Practical Imprecise Computation Model: Theory and Practice

Hiroyuki Chishiro and Nobuyuki Yamasaki

Department of Information and Computer Science, Keio University, Yokohama, Japan
{chishiro,yamasaki}@ny.ics.keio.ac.jp

Abstract

We introduce the research overview of the practical imprecise computation model to achieve imprecise real-time applications. The practical imprecise computation model has multiple mandatory parts as real-time parts and multiple optional parts as non-real-time parts. We explain a new concept of real-time scheduling in the practical imprecise computation model, called semi-fixed-priority scheduling. In addition, we explain a semi-fixed-priority scheduling algorithm, called Rate Monotonic with Wind-up Part (RMWP). RMWP schedules each part in the practical imprecise computation model in Rate Monotonic order. We also introduce a real-time operating system for semi-fixed-priority scheduling algorithms, called RT-Est. We describe programming paradigms for the practical imprecise computation model in RT-Est. RT-Est has the SIM architecture for simulating real-time scheduling algorithms and the visualization tool for drawing simulation results. Finally we give future research directions for the practical imprecise computation model in theory and practice.

1. Introduction

Real-time applications have been encountering overloaded conditions in dynamic environments. For example, autonomous mobile robots [22, 19] run in such dynamic environments. In addition, robots perform tasks to detect and avoid obstacles periodically by many sensors. In order to meet the deadline of each task and support overloaded conditions, an imprecise computation model [17] is presented.

The imprecise computation model has the advantage of supporting overloaded conditions in dynamic environments, compared to Liu and Layland's model [18]. The important point of the imprecise computation model is that the computation in each task is split into two parts: a mandatory part and an optional part. A mandatory part as a real-time part affects the correctness of the result and an optional part as a non-real-time part only affects the quality of the result. By restricting the execution of the optional part only after the completion of the mandatory part, imprecise real-time applications can provide the correct output with lower quality, by terminating the optional part. Under overloaded conditions, the imprecise task terminates its optional part and generates the result with lower quality. However, an imprecise computation model is not practical because the termination of each optional part cannot guarantee the schedulability. In order to guarantee the schedulability of the termination of the optional part, a practical imprecise computation model [12] is presented.

The practical imprecise computation model has multiple mandatory parts as real-time parts and multiple optional parts as non-real-time parts. The execution flow of the practical imprecise task is that mandatory parts and optional parts are executed one after the other. Note that the first and last parts are mandatory parts. After executing optional parts, the practical imprecise computation model executes mandatory parts to terminate or complete the processing of optional parts. An example of mandatory part is to output the result to the actuator for controlling robots.

We have performed the research and development of the practical imprecise computation model. Now we introduce the research overview of the practical imprecise computation model. We explain a new concept of real-time scheduling in the practical imprecise computation model, called semi-fixed-priority scheduling [5]. In addition, we explain a semi-fixed-priority scheduling algorithm, called Rate Monotonic with Wind-up Part (RMWP) [5]. RMWP schedules each part in the practical imprecise computation model in Rate Monotonic (RM) order [18]. We also introduce a real-time operating system for semi-fixed-priority scheduling algorithms, called RT-Est [7]. We describe programming paradigms for the practical imprecise computation model in RT-Est. RT-Est has the SIM architecture for simulating real-time scheduling algorithms and the visualization tool for drawing simulation results. Finally we give future research directions for the practical imprecise computation model in theory and practice.

The remainder of this paper is organized as follows. Section 2 introduces system model for the practical imprecise computation model. Section 3 explains semi-fixed-priority

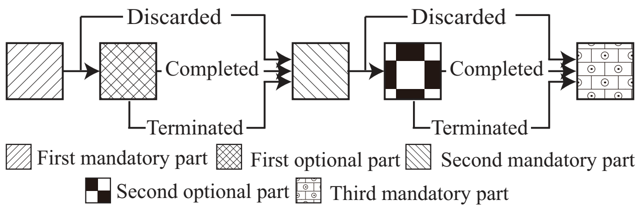

Figure 1. Practical imprecise task with three mandaotry parts and two optional parts

scheduling. Section 4 introduces a real-time operating system for semi-fixed-priority scheduling algorithms, called RT-Est. Section 5 compares our work with related work. Section 6 offers conclusion and gives future research directions.

2. System Model

2.1 Practical Imprecise Computation Model

The practical imprecise computation model [12] has multiple mandatory parts and multiple optional parts. Thanks to the following mandatory parts after executing optional parts, each practical imprecise task guarantees the schedulability of the processing to output the result. Figure 1 shows the practical imprecise task with three mandatory parts and two optional parts. Each practical imprecise task has three execution paths for optional part: discarded, completed, and terminated. Therefore, each practical imprecise task can output the proper quality of result without deadline miss due to the overrun of the optional part.

This paper assumes that the system has M identical processors and a task set Γ consisting of n periodic tasks with implicit deadlines. Task τ_i is represented as the following tuple $((m_i), (o_i), (OD_i), D_i, T_i)$: where m_i is the total Worst Case Execution Time (WCET) of the mandatory parts, o_i is the total Required Execution Time (RET) of the optional parts, OD_i is the group of the relative optional deadline, D_i is the relative deadline, and T_i is the period. The total WCET of mandatory parts of task τ_i is $m_i = \sum_{l=1}^{n_i^m} m_i^l$, where n_i^m is the number of mandatory parts and m_i^l is the WCET of the l^{th} mandatory part. On the other hand, the total RET of optional parts is $o_i = \sum_{l=1}^{n_i^o} o_i^l$, where n_i^o is the number of optional parts and o_i^l is the RET of the l^{th} mandatory part. The group of the relative optional deadline of task τ_i is $OD_i = \{OD_i^1, OD_i^2, ..., OD_i^{n_i^o}\}$, where OD_i^l is the l^{th} relative optional deadline of task τ_i. Since the optional deadline is a time to terminate an optional part, the number of optional deadlines is equal to that of optional parts n_i^o. The detail of the optional deadline is shown in Subsection 2.2. The relative deadline D_i of each task τ_i is equal to its period T_i. The j^{th} instance of task τ_i is called job $\tau_{i,j}$. The utilization of each task is defined as $U_i = m_i/T_i$. The reason why U_i does not include o_i is because the optional part of task τ_i is a non-real-time part, so that completing it is not relevant to scheduling the task set successfully. Hence, the system utilization within n tasks can be defined as $U = \sum_i U_i/M$. All tasks are ordered by increasing their periods and task τ_1 has the shortest period in the task set.

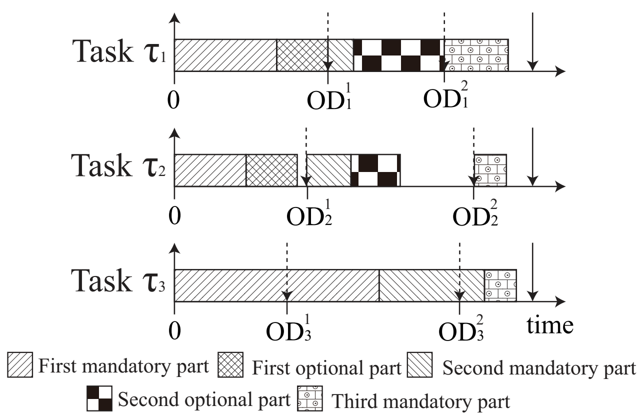

Figure 2. Optional deadline

2.2 Optional Deadline

An optional deadline is defined as a time when an optional part is terminated and a following mandatory part is released. Each following mandatory part is ready to be executed after each optional deadline and can be completed if each mandatory part is completed by each optional deadline. Each optional deadline is set to the time as late as possible to expand the executable range of each optional part. The following mandatory part must not miss its deadline if there is idle processor time or lower priority tasks are executed between the time when the mandatory part is completed and the following mandatory part is released. If each task does not complete its mandatory part until its corresponding optional deadline, the task may miss its deadline. Thanks to this definition, semi-fixed-priority scheduling does not degrade the schedulability compared to fixed-priority scheduling [5, 6].

Figure 2 shows the behavior of optional deadlines. Solid up arrow, solid down arrow, and dotted down arrow represent release time, deadline, and optional deadline, respectively. Now we describe the execution of each task.

- Task τ_1 executes the first mandatory part. When task τ_1 completes the first mandatory part, task τ_1 executes the first optional part. When OD_1^1 expires, task τ_1 terminates the first optional part and executes the second

Table 1. Optional deadline

Algorithm	Optional Deadline with Two Mandatory Parts	Optional Deadline with Multiple Mandatory Parts
RMWP	$OD_k^1 = \max(0, D_k - m_k^2 - \sum_{i<k} \left\lceil \frac{T_k}{T_i} \right\rceil m_i)$ [5]	$OD_k^l = \begin{cases} \max(0, D_k - m_k^{n_k^m} - \sum_{i<k} \left\lceil \frac{T_k}{T_i} \right\rceil m_i) & (l = n_k^o) \\ \max(0, OD_k^{l+1} - m_k^{l+1} - o_k^{l+1}) & (l < n_k^o) \end{cases}$ [9]
G-RMWP	$OD_k^1 = \begin{cases} \max(0, D_k - m_k^2) & (k \leqq M) \\ \max(0, D_k - m_k^2 - \hat{I}_k) & (k > M) \end{cases}$ [6]	$OD_k^l = \begin{cases} \max(0, D_k - m_k^{n_k^m}) & (k \leqq M \text{ and } l = n_k^o) \\ \max(0, D_k - m_k^{n_k^m} - \hat{I}_k) & (k > M \text{ and } l = n_k^o) \\ \max(0, OD_k^{l+1} - m_k^{l+1} - o_k^{l+1}) & (l < n_k^o) \end{cases}$ [9]

Table 2. Least upper bound

Algorithm	Imprecise	Multiprocessor	Least Upper Bound
RMWP	✓		$U_{lub} = n(2^{1/n} - 1)$ [5]
G-RMWP	✓	✓	$U_{lub} = \frac{M}{2}(1 - U_{max}) + U_{max}$ [6]
RM			$U_{lub} = n(2^{1/n} - 1)$ [18]
G-RM		✓	$U_{lub} = \frac{M}{2}(1 - U_{max}) + U_{max}$ [2]

mandatory part. After completing the second mandatory part, task τ_1 executes the second optional part. When OD_1^2 expires, task τ_1 executes the third mandatory part.

- Task τ_2 executes the first mandatory part and then the first optional part. When task τ_2 completes the first optional part, OD_2^1 does not expire, so that task τ_2 sleeps until OD_2^1. Next task τ_2 executes the second mandatory part and the second optional part and sleeps until OD_2^2. After OD_2^2, task τ_2 executes the third mandatory part.

- Task τ_3 does not execute the first optional part because task τ_3 does not complete the first mandatory part until OD_3^1. Since task τ_3 does not complete the second mandatory part until OD_3^2, task τ_3 does not execute the second optional part and executes the third mandatory part.

3. Semi-Fixed-Priority Scheduling

Semi-fixed-priority scheduling [5] is defined as part-level fixed-priority scheduling in the practical imprecise computation model. That is to say, semi-fixed-priority scheduling fixes the priority of each part in the practical imprecise task and changes the priority of each practical imprecise task only in the two cases: (i) when the task completes its mandatory part and executes its following optional part; (ii) when the task terminates or completes its optional part and executes the following mandatory part.

We have proposed three semi-fixed-priority scheduling algorithms for uniprocessor scheduling, multiprocessor global scheduling, and multiprocessor partitioned scheduling, called Rate Monotonic with Wind-up Part (RMWP) [5], Global Rate Monotonic with Wind-up Part (G-RMWP) [6], and Partitioned Rate Monotonic with Wind-up Part (P-RMWP) [8]. RMWP is adapted to uniprocessor scheduling, G-RMWP is adapted to multiprocessor global scheduling, and P-RMWP is adapted to multiprocessor partitioned scheduling. Global scheduling permits tasks to migrate among processors dynamically and partitioned scheduling assigns tasks to processors statically.

RMWP-based schedulers including RMWP, G-RMWP, and P-RMWP manage tasks in a real-time queue, a non-real-time queue, and a sleep queue. A real-time queue manages tasks which are ready to execute their mandatory parts. A non-real-time queue manages tasks which are ready to execute their optional parts. RMWP and P-RMWP schedule tasks in each queue in RM order. In contrast, G-RMWP schedules tasks in each queue in Global Rate Monotonic (G-RM) order. A sleep queue manages tasks which wait until next releases when they complete their jobs or next optional deadlines when they complete their previous optional parts.

Table 1 shows optional deadlines with two and multiple mandatory parts in RMWP and G-RMWP. The optional deadline of each task is statically calculated by these equations before executing tasks. The worst case interference time $\hat{I}_k$ in G-RMWP is the same as that in G-RM, so that the detail of $\hat{I}_k$ is shown in [10]. P-RMWP uses equations of optional deadlines in RMWP after assigning tasks to processors.

Table 2 shows the least upper bounds of RMWP, G-RMWP, RM, and G-RM. The least upper bounds of RMWP and G-RMWP with multiple mandatory parts are the same as those with two mandatory parts, respectively. In addition, the partitioning test of P-RMWP uses the least upper bound of RMWP. In G-RMWP and G-RM, U_{max} is the maximum utilization of all tasks in each task set. We analyzed that the least upper bounds of RMWP and G-RMWP are the same as those of RM and G-RM, respectively.

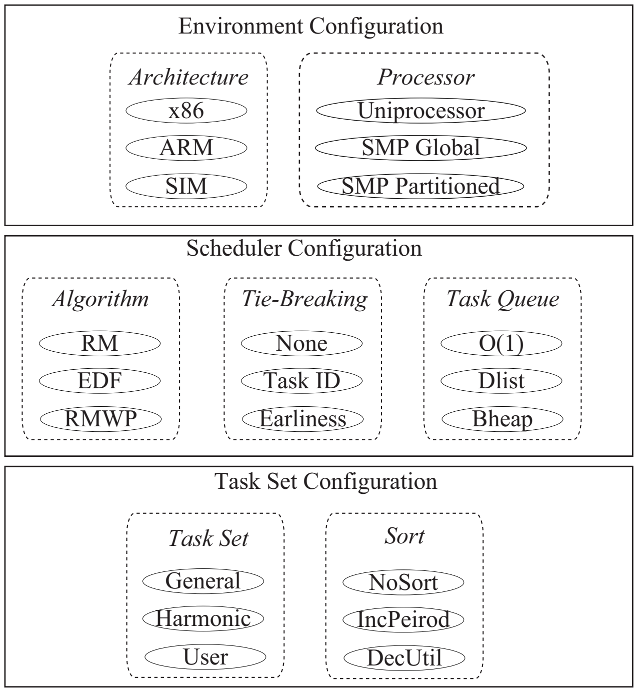

Figure 3. Congiurations of RT-Est

4. The RT-Est Real-Time Operating System

The RT-Est real-time operating system [7] has been developed from scratch to implement semi-fixed-priority scheduling algorithms in the practical imprecise computation model. First of all, we describe the configurations of RT-Est. Next we explain programming paradigms of the practical imprecise computation model. Then we introduce the SIM architecture for simulating real-time scheduling and drawing simulation results.

4.1 Configurations

RT-Est has many configurations to do many experiments including simulation studies and experimental evaluations. The primary motivation of supporting these configurations in RT-Est is to investigate the effectiveness of semi-fixed-priority scheduling in theory and practice. Using these configurations, developers can perform many experiments in many environments easily.

Figure 3 shows the configurations of RT-Est. There are three main configurations: an environment configuration, a scheduler configuration, and a task set configuration.

The environment configuration sets architectures (x86, ARM, and SIM) and processors (Uniprocessor, SMP Global, and SMP Partitioned). x86 is Intel and AMD's processors and ARM is ARM's processors. The detail of SIM is shown in Subsection 4.3. Uniprocessor is a single processor, SMP Global is SMP with global scheduling, and SMP Partitioned is SMP with partitioned scheduling. SMP Partitioned includes the following heuristic policies: first-fit, next-fit, best-fit, and worst-fit.

The scheduler configuration sets algorithms (RM, EDF, and RMWP), tie-breaking rules (None, Task ID, and Earliness), and task queues (O(1), Dlist, and Bheap). RM, EDF, and RMWP are RM, EDF [18], and RMWP algorithms, respectively. None does not perform tie-breaking. Task ID is that the smaller task ID has the higher priority. Here, the definition of earliness is the subtraction of the finishing time of the previous job (if it exists) from current time, so that smaller earliness has higher cache affinity. Therefore, Earliness is that the smaller earliness has the higher priority. O(1) is similar to the $O(1)$ scheduler in the old version of the Linux kernel. Dlist is the double circular linked list to manage tasks and Bheap is the binomial heap queue [26] to manage tasks.

The task set configuration sets task sets (General, Harmonic, and User) and sorting policies (NoSort, IncPeriod, and DecUtil). General is that task sets have no relationship with each other. Harmonic is that periods of tasks are integer multiples of each other, and User is user defined task set. NoSort does not sort task sets, IncPeriod sorts task sets by increasing periods, and DecUtil sorts task sets by decreasing utilizations.

4.2 Programming Paradigms

In this subsection, we introduce programming paradigms of the practical imprecise computation model in C language. Using these programming paradigms for a reference, developers can develop imprecise real-time applications easily.

Figure 4 shows the pseudo code of the practical imprecise computation model. This practical imprecise task has three mandatory parts and two optional parts.

Each task saves its context including general purpose registers and the program counter in `save_context` function. If `save_context` function is called via the timer interrupt routine to terminate the optional part at the optional deadline, `save_context` function returns `MANDATORY2`. Otherwise `save_context` function returns `MANDATORY`.

Each task executes the first mandatory part in `exec_mandatory` function. After completing its mandatory part, each task calls `end_mandatory` function. If the return value of `end_madnatory` function is `DISCARD`, each task discards its optional part and executes the following mandatory part in `exec_mandatory2` function. Otherwise each task executes the first optional part in `exec_optional` function. If each task completes its optional part, each task calls `end_optional` function and executes the following mandatory part in

```
part = save_context();
switch (part) {
case MANDATORY:
  /* execute the first mandatory part */
  exec_mandatory();
  res = end_mandatory();
  if (res != DISCARD) {
    /* execute the first optional part */
    exec_optional();
    /* wait until the first optional deadline */
    end_optional();
  }
case MANDATORY2:
  /* execute the second mandatory part */
  exec_mandatory2();
  res = end_mandatory();
  if (res != DISCARD) {
    /* execute the second optional part */
    exec_optional2();
    /* wait until the second optional deadline */
    end_optional();
  }
case MANDATORY3:
  /* execute the third mandatory part */
  exec_mandatory3();
}
end_job();
```

Figure 4. Pseudo code of the practical imprecise computation model

`exec_mandatory2` function. If each task terminates its optional part at its optional deadline, the scheduler resumes its context and calls `save_context` function. In this case, the return value of `save_context` function is `MANDATORY2`, so that the resumed task executes the second mandatory part in `exec_mandatory2` function. In a similar way, each task executes the second optional part in `exec_optional2` function and the third mandatory part in `exec_mandatory3` function. After completing the third mandatory part, each task calls `end_job` function to complete its job.

Figure 5 shows a programming example of lip reading task to analyze what a speaker says [23]. There are three mandatory parts and two optional parts as well as Figure 4.

In `exec_mandatory` function, the task gets the visual information from camera in `get_visual_info` function and analyzes the visual information in `analyze_visual_info` function. There are some metrics to read the lip: the width of the lip, the height of the lip, and the variation of the height of the lip. In `analyze_visual_info` function, one of the metrics is executed to generate the result with low quality.

In order to improve the quality of result for analyzing visual information, `exec_optional` function calls `analyze_visual_info2` function and `analyze_visual_info3` function. These functions perform other metrics except the metric in `analyze_visual_info` function. Note that the task may terminate its execution while running in these functions and resume its context from `save_context` function in Figure 4.

```
struct info *visual_info, *auditory_info;

void exec_mandatory(void)
{
  visual_info = get_visual_info();
  analyze_visual_info(visual_info);
}

void exec_optional(void)
{
  analyze_visual_info2(visual_info);
  analyze_visual_info3(visual_info);
}

void exec_mandatory2(void)
{
  auditory_info = get_auditory_info();
  analyze_auditory_info(auditory_info);
}

void exec_optional2(void)
{
  analyze_auditory_info2(auditory_info);
}

void exec_mandatory3(void)
{
  do_sensor_fusion(visual_info, auditory_info);
}
```

Figure 5. Programming example of lip reading task

In `exec_mandatory2` function, the task gets the auditory information from microphone in `get_auditory_info` function and analyzes the auditory information in `analyze_auditory_info` function. There are also some metrics to analyze auditory information: log power spectrum and linear predictive coding-derived mel-cepstrum [20]. In `analyze_auditory_info` function, one of the metrics is also executed to generate the result with low quality.

In order to improve the quality of result for analyzing auditory information, `exec_optional2` function calls `analyze_auditory_info2` function. This function performs other metrics except the metric in `analyze_auditory_info` function as well as `exec_optional` function.

In `exec_mandatory3` function, the task calls `do_sensor_fusion` function to integrate the analysis results of visual and auditory information by neural network [16] or hidden markov model [21].

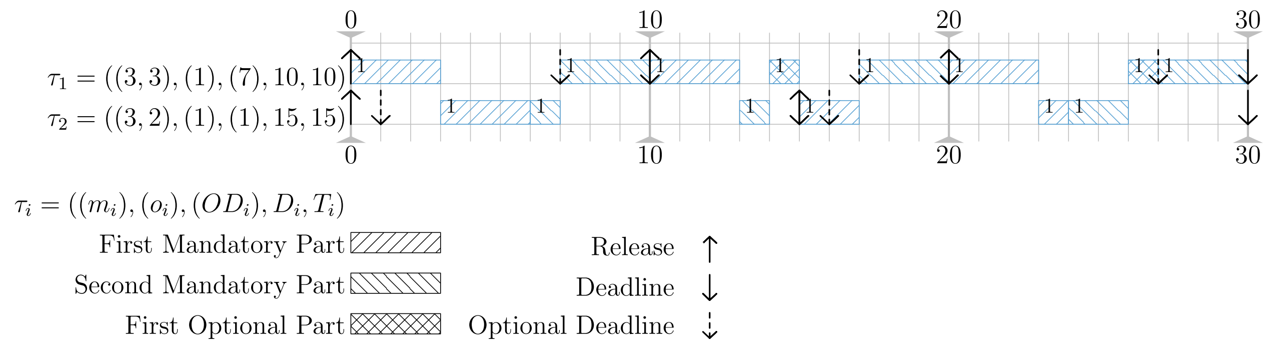

Figure 6. The simulation result in RMWP on uniprocessors

Table 3. Task set

Task	m_i	o_i	OD_i	D_i	T_i
τ_1	(3, 3)	(1)	(7)	10	10
τ_2	(3, 2)	(1)	(1)	15	15

4.3 The SIM Architecture

The SIM architecture is one of the architectures supported in RT-Est. The primary difference between the SIM architecture and other architectures such as x86 and ARM is that the SIM architecture is developed for simulating real-time scheduling algorithms. The SIM architecture has an aspect of debugging tool because we use architecture-independent code to other architectures for developing RT-Est easily.

RT-Est visualizes simulation results for real-time scheduling. Figure 6 visualizes the simulation result in RMWP on uniprocessors. The simulation length is 30 and the parameter of task set is shown in Table 3. In this example, each task has two mandatory parts and one optional part, so that the optional deadline of each task is calculated by $OD_k^1 = \max(0, D_k - m_k^2 - \sum_{i<k} \lceil T_k/T_i \rceil m_i)$ in Table 1. The parameter of each task is represented as $\tau_i = ((m_i),(o_i),(OD_i),D_i,T_i)$. Each part in the practical imprecise computation model is drawn as each pattern, respectively. The value drawn in the left of each task execution is the processor ID. Since the system has one processor, all values of processor ID are 1. Thanks to this tool, developers can check simulation results easily if they are correct.

Now we explain how to record scheduling events to generate Figure 6. When a job releases, the events of release, deadline, and optional deadline are recorded. When a job completes its mandatory part, the event of completing the mandatory part is recorded in `end_mandatory` function. When a job completes its optional part, the event of completing the optional part is recorded in `end_optional` function. Note that if a job terminates its optional part at its optional deadline, this event is not recorded. When a job completes its last mandatory part (e.g., the second mandatory part in Figure 6), the event of completing the job is recorded in `end_job` function.

5. Related Work

5.1 Imprecise Algorithms

There are some real-time scheduling algorithms based on imprecise computation.

Mandatory-First with Earliest Deadline (M-FED) [3] is based on EDF [18] in the imprecise computation model [17]. M-FED does not have the processing to terminate or complete the optional part. Optimization with Least-Utilization (OPT-LU) [1] requires the WCET of each optional part in the imprecise computation model. However, robots run in unknown environments, so that the WCET of each optional part becomes unknown. Therefore, M-FED and OPT-LU are not adapted to the practical imprecise computation model.

Mandatory-First with Wind-up Part (M-FWP) [13, 15] and Slack Stealer for Optional Parts (SS-OP) [14] were proposed to support the practical imprecise computation model. However, M-FWP and SS-OP are too complex to be adapted to multiprocessors because M-FWP and SS-OP calculate the assignable time of each optional part dynamically. In addition, such processing may cause high overhead.

In contrast, RMWP [5], G-RMWP [6], and P-RMWP [8] does not calculate the assignable time of each optional part dynamically, thanks to the optional deadline, so that RMWP can reduce the runtime overhead, compared to M-FWP and SS-OP.

5.2 Imprecise Operating Systems

There are some operating systems that actually implement the imprecise computation model [17].

The Concord system [17] supports the following methods in a client server structure: a milestone method and a sieve function method. The milestone method is used for the imprecise computation model with monotone optional parts. The milestone method saves intermediate results while an optional part is executed, so that the best result is immediately available on termination. The sieve function method is used for the imprecise computation model that has optional parts with 0/1 constraints. The sieve function method requires that the underlying operating system be capable of telling applications the amount of available computation time. Thus, the implementation of the method strongly depends on what scheduling algorithm is used by the operating system.

The imprecise computation server [11] was developed to support 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 [25]. The drawback of these checkpoint mechanisms is that the performance of the system can be degraded substantially, if overheads of checkpoints are not negligibly low.

In the ARTS system [24], a real-time object is defined with a time fence, which can be used to implement the imprecise computation model. The time fence specifies that a certain operation is executed within a predefined time. If this constraint cannot be met at run time, a handler is invoked. The handler should be implemented by the developer of the object to maintain its consistency on termination. Using this mechanism, the imprecise computation model can also be implemented effectively by making the handler to return the best result on termination.

These operating systems support the imprecise computation model and do not support the practical imprecise computation model [12].

RT-Frontier [14] was developed to support the practical imprecise computation model. However, RT-Frontier only supports uniprocessor scheduling such as M-FWP and SS-OP. In order to improve the quality of result and achieve high throughput, imprecise real-time applications usually require multiprocessors, so that RT-Frontier is not adapted to them.

RT-Est [7] was developed to support semi-fixed-priority scheduling in the practical imprecise computation model on multiprocessors. Therefore, RT-Est can support imprecise real-time applications requiring multiprocessors.

6. Conclusion and Future Research Direction

We introduced the research overview of the practical imprecise computation model to achieve imprecise real-time applications. We explained a new priority assignment policy for the practical imprecise computation model, called semi-fixed-priority scheduling, and some semi-fixed-priority scheduling algorithms on uniprocessors and multiprocessors. The RT-Est real-time operating system was developed to implement semi-fixed-priority scheduling algorithms. In addition, we explained programming paradigms and programming examples for the practical imprecise computation model. We developed the SIM architecture for simulating real-time scheduling and visualizing simulation results. We believe that RT-Est is widely used in the research and development of imprecise real-time applications.

We give some future research directions as follows.

- We will present optimal multiprocessor semi-fixed-priority scheduling. Optimal multiprocessor real-time scheduling can use full processor utilization. We believe that optimal multiprocessor semi-fixed-priority scheduling can improve the quality of result to make use of the remaining processor utilization compared to non-optimal multiprocessor semi-fixed-priority scheduling such as G-RMWP and P-RWMP.

- We will analyze the optional deadline to achieve the best reward (quality of result) in semi-fixed-priority scheduling. We proposed one of the approaches to calculate the optional deadline in order to avoid deadline miss if each task overruns, described in Section 3. If we use different approaches to calculate the optional deadline, the task may improve the reward.

- We will analyze overhead-aware schedulability of semi-fixed-priority scheduling. Generally, the overheads of real-time scheduling are mainly release, scheduler, context switch, and tick [4]. We consider how to set the overhead-aware optional deadline of each task. The optional deadline is set to the time as late as possible if the task does not miss its deadline. If the WCET of each task is underestimated, the task is overrunning, which may cause its deadline miss. Also, the overhead-aware reward should be investigated to clarify the difference between theory and practice. If these overheads are high, the overhead-aware reward becomes low.

- We will present the de facto standard for implementing the practical imprecise computation model. We introduce one of the approaches to implement the practical imprecise computation model in RT-Est. We also try to implement the practical imprecise computation model for other approaches and compare the effectiveness of existing approaches with considering usability, versatility, portability, and extendibility. The standard programming paradigms and APIs for the practical imprecise computation model are interesting.

Acknowledgement

This research was supported in part by CREST, JST. This research was also supported in part by Grant in Aid for the JSPS Fellows.

References

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

[2] T. P. Baker. An Analysis of Fixed-Priority Schedulability on a Multiprocessor. *Real-Time Systems*, 32(1-2):49–71, 2006.

[3] S. K. Baruah and M. E. Hickey. Competitive On-line Scheduling of Imprecise Computations. *IEEE Transactions on Computers*, 47:1027–1033, 1996.

[4] B. B. Brandenburg. *SCHEDULING AND LOCKING IN MULTIPROCESSOR REAL-TIME OPERATING SYSTEMS*. PhD thesis, The University of North Carolina at Chapel Hill, 2011.

[5] H. Chishiro, A. Takeda, K. Funaoka, and N. Yamasaki. Semi-Fixed-Priority Scheduling: New Priority Assignment Policy for Practical Imprecise Computation. In *Proceedings of the 16th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 339–348, Aug. 2010.

[6] H. Chishiro and N. Yamasaki. Global Semi-fixed-priority Scheduling on Multiprocessors. In *Proceedings of the 17th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 218–223, Aug. 2011.

[7] H. Chishiro and N. Yamasaki. RT-Est: Real-Time Operating System for Semi-Fixed-Priority Scheduling Algorithms. In *Proceedings of the 2011 International Symposium on Embedded and Pervasive Systems*, pages 358–365, Oct. 2011.

[8] H. Chishiro and N. Yamasaki. Experimental Evaluation of Global and Partitioned Semi-Fixed-Priority Scheduling Algorithms on Multicore Systems. In *Proceedings of the 15th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, pages 127–134, Apr. 2012.

[9] H. Chishiro and N. Yamasaki. Semi-Fixed-Priority Scheduling with Multiple Mandatory Parts. In *Proceedings of the 16th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, June 2013.

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

[11] 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*, pages 174–182, Apr. 1995.

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

[13] H. Kobayashi and N. Yamasaki. An Integrated Approach for Implementing Imprecise Computations. *IEICE transactions on information and systems*, 86(10):2040–2048, 2003.

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

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

[16] P. Kritsada and K. O. Yang. Sensor Fusion by Neural Network and Wavelet Analysis for Drill-Wear Monitoring. *Journal of Solid Mechanics and Materials Engineering*, 4(6):749–760, 2010.

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

[18] C. Liu and J. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of the ACM*, 20:46–61, 1973.

[19] I. Mizuuchi, Y. Nakanishi, Y. Sodeyama, Y. Namiki, T. Nishino, N. Muramatsu, J. Urata, K. Hongo, T. Yoshikai, and M. Inaba. Advanced Musculoskeletal Humanoid Kojiro. In *Proceedings of the 2007 IEEE-RAS International Conference on Humanoid Robots*, pages 294–299, Nov. 2007.

[20] K. Shikano. Evaluation of LPC spectral matching measures for phonetic unit recognition. Technical Report CMU-CS-86-108, Carnegie Mellon University, 1986.

[21] A. Shintani, A. Ogihara, Y. Yamaguchi, Y. Hayashi, and K. Fukunaga. Speech Recognition Using HMM Based on Fusion of Visual and Auditory Information. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, 77(11):1875–1878, 1994.

[22] T. Taira, N. Kamata, and N. Yamasaki. Design and Implementation of Reconfigurable Modular Robot Architecture. In *Proceedings of the 2005 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3566–3571, Aug., 2005.

[23] K. Takahashi. Sensing System Integrating Audio and Visual Information : Concrete Examples of Sensor Fusion Systems. *Journal of the Institute of Electronics, Information, and Communication Engineers*, 79(2):155–161, 1996.

[24] H. Tokuda and C. W. Mercer. ARTS: A Distributed Real-Time Kernel. *ACM Operating Systems Review*, 23(3):29–53, July 1989.

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

[26] J. Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, 21:309–315, Apr. 1978.