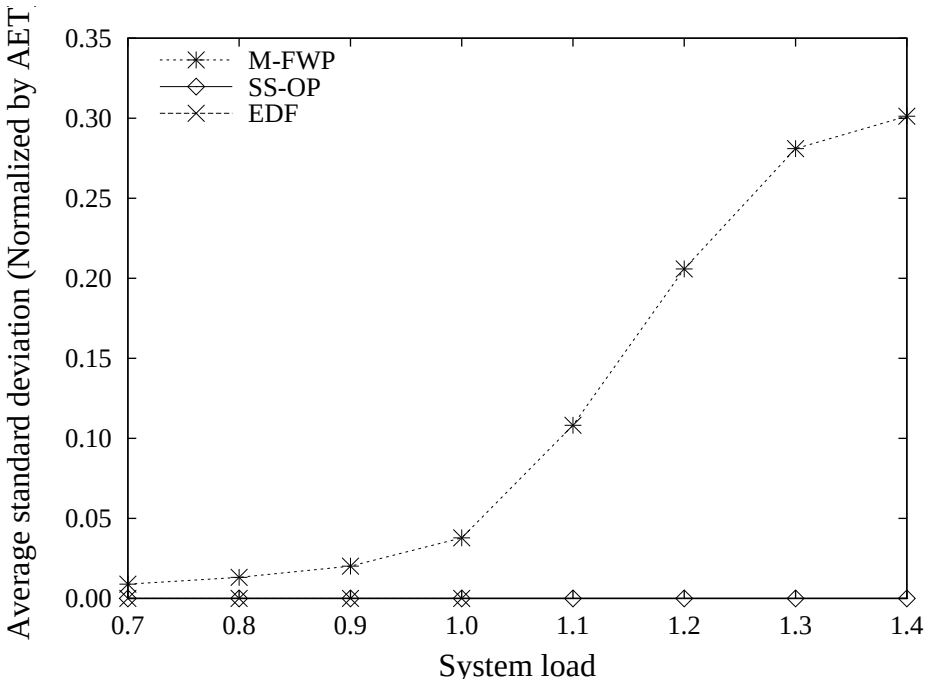


Figure 7.15: Standard deviation of optional computation time (AET/WCET = 1.0)

Standard deviations of computation time allocated to optional parts of a periodic task are shown in Figure 7.15, Figure 7.16, Figure 7.17, Figure 7.18, and Figure 7.19. The results are normalized by the sum of the actual execution time.

Figure 7.15 shows the normalized standard deviation when the actual execution time and the worst case execution time are the same. The standard deviations observed under the EDF algorithm are always 0 when the system is not overloaded. The SS-OP algorithm further succeeds in keeping the standard deviations observed to 0, regardless of whether the system is overloaded or not. By contrast, the standard deviations under the M-FWP algorithm are less than 0.05 when the system is not overloaded, but increases steeply when the system load become higher than 1. The largest standard deviation is measured under the M-FWP algorithm when the system load is 1.4.

Figure 7.16 and Figure 7.17 show the normalized standard deviation when the accuracy of the worst case execution time is 0.9 and 0.8, respectively. By comparing Figure 7.16 and Figure 7.17 with Figure 7.15, the followings can be observed. First, there is no change in the results obtained under the EDF algorithm. They are always zero. Second, the standard deviation obtained under the M-FWP algorithm is less than 0.05 when the system load is less than or equal to 1, but increases as the system load becomes higher. The absolute value of the deviation, however, decreases as the accuracy of the worst case execution time becomes lower. Third, the deviation under the SS-OP algorithm shows different tendency from that observed in Figure 7.15. The deviation stays 0 when the load is lower than 1. With higher system load, the standard deviation

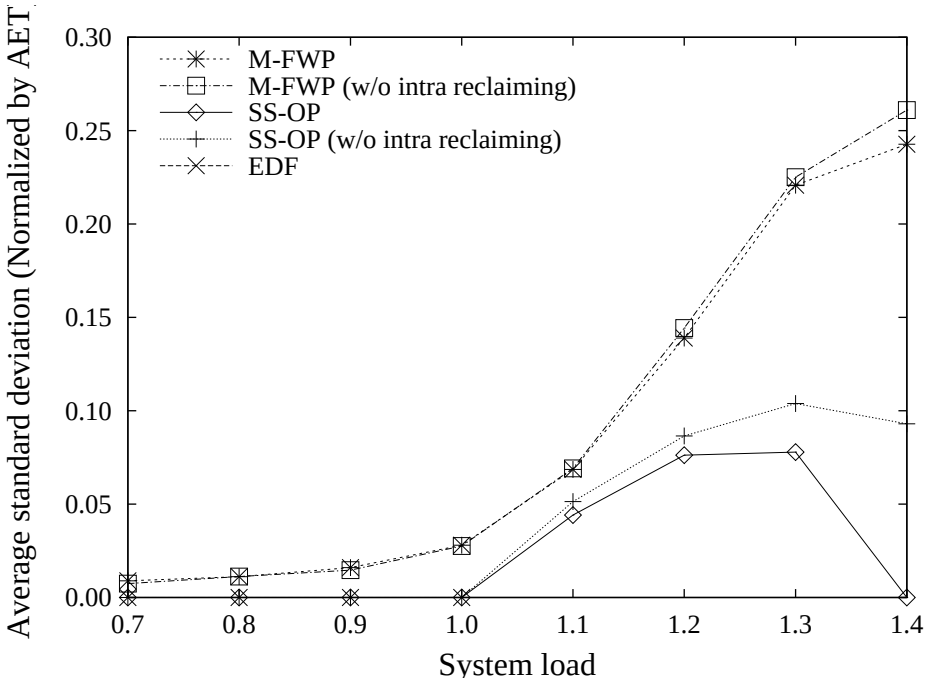


Figure 7.16: Standard deviation of optional computation time (AET/WCET = 0.9)

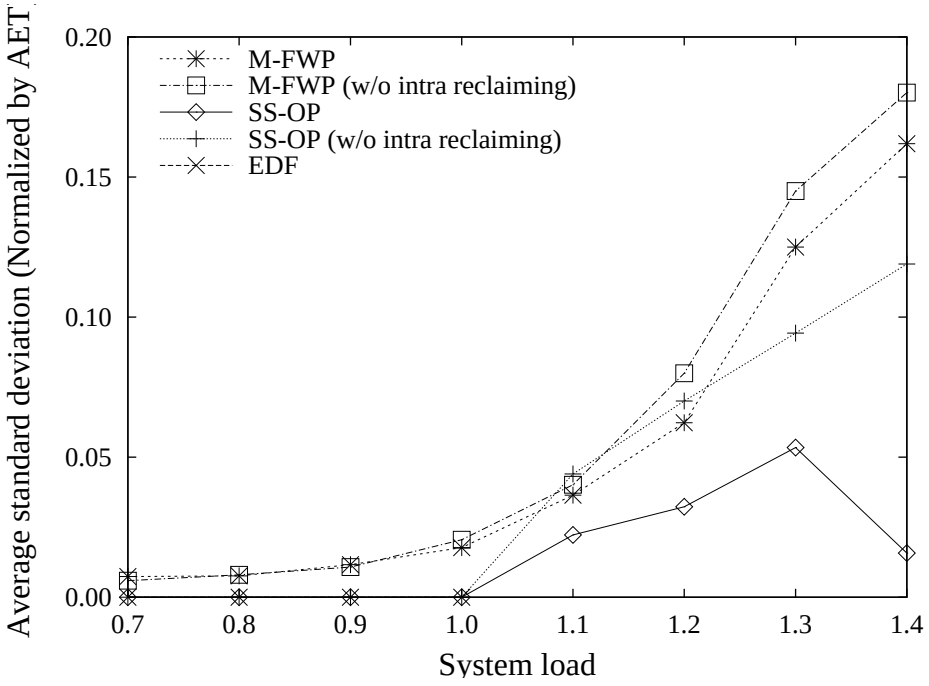


Figure 7.17: Standard deviation of optional computation time (AET/WCET = 0.8)

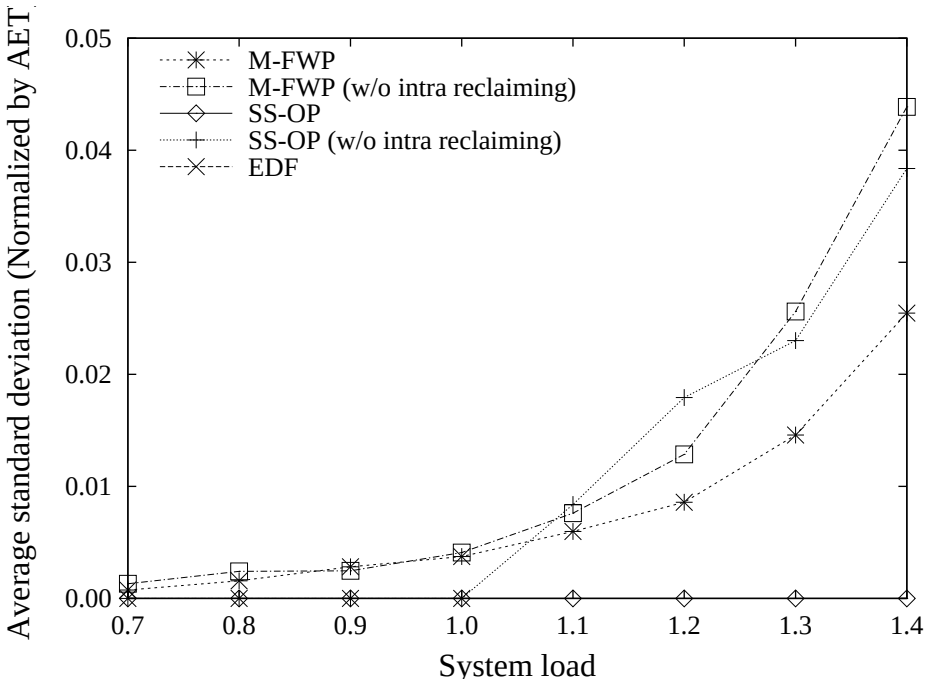


Figure 7.18: Standard deviation of optional computation time (AET/WCET = 0.5)

increases once, but then it decreases afterward. The largest standard deviation is observed when the system load is 1.2 and 1.3 in Figure 7.18 and Figure 7.19, respectively. Fourth, comparing the same algorithm with respect to the intra-task reclaiming ability, the standard deviation is smaller when the reclaiming is performed.

Figure 7.18 and Figure 7.19 show the normalized standard deviation when the accuracy of the worst case execution time is 0.5 and 0.2, respectively. The deviations under these two cases show a similar tendency. The normalized standard deviations under the EDF algorithm and the SS-OP algorithm are both 0. The deviations under the M-FWP algorithm and the SS-OP algorithm without the intra-task reclaiming ability increase as the system load gets higher. Moreover, the deviation for the SS-OP algorithm without the intra-task reclaiming tends to be larger than the M-FWP with the intra-task reclaiming. However, the absolute value is less than 0.1 in both cases. Hence, the results shown in these figures is more similar to those shown in Figure 7.15 than to those shown in Figure 7.16 and in Figure 7.17.

Finally, the average effective processor utilization achieved are shown in Figure 7.20, Figure 7.21, Figure 7.22, Figure 7.23, and Figure 7.24. As Figure 7.20 shows, the EDF algorithm and the SS-OP algorithm maximizes the effective processor utilization when the worst case execution time is precise. The effective utilization achieved by the M-FWP algorithm is lower than that achieved by the EDF algorithm when the system is not overloaded, but keeps increasing the effective utilization until it reaches 1 as the system load increases. By comparing the other figures, we can make the following ob-

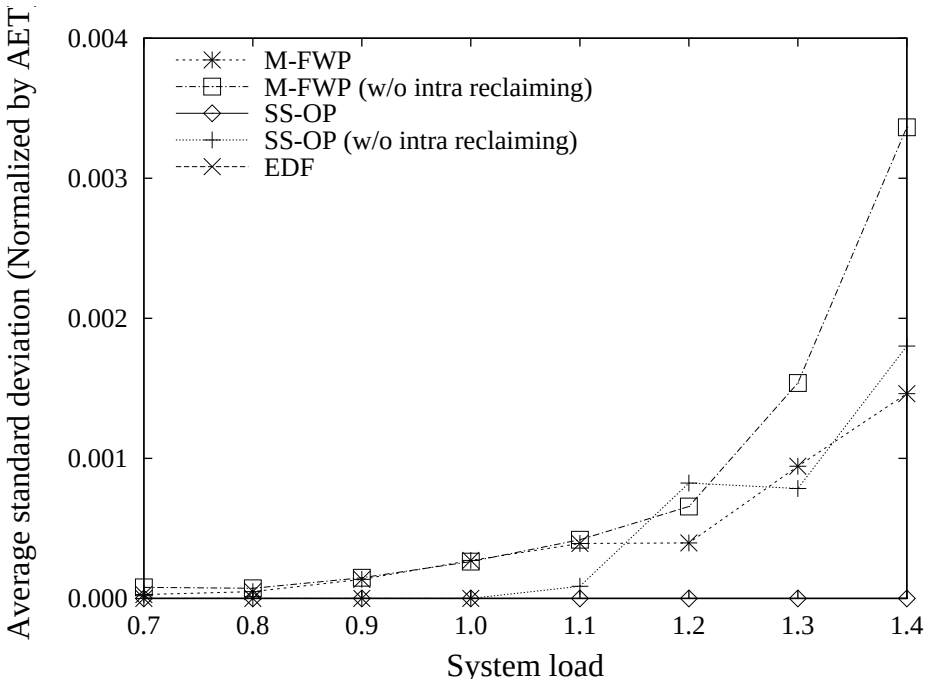


Figure 7.19: Standard deviation of optional computation time (AET/WCET = 0.2)

servations. First, the maximum effective utilization achievable by the EDF algorithm decreases as the accuracy of the worst case execution time becomes lower, since the EDF algorithm can only be used when the system load calculated using the worst case execution time is less than or equal to one. Second, both of the presented scheduling algorithms achieve higher effective processor utilization when the system is nominally under overload. Third, the SS-OP algorithm achieves higher effective utilization than the M-FWP algorithm. Fourth, intra-task reclaiming always increases the effective processor utilization.

7.2.3 Discussion

Important differences between the M-FWP algorithm and the SS-OP algorithm are revealed by this second simulation study.

The average reward accrued in the system for the M-FWP algorithm keeps increasing when the accuracy of the worst case execution time becomes lower. This is easy to understand as the direct consequence of loose estimation is that the actual system load is lower than claimed. Since the M-FWP does not use the notion of processor bandwidth nor utilization to calculate the amount of optional computation time, the total amount of time allocated to optional parts increases.

Under the SS-OP algorithm, on the other hand, the normalized average reward accrued first drops at most to 0.7 and 0.8 when the accuracy of the worst case execution

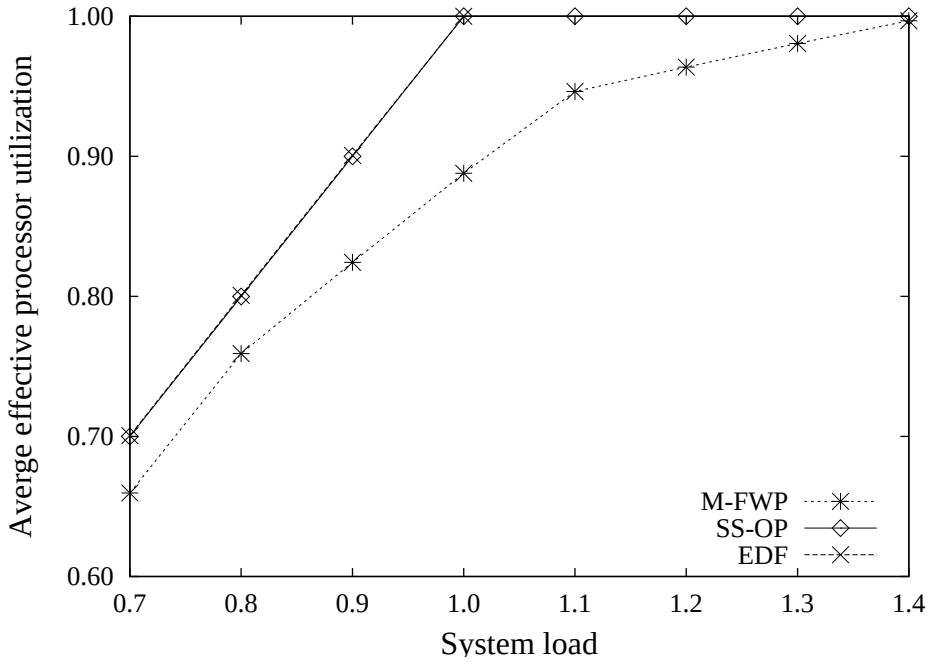


Figure 7.20: Effective processor utilization (AET/WCET = 1.0)

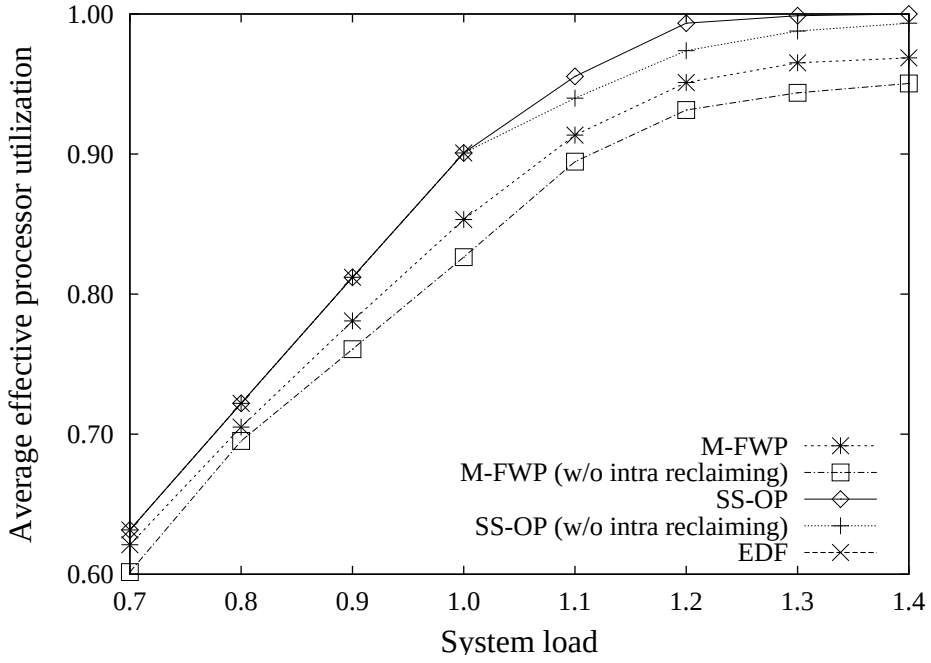


Figure 7.21: Effective processor utilization (AET/WCET = 0.9)

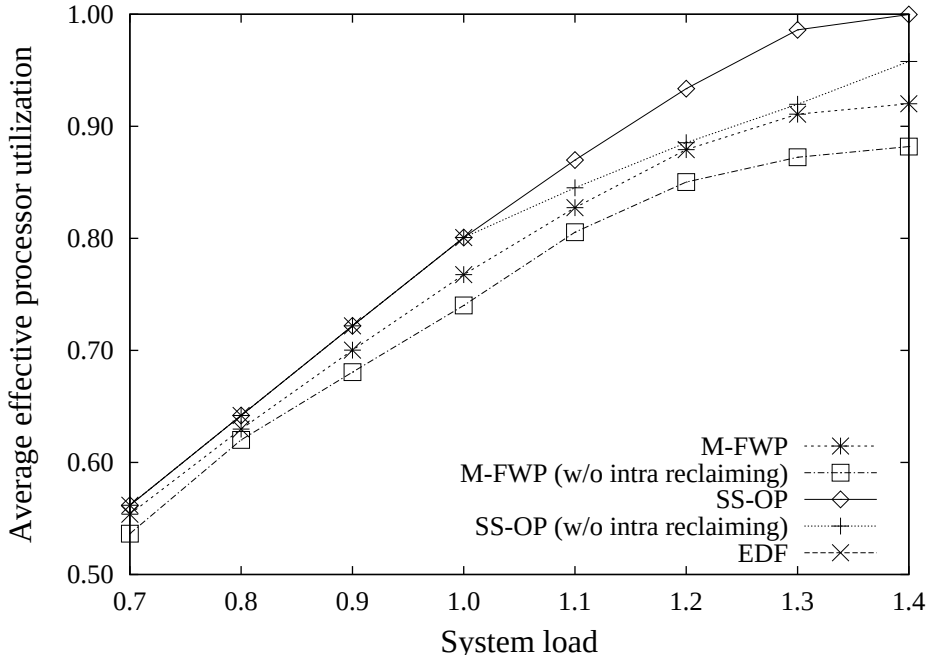


Figure 7.22: Effective processor utilization (AET/WCET = 0.8)

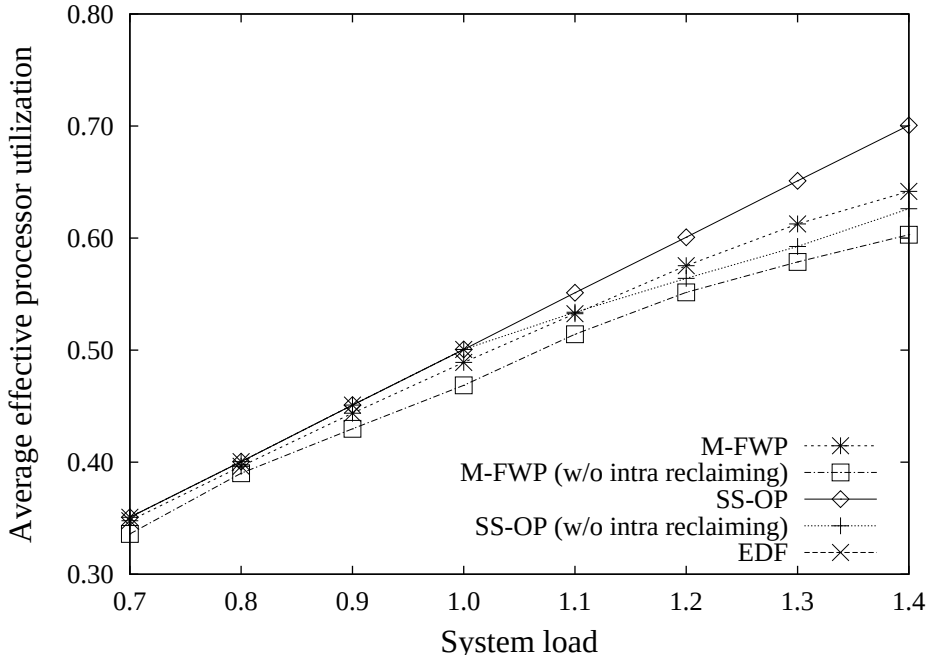


Figure 7.23: Effective processor utilization (AET/WCET = 0.5)

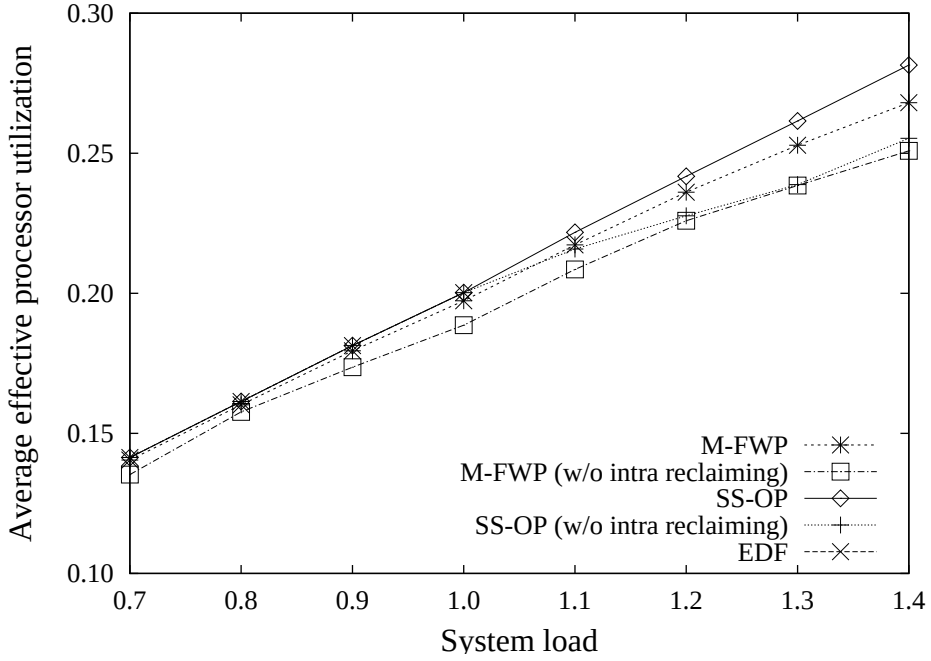


Figure 7.24: Effective processor utilization (AET/WCET = 0.2)

time is 0.9 and 0.8, respectively, but it again becomes larger than 0.95 when the ratio is 0.5 and 0.2. This behavior can be explained by considering the QoS controller used in the SS-OP algorithm. As the accuracy of the worst case execution time becomes lower than 1, the slack distribution calculated by the QoS controller goes farther from the optimal solution, since the QoS controller is not aware of the actual execution time. In other words, the slack distribution can only be close to optimal by using the actual execution time in characterizing the optimization problem. Thus, the normalized average reward decreases as the accuracy of the worst case execution time becomes lower, until the accuracy reaches 0.8 in this simulation. When the accuracy of the worst case execution time becomes even lower, the actual system load further decreases so that the system is no longer overloaded. Thus, by using the reclaimed unused time to complete more and more optional parts, the reward again reaches close to the optimal level. This consideration also explains why the difference between the reward accrued under the SS-OP algorithm with the reclaiming and without the reclaiming once increases and then decreases as the accuracy of the worst case execution time becomes lower.

Another important difference is the stability of the optional computation time allocated to jobs of a periodic task. The optional computation time changes from job to job under the M-FWP algorithm, irrespective of the accuracy of the worst case execution time. Moreover, the deviation increases as the system load becomes higher, since a higher system load can cause a larger portion of optional parts to be discarded. On other other hand, under the SS-OP algorithm, when the worst case execution time is precise,

a constant optional computation time is allocated to all jobs of a periodic task, thanks to the QoS controller. When the accuracy of the worst case execution time becomes lower, the deviation of the optional computation time allocated under overload increases. This is because the QoS controller cannot allocate enough amount of slack to all periodic tasks in a sub-optimal manner. Consequently, the scheduler has to dynamically adjust the slack distribution by reclaiming unused time. Since SS-OP algorithm is a dynamic priority scheduling algorithm, the unused time is likely to be reclaimed by jobs of different tasks. The remedy is the intra-task reclaiming ability. Since the ability allows the unused time to be reclaimed by the same task, it mitigates the affect of low accuracy of the worst case execution time. The above also explains why the deviation increases once and then decreases in Figure 7.16 and Figure 7.17. As for the case shown in Figure 7.17, the system becomes actually overloaded only when the nominal system load is 1.3. Thus, with lower system load, some tasks have too much time allocated to its optional parts, causing a large amount of time to be reclaimed by other tasks. On the other hand, with the system load higher than 1.3, the ratio of tasks that have enough optional computation time decreases, and thus most of the unused time can be reclaimed by the same task. Hence, the deviation decreases.

Finally, comparing the presented algorithms with respect to the effective processor utilization, the SS-OP algorithm is superior to the M-FWP algorithm, although both algorithms succeeded in increasing the effective utilization even after the system becomes nominally overloaded. The reason why the SS-OP algorithm can achieve higher effective utilization is that the slack is distributed to optional parts in a controlled manner to increase the QoS of the system. Since the QoS in terms of reward only increases when larger portions of optional parts are executed, the QoS controller leaves no slack unallocated unless it breaks the rule of allocating the same optional computation time to all jobs of the same task. On the other hand, the M-FWP leaves the execution of optional part as late as possible. This disables efficient calculation of idle time in the schedule. Thus, the M-FWP algorithm performs an estimation that leaves unclaimed idle time in the schedule, degrading the effective processor utilization.

7.3 Summary

This chapter presented the results of simulation studies conducted by running the RT-Frontier operating system in the simulator mode. Under transient overload, the acceptance ratio of activation requests was measured, while under persistent overload, the average reward accrued in the system, its standard deviation, and the effective processor utilization were measured.

The scheduling algorithms developed in this research both showed tolerance to overloaded conditions and succeeded in increasing the effective processor utilization. By comparison, the EDF algorithm provided the best performance when the system was not overloaded. However, the EDF algorithm does not distinguish optional portions within tasks and cannot provide any guarantee if it is used in systems that may fall into

overload. Moreover, as was expected, when the accuracy of the worst case execution time becomes lower, the effective processor utilization decreased. Hence, the EDF algorithm is not adequate for scheduling overload. By contrast, when the estimation of the worst case execution time is loose, the M-FWP algorithm succeeded in increasing the average acceptance ratio of activation requests for aperiodic tasks under transient overload, and the SS-OP algorithm succeeded in increasing the average QoS of periodic tasks under persistent overload.

The conclusions drawn from the results obtained in the simulation studies are listed in the following.

- The M-FWP algorithm has higher ability to accept dynamic activation of aperiodic tasks. This is due to its higher ability to collect unclaimed time for requested aperiodic tasks. This ability enables the M-FWP algorithm to tolerate the transient overload, since the transient overload is triggered by activation of aperiodic tasks. Hence, the M-FWP algorithm is more suited to systems that undergo transient overload.

- The SS-OP algorithm has higher ability to control the optional computation time of tasks. This ability leads to higher QoS of the system. Moreover, the QoS of each task is less sensitive to the loose estimation of worst case execution time. Furthermore, the effective processor utilization achieved is close to the maximum possible. Hence, the SS-OP algorithm is more suited to systems that undergo persistent overload.

- Both algorithms have tolerance to overloaded conditions and attain high effective processor utilization. However, their performance characteristics and tolerance to dynamic changes in workloads are different. Therefore, these algorithms complement each other to handle different overloaded conditions.

CHAPTER 8

IMPLEMENTATION IN THE RT-FRONTIER OPERATING SYSTEM

This chapter describes the implementation of the two proposed scheduling algorithms in the RT-Frontier operating system. The RT-Frontier operating system is designed for embedded real-time computing systems and is developed from scratch in this research. The RT-Frontier operating system is implemented on a *Responsive Processor* [79]. The processing core of the Responsive Processor is a SPARClite, which is a 32-bit SPARC-V8E compliant processor designed for embedded systems.

There are several issues to consider prior to implementing the presented scheduling algorithms.

At the very least, a real-time operating system, in particular its kernel, must manage the system time so that the scheduler can release jobs at specified times. Moreover, the kernel must also manage the amount of time spent in both parts of real-time jobs, so that an overrun caused in an optional part is detected at the moment when optional computation time is exhausted. Otherwise, delayed termination of optional parts can cause an overflow in the schedule.

Terminating a job in an optional part and making it resume in another part requires that the context of every imprecise thread be managed by the kernel. This issue is not trivial when the imprecise computation model with wind-up operations 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 of embedded operating systems is that the kernel should have the ability to allow users to easily change configurations for different situations. This is half accomplished by defining the uniform scheduling interface in Chapter 4. The interface allows the system designer to switch the scheduling algorithms without rebuilding applications at all. The remaining half is the extendibility of the kernel for some new scheduling algorithms and the portability to some new hardware

platforms. This requires that the kernel clearly distinguish the part that is dependent to the scheduling algorithm and to the hardware used from the other independent parts.

It is also requested that, as was stated in Chapter 3, the whole operating system have enough time predictability to fully utilize the capability of imprecise computation to cope with uncertain workloads. To this end, the RT-Frontier kernel is designed to be preemptive. This means that all the system calls including those required to implement the scheduling algorithms are preemptive whenever possible. Moreover, all the threads inside the kernel are scheduled preemptively with other application threads.

Finally, the implementation should be effective. In other words, the scheduling algorithms must be implemented in low overhead. Although low overhead is commonly desired in all software, the requirement is especially important in the RT-Frontier operating system targeted at embedded systems. In particular, the RT-Frontier operating system avoids floating point operations for this purpose. In the theoretical world, the utilization of a processor is always considered in the range of $[0, 1]$. However, many embedded processors do not have a floating point unit. Moreover, even in a system that has a processor with a floating point unit, it is often desired not to use the unit inside the kernel, so that the status of the floating point unit does not need to be saved every time the kernel is executed. Therefore, the RT-Frontier operating system emulates any floating point operations theoretically required to implement scheduling algorithms by integer operations. In particular, the processor utilization is held in a 32-bit integer variable and the value is maintained in the range of $[0, 4096]$. Thus, the utilization is expressed in a fixed point number with the accuracy of $1/4096$.

8.1 Overview of RT-Frontier

The overall structure of the RT-Frontier operating system is illustrated in Figure 8.1. In the RT-Frontier operating system, the only active entities are threads, while address spaces and chunks of memory are allocated to tasks. Thus, there is at least one thread per task. Inside the kernel, there are *system service threads*. These threads are dedicated to providing specific kernel services to applications. An example of their roles is to manage real-time channels [80] between different nodes. A real-time channel is a connection between two threads on different nodes with timing constraints. When establishment of a real-time channel is requested, a dedicated service thread is invoked. The thread finds a route to the destination node that can meet the constraints by communicating with other threads of the same type on other nodes. Unlike many implementations of microkernel based operating systems where service providers are implemented as separate user level tasks, these threads reside inside the kernel space and run in the supervisor mode.

There are also services of the RT-Frontier operating system that do not require an active entity. These services are implemented as passive library functions. These functions can be either linked against application programs or built into the kernel, depending on whether privileged execution is required. Some of the most important services that are

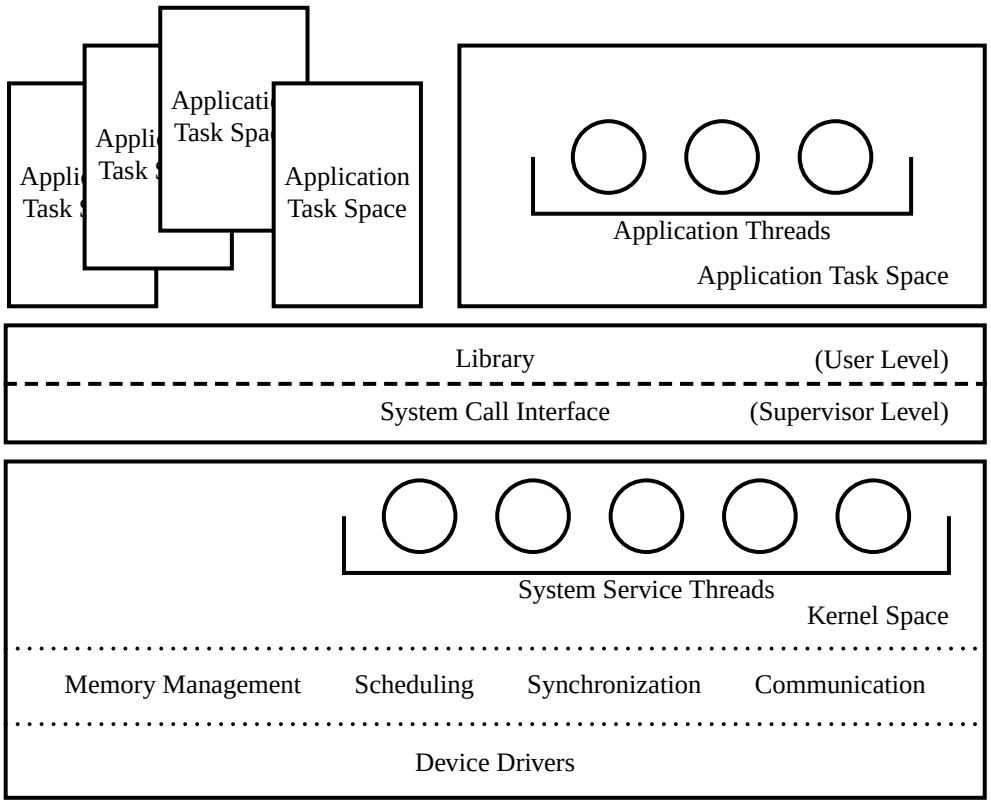


Figure 8.1: Structure of the RT-Frontier operating system

always provided are: memory management, scheduling, synchronization, and communication between threads within the same node. The memory in the system is partitioned into a region statically allocated to the kernel and one that is dynamically allocated. The region of memory used for dynamic allocation is all paged. The scheduling algorithm implemented in the kernel are the two scheduling algorithms presented in this dissertation and the EDF algorithm. Moreover, it supports a static priority scheduling using 256 levels of priority in the background of a dynamic priority scheduling algorithm. The synchronization is closely related to the scheduling algorithm and the SRP protocol is used by default. The communication between threads can use either event messages or state messages [81].

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, that is, 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.

8.2 Time Management

The system time in the RT-Frontier operating system is maintained using one of the timers that the Responsive Processor provides. The time resolution of the timer is 0.05μs. In other words, the length of the shortest interval that can be measure by the timer is 0.05μs. As soon as the interrupt controller is set up in the boot-up stage, the timer is set to generate an interrupt periodically. The period of the interrupt or the tick rate can be changed only statically. On every interrupt, the kernel increments the system time maintained as an integer variable.

Interrupts are handled in two phases. In the first phase, an interrupt handler is invoked. The interrupt service routine executed by the corresponding handler returns an identifier that indicates whether a further activity is required for the generated interrupt. If it is required, a thread that implements the required activity is scheduled in the second phase. The thread triggered by an interrupt is implemented as a sporadic thread in the RT-Frontier operating system. The differences between the work done in the first phase and that in the second phase are that the former is executed with all interrupts disabled and that the former cannot sleep or be preempted.

Figure 8.2 illustrates how the system timer interrupt is handled. It assumes that the user level thread was running when the timer has expired. When an interrupt is generated, the thread enters the kernel model. The first thing to do on any interrupt is to save all registers that might be clobbered before the thread resumes back to the user level. This operation is performed in the part marked as *Entry* in the figure. Then, the interrupt handler is called. The interrupt handler for the system time management only clears the interrupt and increments the system time. There is no need to set the system timer again, since the timer provided by the Responsive Processor does not need to be reset once it is put to the periodic mode. The handler also returns a flag that indicates there is no thread to wake up in the second phase but that the scheduler must be called on return. The scheduler is guarded by a lock in the kernel to achieve mutual execution, because interrupts are enabled during scheduling and the scheduling codes can be executed by the system service threads. In Figure 8.2, the scheduler lock is obtained in the part marked as *Check*. If the lock is obtained, the scheduling code is executed. If a context switch occurs, the context of this thread is saved on the stack and the context of the newly running thread is restored at the end of the part marked as *Scheduling*. After that, the scheduler lock is released and the saved registers are restored in the part marked as *Exit*. If the scheduler lock cannot be obtained, the lock is marked as *requested*, so that the scheduler is invoked at the time the lock is released. Then, the thread executes *Exit* code and resumes to the user level.

The RT-Frontier kernel also keeps track of the amount of time spent by every thread.

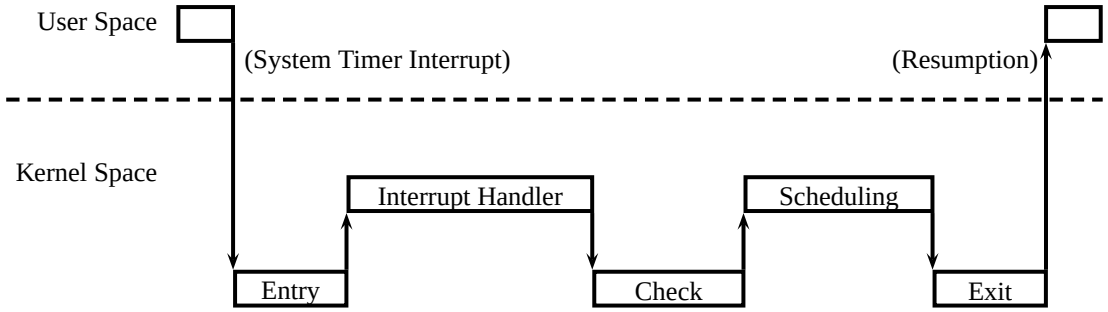


Figure 8.2: Handling of system timer interrupts

The time is separately managed for mandatory parts and optional parts. This execution time management is accomplished by using another timer in a Responsive Processor. This dissertation refers to this second timer as overrun timer. The overrun timer is started when a thread begins execution. Every time the overrun timer is started, it is set to expire when the computation time allocated to the executed part is exhausted. Specifically, if the thread will execute an optional part, the timer is set to expire if its optional computation time is exhausted before completion. When the overrun timer expires, an interrupt handler is invoked. The role of the interrupt handler is simpler than that for the system timer. The handler just clears the interrupt and returns the flag that requests invocation of the scheduler. Then, the scheduler terminates the optional part and modifies the context of the thread. When the thread completes the optional part within its optional computation time, the overrun timer is stopped. The overrun timer is also stopped when the thread is preempted. When the overrun timer is stopped, the value of the counter is read to update the amount of remaining computation time of the running thread.

8.3 Thread Context Management

On the RT-Frontier operating system, all wind-up operations included in mandatory parts are executed in the user space where their preceding optional parts were executed. This is more desirable than executing the wind-up operations in the kernel space, since a wind-up operation may need to modify data structures or send results that are stored in the original task space.

When an optional part is terminated, the context of the executing thread is modified by the kernel so that the thread can execute the successive mandatory part when it resumes execution after an interrupt from the overrun timer. The best the kernel can do on termination of an optional part is to keep a snapshot of the context beforehand and roll back the context of the overrunning thread with the one previously stored, since it is impossible, even by the kernel, to create a complete new context for an unreached point in a program.

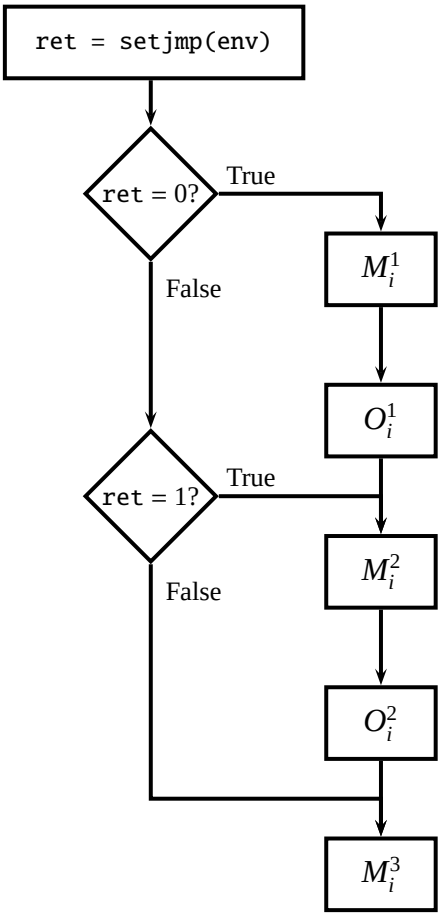


Figure 8.3: Structure for implementing imprecise threads

The RT-Frontier operating system assumes that the structure shown in Figure 8.3 be taken by imprecise threads to workaround this problem. The context of an imprecise thread is saved by a call to the `setjmp()` function before doing any significant work. The saved context will be used to roll back the context of overrunning thread on termination. A pointer to the saved context is stored in the thread control block so that the kernel can find the context with little overhead when an optional part must be actually terminated. The good point about the `setjmp()` function is that it has a return value that gives a hint on which of the execution paths be taken. By taking advantage of this specification, the return value of the `setjmp()` function is used to indicate which part to execute after resumption. Specifically, the return value is set to 0 on the direct call and to j by the kernel on resuming from termination of an optional part O_i^j. For example, the return value of the `setjmp()` function becomes 1 when the optional part O_i^1 is terminated.

One drawback of using `setjmp()` in the beginning is that all threads must begin

their execution from a mandatory part, since a thread that begins with an optional part cannot jump to a mandatory part if it was terminated before it makes call to `setjmp()`. Thus, on the RT-Frontier operating system, a thread can only begin its execution from a mandatory part and the overhead of `setjmp()` is included in the worst case execution time of the first mandatory part. However, this overhead should not be of any problem, since `setjmp()` is often not a system call and its overhead is relatively small compared with that of system calls.

8.4 Modular Scheduler Architecture

The scheduling algorithms presented in this dissertation are designed for different types of overload. Moreover, Chapter 7 showed that each of them performs better than the other for its targeted workloads. On the other hand, for a system where both aperiodic tasks and periodic tasks are activated dynamically, the system developers may not be able to determine which scheduling algorithm to use until they have actually tested them.

The scheduler of the RT-Frontier operating system is implemented in a way that allows selection of different scheduling algorithms without modifying the rest of the system so that the system developers can try out both algorithms in the development process. Moreover, the RT-Frontier operating system facilitates addition of another scheduling algorithm so that the system is evolvable for different situations.

Figure 8.4 illustrates the composition of the RT-Frontier scheduler constructed using modules. The RT-Frontier operating system distinguishes three different types of modules in its implementation: the architecture dependent modules, the scheduling algorithm dependent modules, and the other independent modules. These modules are selected by a configuration tool supplied with the RT-Frontier operating system. The entry to the scheduler occurs on a timer interrupt and on an explicit scheduling request from the running thread. The scheduler includes two architecture dependent modules and one scheduling algorithm dependent module. Most of the source codes that compose the architecture dependent module and the scheduling algorithm dependent modules are implemented as C language functions that can be called from the independent modules.

The architecture dependent modules are implemented for the Responsive Processor (RESP) and for the scheduling simulator (SIM) intended to run on another operating system. The scheduling algorithm dependent module is guarded by scheduler lock acquisition codes in the first architecture dependent module. The scheduler lock operations are architecture dependent, because locking codes are implemented in assembly languages to use atomic memory operations provided by the hardware platform.

The scheduling algorithm dependent modules are implemented for the M-FWP algorithm, the SS-OP algorithm, and the EDF algorithm. These modules include operations for time-driven scheduling and event-driven scheduling, as well as those for checking overrun and preemption. The time-driven scheduling operations handle scheduling events triggered by interrupts from the system timer. For example, a periodic thread

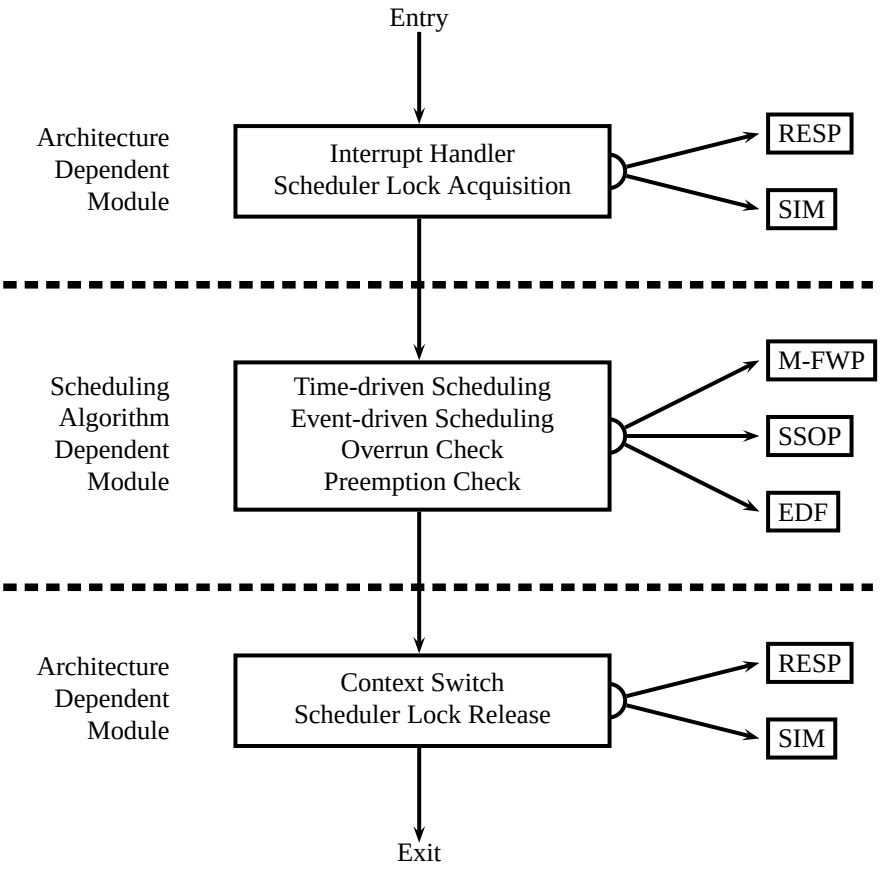


Figure 8.4: Composition of the RT-Frontier scheduler

whose release time is reached is woken up here. The event-driven scheduling operations handle other scheduling events, which includes scheduling triggered by expiration of the overrun timer. After all the scheduling operations are performed, the remaining computation time allocated to the next running thread is checked. If the remaining allocated time is zero, an overrun is detected. The reason why this check is performed after other scheduling operations is that even a thread with zero remaining time may be able to gain some amount of time for its optional part by reclaiming the time left unused by other higher priority threads. The last operation of the algorithm dependent module is to check whether the running thread should be preempted. This check is performed after the overrun check, since by handling the overrun, the operating system may choose another thread as the next running thread. Note that the actual context switch is not algorithm dependent, but rather it is architecture dependent. Hence, the context switch does not occur in a scheduling algorithm dependent module.

Owing to this modular architecture, different scheduling algorithms can be implemented using different data structures. One of the data structures differently implemented between the proposed algorithms is the ready queue. Although both schedulers

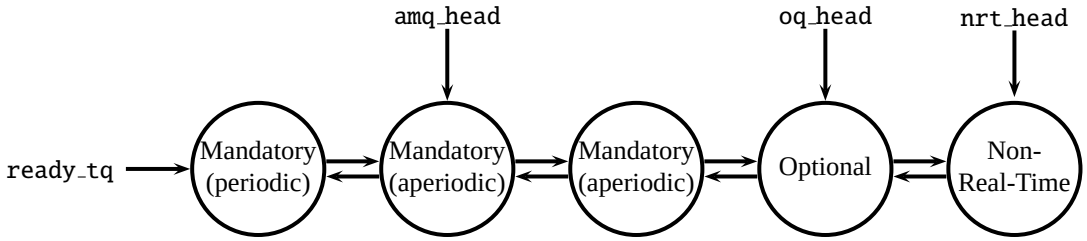


Figure 8.5: M-FWP ready queue

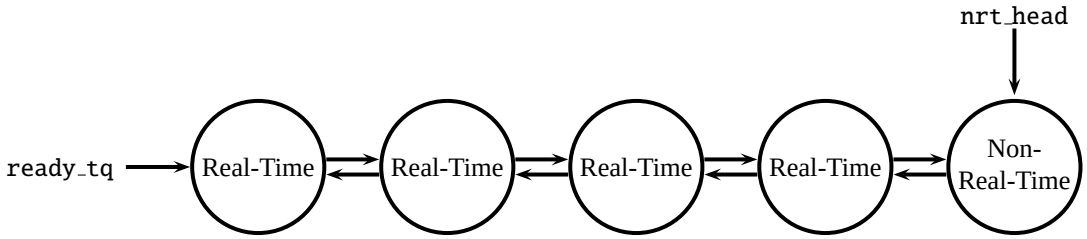


Figure 8.6: SS-OP ready queue

uses a double linked list to manage ready threads in the EDF order, the M-FWP scheduler has three extra pointers shown in Figure 8.5 to emulate the three level EDF queue by a single list, while the SS-OP scheduler has only one extra pointer `nrt_head` shown in Figure 8.6. In the implementation of the M-FWP scheduler, the two pointers `amq_head` and `oq_head` are used to maintain the head of the AMQ and the OQ, respectively. The common pointer `ready_tq` points to the head of the queue and is used to maintain the head of the PMQ, while another common extra pointer `nrt_head` is used to get the head of the non-real-time threads in both implementations. The pointer `nrt_head` is not necessarily required by either of the algorithms, because non-real-time threads are assigned deadline expressed in the largest 256 integers in the RT-Frontier operating system, but it speeds up queuing operations of non-real-time threads. By comparison, a data structure that is only implemented by the SS-OP scheduler is a double linked list that keeps pending activation requests for soft aperiodic threads. In addition, only the SS-OP scheduler implements a priority queue for slack distribution.

8.5 System Calls for Imprecise Computation

Almost all system calls of the RT-Frontier operating system are preemptive. They first push all the processor status words and registers that may be clobbered onto the stack of the thread and then call a reentrant service routine. The saved values are popped up and used to restore the context of the thread when the service routine is completed.

Figure 8.7 shows a case where a context switch occurs in the middle of a system call. White rectangles without a label indicate the execution of a thread that issued the system

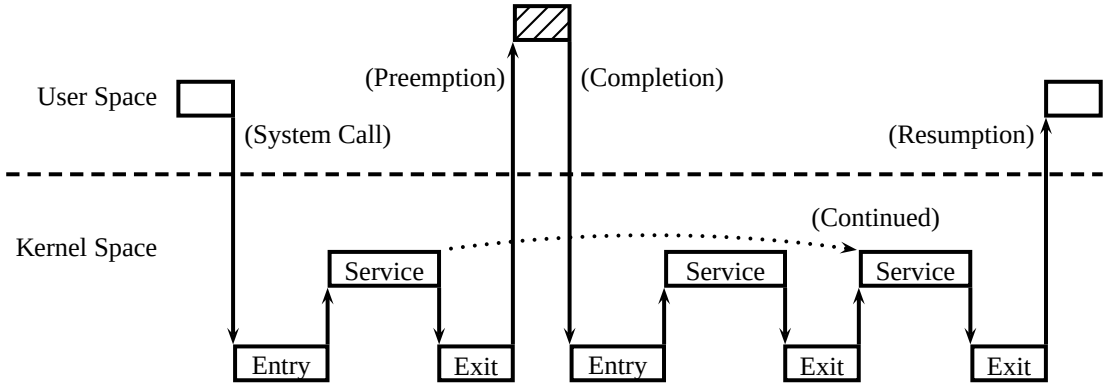


Figure 8.7: System call with preemption

call and a rectangle with lines indicates the execution of another thread that preempted the previous one during the system call. The first *Entry* code sets up the context for executing the service routine of the issued system call. When preemption occurs during the execution of the routine, the *Exit* code is executed to perform a context switch. After the second thread has completed, it issues an system call that indicates the completion of a job. On this latter system call, the second *Exit* code resumes the execution of the first system call. Then, the third *Exit* code resumes the execution of the first thread to its original space.

In the following, we describe system calls implemented in the RT-Frontier operating system to support imprecise computation. The system calls provide the fundamental capability of spawning threads and implement the uniform scheduling interface described in Chapter 4.

8.5.1 Thread Creation and Activation

The system call used to create a thread is `create_thread()`. The system call takes three arguments: the main function of the thread, the argument of the main function, and a pointer to a structure that holds the initial time attributes of the created thread. If there is enough memory left in the system to create the thread, the call returns a pointer to the corresponding thread control block. Otherwise, it returns a `NULL` pointer to indicate the failure.

The created thread does not become active until it is explicitly requested. The system call used to activate a thread is `activate()`. It takes a pointer to the thread control block of the requested thread. It returns 0 if all the acceptance tests of the deployed scheduling algorithm are passed, and it returns -1 otherwise.

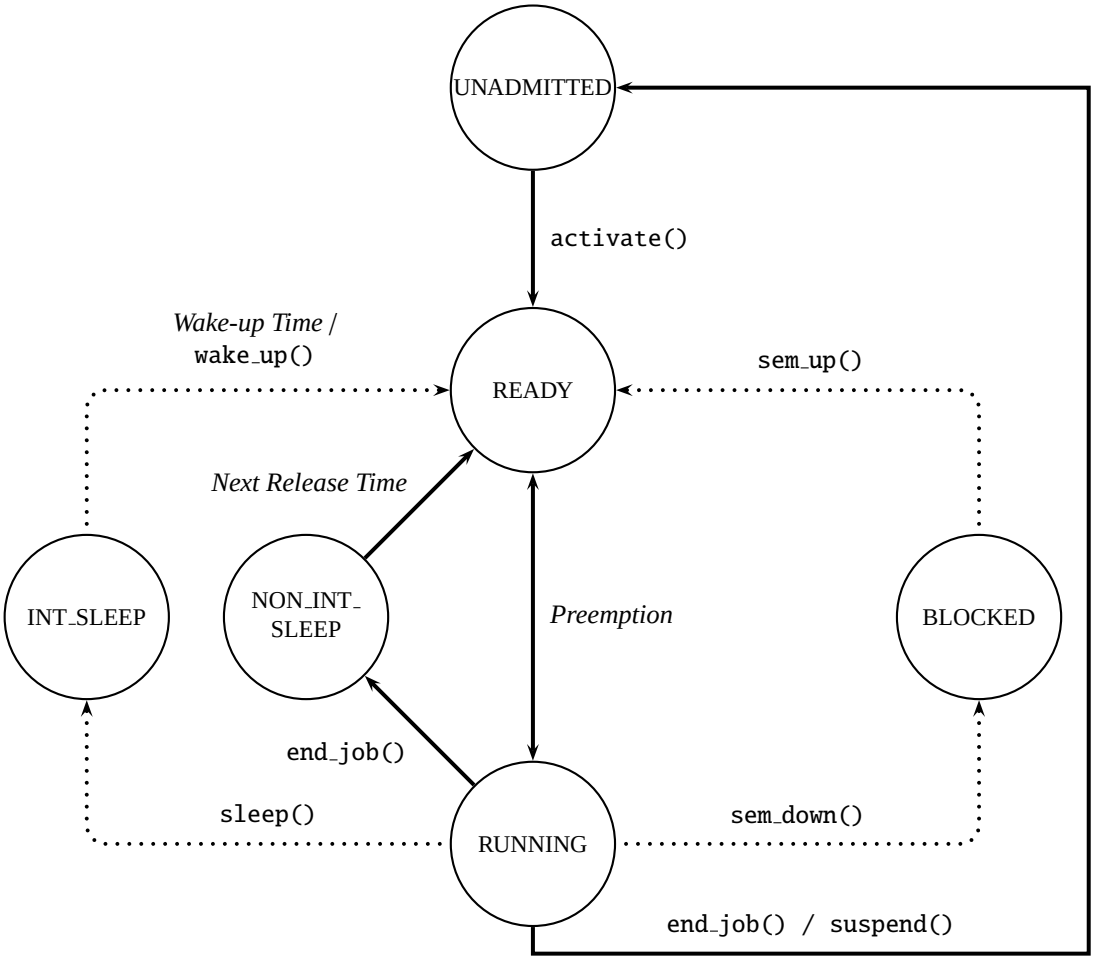


Figure 8.8: Thread state transition diagram

8.5.2 Scheduling Requests

Figure 8.8 shows transitions of thread states. The solid line indicates transitions allowed for both real-time and non-real-time threads, and the dotted line indicates transitions allowed for only non-real-time threads.

When a thread is created, its initial state is `UNADMITTED`, and it becomes `READY` by the `activate()` system call. If the thread is allocated the processor, its state becomes `RUNNING`. When it finishes execution, completion of a job is informed to the scheduler by the `end_job()` system call. This system call neither takes an argument nor returns a value, and it succeeds always. If the thread that finished execution is aperiodic, its state is put back to `UNADMITTED`. If it is periodic, its state is put to `NON_INT_SLEEP`, unless the `suspend()` system call was issued prior to `end_job()`. If suspension of a periodic thread was requested by the `suspend()` system call, the state of the thread is changed to

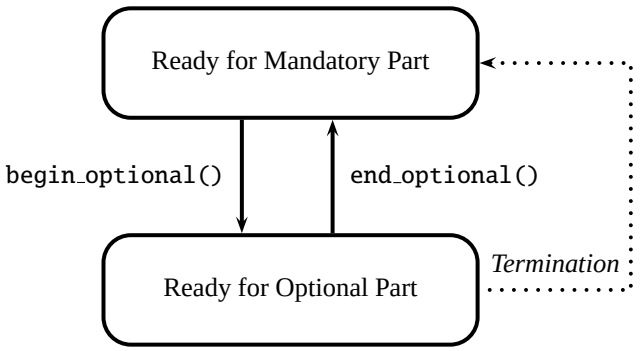


Figure 8.9: State transition diagram for a ready or running imprecise computation

UNADMITTED. Thus, the suspend() system call does not directly change the state of a real-time thread. By comparison, the suspend() system call 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 the sleep() system call, 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 the wake_up() system call. The wake_up() system call takes a pointer to the thread control block of the requested thread as its argument. If the requested thread can be woken up, the system call returns 0, and otherwise it returns -1. The last remaining state BLOCKED can only be reached when a non-real-time thread is blocked on requesting a resource by the sem_down() system call. Thus, on the RT-Frontier operating system, a resource requested by a non-real-time thread should 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 by the sem_up() system call. The two system calls used to acquire and release a resource take a pointer to the semaphore that is guarding the requested resource. The resource acquisition and release by real-time threads are described afterward in this chapter.

8.5.3 Execution State Management

The scheduler further distinguishes the state of a ready or running thread by which part the thread is ready for so that the scheduler can keep track of the amount of time spent by the thread on each part and detect overruns in optional parts. If the overrun cannot be detected at the time it occurs, a mandatory part may not be able to complete before its deadline by an overflow in the schedule.

Figure 8.9 shows the state transition diagram for a ready or running thread. As in the uniform scheduling interface, there are two system calls that are used by a thread to supply enough information to the scheduler on which part the thread is ready for. Both system calls take no argument. The begin_optional() system call tells the

scheduler that the thread has finished a mandatory part and that it requests execution of an optional part. When the system call fails, it return -1. Under the two scheduling algorithms developed in this research, the system call fails only when the optional part must be discarded. In other words, it fails whenever there is no amount of idle time that can be allocated to the thread that requests execution of an optional part. Otherwise the system call succeeds and returns 0, and only at this time, the thread actually becomes ready for optional part. If an optional part is finished, the `end_optional()` system call is used to tell the scheduler that the thread requests execution of a mandatory part. This system call always succeeds and changes the execution state of the thread. In addition, the thread also becomes ready for mandatory part when an optional part is terminated on overrun.

The RT-Frontier operating system provides one more supplementary system call related to the execution state of a thread to conform to the uniform scheduling interface. According to the interface, the scheduler should be able to inform imprecise threads whether their last preceding optional part was terminated or not. The RT-Frontier operating system takes a step further and implements the `prec_opt_status()` system call. It takes no argument and returns the status of the preceding optional part. If the optional part was discarded, the system call returns `OPT_DISC`. Similarly, if it was terminated or completed, the system call returns `OPT_TERM` or `OPT_COMP`, respectively. This enables effective implementation of wind-up operation that should only be executed when its preceding optional part is terminated.

The status of the preceding optional part is maintained by the kernel in the thread control block. For both the M-FWP and SS-OP algorithms, the entry is set to `OPT_DISC` in the `begin_optional()` system call, regardless of whether the system call succeeded or failed. If the system call succeeded, the followings are performed. When the thread overruns, the scheduler sets the entry to `OPT_TERM`. When the thread safely completes its optional part, the entry is set to `OPT_COMP` in the `end_optional()` system call.

Figure 8.10 shows an example program structure of a thread for an imprecise computation. The computation has three mandatory parts and two optional parts, and begins with a mandatory part. The mandatory parts are represented by three functions whose names are mandatory1, mandatory2, and mandatory3. Likewise, the functions that represent the optional parts are named optional1 and optional2. A thread corresponding to this program begins with saving the context of the thread by `setjmp()`. Since its return value in the case of a direct call is 0, the thread continues to execute its first mandatory part (mandatory1). When the mandatory part is finished, the thread issues the `begin_optional()` system call to tell the scheduler that it wants to execute an optional part. If the system call succeeds, it executes the first optional part (optional1). And then, on completion of the optional part, it issues the `end_optional()` system call to mark the end of the first optional part. After returning from the system call, the thread is ready to execute its second mandatory part. If there is no termination at all, the thread executes until the `end_job()` system call in the last line. On the other hand, if execution of an optional part is terminated, the thread resumes back to the first line.

```
 1: stage = setjmp(env);
 2: switch (stage) {
 3:         case 0:
 4:                 mandatory1();
 5:                 if (begin_optional() == 0) {                    // try to execute the first optional part
 6:                         optional1();
 7:                         end_optional();                                 // to the second mandatory part
 8:                 }
 9:         case 1:
10:                 mandatory2();
11:                 if (begin_optional() == 0) {                  // try to execute the second optional part
12:                         optional2();
13:                         end_optional();                                   // to the third mandatory part
14:                 }
15:         case 2: default:
16:                 if (prec_opt_status() == OPT_TERM) {          // only if optional2 was terminated
17:                         mandatory3();
18:                 }
19:                 break;
20: }
21: end_job();
```

Figure 8.10: Pseudo-code of an imprecise computation

Suppose that it was the second optional part (optional2) that was terminated. Then, the return value of `setjmp()` is set to 2 by the kernel. Moreover, before the thread resumes its execution, the state of the terminated thread internally maintained by the scheduler is changed to ready for mandatory part, as is shown in Figure 8.9. On resuming execution at the first line, the thread goes on to check the completion status of the second optional part, and finding out that it was terminated, the thread executes the third mandatory part (mandatory3).

8.5.4 Dynamic Time Reservation and Cancellation

The on-line reservation of execution time for optional parts is accomplished by two system calls. These on-line reservation system calls enable effective execution of optional parts with 0/1 constraints.

The reservation system calls work effectively with the M-FWP algorithm, since the amount of computation time allocated to an optional part can decrease even after the execution of optional part is started. On the other hand, the system calls are also effective with the SS-OP algorithm, because the amount of reward received by executing an optional part only increases discretely in practice. Thus, if a thread knows that there is not enough amount of computation time to execute one of its optional parts to re-

ceive reward, the thread can voluntarily skip the optional part to leave all the remaining optional computation time for its subsequent optional parts.

One of the system calls that reserve computation time is `reserve_opttime()`. The amount of time reserved is requested as the argument to the system call. When the amount of optional computation time that remains allocated to the thread is larger than or equal to the requested amount, the system call succeeds and returns the amount of time reserved. Thus, the return value and the argument are equal when reservation is actually made. Otherwise, the return value indicates the maximum amount of time that could have been reserved at the time this system call was issued. It is important to note that whether the returned value is reliable on a failure depends on the scheduling algorithm. Under the M-FWP algorithm, the amount of the available time may be different even when the same system call is issued in the next line of the program, because it is possible that this thread is preempted between these consecutive system calls and the amount of idle time allocated to this thread may decrease before it resumes. On the other hand, the return values under the SS-OP for the same consecutive calls differs only by the time spent between the calls. Hence, the return value of the system call can provide a hint on how much optional computation time is left.

The other system call that reserves computation time is `reserve_all_opttime()` that also takes one argument. The system call reserves all the optional computation time available to the requesting thread if that amount is larger than or equal to the requested amount. Thus, this system call also succeeds only when at least the requested amount of time is left unused. Using this system call, all remaining optional computation time can also be reserved by setting the requested amount to zero.

The RT-Frontier operating system allows previously made reservation of computation time to be canceled by two other system calls. The `cancel_all_reservation()` system call takes no argument and cancels all previously made reservation. The other system call `fix_reservation()` takes one argument that specifies the amount of computation time that should remain reserved after the system call. Thus, any excessive amount of reservation is canceled by this system call. This system call only succeeds when the amount of remaining computation time reserved is larger than or equal to the specified amount, and fails otherwise.

Figure 8.11 shows an example program structure of an imprecise thread that requests dynamic reservation of optional computation time. The computation in this example has only two mandatory parts and two optional parts. A mandatory part does not have to follow the second optional part, since the program reserves the necessary computation time before requesting execution of optional parts. In other words, termination of an optional part is totally avoided by reserving enough time via the `reserve_opttime()` system call and by actually executing the optional part only when the reservation succeeded. The important point here is that the `begin_optional()` system call must always be issued between two different mandatory parts even when the reservation failed, because only through this system call can the scheduler determine which mandatory or optional part is being executed. The reservation is canceled each time the optional part for which

```
1:  stage = setjmp(env);
2:  switch (stage) {
3:      case 0:
4:          mandatory1();
5:          rsv = reserve_opttime(o_i^1);        // try to reserve time for the first optional part
6:          if (begin_optional() == 0) {                    // always issue this system call
7:              if (rsv == o_i^1) {
8:                  optional1();
9:              }
10:             end_optional();
11:         }
12:         cancel_all_reservation();                       // cancel all remaining reservation
13:     case 1: default:
14:         mandatory2();
15:         rsv = reserve_opttime(o_i^2);        // try to reserve time for the second optional part
16:         if (begin_optional() == 0) {                    // always issue this system call
17:             if (rsv == o_i^2) {
18:                 optional2();
19:             }
20:             end_optional();
21:         }
22:         cancel_all_reservation();                       // cancel all remaining reservation
23:         break;
24: }
25: end_job();
```

Figure 8.11: Pseudo-code of an imprecise computation with dynamic time reservation

the reservation was made is finished.

8.5.5 Resource Acquisition and Release

The RT-Frontier operating system expects that all the shared resources are guarded by semaphores, and it controls resource acquisition and release through semaphore operations.

The `rt_sem_down()` system call tries to acquire a resource. It takes three arguments: a pointer to the semaphore that is guarding the requested resource, a number of units to acquire, and the worst case access duration. The system call compares the amount of remaining optional computation time with the worst case access duration of the requested resource. Specifically, the RT-Frontier operating system checks whether the following condition holds:

$$S_i(t) \geq b_i^j, \tag{8.1}$$

where b_i^j is the worst case access duration of the requested resource Z_j by the requesting job J_i. If the condition is met and there is enough units of the resource available, the system call returns 0 to indicate the success.

In order to implement imprecise computations that share resources in the optional part efficiently, the RT-Frontier operating system also provides the `rt_try_sem_down()` system call. The system call is used to acquire a resource and takes the same three arguments. Moreover, just like the previous system call, this system call performs the same check on the request and returns 0 on success.

The difference between these system calls exists in how the requesting thread is treated when a resource acquisition fails. The `rt_sem_down()` system call is designed for requesting a resource that is indispensable for further execution. Thus, it terminates the optional part of the requesting thread. On the other hand, the `rt_try_sem_down()` system call is designed for requesting an optional resource that is only necessary to enhance the QoS achieved by the execution. Thus, it only returns -1 on failure.

The acquired resources are released by the `rt_sem_up()` system call, which always succeeds. It takes two arguments: a pointer to the semaphore that is guarding the requested resource and a number of units to release.

8.6 System Overhead Accounting

The RT-Frontier operating system rigidly defines what the execution time of an application thread is. The RT-Frontier operating system expects that the worst case execution time supplied at the creation of a thread includes the length of all the intervals where the overrun timer is set, under the condition that there is only the system timer interrupts. In other words, the worst case execution time is expected to include a portion of overheads incurred in handling the system timer interrupts and overrun timer interrupts. This leaves only the length of the intervals where the overrun timer is being stopped as the system overhead.

The system overhead accounts for context switches and scheduling operations. The overhead of these activities are added to the processor demand of each thread by the operating system when a thread is created.

Under the M-FWP algorithm, the upper bound on the number of context switches is bounded by twice the number of the mandatory parts and optional parts. Thus, the overhead is different for each thread. On the other hand, the upper bound on the number of the context switch under the SS-OP algorithm never exceeds two, since all the threads are always executed in the EDF order.

The overhead of scheduling operations are dependent on the number of threads. Since the RT-Frontier scheduler keeps all threads in linear lists, the overhead is linearly proportional to the number of threads. Thus, the system overhead accounted for in the processor demand of each thread is increased each time a thread is activated and is decreased each time a thread is suspended.

Table 8.1: Kernel code size of RT-Frontier

Algorithm	Text	Data	Bss	Total
EDF	33996	600	360	34956
M-FWP	39104	608	360	40072
SS-OP	40148	672	376	41196

(byte)

8.7 Numerical Characteristics

This section presents numeric characteristics of the RT-Frontier kernel to show the practicability of the presented implementation methods. In particular, it presents the impact on the kernel code size first, and then presents the overhead of the scheduler. It also presents the numerical characteristics of the RT-Frontier kernel that implements the EDF algorithm to serve as a baseline for comparison.

In all measurements, the RT-Frontier kernel was compiled with `gcc` 3.4.4 with the second level of optimization (-O2) and the SPARClite core of the Responsive Processor was set to operate at 100MHz.

8.7.1 Kernel Code Size

The code size of the RT-Frontier kernel with the default configuration for a Responsive Processor is shown in Table 8.1. The default configuration includes the following features: real-time and non-real-time resource access control, inter-thread communication, communication with real-time channel, context switch logging, and device drivers for *Responsive Link* and PCI. Details on the Responsive Link implemented in the Responsive Processor is described in [79].

The configuration for the EDF algorithm implements TBS to serve aperiodic jobs and the SRP protocol for resource access control, so that all configurations can handle both periodic and aperiodic workloads with shared resources. However, the configuration for the EDF algorithm does not include any supporting mechanisms for imprecise computation. Thus, the code size shown for the M-FWP algorithm and that for the SS-OP algorithm include the code size required to implement supporting mechanisms for imprecise computation.

The increase in the code size when the scheduling algorithm is switched from the EDF algorithm to the M-FWP algorithm is 5116 bytes, which leads to the increase ratio of 1.146. The increase is mostly in the text section, which proves that the implementation of the M-FWP scheduler requires few extra data structures. When the SS-OP algorithm is used instead of the EDF algorithm, the increase in the code size is 6240 bytes, or the increase ratio is 1.178. Since the SS-OP scheduler implements a priority queue for distributing slack to control the QoS of the system, the increase is in all three sections. Nonetheless, the increase of the kernel size in this case is still roughly the half

of having the real-time channel support compiled in, which takes up 12176 bytes of all kernel sizes shown in Table 8.1.

8.7.2 Overheads

Overheads are measured in the worst case scenario that can occur for a real-time thread. For example, the worst case scenario of enqueuing a thread to the ready queue occurs when it has the latest deadline. For each worst case scenario, the overheads are presented for three cases. The first case where both the instruction and data caches are flushed before the measurements is denoted as "Flush-ID." Similarly, the second case where only the data caches are flushed is denoted as "Flush-D." And the third case where caches are flushed no more than necessary for correct operations is denoted as "No Flush." For these three cases, the result shown is the average of 100 measurements. The reason to show to the result of the second case where only the data cache is flushed is that the instruction cache statistically has higher hit rate than the data cache [82], and thus flushing both caches can lead to too pessimistic situation that never occurs in practice. Therefore, while the theoretical worst case overhead is represented by the first case, a more practical worst case could be represented by the second case. Moreover, the results can also be used to grasp the affect of caches on the worst case execution time. In particular, the larger the difference the more processor bandwidth is wasted in the classical approach where reservation is fully made on the worst case execution time.

Common overheads The overheads that are common to the three scheduling algorithms are shown in Table 8.2. The overhead of the system timer interrupt handling is the length between the expiration of the system timer and the time the interrupted thread resumes. The overhead of the context switch merely consists of saving and reloading appropriate registers and does not included the overhead of scheduling. The null system call overhead is the entire overhead of a system call that does nothing, which is implemented only for measurement purpose. The overhead of the `create_thread()` system call is the length of time needed to allocate memory pages and set entries of the thread control block to the requested values. And finally, the overhead of the `suspend()` system call corresponds to setting a flag that indicates that the thread should be suspended when it completes the latest job. These overheads serve as baselines for comparison to show the practicality of the implementation. Moreover, the overhead of handling the system timer interrupt in particular is useful to determine the practical limit on how fine time can be managed.

The practical limit on the unit time is much coarse than the finest unit that can be managed on a Responsive Processor. The finest unit is 0.05μs, which is the resolution of its internal timer, while by default the RT-Frontier operating system manages the system time in the unit of 1ms and the execution time in the unit of 1μs.

The system time is not managed in the finest unit to keep the processor bandwidth reserved for operating system activities small. Considering that the worst case overhead

Table 8.2: Overheads common to EDF, M-FWP, and SSOP

Operations	Flush-ID	Flush-D	No Flush
System timer interrupt handling	40.75	18.00	11.50
Context switch	26.75	14.50	8.25
Null system call	9.00	2.70	1.70
`create_thread()` system call	151.10	102.40	101.20
`suspend()` system call	12.70	2.60	2.60

(μs)

of handling the system timer interrupt is less than 41μs and that the interrupt is generated every 1ms, the overhead of managing the system time takes up at most 4.1% of the processing time. By comparison, if the system time is managed in the unit of 100μs, then the processor bandwidth that cannot be used for applications becomes as much as 41% in the worst case. It may still not be acceptable in the average case, which is the case where no cache is flushed more than necessary, because about 11% of the total processor bandwidth cannot be used for applications. Since the processing resources in embedded systems are often severely limited, this could lead to a huge degradation in the perceivable system performance. Hence, setting the unit of the system time to a finer value works against the objective of this research, which is to fully utilize the processor for significant activities.

The reason why the execution time is not managed in the unit of 0.05μs is that the length of execution time is usually given in the unit of 1μs. Moreover, managing the execution time in a finer unit can increase the system overhead, since the counter of the hardware timer in the Responsive Processor is only 16-bit long. This means that if a job keeps running for longer than 65536 units, the overrun timer must be reset at least once before the job completes. Since the interrupts from an overrun timer are handled by using the same mechanism as the system interrupt, the overhead of resetting the overrun timer on expiration is about the same as that of handling the system timer interrupt. Thus, when there is one interrupt from the overrun timer per job, the total time required to complete a job must be lengthened by about 10 to 40μs. Thus, we conclude that there is no significance in managing the execution time in a finer unit.

Algorithm dependent overheads The overheads of releasing a periodic thread and that of handling overrun are dependent on the scheduling algorithms. Likewise, the overheads of the rest of the system calls implemented for the uniform scheduling interface are dependent on the scheduling algorithms.

The overheads of releasing one periodic thread is shown in Table 8.3, where n is the number of threads in the system. The measured overhead only accounts for checking whether there is a thread in the sleep queue whose release time is reached and putting it to the ready queue. Thus, the worst case scenario occurs when the released thread is put to the tail of the ready queue. The increase rates with respect to the number of the

Table 8.3: Overheads of releasing one periodic thread

Algorithm	Flush-ID	Flush-D	No Flush
EDF	$0.85n + 17.25$	$0.85n + 6.25$	$0.80n + 4.55$
M-FWP	$0.80n + 24.90$	$0.70n + 10.50$	$0.65n + 7.70$
SS-OP	$0.85n + 22.30$	$0.75n + 8.40$	$0.75n + 7.20$

(μs)

Table 8.4: Overheads of handling overrun

Algorithm	Flush-ID	Flush-D	No Flush
M-FWP	$1.00n + 36.85$	$0.95n + 16.55$	$1.05n + 9.70$
SS-OP	$1.00n + 32.50$	$1.00n + 12.40$	$1.00n + 7.00$

(μs)

threads are almost the same in all algorithms, but the intercept values are the largest for the M-FWP, since the M-FWP scheduler has to maintain extra pointers to implement the three level ready queue.

The overheads of handling overrun are shown in Table 8.4. The overheads under the EDF algorithm is not presented, as it does not detect overrun. The worst case scenario for this operation occurs when the thread that caused overrun finishes its execution with the terminated optional part. And again, the increase rates are almost equal, but the intercepts are larger for the M-FWP scheduler.

The overheads of the two system calls `begin_optional()` and `end_optional()` used to indicate the beginning and end of optional parts are shown in Table 8.5. The worst case scenario for the `begin_optional()` system call occurs when the system call succeeds. Specifically, under the M-FWP algorithm, the worst case scenario occurs when all other threads are periodic and are in the OQ. Under the SS-OP algorithm, the overhead is independent of other threads, since the scheduler can determine the amount of time that should be allocated to each thread without considering the state of the other threads, owing to the QoS controller. The overhead of the `end_optional()` system call is constant under both scheduling algorithms, since neither calculation of optional computation time nor queuing operation is required.

Table 8.5: Overheads of system calls used to indicate optional parts

System Calls	Algorithm	Flush-ID	FLush-D	No Flush
`begin_optional()`	M-FWP	$60.70n + 39.60$	$58.20n - 1.15$	$58.20n - 18.00$
	SS-OP	60.00	31.50	20.80
`end_optional()`	M-FWP	28.80	16.75	11.10
	SS-OP	27.30	13.40	8.90

(μs)

Table 8.6: Overheads of the `end_job()` system call

Thread Type	Algorithm	Flush-ID	Flush-D	No Flush
Aperiodic	EDF	52.75	43.50	38.25
	M-FWP	55.70	36.50	31.80
	SS-OP	93.65	68.45	61.60
Periodic	EDF	$0.90n + 92.90$	$0.90n + 44.10$	$0.85n + 38.45$
	M-FWP	$0.90n + 100.00$	$0.90n + 51.45$	$0.80n + 45.00$
	SS-OP	$0.90n + 92.60$	$0.90n + 47.00$	$0.85n + 38.60$

(μs)

Table 8.7: Overheads of system calls used for resource access control

System Calls	Algorithm	Flush-ID	Flush-D	No Flush
`rt_try_sem_down()`	M-FWP	13.50	9.10	7.50
	SS-OP	21.35	9.90	6.10
`rt_sem_down()`	EDF	53.60	26.85	17.90
	M-FWP	40.30	21.55	13.85
	SS-OP	37.90	17.40	12.10
`rt_sem_up()`	EDF	53.90	38.80	29.55
	M-FWP	95.05	54.90	43.75
	SS-OP	78.30	42.80	31.70

(μs)

The `end_job()` system call has different worst case scenario for an aperiodic thread and a periodic thread, as is shown in Table 8.6. The worst case scenario for aperiodic thread further depends on the scheduling algorithm. Under the EDF algorithm and the M-FWP algorithm, the worst case overhead for an aperiodic thread comes from dequeuing the thread from the ready queue. Under the SS-OP algorithm, the worst case overhead is measured when a soft aperiodic thread issued the system call, because the SS-OP scheduler releases a pending activation requests for a soft aperiodic thread in the `end_job()` system call. Thus, its overhead is almost the twice as that of the other algorithms. On the other hand, when the same system call is issued by a periodic thread, the worst case overheads are proportional to the number of threads, because periodic threads that wait for their next releasing time are kept in a sleep queue. Thus, the worst case scenario for all three algorithms occurs when the thread is put to the tail of the sleep queue.

The overheads of the system calls used for resource access control are shown in Table 8.7. The worst case scenario for the system calls used to acquire a resource occur when `rt_try_sem_down()` succeeds and when `rt_sem_down()` fails. The worst case scenario for the `rt_sem_up()` system call occurs when preemption occurs by releasing a resource. Thus, the overhead of the `rt_sem_up()` system call is measured as the length between the time the system call is issued and the time the next thread resumes.

Table 8.8: Overheads of the `activate()` system call

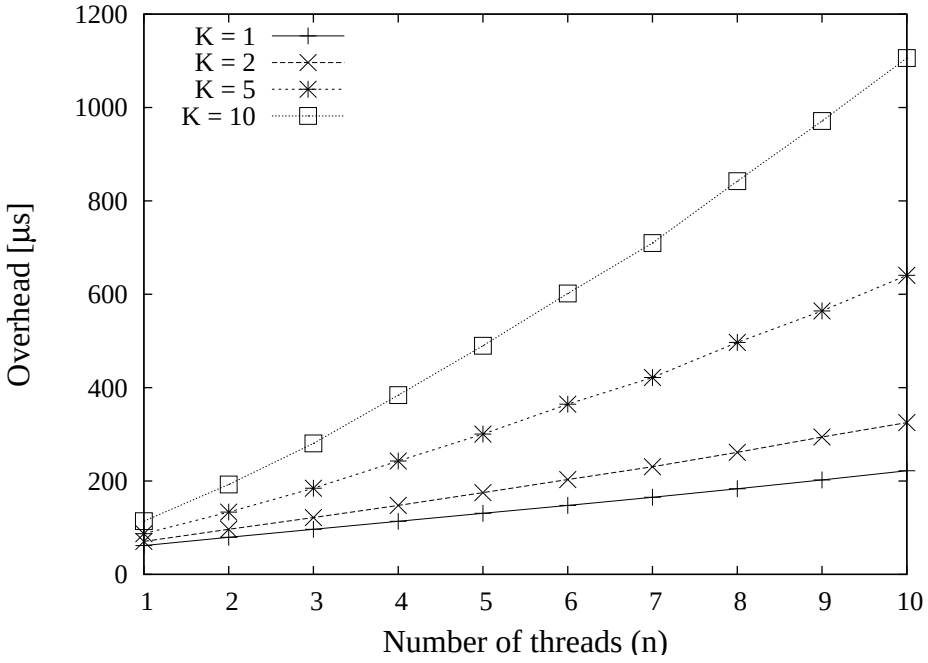
Thread Type	Algorithm	Flush-ID	Flush-D	No Flush
Aperiodic	EDF	$0.85n + 45.85$	$0.85n + 30.70$	$0.85n + 22.55$
	M-FWP	$12.10n + 104.00$	$12.10n + 80.80$	$12.10n + 73.20$
	SS-OP	$2.85n + 62.40$	$2.85n + 31.65$	$2.80n + 24.00$
Periodic	EDF	$4.00n + 80.00$	$4.00n + 43.85$	$4.00n + 35.00$
	M-FWP	$14.95n + 134.70$	$14.90n + 95.75$	$14.90n + 86.65$
	SS-OP	$4.00n + 86.10$	$3.90n + 45.95$	$3.90n + 36.60$

(μs)

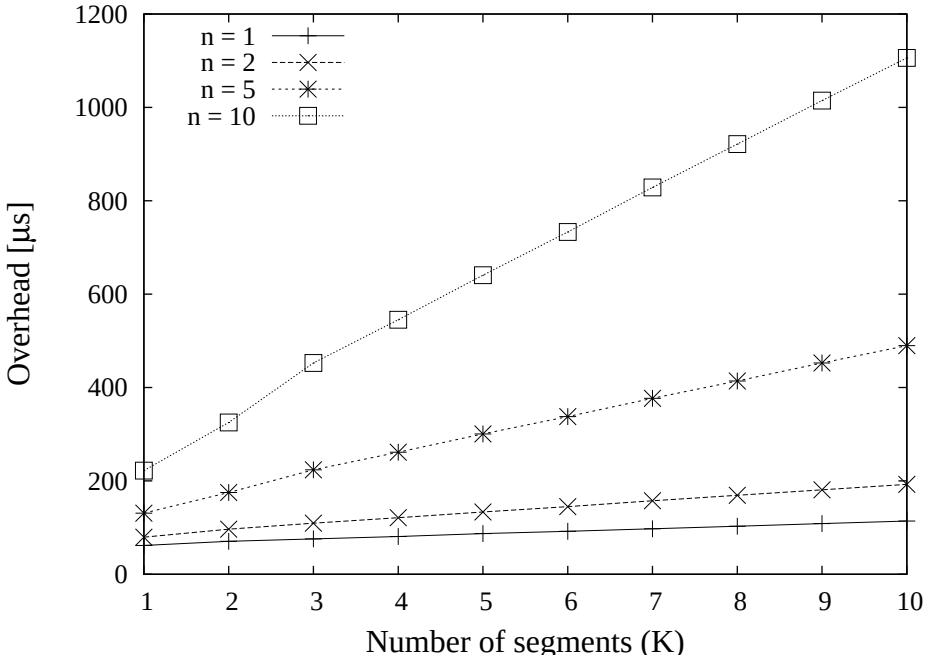
As can be seen, the acquisition overheads are smallest under the M-FWP algorithm, since the system call only puts the scheduler internally into non-preemptive mode or into preemptive mode. The overheads under the EDF algorithm and under the SS-OP algorithm are similar for the `rt_sem_down()` system call, as they both implements the SRP protocol. However, the worst case overhead of the `rt_sem_up()` system call is larger under the SS-OP algorithm than that under the EDF algorithm, and that under the M-FWP algorithm is the largest. These differences come from the overhead of preemption.

Finally, the overheads of the `activate()` system call are shown in Table 8.8 where n is larger than 1. Under the EDF algorithm, the worst case scenario for activating a thread occurs when the activated thread is enqueued to the tail of the ready queue. The reason why the increase rate of the worst case overhead for activating a periodic thread is more than three times larger than that for activating an aperiodic thread is that the former needs more complicated feasibility check. Since it is not assumed that the relative deadline and period of a periodic thread are equal, the EDF scheduler implemented uses a sufficient test [75] whose time complexity is linear with respect to the number of thread. Thus, by adding the overhead of feasibility test, the increase rate becomes larger. The M-FWP scheduler has the largest overhead for activating both types of threads. The worst case scenario here occurs when all the other threads are in the OQ and, by accepting the requested threads, all of them are put back to the PMQ or the AMQ. The worst case overhead of the SS-OP scheduler for activating a periodic thread is also larger than that for activating an aperiodic thread again due to the difference in the acceptance test. As for the SS-OP algorithm, it must be noted that, if the activated periodic thread has at least one optional part, the overhead of distributing slack must be added to the values shown in Table 8.8.

The overhead of distributing slack under the SS-OP algorithm depends both on the number of imprecise periodic threads and the number of segments each of them has. The measured overheads are shown in six figures in Figure 8.12, Figure 8.13, and Figure 8.14. Since the worst case scenario for the slack distribution occurs when all the segments are considered for allocation, the theoretical upper bound on the time complexity is $O(Kn \log n)$ where K is the number of segments each thread has. Thus, if the

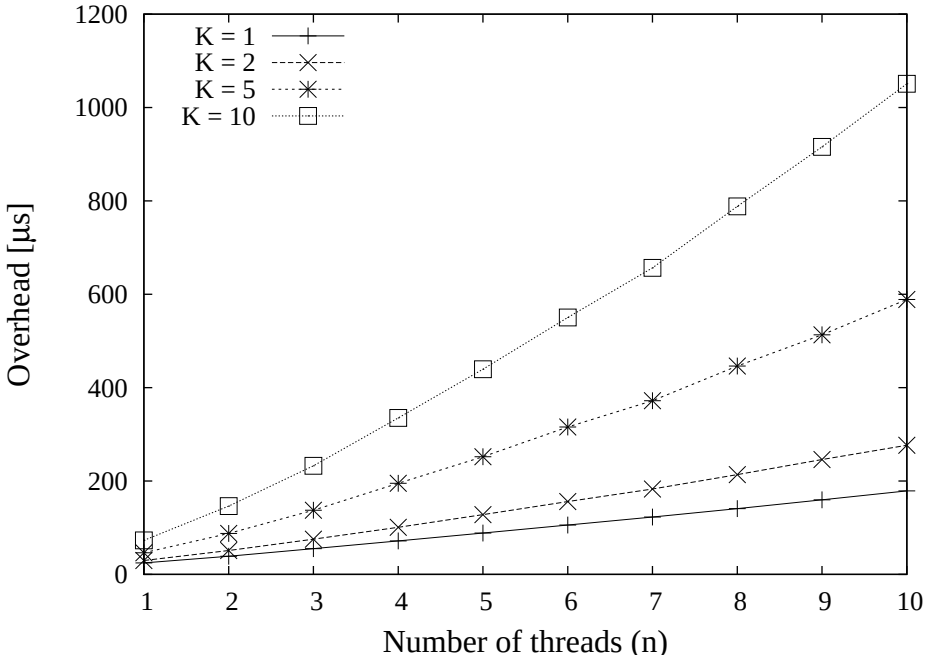


(a) Fixed number of segments

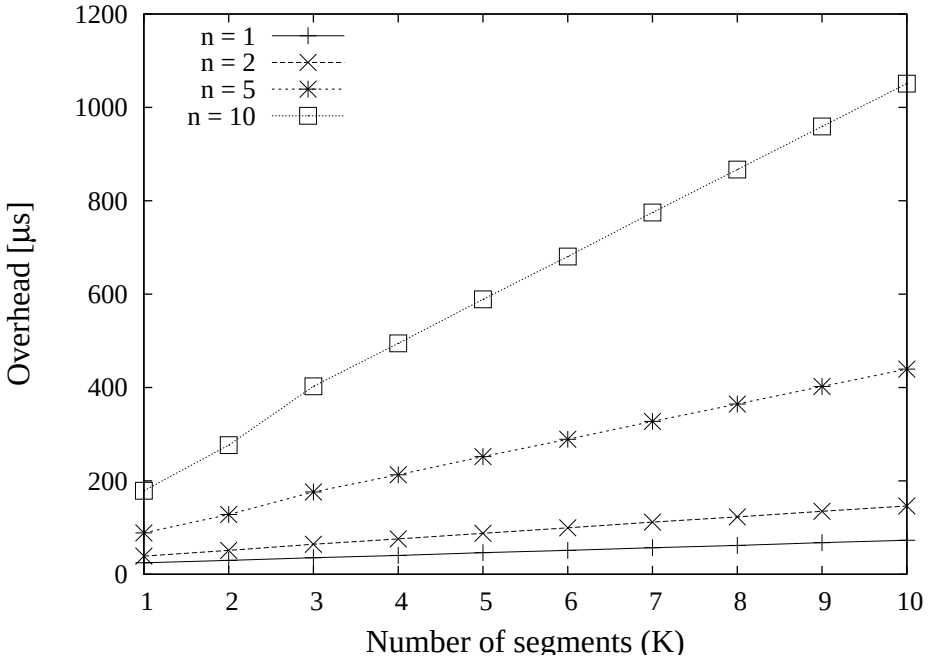


(b) Fixed number of threads

Figure 8.12: Overheads of SS-OP slack distribution (Flush-ID)

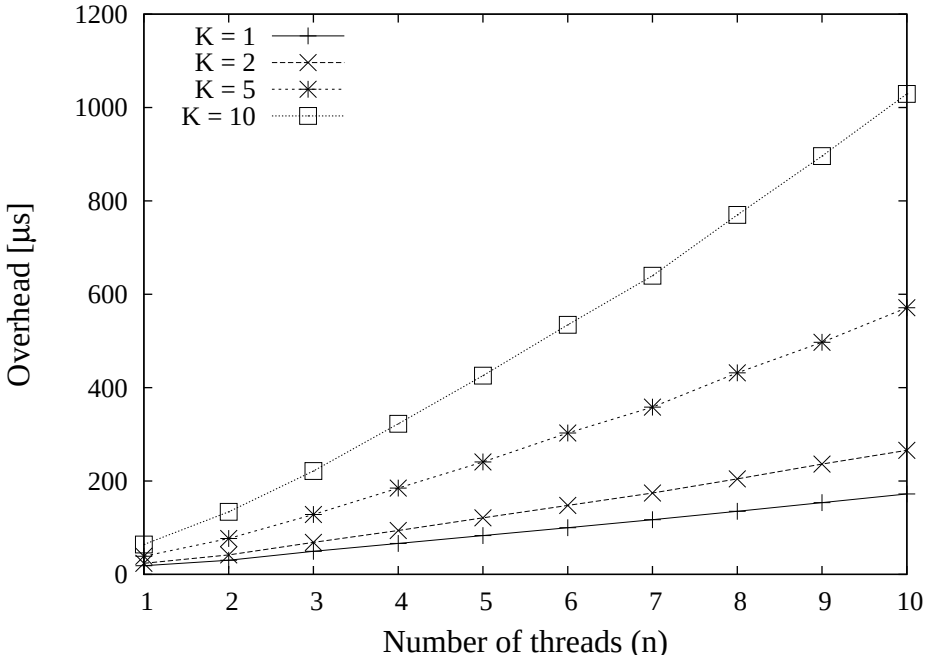


(a) Fixed number of segments

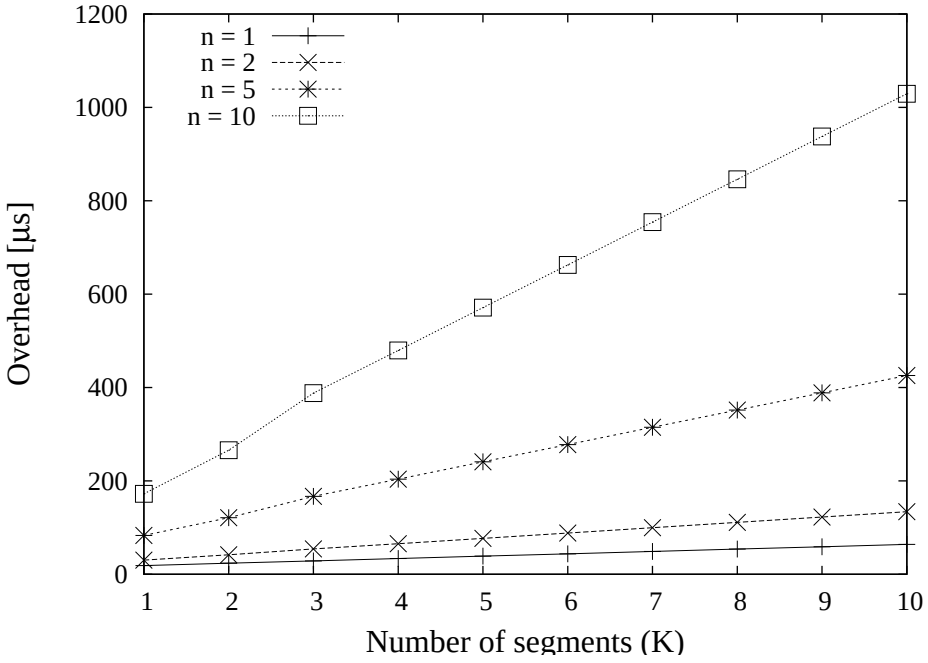


(b) Fixed number of threads

Figure 8.13: Overheads of SS-OP slack distribution (Flush-D)



(a) Fixed number of segments



(b) Fixed number of threads

Figure 8.14: Overheads of SS-OP slack distribution (No Flush)

number of segments is fixed, the overhead increases by $O(n \log n)$, and if the number of threads is fixed, it increases by $O(K)$.

In all six cases, no matter if caches are flushed or not, when the number of threads and that of the segments are both 10, the overhead of slack distribution reaches 1000μs or 1ms, which is the length of the tick interval. This is problematic, because when the scheduler is locked more than one unit of system time, release of a periodic thread could be delayed by one unit. Moreover, this could degrade the performance of the system for two more reasons: a firm or hard real-time computation whose relative deadline is less than the amount of time required for the slack distribution cannot be implemented; and the processor utilization that can be used to execute user application decreases. Although the actual degradation of performance may vary depending on a specific processor used, the results shown in the six figures suggest that the trade-off between the quality of schedule and the system overhead be considered at the time the system is designed. For example, on a Responsive Processor, the number of segments should be limited to 1 or 2 in practice when the number of threads are larger than 10.

Comparing the overhead of the presented algorithms with that of the EDF algorithm for each case, the most significant difference is in the overhead of the `activate()` system call. The largest increase rate appears under the M-FWP algorithm. In this case, when there are 50 threads in the system, the overhead could lead to about 900μs. However, as was stated, this worst case scenario occurs when all threads are in the OQ. Thus, this worst case scenario rarely occurs under transient overload. Similarly, the worst case slack distribution overhead under the SS-OP algorithm rarely occurs under persistent overload, since the worst case scenario occurs only when all segments are allocated optional computation time. Moreover, it is possible to control the overhead of the slack distribution by limiting the maximum number of segments for each task. Furthermore, Chapter 7 revealed that the average QoS of the system is still close to the optimal when the accuracy of the worst case execution time is low, even if slack is not optimally distributed. Hence, the developers of real-time systems can trade off the quality of slack distribution with the system overhead to meet their performance criteria.

The overhead of handling overrun and that of system calls used to indicate optional parts do not exist for the EDF algorithm. Thus, the whole overheads of these system calls must be considered as the drawback of supporting imprecise computation. The overhead of handling overrun is close to the overhead of releasing a periodic thread. The worst case overhead, however, is less than 140μs, even if there is 100 threads in the system. This is smaller than the worst case overhead of the `create_thread()` system call. The largest increase is due to the `begin_optional()` system call under the M-FWP algorithm. The worst case scenario for this system call, however, occurs when all other threads are in the OQ. Thus, in the worst case scenario, there must be enough idle time left in the system. Therefore, the overhead of the `begin_optional()` system call under the M-FWP algorithm would not cause a severe problem in practice. The overheads of other system calls used to indicate optional parts are no more than

60μs. Since the worst case overhead of handling system timer interrupts is 40.75μs, we conclude these overheads do not affect the throughput of the system.

Finally, by regarding the overhead without flushing caches (No-Flush) as the actual execution time and the overhead with flushing both instruction and data caches (Flush-ID) as the estimated worst case execution time, the accuracy of the worst case execution time ranged from 0.19 to 0.73 in these measurements. Considering that it is only the caches that caused the difference, this result fully supports our argument on the need of this research that the effective processor utilization becomes severely low if reservation is made for the worst case execution time in dynamic real-time systems where a complicated processor is used.

8.8 Summary

This chapter considered issues on implementation of the proposed scheduling algorithm and described their actual implementation in the RT-Frontier operating system. Some issues like time management and modular scheduling architecture are commonly important to real-time operating systems with and without imprecise computation support. Other issues like context management for terminated optional parts and the system calls that conform to the uniform scheduling interface are specific to imprecise computation.

This chapter also presented numerical characteristics of the RT-Frontier operating system. The increase in the kernel code size due to supporting imprecise computation and implementing the proposed scheduling algorithms were shown as well as the overheads of specific operations. Each operation was considered separately for its worst case scenario and the overheads for the scenario were measured in three different cases with respect to the state of caches. These results not only showed the effectiveness of the implementation, but also identified how caches can affect the estimation of the worst case execution time, supporting our motivation to increase the effective processor utilization.

CHAPTER 9

CONCLUSIONS

This dissertation presented research on real-time scheduling of practical imprecise tasks to increase the effective processor utilization. The presented approach accomplishes higher processor utilization by enabling construction of theoretically overloaded real-time systems. The research considered the difference between the transient and persistent overload, and developed two scheduling algorithms. The thesis supported by this dissertation is that systematic real-time scheduling of practical imprecise tasks evicts overprovisioning of resources and increases the effective processor utilization without causing a critical timing violation.

9.1 Summary of Contributions

This research made contributions of both theoretical and practical significance to embedded real-time computing. Each of the contributions leads to eviction of resource overprovisioning and contributes to the development of cost effective real-time systems that can work under different conditions and in various environments.

The first contribution is made in the imprecise computation model. The practical imprecise computation model allows application programmers to integrate operations needed to compensate for terminated optional parts. Moreover, the presented model allows more than one mandatory parts and more than one optional parts to interleave each other in the linear task model to express more complex structures. Another important contribution regarding this computation model is the uniform scheduling interface that glues the application tasks and the scheduling algorithm. The interface makes the scheduling algorithms transparent to application tasks so that no recompilation of user programs is required even when scheduling algorithms other than the original one deployed at the development time is used.

The major theoretical contribution is the development of two scheduling algorithms for practical imprecise tasks. The presented scheduling algorithms are developed by considering the types of overload as one of the primary factors in their design. For transient overload, the M-FWP scheduling algorithm is developed. The transient overload

in this research is caused by dynamically activated aperiodic tasks. Thus, the algorithm focuses on resolving the overload in a short period of time and aims to maximize the acceptance ratio of aperiodic tasks by greedily collecting all idle time in the schedule. The SS-OP scheduling algorithm, on the other hand, is designed to work under persistent overload. The persistent overload is caused by dynamic activation of periodic tasks and thus cannot be resolved until one or more periodic tasks are suspended. Thus, the SS-OP algorithm focuses on distributing slack in the schedule to maximize the average QoS. The processor bandwidth of the slack is calculated in a polynomial time and used by the QoS controller to construct a linear optimization problem that are solved on-line. For each of these scheduling algorithms, a resource access control method is developed for imprecise tasks. These methods allow jobs ready for optional part to safely access resources without having terminated in the midst of an access. Moreover, the methods allow aperiodic jobs to share resources with periodic jobs without causing a deadline miss. The effectiveness of the presented scheduling algorithms is shown by simulation studies. The results of simulations showed that the presented scheduling algorithms are capable of achieving higher effective processor utilization than classical theory. It was also shown that the presented approaches outperform the classical scheduling approaches when estimation of worst case execution time is loose.

Further contributions of practical interest are achieved by developing an embedded real-time operating system for imprecise computation. Through the implementation of the presented scheduling algorithms and supporting mechanisms for practical imprecise computations, issues concerning the practical imprecise computation model are also identified and considered. The imprecise computation support in the developed operating system is primarily accomplished by exchanging pieces of necessary information between the kernel and applications through the uniform scheduling interface. The numerical characteristics of the operating system provide support for the practicality of the approach.

Finally, the largest contribution is made by integrating all the above as a systematic approach. The whole approach enables the design of real-time systems to be considered from wider aspects. In particular, the presented approach, designed from the beginning to work under overloaded conditions, provides developers of real-time systems more freedom in design than the classical theory with little or no consideration for overload.

9.2 Future Directions

This research opens up several paths toward the goal of further increasing the effective processor utilization without a single timing violation.

A simple extension of this research is to consider whether wind-up operations need to be executed on all occasions. In particular, it is not always necessary to execute wind-up operations when its preceding optional part is completely discarded. Thus, in this case, the computation time reserved for the wind-up operations could be made available to other tasks earlier than the completion time. This early reclaiming increases the

acceptance ratio of aperiodic tasks and the performance of systems measured in terms of reward. The drawback of the extension is that scheduling algorithms become more complicated.

The M-FWP scheduling algorithm could cause a problem if the transient overload lasted for a long period of time, since it does not consider QoS of each task. One possible workaround is to switch to the SS-OP algorithm when the system was found to be under overload longer than expected. It is, however, possible that the overload is further lengthened by this switching of scheduling algorithms. Moreover, in order to avoid continuously switching back and forth between the two scheduling algorithms unnecessarily, the system must be able to predict the type of future overload. Thus, a sophisticated load prediction mechanism is required to actually accomplish dynamic switching of scheduling algorithms. Another room of improvement for the M-FWP algorithm is its run-time overhead. Since the time allocated to optional parts are dynamically calculated each time an optional part is requested, the M-FWP scheduler has larger run-time overhead than the EDF scheduler.

For the SS-OP algorithm, the area that needs the refinement most keenly is the QoS controller. The QoS controller solves the optimization problem with an extra constraint that the amount of time given to optional parts of a periodic task is constant. However, since the execution time must be of integer value in practice, this additional constraint leaves some amount of slack unusable by periodic tasks even under persistent overload.

The RT-Frontier operating system can support a wider range of applications effectively if the mechanism used for executing wind-up operations could be used for handling general timing faults. For instance, invoking different wind-up operations depending on whether the fault is due to overload or inaccurate worst case execution time seems attractive, because real-time software programmers are becoming to take these cases more seriously as the worst case execution time becomes more and more inaccurate.

For all these directions, a starting point should be found in the research presented in this dissertation that systematically considers scheduling of practical imprecise tasks from the computation model to implementation.

BIBLIOGRAPHY

[1] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *Journal of the Association for Computing Machinery*, vol. 20, no. 1, pp. 46–61, Jan. 1973.

[2] G. C. Buttazzo, "Rate Monotonic vs. EDF: Judgment Day," *Real-Time Systems*, vol. 29, no. 1, pp. 5–26, Jan. 2005.

[3] J. A. Stankovic, M. Spuri, K. Ramamritham, and G. C. B. azzo, *Deadline Scheduling for Real-Time Systems*. Kluwer Academic Publishers, 1998.

[4] 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*, June 1999, pp. 287–296.

[5] G. C. Buttazzo and J. A. Stankovic, "RED: Robust Earliest Deadline Scheduling," University of Massachusetts, Tech. Rep., 1993.

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

[7] 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, Sept. 1997.

[8] 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*, May 1993, pp. 76–80.

[9] X. Huang and A. M. K. Cheng, "Applying Imprecise Algorithms to Real-Time Image and Video Transmission," in *Proceedings of Real-Time Technology and Applications Symposium*, May 1995, pp. 96–101.

[10] 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*, Sept. 1997, pp. 390–397.

[11] 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, June 1999.

[12] 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*, 1994, pp. 202–208.

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

[14] 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*, July 1998, pp. 246–255.

[15] 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*, July 1999, pp. 519–524.

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

[17] 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 & Automation*, May 1999, pp. 1347–1352.

[18] 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*, Aug. 1993, pp. 1402–1407.

[19] 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, Jan. 1997.

[20] 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, Dec. 1993.

[21] 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*, May 2002, pp. 233–238.

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

[23] W. Feng and J. W. S. Liu, "Algorithms for Scheduling Real-Time Tasks with Input Error and End-to-End Deadlines," *IEEE Transactions on Software Engineering*, vol. 23, no. 2, pp. 93–106, Feb. 1997.

[24] J. K. Dey, J. F. Kurose, D. Towsley, C. M. Krishna, and M. Girkar, "Efficient On-Line Scheduling for a Class of IRIS Real-Time Tasks," *ACM SIGMETRICS Performance Evaluation Review*, vol. 21, no. 1, pp. 217–228, June 1993.

[25] J. K. Dey, J. Kurose, and D. Towsley, "On-Line Scheduling Policies for a Class of IRIS (Increasing Reward with Increasing Service) Real-Time Tasks," *IEEE Transactions on Computers*, vol. 47, no. 7, pp. 802–813, July 1996.

[26] T. Dean and M. Boddy, "An Analysis of Time-Dependent Planning," in *Proceedings of 7th National Conference on Artificial Intelligence*, Aug. 1988, pp. 49–54.

[27] S. J. Russell and S. Zilberstein, "Composing Real-Time Systems," in *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, Aug. 1991, pp. 212–217.

[28] G. Koren and D. ShaSha, "Skip-Over: Algorithms and Complexity for Overloaded Systems that Allow Skips," in *Proceedings of the 16th IEEE Real-Time Systems Symposium*, Dec. 1995, pp. 110–117.

[29] M. Hamadaoui and P. Ramanathan, "A Dynamic Priority Assignment Technique for Streams with (m, k)-Firm Deadlines," *IEEE Transactions on Computers*, vol. 44, no. 12, pp. 1443–1451, Dec. 1995.

[30] R. West and karsten Schwan, "Dynamic Window-Constrained Scheduling for Multimedia Applications," in *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, vol. 2, June 1999, pp. 87–91.

[31] G. C. Buttazzo, G. Lipari, and L. Abeni, "Elastic Task Model for Adaptive Rate Control," in *Proceedings of the 19th IEEE Real-Time Systems Symposium*, Dec. 1998, pp. 286–295.

[32] A. J. Garvey and V. R. Lesser, "Design-to-Time Real-Time Scheduling," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 23, no. 6, pp. 1491–1502, Nov./Dec. 1993.

[33] D. J. Musliner, E. H. Durfee, and K. G. Shin, "CIRCA: A Cooperative Intelligent Real-Time Control Architecture," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 23, no. 6, Nov./Dec. 1993.

[34] T.-W. Kuo and A. Mok, "Load Adjustment in Adaptive Real-Time Systems," in *Proceedings of the Twelfth IEEE Real-Time Systems Symposium*, Dec. 1991, pp. 160–170.

[35] D. Seto, J. P. Lehoczky, L. Sha, and K. G. Shin, "On Task Schedulability in Real-Time Control Systems," in *Proceedings of the 17th IEEE Real-Time Systems Symposium*, Dec. 1996, pp. 13–21.

[36] K. G. Shin and C. L. Meissner, "Adaptation and Graceful Degradation of Control System Performance by Task Reallocation and Period Adjustment," in *Proceedings of the 11th Euromicro Conference on Real-Time Systems*, June 1999, pp. 29–36.

[37] A. A. Avižienis, *The Methodology of N-Version Programming*. John Wiley & Sons, 1995, ch. 2.

[38] J. A. Stankovic, M. Spuri, M. D. Natale, and G. C. Buttazzo, "Implications of Classical Scheduling Results for Real-Time Systems," *IEEE Computer*, vol. 28, no. 6, pp. 16–25, June 1995.

[39] W. K. Shih, J. W. S. Liu, and J. Y. Chung, "Algorithms for Scheduling Imprecise Computations with Timing Constraints," *SIAM Journal on Computing*, vol. 20, no. 3, pp. 537–552, June 1991.

[40] W. K. Shih and J. W. S. Liu, "Algorithms for Scheduling Imprecise Computations with Timing Constraints to Minimize Maximum Error," *IEEE Transactions on Computers*, vol. 44, no. 3, pp. 466–471, Mar. 1995.

[41] W.-K. Shih, C.-R. Lee, and C.-H. Tang, "A Fast Algorithm for Scheduling Imprecise Computation with Timing Constraints to Minimize Weighted Error," in *Proceedings of the 21st IEEE Real-Time Systems Symposium*, Nov. 2000, pp. 305–310.

[42] H. Aydin, R. Melhem, D. Mossé, and P. Mejía-Alvarez, "Optimal Reward-Based Scheduling for Periodic Real-Time Tasks," *IEEE Transactions on Computers*, vol. 50, no. 2, pp. 111–130, Feb. 2001.

[43] W. K. Shih and J. W. S. Liu, "On-Line Scheduling of Imprecise Computations to Minimize Error," *SIAM Journal on Computing*, vol. 25, no. 5, pp. 1105–1121, Oct. 1996.

[44] S. K. Baruah and M. E. Hickey, "Competitive On-Line Scheduling of Imprecise Computations," *IEEE Transactions on Computers*, vol. 47, no. 8, pp. 1027–1032, Sept. 1998.

[45] J.-Y. Chung, J. W. S. Liu, and K.-J. Lin, "Scheduling Periodic Jobs That Allow Imprecise Results," *IEEE Transactions on Computers*, vol. 39, no. 9, pp. 1156–1174, Sept. 1990.

[46] J. Hansson, M. Thuresson, and S. Son, "Imprecise Task Scheduling and Overload Management using OR-ULD," in *Proceedings of the Seventh International Conference on Real-Time Computing Systems and Applications*, Dec. 2000, pp. 307–314.

[47] P. Mejía-Alvarez, R. Melhem, and D. Mossé, "An Incremental Approach to Scheduling during Overloads in Real-Time Systems," in *Proceedings of the Real-Time Systems Symposium*, Dec. 2000, pp. 283–293.

[48] J. Hansson, S. Son, J. Stankovic, and S. Andler, "Dynamic Transaction Scheduling and Reallocation in Overloaded Real-Time Database Systems," in *Proceedings of the 5th Conference on Real-Time Computing Systems and Applications*, Oct. 1998, pp. 293–302.

[49] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," *IEEE Transactions on Computers*, vol. 39, no. 9, pp. 1175–1185, Sept. 1990.

[50] M.-I. Chen and K.-J. Lin, "A Priority Ceiling Protocol for Multiple-Instance Resources," in *Proceedings of Twelfth Real-Time Systems Symposium*, Dec. 1991, pp. 140–149.

[51] K. Jeffay, "Scheduling Sporadic Tasks with Shared Resources in Hard-Real-Time Systems," in *Proceedings of the 13th Real-Time Systems Symposium*, Dec. 1992, pp. 89–99.

[52] T. P. Baker, "Stack-Based Scheduling of Realtime Processes," *Real-Time Systems*, vol. 3, no. 1, pp. 67–99, Mar. 1991.

[53] G. Lipari and G. Buttazzo, "Schedulability Analysis of Periodic and Aperiodic Tasks with Resource Constraints," *Journal of System Architecture*, vol. 46, no. 4, pp. 327–338, Feb. 2000.

[54] J. Huang, J. A. Stankovic, and K. Ramamritham, "On Using Priority Inheritance In Real-Time Databases," in *Proceedings of Twelfth Real-Time Systems Symposium*, Dec. 1991, pp. 210–221.

[55] T.-W. Kuo, M.-C. Liang, and L. Shu, "Abort-Oriented Concurrency Control for Real-Time Databases," *IEEE Transactions on Computers*, vol. 50, no. 7, pp. 660–673, Dec. 2001.

[56] A. Damm, J. Reisinger, W. Schwabl, and H. Kopetz, "The Real-Time Operating System of MARS," *ACM Operating Systems Review*, vol. 23, no. 3, pp. 141–157, July 1989.

[57] J. A. Stankovic and K. Ramamritham, "The Spring Kernel: A New Paradigm for Real-Time Systems," *IEEE Software*, vol. 8, no. 3, pp. 62–72, May 1991.

[58] K. Ramamritham, J. A. Stankovic, and P. Shiah, "Efficient Scheduling Algorithms for Real-Time Multiprocessor Systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, no. 2, pp. 184–194, Apr. 1990.

[59] W. Burleson, J. Ko, D. Niehaus, K. Ramamritham, J. A. Stankovic, G. Wallace, and C. Weems, "The Spring Scheduling Coprocessor: A Scheduling Accelerator," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 7, no. 1, pp. 38–47, Mar. 1999.

[60] R. Bettati, N. S. Bowen, and J.-Y. Chung, "On-Line Scheduling for Checkpointing Imprecise Computation," in *Proceedings of the Fifth Euromicro Workshop on Real-Time Systems*, June 1993, pp. 238–243.

[61] Y. Zhang and K. Chakrabarty, "Dynamic Adaptation for Fault Tolerance and Power Management in Embedded Real-Time Systems," *ACM Transactions on Embedded Computing Systems*, vol. 3, no. 2, pp. 336–360, May 2004.

[62] G. Bernat, J. Miro-Julia, and J. Proenza, "A Technique to Analyze the Tolerance to Transient Overloads of a Fault-Tolerant Real-Time Systems," in *Proceedings of High-Assurance Systems Engineering Workshop*, Aug. 1997, pp. 221–226.

[63] R. Rodrigues, M. Castro, and B. Liskov, "BASE: Using Abstraction to Improve Fault Tolerance," in *Proceedings of the 18th Symposium on Operating Systems Principles*, Oct. 2001, pp. 15–28.

[64] K. B. Kenny and K.-J. Lin, "Building Flexible Real-Time Systems Using the Flex Language," *IEEE Computer*, vol. 24, no. 5, pp. 70–78, May 1991.

[65] 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*, Mar. 1990, pp. 107–115.

[66] A. D. Stoyenko and W. A. Halang, "Extending Pearl for Industrial Real-Time Applications," *IEEE Software*, vol. 10, no. 4, pp. 65–74, July 1993.

[67] 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*, Apr. 1995, pp. 174–182.

[68] H. Tokuda, T. Nakajima, and P. Rao, "Real-Time Mach: Towards Predictable a Real-Time System," in *Proceedings of the USENIX Mach Workshop*, Oct. 1990, pp. 73–82.

[69] H. Tokuda and C. W. Mercer, "ARTS: A Distributed Real-Time Kernel," *ACM Operating Systems Review*, vol. 23, no. 3, pp. 29–53, July 1989.

[70] H. Tokuda, "Operating System Support for Continuous Media Applications – RT-Mach Extensions –," in *Proceedings of the Second Real-Time Computing Systems and Applications*, Oct. 1995, pp. 256–262.

[71] D. B. Stewart and P. K. Kohsla, "Mechanisms for Detecting and Handling Timing Errors," *Communications of the ACM*, vol. 40, no. 1, pp. 87–93, Jan. 1997.

[72] M. Humphrey and J. A. Stankovic, "Predictable Threads for Dynamic, Hard Real-Time Environments," *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 3, pp. 261–296, Mar. 1999.

[73] M. Spuri and J. A. Stankovic, "How to Integrate Precedence Constraints and Shared Resources in Real-Time Scheduling," *IEEE Transactions on Computers*, vol. 43, no. 12, pp. 1407–1512, Dec. 1994.

[74] S. K. Baruah, A. K. Mok, and L. E. Rosier, "Preemptively Scheduling Hard-Real-Time Sporadic Tasks on One Processor," in *Proceedings of 11th Real-Time Systems Symposium*, Dec. 1990, pp. 182–190.

[75] U. C. Devi, "An Improved Schedulability Test for Uniprocessor Periodic Task Systems," in *Proceedings of the 15th Euromicro Conference on Real-Time Systems*, July 2003, pp. 187–195.

[76] K. Albers and F. Slomka, "An Event Driven Approximation for the Analysis of Real-Time Systems," in *Proceedings of the 12th Euromicro Conference on Real-Time Systems*, July 2004, pp. 187–195.

[77] M. Spuri and G. Buttazzo, "Scheduling Aperiodic Tasks in Dynamic Priority Systems," *Journal of Real-Time Systems*, vol. 10, no. 2, pp. 179–210, Mar. 1996.

[78] M. Caccamo, G. Lipari, and G. Buttazzo, "Sharing Resources among Periodic and Aperiodic Tasks with Dynamic Deadlines," in *IEEE Real-Time Systems Symposium*, Dec. 1999, pp. 284–293.

[79] N. Yamasaki, "Responsive Processor for Parallel/Distributed Real-Time Control," in *Proceedings of IEEE/RSJ International Conference on Interlligent Robots and Systems*, Oct. 2001, pp. 1238–1244.

[80] D. Ferrari and D. C. Verma, "A Scheme for Real-Time Channel Establishment in Wide-Area Networkds," *IEEE Journal on Selected Areas in Communications*, vol. 8, pp. 368–379, Apr. 1990.

[81] H. Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 1997.

[82] J. L. Henessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 3rd ed. Morgan Kaufmann, 2003, ch. 5.

List of Papers

Articles on Periodicals

- Hidenori Kobayashi and Nobuyuki Yamasaki, "Real-time Scheduling of Extended Imprecise Tasks with Resource Constraints," *IPSJ Transactions on Advanced Computing Systems*, vol. 46, no. SIG 16, pp. 69–84, Dec. 2005.
- Hidenori Kobayashi and Nobuyuki Yamasaki, "An Integrated Approach for Implementing Imprecise Computations," *IEICE Transactions on Information and Systems*, vol. E86-D, no. 10, pp. 2040–2048, Oct. 2003.

Articles on International Conference Proceedings

- Hidenori Kobayashi and Nobuyuki 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*, May 2004, pp. 255–264.
- Hidenori Kobayashi, Nobuyuki Yamasaki, and Yuichiro Anzai, "Scheduling Imprecise Computations with Wind-up Parts," in *Proceedings of 18th International Conference on Computers and Their Applications*, Mar. 2003, pp. 232–235.
- Hidenori Kobayashi, Nobuyuki Yamasaki, and Yuichiro Anzai, "A Real-Time Network Manager for Distributed Imprecise Computation," in *Proceedings of the IFAC Conference on New Technologies for Computer Control*, Nov. 2001, pp. 59–64.