Published in: "Foundations of Dependable Computing:
Paradigms for Dependable Applications," Koob and Lau,
ed., Kluwer, 1994

SECTION 3.1

Use of Imprecise Computation to Enhance Dependability of Real-Time Systems

Jane W. S. Liu, Kwei-Jay Lin,
Riccardo Bettati, David Hull and Albert Yu

Abstract

In a system based on the imprecise-computation technique, each time-critical task is designed in such a way that it can produce a usable, approximate result in time whenever a failure or overload prevents it from producing the desired, precise result. This section describes ways to use this technique together with traditional fault-tolerance methods to reduce the costs of providing fault tolerance and enhanced availability. Specifically, an imprecise mechanism for the generation and use of approximate results can be integrated in a natural way with traditional checkpointing and replication mechanisms. Algorithms and process structures for this integration and rules for determining when approximate results can be used in place of the desired results are discussed.

3.1.1 Introduction

The imprecise computation technique was proposed as a way to handle transient overloads in real-time systems [1–3]. Here, by *real-time system*, we mean a computing and communication system in which a significant portion of the tasks have *deadlines*. The term *task* refers to a unit of work to be scheduled and executed. A task may be the computation of a control law, the transmission of an operator command, the

Jane W. S. Liu, Riccardo Bettati, and David Hull are at the Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois 61801.

Kwei-Jay Lin is at the Department of Electrical and Computer Engineering, University of California, Irvine, Irvine, California.

Albert Yu is at Hughes Aircraft, Radar Systems Group, Los Angeles, California 90009.

This work was partially supported by the U. S. Navy ONR contracts No. NVY N00014 89-J-1181 and No. NVY N00014 89-J-1146.

retrieval of a file, etc. As the result of its execution, each task delivers some data or service. The failure of a time-critical task to deliver its result by its deadline is a timing fault. A real-time system functions correctly only in the absence of timing faults. An example is the TCAS (Traffic Alert and Collision Avoidance System) used in commercial aircraft to alert pilots of potential collisions. The command telling the pilot of the conflict traffic and the necessary evasive action not only must be correct but also must be issued in time. Other examples of real-time systems include flight control and management, intelligent manufacturing, and various monitoring systems.

For many real-time applications, we may prefer to have approximate results of a poorer but acceptable quality on a timely basis to late results of the desired quality. For example, it is better for a collision avoidance system to issue a timely warning together with an estimated location of the conflict traffic than a late command specifying the exact evasive action. Other examples are video and voice transmissions. While poor quality images and voices may be tolerable, late frames and long silences are often not. The imprecise computation technique was motivated by this observation and the fact that good approximate results can often be produced with much less processor time than results of the desired quality. By trading the quality of the results for the amount of time and resource required to produce them, a system based on this technique tries to make approximate results of acceptable quality available whenever it cannot produce results of the desired quality in time. Hereafter, we will refer to such a system as an *imprecise system* and a result of the desired quality a *precise result*.

An imprecise computation is a generalization of an anytime computation, a term used in the AI literature. The focus of the AI community has been on anytime computational algorithms and reasoning about the results produced by them (e.g., [4–6]). On the other hand, the work by the real-time systems community has been concerned with how to structure the operating and run-time systems to take advantage of the scheduling flexibility provided by imprecise computations. In recent years, several models have been developed to characterize the behavior of imprecise computations and to quantify the costs and benefits of the tradeoff between result quality and required processing time for different classes of applications and implementation methods. Many efficient scheduling algorithms that achieve the optimal or suboptimal tradeoff are now available; examples are those described in [7–18]. These algorithms have made the imprecise computation approach to overload handling feasible.

The imprecise computation technique is also a natural means for enhancing fault tolerance and graceful degradation of real-time systems. We consider here only hardware faults and transient software faults. Permanent software faults are not considered. To see how availability and fault tolerance can be increased in an imprecise system, we consider tracking and control systems for example. A transient fault may cause a tracking computation to terminate prematurely and produce an approximate result. No recovery action is needed if the result allows the system to maintain track of the targets.

Similarly, as long as the approximate result produced by a control law computation is sufficiently accurate for the controlled system to remain stable, the fault that causes the computation to terminate prematurely can be tolerated. In embedded systems, this technique can be used together with the traditional replication and checkpointing techniques (e.g., [19,20]). The result is a reduction of the costs of providing fault tolerance and enhanced availability.

This section describes an architecture that integrates the *imprecision mechanism* for the storage and return of intermediate, approximate results of computations with the fault-tolerant mechanisms that support traditional checkpointing and replication. The Imprecise Computation Server (ICS) system is being built on the Mach operating system to implement this architecture. ICS will make it easy for us to implement imprecise computations and experiment with them. When it is completed, we plan to implement several representative applications using the ICS system in order to evaluate experimentally the effectiveness of the imprecise computation technique. The applications we plan to implement and experiment with include multimedia data transmission, direct digital control and optimal control, and database queries and updates. A reason for choosing these types of applications for in depth examination is that they require different *imprecision management rules*. These rules govern whether approximate results produced by prematurely terminated tasks are acceptable and what error recovery operations are to be carried out when the results are not acceptable.

Following this introduction, Section 3.1.2 gives a brief overview of the different ways to implement imprecise computations, workload models used to characterize them, and algorithms to schedule them. Section 3.1.3 describes an architectural framework for the integration of an imprecision mechanism and a traditional checkpointing mechanism, as well as the ICS system built on this framework. Section 3.1.4 describes algorithms for scheduling replicated imprecise computations and Section 3.1.5 a process structure that supports imprecision and replication-masking. Section 3.1.6 discusses the imprecision management rules for different types of applications. Future extensions are discussed at the end of each section. Section 3.1.7 is a summary.

3.1.2 Imprecise Computation

An imprecise system can be represented abstractly by a variation of the precedence graph in Figure 3.1.1. Specifically, this graph represents an imprecise system $\mathbf{T}$ as a set of related tasks. Each task is represented by a node. Nodes are depicted by boxes of all shapes in this figure. Tasks may be dependent; data and control dependencies between tasks impose precedence constraints on the order of their execution. There is an edge from a node T_i to a node T_j if the task T_i must be completed before T_j can begin execution.

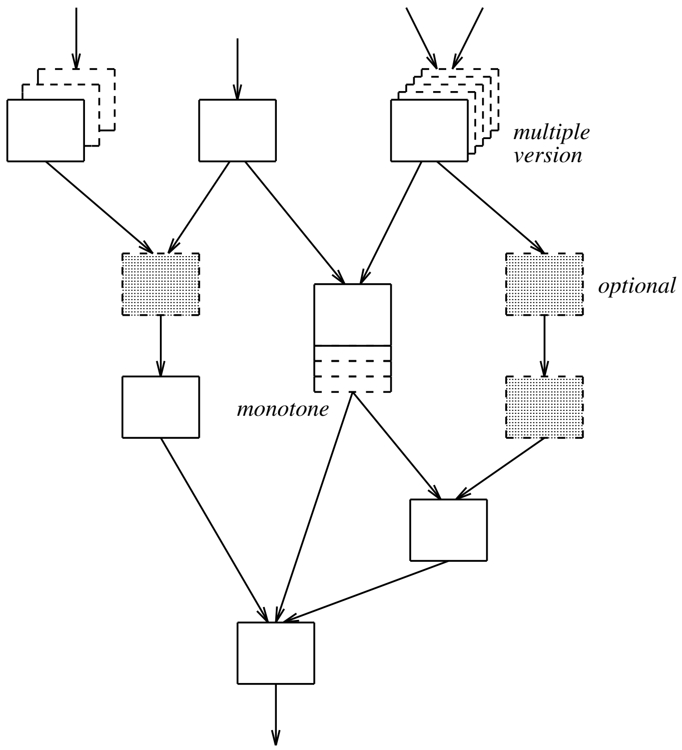

Figure 3.1.1: General Imprecise Computation Model

Implementation Methods

In an imprecise system, some tasks are identified by the programmer as *mandatory*. Like tasks in traditional real-time systems, mandatory tasks must be executed and completed by their deadlines in order for the system to function correctly. These tasks are shown as solid boxes in Figure 3.1.1. The programmer identifies some less important tasks as optional, meaning that these tasks can be skipped (i.e., not executed) without causing intolerable degradation in the output of the system. Optional tasks are represented by dotted, shaded boxes in Figure 3.1.1. The system may skip optional tasks during overload so that mandatory tasks can complete in time.

Sieve Method and the 0/1 Constraint For example, in radar signal processing the task that estimates the background noise level in the received signal can be skipped.

An old estimate of noise level can be used when a new estimate cannot be produced in time. Therefore this task is optional. Similarly, in a flight management system the step that updates the estimated time of arrival can be skipped when the system is busy dealing with local turbulence and nearby traffic and is, therefore, optional. In general, we call a task whose sole purpose is to produce a result that is at least as precise as the corresponding input or an old output a *sieve* function. A sieve can be skipped in order to save time and resource. When it is skipped, the result is less precise. Conversely, when a sieve completes in time, it improves the quality of the result. This method for producing approximate results is called the *sieve method*.

There is no benefit gained by completing a sieve in part. Therefore, we want to either execute such an optional task to completion before its deadline or skip it entirely. In this case, we say that the execution of the optional task satisfies the *0/1 constraint*. When tasks have the 0/1 constraint, the scheduler must decide before the execution of each task begins whether to schedule the task entirely or to skip it entirely. Some scheduling flexibility is thus lost.

Monotone Tasks and Milestone Method A task, as well as the underlying computational algorithm, is said to be *monotone* if the quality of the intermediate result produced by it is non-decreasing as it executes longer. We can logically decompose each monotone task into a mandatory task and an optional task. A monotone task produces a precise result when the entire task completes. An approximate result can be made available by recording the intermediate results produced by the task at appropriate instants of its execution. If the task is terminated before it is completed, the approximate result produced by it at the time of its termination is the best among all the intermediate results produced before its termination. This result is usable as long as its mandatory task is completed. This method for returning approximate results is called the *milestone method*.

Clearly, the milestone method relies on the use of monotone computational algorithms. Such algorithms exist in many application domains, including numerical computation, statistical estimation and prediction, sorting, facsimile transmission [21], video and voice transmission [22,23], and database query processing [24,25]. Monotone tasks can be easily implemented in most existing programming languages; an example illustrating how to implement monotone tasks in Ada 9x can be found in [26].

When tasks are monotone, the decision on which optional task and how much of the optional task to schedule at any time can be made dynamically. Because the scheduler can terminate a task any time after it has produced an acceptable result, scheduling monotone tasks can be done on-line or nearly on-line. Consequently, we have the maximum flexibility in scheduling when imprecise computations are implemented using this method.

Multiple-Version Method For some applications, neither the milestone method nor the sieve method can be used. Trading off result quality for processing time is nevertheless possible by using *multiple versions* of tasks. Using this method, we need to provide at least two versions of each task: the primary version and the alternate version(s). The primary version of each task produces a desired, precise result but has a longer processing time. An alternate version has a shorter processing time but produces an approximate result. During a transient overload, when it is not possible to complete the primary version of every task by its deadline, the system may choose to schedule alternate versions of some tasks.

The feasibility and effectiveness of the multiple version method has been demonstrated for both real-time computing and data communication; tools and environments have been developed to support this method [27–33]. A higher storage overhead incurs because of the need for maintaining multiple versions. Performance data shows that there is little advantage to be gained by having more than two versions [30,32].

For scheduling purpose, we model the alternate version of each two-version task as a mandatory task and the primary version as a mandatory task followed by an optional task. The processing time of the mandatory task is the same as that of the alternate version, and the processing time of the optional task is equal to the difference between the processing times of the two versions. The optional task must be either scheduled and completed by its deadline, corresponding to the primary version being scheduled, or skipped entirely, corresponding to the alternate version being scheduled. Consequently, scheduling tasks that have two versions is the same as scheduling tasks with the 0/1 constraint.

Error Characteristics and Scheduling Algorithms

For the purpose of scheduling, we logically decompose each task T_i into two tasks, the *mandatory* task M_i followed by the *optional* task O_i, independent of the method used to implement it. Let τ_i, m_i and o_i denote the processing times of T_i, M_i and O_i, respectively. Clearly, $m_i + o_i = \tau_i$. The classical deterministic model is a special case of this imprecise computation model where all tasks are mandatory, that is, $o_i = 0$ for all i. Similarly, a sieve or an anytime computation is a task that is entirely optional.

In general, the error of a task T_i is a function of the processing time of the portion that is executed, as well as the errors in the inputs of the task. Let σ_i denote the amount of processor time that is assigned to the optional task O_i. All existing scheduling algorithms assume that the inputs to every task have zero error. In other words, the *error* ϵ_i in the result produced by the task T_i (or simply the error of T_i) depends solely on σ_i and is given by the *error function* $E_i(\sigma_i)$ of the task. Moreover, when σ_i is equal to o_i, that is, when the scheduler allows T_i to execute until it terminates normally, its error ϵ_i is zero.

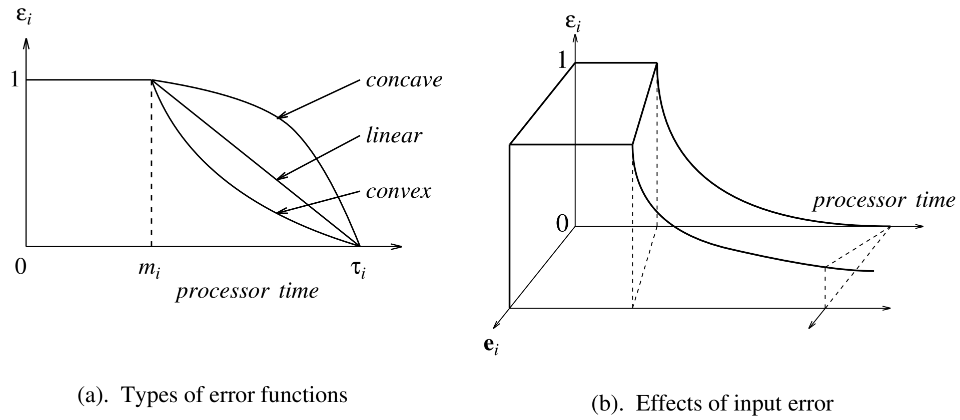

Figure 3.1.2: Error Characteristics

The error in the result of a task typically decreases in discrete steps and is, therefore, discontinuous. Because it is difficult to work with discontinuous functions, we typically approximate them by the continuous error functions. The continuous error functions were found to be good approximations except when the error decreases to zero in very few (e.g., two or three) steps [8]. Figure 3.1.2(a) shows three types of error functions that characterize the general behavior of different monotone computations. When the scheduler works correctly, it allows every mandatory task to complete. Therefore, the value of the error in the range where the amount of assigned processor time is less than the mandatory processing time is unimportant; for the sake of convenience, we let this value be 1.

When the exact behavior of error functions is not known, a reasonable choice is $\epsilon_i = (o_i - \sigma_i)/o_i$ for all i; that is, the error of a task is equal to the processing time of its skipped portion. For this reason, most studies on scheduling imprecise computations assume this linear error function. For a given schedule, the *total error* of the task set $\mathbf{T}$ is $\epsilon = \sum_{i=1}^{n} w_i \epsilon_i$, where w_i are the weights of the tasks. By choosing the weights, we can account for the different degrees to which the errors in the results produced by the individual tasks impact the overall quality of the result produced by the entire task set. Examples of optimal and suboptimal algorithms that minimize the total error can be found in [8,11,15].

An iterative algorithm, such as Newton's method for finding roots of polynomials, typically gives us a monotone task whose error decreases faster earlier during its execution. The rate of convergence to zero error decreases as such a task continues to execute. Its error function behaves like the convex function in Figure 3.1.2(a). Many other computations, such as statistical estimation, also have this type of error function. The average error of tasks with convex error functions can be kept small by making the maximum normalized error of all tasks as small as possible. Given a schedule of a set $\{T_i\}$ of tasks with identical weights, the *maximum normalized error* of the task set is $\epsilon = \max_i \{(o_i - \sigma_i)/o_i\}$. Polynomial-time algorithms for finding optimal schedules with the smallest maximum normalized error can be found in [13,17].

The error of a monotone task whose error function is concave decreases at a faster rate as it executes longer. The optional part of such a task should be scheduled as much as possible or not scheduled at all. In the limit, the error of an optional task with the 0/1 constraint stays at the maximum value of 1 until the optional task completes. In a schedule that satisfies the 0/1 constraint, the amount of processor time assigned to every optional task is equal to either o_i or 0. The general problem of scheduling to meet the 0/1 constraint and timing constraints, while minimizing the total error is NP-complete when the optional tasks have arbitrary processing times. Approximate algorithms with reasonably good worst-case performance can be found in [14]. When the optional tasks have identical processing times and weights, tasks with the 0/1 constraint can be optimally scheduled in $O(n \log n)$ time or $O(n^2)$ time depending on whether the tasks have identical or different release times. These algorithms can be found in [11].

Future Work in Scheduling

Again, all existing algorithms for scheduling imprecise computations assume that tasks have zero input errors. Moreover, the release times and deadlines of individual tasks are given. Both these assumptions are often not valid. The timing constraints of a set of dependent tasks are often end-to-end in nature. As long as the last task(s) in the set completes before its end-to-end deadline, when the individual tasks in the set complete is unimportant. The possibility of postponing the deadlines of some tasks gives an additional dimension in the tradeoff between the quality and the timeliness of the overall result and makes the problem of scheduling imprecise computations more difficult.

The result produced by a task may be an input to its immediate successors. When this result is imprecise, the input is erroneous. We can account for the effect of errors in the inputs of a task T_i by using an error function that is also a function of input errors. Specifically, let the errors in the inputs of a task T_i be denoted by the vector $\mathbf{e}_i$. The error in the result produced by T_i whose optional task is assigned σ_i units of processor time is given by $\epsilon_i = E_i(\sigma_i, \mathbf{e}_i)$, where $E_i(\sigma_i, \mathbf{e}_i)$ is a non-increasing function of σ_i and a non-decreasing function of $\mathbf{e}_i$. Figure 3.1.2(b) illustrates the effect of input errors. A task may need to do some additional work to correct its input error, and a poorer input may slow down the rate at which its result converges to the precise one. Consequently, the processing times of both M_i and O_i may become larger as the input error e_i increases. When the magnitude of e_i becomes larger than some threshold, the processing time $o_i(\mathrm{e}_i)$ of the optional task O_i may become infinite, and the task T_i can never produce a precise result no matter how long it executes. The need to keep the error in an input below this threshold determines when a result produced by each predecessor task is acceptable and how large the processing time of the mandatory part of the predecessor task should be. We are developing scheduling algorithms that take input errors into account and make use of the end-to-end nature of the deadlines when trying to minimize the error in the overall result.

3.1.3 Integration of Imprecision with Checkpointing

As stated earlier, the mechanism for the storage and return of the intermediate, approximate results of computations can be easily integrated with the traditional checkpointing mechanism for fault tolerance. This section describes an architecture that allows this integration and the Imprecise Computation Server (ICS) system which implements this architecture. Throughout this section, we confine our attention to transient faults and failures.

To make approximate results available in an imprecise system, the intermediate results produced by each task must be recorded as the task progresses toward completion. The application programmer defines the intermediate result variables and the accuracy measure variables to record, as well as the time instants at which their values are recorded. Each set of recorded values can be viewed as a checkpoint. Therefore, we call the mechanism for returning intermediate approximate results *user-directed checkpointing*. In contrast, the traditional checkpointing mechanism is *system-directed*.

Using user-directed checkpointing to supplement system-directed checkpointing has several obvious benefits. Because the application programmer can make use of the semantics of the individual computations, it is possible to keep the amount of state information recorded at each user-directed checkpoint small. For example, it suffices to record two consecutive intermediate roots produced by an iterative root-finding computation or the current sample mean and variance of a simulation run. In the absence of user provided checkpointing routines, the operating system must save the entire address space of each task. Consequently the cost of user-directed checkpointing is usually lower than for system-directed checkpointing. Furthermore, since fault recovery can make use of user-directed checkpoints as well as system-directed checkpoints, system-directed checkpoints need not be done as often, thus reducing the cost of providing fault tolerance. When a task terminates before it is completed, recovery action is not necessary if the last result returned was sufficiently good. Thus the processing time that would be spent repeating the work between the last checkpoint and the point of failure may be saved.

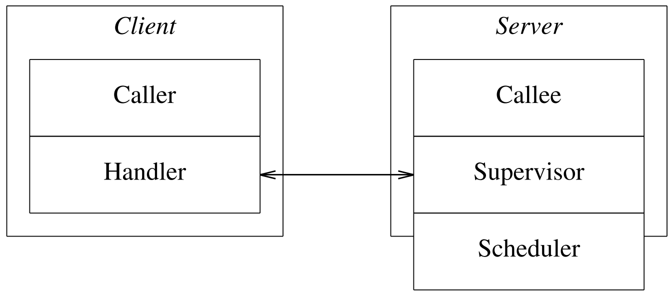

Figure 3.1.3: ICS Process Structure

An Imprecise System Architecture

The process structure that we use for the result-saving purpose is the simple variation of the client-server model shown in Figure 3.1.3. There is a server type for each service provided by an imprecise system. Each server consists of a callee and a supervisor, while each client consists of a caller and a handler. The caller and callee are application-specific and are written by the application programmer. The handler and supervisor are part of the underlying system. The supervisor is created at configuration time or at the instantiation time of the server. It may be dedicated to one server or shared by several servers. These details are unimportant for the purpose of this discussion.

As in traditional systems, the callee executes whenever it is called by the client. The client calls a server by sending an invocation request to the server's supervisor. Typically, only the client knows how accurate a result must be for it to be acceptable and how many faults each server invoked by it must tolerate. On the other hand, only the scheduler has the information on the demand and the availability of all resources. Both kinds of information are needed to support the decision on what quality of service to provide to each client and which tasks to terminate when an imprecise system recovers from a transient overload. To support this decision, the information regarding the result accuracy required by the client, the minimal level(s) of accuracy achievable by the server, and the required number of tolerated faults is exchanged between the client, the server, and the scheduler at invocation time. We will return shortly to describe the interactions between these system components for the purpose of this information exchange.

When the supervisor grants the request, it activates the callee. When the time allocated for the callee's execution expires, the supervisor terminates the callee. The client has the final, precise result when the callee terminates normally. If the callee terminates prematurely, the client has the best approximate result produced by the callee before it terminates. Based on the latest recorded value of the accuracy measure variable, the handler can decide whether this approximate result is acceptable.

In addition to aiding the result-saving process, the supervisor also checkpoints the callee and carries out sanity checks on a periodic basis in order to enhance fault tolerance. Because the supervisor and the callee do not communicate during the callee's execution, a fault occurring during the callee's execution does not affect the supervisor. We further assume that the effect of any fault occurring during the callee's execution is confined to it. This simplifying assumption allows us to focus on the interaction between the imprecision and fault-tolerance mechanisms.

When a failure is detected before the callee's mandatory portion completes, recovery is necessary. The supervisor restores the callee to its state at the time of the last checkpoint and resumes its execution from that state. We call such an action a *system-directed recovery action*.

An imprecise system must also support *user-directed recovery*. For real-time applications, forward recovery is typically more appropriate. Moreover, it can often be done effectively, especially for embedded applications. To carry out forward recovery, the client may want the sequence of approximate results produced by the callee during the course of its execution, not just the last result before its termination. This sequence of approximate results is often the consecutive discrete-time sample values of a continuous-time function. The future values of such a function can be estimated by extrapolating from the available past sample values. Depending on the semantics of the problem, there may be several ways to do this extrapolation to arrive at different acceptable estimates of the final result. For example, the application programmer may provide the client with some routines that implement filters with prediction or pure prediction filters. If the callee terminates prematurely, the client would take the approximate results and apply one of the given routines on them to generate a better approximation of the final result. In this way, the application system can better control of the quality of approximate results.

Scheduling Checkpointed Imprecise Tasks

From the scheduler's point of view, an imprecise application system $\mathbf{T}$ is a set of n (callee/server) tasks. Suppose that the supervisor of the callee T_i takes a checkpoint every l_i units of time, and that it takes c_i units of processor time to generate a checkpoint. While a checkpoint is generated, a sanity check is made. We assume that sanity checks never fail, and a failure occurring at any time instant is detected by the next sanity check after the instant. The callee task is allowed to fail k_i times. (k_i is specified by the client in its invocation request, in a manner described later in this section.) Some other recovery measure will be taken if it fails more than k_i times. Let ρ_i denote the amount of time the supervisor executes in order to roll back the callee task. A task T_i

can tolerate k_i faults only when a sufficient amount of processor time is allocated so that its mandatory portion can complete before its deadline even when it must be rolled back k_i times.

In general, we say that a schedule is k_i-tolerant for task T_i if the task can recover from k_i failures without causing any task in the system $\mathbf{T}$ to miss its deadline. A schedule for the system $\mathbf{T}$ is $\mathbf{k}$-tolerant, where $\mathbf{k}$ denotes the vector $(k_1, k_2, \ldots, k_n)$, if it is k_1-tolerant for T_1, k_2-tolerant for T_2, and so on. The goal of scheduling checkpointed imprecise tasks in $\mathbf{T}$ is to find a $\mathbf{k}$-tolerant schedule that minimizes the total error when all $k_1 + k_2 + \cdots + k_n$ failures occur. (Here, the error of each callee task T_i is equal the processing time of its skipped optional portion.) To accomplish this goal, the scheduler must allocate at least $m_i{}' = m_i + \lfloor m_i/l_i \rfloor c_i + k_i(l_i + c_i + \rho_i)$ units of time to the task when the processing time of its mandatory portion is m_i.

When tasks are statically bound to processors and tasks on each processor are scheduled as in a uniprocessor system, the algorithm called Algorithm C_{OL} in [33], can be used for this purpose. When the tasks are off-line, this algorithm is optimal, in the sense that it can always find a k-tolerant schedule if such a schedule exists and the total error of all the tasks in $\mathbf{T}$ is minimized. (A task system is off-line when the release times, deadlines and processing times of all the tasks in it are known before the processor begins to execute any task.) The version described below assumes that the tasks are on-line; the algorithm is optimal only when the system satisfies the feasible mandatory constraint [15].

Similar to the NORA algorithm for scheduling on-line imprecise tasks which are ready for execution at their arrival times [15], Algorithm C_{OL} maintains a reservation list of time intervals for all tasks that have been released but are not yet completed. This list is updated each time a new task T_i is released; as a result of this update, an interval (or several intervals) of length equal to $m_i{}'$ is reserved for it. The reservation list is derived from a feasible schedule of all the unfinished portions of the mandatory tasks. This schedule is generated by backward scheduling according to the latest-ready-time-first rule; a time interval is reserved if some task is scheduled in the interval. The scheduler uses the reservation list as a guide. It never schedules any optional task in a reserved interval. The reservation list is also updated as the execution of each mandatory task M_i progresses to completion. At an update when y units of the mandatory portion have already been completed and z of the k_i tolerable failures have occurred, the amount of time reserved for M_i is reduced to $m_i - y + \lfloor (m_i - y)/l_i \rfloor c_i + (k_i - z)(l_i + c_i + \rho_i)$. The reservation for it is eventually deleted when the mandatory portion completes.

Algorithm C_{OL} differs from the NORA algorithm in the way it schedules mandatory and optional tasks. Algorithms C_{OL} maintains an EDF-ordered main task queue and an optional task queue. When a task is released it is put into the main task queue after the reservation list is updated. The first task in this queue is scheduled for execution. The task executes for as long as there is time before the beginning of the earliest reserved interval. When the task completes or is terminated at its deadline, it is removed from the main task queue and its reservation is deleted from the reservation list. If before its completion or its deadline, the beginning of the earliest reservation interval is reached, the reservation of the task is deleted if it has any reserved time according to the current reservation list. Otherwise, the task is terminated and is removed from the main task queue. If a task is not completed when it is removed from the main task queue, it is placed in the optional task queue, which is also EDF-ordered. When the main task queue is empty, the processor executes the optional task at the head of the the optional task queue, until the completion or the deadline of the task; it is then removed from the optional task queue. The elegance of this algorithm in combination with a checkpointing scheme is that any unused recovery time is automatically made available for the other tasks, allowing their optional portion to execute longer. The performance of this algorithm for scheduling on-line transactions that have bounded-response-time and high-availability requirements has been evaluated and performance data can be found in [33].

The ICS System

We are implementing the ICS (Imprecise Computation Server) system based on the architecture described above. ICS runs on top of the Mach operating system and is integrated with the Mach Interface Generator (MIG) [34]. MIG takes an interface description for a service and generates interface code for the client and the server. This interface hides the call to the remote server from the client, making it appear to be a local procedure call.

Interfaces To develop an application system using ICS, the application programmer first defines the interface for each service, and writes a MIG specification file that describes the interface. A modified version of MIG, called IMIG, is run to generate the interface code. IMIG adds an argument to each of the interfaces, which is used to exchange ICS-specific information between the caller and the callee. Specifically, it is a pointer to a data structure containing scheduling information; a portion of this structure is shown in Figure 3.1.4. The *requiredAccuracy* and *outputAccuracy* parameters measure the accuracy of the result. The *deadline* and *mandatoryTime* parameters allow the programmer to specify the timing requirements of the application system. The *resources* parameter is used for server selection. The *toleratedFaults* parameter specifies the number of faults that the server must be able to tolerate. This information, together with the value *mandatoryTime*, is used by the scheduler to determine the amount of processor time to reserve for the mandatory portion.

Next, the programmer writes the code that implements the callers and callees in the application system and then compiles and links the code with the ICS library, which

```
struct icsInfo {
    long  requiredAccuracy;     /* User Inputs.  */
    struct timeval deadline;
    long resources;

    int serviceNumber;          /* System Inputs.  */
    struct timeval mandatoryTime;
    int toleratedFaults;

    long outputAccuracy;        /* Outputs.  */
};
```

Figure 3.1.4: The *icsInfo* Structure

	Client Side	Server Side
ICS-supplied	icsFindService client interface routine	icsRegisterService icsService result-saving routine
User-supplied		server interface routine checkpoint routine restart routine

Table 3.1.1: ICS Functions

contains the supervisor and handler code. Each client or server produces a separate application program.

ICS's interface routines fall into two groups: client-side routines and server-side routines. They may be further divided into those that are provided by ICS to be called by the application and those that are written by the programmer to be called by ICS. The routines and their classifications are listed in Table 3.1.1. Details on these routines and their usage can be found in [35].

The client interface routine is the client's interface to a server and is called by the client when it wishes to make use of a service. The server interface routine is the corresponding interface on the server side. When a client interface routine is called, the server interface routine is eventually invoked by a supervisor on the selected server.

The checkpoint routine provided by the application programmer is used for both result-saving and checkpointing purposes. If no checkpoint routine is specified, ICS will checkpoint the callee in the traditional manner by saving the callee's entire address space. In this case, ICS will be unable to return an approximate result to the caller if the callee fails; the computation will have to be restarted from the last checkpoint.

Service Establishment The programmer starts the application system by starting the application programs for each of the servers and clients in the system. Each server, after performing any necessary initialization, registers itself with its supervisor by calling *icsRegisterService*. This call informs the supervisor of the name of the service, the resources required by the callee, and the routines to call to perform checkpointing and to restart the callee from a checkpoint. Once the service has been registered, the server calls the *icsService* routine, which passes control to the supervisor. The supervisor advertises the services that it provides and waits for requests from clients.

To make use of a service, a client first calls the *icsFindService* routine with the name of the service it wishes to call. *icsFindService* returns a service number. The service number is used by the handler to keep track of all of the servers for that service.

When a client calls a server, the caller fills in the *icsInfo* structure and calls the ICS-supplied client interface routine. This interface routine invokes the handler. The handler uses the scheduling information provided in this structure to select a server and sends a message to the chosen server's supervisor. The supervisor creates a new thread of control for the callee and informs the scheduler of the callee's deadline and processing time requirements. It then starts the callee when the scheduler grants its request to start the callee.

After starting the callee, the supervisor sets a timer. When the timer expires, the supervisor calls the callee's checkpoint routine, which instructs the callee to call its result-saving routine. When the callee wants to save an approximate result, it calls its result-saving routine itself. The result-saving routine, which is generated by IMIG, sends the result and the callee's current state to the handler that originated the request. The result-saving routine also resets the timer, thus delaying the next supervisor-invoked checkpoint. If the handler finds that the server has died or has missed its deadline, it can use the state information received at the last checkpoint to decide whether to return the result that has been computed so far to the caller, or to carry out some user-directed recovery action, or to use the saved state to restart the request on another server.

Extensions to ICS We have implemented a prototype version of ICS and the design seems reasonable. Once we complete the version described here, there are several extensions we would like to make. One extension is to replicate servers on redundant processors. The following section discusses an approach to use the imprecise computation technique to reduce the overhead of replication. Another approach is to replicate only the mandatory portions. The approximate results produced by the mandatory portions are used to ensure the correctness of the refined results produced by the unreplicated optional portion. We will take this approach because it is more compatible with the ICS architecture.

In the near future, we will extend the ICS system to support *imprecise service negotiation*. When it is not possible for the scheduler to allocate sufficient amounts of processor time or other resources to grant a request, it informs the supervisor of the available amounts. Rather than simply denying the request, the supervisor and client may enter an optional negotiation phase. The willingness of the client to accept a poorer quality result that can be produced with the available resources is determined. The callee is scheduled if the client can use such a result. In this way, the quality of each imprecise computation can be negotiated at invocation time.

3.1.4 Integration of Imprecision and Replication

For real-time applications, replication and masking is a more appropriate approach to provide fault tolerance whenever there is more than one processor. This section describes first algorithms for scheduling replicated periodic imprecise tasks and then a process structure that allows replication be incorporated in an imprecise system.

Scheduling Replicated Imprecise Tasks

We consider here a system **T** of n independent, replicated periodic tasks. They are to be assigned to identical non-faulty processors and allocated sufficient processor time for them to produce acceptable results in time.

The Allocation Problem We say that the system is in *mode j* when the system contains j non-faulty processors. Each task T_i is replicated. There are $c_i(j)$ clones, or copies, of T_i when the system is in mode j. $c_i(j)$ is chosen by the application programmer. The set $\{c_i(j)\}$ is an input of the task assignment module. When invoked, this module allocates a fraction of processor time to (each clone of) each task and finds an assignment of the clones of all the tasks to the j processors.

An assignment in mode j is a valid one if the $c_i(j)$ clones of every task T_i is assigned on a one-clone-per-processor basis to $c_i(j)$ non-faulty processors. Moreover, let $x_i(j)$ be the amount of processor time per period that is allocated to each clone of task T_i. Because the mandatory portion of every task must always be completed, $x_i(j)$ must be larger than the processing time m_i of the mandatory task M_i of T_i. Since the optional portion need not be completed, $x_i(j)$ may be less than the processing time τ_i of the task as a whole. Let p_i be the period of T_i, and $u_i(j) = x_i(j)/p_i$ be the fraction of processor time allocated to each clone of T_i. We refer to $u_i(j)$ as the *allocated* *utilization*, or simply the *utilization*, of T_i. For an assignment to be valid, the allocated utilization $u_i(j)$ must be such that $m_i/p_i \leq u_i(j) \leq \tau_i/p_i$ for all $1 \leq i \leq n$.

Once assigned to a processor, a clone is bound to the processor until the system is reconfigured, when the task assignment module is invoked again to generate a new assignment. The clones assigned to each processor are scheduled according to a local scheduling algorithm, such as the rate-monotonic or the earliest-deadline-first algorithm. The *schedulability threshold* is determined by the algorithm used. (The value of this parameter gives the maximum total utilization of the tasks assigned to a processor to ensure the schedulability of the tasks)

An assignment is *feasible* if it is valid and if the mandatory portion of every clone can complete and produce an acceptable result by its deadline. A sufficient condition that ensures the timely completion of each clone is to keep the total utilization of all the clones assigned to every processor lower than the schedulability threshold of the processor.

A system often has more than one feasible assignment. Our algorithms compare different feasible assignments according to the *total weighted utilization*, which is $U(j) = \sum_{i=1}^{n} w_i u_i(j)$ where w_i is the weight of the task T_i. The larger the value of $U(j)$, the more processor time is available to execute optional tasks, and the better is the overall result quality.

The allocation problem solved by our algorithms can now be stated more precisely: We are given a set of n independent periodic tasks, j identical non-faulty processors, the set $\{c_i(j)\}$ of required clones per tasks, and the schedulability threshold of each non-faulty processor to ensure the timely completion of all the clones on the processor. The algorithms try to find a feasible assignment which has the maximum total weighted utilization among all feasible assignments.

Assignment Algorithms We have designed two heuristic algorithms, the *Allocate* algorithm and the *Modify* algorithm, to solve the problem described above [36,37]. The Allocate algorithm is used at system configuration time and during major reconfigurations. It seeks a good initial assignment of tasks to processors. In contrast, the Modify algorithm works with an existing assignment and tries to find a new feasible assignment with the least changes from the existing assignment. It is used for clone reconfiguration as processors fail or recover.

Both algorithms have three phases. In the first phase, a greedy algorithm computes an upper bound of $U(j)$ and an upper bound of the allocated utilization $u_i(j)$ of each task. The greedy algorithm derives these upper bounds by ignoring some constraints in the original problem. (Specifically, it may assign more than one clone of a task to the same processor. It also assumes that a clone can execute on one processor and then migrate to another processor to complete its execution.) Because these constraints are ignored, the assignment produced by the greedy algorithm may not be valid. In the second phase, each algorithm generates a valid task assignment using the upper bounds derived in phase one as input. The algorithms used in this phase are based on well-known heuristics for bin packing. Examples are least-utilization-best-fit, least-number-of-clones-first-fit, and multifit. Because the upper bounds of task utilizations are used, it is likely that the total utilizations of some processors exceed the schedulability threshold. The third phase adjusts the allocated utilizations of the tasks so that the total utilization of each processor is less than the schedulability threshold and the total weighted utilization $U(j)$ is maximized. The problem solved in the third phase is formulated as a linear-programming problem, and is solved using known efficient linear-programming algorithms.

The difference between the Allocate algorithm and the Modify algorithm lies in the second phase in which different routines are invoked to produce desirable task assignments. The desirable task assignments sought by the Allocate algorithm are those close to being optimal. On the other hand, the Modify algorithm seeks feasible assignments that maintain as much location continuity as possible with respect to the existing task assignment that was given as input. To achieve this objective, the routine called by the Modify algorithm in phase two will not move those clones already assigned to the non-faulty processors and tries to maximize $U(j)$ under this constraint. Detailed descriptions of these algorithms and their performance can be found in [36,37].

3.1.5 Process Structure for Replication

With replication, the system can tolerate both transient and permanent physical faults in the processors. We assume that the mean time between failures is much greater than the time needed for a failure recovery. During a failure recovery, the system will remain functional as long as there are enough existing clones to mask the failures.

We can also incorporate replication with imprecision using a server-client process structure. Specifically, in a system where both the client T_c and the server T_s are replicated, the result produced by each clone of a server is broadcast to all the clones of the client. The additional time needed for broadcasting is included in the processing time of server task. Because an accuracy measure is returned with each result, a client clone may choose to use any one of the results that arrive in time based on the quality of the results. Neither agreement nor synchronization among the client clones is necessary.

It is easy to show that when both the client and the server are periodic, the maximum latency for detecting a failure of the server is equal to $p_c + p_s$, where p_c and p_s are the periods of the client and server, respectively. The system provides two types of servers, *diagnosis servers* and *repair servers*, to detect failures and recover from them. Each diagnosis server is responsible for monitoring some number of processors. It polls the processors it services, either upon request by some task or periodically, to determine their status. Each processor is monitored by more than one diagnosis server. In this way, the diagnosis servers are independently replicated to provide concurrent, independent services. When a faulty processor is detected by a diagnosis server, it calls a repair server.

Upon request by a diagnosis server, a repair server carries out a *clone reconfiguration*. This is done in two phases. Using the Modify algorithm, it first finds a new task assignment to relocate the clones assigned to the faulty processor. It then installs the new assignment on the non-faulty processors. During this installation, the amount of processor time allocated to each clone may be increased or decreased. Some new clones may need to be added to some non-faulty processors. This is essentially a mode change operation. This operation can be done in an iterative manner. The repair server sends the information on the new clone allocated utilization iteratively to each non-faulty processor. The number of clones and the allocated utilizations of the clones are changed in such a way that the number of clones of every task T_i on non-faulty processors is never less than $c_i(j)$ when the system is in mode j [38].

In particular, according to the clone reconfiguration scheme described in [38], the repair server is monotone. Therefore, if another processor fails while a repair server is carrying out clone reconfiguration, we can terminate the ongoing repair service. Because the Modify algorithm is used, the server monotonically reduces the number of the excessive clones and the number of the missing clones to zero. If a repair server is terminated during the first phase before a new assignment is found, the incomplete assignment it has generated before its termination is treated as an approximation of the final assignment. Another clone reconfiguration can be started based on this approximate assignment. When a clone reconfiguration is interrupted during the second phase of the execution, some clones may have been partially reconfigured. Since the method used in the second phase is iterative, it can also be aborted at any time. The system is in a partially reliable state if some of the new clones are not installed at the time, but the new configuration is an improvement of the existing one, and each iteration by the server improves or preserves the reliability of the system.

3.1.6 User-Directed Recovery and Imprecision Management

In practice, it is often impossible to bound the time required to produce an acceptable result. As an example, we consider a task that iteratively finds a root of a polynomial. How long the task must execute to reduce the error to an acceptable level depends not only on the coefficients of the polynomial but also on the region where is the root is expected to be found. The supervisor may underestimate the processing time of the mandatory part. Furthermore, the required accuracy of the result may depend on the value of the result. For example, how accurate the estimated positions of several cooperative robot arms may be depends on the positions of the arms. The closer they

Application Domain Type	Decision Support Information or Recovery Actions
Non-cumulative	Acceptance of x_i is based on x_i. Recovery action depends on x_i. Alternative recovery actions: refine x_i.
Limitedly-cumulative	Acceptance of x_i is based on x_i and x_{i-1}. Recovery action depends on x_i and x_{i-1}. Alternative recovery actions: refine x_i or precompute x_{i+1}.
Cumulative	Acceptance of x_i is based on $x_i, x_{i-1}, x_{i-2}, \ldots$. Recovery action depends on $x_i, x_{i-1}, x_{i-2}, \ldots$. Alternative recovery actions: refine x_i, or precompute x_{i+1}, or recompute $x_{i-1}, x_{i-2}, \ldots$.

Table 3.1.2: Imprecision Management Policies

are and the greater the likelihood of collision, the more accurate the estimates must be. Consequently, the client may find that it requires better accuracy than it requested during invocation after examining the result.

The above examples illustrate that sometimes a client may find a result unacceptable and recovery is necessary, even though the mandatory part of the callee has been completed. In this case, the appropriate recovery action to take is typically application dependent. Whether some recovery action is necessary and what recovery action should be taken are likely to be dependent on the semantics of data, operations and applications. We call a policy that defines the rules governing when and what application-directed recovery is done an *imprecision management policy* and the mechanism that enforces a set of such rules an *imprecise management mechanism*.

To support a wide range of embedded, real-time applications, an imprecise system must provide three types of imprecision management mechanisms for three types of applications: (error) non-cumulative, limitedly-cumulative, and (error) cumulative. The policies governing these mechanisms are summarized in Table 3.1.2.

To explain the different application types, we consider a sequence of (server) tasks $\ldots, T_{i-2}, T_{i-1}, T_i, T_{i+1}, \ldots$ whose results are $\ldots, x_{i-2}, x_{i-1}, x_i, x_{i+1}, \ldots$, respectively. The client invokes the task T_i and requests the result x_i be produced by the time instant t_i in a sequence of time instants $\ldots, t_{i-2} < t_{i-1} < t_i < t_{i+1}, \ldots$. When each result is available, the client must decide whether to accept the result as it is or try to improve it in some way.

Non-cumulative Type In the simplest case, the tasks are independent and their inputs are precise. No information on the results can be carried from one task to the other tasks. The decision about whether to accept an approximate result x_i produced by a prematurely terminated task T_i of this type can be made independently of previous results and future results. When the system recovers from a fault causing a premature termination and there is time, it is clear that recovery action should be carried out to improve the result as much as possible. The problem is simply that of deciding how much of the available processor time allocated to the client should be spent on improving this approximate result when there are other approximate results competing for time. This problem is essentially one of scheduling.

In addition to some numerical computations and monotone database query processing [24,25], image transmission is an example of this type. Suppose that a progressive built-up method is used for the transmission of still images [21]. The data encoding each image frame is divided into four blocks; each additional block gives a clearer image. The input data encoding the image presented to each transmission task contain all four blocks. If the transmission of any image terminates prematurely due to network partition or congestion, the approximate result is a fuzzy picture as long as the transmission of one of the four blocks is completed. The receiver can decide whether the fuzzy picture is acceptable without considering the quality of other pictures. In this case, the recovery operation is relatively straightforward.

Transmission of compressed video (or voice) is a more complex application of non-cumulative type. Typically, frames of images are transmitted periodically. When the transmissions of some frames are terminated prematurely, the average error in incompletely transmitted frames over several consecutive periods has an observable effect. It is not necessary to complete the transmission of any frame. However, a sufficient number of frames of good quality must be transmitted every second in order to maintain an overall acceptable average quality. The decision on whether to accept an approximate result, that is an incomplete frame, must be made based on the average error over several periods. The fact that the timely delivery of each frame is more critical than the quality of each individual frame is also a critical factor in this decision.

Limitedly-Cumulative Type In an application of the limitedly-cumulative type, the current task T_i to be completed by t_i is similar to the previous task T_{i-1} with small changes due to small changes in the input data. The previous result x_{i-1} can be carried forward as an input to the current task. In particular, the result x_{i-1} can be used as an initial approximation of the result x_i to be produced by the current task, possibly even one accurate enough for use if there is insufficient time to execute T_i. The decision on whether to accept an approximate result x_i can no longer be made independently of the previous result. When the client receives x_i, it is no longer clear what to do: to carry out the recovery operation and improve the approximate result even though the result is sufficiently accurate, or to accept the approximate result and use the available processor time to start the next task. To make the optimal choice in this case requires the use of a cost function that includes the estimated errors of results at several time instants.

Traditional linear or approximately piece-wise linear feedback control is an example of limitedly-cumulative applications. A control law is typically computed periodically. The answer produced in a period can be used as an initial approximation of the answer in the next period. Because the effects of inaccuracy accumulate from one period to the next, when deciding whether to accept an approximate result in the current period we must also consider the accuracy of the result produced in the previous period.

Cumulative Type In an application of the (error) cumulative type, the current task T_i has as inputs the results of all previous tasks. An example of this type of application is the continuing solution of a differential equation in real-time simulation where the problem to be solved at any time instant is defined by the solutions of the problems solved at earlier time instants. In other words, the results produced by previous tasks determine the parameters of the current task. They may also be used to provide an initial approximation of the current result x_i. Again, it is not obvious how best to use the available processor time after an acceptable approximate result has been returned. In some situations it might be preferable to go back to improve some previous results, since the error remaining there will affect the accuracy of the current result, as well as future results.

Multiple-target tracking, non-linear control systems (such as bang-bang control) and hybrid systems (such as knowledge-based control and discrete-event/continuous control) are other examples of cumulative type. Imprecision management is considerably more complex for this type of application. The cost function used by the imprecision management mechanism must take into account the accuracy of results produced by many, or all, previous tasks. The dependency of the accuracy of future results on the accuracy of the current result must also be considered.

3.1.7 Summary

We have described the architecture of an imprecise, fault-tolerant system. In this system, the imprecision mechanism that is needed to record and return intermediate results of computations is integrated with the traditional checkpointing and recovery mechanism. They complement each other. The imprecision mechanism is application dependent and hence is only applicable in some application domains. Whenever it is applicable, it provides fault tolerance at a low cost. In contrast, the traditional mechanism is application independent and can be broadly applied and supported system-wide, but has a relative high overhead.

We are implementing the ICS system based on this architecture. ICS will make it easy for us to implement imprecise computations and experiment with them. Specifically, to develop an application system in ICS, we will only need to write the code to implement the application, leaving the ICS system to provide the code for the underlying support modules. Clearly, the result saving process must be efficient so that it will not degrade the normal system performance substantially. Similarly, the overhead incurred in the imprecise service establishment must be kept low. We are evaluating several alternative ways to implement these protocols to determine how best to meet these performance objectives. Once this version of the ICS system is completed, we plan to extend the ICS so that the application programmer can easily replicate the mandatory portions of some tasks if the replication approach is chosen.

The effectiveness and limitations of the imprecise computation technique can be demonstrated clearly only by examining application systems in a diverse spectrum of application domains. For this reason, we will focus on the experimental evaluation of this technique for several several representative applications. Again, the applications we plan to implement using the ICS system and evaluate in depth include multimedia data transmission, direct digital control and optimal control, and database queries and updates. We note that it is relatively easy to judge the quality of voices and images. Therefore, rules governing when a partially transmitted message carrying voice and images is acceptable are relatively simple. In the other extreme, the question of whether a partially completed data query or update is sufficiently good is much more difficult to answer.

References

[1] Liu, J. W. S., K. J. Lin and C. L. Liu, "A position paper for the IEEE 1987 Workshop on Real-Time Operating Systems," Cambridge, Mass, May, 1987.

[2] Lin, K. J., S. Natarajan, J. W. S. Liu, "Imprecise results: utilizing partial computations in real-time systems," *Proceedings of the IEEE 8th Real-Time Systems Symposium*, San Jose, California, December 1987.

[3] Liu, J. W. S., S. Natarajan, and K. J. Lin, "Scheduling real-time, periodic jobs using imprecise results," *Proceedings of Eighth Real-Time Systems Symposium*, pp. 252–260, San Jose, CA, December 1987.

[4] Dean, T. and M. Boddy, "An analysis of time dependent planning," *Proceedings of Conference of the AAAI*, 1988.

[5] del Val, A., "Approximate belief update," *Proceedings of Workshop on Imprecise and Approximate Computations*, Phoenix, Arizona, December 1993.

[6] Decker, K., V. Lesser, and R. Whitehair, “Extending a blackboard architecture for approximate processing,” *Real-Time Systems Journal*, 2, 1990.

[7] Leung, J. Y-T., T. W. Tam, C. S. Wong, and G. H. Wong, “Minimizing mean flow time with error constraints,” *Proceedings of the 10th IEEE Real-Time Systems Symposium*, December 1989.

[8] Chung, J. Y., J. W. S. Liu, and K. J. Lin, “Scheduling periodic jobs that allow imprecise results,” *IEEE Transactions on Computer*, Vol. 39, No. 9, pp. 1156–1174, September 1990.

[9] Leung, J. Y-T. and C. S. Wong, “Minimizing the number of late tasks with error constraints,” *Proceedings of the 11th Real-Time Systems Symposium*, Orlando, Florida, December 1990.

[10] Liu, J. W. S., K. J. Lin, W. K. Shih, A. C. Yu, J. Y. Chung, and W. Zhao, “Algorithms for scheduling imprecise computations,” *IEEE Computer*, pp. 58–68, May 1991.

[11] Shih, W. K., J. W. S. Liu and J. Y. Chung, “Algorithms for scheduling tasks to minimize total error,” *SIAM Journal of Computing*, Vol. 20, No. 3, pp. 537–552, June 1991.

[12] Zhao, W., S. Vrbsky and J. W. S. Liu, “An analytical model for multi-server imprecise systems", *Proceedings of the 5th International Conference on Parallel and Distributed Computing and Systems*, Pittsburgh, PA, September 1992.

[13] Ho, K. I. J., J. Y. T. Leung and W. D. Wei, “Minimizing maximum weighted error of imprecise computation tasks,” Technical Report, Department of Computer Science and Engineering, University of Nebraska, 1992.

[14] Ho, K. I, J., J. Y. T. Leung, and W. D. Wei, “Scheduling imprecise computation tasks with 0/1 constraints,” Technical Report, Department of Computer Science and Engineering, University of Nebraska, 1992.

[15] Shih, W. K. and J. W. S. Liu, “On-line scheduling of imprecise computations to minimize total error,” *Proceedings of the 13th IEEE Real-Time Systems Symposium*, Phoenix, Arizona, pp. 280–289, December 1992.

[16] Cheong, I., “Heuristic algorithms for scheduling error-cumulative, periodic jobs,” Ph.D. thesis, Department of Computer Science, University of Illinois, January 1993.

[17] Shih, W. K. and J. W. S. Liu, “Minimization of the maximum error of imprecise computations.” submitted.

[18] Ho, K. I. J, V. K. M. Yu, and W. D. Wei, “Minimizing the weighted number of tardy tasks units,” to appear in *Discrete Applied Math*.

[19] Obradovic, M. and P. Berman, “Voting as the optimal static pessimistic scheme for managing replicated data,” *Proceedings of the 9th IEEE Symposium on Reliable Distributed Systems*, October 1990.

[20] Koo, B. and S. Toueg, “Checkpointing and rollback-recovery for distributed systems,” *IEEE Transactions on Software Engineering*, January 1987.

[21] Wallace, G. K., “Overview of the JPEG (ISO/CCITT) still image compression standard,” Visual Communication and Image Processing, ’89, SPIE, Philadelphia, November 1989.

[22] Wood, J. and S. O’Neil, “Sub-band coding of images,” *IEEE Transactions on Acoustic Speech Signal Communications*, 34, October 1986.

[23] Suzuki, J. and M. Taka, “Missing packet recovery techniques for low-bit-rate coded speech,” *IEEE Journal of Selected Areas in Communications*, 9(7), September 1991.

[24] Buneman, P., S. Davidson, and A. Watters, “A semantics for complex objects and approximate queries,” *Proceedings of the Seventh Symposium on the Principles of Database Systems*, pp. 305–314, March 1988.

[25] Vrbsky, S. and J. W. S. Liu, “Approximate: a monotone query processing,” em IEEE Transactions on Knowledge and Data Engineering, October 1993.

[26] *Ada 9X Mapping*, Version 3.1, Ada 9X Mapping/Revision Team, Intermetric, Inc., Cambridge, Mass., August 1991.

[27] 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, October 1986.

[28] Gopinath, P. and R. Gupta, “Applying compiler techniques to scheduling in real-time systems,” *Proceedings of the 11th IEEE Real-Time Systems Symposium*, Orlando, Florida, December 1990.

[29] Kim, B. and D. Towsley, “Dynamic flow control protocols for packet-switching multiplexers serving real-time multipacket messages,” *IEEE Transactions on Communications*, Vol. COM-34, No. 4, April 1986.

[30] Yemini, Y., “A bang-bang principle for real-time transport protocols,” *Proc. SIGCOMM ’83 Symp. Commun. Architect. Protocols*, pp. 262–268, May 1983.

[31] Zhao, W. and E. K. P. Chong, “Performance evaluation of scheduling algorithms for dynamic imprecise soft real-time computer systems", *Australian Computer Science Communications*, Vol. 11, No. 1, pp 329–340, 1989.

[32] Lopez-Millan, V., W. Feng, and J. W. S. Liu, “A congestion control scheme for real-time traffic switching element using the imprecise computation technique,” submitted to the 1994 International Conference on Distributed Computing Systems.

[33] Bettati, R., N. S. Bowen, and J. Y. Chung, “On-Line Scheduling for Checkpointing Imprecise Computation,” *Proceedings of Euromicro 93 Workshop on Real-Time Systems*, Oulu, Finland, June 1993.

[34] *Mach 3 Server Writer’s Guide*, edited by K. Loepere, Open Software Foundation and Carnegie Mellon University, 1990.

[35] Hull, D. and J. W. S Liu, “ICS: A System for Imprecise Computations,” *Proceedings of AIAA Conference*, October 1993, also technical report in preparation.

[36] Yu, A., “Scheduling parallel real-time tasks that allow imprecise computations,” Ph.D. thesis, Technical Report UIUCDCS-R-92-1738, University of Illinois, 1992.

[37] Yu, A. and K. J. Lin, “A scheduling algorithm for replicated real-time tasks," *Proc. Phoenix Conference on Computers and Communications*, pp. 395–402, April 1992.

[38] Yu, A. and K. J. Lin, “Recovery manager for replicated real-time imprecise computations,” *Proceedings of IEEE Workshop on Parallel and Distributed Real-Time Systems*, Newport Beach, CA, April 1993.