Scheduling Real–Time, Periodic Jobs Using Imprecise Results

by

Jane W. S. Liu
Kwei–Jay Lin
Swaminathan Natarajan

1304 West Springfield Avenue
Department of Computer Science
University of Illinois
Urbana, Illinois 61801

(217) 333–0135

This work was partially supported by the NASA Contract NAG 1 613.

Scheduling Real-Time, Periodic Jobs Using Imprecise Results

ABSTRACT

A process is called a monotone process if the accuracy of its intermediate result is non-decreasing as more time is spent to obtain the result. The result produced by a monotone process upon its normal termination is the desired result; the error in this result is zero. External events such as timeouts or crashes may cause the process to terminate prematurely. If the intermediate result produced by the process upon its premature termination is saved and made available, the application may still find the result usable and, hence, acceptable; such a result is said to be an imprecise one. The error in an imprecise result is nonzero. This paper discusses the problem of scheduling periodic jobs to meet deadlines on a system that provides the necessary programming language primitives and run-time support for processes to return imprecise results. This problem differs from the traditional scheduling problems since the scheduler may choose to terminate a task before it is completed, causing it to produce an acceptable but imprecise result. Consequently, the amounts of processor time assigned to tasks in a valid schedule can be less than the amounts of time required to complete the tasks. A meaningful formulation of this problem taking into account the quality of the overall result is discussed. Three algorithms for scheduling jobs for which the effects of errors in results produced in different periods are not cumulative are described, and their relative merits are evaluated.

I. Introduction

A *real-time task* must be completed by its deadline. In a hard real-time system [1], a *timing fault* is said to occur when a *real-time process* delivers its result too late. Often, one is willing to accept approximate, poorer quality results that can be produced on time in order to avoid timing faults.

This paper describes an approach to avoid timing faults in hard real-time systems. To motivate this approach, we note that for many applications, the quality of the result produced by a process improves with the length of time invested to produce it. For example, this is true of most iterative processes [2–4]. The result produced by an iterative process can be viewed as consisting of two components: the returned value(s) and the tolerance (a measure of error) in the value. The tolerance usually decreases as more iterations are carried out. Ideally, the process executes until a result with a desirably small tolerance has been obtained. When time is limited, it may be necessary to terminate the process before it completes a sufficient number of iterations. The result produced by a prematurely terminated process may be not as precise as desired, but it may still be acceptable, and therefore, can be used by the application. Other examples of applications where the precision of the intermediate result improves as execution progresses include dynamic programming, statistical programming techniques, and multi-phase processes [5–10].

We call a process a *monotone process* if the accuracy of its intermediate result is non-decreasing as more time is spent to obtain the result. For the sake of discussion here, it is reasonable to assume that the result produced by a monotone process upon its normal termination is the desired result; this result is said to be a *precise* one. External events such as timeouts, interruptions, or crashes may cause the process to terminate prematurely. If the intermediate result produced by the process upon its premature termination is saved and made available, the application may still find the result usable and, hence, *acceptable*; such a result is said to be an *imprecise* one. **Concord** is a system that provides the necessary programming language primitives and system support for processes to return imprecise results [11,12]. Specifically, real-time processes on Concord are either iterative or multi-phased. Concord allows

the programmer to specify the intermediate result variables to be recorded and the times to record them so that the resultant process is monotone. It also allows the programmer to define a set of *error indicators*. The values of the error indicators are made available to the application process along with the intermediate results. By examining these error indicators, the application process can decide whether an intermediate result is acceptable when the desired precise result cannot be obtained in time.

The capability to return imprecise results can be used to ensure timeliness of system responses in hard real-time systems. In particular, deadline mechanisms based on the use of primary and alternate versions [13] can be easily implemented on Concord. An iterative (or multi-phase) process returning an intermediate result after each iteration (or phase) accomplishes similar objectives as a set of multi-version processes that have different execution times. In both cases, the intent is to provide timely, but possibly degraded, services by making results that are of poorer quality available when the desired results cannot be obtained in time. However, since only one process for all versions is executed on Concord rather than one process per version, the overhead of coordinating multiple processes is avoided. Furthermore, our approach allows dynamic scheduling of real-time processes to meet deadlines with processors fully utilized to make the results as precise as possible.

This paper is concerned with the problem of scheduling periodic jobs to meet deadlines on a system such as Concord. This problem differs from traditional scheduling problems [14–16] for the following two reasons: (1) The scheduler may choose to terminate a task before it is completed, causing it to produce an acceptable but imprecise result. Consequently, the amounts of processor time assigned to tasks in a valid schedule can be less than the amounts of time required to complete the tasks. (2) A meaningful formulation of this problem must take into account the qualities of the results. For these reasons, we seek to find algorithms that will give us schedules in which all tasks meet their deadlines while some cost function of the result errors is kept small.

The rest of the paper is organized as follows. Section II discusses the structure of the Concord prototype system. Similarities between Concord and well-known systems such as SIFT

[17] and Argus [18,19] are summarized. A formal model of imprecise computations is discussed in Section III to provide a theoretical basis. Section IV discusses the problem of scheduling periodic jobs to meet deadlines making use of imprecise results. Three heuristic scheduling algorithms are described in Section V, along with their performance. Section VI is a short summary.

II. The Concord System

The general process structure of Concord is based on a variation of the client–server model [20,21]. A server type is defined for each real–time service provided by the system. Each (instance of a) server (type) is split into a *callee* and a *supervisor*. A client (caller) invokes a service by sending an invocation request to the supervisor of the server providing the service. When the supervisor receives the invocation request, it initializes the callee and requests the system to schedule the callee's execution. The supervisor executes concurrently with and monitors the progress of the callee. In particular, the supervisor records intermediate results produced by the callee at appropriate instances of callee's execution. It also records the values of those variables specified by the programmer as error indicators. When the callee terminates normally, the result produced by it is passed to the client through the supervisor. If the callee crashes or if its execution is terminated by the scheduler, the supervisor passes to the client the latest recorded values of the result and error indicator variables. Based on the values of error indicators, the client can decide whether the intermediate result is acceptable.

There are two Concord programming language primitives: *impreturn* is used in the callee procedure to return imprecise results. *impresult* is used in the caller procedure to accept imprecise results. An imprecise result along with error indicators is recorded by the supervisor in the server whenever an *impreturn* statement is executed. The example shown in Figure 1 illustrates the use of this statement. The iterative procedure implements Newton's method for finding roots of an equation. The use of the *impreturn* statement allows the result variable *xguess* and error indicators *change* and *error* be easily identified at compile time. These variables are recorded each time the *impreturn* statement is executed. Iteration continues till the scheduler terminates the procedure or the tolerance is satisfied. The value of the result variable recorded

```
/* Error indicators are change and error. */
while (abs(xnew - xguess) > tolerance) do
{
        xguess = xnew;
        xnew = xguess - f(xguess)/fprime(xguess);
        change = xnew - xguess;
        error = f(xnew)/f(xguess) * (xnew-xguess);
        if (f(xnew)/f(xguess)*(xnew-xguess) < precision)
                impreturn(xguess, change, error);
}
return(xnew);
```

Figure 1. Example of the *impreturn* construct

at the end of an iteration is the result returned to the client if the procedure terminates before the completion of the next iteration.

The semantics of the *impresult* statement is similar to that of the exception handling mechanisms supported by many programming languages. The *impresult* construct introduces an imprecise result handler. A client requesting an imprecise service attaches an *impresult* construct to its call statement. When an imprecise result is received from the server, it is treated as an exception condition. The imprecise result handler is invoked to evaluate the result. When the imprecise result is acceptable, the client resumes its execution as if it had received a final result. The callee is terminated after the result is returned.

The prototype Concord system has two components: the language processor and the run-time system. The Concord language processor has a compiler that converts *impreturn* and *impresult* primitives into run-time system calls and creates a supervisor for each procedure that will return imprecise results. The language processor also has a loader which can distribute a concurrent program in a distributed environment. Currently, each program is divided into modules for distribution among processors by the programmer. The general Concord run-time support is a layered distributed system. The prototype Concord system makes use of the run-time system designed for the Resilient Procedure (RP) project [22]. The RP supports imprecision

arising from multi-version software [23] and hardware.

In this paper, we are concerned only with the portion of Concord that runs on the multiprocessor system Encore Multimax. Communication delays between clients and servers on this tightly-coupled system are negligibly small. The task of assigning processors to execute invoked procedures is carried out by a scheduler. When the execution of a procedure is preempted by the scheduler to be resumed later, no result is returned to the client. A result is returned only when the callee procedure terminates either normally or involuntarily. The latter occurs when the scheduler terminates the callee procedure.

We note that Concord servers as described above can be naturally implemented as objects in object-based systems [17,18,24,25]. The callee procedure in a server implements an operation of the object. The supervisor, being a process that executes in parallel with the execution of the invoked operation, is the handler of the server object. Since each object contains more than one concurrently executing process, the Concord server objects must include explicit constructs for handling concurrency. In this way, Concord servers resemble Emerald objects [25]. Some of the functions of supervisors in Concord servers are the same as the functions of guardians in the MIT Argus system [16,17].

Both SIFT [15] and Concord rely on the iterative nature of the tasks. On the SIFT system, each task is a sequence of iterations. Each iteration is executed on a number of processors. The result produced by each processor is recorded at the end of the iteration. The intermediate result of the iteration is obtained by majority voting on the results recorded by different processors. Thus, error masking is achieved. Recording results of individual iterations makes it possible for voting to be carried out only at the beginning of each iteration. However, the intermediate results are not intended to be used as substitutes of the final results on SIFT as in the case of Concord.

III. Model of Imprecise Computations

The Concord system is based on the notion of imprecise computations. This section defines rigorously the correctness of (and error in) computation results and describes a formal model of imprecise computations. Again, we are particularly concerned with processes that have the

monotone property: the precision of the result produced by the continued execution of a process with this property is monotonically non-decreasing.

More specifically, we define the *correctness* C of the result R produced by a process P to be the extent to which the execution of P has progressed to produce the result. When R is not acceptable, C has a value of 0. When the process executes to its normal termination, C is 1. Let τ be the time required for the process to terminate normally and t_m be the minimum time required to produced an acceptable result. When a monotone process has executed for t units of time, the correctness function C of its result is given by

$$C(t) = \begin{cases} 0 & \text{if } t < t_m \\ c(t) & \text{if } t_m < t \leq \tau \\ 1 & \text{if } t \geq \tau \end{cases}$$

where $c(t)$ is in the range (0,1) and $c(t_1) \leq c(t_2)$ for $t_m \leq t_1 \leq t_2 \leq \tau$.

To model computations that sometimes return imprecise results, we note that if the computation environment is perfect, an independent computation P can be viewed as a transformation function:

$$P: \mathbf{I} \times \mathbf{S} \rightarrow \mathbf{O} \times \mathbf{S}$$

where $\mathbf{S}$ is the set of states of a state machine M, $\mathbf{I}$ is set of input values, and $\mathbf{O}$ is the set of output values [26-29]. In other words, for an initial state S and input value I, the output value O and the new state S' are given by

$$(O, S') = P(I, S) \tag{1}$$

We call this model the *precise computation model.*

In practice, a program may produce any of several different acceptable results depending on the environment. We are particularly concerned with the case when time constraints may cause a procedure to terminate prematurely. Allowing different amounts of time for execution will produce different results given the same input and initial state. When these results can be regarded as approximations to the desired result, the original program does not define uniquely the computation that will be performed. Rather, it defines a set of possibilities, or *imprecise*

computations.

For our purposes, it suffices to model imprecise computations as follows: Let **W** be the set of states of the computation environment of P. We model an independent imprecise computation P by the following transition function:

$$P: \mathbf{I} \times \mathbf{S} \times \mathbf{W} \rightarrow \mathbf{O} \times \mathbf{S} \tag{2}$$

In other words, given an initial state S, input values I, and an environment state W, we have

$$(O, S') = P(I, S, W) \tag{3}$$

for some output value O and final state S'. The side effects that the computation P might have on its environment are not of interest to us here and are not modeled explicitly in Eqs. (2) and (3).

This model allows us to state the notion of imprecision more rigorously. For example, an independent, iterative (or multi-phase) computation P that terminates normally after n iterations (phases) can be decomposed into n subcomputations. Let P_i be the transformation function performed by the ith iteration (phase) which is a precise computation modeled by (1). If P is allowed to execute until normal termination, its output and final state are given by

$$(O, S') = P(I, S) = P_n P_{n-1} \cdots\cdots P_2 P_1 (I, S) \tag{4}$$

To model a corresponding imprecise iterative (multi-phase) computation, let W_m be an environment state that will allow the first m iterations (phases) to complete but not the $(m+1)$th iteration (phase) for some $1 \leq m \leq n$. The output value and the final state of the imprecise computation is

$$(O_m, S'_m) = P(I, S, W_m) = P_m P_{m-1} \cdots\cdots P_2 P_1 (I, S) \tag{5}$$

Typically, W_m is not known. Hence, the result (O_m, S'_m) is one of n results defined by (5). Thus, we can model the multiple possibilities for program executions.

This model also allows us to define monotone property so that it can be formally treated. We have used the approach in [26,28,29] to specify Concord servers. In this approach, each computation object is modeled as a state machine M. The state machine is specified by two

basic types of functions: V-functions and O-functions. Associated with each state variable in the state space of M is a primitive V-function. The state of M at a particular moment is specified by the values of all primitive V-functions. Each O-function performs an operation that changes the state of M. State transitions, called effects, are specified by assertions relating new values of primitive V-functions to their prior values. The set of transition operations of M is the set of O-functions defined for the object. The interface to M consists of a set of derived V-functions (each of which returns a value computed from the values of primitive V-functions) and all the O-functions of M. Using this approach, each Concord server is an object whose interface contains one O-function and one or more derived V-functions (one for each result and error indicators). In the interface of the callee, which is accessed only by the supervisor in the server, this O-function corresponds to a sequence of consecutively invoked sub-O-functions (as in Eq.(4)). The values of the result and error indicator variables (i.e., derived V-functions) are changed by the invocations of these sub-O-functions. The monotone property can be stated in terms of assertions relating the values of error indicators before and after the invocation of each sub-O-function. The supervisor will record the new result and error indicators only when these assertions are true. Thus, intermediate results with increasing correctness can be obtained even when a computation does not have the monotone property naturally.

IV. Scheduling Periodic Jobs to Meet Deadlines

We adopt here the periodic-job workload model commonly used in studies on scheduling hard real-time jobs [30-39]. (From this point on, the terminology commonly used in studies on scheduling disciplines are used.) In this model, we are given a set $\mathbf{J} = \{J_k\}$ of K jobs. Each *job* is a unit of computation that can be scheduled independently from other jobs. The job J_k consists of a periodic sequence of *tasks* $T_{k,j}$ for $j = 1, 2, 3, ...$ The execution time of any task $T_{k,j}$ in J_k, denoted by τ_k, is the amount of processor time required to complete the task, i.e. for the task to execute until its normal termination. Let $\alpha_{k,j}$ be its *ready time* before which its execution cannot begin. Its *deadline* is the time instant at or prior to which it must produce an acceptable result. In particular, the deadline of the task $T_{k,j}$ is the ready time of the task $T_{k,j+1}$. Let $p_k = \alpha_{k,j+1} - \alpha_{k,j}$ be the *repetition period* of the job J_k and $r_k = 1/p_k$ be its *repetition rate*. (We choose to use this term instead of the more commonly used term iteration rate to avoid possible

confusion due to overloading the word "iteration".) The job J_k is, therefore, specified by the 2-tuple (p_k, τ_k). Let p be the least common multiple of the periods p_k for $k = 1, 2, \cdots K$, and $r = 1/p$.

The problem of scheduling tasks to meet deadlines on a system such as Concord differs from traditional scheduling problems in the following way: A task may be terminated before it is completed, producing an imprecise result. Consequently, it is possible for the total processor time assigned to a task in a valid schedule, referred to as its *assigned time*, to be less than its execution time. Hereafter, by a *schedule* we mean one in which the assigned time of every task $T_{k,j}$ is at least equal to $t_m(k)$, the minimum units of time required to produce an acceptable result. We refer to $t_m(k)$ as the *minimal execution time* of the tasks in job J_k. A schedule in which the assigned time of every task is equal to its execution time is called a *precise schedule.* (We note that only precise schedules are valid schedules in the traditional sense.) If the assigned time σ of a task is equal to its execution time τ, the result produced by the task is of the desired quality; the error in the result is zero. If its assigned time is less than its execution time (corresponding to involuntary termination of the task), the error $\epsilon(\sigma)$ in its result is a non-increasing function of σ. We consider here only preemptive schedules. The result produced by any task is returned to the client only at the time when the task terminates, either normally when it attains τ units of processor time, or involuntarily when its deadline is reached.

The term *feasible schedule* of the job set **J** refers to a schedule in which every task meets its deadline. In a feasible schedule, the total processor time assigned to every task prior to its deadline is equal to or larger than its minimal execution time. The problem of finding a feasible schedule of periodic jobs without regard to errors in the results is a relatively easy one. — Given a job set **J**, we define a periodic job set $\mathbf{F} = \{F_k\}$ in which the job F_k is specified by $(p_k, t_m(k))$. In other words, F_k has the same repetition rate as J_k but consists of tasks with execution time $t_m(k)$. The problem of finding a feasible schedule of the job set **J** is the same as the problem of finding a precise feasible schedule of the job set **F**. Known results [30–39] can be applied directly here.

A more meaningful formulation of the scheduling problem on a system that supports imprecise computations must take into consideration the quality of the results produced by the tasks. For this reason, we want to find algorithms leading to feasible schedules that keep some functions of errors small.

To motivate our choices of performance measures, we note that for different types of jobs, errors in the results cause different undesirable effects. As an example, consider a job consisting of tasks that periodically receive, enhance, and transmit frames of video images. The effect of errors in imprecise results is a reduction in the quality of the transmitted video, which is tolerable for many applications. Hence, no timing fault occurs as long as sufficient processor time is assigned to every task before its deadline so that an enhanced frame of acceptable quality is transmitted on time. For applications such as this one, errors in results produced in different periods are not accumulative. We referred jobs of this type as *type N(on-accumulative) jobs.*

A reasonable performance measure for type N jobs is the average error of all results. Given a feasible schedule of the job set J, the average error is defined as follows: Let the error in the result produced by the task $T_{k,j}$ when its assigned time is σ_{kj} be denoted by $\epsilon_k(\sigma_{kj})$. Let N_k be the number of consecutive periods over which the average error in job J_k is computed. At any time s beyond the deadline of the (N_k+i-1)th task but before the deadline of the (N_k-i)th task in J_k for some i, the average error in job J_k is

$$E_k = \left[\sum_{j=i}^{i+N_k-1} \epsilon_k(\sigma_{kj}) + \epsilon_k(\sigma_{kc}) \right] / N_k$$

where σ_{kc} is the processor time attained by the current task at time s. Let $E_k(s-)$ be the contribution to the average error of J_k by errors in the results produced during the N_k-1 periods prior to s. This equation can be rewritten as

$$E_k = E_k(s-) + \epsilon_k(\sigma_{kc}) / N_k \tag{6}$$

We may choose to compute the average error over a time interval of duration p, the least common multiple of all periods. In this case, the average error of J_k is approximately equal to

$$E_k = rp_k \sum_{j=l}^{j=l+pr_k} \epsilon_k(\sigma_{kj}) \tag{7}$$

The average error over all jobs in **J** is

$$E = [\sum_{k=1}^{K} w_k E_k] / K \tag{8}$$

where w_k are constant weights and $\sum_{k=1}^{K} w_k = K$. (These weights reflect the relative importance of different jobs.) Given a set of type N jobs, we want to find feasible schedules that minimize the average error E among all feasible schedules of **J** .

As an example of a different type of applications, suppose that we have a periodic job in which each task processes the returned signals from an air traffic control radar and to generate the coordinates and the velocity of a tracked target for display purposes. When a task terminates prematurely, it produces coarse estimates of the target position and velocity in that period. Typically, it is essential that a precise result be obtained every now and then. For example, the position of the target must be accurately displayed every 30 seconds. Hence, if the results produced by the position computation tasks in several consecutive periods are imprecise, the task in the next period must complete normally and produce a precise result. Otherwise, a timing fault occurs. We referred to jobs of this type as *type A(ccumulative) jobs*. One formulation of the scheduling problem for type A jobs is as follows: Let the cost $\beta(\epsilon_k(\sigma_{kj}))$ be a monotone non-decreasing function of the error $\epsilon_k(\sigma_{kj})$. At any time s between the deadlines of $T_{k\,n}$ and $T_{k(n+1)}$, let the cumulative cost due to errors in job J_k be

$$\gamma_k = \sum_{j=l}^{n} \beta(\epsilon_k(\sigma_{kj})) \tag{9}$$

where $T_{k\,l}$ $(l<n)$ was the last task in J_k that terminated normally and produced zero error. A timing fault occurs if the assigned time of any task is less than $t_k(m)$ and if when the cumulative cost γ_k exceeds an upper limit, the deadline of current task $T_{k(n+1)}$ is reached before it attains τ_k units of processor time. The scheduling problem we want to consider is: given a set of jobs, find a schedule which guarantees no timing fault while keeping the average error small.

V. Heuristic Scheduling Algorithms for Type N Jobs

We consider here a class of heuristic algorithms based on the rate–monotone algorithms [30,31]. This section describes three algorithms for type N jobs.

Given a set **J** of type N jobs, we first use either the rate–monotonic next–fit algorithm or the rate–monotonic first–fit algorithm [31] to assign individual jobs to processors. According to these algorithms, jobs in **J** are sorted in the order of increasing period lengths and are assigned to processors on next–fit or first–fit basis. In deciding whether a job can be assigned to a processor, we use the minimal execution time of its tasks instead of the execution time to compute the utilization factors. More specifically, the *minimal utilization factor* of job J_k is $u_k = r_k\, t_m(k)$. Suppose that n jobs with a total minimal utilization factor u are already assigned to a processor. If an addition job with repetition rate r, execution time τ, and minimal execution time t is assigned on this processor, the total minimal utilization factor of the $n+1$ jobs is $u+rt$. This job is assigned to the processor only if

$$u+rt \leq (n+1)(2^{\frac{1}{n+1}}-1)$$

We note that these jobs are not precisely schedulable in general. However, they are schedulable if the assigned time of every task is equal to its minimal execution time [31].

We now assume that jobs are assigned to processors as described above and confine our attention of the problem of how to schedule periodic jobs on a single processor to meet all deadlines while keeping the average error of all jobs on the processor small. The mixed scheduling algorithms described below use the rate–monotone algorithm to ensure that all deadlines are met but use different strategies to improve result quality. They work as follows: Given a job set **J** of K jobs to be scheduled, we define two job sets, the *first set*, **F**, and the *last set*, **L**. For each job $J_k = (p_k, \tau_k)$ in **J**, there is a job $F_k = (p_k, t_m(k))$ in the first set **F**. In particular, the task $T_{k,j}(F)$ in the job F_k is the first portion of the task $T_{k,j}$ in J_k, and the execution time of $T_{k,j}(F)$ is $t_m(k)$ for all j. Similarly, for each job J_k in **J**, there is a job $L_k = (p_k, \tau_k - t_m(k))$ in the last set **L**. The task $T_{k,j}(L)$ in the job L_k is the last portion of the task $T_{k,j}$ in J_k, and the execution time of $T_{k,j}(L)$ is $\tau_k - t_m(k)$. Furthermore, the ready times and

deadlines of the tasks $T_{k,j}$, $T_{k,j}(F)$, and $T_{k,j}(L)$ are the same. Instead of the job set **J**, we schedule the sets **F** and **L** using the following preemptive, priority-scheduling strategy: All jobs in **F** are given higher priorities than jobs in **L** and are precisely scheduled according to the rate-monotone algorithm. Again, by being precisely scheduled, we mean that the assigned processor time of every task is equal to its execution time. Because

$$\sum_{k=1}^{K} r_k \, t_m(k) \leq K(2^{\frac{1}{K}} - 1)$$

this strategy ensures that all tasks in **F** complete before their deadlines independent of how tasks in **L** are scheduled. Since, for every schedule of **F** and **L** obtained in this manner, there is an equivalent schedule of jobs in **J** in which the assigned time of every task is at least equal to its minimal execution time, this strategy allows us to schedule jobs in **J** such that all deadlines are met. Moreover, only a fraction $K(2^{\frac{1}{K}} - 1)$ (e.g, 0.82 for K=1 and ln 2 for large K) of processor time is used to execute tasks in **F**. Whenever there is no task in **F** to be executed, the spared processor time can be used to execute tasks in **L**. Thus, quality of the results can be improved. The algorithms described below differ in how priorities are assigned to jobs in **L**.

V.1. The Least Utilization Algorithm

To see how priorities should be assigned to jobs in **L** so that the average error is kept small, let us first consider the special case of linear error functions given by

$$\epsilon_k(\sigma_{kj}) = 1 - \sigma_{kj}/r_k \tag{10}$$

From Eq.(7) and (8), the average error of the job set **J** over a time interval of length p is given by

$$E = 1 - \frac{r}{K} \sum_{k=1}^{K} \frac{w_k}{U_k} \sigma_k \tag{11}$$

where $\sigma_k = \sum_{j=l}^{l+pr_k} \sigma_{kj}$ is the total processor time assigned to all tasks in J_k over pr_k periods, and $U_k = r_k \, \tau_k$ is the *utilization factor* of J_k.

Equation (11) suggests the following algorithm, referred to as *least utilization algorithm.* It is preemptive and priority-driven. Given a job set **J**, we divide it up into a first set **F** and a last set **L** as defined above. All jobs in **F** are assigned higher priorities than the jobs in the **L**. Priorities of the individual jobs in **F** are assigned statically on rate monotone basis, i.e., jobs with higher repetition rates are assigned higher priorities. Priorities of jobs in **L** are assigned also statically but according to their values of the ratio w_k/U_k. The larger this ratio is, the higher the priority. (When the constant w_k for all jobs are the same, jobs with smaller utilization factors are assigned higher priorities, and hence the name of this algorithm.) Without loss of generality, let

$$\frac{w_1}{U_1} > \frac{w_2}{U_2} > \ldots\ldots > \frac{w_K}{U_K}$$

At any time when there is no task in **F** to be executed, the processor is asigned to execute the current task in L_1 until it is completed. If the processor is free after the current task in L_1 is completed, it is asigned to execute the current task in L_2, and so on. As shown above, the least utilization algorithm ensures that all deadlines are met. Furthermore, when the errors are linear functions of the assigned time as in Eq.(10), the average error E is minimized when jobs are scheduled according to the least utilization algorithm.

V.2. Other Heuristic Algorithms

In general, the underlying procedure may not converge linearly, and the error in the result produced by a task does not decrease linearly as its execution continues. When the errors are not linear functions of assigned times, the least utilization algorithm does not minimize the average error. We consider two additional algorithms that keep the average error small for general error functions. Again, they differ from the least utilization algorithm only in how priorities are assigned to jobs in the last set **L**.

According to the *least attained time algorithm*, priorities are assigned to jobs in **L** dynamically in the following manner: At any time when the processor is free from executing tasks in **F**, the highest priority is assigned to the job whose current task has the least attained processor time. In other words, tasks in the last set **L** are scheduled on the shortest-elapse-

time-first basis until their deadlines are reached. Many iterative procedures converges faster during earlier iterations than later iterations; the error decreases faster during earlier iterations and slower during later iterations. In this case, the least attained time algorithm yields smaller average error than the least utilization algorithm.

To see how the performance of least attained time algorithm depends on the behavior of the the error function ϵ_k, we examine the average error of J_k given by Eq.(6) at time s when the attained processor time of the current task of J_k is less than its execution time, i.e. $\sigma_{kc} < \tau_k$. If additional processor time of duration δ is assigned to this task, the average error of J_k is approximately given by

$$E_k = E_k(s-) + [\epsilon_k(\sigma_{kc}) + \epsilon_k'(\sigma_{kc})\delta + \frac{1}{2}\epsilon_k''(\sigma_{kc})\delta^2] / N_k$$

where $\epsilon_k'(x)$ and $\epsilon_k''(x)$ are the first and second derivatives of the function $\epsilon_k(x)$, respectively. The former is always negative (due to monotone property.) However, for a large class of error functions of practical interest, the second derivative of $\epsilon_k(\sigma_{kc})$ is positive; the error functions are convex. In this case, the optimality of the least attained time algorithm depends on the relative values of these derivatives and the ratio w_k / N_k. An algorithm that assigns higher priorities to jobs with higher values of

$$\frac{w_k}{N_k}[\epsilon_k'(\sigma_{kc})\delta + \frac{1}{2}\epsilon_k''(\sigma_{kc})\delta^2]$$

would achieve the smallest average error. However, this algorithm is not a practical one. The exact behavior of the error functions are typically not known making it impossible to compute their derivatives statically. Computing the derivatives on dynamically basis would introduce unacceptably high overhead.

When the behavior of the error function is not known, we may want to use an algorithm that attempts to keep the average error small by making good use of slack time between the time instant at which the task in the current period first attains $t_k(m)$ units of processor time to the beginning of the next period, the ready time of the next task. The lower bound on slack times derived in [35] provides the basis of the *best slack time algorithm.* It has been shown that after

the completion of the task $T_{k,j}(F)$, the slack time to the ready time of the task $T_{k,j+1}(F)$ is at least equal to the $0.207\,q_{k,j}$ where $q_{k,j}$ is the length of the last quantum of processor time assigned to the task $T_{k,j}(F)$. This lower bound on slack times of individual jobs allows us to bound the slack times s_k between the instant when the processor becomes free from executing tasks in **F** and the ready times of the next tasks in **F**. At the time when the processor is free to execute tasks in **L**, the best slack time algorithm assigns the highest priority to the job with the largest slack time. We are currently evaluating the performance of this algorithm.

VI. Summary

This paper discusses the problem of scheduling periodic jobs on a system that allows tasks to terminate prematurely and produce imprecise results. When a task terminates normally after it is completed, the error in the result produced by it is zero. When it terminates prematurely, the result produced by it is acceptable as long as the duration of its execution is equal to or longer than its minimal execution time. Hence, for all deadlines to be met, it is only necessary for the amounts of processor time assigned to all tasks prior to their deadlines be at least equal to their minimal execution time. A good scheduling algorithm should allow us to find feasible schedules in which all deadlines are met and errors in the results are kept small.

Depending on the different types of undesirable effects caused by errors, jobs are classified as type N or type A. For type N jobs, the effects caused errors in results produced in different periods are not cumulative. A reasonable performance measure is the average error over all jobs to be scheduled. Three algorithms that lead to feasible schedules with small average error for type N jobs are described. Their performance in terms of average error depend on how errors as functions of the assigned times vary. The least utilization algorithm minimizes the average error when errors for all jobs are linearly dependent on assigned times. The least attained time algorithm achieves less average error for the case when errors are convex functions of assigned times.

For type A jobs, the undesirable effects of errors produced in different periods are cumulative. One special case of practical interest is when the cost function $\beta(x)$ in Eq.(9) is equal to 1 for $x > 0$ and is equal to zero for $x = 0$. In this case, requiring the cumulative cost be under

an acceptable upper limit is the same as requiring that at least one task among M_k tasks in each job J_k be completed normally and produce zero error. Known algorithms such as the rate-monotone algorithm and earliest deadline algorithm can be used to schedule this type of job to meet deadlines. Criteria of schedulability for type A jobs will be presented in a future paper.

References

[1] Kligerman, E. and A. D. Stoyenko "Real-time Euclid: a language for reliable real-time systems," *IEEE Trans. on Software Eng.* vol. SE-12, No. 9, pp. 941-949, Sep. 1986.

[2] Basu, A.K., "On development of iterative programs from function specifications," *IEEE Trans. Software Eng.,* vol. SE-6, pp. 170-182, Mar. 1980.

[3] Pizer, S.M., *To Compute Numerically,* Little, Brown and Co., Boston, MA, 1983.

[4] Turski, W.M., "On programming by iterations," *IEEE Trans. Software Engineering,* vol. SE-10, pp. 175-178, Mar. 1984.

[5] Dolev, D., M. Klawe, and M. Rodeh, "An $O(n\log n)$ unidirectional distributed algorithm for extrema finding in a circle," *Journel of Algorithms,* vol. 3, pp. 245-260, 1982.

[6] Lampson, B.W., and H.E. Sturgis, "Crash recovery in a distributed storage system," unpublished paper, Comp. Sci. Lab., Xeros Palo Alto Research Center, Palo Alto, CA, 1976.

[7] Chandy, K. M., J. Misra, and L.M. Haas, "Distributed deadlock detection," *ACM Transactions on Computer Systems,* vol.1, No.2, pp. 144-156, May 1983.

[8] Bellman, R. and S. Dreyfus, *Applied Dynamic Programming,* Princeton University Press, Princeton, N.J., 1962.

[9] Shreider, Y., ed., *Method of Statistical Testing: Monte Carlo Method,* Elsevier Publishing Company, New York, 1964.

[10] Rabin, M.O., "Randomized Byzantine generals," in *Proc. 24th Symp. Foundations of Comput. Sci.,* Tucson, AZ, pp. 403-409, Nov. 1983.

[11] Lin, K. J., S. Natarajan, and J. W. S. Liu, "Concord: a system of imprecise computations," to appear in the *Proceedings of the 1987 IEEE Compsac*, Japan, October, 1987.

[12] Lin, K. J., S. Natarajan, and J. W. S. Liu, "Imprecise results: Utilizing partial computations in real-time systems," submitted.

[13] Liestman, A. L. and R. H. Campbell, "A fault-tolerant scheduling problem," *IEEE Transactions on Software Engineering*, vol. SE-12, No. 10., pp. 1089-1095, Oct. 1986.

[14] Coffman, E. G. Jr. and R. Graham *Scheduling Theory*, John Wiley and Sons, New York.

[15] Lenstra, J. K. and A. H. G. Rinnooy Kan, "Scheduling theory since 1981: an annotated bibliograph," Report No. BW 188/83, Mathematisch Centrum, Amsterdam, the Netherlands, 1983.

[16] Lenstra, J. K. and A. H. G. Rinnooy Kan, "New directions in scheduling theory," *Operations Res. Lett.*, vol. 2, pp. 255-259, 1984.

[17] Wensley, J. H., L. Lamport, J. Goldberg, M. W. Green, K. N. Levitt, P. M. Milliar-Smith, R. E. Shostak, and C. B. Weinstock, "SIFT: design and analysis of a fault-tolerant computer for aircraft control," *Proceedings of IEEE*, Vol. 66, No. 10, pp. 1240-1255, October, 1978.

[18] Liskov, B. and B. Scheifier, "Guardian: A methodology for robust, distributed programs," *ACM Trans. on Program. Lang. Syst.*, vol. 5, pp. 381-405, July 1983.

[19] Liskov, B., "Overview of the Argus language and system," Lab. of Computer Science, MIT, Programming Methodology Group Memo, 1984.

[20] Tannenbaum, A. S., *Computer Networks*, pp.464-465., Prentice-Hall, NJ., 1981.

[21] Gray, J. N., "An approach to decentralized computer systems," *IEEE Transaction on Software Engineering*, vol. SE-12, No. 6, pp. 684-692, Jun. 1986.

[22] Lin, K.-J., "Resilient procedures - an approach to highly available system," in *Proc. IEEE Computer Society International Conf. on Computer Lang.*, Miami, pp. 98-106, Oct. 1986.

[23] Chen, L. and Avizienis, A., "N-version programming: A fault-tolerance approach to reliability of software operation," in *Proc. 8th Symp. Fault-Tolerance Computing,* pp. 3-9, 1978.

[24] Almes, G. T., A. P. Black, E. D. Lazowska, "The Eden system: a technical review," *IEEE Trans. on Software Eng., vol. SE-11, pp. 43-58, Jan. 1985.*

[25] Black, A., N. Hutchinson, E. Jul, H. Levy, and L. Carter, "Distribution and abstract typies in Emerald," *IEEE Trans. on Software Eng.*, vol. SE-13, No.1, Jan. 1987.

[26] Parnas, D. L., "A technique for software module specification with examples," *Comm. ACM*, vol. 15, No. 5, May 1972.

[27] Cheheyl, M. H., Gasser, M., et. al., "Verifying security," *ACM Computing Surveys*, vol. 13, No. 3, Sep. 1981.

[28] Levitt, K. N., L. Robinson, and B. A. Silverberg, "The HDM handbook," Vols. 1-3, Computer Science Lab., SRI International, Menlo Park, California, June 1979.

[29] Silverberg, B. A., "An overview of the SRI hierarchical development methodology," Computer Science Lab, SRI International, Menlo Park, California, July 1980.

[30] Liu, C. L. and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *J. Assoc. Comput. Mach.*, vol. 20, pp. 46-61, 1973.

[31] Dhall, S. K. and C. L. Liu, "On a real-time scheduling problem," *Operations Res.*, vol. 26, pp. 127-140, 1978.

[32] Leung, J. Y.-T. and M. L. Merrill, "A note on preemptive scheduling of periodic, real-time tasks," *Inform. Process. Lett.*, vol. 11, pp. 115-118, 1980.

[33] Lawler, E. L. and C. U. Martel, "Scheduling periodically occurring tasks on multiple processors," *Inform. Process. Lett.*, vol. 12, pp. 9-12, 1981.

[34] Leung, J.Y.-T. and J. Whitehead, "On the complexity of fixed-priority scheduling of periodic, real-time tasks," *Performance Eval.*, vol. 2, pp. 237-250, 1982.

[35] Liu, C. L., J. W. S. Liu and A. L. Liestman, "Scheduling with slack times," *Acta Informatica*, vol. 17, pp. 31–41, 1982.

[36] Bertossi, A.A. and M.A. Bonuccelli, "Preemptive scheduling of periodic jobs in uniform multiprocessor systems," *Inform. Process. Lett.* vol. 16, pp. 3–6, 1983.

[37] Stankovic, J., K. Ramamritham, and S. Chang, "Evaluation of a flexible task scheduling algorithm for distributed hard real-time systems," *IEEE Transactions on Computers*, Vol. C–34, No. 12 pp. 1130–1144, Dec. 1985.

[38] Zhao, W. and K. Ramamritham, "Distributed scheduling using bidding and focused addressing," *Proceedings of IEEE Real-time Symposium,* December 1985.

[39] Leinbaugh, D. W. and M. Yamini, "Guaranteed response time in a distributed hard real-time environment," *Proceedings of Real-Time Systems Symposium*, December, 1982.