An Integrated Approach for Implementing Imprecise Computations

Hidenori KOBAYASHI, Student Member and Nobuyuki YAMASAKI, Nonmember

SUMMARY The imprecise computation model is one of the flexible computation models used to construct real-time systems. It is especially useful when the worst-case execution times are difficult to estimate or the execution times vary widely. Although there are several ways to implement this model, they have not attained much attention of real-world application programmers due to their unrealistic assumptions and high dependency on the execution environment. In this paper, we present an integrated approach for implementing the imprecise computation model. In particular, our research covers three aspects. First, we present a new imprecise computation model which consists of a mandatory part, an optional part, and another mandatory part called wind-up part. This wind-up part allows application programmers to explicitly incorporate into their programs the exact operations needed for safe degradation of performance when there is a shortage in resources. Second, we describe a scheduling algorithm called Mandatory-First with Wind-up Part (MFWP) which is based on the Earliest Deadline First strategy. This algorithm, unlike scheduling algorithms developed for the classical imprecise computation model, is capable of scheduling a mandatory part after an optional part. Third, we present a dynamic priority server method for an efficient implementation of the MFWP algorithm. We also show that the number of the processed sequence at most needed per node is one. In order to estimate the performance of the proposed approach, we have implemented a real-time operating system called RT-Prud. The experimental analysis has proven its ability to implement tasks based on the imprecise computation model without requiring any knowledge on the execution time of the optional parts. Moreover, it also showed performance gain over the traditional checkpointing technique.

key words real-time systems, imprecise computation model, operating systems, scheduling algorithms

1. Introduction

One of the most important properties that a real-time system should have is predictability, which has been achieved through the use of the worst case model in the past. Several factors within modern real-time systems, however, have affected the situation in negative ways. First of all, development of complex hardware has made estimation of the worst case execution times substantially difficult. Cache, DMAs, and TLBs are all designed to boost the average performance at the expense of latency penalty in the rare cases. It is true, however, that these technologies contribute to the high performance of modern processors and that real-time system designers can not just stay away from them. 

The term ‘flexible computation model’ nowadays covers a large area. However, a group of flexible computation models which does not fully rely on the knowledge of the worst execution time resembles more or less the imprecise computation model presented in [15]. It logically splits one computation into two parts: a mandatory part and an optional part. The mandatory part is requested to complete before the deadline of the original computation in order to produce acceptable result of minimal quality. On the other hand, the optional part, which follows the former, merely enhances the quality of the result. Hence the optional part can be discarded or terminated in an overloaded condition. The IRRS (Increasing Reward with Interest Service) model described in [8] takes a similar approach. Although their model does not have a mandatory part, it is stated in the same paper that it can be converted to the imprecise
computation model and vice versa. Systolic concepts are also found in the artificial intelligence technologies as anytime algorithms, whose examples can be found in [6]. These approaches can also be combined with the imprecise computation model as shown in [4].

Previous research on the flexible computation model is mostly performed on the scheduling algorithm. Algorithms for minimizing the ratio of the uncompleted optional portions are presented in [14] and [15]. In [2], a competitive on-line scheduling algorithm is presented. Alternatively, the reward-based scheduling problem where the reward functions are linear, concave, and convex is addressed in [1]. The problem is that most of them remain in the theoretical work or only validated through simulation studies. Thus, some are not applicable to the class of computations targeted in this paper, in that they either require strong constraints on the execution environment or precise prior knowledge of optional parts. For example, OR-ULD presented in [11] requires the worst case execution time of each transaction to be given a priori. This restriction was not a problem in their work because it was targeted for database systems whose transactions have known time attributes. However, it becomes an unrealistic assumption in our target domain.

Other work is performed as a case study which incorporates flexible computation models into the design of various applications. For example, a scheme for performing congestion control on an ATM switch is presented in [9]. In [5], an imprecise compressed image transmission method is described. These schemes are, however, highly dependent on one specific system and cannot be directly applied to another. On the other hand, there had been no approach other than the checkpointing technique [3], which aims to provide run-time support for imprecise computations. For example, ICS (Imprecise Computation Server) presented in [12] ensures the minimum quality of result using checkpoints. However, since the checkpointing semantics is also very limited due to its high dependency on the system software.

3. Computation Model

In this section, we describe a novel form of imprecise computations called the imprecise computation with wind-up part model. The development of this new model was motivated by a mismatch between the structure of real-world applications and that of the classical imprecise computation model.

The classical imprecise computation model requires a mandatory part to be completed before executing its corresponding optional part so that the optional phase can be terminated at an arbitrary point. Although this strategy seems to work well with the checkpointing technique, it is often not sufficient for the termination of real-world applications. For example, the need for transmitting the imprecise result available on the termination can not be handled efficiently. This is because the classical imprecise computation model does not allow any mandatory part to follow its optional part, despite the fact that this transmission is mandatory. In other words, the problem is that real-world applications require some compensation codes for the early termination of optional parts, whereas the classical imprecise computation model forbids their existence.

We have solved this mismatch through adding another mandatory part, called the wind-up part, to the end of the classical imprecise computation model. The wind-up part allows programmers to explicitly specify exact operations needed to terminate the optional part. Since those operations are usually common for the normal completion and the early termination, the wind-up part is defined to be a part that is always executed at the end of the computation as shown in Fig. 1. In other words, the wind-up part is required to complete before its deadline regardless of whether the optional part is completed, terminated, or discarded.

Operations which belong to the wind-up part are diverse, whose possible examples include:

- notifying other parts of the system of its premature termination;
- transmitting the result to different nodes;
- unrolling the effect of half completed operations;
- releasing acquired locks; and
- storing the quality of the result for later statistical feedbacks.

The major advantage of this model over the classical imprecise computation model can be illustrated in an example where a server application is required to transmit the result to multiple clients on different nodes. In this example, application programmers usually desire the server to transmit the results as late as possible, because the results would be of higher quality if larger portion of the optional part is executed. However, if the transmission is only programmed at the end of the optional part, it may not occur at all on overloaded conditions. To avoid this situation, in a system where the checkpointing technique is performed, the underlying software needs to keep transmitting intermediate results at every checkpoints as shown in Fig. 2. However, these overheads may well be too large for an overloaded system. On the other hand, using the hard-
Table 1: Comparison of computation models

<table>
<thead>
<tr>
<th>Term</th>
<th>Traditional</th>
<th>Proposed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of parts</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Function to result</td>
<td>mandatory</td>
<td>mandatory, wind-up</td>
</tr>
<tr>
<td>Dependency on OS</td>
<td>high</td>
<td>low</td>
</tr>
<tr>
<td>Runtime overheads</td>
<td>high</td>
<td>low</td>
</tr>
</tbody>
</table>

Fig. 2 Transmitting intermediate results at checkpoints.

were supported reflective memory keeps the overhead of the checkpoint low but may not be acceptable in modern real-time systems where cost is an important factor.

The other advantage is that the application programs become less dependent on the underlying software. Since almost all the operating system used in a distributed system provides some way of communication between different nodes, a program which transmits the result in its wind-up part may also work in another system with a slight modification. Meanwhile, a program that requires transmission of the result at checkpoints cannot be easily ported to another system which does not implement the checkpointing technique, because all the transmission of the result were entirely left to the system software.

One possible drawback of the proposed model may be that it nonetheless requires certain operating system facilities. For example, a periodic computation terminated in the middle of its optional part needs to have its execution context prepared for the next cycle, which usually requires modification of information held within the operating system. However, operating systems which aim to support periodic imprecise computations generally need to provide this facility even if no wind-up part is supported. Thus, this is not a true drawback of the wind-up part. The other possible drawback is that more than one wind-up part that are essentially the same may coexist on the same node, which leads to an increase in the amount of memory required and the number of cache misses. However, its effect can be minimized if the common cases are put into a library. In that case, only operations that are application specific have a possibility of being redundant, which rarely occurs. Thus, we conclude that the advantage of the wind-up part is significantly larger than its possible drawbacks.

Comparing the classical imprecise computation model and the proposed model made in the above arguments are summarized in Table 1.

Finally, time attributes of a task based on the imprecise computation with wind-up part model is characterized with five given parameters. Here, we use the term ‘task’ to mean a logical unit of execution, and in the following, assume that a task denoted with a subscript has five parameters of the same subscript as \( r_i, m_i, w_i, d_i, T_i \) where:

- \( r_i \) is the release time of the task;
- \( m_i \) is the worst case execution time of the mandatory part;
- \( w_i \) is the worst case execution time of the wind-up part;
- \( d_i \) is the deadline of the task; and
- \( T_i \) is the period of the task.

We note that \( T_i \) is not specified for tasks other than the periodic tasks. We also note that the execution time of the optional part is intentionally omitted from the list. In many applications, the process of producing a logically correct result does not vary with input data and its execution time is relatively easy to estimate. On the contrary, the execution time needed to enhance the quality of result is often highly input driven and thus cannot be easily predicted. For instance, in a target tracking application, the mandatory part may correspond to analyzing just one target which was the nearest to itself, while the optional part corresponds to analyzing many targets as possible. Accordingly, in order to be realistic, the proposed model regards that the execution time of the optional part is not given in any form.

4. Scheduling Algorithms

An algorithm for scheduling tasks based on the imprecise computation with wind-up part model needs to guarantee that every wind-up part completes before its deadline. It should also follow the precedence constraint shown in Fig. 1. Additionally, the wind-up part should be executed in a preemptive context, because its execution time can be arbitrarily long. Since there was no scheduling algorithm appropriate for this, we have developed a new scheduling algorithm called Mandatory-First with Wind-up Part (M-FWP).

4.1 Task Types

The M-FWP algorithm assumes that the system consists of three types of tasks shown in Table 2. In particular, a periodic task consists of an infinite number of jobs, while a sporadic task or an aperiodic task consists of only one job. Periodic tasks with hard deadlines are
Table 2: Type of tasks scheduled by M-FWP.

<table>
<thead>
<tr>
<th>task type</th>
<th>deadline type</th>
<th>computation model</th>
</tr>
</thead>
<tbody>
<tr>
<td>periodic</td>
<td>hard or firm</td>
<td>precise or imprecise</td>
</tr>
<tr>
<td>sporadic</td>
<td>firm</td>
<td>precise or imprecise</td>
</tr>
<tr>
<td>sporadic</td>
<td>none</td>
<td>precise</td>
</tr>
</tbody>
</table>

MQ

Tasks ready for the mandatory or wind-up part

OG

Tasks ready for the optional part

Aperiodic tasks

Fig. 3: Ready queues.

4.2 Mandatory-First with Wind-Up Part Algorithm

The basic strategy the M-FWP algorithm takes is to schedule tasks in a preemptive Earliest Deadline First (EDF) manner. There are however two exceptions to this. One exception is that the aperiodic tasks are scheduled in the background of the other tasks, because they are not associated with significant deadlines. In order to efficiently enforce this, we here assign aperiodic tasks the logically latest deadline. The other exception is that mandatory parts and wind-up parts always precede any optional parts, regardless of the task they belong to, so that no execution of the optional part forces any task to miss its deadline. On the other hand, with the aim of having as much execution time as possible for the optional part, the M-FWP algorithm schedules mandatory parts as early as possible and wind-up parts as late as possible.

These two exceptions force the M-FWP scheduler to manage two logical queues in the EDF order as in Fig. 3. One of them holds tasks which are ready for the mandatory part and those ready for the wind-up part. The other holds tasks which are aperiodic and those ready for the optional part. In the following, we refer to the former as the mandatory ready queue (MQ) and the latter as the optional ready queue (OQ).

In the beginning, all tasks are in the MQ. When a task in the MQ completes its mandatory part, the M-FWP scheduler puts it into the OQ. When its optional part is completed afterward, the scheduler puts it back to the MQ to execute its wind-up part. The superiority of the mandatory parts and the wind-up parts over the optional parts is accomplished by seeking a ready task first in the MQ.

The most crucial operation in the M-FWP algorithm is the detection of the correct timings for putting back tasks to the MQ, especially when the actual execution time of the optional part is larger than the amount of time available to it. The M-FWP algorithm satisfies this requirement by attaching every task a variable called the remaining. In the following, we use \( R_i(t) \) to denote the remaining of a task \( \tau_i \) at time \( t \).

The meaning of \( R_i(t) \) changes depending on which queue the task is in. When the task is in the MQ, \( R_i(t) \) corresponds to the amount of uncompleted portion for the mandatory part or the wind-up part, whichever is appropriate. The M-FWP algorithm accomplishes this by initializing \( R_i(t) \) to the worst case execution time of the ready part and decreasing it by the amount of time spent as each execution proceeds. On the other hand, when the task is in the OQ, \( R_i(t) \) represents the maximum amount of time that can be allocated to the optional part without missing its deadline. In other words, it means that if \( R_i(t) \) has become zero before the task completed its execution, its execution must be stopped immediately for its wind-up part to complete within its deadline.

In order to calculate this amount correctly, we first define a function which gives an upper bound for the amount of processing time that a periodic task \( \tau_i \) in the MQ spends in \((t_1, t_2)\) prior to tasks in the OQ as:

\[
D_i(t_1, t_2) = \min\left(m_i + w_i \cdot \left(t - t_1 \mod T_i \right) + \left\lceil \frac{m_i + w_i}{T_i} \right\rceil \cdot m_i + w_i \right).
\]

Since the execution time of the latest wind-up part may not have been executed before \( t_2 \) in practice, the value given by \( D_i(t_1, t_2) \) is at most \( w_i \) larger. However, a precise calculation requires the same computational complexity as the dynamic calculation of the system slack, which generally is impracticable. Using this estimation function, we now define function \( \alpha_i(t) \), which gives a lower bound for the amount of execution time that can be allocated to the optional part \( m_i, d_i \) as:

\[
\alpha_i(t) = d_i - w_i - t - \sum_{r_i \in \Gamma_i \cap \tau_i} R_i(t) - \sum_{r_i \in \Gamma_i \cap \tau_i} (R_i(t) + w_i) - \sum_{r_i \in \Gamma_i \cap \tau_i} D_{p_i}(t_0, T_{p_i}, d_i - w_i) - \sum_{r_i \in \Gamma_i \cap \tau_i} D_{a_i}(t_0, d_i - w_i),
\]

where

- \( \Gamma_i \) is a group of tasks in the MQ;
- \( \Gamma_i \cap \tau_i \) is a group of tasks in the OQ whose deadlines are earlier than or equal to that of \( \tau_i \).
\[ \Gamma_{a}(t) \] is a group of periodic tasks that belong to \( \Gamma_{a}(t) \), and
\[ \Gamma_{s}(t) \] is a group of periodic tasks which have finished executing their job in the last cycle but are not ready for the next one.

Owing to the error contained in calculation of \( D_{a}(t_{a}, t_{b}) \), the value of \( n_{a}(t) \) can theoretically be negative, even if incoming workloads are properly shaped by the admission control, which we will describe shortly. In that case, the M-FWP scheduler can simply set the value to zero.

The remaining of a task in the OQ is also gradually consumed except for cases: when another task \( \tau_{e} \) enters the OQ and when another task \( \tau_{r} \), newly arrives the system or is activated. In the first case, the tasks whose remaining need to be adjusted are those in the OQ with deadline later than \( d_{e} \), and the total amount of time that must be subtracted is \( n_{a}(t) \). In the second case, the affected tasks are those in the OQ whose deadline is later than \( d_{e} \), and the total amount of time subtracted is, depending on the type of \( \tau_{e} \):

\[
\sigma_{e}(t) = \begin{cases} 
D_{a}(\text{max}(t, d_{e}), d_{e}) & \text{if periodic} \\
0 & \text{otherwise} 
\end{cases}
\]

where \( t_{d} \) is the latest deadline among tasks in the OQ at that time. The adjustment of these remainings is carried out by subtracting as much remaining as possible from the above tasks in the EDF order until the total amount equals to the requested amount or no more task is left in the OQ. By the term ‘possible’, we mean that the remaining of the task is larger than zero.

Finally, the entire M-FWP scheduling algorithm is shown in Fig. 4. Although, each scheduling event and its operations are straightforward from the above arguments, it is worth mentioning that one scheduling event may occur as a consequence of another one. For instance, a task just enqueued to the head of the OQ, as in 1-3(c), may have the remaining of zero and thus immediately returns to the MQ, as in 1-5(c). It is also noteworthy that the unused execution time is automatically reclaimed among incomplete optional parts by passing the amount of remaining to the next task in the OQ.

4.3 Admission Control

The M-FWP algorithm does not assume the feasible mandatory constraint defined by Shih and Liu in [15], since the condition is generally not realistic. Hence, a system which adopts the M-FWP algorithm requires admission control to shape the incoming workloads so that every mandatory and wind-up part completes before its deadline.

The admission control of the M-FWP algorithm consists of two acceptance tests. The first test is theoretically supported by the schedulability analysis of

1. If one of the following event occurred:
   a. a task \( \tau_{r} \) has newly arrived or is activated:
      i. subtract a total of \( \sigma_{e}(t) \) from the remaining of tasks in the OQ whose deadline is later than \( d_{e} \),
      b. a task \( \tau_{r} \) has become ready:
         i. set \( R_{e}(t) \) to \( m_{r} \) and enqueue the task to the MQ,
   c. a task \( \tau_{r} \) has completed its mandatory part:
      i. set \( R_{e}(t) \) to \( \text{max}(R_{e}(t), 0) \),
      ii. move \( R_{e}(t) \) from the MQ to the OQ,
      iii. subtract a total of \( \sigma_{e}(t) \) from the remaining of tasks in the OQ whose deadline is later than \( d_{e} \),
   d. a task \( \tau_{r} \) at the head of the OQ has completed its optional part or \( R_{e}(t) \) has become zero:
      i. add \( R_{e}(t) \) to the remaining of the next ready task in the OQ if \( \sigma_{e}(t) \) exists,
      ii. set \( R_{e}(t) \) to \( 0 \),
      iii. move \( \tau_{r} \) from the MQ to the OQ,
   e. a task has completed its wind-up part:
      i. dequeue the task from the MQ,

2. If a ready task is found, by searching first in the MQ and then in the OQ, execute the task and decrease its remaining by the same amount.

3. Otherwise, do nothing.

Fig. 4 The M-FWP algorithm.

EDF [7] based on the density of a task which is defined as

\[
\sigma_{e}(t) = \begin{cases} 
m_{r} + w_{r} & \text{if periodic} \\
0 & \text{otherwise} 
\end{cases}
\]

for a task \( \tau_{r} \) based on our model. Upon an arrival of a sporadic task \( \tau_{e} \), we reject the task if the total density of the arrived task, those of the active periodic tasks in the system, and those of the already accepted sporadic tasks whose deadline is before \( d_{e} \) exceeds one. Similarly, we reject an activation of a periodic task if the total density of the activated task and those of all the already accepted tasks exceeds one.

If the first test is passed, the second test is performed to ensure that the already executed portions of the optional parts do not lead to any deadline miss. It is performed by checking the remaining of already accepted tasks while pretending that the newly arrived or activated task \( \tau_{e} \) is accepted. As is shown in Fig. 4, we re-calculate the remaining of every task in the OQ whose deadline is later than \( d_{e} \). If at least one of the newly calculated remainings was negative, we reject the task \( \tau_{e} \) because its acceptance may lead to a deadline miss.

These acceptance tests only form a sufficient condition, thus the admission control is correct but not optimal. However, we note again that the precise acceptance test requires the dynamic slack computation
process, which is computationally too complex for prac-
tical uses.

5. Operating System Support

In this section, issues concerning the implementation of the M-FWP algorithm in our custom operating system called RT-Frontier is described.

RT-Frontier is a real-time operating system which is designed with an aim to handle uncertainties that are often caused by the dynamic environment and to provide a reliable and predictable execution environ-
ment for real-time applications. Its general strategy is to take advantage of the potential flexibility contained within each activities. Specific parts of the RT-Frontier 
operating system related to the implementations of the M-FWP algorithm are a method for detecting overruns in optional parts, a scheme for supporting the execution of wind-up part, and the application program interface for specifying the structure of the imprecise computa-
tion with wind-up part model.

5.1 Overrun Detection

In order to implement the M-FWP algorithm, each scheduling event listed in Fig. 4 must be correctly de-
tected and notified to the scheduler. In order to do so, the RT-Frontier operating system provides two primitives to be embedded in application programs: end.mandatory() and end.optional(). They are im-
plemented as system calls which notifies the completion of a mandatory part and an optional part, respectively. The former primitive, in addition, has a return value which indicates whether the scheduler has decided to discard its optional part or not. Thus it allows the ap-
lication itself to dynamically skip its entire optional part without any further help from the operating sys-
tem. In addition, the completion of the wind-up part likewise needs to be notified to the scheduler. However, the completion of a wind-up part is, from the viewpoint of the scheduler, the same as the completion of an or-
dinary task without the optional part, thus the RT-
Frontier operating system only provides one primitive for both purposes: end.job().

By contrast, an application task generally is not capable of detecting, the exhaustion of its remaining. Thus, the scheduler must be equipped with the abil-
ity to detect this overrun. We here note that although this illegal overrun theoretically only occurs during the execution of optional parts, it is of paramount impor-
tance for real-time operating systems to detect viola-
tions of time attributes specified from application pro-
grammers. Since the resolution of the system tick man-
aged by a hardware timer in a typical real-time oper-
ating system is usually too coarse for managing the length of the execution, we use another hardware timer which interrupts the operating system on its expiration to detect any overrun caused by any part. The length measured by the timer is of course set to the remaining of the next running task. When a context switch must take place during its execution, this hardware timer for overrun detection is managed in the following steps:

1. stop the timer and update the remaining of the last running task reading the value of the hardware counter;
2. perform usual process for switching contexts;
3. set the counter of the timer according to the re-
maining of the next to be running task and start it again;

In this manner, the exhaustion of the remaining for an optional part is safely detected just before it overruns.

5.2 Wind-up Server Method

On the exhaustion of the remaining for an optional part, its owner task is required to switch its execution context to the beginning of the wind-up part. The prob-
lem is that an application program is never capable of jumping from an arbitrary point specified at run-time to another.

Our solution is to setup a dedicated task called the wind-up server. Its role is to execute the wind-up part on behalf of the task whose optional part is ter-
minal. An important point here is that the wind-up server should look exactly the same as the original task from the scheduler's point of view. To meet this re-
quirement, on the invocation of the wind-up server, its deadline is set to that of the original task and its re-
maining is set to the worst case execution time of the wind-up part which is to be executed. Since operations in the wind-up part widely differs between applications, the RT-Frontier operating system requires a function corresponding to the wind-up part be registered a pri-
ori.

Additionally, in order to schedule a wind-up server properly in a timely manner, the last two instructions of the step 1-(d) in Fig. 4 must be split and modified to:

1. decrease τi from the OQ;
2. if Ri(1) is zero and its optional part has already started its execution, enqueue the wind-up server to the MQ with its deadline and its remaining set to d, and w, respectively;
3. otherwise, set R(i) to w, and enqueue τi to the MQ.

An issue concerning the implementation of the wind-up server is the management of the context of the original task. If the task was sporadic or aperi-
odic, it will leave the system after the completion of its wind-up part and thus it will not become a prob-
lem. However, if the task was a periodic one, it of-
ten needs to begin its next cycle from the beginning.
of the mandatory part. For this purpose, we provide a primitive, `begin_mandatory()`, to let the application programmer mark the beginning of its mandatory part. The RT-Frontier operating system takes a snapshot of the context of the task on its invocation and later uses it to overwrite the terminated context when the execution of the wind-up server is finished.

Another practically important issue is the sufficient number of the wind-up server needed per node. We can actually limit this number to one, because multiple wind-up servers are only necessary when a wind-up server needs to be preempted by another one. This only occurs when a task with deadline earlier than that of an active wind-up server, has its optional part executed after the previous wind-up server is put to the MQ. However, this situation never happens due to the superiority of the MQ over the OQ.

5.3 Application Program Interface

When developing an application program based on the imprecise computation with wind-up part model on the RT-Frontier operating system, an application programmer can either use the primitives we have just described or an interface supplied in the library. The RT-Frontier provides:

- `create_impph(type, mfunc, ofunc, wfunc)`

The first argument of the function is the type of the task, followed by the name of the function corresponding to the mandatory part, the optional part, and then the wind-up part. Figure 3 shows the simplified internal structure of `create_impph()` in pseudo C code. Since any misuse of the primitives surely causes the scheduler to malfunction and therefore system to collapse, we strongly recommend that application programmers use the library function.

6. Experimental Results

Experiments were performed to estimate the effectiveness of the presented approach on the RT-Frontier operating system. The RT-Frontier operating system currently runs on a Responsive Processor whose processing core is a SPARCmicro which can achieve up to 121 relative MIPS. The Responsive Processor is equipped with an original network called Dispatcher. Details of the hardware are described in [16]. Throughout the experiments, the resolution of system tick used in the RT-Frontier operating system was set to 1 us. This system tick was managed using one of the hardware timers provided in the Responsive Processor.

First, we measured the worst case overheads of the four primitives provided in the RT-Frontier operating system. The result measured with a hardware timer whose resolution is 0.005 us when only one task existed in the system, is shown in Table 3. The overheads did not change even when the number of tasks in the system increased, except for `end_mandatory()`. Its largest overhead was measured when all the other tasks belonged to `Par`, and its approximation was given by $5.2 \times n + 48.2(\mu$s) where $n \geq 1$ is the number of the tasks in the system. Its relatively large overheads and the difference between the case when $n = 0$ and the value in Table 3 are both accounted for by the fact that the SPARCmicro in the Responsive Processor does not implement a divide instruction needed in Eq. (1), which forced the compiler to use a software library function instead.

We then, compared the performance of the M-FWP algorithm with that of the M-FED algorithm [2] to estimate the advantage of using wind-up parts. The M-FED algorithm was implemented with equally separated intermediate checkpoints. We denote the M-FED, the M-FWP, and the intermediate checkpoints in addition to the two essential checkpoints performed on the completion of the mandatory part and that of the optional part.

We set the execution time of the mandatory part to 4 ms and its period to 50 ms. The execution time for the optional part was randomly selected from a discrete uniform distribution of [25, 35] ms. In this experiment, we regarded that the optional part is monotone and that the quality of the result enhances linearly with the increasing execution time. Under this assumption, the length of the executed optional part indicates the quality of the result when the M-FWP algorithm is used. On the other hand, the executed optional part becomes void, that is, it contributes to the quality of the result, only after each checkpoint is performed when the M-FED, algorithm is used, which in

```
begin_mandatory();
for (; ; )
  mfunc();
  result = end_mandatory();
  if (result != DISCARDED)
    ofunc();
    end_optional();
  wfunc();
  end_job();
```

Table 3 Worst case overheads.

<table>
<thead>
<tr>
<th>primitives</th>
<th>overheads</th>
</tr>
</thead>
<tbody>
<tr>
<td>begin_mandatory()</td>
<td>30.05 µs</td>
</tr>
<tr>
<td>end_mandatory()</td>
<td>30.05 µs</td>
</tr>
<tr>
<td>end_optional()</td>
<td>10.70 µs</td>
</tr>
<tr>
<td>end_job()</td>
<td>20.40 µs</td>
</tr>
</tbody>
</table>
turn require no wind-up part however. The overheads of a checkpoint and a wind-up part were both set to 400 μs, which equals the worst case overhead of pushing out a 1 KB message to the Responsive Link. The condition set up for this experiment reflects characteristics and time attributes of tasks that capture, process, and transmit sensor informations that are found in many real-time systems. Similar experimental environments can also be found, for example, in [10].

The actual experiments were performed under two conditions in terms of sporadic task arrivals. The first experiment was performed when there were only off-line periodic tasks. An off-line task is one that always resides in the system and never is suspended, thus the variations in the workload are only that of the execution time in optional parts. Under this condition, the length of the executed optional part which contributed to the quality of the result was measured. In other words, since the performance of a system that is based on the imprecise computation model is generally expressed in terms of the quality of the result, we did not count in the length of the optional part which executed in vain. The result of this experiment is shown in Fig. 6. The term ‘essential load’ in the figure represents the total density of all the active tasks. Moreover, the length of valid execution time is normalized to the value within [0, 1] and is shown in the vertical axis. In the second experiment, sporadic tasks were set to arrive the system dynamically. Here, off-line periodic tasks also existed and the sum of their total densities accounted for 0.2 of the essential load. The relative deadlines of sporadic tasks were randomly selected from a discrete uniform distribution of [35, 45]. Similarly, their execution times were randomly selected from a discrete uniform distribution of [2, 3, 5, 7, 9] where the value of E was determined so that their total density accounted for 0.2 of the essential load. The result is shown in Fig. 7. We note that no significant result was obtained in the essential load was larger than 0.8 due to rejections of arrived tasks.

Both of the results showed a graceful decrease for all algorithms with respect to increase in the essential load, however, the curve of degradation for M-FED2 is not smooth enough that we could actually see that it almost forms a two-step stair. Another interesting point is that the M-FED2 performed better than the M-FED, when the essential load was 9.85, which was due to lesser cumulative overheads incurred with smaller number of checkpoints. On the other hand, when the essential load was higher, the M-FED2 outperformed the M-FED owing to its more frequent checkpoints. Finally, the M-FWP algorithm only had the overheads of only one wind-up part on the termination of each computation, which allowed it to achieve the highest performance under any conditions.

7. Conclusions

In this paper, we have presented a new paradigm and a practical method for constructing a real-time system which potentially contain a considerable amount of uncertainty in workloads.

One of the major contributions of this work is the improvement in the applicability of the flexible computation model. Quite unlike the existing model, the proposed model provides application programmers with the freedom and the clearer view on how the application would be handled on overloaded conditions. As a consequence, it increases the chance of imprecise approaches being adopted by application programmers who have given it up due to its high dependency on the system software. Another contribution is that it increases the portability of the applications. Since our approach does not require any application specific support in the operating system, it encourages the system designers to reuse the same software in another system, as differences between the execution environments are safely absorbed using the flexibility provided by optional parts. Furthermore, it contributes to the overall system performance. As was shown, the proposed
model and its implementation method succeeded in utilizing partial results with lesser run-time overheads than the checkpointing technique, which was the tra-
ditional and standard approach for implementing the classical imprecise computation model.

As for future work, in the hope of providing the ba-
sis for open real-time systems, we are currently working to inte-
grate methods for handling overruns caused by
precise computations, which do not have any optional
part. We are also investigating a method for estimating execu-
tion times at run-time in a way that suits impre-
Cise computations.

Acknowledgement

This study was performed through Special Coordi-
nation Funds of the Ministry of Education, Culture,
Sports, Science and Technology of the Japanese Gov-
ernment.

References

[4] J.R. Chen, "On applying imprecise computation to real-
Sixth International Conference on Computer Communica-
[8] J.K. Day, J. Kurucz, and D. Towsley, "On-line scheduling policies for a class of I/O-bound increasing reward with increas-
[10] G. Gourlay, B. Shrikumar, J.A. Stanovic, and K. Ra-
manathan, "Adaptive fault tolerance and graceful degra-
dation under dynamic hard real-time scheduling," Proc.
ing and load balancing algorithm using OR-ULD," Proc. Seventh International Conference on Real-Time Com-
[12] D. Hull, W. Feng, and J.S. Liu, "Enhancing the per-
formance and dependability of real-time systems," Proc.
IEEE International Computer Performance and Depend-
no.5, pp.1105-1122, 1996.
[16] N. Yamazaki, "Design and implementation of responsive processor for parallel/distributed control and its develop-
ment environment," J. Robotics and Mechatronics, vol.13,

Hidenori Kobayashi received the B.S. degree in information and computer science and the M.S. degree in science for open and environmental systems from Keio University in 1990 and 2002, respec-
tively. He is currently a Ph.D. student at the Graduate School of Keio Univers-
ity. His research interests include real-time operating systems, flexible comput-
tion models, distributed systems, and real-time scheduling algorithms. He is a student member of IPSJ.

Nobuyuki Yamazaki received the B.S. degree in physics, and the M.S. and Ph.D. degrees in computer science from Keio University, Yokohama, Japan, in 1991, 1993, and 1996, respectively. From 1996 to 1998, he was a researcher with Electronic Technology Laboratory. Since Octo-
ber 1998, he has been with the faculty of information and computer science, Keio University. Now, he is an assistant pro-
fessor with Keio University. He is a great researcher with Digital Woman Laboratory, AIST. He is also a researcher with GRIST, JST. His research interests include paral-
lel/distributed systems, real-time systems, computer architec-
ture, robotics, and VLSI design.