RT-Frontier: A Real-Time Operating System for Practical Imprecise Computation

Hidenori Kobayashi Nobuyuki Yamasaki
School of Science for Open and Environmental Systems
Graduate School of Keio University
Yokohama, 223-8522, Japan
E-mail: {kobahide,yamasaki}@ny.ics.keio.ac.jp

Abstract

Imprecise computation is known as an effective technique for dynamically resolving trade-offs between the amount of resources and the quality of the result. However, its implementation and operating system support methods have not been exploited enough from a practical point of view. This paper presents a new approach taken in the RT-Frontier operating system to support imprecise computation. Applications that allow imprecise computation are first transformed to tasks composed of three parts based on an extended imprecise computation model. All tasks are then uniformly scheduled according to a novel scheduling algorithm called Slack Stealer for Optional Parts (SS-OP). The SS-OP algorithm is designed to handle imprecise computations with small overhead, which is at a comparable level of that of the Earliest Deadline First (EDF) algorithm. The results of experiments show that the presented approach is cost-effective enough to be considered as a practical basis for embedded real-time systems.

1. Introduction

The real-time computing community has long relied on resource reservation approaches based on the worst case scenarios. However, the worst case execution time (WCET) of an activity is now harder to estimate than ever, due to the use of complex hardware, multiprogramming, and commercial off-the-shelf components. Additionally, the amount of resources required within real-time systems tend to change from time to time, because their behaviors are by nature subject to its environment. These situations imply that it is almost impossible or impracticable to determine what the real worst case is.

The imprecise computation model presented in [13] is one of the techniques used to cope with such uncertainty. The crucial point is that the computation is split into two parts: a mandatory part which affects the correctness of the result and an optional part which only affects the quality of the result. By restricting the execution of the optional part to only after the completion of the mandatory part, an application based on the imprecise computation model is able to provide a sane output with lower quality, by terminating the optional part whenever resources are scarce. Also, there is an advantage that the inevitable reservation can be made tighter, since the worst case scenario just for the mandatory part is often easier to determine than the one including the optional part.

The application domain of the imprecise computation model is diverse, some of which are multimedia processing [7, 10, 5], planning and artificial intelligence [21, 8, 16], and database systems [9, 1]. Despite a large number of application candidates, the imprecise computation model has not been widely accepted in industrial practice, partly due to its strong assumptions, described in the following sections, and partly due to the lack of cost-effective support methods that can be easily implemented in an embedded operating system.

This paper describes a novel approach that the RT-Frontier operating system takes to implement and support practical imprecise computation. The RT-Frontier operating system is developed from scratch for medium embedded real-time systems. These embedded systems require a mechanism to add flexibility and to resolve overload more keenly than the other systems, since no manual adjustment can be made at run-time.

Our goal is to support imprecise computation at low cost as a means to cope with transient overloads. Embedded systems often demand imprecise computation to be carried out at low cost rather than in an optimal manner for the following three reasons. First of all, these systems are usually constructed to be underloaded in the most of the expected conditions, in which all the optional parts can be completed. Secondly, cheap processors with limited pro-

Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'04)
1080-1812/04 $ 20.00 © 2004 IEEE

cessing power used in embedded systems cannot afford to execute optimization processes that are unnecessary while the system is underloaded. Thirdly, dynamic optimization processes during overload can increase the total workload even more, since they are mostly computation intensive. It must be noted that, nevertheless, the imprecise computation technique is still required when any timing fault can lead to a catastrophic situation.

The rest of this paper is organized as follows. Section 2 reviews related work. Section 3 describes the system model, which includes the extended imprecise computation that forms the basis of the following arguments. Section 4 addresses the problem of scheduling imprecise computation. Section 5 presents the experimental results and Section 6 concludes the paper.

2. Related Work

Most of the previous research on the imprecise computation model focused on scheduling algorithms that aim at minimizing the error in computations. The error of imprecise computation is usually given by a function, called an error function, which characterizes the relationship between the amount of resources and the error.

The off-line scheduling of imprecise computation has been studied quite extensively and several optimal algorithms have been presented in [18, 15, 2]. It is also proved that the problem becomes NP-Hard when tasks have 0/1 constraints [14] or the error function is concave [2].

As for the on-line scheduling of imprecise computation, it is known that there exists no algorithm that always finds a feasible schedule with the minimum total error whenever a feasible schedule exists [17]. Nonetheless, two strategies were developed. The mandatory first strategy statically assigns higher priorities to all mandatory parts than optional parts [6, 3], while the other strategy performs some form of slack stealing to allocate processor time for executing optional parts [17].

One of the common assumptions made throughout the above is that the optional parts have a constant upper bound on their execution time. However, the behavior of a real-time application is often subject to input data, and the time needed to produce a precise result is unlikely to be constant. For example, the execution time of an obstacle tracking application may depend on the number of obstacles identified at each instant. Another questionable assumption is that error functions are always given. Generally, determining an error function requires significant amount of empirical work. Moreover, a real-world error function is not promised to be expressed in a simple mathematical form.

The server mechanisms [19] developed for allocating execution time to aperiodic tasks in the presence of hard periodic tasks do not make the assumptions mentioned above, but they cannot be used directly to serve optional parts, because the optional parts already have deadlines whereas aperiodic tasks do not.

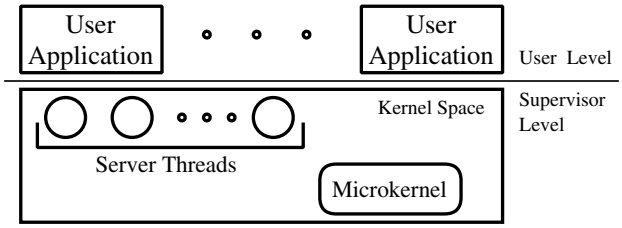

Figure 1. Structure of RT-Frontier

A little work focuses on the implementation platform and on run-time support. Concord [13] and ICE [11] provide support based on the client server model, which can be augmented by a checkpointing scheme [4]. Although checkpointing provides fault tolerance, it also increases the run-time overhead. Spring thread package [12] provides a platform for implementing imprecise computation, in which each part of the imprecise computation is mapped to a thread. While it provides great flexibility, spawning a thread can be expensive in terms of run-time overhead and memory consumption. Thus, a simpler approach may suit better as long as the mandatory and optional parts do not need to execute in parallel.

3. System Model of RT-Frontier

The overall structure of the RT-Frontier operating system is illustrated in Figure 1. The majority of the operating system services are provided by server threads. These threads are implemented inside the kernel space and run in supervisor mode. The microkernel of RT-Frontier implements the scheduler and is responsible for three things: handling hardware interrupts and exceptions, emitting signals to corresponding threads, and scheduling threads.

There are two ways an application can be implemented. One is to implement it as a user level task. The user level tasks receive services from the server threads via port-based communication or by entering supervisor mode via system calls. The other way is to implement it as one of the server threads, that is, make the application as a part of the operating system. The first way has an advantage in 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 easier and faster. 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

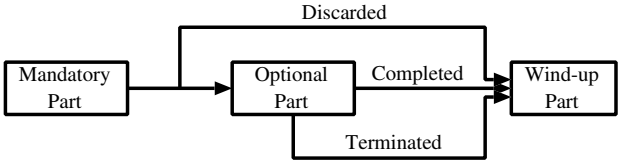

Figure 2. Imprecise computation with wind-up part

be critical enough to reside in the kernel space and to access peripheral I/O devices directly.

Applications implemented as above can freely contain computation with and without an optional part. However, the RT-Frontier operating system restricts applications that allow imprecise computation to conform to the original extended imprecise computation model described in the following.

The extended imprecise computation model, shown in Figure 2, is similar to the traditional imprecise computation model in a sense that the optional part follows the mandatory part. However, this extended model is significantly different from the traditional model in that an extra operation can be executed after the optional part is terminated prematurely. Consider the case where a periodic imprecise computation is terminated in its optional part. If its context is not modified at all, it will resume, in its next cycle, exactly at where it has been stopped. If this is not what the programmer intended and the intension was to execute the program from the top of the mandatory part in each cycle, some compensation operations are indispensable for fixing the context. Such operations are commonly desired, for example, for transmitting the results to other nodes, for unrolling the effect of prematurely terminated operations, and for performing statistical feedbacks based on the quality of the result.

In order to meet these demands, our extended model adds a new part called the wind-up part for putting these compensation operations. The wind-up part is required to be completed without exception, since operations in this part mostly need to be executed even when the optional part is completed. If the programmers do not want to have the wind-up part executed when the optional part is completed, they can still structure their programs to check whether the optional part is completed before executing the wind-up part. Or alternatively, the wind-up part may be structured as idempotent. By contrast, the traditional imprecise computation model does not permit these operations after the optional part. Thus, application designers had to either seek support from the operating system or integrate them into the mandatory part of the next cycle. The first approach, however, leads to poor portability, while the second one cannot be used for aperiodic requests.

4. Scheduling Method

This section describes a novel scheduling algorithm called Slack Stealer for Optional Parts (SS-OP) and its implementation on the RT-Frontier operating system. The SS-OP algorithm is developed for on-line preemptive scheduling on a uniprocessor and schedules independent imprecise computation in a deadline driven manner.

The objective of SS-OP is to provide support for applications with optional parts whose maximum execution times are not constant. Its primary focus is on keeping its runtime overhead small enough so that the system performance is not unnecessarily lowered when the system is not overloaded. We note that it is not the focus of SS-OP to theoretically minimize the error of computation, because the error generated upon terminating an optional part cannot be estimated properly when the length of the time required for its completion is not known.

There are two rules for maintaining the overhead of SS-OP to a practicable level. The first rule is to keep the computational complexity at the same level of that of EDF for every common scheduling event. The second rule is to keep the overhead of SS-OP to $O(1)$ for every new scheduling event which is not present in EDF. These rules allow the overhead of the SS-OP scheduler to be accounted for, in a way that is similar to the EDF scheduler. This property adds a significant advantage to SS-OP, because EDF, being one of the simplest yet most effective algorithms, has been used in many systems and analyzed thoroughly. Consequently, the hurdle for adopting SS-OP becomes lower than otherwise.

The performance of SS-OP, on the other hand, is improved by reclaiming unused processor time dynamically for optional parts. As was stated, the performance of imprecise computation is usually measured in terms of the error, which only decreases by executing a larger portion of the optional part. The resource reclaiming has good potential of improving the performance, because the execution times of the mandatory and wind-up parts must be reserved for their worst cases which rarely occur.

4.1. Assumptions and Terminology

The SS-OP algorithm allows workload that consists of hard periodic tasks and soft aperiodic tasks. The periodic tasks are assumed to have the relative deadline that is the same as their period, while the aperiodic tasks only request a short response time. Since aperiodic tasks have no deadline, it is assumed that they do not have any optional part ei-

ther. It is also assumed that periodic tasks are not activated dynamically.

Each instance of the tasks, or the job, J_i is characterized by four parameters: a release time (r_i), a deadline (d_i), a WCET of the mandatory part (m_i), and a WCET of the wind-up part (w_i). Since precise jobs do not have optional parts, w_i is zero for them. It is assumed that no mandatory nor wind-up part overruns. This should be a fair assumption, because the ability of SS-OP to perform dynamic resource reclaiming would encourage programmers to set the WCET with a wider margin. The WCET of the optional part and the error function is not required in SS-OP, because these are hard to determine in many applications.

The utilization of a task which J_i belongs to is defined as follows. If the task is periodic,

$$u_i = \frac{m_i + w_i}{T_i}, \tag{1}$$

where T_i is the period of the task. And otherwise u_i is zero. Accordingly, the utilization of the system with n tasks can be defined as $U_e = \sum_{i=1}^{n} u_i$. We call this U_e the essential utilization, since it represents the processor bandwidth that is essential for meeting all timing constraints. We also use U_o to denote the optional utilization, which is the processor bandwidth that can be spared to execute optional parts and aperiodic jobs. Thus, $U_o = 1 - U_e$. We assume that U_e is less than one, because otherwise there is no point in adopting the imprecise computation technique.

In addition, we define the following variables.

- Γ_s: the group of periodic jobs that are executing their optional part and all aperiodic jobs
- J_E: the job with the earliest deadline in Γ_s
- $J_{p(i)}$: the job whose deadline is the latest among all jobs J_k such that $d_k \leq d_i$
- $J_{n(i)}$: the job whose deadline is the earliest among all jobs J_k such that $d_k \geq d_i$
- t_c: the current time of the system
- R_i: the amount of remaining execution time which job J_i is allowed to spend before any scheduling event occurs
- S_i: the amount of slack which job J_i is allocated

Finally for the sake of description, we use E_i to represent R_i if the job J_i has finished its mandatory part or J_i is aperiodic, and to represent S_i otherwise.

4.2. SS-OP Algorithm

As the name implies, the SS-OP scheduler steals an amount of slack from the mandatory and wind-up parts to execute the optional parts. The mandatory first approach cannot be used in RT-Frontier, since we need to schedule the wind-up part after the optional part. Prior to all the description, we show the entire algorithm in Figure 3.

Unless one of the following events occurs, execute the job J_i with the earliest deadline and decrease R_i accordingly.

i. When an aperiodic job J_i has become ready, set $R_i = m_i$, $S_i = 0$, and then set d_i according to Equation (3).
ii. When a periodic job J_i has become ready, set $R_i = m_i$ and S_i according to Equation (2), and if $J_{n(i)}$ exists, then set $E_{n(i)} = E_{n(i)} - S_i$.
iii. When an imprecise job J_i has completed its mandatory part, set $R_i = R_i + S_i$ and $S_i = 0$.
iv. When an imprecise job J_i has completed its optional part or when R_i has become zero, set $R_i = R_i + w_i$.
v. When an aperiodic job J_i has consumed all the allocated time as a consequence of being stolen e unit of time, set $R_i = e$ and then set d_i according to Equation (3).
vi. When a job J_i is completed and if the job $J_{n(i)}$ exists, then set $E_{n(i)} = E_{n(i)} + R_i$.

Figure 3. SS-OP algorithm

The SS-OP algorithm executes jobs in the EDF manner and has six scheduling events associated with operations for managing the amount of the slack. In other words, the major difference between SS-OP and EDF is in the management of the slack.

Under SS-OP, all jobs in the system receive an amount of slack, in addition to the execution time reserved using the essential utilization. A periodic job only consumes this slack in its optional part, whereas an aperiodic job must consume it at any instant. The slack allocated to a precise periodic job is never consumed, but it can be stolen and consumed by other jobs.

In the following, we first describe the term slack interval and explain how slack is allotted to jobs. Then, we describe how SS-OP calculates the amount of slack for periodic tasks and how it assigns a deadline to aperiodic tasks.

4.2.1. Slack interval. A slack interval is what shows the amount of slack available to its owner. The rationale behind the slack interval is derived from the notion of the processor bandwidth. The notion of processor bandwidth allows the amount of slack in any interval $[t_1, t_2)$ to be calculated as $U_o(t_2 - t_1)$. We call this interval the slack interval. Note that the length of a slack interval and the amount of slack available in it are equivalent, because they can be converted to each other using the optional utilization U_o.

We regard that a slack interval $[t_1, t_2)$ is first created when a job succeeds in stealing an amount of slack from the system. As the job consumes x unit of this allocated slack, the corresponding slack interval shortens accordingly to $[t_1 + x/U_o, t_2)$. And finally, the slack interval is regarded

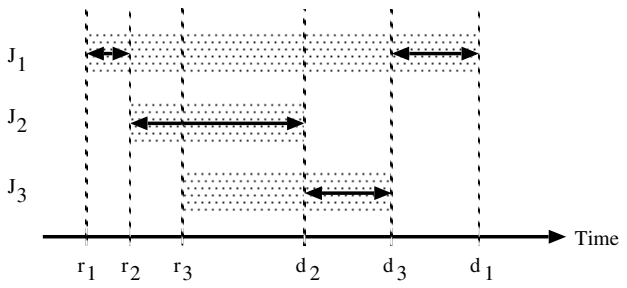

Figure 4. Allotment of slack intervals under SS-OP

to vanish from the system at the time when its length has become zero.

Figure 4 illustrates how SS-OP allots slack intervals to jobs. The largest possible interval from which a job J_i can claim slack is clearly confined to $[r_i, d_i)$. When these intervals overlap, that is when more than one job contends for the slack in the same interval, the overlapping interval is assigned to the job with the earliest deadline. In other words, the slack is allocated to jobs in the same order as they are executed. This simplifies dynamic slack management greatly, because the scheduler does not have to maintain any extra data structure, such as a list, for merely managing the amount of slack. And this means that SS-OP and EDF has the same order of complexity for managing ready jobs.

We continue to use the slack interval in the following description, since it is useful in expressing the slack allotment scheme. However, the scheduler implemented in RT-Frontier does not use the slack interval directly as described above, because it is impracticable to keep track of all the slack intervals at run-time. Consider the case where, at time r_2 of Figure 4, J_1 has not consumed all the slack in the interval $[r_1, r_2)$. Since $d_2 < d_1$, the slack interval requested by J_2 must be deallocated from J_1. If the scheduler tries to keep track of all the beginning and end of the intervals, it needs to maintain two separate intervals for J_1, one before r_2 and another after d_2. And in the worst case, the scheduler needs to consider $O(n)$ separate intervals after n times of preemption.

In order to avoid this undesirable situation, the SS-OP scheduler calculates the maximum amount of slack at the same time as the arrival of jobs, although the slack is not actually required until the mandatory part is completed. This early calculation is crucially important, because an amount of slack can then be handed on from one job to another through simple addition and subtraction, even when the jobs involved in this transfer of slack have not finished their mandatory part. As a result, the scheduler does not need to maintain discrete slack intervals any more.

4.2.2. Slack stolen by periodic jobs. The SS-OP scheduler calculates the amount of slack S_i for a newly arrived periodic job J_i as follows. Let t_E be the beginning of the earliest slack interval that still exists in the system. Then,

$$S_i = \begin{cases} 0 & \text{if } d_i \le t_E \\ U_o(d_i - \max\{d_{p(i)}, t_E, r_i\}) & \text{otherwise.} \end{cases} \quad (2)$$

The first condition corresponds to the case where all the slack that J_i tries to steal has already been consumed by other jobs. In the second case, there is still slack left for J_i. As was stated, the slack interval must be always allocated to the job with the earliest deadline on contention. Thus, if the job $J_{p(i)}$ exists, J_i cannot not steal slack from the slack interval before $d_{p(i)}$. Conversely, if the job $J_{n(i)}$ exists, a part of the slack interval before d_i must be deallocated from $J_{n(i)}$ and given to J_i. And otherwise, if there is no job with the deadline later than d_i, a new slack interval is safely created for J_i without affecting the slack intervals already allocated to other jobs.

It must be noted that t_E can be later than $d_{p(i)}$. For example, consider the case shown in Figure 4 again. Suppose that all jobs are periodic and that $r_1 = 0$, $r_2 = 3$, $r_3 = 4$, $d_1 = 10$, $d_2 = 8$, $d_3 = 9$, $m_1 = 1$, $m_2 = m_3 = 2$, $w_1 = 0.5$, and $w_2 = w_3 = 0$. Then at time 0, the scheduler calculates S_1 as 0.5. Since J_1 exhausts this slack at time 1.5, t_E already reaches 10 at this moment. Thus, at the time the job J_3 arrived, t_E is later than d_2.

4.2.3. Deadline assigned to aperiodic jobs. An aperiodic job is similar to an optional part from the viewpoint of scheduling, since it needs to steal slack to execute any of its portion. However, they are actually the opposite, since the aperiodic job does not have a deadline but a fixed execution time. Thus, it is an appropriate deadline that the scheduler must assign to a newly arrived aperiodic job. The deadline of an aperiodic job should be as early as possible to make its response time short, but it should be adequately late for the job to steal enough slack without violating the timing constraints of other jobs.

Upon an arrival of an aperiodic job J_i, the SS-OP scheduler calculates its deadline d_i as

$$d_i = \max\{t_c, d_L\} + \frac{e}{U_o}, \quad (3)$$

where d_L is the latest deadline among ready jobs and e is the maximum amount of slack which J_i requires for its completion. Calculating the deadline as above is virtually equivalent to assigning the job a slack interval which begins just after the latest interval, which leaves the slack intervals of all the other jobs unaffected.

The value of e in Equation (3) equals to m_i when a job J_i has become ready. On the other hand, there is also a case where the value of e does not equal to m_i. Consider the case where an imprecise job has become ready after a deadline

was assigned to an aperiodic job and where the deadline of the imprecise job is earlier than that of the aperiodic job. In this case, for the better performance of the system, SS-OP makes the imprecise job steal slack from the aperiodic job. Since no aperiodic job is given an extra amount of time, this aperiodic job will overrun. Thus, the scheduler will have to assign a later deadline to this job on its overrun. And this time, e must equal the amount of time that was stolen by the imprecise job.

4.3. Schedulability Analysis

From the theoretical results on the assessment problem of task set feasibility under EDF, it is known that the system utilization must be no more than one to have a feasible schedule. Likewise, SS-OP produces a correct schedule if $U_e \leq 1$, because all optional parts are discarded in the worst case, allowing us to regard the imprecise computation as precise. Therefore, in order to show the correctness of SS-OP, it is sufficient to prove that the sum of the processor bandwidth allocated to the optional parts and the aperiodic jobs does not exceed U_o under the condition $U_e < 1$. This is shown in the following lemma.

Lemma 1 *In any interval of time $[t_1, t_2)$, the processor demand $h_{[t_1,t_2)}$ of all optional parts and all aperiodic jobs is less than or equal to $U_o(t_2 - t_1)$.*

Proof. Without loss of generality, we assume that there exist n jobs $J_1, J_2, \cdots, J_n$ whose release times are later than or equal to t_1 and whose deadlines are less than or equal to t_2. We also assume that if $j < k$ then $d_j < d_k$.

The lemma holds clearly if n is equal to 1, since the maximum amount of slack time given to J_1 is $U_o(d_1 - r_1)$. If we suppose that the lemma holds for the interval $[t_1, t_2)$, it is sufficient to prove that the lemma still holds after a new job J_i is released.

If J_i belongs to a periodic task, we have three cases to consider with respect to its deadline d_i.

First, if $d_i < d_1$, then slack $U_o(d_i - \max\{t_E, r_i\})$ is transferred from J_1 to J_i. By definition of t_E, the amount of slack which J_1 had held just before the arrival of J_i must have been $U_o(d_1 - \max\{t_E, r_1\})$. Since this is larger than the requested amount, the sum of the processor demand in $[t_1, t_2)$ remains the same.

Second, if there exists k such that $d_k < d_i < d_{k+1}$, then slack $U_o(d_i - \max\{d_k, t_E, r_i\})$ is stolen from J_{k+1} to J_i. Since the amount of slack which J_{k+1} had held just before the arrival of J_i must have been $U_o(d_{k+1} - \max\{d_k, t_E, r_{k+1}\})$, the processor demand in $[t_1, t_2)$ does not change.

And third, if $d_n < d_i < t_2$ holds, then slack $U_o(d_i - \max\{d_n, t_E, r_i\})$ is stolen from a new slack interval that begins at $\max\{d_n, t_E, r_i\}$. Since d_n was the latest deadline in Γ_s before J_i has arrived, the processor demand in $[t_1, t_2)$ must have been the same in $[t_1, d_n)$. Therefore, the processor demand in $[t_1, t_2)$ becomes

$$\begin{aligned} h_{[t_1,t_2)} &= h_{[t_1,d_n)} + S_i \\ &\leq U_o(d_n - r_1) + S_i \\ &\leq U_o(d_i - r_1) \\ &\leq U_o(t_2 - t_1). \end{aligned}$$

Thus, the lemma holds for the periodic job arrival.

If J_i belongs to an aperiodic task and the amount of slack requested by J_i is e, this job is assigned a deadline according to Equation (3). By transforming the Equation (3), we have $e = U_o(d_i - \max\{t_c, t_E, d_n\})$. Thus, assuming the release time of J_i as t_c and substituting S_i with e, the arguments for the third case of the aperiodic job arrival applies to this case. Thus, the lemma holds for the aperiodic job arrival.

Hence, the lemma holds. □

4.4. Implementation of SS-OP

The RT-Frontier operating system implements the SS-OP algorithm through the coordination of three different components, namely the scheduler, a hardware timer, and jobs. It is the scheduler that controls the other two components for the most of the time. The role of the scheduler is to manage the maximum allowable time and slack for the jobs. The timer is used to measure the length of execution and to notify the scheduler when the running job has run out of its allocated time. The jobs, unlike other scheduling algorithms, also take an important role to detect scheduling events.

4.4.1. Detection of scheduling events. First of all, the scheduling events that are unique to SS-OP are the completion of the mandatory parts and that of the optional parts. These two events are rather annoying, since they cannot be detected by the scheduler.

The RT-Frontier operating system seeks help from the jobs here. A job knows when its mandatory or optional part is completed. Thus, a job can tell the scheduler when these events occur, through software traps that can be generated using one of the trap instructions. Since these software traps must be issued at proper timings, they are embedded in a library routine for spawning an imprecise thread.

Figure 5 shows the inner body of the library in pseudo code. The pointers to the functions that represent the mandatory, optional, and wind-up part are given as the arguments to the routine. The first trap for signaling the completion of mandatory part must always be issued safely within the reserved time m_i for the job to determine whether the optional part must be discarded. On the other hand, the second trap needs to be issued only after the completion of the optional part. Thus, if the optional

```
mfunc();        /* mandatory part */
res = end_mandatory(); /* 1st trap */
if (res != DISCARD) {
    ofunc();    /* optional part*/
    end_optional();    /* 2nd trap */
}
wfunc();        /* wind-up part */
```

Figure 5. Pseudo code for imprecise computation

part is discarded or terminated, the second trap is not issued at all.

Although we hope to eradicate any misuse of these traps with the help of this library, it may also happen that a malicious programmer tries to break down the system using the same trap instruction. The remedy is that SS-OP can ensure that no timing violation occurs to the rest of the jobs. If a job tries to cause an overrun in its mandatory or wind-up part, the SS-OP scheduler can immediately detect that the job is insane, since the WCET of these parts is given. Thus, the only chance where a job can illegally consume processor time is when the second trap is issued too late or not at all. However, due to the fourth event in Figure 3, any optional part is terminated as soon as its allocated time is exhausted. Hence, no job can execute illegally long to cause a timing fault to any other job.

Secondly, an overrun of a job, which is in the fourth and the fifth event, is detected using a hardware timer. Specifically, the following steps are taken by the scheduler. Before making a job J_i running, the scheduler sets the timer to the maximum allowable time R_i. If J_i is completed or preempted before the timer expires, then the scheduler updates the remaining time R_i by reading the counter of the timer. Otherwise, the scheduler is invoked by an interrupt generated by the timer, in which the variable R_i is set to zero. Note that in this manner, any operation in Figure 3 requires the value of R_i to be updated beforehand.

Finally, the rest of the events are easy to detect. Since it is the scheduler that makes jobs ready, the detection of the first and the second event is trivial. Similarly, the normal completion of a job, which is the sixth event, can be detected without any modification from an EDF scheduler.

4.4.2. Detection of earliest slack interval. In the previous arguments, we have ignored how the scheduler can find the earliest interval to steal slack. In other words, the procedure for obtaining the value of t_E in Equation (2) has not been clarified.

Obviously, t_E is zero in the beginning. The scheduler updates this value only when a context switch occurs that involves a job in Γ_s, because this value is unchanged or not dominant in any other cases. In particular, it is sufficient to consider the cases where J_E is preempted or completed, because no other job has been able to consume any amount of slack since J_E was released. In either case, the scheduler updates

$$t_E = \max\{d_E, t_E\} - \frac{R_E}{U_o}. \quad (4)$$

This equation moves t_E backward from the latest time which t_E would have reached if J_E had consumed all the allocated slack. This is easier to implement than moving t_E forward by the amount of slack consumed, because a hardware timer, necessary for measuring the length of execution, usually decreases the counter.

The maximum operator in Equation (4) is crucially important, as we aim to improve the performance by reclaiming any unused time. In order to reclaim all unused time in the mandatory part of a job for its optional part, the scheduler simply gives the job the sum of the unused time R_i and the slack S_i in the third event of Figure 3. It is safe to reclaim this unused time, because SS-OP, like EDF, does not make the processor idle, whenever a ready job exists. Since this reclaimed time can be regarded as newly generated slack, the job J_i can also enter Γ_s when $S_i = 0$. As a consequence, t_E can be later than the deadline of J_E, which is d_E, in Equation (4).

4.4.3. Termination of optional parts. The extended imprecise computation model proposed in this paper requires a scheme to execute its wind-up part whenever its optional part is terminated before completion. The crucial point is that the SS-OP algorithm requires the wind-up part to be executed in a preemptive manner.

The RT-Frontier operating system allows the wind-up part to be executed by the same job using a mechanism similar to the signal handler. One of the prerequisites is the stack context of the job in the beginning of the mandatory part, which can be easily saved, for example, using the 'setjmp' defined in POSIX. Another prerequisite is the registration of a function that corresponds to the wind-up part.

When an optional part needs to be terminated, the kernel constructs another stack on top of the old one, which the terminated job has been using to execute its optional part, and make the job switch to this new stack. And at the same time, the kernel overrides the program counter of this job to the registered address of its wind-up part and also modifies the return address of the wind-up part to generate a software trap. Consequently, when this job resumes, it executes the function that corresponds to the wind-up part using the new stack and generates a trap on its return. This trap allows the kernel to override the context of this job again with the context that was saved in the beginning of the mandatory part.

4.4.4. Overhead reduction for processors without FPU. The SS-OP algorithm requires division and multiplication by the optional utilization U_o. In a naive way of implementation, the utilization of the system would be maintained in a floating point variable that is in the range of $[0, 1]$. However, some embedded processors do not have any floating point unit (FPU) and emulating an FPU with a software library can be very expensive.

In the RT-Frontier operating system, any variable that may be in the range of $[0, 1]$ are shifted and maintained as integer variables, unless there is danger of causing an overflow. The idea behind this conversion is to emulate floating point calculations by fixed point calculations. More specifically, the scheduler maintains the essential utilization and the optional utilization in two integer variables that hold the value 1024 times larger than the actual value. In other words, these utilizations are maintained in the range of $[0, 1024]$ by calculating the utilization of a job J_i as $\lceil (m_i + w_i) \times 1024/T_i \rceil$. As a result, floating point division by U_o is replaced by integer division by $U_o/1024$, which certainly decreases the overhead on processors without an FPU.

5. Performance Estimation

This section estimates the performance of the SS-OP algorithm implemented on the RT-Frontier operating system, in terms of the cost, the error of imprecise computation, and the response time of aperiodic jobs. The RT-Frontier operating system currently runs on a Responsive Processor described in [20], whose processing unit is a SPARClite which can achieve 121 relative MIPS. Using one of the hardware timers, the system tick in RT-Frontier is managed in the unit of $1ms$.

In the compilation of all programs, we used version 2.95.3 of GNU gcc compiler, patched to generate ELF binaries. The optimization level of the compiler was set to the second level with the option of '-O2'.

5.1. Cost of SS-OP

The cost of SS-OP is estimated from two different aspects. One aspect is the cost for the implementation and the other is that for the computation. The EDF scheduler is used as the baseline, because the purpose of this estimation is to confirm that the cost of SS-OP is at a comparable level of that of EDF.

The implementation cost can be estimated from the size of the memory required by the scheduler. Since it is difficult to grasp the precise amount of the memory solely required by the scheduler, we measured the total size of the whole kernel and the supporting library. From the result shown in Table 1, the memory requirement of SS-OP is only 2008 bytes larger than that of EDF. This could be satisfactory considering that the size of the RT-Frontier kernel roughly ranges from 20 kilobytes to 50 kilobytes, depending on its configuration. Moreover, the impact on the size of the RAM is very small, since the most of the increase comes from the text section.

Algorithm	text	data	bss	total
EDF	25304	312	824	26440
SS-OP	27296	328	824	28448

Table 1. Size of RT-Frontier (byte)

Calculated value	Overhead (μs)
Slack of periodic job (S_i)	20.00
Deadline of aperiodic job (d_i)	11.50

Table 2. Overhead of run-time calculation

On the other hand, the computational complexity can be estimated by the run-time overhead additionally incurred in SS-OP. Comparing the SS-OP scheduler with the EDF scheduler, the difference is merely in the way slack is managed, because jobs are executed in the same order under both algorithms. Moreover, the overhead for managing slack is in $O(1)$, since Equation (2) and Equation (3) as well as all the operations shown in Figure 3 do not depend on the number of jobs is the system. Thus, the run-time overhead of SS-OP is theoretically guaranteed to be at a comparable level of that of EDF.

In order to quantitatively estimate the overhead additionally incurred in the slack management, we have measured the overhead of calculating Equation (2) and Equation (3) after flushing all the caches.

The result obtained using a hardware timer whose resolution is $0.05\mu s$ is shown in Table 2. Each value within this table is the average of 100 measurements. The overhead for calculating the slack of periodic jobs is larger than the other, because the scheduler must perform two operations beforehand. One is the calculation of the remaining execution time R_i for the currently running job J_i. And the other is the calculation of the beginning of the earliest interval t_E. Subtracting these overheads, the pure overhead of calculating the slack for periodic jobs can be estimated as $7.70\mu s$.

These rather large overheads are due to the fact that the SPARClite only implements an integer divide step instruction, instead of an integer divide instruction. In particular, the scheduler needs to issue 32 integer divide step instructions to obtain a quotient or a remainder from 32-bit integer variables.

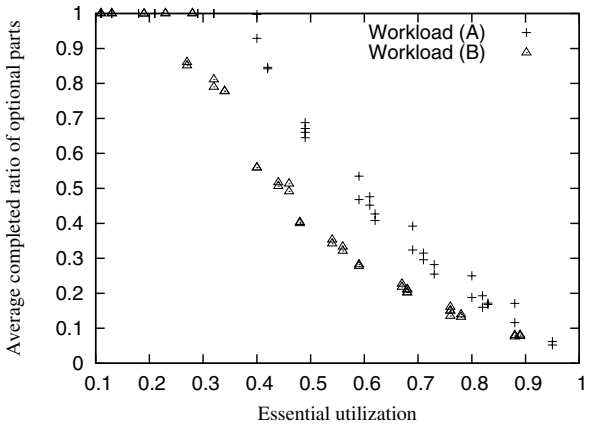

Figure 6. Average completed ratio

5.2. Error of Imprecise Computation

An experiment was performed to confirm that the imprecise computation is properly managed by the SS-OP algorithm. We used two types of synthetic workload, Workload (A) and Workload (B), which consist of periodic imprecise tasks. Jobs that belong to the same task has a fixed WCET of the mandatory part and a fixed WCET of the wind-up part. Their optional parts, however, require different execution times.

All the parameters that characterize the jobs are randomly selected from discrete uniform distributions. In the Workload (A), the periods, the WCET of mandatory parts, and the WCET of wind-up parts are respectively chosen from $[20, 60]$, $[30, 50]$, and $[1, 10]$. Using thus chosen WCET of the mandatory part, the execution time of optional parts is chosen from the distribution of $[2m_i - 25, 2m_i + 25]$. The Workload (B) is only different from the Workload (A) in that the WCET of the mandatory parts and the execution time of the optional parts are chosen from $[T_i - 10, T_i + 10]$ and $[3m_i - 25, 3m_i + 25]$, respectively. For both types of workload, the unit of the periods is $1ms$, while the unit of all the other ranges is $100\mu s$. These ranges are not set to express the finest timing constraints the RT-Frontier operating system can handle, because there is usually only little notion of quality in such processes. For instance, there is scarcely a notion of quality in the way a value is read from a sensor or an actuator is stopped.

The experiment was performed ten times for every task set. Each run started in phase and lasted $5s$. The result is shown in Figure 6. Every one of the plotted data represents the average of the completed ratio of the optional parts. We did not convert these ratios into the error, since an error function can not characterize the relationship properly for tasks whose optional part does not have any fixed maximum execution time. As the figure shows, the completed ratio got lower as the task sets with the higher utilization were tested. This result confirms that SS-OP succeeds in managing imprecise computation under various essential utilizations, without requiring any information on the optional parts.

Total Utilization	Essential Utilization	Response time (ms) Ave.	Max.	Min.
0	0	30.0	30.0	30.0
0.30	0.11	40.8	47.8	30.0
0.81	0.29	134.8	175.9	88.6
1.36	0.49	750.2	1492.7	203.5

Table 3. Response time of aperiodic jobs

5.3. Response Time of Aperiodic Tasks

The last experiment was performed to estimate the response time of aperiodic jobs. In this experiment, aperiodic jobs with the execution time of $30ms$ are dynamically added to the system while periodic tasks accounted for different utilizations.

The result from ten times of measurement with a hardware timer whose resolution is $0.1ms$ is shown in Table 3. The total utilization in the table refers to the sum of the processor demand required by the mandatory parts, that by wind-up parts, and the average processor demand required by the optional parts. As the result shows, the average response time of an aperiodic job is longer when the total utilization is higher. This result is not pessimistic, because we were able to confirm that aperiodic jobs with the logically largest deadline, which simulate non-real-time jobs, were not able to execute at all when the total utilization was 1.36. Since it is probable that the sum of the requested processor bandwidth occasionally goes over one in systems where the imprecise computation model is used, it can be concluded that SS-OP successfully discriminated soft aperiodic jobs from non-real-time jobs.

6. Conclusions

The RT-Frontier operating system presents a new framework for constructing real-time systems based on the notion of imprecise computation. Under this framework, computation is logically split into three parts, namely the mandatory part, the optional part, and the wind-up part. This proposed model allows the imprecise computation to be terminated in a less demanding manner than the traditional model, owing to the newly added wind-up part, which greatly improves the applicability of the imprecise computation technique.

The presence of the wind-up part also works in favor of enhancing the portability of applications, because it obviates the need for application specific support in the underlying operating system. The SS-OP algorithm, developed to manage imprecise computation with a wind-up part, can be used with minimal effort in systems where EDF has been used, because the run-time overhead of the SS-OP scheduler is at a comparable level of that of the EDF scheduler.

Our approach can be considered as a powerful alternative to other approaches that try to adjust the resource reservation amount, possibly incurring high overhead every time the condition changes. The most significant advantage in our approach is that dynamic adjustment against different environment is possible without explicitly determining and adjusting the resource requirements. This is a crucial property in modern real-time embedded systems that contain a significant amount of uncertainty.

As for future work, we plan to further extend SS-OP to allow dynamic activation of periodic tasks and to handle shared resources between computations with the intention of supporting broader ranges of real-time applications.

Acknowledgment

This study was performed through Special Coordination Funds of the Ministry of Education, Culture, Sports, Science and Technology of the Japanese Government.

References

[1] M. Amirijoo, J. Hansson, and S. H. Son. Error-Driven QoS Management in Imprecise Real-Time Databases. In *Proceedings of the 15th Euromicro Conference on Real-Time Systems*, pages 63–72, July 2003.

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

[3] S. Baruah and M. Hickey. Competitive On-Line Scheduling of Imprecise Computations. *IEEE Trans. Comput.*, 47(9):1027–1032, Sept. 1998.

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

[5] 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*, pages 390–397, Sept. 1997.

[6] J.-Y. Chung, J. W. S. Liu, and K.-J. Lin. Scheduling Periodic Jobs That Allow Imprecise Results. *IEEE Trans. Comput.*, 39(9):1156–1174, Sept. 1990.

[7] 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*, pages 76–80, May 1993.

[8] 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*, pages 1347–1352, May 1999.

[9] 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*, pages 307–314, Dec. 2000.

[10] 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*, pages 96–101, May 1995.

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

[12] M. Humphrey and J. Stankovic. Predictable Threads for Dynamic, Hard Real-Time Environments. *IEEE Transactions on Parallel and Distributed Systems*, 10(3):281–295, Mar. 1999.

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

[14] J. W. S. Liu, K. J. Lin, W. K. Shih, A. C. Yu, J. Y. Chung, and W. Zhao. Algorithms for Scheduling Imprecise Computations. *IEEE Computer*, 24(5):58–68, 1991.

[15] J. W. S. Liu and W. K. Shih. Algorithms for Scheduling Imprecise Computations with Timing Constraints to Minimize Maximum Error. *IEEE Trans. Comput.*, 44(3):466–471, 1995.

[16] G. B. Parker. Punctuated Anytime Learning for Hexapod Gait Generation. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and System*, volume 3, pages 2264–2671, Oct. 2002.

[17] W.-K. Shih and J. W. S. Liu. On-Line Scheduling of Imprecise Computations to Minimize Error. *SIAM J. Comput.*, 25(5):1105–1121, 1996.

[18] W.-K. Shih, J. W. S. Liu, and J.-Y. Chung. Algorithms for Scheduling Imprecise Computations with Timing Constraints. *SIAM J. Comput.*, 20(3):537–552, June 1991.

[19] M. Spuri and G. Buttazzo. Scheduling Aperiodic Tasks in Dynamic Priority Systems. *The Journal of Real-Time Systems*, 10(2):179–210, Mar. 1996.

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

[21] 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*, pages 1402–1407, Aug. 1993.