Algorithms for Scheduling Imprecise Computations

Jane W.S. Liu, Kwei-Jay Lin, Wei-Kuan Shih, and Albert Chuang-shi Yu,
University of Illinois at Urbana-Champaign
Jen-Yao Chung, IBM T.J. Watson Research Center
Wei Zhao, Texas A&M University

In a hard real-time system, every time-critical task must meet its timing constraint, typically specified as its deadline. (A task is a granule of computation treated by the scheduler as a unit of work to be allocated processor time, or scheduled.) If any time-critical task fails to complete and produce its result by its deadline, a timing fault occurs and the task's result is of little or no use. Such factors as variations in the processing times of dynamic algorithms make meeting all deadlines at all times difficult.

The imprecise computation technique[1-3] can minimize this difficulty. It prevents timing faults and achieves graceful degradation by giving the user an approximate result of acceptable quality whenever the system cannot produce the exact result in time. Image processing and tracking are examples of real-time applications where the user may accept timely approximate results: Frames of fuzzy images and rough estimates of location produced in time may be better than perfect images and accurate location data produced too late.

In this article, we review workload models that quantify the trade-off between result quality and computation time. We also describe scheduling algorithms that exploit this trade-off.

Imprecise computation techniques provide scheduling flexibility by trading off result quality to meet computation deadlines. We review imprecise computation scheduling problems: workload models and algorithms.

Imprecise computation technique

A basic strategy to minimize the bad effects of timing faults is to leave the less important tasks unfinished if necessary. The system schedules and executes to completion all mandatory tasks before their deadlines, but may leave less important tasks unfinished.

The imprecise computation technique uses this basic strategy but carries it one step further. In addition to dividing tasks into mandatory and optional, programmers structure every time-critical task so it can be logically decomposed into two subtasks: a mandatory subtask and an optional subtask. The mandatory subtask is required for an acceptable result and must be computed to completion before the task deadline. The optional subtask refines the result. It can be left unfinished and terminated at its deadline, if necessary, lessening the quality of the task result.

The result produced by a task when it completes is the desired *precise* result, which has an error of zero. If the task is terminated before completion, the intermediate result produced at that point is usable as long as the mandatory subtask is complete. Such a result is said to be *imprecise*. A programming language in which imprecise computations can be easily implemented is Flex,[1] an object-oriented language that supports all C++ constructs along with timing-constraint and imprecision primitives.

0018-9162/91/0500-0058$01.00 © 1991 IEEE

Monotone time-critical computations provide maximum flexibility in scheduling. A task is monotone if the quality of its intermediate result does not decrease as it executes longer. Underlying computational algorithms enabling monotone tasks are available in many problem domains, including numerical computation, statistical estimation and prediction, heuristic search, sorting, and database query processing.[4]

To return an imprecise result of a monotone task, the intermediate results produced by the task are recorded at appropriate instances of its execution. Flex provides language primitives with which the programmer can specify the intermediate result variables and error indicators, as well as the time instants to record them. The latest recorded values of the intermediate result variables and error indicators become available to the user if the task prematurely terminates. The user can examine these error indicators and decide whether an imprecise result is acceptable. This method for returning imprecise results is called the *milestone method.*

For some applications, making all computations monotone is not feasible. In this case, result quality can be traded off for processing time through *sieve functions* — computation steps that can be skipped to save time. In radar signal processing, for example, the step that computes a new estimate of the noise level in the received signal can be skipped; an old estimate can be used.

In applications where neither the milestone method nor the sieve method is feasible, the *multiple version* method[5] almost always works. The system has two versions of each task: the primary version and the alternate version. The primary version produces a precise result but has a longer processing time. The alternate version has a shorter processing time but produces an imprecise result. During a transient overload, the system executes a task's alternate version instead of its primary version.

Programmers can easily implement both the milestone and sieve methods in any existing language. Tools and environments have been developed to support the multiple-version method in real-time computing and data communication.

The cost of the milestone technique is the overhead in recording intermediate results. The cost of the sieve technique is the higher scheduling overhead. Since there is no benefit in completing part of a sieve, while incurring the cost in processing that part, the execution of such an optional subtask must satisfy the *0/1 constraint:* The system must either execute it to completion before its deadline or not schedule it at all. Algorithms for scheduling tasks with the 0/1 constraints are more complex than the ones for monotone tasks.[2]

The cost of the multiple-version method is the overhead to store multiple versions, as well as the relatively high scheduling overhead. Scheduling tasks that have two versions is the same as scheduling tasks with the 0/1 constraint. For scheduling purposes, we can view the primary version of a task as consisting of a mandatory subtask and an optional subtask, and the alternate version as the mandatory subtask. The processing time of the mandatory subtask is the same as the processing time of the task's alternate version. The processing time of the optional subtask in the primary version is equal to the difference between the processing times of the primary and alternate versions. Thus, scheduling the primary version corresponds to scheduling the mandatory subtask and the entire optional subtask, while scheduling the alternate version corresponds to scheduling only the mandatory subtask.

To ensure that imprecise computation works properly, we need to make sure that all the mandatory subtasks have bounded resource and processing time requirements and are allocated sufficient processor time to complete by their deadlines. The system can use leftover processor time to complete as many optional subtasks as possible. For guaranteed performance and predictable behavior, we can use a conservative scheduling discipline such as the rate-monotone algorithm[6] to schedule the mandatory subtasks. To schedule optional subtasks for optimal processor use, we can use more dynamic disciplines, such as the earliest-deadline-first algorithm,[6] which may have unpredictable behavior. Because a monotone task can be terminated any time after it has produced an acceptable result, the system can decide — on line or nearly on line — how much of each optional subtask to schedule.

Basic workload model

The problems in scheduling imprecise computations are at least as complex as the corresponding classical real-time-scheduling problems. Almost all problems beyond scheduling unit-length, dependent tasks on two processors are NP-hard, and most known heuristic algorithms for multiprocessor scheduling of dependent tasks have poor worst-case performance.[7] For this reason, a better approach to scheduling dependent tasks is first to assign the tasks statically to processors and then schedule the tasks on each processor using an optimal or near-optimal uniprocessor scheduling algorithm. When the tasks are independent, optimal preemptive multiprocessor schedules can be obtained by transforming an optimal uniprocessor schedule using McNaughton's rule.[8]

All the imprecise computation models are extensions and variations of the following basic model. We have a set of preemptable tasks $\mathbf{T} = \{T_1, T_2, ..., T_n\}$. Each task T_i is characterized by parameters, which are rational numbers:

- Ready time r_i' at which T_i becomes ready for execution
- Deadline d_i' by which T_i must be completed
- Processing time τ_i, the time required to execute T_i to completion in the traditional sense on a single processor
- Weight w_i, a positive number that measures the relative importance of the task

Logically, we decompose each task T_i into two subtasks: the *mandatory* subtask M_i and the *optional* subtask O_i. Hereafter, we refer to M_i and O_i simply as tasks rather than subtasks: M_i and O_i mean the mandatory task and the optional task of T_i. We use T_i to refer to the task as a whole. The processing times of M_i and O_i are m_i and o_i, respectively. Here m_i and o_i are rational numbers, and $m_i + o_i = \tau_i$. The ready times and deadlines of the tasks M_i and O_i are the same as those of T_i.

A schedule on a uniprocessor system is an assignment of the processor to the tasks in $\mathbf{T}$ in disjoint intervals of time. A task is scheduled in a time interval if the processor is assigned to the task in the interval. A valid schedule assigns the processor to at most one task at any time, and every task is scheduled after its ready time. Moreover, the total length of the intervals in which the processor is assigned to T_i, referred to as the total amount of processor time assigned to the task, is at least equal to m_i and at most equal to τ_i. A task is completed in the traditional sense at an instant t when the total amount of processor time assigned to it becomes equal to its processing time at t.

A mandatory task M_i is completed when it is completed in the traditional sense. The optional task O_i depends on the mandatory task M_i and becomes ready for execution when M_i is completed. The system can

terminate O_i at any time; no processor time is assigned to it after it is terminated.

A task T_i is completed in a schedule whenever its mandatory task is completed. It is terminated when its optional task is terminated. Given a schedule S, we call the earliest time instant at which the processor is assigned to a task the *start time* of the task and the time instant at which the task is terminated its *finishing time*.

The traditional workload model of hard real-time applications is a special case of this model in which all the tasks are mandatory, that is, $o_i = 0$ for all i. Similarly, the traditional soft real-time workload model is also a special case in which all tasks are optional, that is, $m_i = 0$ for all i.

Precedence constraints specify the dependences between the tasks in **T**. The constraints are given by a partial-order relation "<" defined over **T**. $T_i < T_j$ if the execution of T_j cannot begin until the task T_i is completed and terminated. T_j is a *successor* of T_i, and T_i is a *predecessor* of T_j, if $T_i < T_j$. For a schedule of **T** to be valid, it must satisfy the precedence constraints between all tasks. A set of tasks is independent if the partial-order relation is empty, and the tasks can be executed in any order. A valid schedule is *feasible* if every task is completed by its deadline. A set of tasks is *schedulable* if it has at least one feasible schedule.

The given deadline of a task can be later than that of its successors. Rather than working with the given deadlines, we use modified deadlines consistent with the precedence constraints and computed as follows. The modified deadline d_i of a task T_i that has no successors is equal to its given deadline d_i'. Let A_j be the set of all successors of T_j. The modified deadline d_j of T_j is

$$\min\left\{d_j', \min_{T_i \in A_j}\{d_i\}\right\}$$

Similarly, the given ready time of a task may be earlier than that of its predecessors. We modify the ready times of tasks as follows. The modified ready time r_i of a task T_i with no predecessors is equal to its given ready time r_i'. Let B_j be the set of all predecessors of T_j. The modified ready time r_j of T_j is

$$\max\left\{r_j', \max_{T_i \in B_j}\{r_i\}\right\}$$

A feasible schedule on a uniprocessor system exists for a set of tasks **T** with the given ready times and deadlines if and only if a feasible schedule of **T** with the modified ready times and deadlines exists.[7] Working with the modified ready times and deadlines allows the precedence constraints to be ignored temporarily. If an algorithm finds an invalid schedule in which T_i is assigned a time interval later than some intervals assigned to T_j but $T_i < Tj$, it can construct a valid schedule by exchanging the time intervals assigned to T_i and T_j to satisfy their precedence constraint without violating their timing constraints. In our subsequent discussion, the terms ready times and deadlines mean *modified* ready times and deadlines. We call the time interval $[r_i, d_i]$ the *feasibility interval* of the task T_i.

When the amount of processor time σ_i assigned to O_i in a schedule is equal to o_i, the task T_i is precisely scheduled. The error ϵ_i in the result produced by T_i (or simply the error of T_i) is zero. In a precise schedule, every task is precisely scheduled. Otherwise, if σ_i is less than o_i, we say that a portion of O_i with processing time $o_i - \sigma_i$ is discarded, and the error of T_i is equal to

$$\epsilon_i = E_i(o_i - \sigma_i) \tag{1}$$

where the error function $E_i(\sigma_i)$ gives the error of the task T_i as a function of σ_i. We assume throughout this article that $E_i(\sigma_i)$ is a monotone nonincreasing function of σ_i.

Imprecise scheduling problems

Depending on the application, we use different performance metrics as criteria for comparing different imprecise schedules. Consequently, there are many different imprecise scheduling problems. We describe some in this section.

Minimization of total error. In practice, the exact behavior of error functions $E_i(x)$ is often not known. A reasonable choice is the simplest one:

$$\epsilon_i = o_i - \sigma_i \tag{2a}$$

for all i. For a given schedule, the total error of the task set **T** is

$$\epsilon = \sum_{i=1}^{n} w_i \epsilon_i \tag{2b}$$

Again, $w_i > 0$ are the weights of the tasks. Sometimes, we also refer to ϵ as the total error of the schedule. The basic imprecise scheduling problem is, given a set **T** of n tasks, to find a schedule that is optimal in that it is feasible and has the minimum total error given by Equations 2a and 2b. An optimal scheduling algorithm always finds an optimal schedule whenever feasible schedules of **T** exist. In later sections, we consider this problem for dependent tasks on uniprocessor systems or independent tasks on multiprocessor systems.

Minimization of the maximum or average error. Two straightforward variations of the minimization of total error performance metric are concerned with the average error and the maximum error. Given a schedule of the task set **T** and the errors ϵ_i of the tasks, the maximum error of the task set is

$$\max_i [w_i \epsilon_i]$$

For some applications, we may want to find feasible schedules with the smallest maximum error, rather than the total error. There are polynomial-time, optimal algorithms for solving this scheduling problem. (We are preparing a manuscript describing them.)

Minimization of the number of discarded optional tasks. In a schedule that satisfies the 0/1 constraint, σ_i is equal to o_i or 0 for every task. The general problem of scheduling to meet the 0/1 constraint and timing constraints, as well as to minimize the total error, is NP-complete when the optional tasks have arbitrary processing times. Often — for example, when scheduling tasks with multiple versions — we are concerned only with the number of discarded optional tasks. A reasonable strategy for scheduling to minimize the number of discarded optional tasks is the shortest-processing-time-first strategy, which tries to schedule the optional tasks with shorter processing times first. Given a set **T** of n tasks, N_s and N_o are the numbers of optional tasks discarded in a schedule produced using this strategy and discarded in an optimal schedule, respectively. Our conjecture is that $N_s \leq 2N_o$.

When optional subtasks have identical processing times, tasks with 0/1 constraints and identical weights can be optimally scheduled in $O(n \log n)$ time or $O(n^2)$ time, depending on whether the tasks have identical or different ready times. Optimal algorithms for this case can be found elsewhere.[2]

Minimization of the number of tardy tasks. As long as the total error of the tasks is lower than a certain acceptable limit, its value is often not important. We may then

want to minimize the number of tasks that are tardy — that is, tasks that complete and terminate after their deadlines — for a given maximum, tolerable total error. Leung and Wong[9] presented a pseudopolynomial time algorithm and a fast heuristic algorithm for preemptive uniprocessor scheduling of tasks whose feasibility intervals include each other. In the worst case, the number of tardy tasks in a schedule found by the heuristic algorithm is approximately three times the number in an optimal schedule.

Minimization of average response time. Given a schedule S and the finishing time $f(T_i, S)$ of every task, the *mean flow time* of the tasks according to the schedule is equal to

$$F = \sum_{i=1}^{n} f(T_i, S) / n$$

The mean flow time of the tasks measures the average response time, the average amount of time a task waits until it completes. The goal of scheduling is to minimize the mean flow time, subject to the constraint that the total error is less than an acceptable value. Unfortunately, all but the simplest special cases of this scheduling problem are NP-hard.[10] In a later section, we discuss the queuing-theoretical formulation, a more fruitful approach to this problem.

Scheduling to minimize total error

Two optimal algorithms for scheduling imprecise computations to meet deadlines and minimize total error use a modified version of the classical earliest-deadline-first algorithm.[7] This is a preemptive, priority-driven algorithm that assigns priorities to tasks according to their deadlines. Tasks with earlier deadlines have higher priorities. In our version, every task is terminated at its deadline even if it is not completed at the time. We call this algorithm the *ED algorithm*. Its complexity is $O(n \log n)$. In any ED algorithm schedule, every task is scheduled in its feasibility interval.

We use the ED algorithm to test whether a task set **T** can be feasibly scheduled. In the feasibility test, we schedule the mandatory set **M** using the ED algorithm. If the resultant schedule of **M** is precise, then the task set **T** can be feasibly scheduled. Otherwise, no feasible schedule of **T** exists.

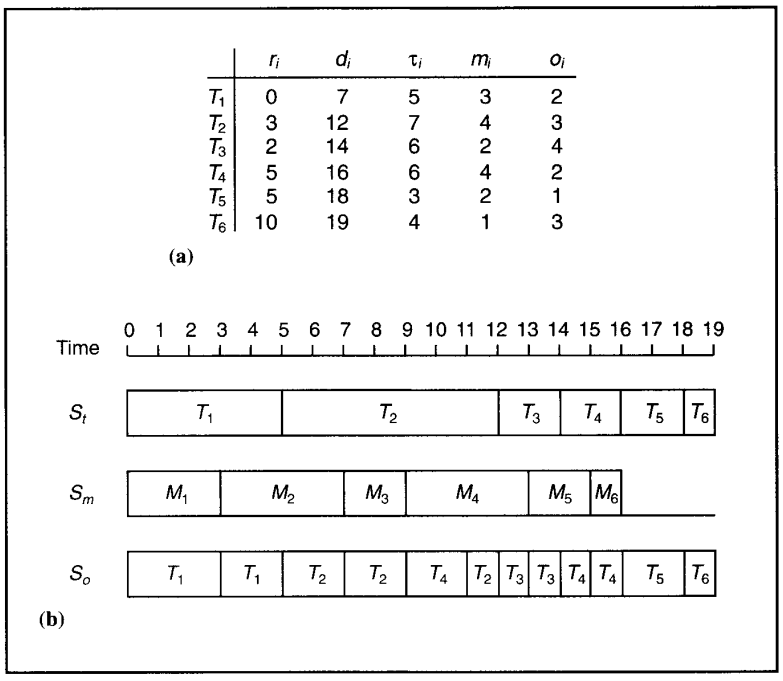


Figure 1. An example showing the need for step 3 in algorithm F.

Identical-weight case. An ED schedule of a set of entirely optional tasks is, by definition, a feasible schedule of the set. Moreover, because such a schedule is priority-driven, the processor is never left idle when there are schedulable tasks. A portion of an optional task is discarded only when necessary. Therefore, the ED algorithm is optimal when used to schedule optional tasks that have identical weights.[2]

The optimality of the ED algorithm provides the basis of algorithm F, which schedules a set $\mathbf{T} = \{T_1, T_2, ..., T_n\}$ of n tasks with identical weights on a uniprocessor system. We decompose the set **T** into two sets, the set **M** of mandatory tasks and the set **O** of optional tasks. Algorithm F works as follows:

(1) Treat all mandatory tasks in **M** as optional tasks. Use the ED algorithm to find a schedule S_t of the set **T**. If S_t is a precise schedule, stop. The resultant schedule has zero error and is, therefore, optimal. Otherwise, carry out step 2.

(2) Use the ED algorithm to find a schedule S_m of the set **M**. If S_m is not a precise schedule, **T** is not schedulable. Stop. Otherwise, carry out step 3.

(3) Using S_m as a template, transform S_t into an optimal schedule S_o that is feasible and minimizes the total error.

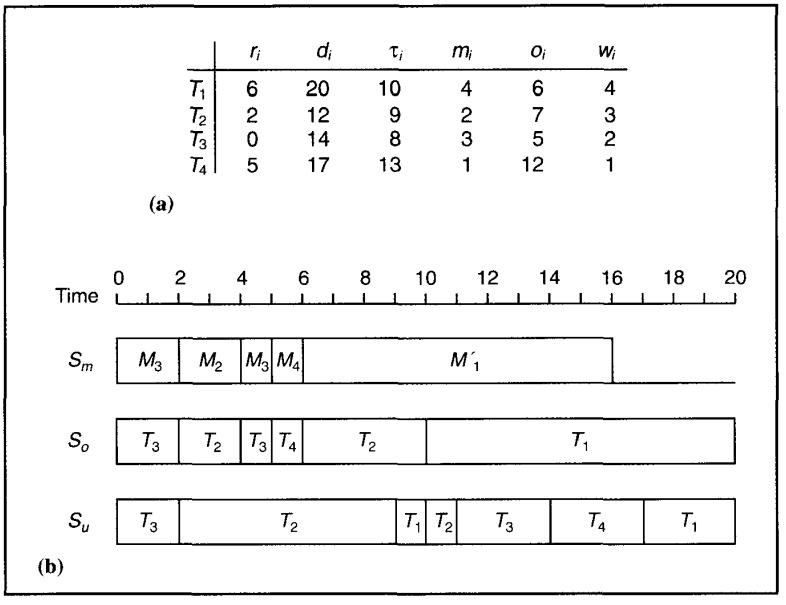
The example in Figure 1 shows the need for step 3. The task set in Figure 1a consists of six tasks of identical weights. Figure 1b shows the schedules S_t and S_m generated in steps 1 and 2, respectively. S_m of the mandatory set **M** is precise. Therefore, feasible schedules of **T** exist.

The schedule S_t is, in general, not a feasible schedule of **T**. Because step 1 treats all the tasks as entirely optional, some tasks may be assigned insufficient processor time for their mandatory tasks to complete. In this example, step 1 assigns T_4 only two units of processor time in S_t, which is less than the processing time of M_4. In step 3, we transform S_t into a feasible schedule of **T** by adjusting the amounts of processor time assigned to the tasks, so every task T_i is assigned at least m_i units of processor time in the transformed schedule.

The transformation process in step 3 has as inputs the schedules S_m and S_t. Let a_1 and a_{k+1} be, respectively, the earliest start time and the latest finishing time of all tasks in the schedule S_m. We partition the time interval $[a_1, a_{k+1}]$ according to S_m into k disjoint intervals $[a_j, a_{j+1}]$, for $j = 1, 2, ..., k$, so in S_m the processor is assigned to only one task in each of these intervals and is assigned to different tasks in adjacent intervals. In the example in Figure 1, k is equal to 6, and the time instants $a_1, a_2, ..., a_7$ are

```
Tasks are indexed so that w_1 ≥ w_2 ≥ ... ≥ w_n
begin
  Use the ED algorithm to find a schedule S_m of M.
  If S_m is not precise, stop; the task set T cannot be feasibly scheduled.
  else
    The mandatory set M' (= {M_1', M_2', ..., M_n'}) = M
    i = 1
    while (1 ≤ i ≤ n)
      Use algorithm F to find an optimal schedule S_o^i of M' ∪ {O_i};
        O_i' = the portion of O_i with processing time σ_i^o scheduled in S_o^i
      M_i' = M_i ∪ O_i'
      i = i + 1
    endwhile
    The optimal schedule sought is S_o^n
  endif
end algorithm LWF
```

Figure 2. Pseudocode for the LWF algorithm.



	r_i	d_i	τ_i	m_i	o_i	w_i
T_1	6	20	10	4	6	4
T_2	2	12	9	2	7	3
T_3	0	14	8	3	5	2
T_4	5	17	13	1	12	1

Figure 3. An illustration of the LWF algorithm.

0, 3, 7, 9, 13, 15, and 16, respectively. Let $M(j)$ denote the mandatory task scheduled in the interval $[a_j, a_{j+1}]$ in S_m, and let $T(j)$ be the corresponding task.

Step 3 adjusts the amounts of processor time assigned to the tasks in S_t, using S_m as a template. In this step, we scan the schedule S_t backward from its end a_{k+2}. The segment of S_t after a_{k+1} is left unchanged. We compare in turn, for $j = k, k-1, \ldots, 1$, the total amounts $L_t(j)$ and $L_m(j)$ of processor time assigned to the tasks $T(j)$ and $M(j)$ after a_j according to S_t and S_m, respectively. If $L_t(j) \geq L_m(j)$, the segment of S_t in $[a_j, a_{j+1}]$ is left unchanged. Otherwise, let $\Delta = L_m(j) - L_t(j)$. We assign Δ additional units of processor time in $[a_j, a_{j+1}]$ to $T(j)$. These units may be originally assigned to other tasks in S_t. Arbitrarily, we choose some of these tasks. We decrease the amounts of processor time assigned to them in this interval by a total of Δ units and update accordingly the values of $L_t(l)$ (for $l = 1, 2, \ldots, j$) for all the tasks affected by this reassignment. This reassignment can always be done because Δ is less than or equal to $a_{j+1} - a_j$ and $T(j)$ is ready in the interval.

In the example in Figure 1, $L_t(6)$ and $L_t(5)$ are left unchanged because they are equal to $L_m(6)$ and $L_m(5)$, respectively. $L_t(4)$ is 2 while $L_m(4)$ is 4; therefore, we assign two additional units of processor time to $T(4)$, which is T_4. These two units of time are taken from T_2. T_2 has three units of processor time in the interval [9, 13] before the reassignment and only one unit after the reassignment. The new values of $L_t(j)$ are 5, 5, 2, and 4 for $j = 1, 2, 3$, and 4, respectively. Similarly, we compare $L_t(3)$ and $L_m(3)$, and so on. The result of step 3 is the optimal schedule S_o.

The complexity of algorithm F is the same as that of the ED algorithm, that is, $O(n \log n)$. Algorithm F always produces a feasible schedule of **T** as long as **T** can be feasibly scheduled — this follows directly from the algorithm's definition. The schedule S_t obtained in step 1 achieves the minimum total error for any set of tasks with identical weights. Since step 3 introduces no additional idle time, the total error remains minimal. Hence, algorithm F is optimal for scheduling tasks with identical weights to minimize total error.[2] With McNaughton's rule,[8] it can be modified to optimally schedule independent tasks with identical weights on a multiprocessor system containing v identical processors. Then its complexity is $O(vn + n \log n)$.

Different-weight case. When tasks in **T** have different weights, we can number them according to their weights: $w_1 \geq w_2 \geq \ldots \geq w_n$. Algorithm F is not optimal for scheduling tasks with different weights. We use the largest-weight-first algorithm, which is optimal. Figure 2 shows the LWF algorithm in pseudocode.

Starting from the task with the largest weight, in the order of nonincreasing weights, the LWF algorithm first finds for each task T_i in turn the maximum amounts of processor time σ_i^o that can be assigned to its optional task O_i. An optimal schedule is a feasible schedule in which the amount of processor time assigned to each optional task T_i is σ_i^o.

The LWF algorithm makes use of the fact that the amount of processor time σ_i^o assigned to the only optional task in the set $\mathbf{M} \cup \{O_i\}$ in an optimal schedule of this set is as large as possible. (In Figure 2, the notation for the optimal schedule of the set is S_o^i.) It uses algorithm F to schedule the

tasks in the set $\mathbf{M} \cup \{O_1\}$. Again, T_1 is the task with the largest weight. In the resultant schedule S_o^1, the optional task O_1 is assigned the maximum possible amount of processor time σ_1^o. There are optimal schedules of $\mathbf{T}$ in which O_1 is assigned σ_1^o units of processor time.[2] We commit ourselves to finding one of these schedules by combining M_1 and the portion O_1' of O_1 that is scheduled in S_o^1 into a task M_1'. We treat the task M_1' as a mandatory task in the subsequent steps.

In the next step, we again use algorithm F to schedule the task set $\{M_1', M_2, ..., M_n\} \cup \{O_2\}$. Let O_2' be the portion of the optional task O_2 that is scheduled in the resultant optimal schedule S_o^2. O_2' has the maximum possible amount of processor time σ_2^o. There are optimal schedules of $\mathbf{T}$ in which the amounts of processor time assigned to O_1 and O_2 are σ_1^o and σ_2^o, respectively. We commit ourselves to finding one of these schedules by combining M_2 and O_2' into the mandatory task M_2'.

We repeat these steps for $i = 3, 4, ..., n$ until all σ_i^o are found. The schedule S_o^n found in the last step is an optimal schedule of $\mathbf{T}$ with minimum total error.

The time complexity of the first step when algorithm F is used is $O(n \log n)$, but in subsequent steps, algorithm F requires only $O(n)$ time. Hence, the time complexity of the LWF algorithm is $O(n^2)$.

Figure 3 shows how the LWF algorithm works. Figure 3a lists four tasks and their weights. Figure 3b shows the schedule S_m of $\mathbf{M} \cup \{O_1\}$ produced by algorithm F. We commit ourselves to finding an optimal schedule in which the amount of processor time assigned to O_1 is 6. This is an earliest-deadline-first schedule, and we use it in the second step as a template to find an optimal schedule of the task set $\{M_1', T_2, M_3, M_4\}$. Figure 3b shows the resultant schedule S_o. The total error of the tasks is 25. Also shown is a schedule S_u, which minimizes the unweighted total error.

Scheduling periodic jobs

In the well-known periodic-job model,[3,6] there is a set $\mathbf{J}$ of n periodic jobs. Each job consists of a periodic sequence of requests for the same computation. The period π_i of a job J_i in $\mathbf{J}$ is the time interval between two consecutive requests in the job. In terms of our basic model, each request in job J_i is a task whose processing time is τ_i. The ready time and deadline of the task in each period are the beginning and the end of the period, respectively. Therefore, we can specify each job J_i by the two-tuple (π_i, τ_i). In the extended-workload model used to characterize periodic imprecise computations,[3] each task in J_i consists of a mandatory task with processing time m_i and an optional task with processing time $\tau_i - m_i$. The optional task is dependent on the mandatory task. In other words, we decompose each job $J_i = (\pi_i, \tau_i)$ into two periodic jobs: the mandatory job $M_i = (\pi_i, m_i)$ and the optional job $O_i = (\pi_i, \tau_i - m_i)$. The corresponding sets of mandatory jobs and optional jobs are denoted by $\mathbf{M}$ and $\mathbf{O}$, respectively. Let

$$U = \sum_{i=1}^{n} \tau_i / \pi_i$$

denote the *utilization factor* of the job set $\mathbf{J}$. U is the fraction of processor time required to complete all the tasks in $\mathbf{J}$ if every task is completed in the traditional sense. Similarly, let

$$u = \sum_{k=1}^{n} m_k / \pi_k$$

Here, u is the utilization factor of the mandatory set $\mathbf{M}$.

Because the worst-case performance of priority-driven strategies for scheduling periodic jobs on multiprocessor systems is unacceptably poor, it is common to assign jobs once and for all to processors, and schedule the jobs assigned to each processor independently of the jobs assigned to the other processors. We can formulate the problem of finding an optimal assignment of jobs to processors and using a minimum number of processors as a bin-packing problem.

A heuristic job-assignment algorithm with reasonably good worst-case performance is the rate-monotone next-fit (or first-fit) algorithm. According to this algorithm, jobs in $\mathbf{J}$ are sorted in the order of nonincreasing rates and are assigned to the processors on the next-fit (or first-fit) basis. A job fits on a processor if it and the jobs already assigned to the processor can be feasibly scheduled according to the rate-monotone algorithm.[6] (The rate-monotone algorithm is a preemptive, priority-driven algorithm that assigns priorities to jobs according to their rates: the higher the rate, the higher the priority.) When deciding whether an imprecise job fits on a processor, the algorithm considers only the mandatory set $\mathbf{M}$. Let $u_i = m_i/\pi_i$. Suppose that k jobs are already assigned to a processor, and their mandatory jobs have a total utilization factor

$$u = \sum_{i=1}^{k} u_i$$

If an additional job J_{k+1} is also assigned to this processor, the total utilization factor of the $k+1$ mandatory jobs is $u + m_{k+1}/\pi_{k+1}$. J_{k+1} is assigned to the processor only if

$$u + m_{k+1}/\pi_{k+1} \le (k+1)(2^{1/(k+1)} - 1) \quad \textbf{(3)}$$

Hereafter, $\mathbf{J}$ denotes the set of n jobs assigned to one processor in this manner. The utilization factor U of the job set $\mathbf{J}$ may be larger than $n(2^{1/n} - 1)$. Consequently, $\mathbf{J}$ may not be precisely schedulable to meet all deadlines according to the rate-monotone algorithm. However, the utilization factor u of the mandatory set is less than $n(2^{1/n} - 1)$. Hence, the mandatory set $\mathbf{M}$ is always precisely schedulable.[6] Since the value of u is less than 1 (for example, 0.82 for $n = 2$ and ln 2 for large n), a fraction of processor time is always available to execute tasks in the optional set $\mathbf{O}$.

Depending on the kind of undesirable effect that errors cause, applications can be classified as either *error-noncumulative* or *error-cumulative*. Different performance metrics are appropriate for each.

For an error-noncumulative application, only the average effect of errors in different periods is observable and relevant. Optional tasks are truly optional because none need to be completed. Image enhancement and speech processing are examples of this type of application.

In contrast, for an error-cumulative application, errors in different periods have a cumulative effect. The optional task in one period among several consecutive periods must be completed within that period and, hence, is no longer optional. Tracking and control are examples of this type of application.

The complexity of scheduling error-cumulative jobs and an approximate algorithm for scheduling them with identical periods have been discussed elsewhere.[3] Much work remains in finding effective algorithms and schedulability criteria for scheduling error-cumulative jobs that have arbitrary periods and error characteristics. The workload on practical systems typically is a mixture of error-cumulative jobs, error-noncumulative jobs, aperiodic jobs, and dependent jobs. Programmers need good heuristic algorithms for scheduling such complex job mixes.

Error-noncumulative jobs. Now, we focus on error-noncumulative jobs. Since each periodic job can be viewed as an

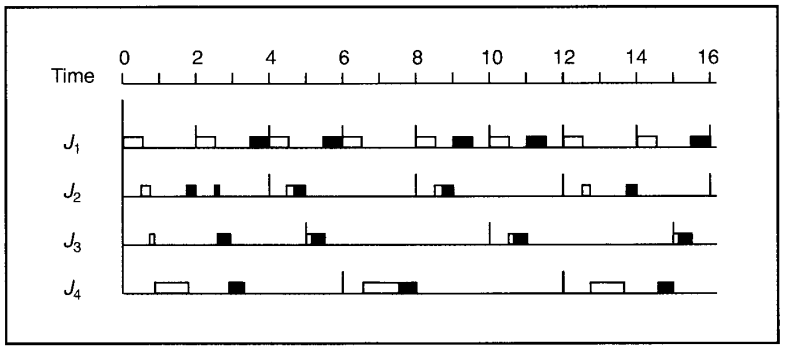


Figure 4. Scheduling error-noncumulative jobs.

infinite chain of tasks, the total error defined in Equations 2a and 2b is infinite for an imprecise schedule. A more appropriate performance metric of the overall result quality is the average error in the results produced in several consecutive periods. While the duration over which the average error is computed can be arbitrary, a convenient choice of this duration is π, the least common multiple of all the periods in **J**. For this duration, the average error ϵ_i of J_i at any time is the average value of $E_i(o_i - \sigma_{i,j})$ over the past π/π_i periods, where $\sigma_{i,j}$ is the amount of processor time assigned to the task in the jth period of J_i. Again, the error function $E_i(\sigma_{i,j})$ is a nonincreasing function of $\sigma_{i,j}$. The average error over all jobs in **J** is

$$\epsilon = \sum_{i=1}^{n} w_i \epsilon_i$$

where w_i is a nonnegative constant weight and

$$\sum_{i=1}^{n} w_i = 1$$

A class of heuristic algorithms for scheduling error-noncumulative periodic jobs on uniprocessors to minimize the average error has been designed and evaluated.[3] All the algorithms in this class are preemptive and priority-driven, and all use the same strategy: They execute optional tasks only after all the ready mandatory tasks have completed. Specifically, given a job set **J** and its associated mandatory set **M** and optional set **O**, all the jobs in **M** have higher priorities than all the jobs in **O**. Moreover, the rate-monotone algorithm schedules jobs in **M** precisely. Because of the condition given by Equation 3, the set **M** can always be feasibly scheduled.[6] Such algorithms meet all the deadlines, regardless of how jobs in **O** are scheduled.

Figure 4 shows an example in which the job set **J** consists of four jobs. They are (2, 1), (4, 0.5), (5, 0.5), and (6, 1.5), and their mandatory tasks have processing times 0.5, 0.2, 0.1, and 1.0, respectively. The utilization factor of the job set **J** is 0.975. **J** is not precisely schedulable according to the rate-monotone algorithm. In a rate-monotone schedule, the task in the first period of J_4 misses its deadline. However, the mandatory set **M** consists of (2, 0.5), (4, 0.2), (5, 0.1), and (6, 1.0) with a utilization factor 0.4867. It is precisely schedulable according to the rate-monotone algorithm.

White boxes in the timing diagram in Figure 4 show the time intervals when the processor is assigned to tasks in **M** in a rate-monotone schedule of **M**. Black bars indicate the time intervals during which the processor is assigned to jobs in the optional set **O**, consisting of (2, 0.5), (4, 0.3), (5, 0.4), and (6, 0.5).

Types of heuristic algorithms. The heuristic algorithms[3] differ only in how they assign priorities to optional jobs. Some make priority assignments to optional tasks on the basis of error function behavior. Examples include the least-utilization algorithm, the least-attained-time algorithm, and the first-come-first-serve algorithm.

The *least-utilization algorithm* statically assigns higher priorities to optional jobs with smaller weighted utilization factors: $(\tau_i - m_i)/\pi_i w_i$. It minimizes the average error when the error functions $E_i(x)$ are linear and when all jobs have identical periods and weights.

At any time, the *least-attained-time algorithm* assigns the highest priority to the optional task that has attained the least processor time among all the ready optional tasks. This algorithm tends to perform well when the error functions $E_i(x)$ are convex, that is, when the error in the result decreases faster earlier and more slowly later, as the computation proceeds.

The *first-come-first-serve algorithm* performs well when the error functions $E_i(x)$ are concave, that is, when the underlying procedure converges more slowly earlier and faster later, as the computation proceeds.

When we do not know the exact behavior of the error function, we can use an algorithm that ignores the error functions in assigning priorities to optional tasks. The *shortest-period algorithm*, which also assigns priorities to optional jobs on a rate-monotone basis, is such an algorithm.

Another example is the *earliest-deadline algorithm*. This algorithm assigns priorities dynamically to optional tasks according to their deadlines: the earlier the deadline, the higher the priority. If any of the heuristic algorithms we have described here can precisely schedule a set of jobs, the earliest-deadline algorithm can precisely schedule it, too.[3]

Quantitative data on achievable average errors with the algorithms described above for different values of the utilization factors of **M** and **J** are available elsewhere.[3] These algorithms have the advantage of the rate-monotone algorithm: Tasks miss deadlines in a predictable manner during a transient overload. Like the classical earliest-deadline-first algorithm, these algorithms also use the processor to its fullest extent. They are ideal for applications where transient overloads occur frequently or actual task processing times vary widely. Usually the average error remains tolerably small when U becomes larger than 1 and no classical algorithm can schedule the tasks satisfactorily.

The advantages are realized at the expense of not being optimal. For example, these algorithms may lead to schedules with a nonzero average error for job sets that can be precisely scheduled to meet deadlines by the classical rate-monotone or earliest-deadline-first algorithms.

Scheduling parallelizable tasks

A task is parallelizable if it can be executed in parallel on a number of processors to complete in less time. The degree of concurrency of any task in an interval refers to the number of processors on which the task executes in parallel in the interval. In our model of parallelizable tasks, the

degree of concurrency of any task may change during its execution.

The parameters that characterize each parallelizable task T_i in a set **T** of n tasks include its ready time r_i, deadline d_i, processing time τ_i, and weight w_i — in short, the parameters that characterize any sequential task. A parallelizable task also has the following two parameters:

- Maximum degree of concurrency C_i, the number of processors on which T_i can execute in parallel
- Multiprocessing overhead factor θ_i, a proportional constant used to compute the overhead in the parallel execution of T_i

Like sequential tasks, each parallelizable task T_i is logically composed of a mandatory task M_i and an optional task O_i, whose processing times on a single processor are m_i and o_i, respectively. In this section we use task to refer to a parallelizable task. We consider only cases where the tasks are independent.

A parallel schedule of the task set **T** on a system containing v identical processors assigns no more than one task to any processor at any time and assigns each task T_i to at most C_i processors. For a given a task set **T**, let $\mathbf{a} = \{a_1, a_2, \ldots, a_{k+1}\}$ be an increasing sequence of distinct numbers obtained by sorting the list of ready times and deadlines of all the tasks in **T** and deleting duplicate entries in the list. (Here $k + 1 \leq 2n$.) This sequence divides the time between the earliest ready time a_1 and latest deadline a_{k+1} into k intervals $I_j = [a_j, a_{j+1}]$ for $j = 1, 2, \ldots, k$.

We divide the problem of finding feasible parallel schedules of **T** into two subproblems: the time allocation problem and the schedule construction problem. To solve the time allocation problem, we decide how many units of processor time in each of the k intervals I_j should be allocated to each task T_i, so the tasks meet their timing constraints and the total error is minimized. Given this solution, we then solve the second problem to obtain a parallel schedule on v processors.

Time allocation problem. When a system executes a task in parallel on more than one processor, it wastes some processor time in interprocessor communication and synchronization. This multiprocessing overhead Θ_i of a task T_i in any time interval depends on the task's degree of concurrency c_i and, consequently, on the amount of processor time allocated to the task in the interval.

$$\text{minimize} \sum_{i=1}^{n} w_i \{\tau_i - \sum_{j=1}^{k} [x_i(j) - \theta_i(j)]\}$$

$$\sum_{j=1}^{k} x_i(j) \leq \tau_i + \sum_{j=1}^{k} \theta_i(j) \quad i = 1, 2, \ldots, n$$

$$\sum_{j=1}^{k} x_i(j) \geq m_j + \sum_{j=1}^{k} \theta_i(j) \quad i = 1, 2, \ldots, n$$

$$v t_j \geq \sum_{i=1}^{n} x_i(j) \quad j = 1, 2, \ldots, k$$

$$(C_i - 1) \geq Y(x_i(j)) \geq 0 \quad i = 1, 2, \ldots, n$$

$$x_i(j) \geq 0 \quad i = 1, 2, \ldots, n; \; j = 1, 2, \ldots, k$$

Figure 5. Linear programming formulation.

Studies on scheduling parallelizable (precise) tasks typically assume that for c_i larger than 1, Θ_i is either a positive constant or a monotone nondecreasing function of c_i. We present a special case where the multiprocessing overhead is a linear function of the degree of concurrency. This assumption allows us to formulate the time allocation problem as a linear programming problem and solve it using any of the well-known techniques.[11]

To calculate the multiprocessing overhead, we suppose that a task T_i is allocated a total of x units of processor time on all processors in an interval of length t. If $x \leq t$, this task is not parallelized in this interval, and its multiprocessing overhead in this interval is zero. If $x > t$, the minimum degree of concurrency of the task in this interval is $\lceil x / t \rceil$. Rather than make the multiprocessing overhead proportional to this nonlinear function of x, we let the multiprocessing overhead of T_i in this interval be proportional to $Y(x) = \max(x/t - 1, 0)$. Then $\theta_i Y(x)$ units of processor time is wasted as the multiprocessing overhead.

The actual amount of processor time available to the task in this interval for its execution toward completion is $x - \theta_i Y(x)$. We say that this amount of processor time is actually assigned to the task T_i. Again, Equations 2a and 2b give us the error ϵ_i of a task T_i in terms of the amount σ_i of processor time actually assigned to its optional task O_i in all k intervals.

Let t_j denote the length of the interval I_j, and $x_i(j)$ denote the amount of processor time allocated to the task T_i in I_j. Here $x_i(j)$ is zero if the feasibility interval of T_i does not include the interval I_j. Let

$$\Theta_i(j) = \theta_i Y(x_i(j)) = \theta_i \max(x_i(j)/t - 1, 0)$$

be the multiprocessing overhead of T_i incurred in this interval when its allocated processor time is $x_i(j)$.

Figure 5 gives the linear programming formulation of the processor time allocation problem. We want to find the set $\{x_i(j)\}$ that minimizes the objective function, the total (weighted) error expressed here in terms of $x_i(j)$. The first set of constraints specifies that the total processor time allocated to every task in all k intervals is no more than its processing time τ_i plus its total multiprocessing overhead. These constraints ensure that no task gets more processor time than its processing time. The second set of constraints specifies that the total processor time allocated to every task T_i in all k intervals is no less than the sum of the processing time m_i of the mandatory task M_i and the total multiprocessing overhead. These constraints ensure that the schedule assigns sufficient processor time to every mandatory task for it to complete in the traditional sense. Together, they ensure that we can construct a valid schedule from the resultant set $\mathbf{X} = \{x_i(j)\}$ of processor time allocations.

The third set of constraints requires that the total processor time allocated to all tasks in every interval I_j is no greater than the total amount of processor time available on all v processors. The fourth set of constraints ensures that the degree of concurrency of each task is at most equal to its maximum degree of concurrency. The fifth set states that every $x_i(j)$ is nonnegative.

The optimal solution of this linear program, if one exists, gives a set **X** of processor time allocations from which we can construct a feasible parallel schedule of **T** with the minimum total error. The complexity of the processor time allocation problem is the same as the complexity of the most efficient algorithm for linear programming.[11] One efficient algorithm for linear programming requires $O((\alpha + \beta)\beta^2 + (\alpha + \beta)^{1.5}\beta)$ operations, where α is the number of inequalities and β is the number of variables. For our problem, α is equal to $3n + k$, and β is at most equal to nk.

In the example in Figure 6, there are three tasks; Figure 6a lists their parameters. Their ready times and deadlines divide the time between 0 to 14 into four intervals beginning at 0, 4, 6, and 12. The values of t_j for $j = 1, 2, 3$, and 4 are 4, 2, 6, and 2, respectively. Figure 6b shows the solution of the corresponding linear pro-

	r_i	d_i	τ_i	m_i	o_i	w_i	C_i	θ_i
T_1	0	6	8	3	5	3	2	2
T_2	4	12	13	10	3	2	2	2
T_3	0	14	17	7	10	1	2	2

(a)

	$x_i(1)$	$x_i(2)$	$x_i(3)$	$x_i(4)$	$\sum x_i(j)$	Y_1	Y_2	Y_3	Y_4	$\sum Y_i$	$w_i \in_i$
a_i	0	4	6	12		0	4	6	12		
T_1	6	2			8	0.5	0			0.5	3
T_2		2	9		11		0	0.5		0.5	6
T_3	2	0	3	2	7	0	0	0	0	0	10

(b)

Figure 6. An example of processor time allocation.

gram. A blank entry at a row T_i and a column $x_i(j)$ indicates that the feasibility interval of T_i does not include the interval $[a_j, a_{j+1}]$. To save space in the tabulation, Figure 6b lists $Y(x_i(j))$ simply as Y_j. The total error of the feasible schedule is 19.

Schedule construction. The solution of the linear program is the set $\mathbf{X} = \{x_i(j)\}$ of processor time allocations, which gives us the amounts of processor time allocated to the n tasks in $\mathbf{T}$ in each time interval I_j. Given $\mathbf{X}$, we need to decide which task is to run on which processor(s) in each interval I_j, so we can construct a parallel schedule. A straightforward approach is first to construct independently a segment of the parallel schedule in each interval I_j on the basis of the processor time allocations $x_i(j)$ for the interval. After constructing the schedule segments in all k intervals, we rearrange the order in which tasks are assigned in adjacent segments to reduce the total number of preemptions and migrations. If a task is scheduled in two adjacent segments on two different processors, in this rearrangement step we move them whenever possible so they are scheduled contiguously on the same processor(s) in these segments. To do this, we can use an $O(n^2 \log n)$ algorithm based on a solution of the bipartite matching problem.

Returning to how to construct a parallel schedule segment from the processor time allocations of an interval I_j, we consider now the first interval I_1. Segments in the other intervals can be constructed in the same manner. Without loss of generality, let T_1, T_2, ... T_l be all the tasks allocated nonzero processor time in this interval. The portion of each task T_i assigned in I_1 is divided into $\nu_i = \lfloor x_i(1)/t_1 \rfloor$ subtasks with processing time t_1 and a fractional subtask with processing time $\psi_i = t_1 - \lfloor x_i(1)/t_1 \rfloor t_1$. After all the subtasks with processing time t_1 are assigned on

$$\sum_{i=1}^{l} \nu_i$$

processors, we try to pack the l fractional subtasks on the remaining processors. We can use a pseudopolynomial algorithm for this knapsack problem.

Queuing-theoretical formulation

A performance metric common in many applications is the *average response time* of tasks, that is, the average amount of time between the instant when a task is ready and the instant at which the task is completed (and leaves the system). The section on imprecise scheduling briefly describes the deterministic formulation of the problem: Find optimal schedules with the minimum average response time, subject to the constraint of a maximum acceptable total error. This problem is NP-hard[10] for most practical cases, so the queuing-theoretical approach is more fruitful than the deterministic formulation. Here we briefly describe two queuing-theoretical formulations.

The simplest model of an imprecise multiprocessor system with v identical processors is an open v-server Markov queue. Tasks arrive and join a common queue according to a Poisson process with an average rate of λ. They are serviced (that is, executed) on a first-come-first-serve basis. The processing times (that is, service times) of all tasks are exponentially distributed. (Later, we say more about this assumption.) This simple Markov multiserver queue is analytically tractable. For most practical cases, however, it models imprecise computation systems in sufficient detail to provide the performance data we need to choose design parameters of imprecise service disciplines.

First, we consider imprecise computations implemented by providing two versions of each task. A task is serviced at the full level when its primary version is scheduled and executed, or at reduced level when its alternate version is scheduled and executed. When the system has a light load and the response time is small, it services every task at the full level. When the load becomes heavy, the system reduces the overall response time by servicing some tasks at the reduced level. Such a scheduling scheme is called a *two-level scheduling discipline*.[12] The full-level processing times of all tasks are statistical-

ly independent, exponentially distributed with a mean of $1/\mu$. The reduced-level processing time of a task is a constant fraction γ of its full-level processing time, where γ is a real number between 0 and 1. Let $\rho = \lambda/\nu\mu$ denote the *offered load* of a processor in the system, that is, the fraction of time each processor is busy if all tasks are serviced at their full level. It is easy to see that the system is not saturated as long as $\rho < 1/\gamma$.

Here γ is a design parameter of an imprecise computation system. We assume that the larger γ is, the better the result quality of the tasks that are serviced at the reduced level. A design parameter of the two-level service discipline is the threshold H: As long as the number of tasks in the system is less than H, the system load is light. For a given γ, we choose the value of H to achieve a desired trade-off between the average waiting time and the average result quality. The trade-off reduces the average waiting time W of tasks — the average time a task spends in the queue before its execution begins.

Thus W plus the average processing time of the tasks corresponds to the mean flow time F defined in the section on imprecise scheduling problems. We can minimize it easily by servicing every task at the reduced level. Therefore, we must consider the cost of this trade-off. Studies on the two-level scheduling discipline[12] measure this cost in terms of the average result quality, the average fraction G of tasks serviced at the full level. G measures the system's quality of service. In the steady state, $G = (U - \gamma\rho)/(1 - \gamma)\rho$, where U is the average utilization of each processor.

We choose H on the basis of the performance data on W and G. Such data on two-level scheduling in uniprocessor systems are available elsewhere.[12] More recently, we evaluated the performance of imprecise multiprocessor systems. The results indicate that an appropriate choice of H makes an imprecise system with a two-level scheduling discipline perform almost as well as the corresponding precise system in terms of the average service quality, when the offered load of the system is small. When the offered load is high, the two-level scheduling scheme can significantly improve the average task waiting time by sacrificing service quality. This trade-off is most effective when the offered load per processor is near 1. While the average waiting time in a precise system approaches infinity, the two-level scheduling scheme keeps the average waiting time in an imprecise system small, with a reasonably small decrease in the average service quality.

An imprecise computation system that uses monotone tasks is more appropriately modeled as an open $M/E_{K+1}/\nu$ queue. Each task T_i is composed of a mandatory task M_i followed by K optional tasks $O_{i,j}$. (In the deterministic models discussed in earlier sections, K is at most equal to 1.) Let $o_{i,j}$ denote the processing time of $O_{i,j}$. The processing time τ_i of the task T_i is given by

$$\tau_i = m_i + \sum_{j=1}^{K} o_{i,j}$$

The processing times m_i of the mandatory tasks, as well as the $o_{i,j}$, are all statistically independent and exponentially distributed random variables.

When a monotone imprecise system is overloaded, it may discard a number x ($x \leq K$) of optional tasks in some task or tasks in the system. The decrease in result quality can be quantified in part by the fraction of optional tasks that the system discards. The expected value of this fraction gives a rough measure of the average error ϵ in the task results.

Since the system can discard a variable number of optional tasks in each task, the average error does not give us a complete picture of the incurred cost. Another cost function is the imprecision probability, the probability of any task being imprecise because the system discarded some of its optional tasks. We are studying the dependence of these cost factors on parameters K, x, and H of the monotone imprecise system.

A direction of our future study concerns the way a system determines the kind of service each task receives. Past studies on two-level scheduling disciplines assume that the system checks the number of its tasks at each instant immediately before a processor begins to execute a task.[12] The system services the task at the head of the queue at full level if its load is light, and at the reduced level otherwise. In other words, the system is reasonably responsive to overload conditions.

Similarly, we have proposed that in monotone imprecise systems, the system could check the total number of tasks in the queue at each instant when a new task arrives and immediately before it begins to service a task. As long as the queue length is equal to or greater than H, the system discards x optional tasks in the tasks being served. This scheme, called the *responsive service scheme*, is highly responsive to overloads: The system does very well in reducing its backlog and clearing up the overload whenever such a condition occurs. However, it cannot guarantee service quality. A task that arrives when the system is lightly loaded may have its optional tasks discarded if the system becomes overloaded during the time the task waits in the system.

With the *guaranteed-service scheme*, on the other hand, the system checks its total number of tasks at each arrival instant. It tags for reduced service a task arriving when H or more tasks are in the queue. The tasks already in the system before the overload are fully serviced to completion. With this scheme, an imprecise system does not respond as quickly as possible to correct an overload. However, the quality of results produced by tasks arriving to the system when it is not overloaded is guaranteed to be good. This scheme is good for applications in which overloads can be cleared quickly.

We have reviewed different approaches for scheduling imprecise computations in hard real-time environments. We also described several imprecise computation models that explicitly quantify the costs and benefits in the trade-off between result quality and computation time requirements. An imprecise computation scheduler must balance the benefit in enhanced system response with the cost in reduced result quality.

Because the criteria for measuring costs and benefits vary according to application, there are many different imprecise scheduling problems. We have presented our recent progress toward solving some of these problems, and the directions we plan to take in our future work on each of these problems. ■

Acknowledgments

We thank Susan Vrbsky for her comments and suggestions. This work was partially supported by US Navy Office of Naval Research contracts NVY N00014 87-K-0827 and NVY N00014 89-J-1181.

References

1. K.-J. Lin and S. Natarajan, "Expressing and Maintaining Timing Constraints in Flex," *Proc. Ninth IEEE Real-Time Systems Symp.*, IEEE CS Press, Los Alamitos, Calif., Order No. 894, 1988, pp. 96-105.

2. W.K. Shih, J.W.S. Liu, and J.Y. Chung, "Algorithms for Scheduling Imprecise Computations with Timing Constraints," to be published in *SIAM J. Computing*, July 1991.

3. J.Y. Chung, J.W.S. Liu, and K.-J. Lin, "Scheduling Periodic Jobs That Allow Imprecise Results," *IEEE Trans. Computers*, Vol. 19, No. 9, Sept. 1990, pp. 1,156-1,173.

4. S. Vrbsky and J.W.S. Liu, "An Object-Oriented Query Processor That Returns Monotonically Improving Answers," *Proc. Seventh IEEE Int'l Conf. Data Eng.*, IEEE CS Press, Los Alamitos, Calif., Order No. 2138, 1991.

5. K. Kenny and K.J. Lin, "Structuring Real-Time Systems with Performance Polymorphism," *Proc. 11th IEEE Real-Time Systems Symp.*, IEEE CS Press, Los Alamitos, Calif., Order No. 2112, 1990, pp. 238-246.

6. C.L. Liu and J.W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *J. ACM*, Vol. 20, No. 1, Jan. 1973, pp. 46-61.

7. E.L. Lawler et al., "Sequencing and Scheduling: Algorithms and Complexity," tech. report, Centre for Mathematics and Computer Science, Amsterdam, 1989.

8. R. McNaughton, "Scheduling with Deadlines and Loss Functions," *Management Science*, Vol. 12, No. 1, Oct. 1959, pp. 1-12.

9. J.Y.-T. Leung and C.S. Wong, "Minimizing the Number of Late Tasks with Error Constraints," *Proc. 11th IEEE Real-Time Systems Symp.*, IEEE CS Press, Los Alamitos, Calif., Order No. 2112, 1990, pp. 32-40.

10. J.Y.-T. Leung et al., "Minimizing Mean Flow Time with Error Constraints," *Proc. 10th IEEE Real-Time Systems Symp.*, IEEE CS Press, Los Alamitos, Calif., Order No. 2004, 1989, pp. 2-11.

11. N. Karmarker, "A New Polynomial-Time Algorithm for Linear Programming," *Combinatorica*, Vol. 4, No. 4, 1984, pp. 373-395.

12. E.K.P. Chong and W. Zhao, "Task Scheduling for Imprecise Computer Systems with User Controlled Optimization," *Computing and Information*, Elsevier Science Publishers, North Holland, 1989.

Readers many write to Jane W.S. Liu, Department of Computer Science, University of Illinois at Urbana-Champaign, 1304 W. Springfield Ave., Urbana, IL 61801.

Jane W.S. Liu is a professor of computer science and of electrical and computer engineering at the University of Illinois at Urbana-Champaign. Her research interests include real-time systems, distributed systems, and computer networks.

Liu received a BS in electrical engineering from Cleveland State University. She received her masters of science and electrical engineering degrees and her doctor of science degree from MIT. She is a member of the IEEE Computer Society and the ACM, and chairs the Computer Society Technical Committee on Distributed Processing.

Kwei-Jay Lin is an assistant professor in the Department of Computer Science at the University of Illinois at Urbana-Champaign. His research interests include real-time systems, distributed systems, programming languages, and fault-tolerant systems. He was program chair for the Seventh IEEE Workshop on Real-Time Operating Systems and Software in May 1990.

Lin received his BS in electrical engineering from the National Taiwan University in 1976, and his MS and PhD in computer science from the University of Maryland in 1980 and 1985. He is a member of the IEEE Computer Society.

Wei-Kuan Shih is a PhD student in computer science at the University of Illinois at Urbana-Champaign, where his research interests include real-time systems, scheduling theory, and VLSI design automation. From 1986 to 1988, he was with the Institute of Information Science, Academia Sinica, Taiwan.

Shih received his BS and MS in computer science from the National Taiwan University.

Albert Chuang-shi Yu is a doctoral candidate in the Department of Computer Science at the University of Illinois at Urbana-Champaign. His research interests include all aspects of real-time systems, parallel processing, and artificial intelligence. He is currently supported by the NASA graduate student researcher program.

Yu received a BA in computer science and physics from the University of California, Berkeley, and an MS in computer science from the University of Illinois. He is a member of the IEEE Computer Society and Sigma Xi.

Jen-Yao Chung is a research staff member at the IBM T.J. Watson Research Center. His research interests include job scheduling and load balancing in hard real-time system, object-oriented programming environments, and operating system design.

Chung received his BS in computer science and information engineering from the National Taiwan University, and his MS and PhD in computer science from the University of Illinois at Urbana-Champaign. He is a member of the IEEE Computer Society, IEEE, ACM, Tau Beta Pi, Sigma Xi, and Phi Kappa Phi.

Wei Zhao is an associate professor in the Department of Computer Science at Texas A&M University. His research interests include distributed real-time systems, concurrency control for database systems, and resource management in operating systems and knowledge-based systems. He was a guest editor for a special issue of *Operating System Review* on real-time operating systems.

Zhao received his diploma in physics from Shaanxi Normal University, Xian, China, and his MS and PhD in computer science from the University of Massachusetts, Amherst. He is a member of the IEEE Computer Society and ACM.