A Fault-Tolerant Scheduling Algorithm for Real-Time Periodic Tasks with Possible Software Faults

Ching-Chih Han, *Member*, *IEEE*, Kang G. Shin, *Fellow*, *IEEE*, and Jian Wu, *Student Member*, *IEEE*

Abstract—A hard real-time system is usually subject to stringent reliability and timing constraints since failure to produce correct results in a timely manner may lead to a disaster. One way to avoid missing deadlines is to trade the quality of computation results for timeliness and software fault tolerance is often achieved with the use of redundant programs. A deadline mechanism which combines these two methods is proposed to provide software fault tolerance in hard real-time periodic task systems. Specifically, we consider the problem of scheduling a set of real-time periodic tasks each of which has two versions: *primary* and *alternate*. The primary version contains more functions (thus more complex) and produces good quality results, but its correctness is more difficult to verify because of its high level of complexity and resource usage. By contrast, the alternate version contains only the minimum required functions (thus simpler) and produces less precise, but acceptable results and its correctness is easy to verify. We propose a scheduling algorithm which 1) guarantees either the primary or alternate version of each critical task to be completed in time and 2) attempts to complete as many primaries as possible. Our basic algorithm uses a fixed priority-driven preemptive scheduling scheme to preallocate time intervals to the alternates and, at runtime, attempts to execute primaries first. An alternate will be executed only 1) if its primary fails due to lack of time or manifestation of bugs or 2) when the latest time to start execution of the alternate without missing the corresponding task deadline is reached. This algorithm is shown to be effective and easy to implement. This algorithm is enhanced further to prevent early failures in executing primaries from triggering failures in the subsequent job executions, thus improving efficiency of processor usage.

Index Terms—Real-time systems, deadline mechanisms, notification time, primary, alternate, backwards-RM algorithm, CAT algorithm, EIT algorithm.

1 Introduction

In some hard real-time systems, such as computer-integrated manufacturing and industrial process control, a number of tasks are periodically invoked and executed in order to collectively accomplish a common mission/function and each of them must be completed by a certain deadline. Failure to complete such a task in time may lead to a serious accident. In case both the timing and computation-quality constraints cannot be met, one way of meeting the timing constraints is to trade computation quality for timeliness. For example, Lin et al. [1] developed a concept, called *imprecise computation*, to deal with time-constrained iterative calculations. The main idea of imprecise computation is that one may meet the deadline of a task by executing a version of the task that is faster, but inferior in some other respects, such as the resources it consumes or the precision of the results it generates. A variation of imprecise computation, called *performance polymorphism*, is to use two or more means to carry out a computation [2].

In addition to the timeliness requirement, two other important requirements of real-time systems are *predictability* and *reliability* [3]. For example, the reliability requirement for commercial transport aircraft is specified in terms of the allowable probability of failure per mission and a figure of 10^{-9} has been specified by NASA for commercial aircraft for a 10-hour flight [4]. In order to meet these stringent requirements, the system must complete its tasks in time even in the presence of program (software) execution failures (due to software bugs). Campbell et al. proposed a *deadline mechanism* to provide the required fault tolerance in real-time software systems [5], [6]. In the deadline mechanism, two versions of programs are provided for each real-time task: *primary* and *alternate*. The primary version contains more functions (thus more complex) and produces good quality results, but its execution is more prone to failure because of its high level of complexity and resource usage. The alternate version, on the other hand, contains only the minimum required functions (thus simpler) and produces less precise but acceptable results. Since it is simpler and requires less resources, its correctness (thus reliability) is assumed to have been fully tested a priori and no failure occurs due to software bugs during its execution.[1]

• *The authors are with the Real-Time Computing Laboratory, Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109-2122.*
E-mail: {cchan, kgshin, wujz}@eecs.umich.edu.

Manuscript received 3 Oct. 2000; revised 21 Jan. 2002; accepted 22 Apr. 2002.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 112937.

1. The execution of an alternate may fail due to a processor failure, but such failures can be handled by using redundant hardware which is *not* the subject of this paper.

In this paper, we address the problem of scheduling real-time periodic tasks by using the deadline mechanism for providing software fault tolerance. The objective of our scheduling algorithm is to guarantee either the primary or the alternate version of each task to be correctly completed before the corresponding deadline while trying to complete as many primaries as possible. However, if the primary of a task fails (due to manifestation of a bug) during its execution or if its successful completion cannot be guaranteed (due to insufficient processor time), we must activate the alternate of the task. Since the invocation behavior of a set of periodic tasks repeats itself once every T time units, where T, called the *planning cycle* of the task set, is the least common multiple of the periods of all periodic tasks, we only need to consider all the task invocations in a planning cycle.

Many researchers investigated the issues of scheduling a set of periodic tasks [7], [8], [9], [10], [11], [12] and scheduling both periodic and aperiodic tasks [13], [14], [15], [16], [17], [18]. Sha [19] developed the Simplex Architecture to support safe and reliable online upgrade of hardware and software components in spite of errors in the new module. It is achieved by the use of analytic redundancy and real-time dynamic component binding. Melhem et al. [20] proposed a scheme to provide the fault-tolerant functionality to real-time systems with transient and intermittent faults. Based on the assumption that at most one task execution is affected by hardware malfunctions during one time interval Δ_f, two algorithms were presented to reserve fault-tolerant execution time to a queue of real-time tasks. Liestman and Campbell [6] studied the aforementioned fault-tolerant scheduling problem under the assumption that the task system is *simply periodic*, i.e., the period of each task is a multiple of the next smaller period. They proposed two approaches to maximizing the number of primaries scheduled. In the first approach, they first constructed a schedule for the task set in a planning cycle and then rescheduled the task set in a partially-executed schedule whenever a primary is completed successfully. The second approach used the algorithms of the first approach to construct a tree of schedules and then used simple schedulers to implement the scheduling algorithm with the table representations of these trees. In addition to the restricted assumption that the task system is simply periodic, both approaches have their own drawbacks. Since the first approach needs to construct a schedule for the task set and reconstruct the whole schedule for the current planning cycle whenever the execution of a primary succeeds, it incurs significant online overhead. On the other hand, since the second approach uses table-driven schedulers which need to store all possible schedules in a tree data structure, the need for an excessive amount of memory makes the scheduling algorithms impractical. Moreover, their approaches did not reconstruct the schedule if a primary fails to complete and/or produce correct results in time, thus wasting the processor time originally allocated to the failed primary.

Chetto and Chetto [21] used a *last chance strategy* to achieve the same objective. An offline scheduler reserves time intervals for the alternates. Each such interval is chosen so that any alternate starts its execution at the latest possible time. At runtime, the primaries are scheduled during the remaining intervals before their alternates. The alternates can preempt a primary when a time interval reserved for the alternates is reached. Whenever a primary is completed successfully, the execution of its corresponding alternate is no longer needed and, hence, an online scheduling algorithm must dynamically deallocate the time interval(s) reserved for the alternate so as to increase the processor time available for the execution of other primaries. Their algorithm is based on a *dynamic priority-driven preemptive scheduling scheme*, called the *earliest-deadline-first* (EDF) algorithm. They first used an offline EDF, called *earliest-deadline-first as late as possible* (EDL), to reserve time intervals for the alternates. Then, at runtime, they used any online preemptive scheduling algorithm to schedule the primaries and, whenever a primary is successfully completed the reserved time intervals for the alternates are reconstructed/modified. Such reconstruction is achieved by removing the alternate corresponding to the completed primary and rescheduling the remaining alternates according to EDL from the time the corresponding primary was completed to the end of the current planning cycle. The reconstruction takes a significant amount of time, making the online overhead of their algorithm high. Moreover, their algorithm does not consider the case in which one early job failure could lead to lots of subsequent job failures, thus degrading the system performance significantly.

Our algorithm follows the last chance strategy, except that it is based on fixed priority-driven preemptive scheduling scheme, such as the *rate-monotonic* (RM) algorithm [11], to reserve a priori time intervals for the alternates. At runtime, we dynamically adjust/reconstruct the reserved time intervals for the alternates whenever a primary is successfully completed and, hence, the corresponding alternate is no longer needed. Our reconstruction algorithm takes less time since we use RM algorithm and we need to reconstruct the reserved time intervals only for those alternates affected, specifically, the tasks with lower priority, thus reducing the runtime computation overhead. Moreover, we add two simple modifications to the online scheduling of primaries or alternates, making significant improvements in efficiency of processor usage as well as percentage of successful primary executions.

The rest of the paper is organized as follows: Section 2 briefly reviews periodic task systems and the associated scheduling algorithms and presents a formal description of our problem. In Section 3, we first describe the basic fault-tolerant scheduling algorithm, then discuss its potential problem, and, finally, present two simple ideas which can be integrated into the basic algorithm to achieve more efficient processor usage. Section 4 presents the simulation results of our proposed algorithm. The paper concludes with Section 5.

2 Background and Problem Formulation

The main intent of this paper is to schedule, with the deadline mechanism, real-time periodic tasks, each of which uses two independent versions of the task for the purpose of meeting timing constraints while tolerating software faults. For convenience of presentation, we will first briefly review the

traditional real-time periodic task system and several commonly used scheduling algorithms. We will then formally state the problem addressed in this paper.

2.1 Periodic Task Systems and Scheduling Algorithms

A real-time periodic task system [11] consists of a set of n periodic tasks $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$. Each task τ_i must be executed once every T_i time units, where T_i is the *period* of τ_i. Each execution of a periodic task is called a *job* (or *request*), and every job of τ_i has a *computation* (or *execution*) *time* e_i. The *j*th job of τ_i is denoted as J_{ij} for all $1 \leq i \leq n$ and $j \geq 1$. J_{ij} is ready for execution at time $(j-1) \cdot T_i$ and must be completed by the start time of the next job period of the same task, which is equal to $j \cdot T_i$. We define $r_{ij} = (j-1) \cdot T_i$ to be the *request time* (or *release time*) and $d_{ij} = j \cdot T_i$ the *deadline* of J_{ij}.

A commonly used online scheduling scheme for the traditional real-time periodic task system is the priority-driven preemptive scheduling scheme. For preemptive scheduling, tasks (jobs) can be suspended during their execution and resumed at a later time. For priority-driven scheduling, each task is given a (fixed or dynamic) priority. At runtime, the scheduler always chooses among the *active* jobs the one with the highest priority to execute next, where an active job is one whose execution has been requested but unfinished. If each task is given a fixed priority for all of its executions (jobs), the scheduling scheme is said to be *fixed* or *static* priority-driven. If the priority of a task changes from one execution to another (i.e., each job of the task has its own distinct priority), then the scheduling scheme is called *dynamic* priority-driven.

If a scheduling algorithm produces a schedule for a set of tasks in which each job J_{ij} starts its execution after its request time r_{ij} and finishes before its deadline d_{ij}, then the schedule is said to be *feasible* and the scheduling algorithm is said to *feasibly schedule* the task set. A scheduling algorithm is said to be *optimal* for a particular scheduling scheme if, for any task set that can be feasibly scheduled by any other algorithm of the same scheme, it can also be feasibly scheduled by the algorithm. Liu and Layland [11] showed that 1) the rate-monotonic (RM) algorithm which assigns priorities to tasks according to the *rate-monotonic rule*—the shorter the period the higher the priority—is optimal for fixed priority-driven scheduling schemes and 2) the *deadline-driven* algorithm, or termed elsewhere the earliest-deadline-first (EDF) algorithm, which assigns priorities to jobs according to their deadlines—the earlier the deadline, the higher the priority—is optimal for dynamic priority-driven scheduling schemes. Liu and Layland also showed that a task set can be feasibly scheduled by the EDF algorithm if and only if the (*processor*) *utilization factor* $U(\tau) = \sum_{i=1}^{n} e_i/T_i$ is less than or equal to one and the least upper bound for a task set to be feasibly scheduled by the RM algorithm is $K(n) = n(2^{1/n} - 1)$, i.e., if $U(\tau) = \sum_{i=1}^{n} e_i/T_i \leq K(n) = n(2^{1/n} - 1)$, then the task set is guaranteed to be schedulable by the RM algorithm.

2.2 Problem Formulation

The fault-tolerant real-time periodic task system considered in this paper is formally defined as follows: Consider a set of n real-time periodic tasks $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$. Each task τ_i has a period T_i and two independent versions of computation program: the *primary* P_i and the *alternate* A_i. The primary contains more functions and, when executed correctly, produces good quality results, but its reliability cannot be guaranteed because of its complicated functions (to produce good quality results) that are difficult to test/verify. The alternate is reliable due to its simple functions that are easy to test and produces less precise but acceptable results. P_i has a computation time p_i, A_i has a computation time a_i, and, usually, $p_i \geq a_i$ for $1 \leq i \leq n$. In order to define the possible task failures, we use FP_i to denote the probability that the primary fails during its execution.

Let the planning cycle, T, be the least common multiple (LCM) of $T_1, T_2, \ldots, T_n$. Then, $n_i = T/T_i$ is the number of jobs of τ_i in each planning cycle. Since the task-invocation behavior repeats itself for every planning cycle, we only need to consider all task invocations during any one planning cycle. Thus, without loss of generality, we can consider the problem of scheduling tasks for the first planning cycle $[0, T]$. The primary and the alternate of the *j*th job J_{ij} of τ_i are denoted by P_{ij} and A_{ij}, respectively. For each J_{ij} of τ_i, either P_{ij} or A_{ij} must be completed by its deadline $j \cdot T_i$. Since P_{ij} provides a better computation quality, we would prefer the execution of P_{ij} to that of A_{ij}. However, in case P_{ij} fails, we must ensure A_{ij} to be completed by its deadline, thus providing an acceptable, though possibly degraded, computation quality. That is, we want to complete as many primaries as possible while guaranteeing either the primary or the alternate of each task to be successfully completed by its deadline.

3 The Proposed Approach

In this section, we first present our basic fault-tolerant algorithm and the corresponding schedulability analysis. Then, we point out some problems existing in the basic algorithm and propose two more ideas to improve the percentage of the successful primaries.

3.1 The Basic Algorithm

As mentioned earlier, our algorithm uses the last chance philosophy [21]: If there are primaries pending for execution, alternates will not be scheduled until the latest possible time, called the *notification time*, on or before which if alternates are not scheduled, they will not be completed in time. We first give an overview of the algorithm and then illustrate the algorithm with examples.

3.1.1 Overview

Given a real-time periodic task set τ, we first use any fixed priority-driven scheduling algorithm to reserve time intervals as late as possible for all the alternates in a planning cycle before runtime. At runtime, if there are primaries pending during the time intervals that were not reserved by alternates, the scheduler chooses the primaries to execute first. The primaries can be scheduled by any online scheduling algorithm, such as a (fixed or dynamic)

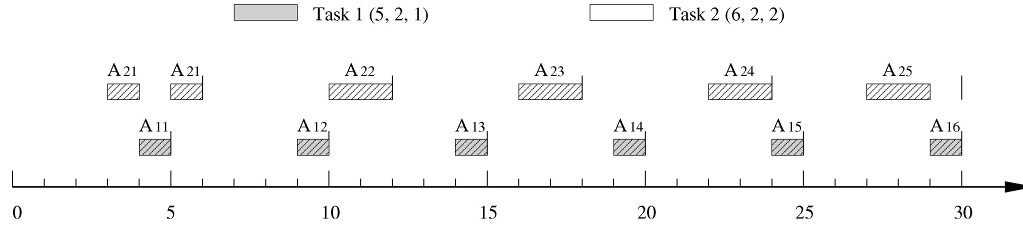

Fig. 1. The notification times calculated by the backward-RM algorithm.

priority-driven preemptive scheduling scheme with the RM or EDF priority assignment. A primary may fail (because of software bugs or taking too long to complete) at any time during its execution. If a primary fails, its corresponding alternate must be executed. Moreover, when the notification time, v_{ij}, of alternate A_{ij} is reached, yet its corresponding primary P_{ij} has not been completed or has failed, A_{ij} is activated (thus preempting the execution of any primary, including P_{ij}, or other lower-priority alternates). The primary P_{ij}, if it has not been finished, will be aborted since its alternate A_{ij} is now chosen to be executed. Every alternate, if activated on or after its notification time, has higher priority than all primaries and the activated alternates are executed according to their priorities assigned by the offline fixed-priority algorithm.

Note, however, that an alternate need not be activated if its corresponding primary has been successfully completed before its notification time. That is, if P_{ij} finishes its execution successfully before the notification time v_{ij}, the alternate A_{ij} need not be activated and, hence, the time interval(s) allocated to A_{ij} can be reallocated to other primaries or alternates. In this case, the notification time of A_{ij} is no longer needed and is thus cancelled. Moreover, the notification times of other alternates need to be adjusted since the time reserved for A_{ij} is now freed.

3.1.2 Details and Examples

We now present the details of the algorithm. The notification time v_{ij} of an alternate A_{ij} can be calculated by any fixed-priority algorithm as follows: In calculating the notification times, we consider only the alternates A_{ij}s (i.e., we use a_i as the computation time of τ_i as in the traditional periodic task system, which has only one version per task) and use a fixed-priority algorithm F to construct a schedule (for example, by using the *slack interval* method described in [17]) backward from time T to time 0, and find the "finish" time v_{ij} of A_{ij} in the schedule, for each $1 \leq i \leq n$ and $1 \leq j \leq n_i$. We will call this algorithm the *backward-F* algorithm for a given fixed-priority scheduling algorithm F. We then use v_{ij} as the notification time of A_{ij} at runtime. That is, the notification times are simply the finish times of the alternates if they are scheduled by a fixed-priority algorithm backward from time T. (We assume that the alternates can be feasibly scheduled by the algorithm.) Note that the notification times force the alternates to be scheduled as late as possible and, hence, the scheduling algorithm leaves the largest possible room for executing the primaries before executing the alternates. The following example illustrates how the notification times are calculated under the assumption that the underlying fixed-priority algorithm is the RM algorithm.

Fig. 1 illustrates how the notification times are calculated under the assumption that the underlying fixed-priority algorithm is the RM algorithm. Consider a periodic task set $\tau = \{\tau_1, \tau_2\}$ with $(T_i, p_i, a_i = (5, 2, 1))$ [2] and (6, 2, 2), respectively, thus the planning cycle $T = LCM(5, 6) = 30$. (Note that the numbers for T_is, p_is, and a_is in this example are chosen for ease of demonstration.) We use the RM algorithm to schedule the alternates backward from time 30 to time 0 and find the notification times of all the alternates in $[0, 30]$. As shown in Fig. 1, since Task 1 has higher priority than Task 2, the notification times for Task 1 are first calculated as $v_{1j} =$ 4, 9, 14, 19, 24, and 29, for $1 \leq j \leq 6$. Then, the alternates for Task 2 are reserved for the time slots left over by the accommodation of the alternates for Task 1 and the corresponding notification times are $v_{2j} =$ 3, 10, 16, 22, and 27, for $1 \leq j \leq 5$.

In the above calculation of the notification times, we assume that all tasks are requested simultaneously from the start. Now, we discuss the phasing problem and show the worst case for task scheduling happens when all tasks start simultaneously. As mentioned before, we want to leave the largest possible room for executing the primaries by setting up the notification time as late as possible. In other words, the worst case happens when the notification times are set at the earliest possible time point. If we use RM to schedule backward from time T to time 0, the case mentioned above is equivalent to the case that the "finish" time of each job is delayed as much as possible. This is exactly the same phasing problem analyzed in Liu and Layland's paper [11] and they have proven that the worst case happens when all backward tasks are initiated at time T.

At runtime, the primaries are scheduled during the remaining intervals before their alternates. Whenever a primary is successfully completed, the time interval(s) reserved for the execution of its corresponding alternate is (are) no longer needed and, hence, can be deallocated. Otherwise, the corresponding alternate has to be executed later for providing reliability. The alternates can preempt a primary when a time interval reserved for the alternates is reached. When the notification time of an alternate A_{ij} is reached and if the corresponding primary P_{ij} has not been completed, P_{ij} has to be aborted because we are not sure whether or not the execution of P_{ij} will finally be successful and we do not want to end up with no completed primary or alternate by giving more time for P_{ij}'s execution. Fig. 2 illustrates all these runtime cases for the task set in the last

2. We omit the failure probability parameter FP_i for simplicity.

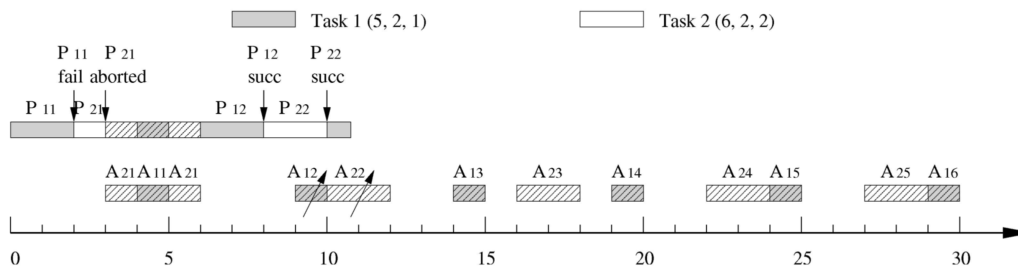

Fig. 2. The runtime dynamic job scheduling.

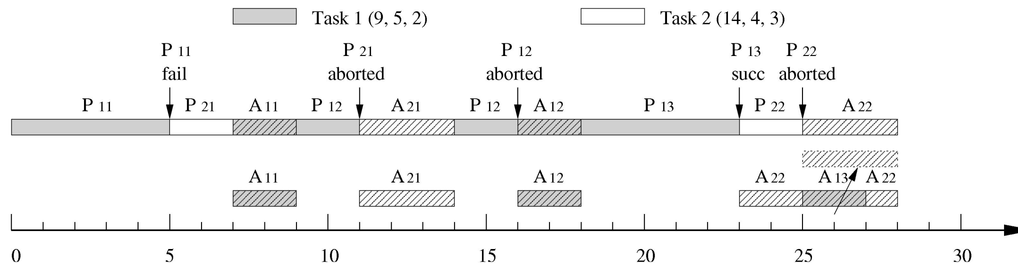

Fig. 3. The pathological case demonstrating the drawback of the basic scheduling algorithm.

example. At time 0, we execute primary P_{11} and suppose it fails (due to software bugs) at the end of its execution time 2. The time interval $[4, 5]$ allocated to A_{11} cannot be cancelled and it will be executed later. From time 2, P_{21} is executed until time 3, at which its execution will be aborted because the notification time of the corresponding alternate A_{21} is reached. Then, in interval $[3, 6]$, two alternates A_{11} and A_{21} are executed in their preallocated time intervals. At time 6, there are two active primary jobs P_{12} and P_{22}. Since P_{12} has higher priority, we select P_{12} to start its execution and suppose it succeeds at time 8. The time interval $[9, 10]$ allocated to A_{12} is no longer needed and, hence, can be freed. Now, P_{22} can start its execution and complete successfully at time 10. Finally, the corresponding preallocated alternate time interval $[10, 12]$ is also removed. Similarly, the subsequent steps can be reached by applying the basic online scheduling algorithm.

3.1.3 Schedulability Analysis

Our proposed algorithm has two main objectives: 1) guarantee either primary or alternate of each task (job) to be successfully completed before their corresponding deadlines; (2) complete as many primaries as possible to achieve better computation quality. The first objective is achieved by using an offline fixed priority scheduling algorithm (such as RM) to ensure the successful accommodation of all alternate jobs. This offline schedule is constructed backward from time T and the alternates are executed as late as possible, thus leaving the largest possible room for the execution of the primaries to accomplish the second goal.

Therefore, our proposed algorithm is based on the condition that all the alternates can be scheduled successfully by the offline static priority scheduling algorithm. Suppose we use the backward-RM algorithm to construct the offline schedule. Since the least upper bound of the processor utilization for a task set to be schedulable by the RM algorithm is $n(2^{1/n} - 1)$ [11], our approach is feasible when the utilization of the alternates is no more than $n(2^{1/n} - 1)$, that is, $U_p = \sum_{i=1}^{n} a_i/T_i \leq n(2^{1/n} - 1)$.[3]

3.2 The Modified Algorithm

3.2.1 Existing Problems

Our basic scheduling algorithm seems good in the sense that it has provided timeliness and reliability at the same time. Unfortunately, we found that there exist some cases in which the basic fault-tolerant algorithm suffers very large system degradation, which is, "one job failure could lead to lots of subsequent job failures." Fig. 3 illustrates one of such examples. Suppose the task set is $\tau = \{\tau_1, \tau_2\}$ with (T_i, p_i, a_i) = (9, 5, 1) and (14, 4, 3) and the planning cycle $T = LCM(9, 14) = 126$. Suppose the primary P_{11} fails at the end of its execution, or at time 5. So, the time allocated to its corresponding alternate $[7, 9]$ cannot be cancelled. We apply the basic algorithm to schedule the execution of primaries or alternates and the scheduling details are omitted for simplicity. Fig. 3 shows a small part of the schedule. As can be seen in the figure, at least three subsequent primaries are affected by P_{11}'s failure and their executions are aborted because of having insufficient processor resources. This example exhibits the disadvantages of the basic algorithm: It lacks some functionality to provide protection for subsequent jobs from being affected by the early failures.

3.2.2 The Checking Available Time (CAT) Algorithm

We can deal with the aforementioned problem by making careful use of processor cycles in case of failures in executing primaries. Note that, in the above example, some primary execution is wasted and the design of a better algorithm to utilize the processor more efficiently is possible. As seen in Fig. 3, at time 9, we choose to execute P_{21} since it has higher priority. P_{21} is executed during time

3. The sufficient condition for a task set to be schedulable by RM is presented in [11].

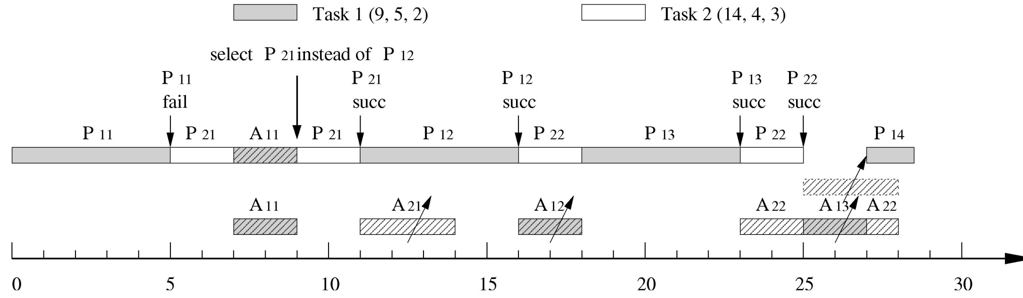

Fig. 4. The task set scheduled by the CAT algorithm.

interval $[9, 11]$ and $[14, 16]$, but finally gets aborted at time 16, the notification time of its corresponding alternate A_{21}. However, at time 9, it is known that the unallocated time interval between that time and its notification time 16 are only four units of time, which is less than the required five units of time for P_{21}'s execution. The system has insufficient processor cycles available to accommodate the primary P_{12}, so there are no benefits to be gained from this execution (actually, it will result in waste of processor time). We can use such wasted time for other jobs to get more productive processor usage.

Therefore, the basic idea of our new algorithm is that, before selecting a primary to execute, we have to verify whether or not there are enough unallocated units of time for its completion. If we apply this new algorithm, then, at time 9, P_{21} is selected to start execution instead of P_{12} since there is an insufficient amount of time for P_{12} to complete its execution before the corresponding notification time. After such a small modification, as shown in Fig. 4, the new algorithm makes a significant improvement since there are no more subsequent primary failures due to the early failure of primary job P_{11}.

The new algorithm is formalized as follows: As mentioned before, at runtime, the scheduler always chooses, among the *active* jobs, the one with the highest priority to execute next, where an *active* job is the one which is released but unfinished. Our new approach modifies the definition of the *active* job and adds one more constraint to it: The *active* job must have enough unreserved units of time to complete before its corresponding deadline. Suppose, at time t, there are $k \le n$ primaries which have been released but uncompleted. For each of these primaries P_{ij}, the corresponding notification time is v_{ij}. Suppose there are ℓ time intervals $\{I_i, 1 \le i \le \ell\}$ that have been allocated for alternates by the offline backward-RM algorithm between time t and time v_{ij}, the unallocated time interval for P_{ij} is calculated as

$$AT_{ij} = (v_{ij} - t) - \sum_{i=1}^{l} I_i,$$

where AT stands for "Available Time." P_{ij} is not regarded as an active job unless the calculated available time AT_{ij} is greater than, or equal to, the required execution time for its completion. For example, in the above example, P_{12} is not an active job at time 9 because the unallocated time available at that instant is only four units of time, but it needs five units of time to complete. The new algorithm is called the CAT (Checking Available Time) algorithm.

In the CAT algorithm, we calculate the available processor time for each primary P_{ij} only by subtracting the preallocated time intervals for alternates from the interval between that time instant and the corresponding deadline. However, at runtime, there could be other higher-priority primaries that will be invoked later, but before P_{ij}'s completion, and preempt P_{ij}. Moreover, the preallocated time units for alternates in the time interval $[t, v_{ij}]$ change dynamically because they can be deallocated if their corresponding primaries are completed successfully. In what follows, we prove that the available time estimated by the CAT algorithm is the maximum possible number of time units available for P_{ij}'s execution.

Theorem. *The AT_{ij} calculated by the CAT algorithm estimates the maximum available time that can be used for P_{ij}'s execution when the inequality $a_i \le p_i$ holds for each task τ_i, $1 \le i \le n$.*

Proof. As shown in Fig. 5, two types of conditions are possible to occur during the time interval $[t, v_{ij}]$ that might change the processor cycles available for primary P_{ij}'s execution.

Case 1. A certain higher-priority primary, such as P_{xy} in Fig. 5, is invoked and preempts the execution of P_{ij}. Thus, the actual available time for P_{ij} becomes $AT_{ij}^{acutal} = AT_{ij} - p_x$, where p_x is the execution time of P_{xy}. Obviously, $AT_{ij}^{acutal} \le AT_{ij}$.

Case 2. Some primary, for example, P_{vw} in Fig. 5, with higher priority than P_{ij} is invoked, preempts P_{ij}, and, finally, finishes its execution successfully. According to the basic algorithm, the processor time allocated for alternate A_{vw} should be deallocated. Suppose that the time allocated to A_{vw} is located in the interval $[t, v_{ij}]$, then the actual time available for P_{ij}'s execution becomes $AT_{ij}^{actual} = AT_{ij} - p_v + a_v$, where p_v and a_v are the execution times of P_{vw} and A_{vw}, respectively. Since the alternate usually contains fewer functions than the corresponding primary, we assume that the inequality $a_v \le p_v$ always holds. Therefore, the actual available time

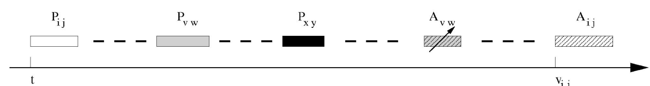

Fig. 5. The proof of the validity of the CAT algorithm.

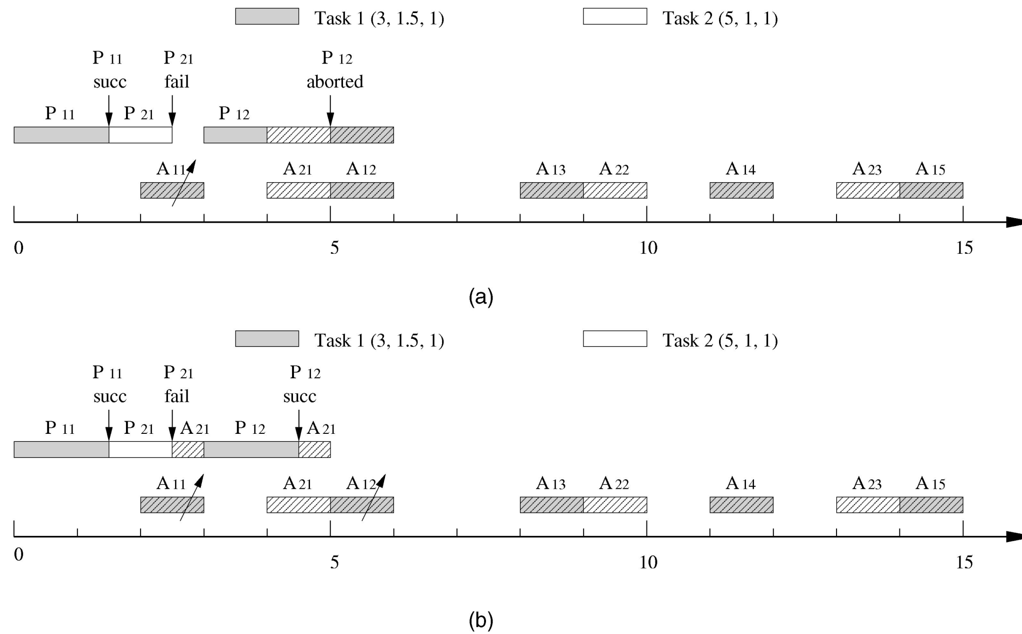

Fig. 6. The task schedule illustrating the advantages of the EIT algorithm. (a) The task set scheduled by the basic algorithm. (b) The task set scheduled by the EIT algorithm.

is also no greater than the estimated available time by the CAT algorithm.

Combining **Case 1** and **Case 2** proves that the available time calculated by the CAT algorithm is the largest possible processor time available for P_{ij}'s execution. In other words, if the calculated available time AT_{ij} is less than the time required to complete P_{ij}'s execution, it is impossible to finish P_{ij} before its deadline, no matter what happens at runtime. Therefore, it is logical to disallow P_{ij}'s execution and utilize the available time for other jobs' execution. □

The merit of the CAT algorithm lies in that it tries to eliminate the wasteful execution of primaries as much as possible and utilize processor time thus saved for productive execution of other jobs. Note, however, that the CAT algorithm incurs an online computation overhead because it needs to calculate the available time for each primary that has been released but unfinished. Moreover, in the above proof, it is shown that the CAT algorithm estimates the maximum possible unallocated/available time. Since, at runtime, some higher-priority primaries could preempt P_{ij}'s execution, e.g., **Case 1** in the above proof, there is actually less time to be allocated for P_{ij}'s execution and, consequently, P_{ij} may fail because of reduced (hence insufficient) processor time available to it. Under this condition, the amount of time that has already been consumed by P_{ij}'s execution would also become a waste. Taking into account this dynamic execution of primaries instead of merely considering the reserved processor cycles for alternates during $[t, v_{ij}]$ can achieve even more efficiency in processor usage, but this incurs more computational overhead. The CAT algorithm makes a relatively good tradeoff between elimination of the waste of processor time and reduction of runtime computational overhead.

3.2.3 The Eliminating Idle Time (EIT) Algorithm

Another algorithm can be incorporated into the CAT algorithm to achieve better system performance. In case there are no active primaries waiting for execution and none of the alternates' notification times has been reached, the processor will become idle, hence wasting the processor cycles. Moreover, since the CAT algorithm adds one more constraint on the primaries being "active," the chance that there are no active primaries and the processor becomes idle is further increased .

Namely, the Eliminating Idle Time (EIT) algorithm chooses an alternate to execute when the processor is about to be idle even though its latest start time, or notification time, is not reached. If a primary arrives or the notification time of another alternate is reached during the execution of this alternate, say A_{xy}, prior to its notification time, A_{xy} will be preempted and its notification time should be adjusted to reflect the progress made in its execution, i.e., the reduced computation time. Moreover, the notification times of other alternates should also be adjusted accordingly. As a result, each alternate has two execution modes: *prenotification* and *postnotification*. An alternate executing before its notification time is said to be in its prenotification mode and has lower priority than all primaries. An alternate executing on and after its notification time is said to be in its postnotification mode and has higher priority than any primary (and, hence, has higher priority than any alternate in its prenotification mode). By using the EIT algorithm, the processor's idle time is utilized by the alternate whose execution is advanced and the time thus saved is made available for possible productive execution of jobs at a later time.

Fig. 6 illustrates the advantages of the EIT algorithm. Consider a task set $\tau = \{\tau_1, \tau_2\}$ with (T_i, p_i, a_i) = (3, 1.5, 1) and (5, 1, 1) and the planning cycle $T = LCM(3, 5) = 15$. As shown in Fig. 6a, suppose P_{11} finishes its execution successfully at time 1.5, the corresponding alternate A_{11}

```
Procedure Modified-Algorithm( )
  static:
    alternateList, the alternate job list in ascending order with time;
    t, the current instant, initiated to be 0;
    primaryList, the primary job set in descending order with priority;
    candidate, the active primary with the highest priority;

  Use the backwards-RM algorithm to construct alternateList;
  while(TRUE)
    if t ≥ alternateList.head.notificationTime then
      execute alternateList.head and remove it after completion;
      goto next;
    end
    for each primary P_ij in primaryList
      calculate AT_ij using the CAT algorithm;
      if AT_ij ≥ the required time of P_ij then
        set candidate as P_ij;
        break;
      end
    end
    if candidate ≠ null then
      execute candidate;
      if candidate is completed successfully then
        remove the corresponding alternate from alternateList;
        if needed, adjust the execution time of other alternates affected;
      goto next;
    else
      processor is idle, advance the alternate with lowest priority (the EIT algorithm);
      if needed, adjust the execution time of other alternates;
    end
next:
    set the next trigger time t;
  end
```

Fig. 7. The pseudocode of the modified fault-tolerant scheduling algorithms.

releases the preallocated time interval $[2, 3]$. P_{21} executes during $[1.5, 2.5]$, and suppose it finally fails at time 2.5. At time 2.5, there are no more active primaries waiting for execution and the processor is idle. At time 3, P_{12} is invoked and continues execution until time 4, when the notification time of A_{21} is reached. P_{12} is preempted and aborted when the notification time of A_{12} is reached. In Fig. 6b, the EIT algorithm is applied to save the processor's idle time. At time 2.5, instead of leaving the processor idle, the EIT algorithm advances the execution of alternate A_{21} and saves the idle time interval $[2.5, 3]$ into interval $[4, 4.5]$ that is used later for P_{12}'s execution. P_{12} is completed successfully at time 4.5 and the corresponding alternate A_{12} is deallocated. Consequently, the EIT algorithm enhances processor time usage and the percentage of successful primaries as well.

When the processor is idle, the execution of any alternate whose corresponding primaries have not been completed successfully could be advanced. As mentioned before, advancing the execution of one alternate could lead to the adjustment of other alternates' execution times. Our approach is to choose, among all alternates, the one with the lowest priority. It is easy to show that this eliminates the maximum computation overhead in adjusting the notification time of other alternates.

3.2.4 *Pseudocode Description of a Combined CAT and EIT Algorithm*

In this section, we present, in Fig. 7, the pseudocode of our modified fault-tolerant scheduling algorithm which integrates the CAT and EIT algorithms into the basic scheduling algorithm.

4 Simulation Results

This section presents the simulation results of our proposed algorithm for different task sets. We demonstrate the strength of our modified scheduling algorithm by comparing its simulation results with those generated with the basic algorithm.

4.1 Metrics

As mentioned earlier, the objective of our fault-tolerant scheduling algorithm is to guarantee either the primary or alternate version of each job to be successfully completed before its corresponding deadline while trying to complete as many primaries as possible. Therefore, we define the following two metrics: $PctSucc_i$, which indicates the percentage of successfully completed primaries for each

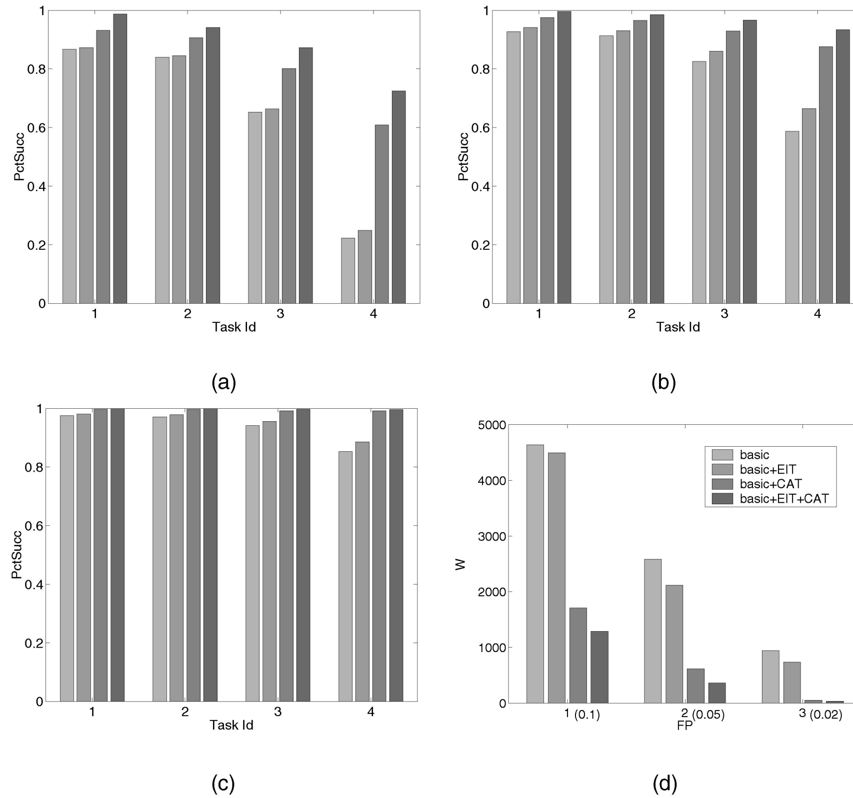

Fig. 8. The results of Simulation I. (a) FP_ = 0.1. (b) FP = 0.05. (c) FP = 0.02. (d) Wasted time.

task, and W, the processor time wasted by executing unsuccessful primaries during the whole time span of the schedule.

4.2 Simulation I

The first simulation aims to demonstrate the merits of our proposed algorithm by showing the improvement of the algorithm achieved in terms of the above two metrics. The task set used in the simulation is $\tau = \{\tau_1, \tau_2, \tau_3, \tau_4\}$ with (T_i, p_i, a_i) = (13, 3, 2), (24, 7, 3), (39, 9, 7), and (144, 23, 17) and the planning cycle is $T = LCM(13, 24, 39, 144) = 1,872$. For each task set, the following four algorithms (basic, basic+CAT, basic+EIT, and basic+EIT+CAT) are independently applied to run for 19 planning cycles, which are 35,568 units of time[4] and the simulation results are recorded correspondingly. We also test these algorithms for different primary failure probabilities. the simulation results are plotted in Fig. 8.

Here, we give an example to show how to calculate the two metrics: *PctSucc* and *W*. Suppose FP (failure probability) is 0.1 and there are 20 jobs, then there are two ($= 0.1 \times 20$) primary failures because of the software bugs and at most 18 successful primaries. If the actual schedule accommodates only nine successful primaries, then $PctSucc = 9/18 = 50\%$. That is, *PctSucc* is the percentage of actual successful primaries among the maximum possible successful primaries, thus representing how many subsequent primaries are affected by the early failures and how well the corresponding scheduling algorithm deals with early primary failures. If the execution of a primary is aborted when the corresponding notification time is reached, the amount of time that has already been consumed by the primary is regarded as wasted. W, the wasted processor time, is calculated by summing up the time slots wasted by all unsuccessful primaries.

4. Multiple planning cycles are used just for enlarging the graphs of the simulation results.

As shown in Fig. 8, the simulation results exhibit the following features:

1. For the basic algorithm, the lower-priority task suffers failure more significantly. For example, for FP = 0.1, the percentage of successful primaries for task 4 degrades to only 20 percent, while that of task 1 is greater than 80 percent. Task 4 gets less of a chance to be executed because of its low priority.
2. Integrating either the CAT algorithm or the EIT algorithm decreases the wasted processor time and enhances the percentage of successful primaries. As shown in Fig. 8, the "basic+EIT" and "basic+CAT" algorithms make some improvements over the basic algorithm. The highest system performance is achieved by the basic algorithm combined with both EIT and CAT. As shown in Fig. 8a, for task 4, the "basic+EIT+CAT" algorithm increases the percentage of successful primaries from 20 percent to 75 percent, improved by nearly 300 percent. In the meantime, in Fig. 8d, the wasted time decreases from 4,700 to only 1,200, reducing by 75 percent.
3. As shown in Fig. 8c and Fig. 8d, when the failure probability FP is small, e.g., FP = 2%, the "basic+

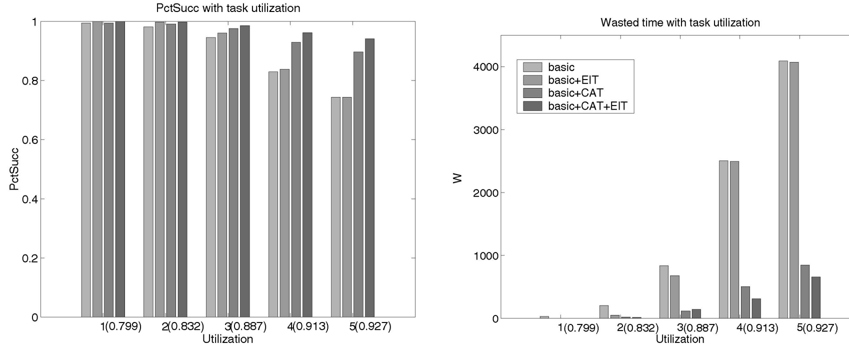

Fig. 9. The results of Simulation II.

EIT+CAT" algorithm avoids almost all the wasted time on primaries that are unable to be completed because of insufficient processor time, thus protecting the subsequent primaries from being affected by early failures.

4. It is noticed that, when FP is large (5 percent or more), even if the best algorithm, the "basic+EIT+-CAT" algorithm, is applied, there is still some execution time wasted, especially under a poor system condition (i.e., with a large failure probability). As discussed in Section 3.2.2, accounting for the dynamic execution of primaries instead of only considering the reserved times for alternates can make more improvements. However, this is not worthy of application in the sense that 1) it incurs high run-time computation overhead and 2) it is very difficult at runtime to estimate how many failures will occur during one primary's execution, especially when the failure of each task happens independently from that of the others.

4.3 Simulation II

In the first simulation, we demonstrated the performance of our proposed scheduling algorithms by applying them to a specific task set. In this section, we show that the performance that the proposed algorithm achieves depends on the specific task set. We apply the algorithms to five task sets with different processor utilizations, which is $\sum_{i=1}^{5}(p_i/T_i)$, where p_i, T_i are the execution time of primaries and the period, respectively, for task τ_i. As shown in Fig. 9,[5] in case the processor utilization is low, for example, the first task set whose utilization is 0.799, the system performance does not suffer large degradation even if the basic algorithm is applied, meaning that the system has enough unallocated time to handle failures. In such cases, there is no benefit of applying the CAT and EIT algorithms that incur computational overheads. Only when the original processor utilization is very high, for example, the task set with utilization 0.927, and the system cannot deal with failures easily, does the application of the modified algorithm achieves significant performance improvement.

5. The five task sets have the processor utilization of 0.709, 0.832, 0.887, 0.913, and 0.927, respectively. For ease of illustration, we do not include the specific parameters of each task set. The failure probability in this simulation is fixed at 0.05.

4.4 A Further Extension

In both the simulation results, the metric W (Wasted Time) is found to be a good indicator of the current system performance. A large W indicates that many primaries have been aborted because of insufficient processor time, so our modified algorithm enhances the efficiency of processor usage, thus alleviating the performance degradation. If the current wasted time W is small, meaning that the system is able to deal with the failures, there is no need to adopt the modified algorithm which incurs computation overhead.

An adaptive algorithm can be designed as follows: Because the wasted time W is the sum of all the wasted times by individual unsuccessful primaries thus far, instead of using the absolute value, we will calculate the increment of W (denoted as ΔW during a certain period as the system performance indicator. If ΔW is larger than some predetermined threshold, then the modified algorithm, the "basic+-CAT+EIT" algorithm, is activated; else, only the basic algorithm is used to save much computation overhead.

5 Conclusion

With efficient scheduling algorithms and software fault-tolerant deadline mechanisms, one can design a system that meets its task timing constraints while tolerating software faults.

In this paper, we considered the problem of scheduling real-time periodic tasks using the deadline mechanism to provide software fault tolerance. We proposed a scheduling algorithm based on the last chance philosophy to schedule the alternates as late as possible and leave as much room as possible for executing the primaries before their deadlines. Our basic algorithm makes significant performance improvements by integrating two very simple but elegant heuristics. The simulation results have shown that, in some worst cases, specifically, the cases with high failure probability and/or high task utilization, the modified algorithm can make an impressive performance enhancement; for example, the percentage of successful primaries is increased by 300 percent in certain cases. In the final part of this paper, we also proposed a more adaptive algorithm, enabling our algorithm to deal with dynamic task systems.

ACKNOWLEDGMENTS

The work reported in this paper was supported in part by the US Office of Naval Research under Grant N00014-99-1-0465 and the US Defense Advanced Research Projects Agency under US Airforce Contract F49620-01-1-020. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

REFERENCES

[1] K.-J. Lin, S. Natarajan, and J.W.-S. Liu, "Imprecise Results: Utilizing Partial Computations in Real-Time Systems," *Proc. Real-Time Systems Symp.*, pp. 210-217, Dec. 1987.

[2] K.B. Kenny, "Structuring Real-Time Systems Using Performance Polymorphism," PhD thesis, Univ. of Illinois at Urbana-Champaign, Nov. 1990.

[3] J.A. Stankovic, "Misconceptions about Real-Time Computing," *Computer*, pp. 10-19, Oct. 1988.

[4] J. Goldberg et al., "Development and Analysis of SIFT," NASA contractor report 17146, NASA Langley Research Center, Feb. 1984.

[5] R.H. Campbell, K.H. Horton, and G.G. Belford, "Simulations of a Fault-Tolerant Deadline Mechanism," *Proc. Ninth Fault-Tolerant Computing Symp. (FTCS-9)*, pp. 95-101, June 1979.

[6] A.L. Liestman and R.H. Campbell, "A Fault-Tolerant Scheduling Problem," *IEEE Trans. Software Eng.*, vol. 12, no. 11, pp. 1089-1095, Nov. 1986.

[7] M.G. Harbour, M.H. Klein, and J.P. Lehoczky, "Timing Analysis for Fixed-Priority Scheduling of Hard Real-Time Systems," *IEEE Trans. Software Eng.*, vol. 20, no. 1, pp. 13-28, Jan. 1994.

[8] J. Lehoczky, L. Sha, and Y. Ding, "The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior," *Proc. Real-Time Systems Symp.*, pp. 166-171, Dec. 1989.

[9] J.Y.-T. Leung and M.L. Merrill, "A Note on Preemptive Scheduling of Periodic, Rreal-Time Tasks," *Information Processing Letters*, vol. 11, no. 3, pp. 115-118, Nov. 1980.

[10] J.Y.-T. Leung and J. Whitehead, "On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks," *Performance Evaluation*, vol. 2, pp. 237-250, 1982.

[11] C.L. Liu and J.W. Layland, "Scheduling Algorithms for Multi-programming in a Hard Real-Time Environment," *J. ACM*, vol. 20, no. 1, pp. 46-61, Jan. 1973.

[12] L. Sha, R. Rajkumar, and J.P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," *IEEE Trans. Computers*, vol. 39, no. 9, pp. 1175-1185, Jan. 1990.

[13] C.-C. Han and K.G. Shin, "A Globally Optimal Algorithm for Scheduling Both Hard Periodic and Soft Aperiodic Tasks," *IEEE Trans. Computers*, submitted.

[14] N. Homayoun and P. Ramanathan, "Dynamic Priority Scheduling of Periodic and Aperiodic Tasks in Hard Real-Time Systems," *Real-Time Systems J.*, vol. 6, pp. 207-232, 1994.

[15] J.P. Lehoczky, L. Sha, and J.K. Strosnider, "Enhanced Aperiodic Responsiveness in Hard Real-Time Environments," *Proc. Real-Time Systems Symp.*, pp. 261-270, Dec. 1987.

[16] J.P. Lehoczky and S. Ramos-Thuel, "An Optimal Algorithm for Scheduling Soft-Aperiodic Tasks in Fixed Priority Preemptive Systems," *Proc. Real-Time Systems Symp.*, pp. 110-123, Dec. 1992.

[17] T.-H. Lin and W. Tarng "Scheduling Periodic and Aperiodic Tasks in Hard Real-Time Computing Systems," *Performance Evaluation Rev.*, vol. 19, no. 1, pp. 31-37, May 1991.

[18] B. Sprunt, J. Lehoczky, and L. Sha, "Exploiting Unused Periodic Time for Aperiodic Service Using the Extended Priority Exchange Algorithm," *Proc. Real-Time Systems Symp.*, pp. 251-258, Dec. 1988.

[19] L. Sha, "Dependable System Upgrade," *Proc. 19th IEEE Real-Time Systems Symp.*, pp. 440-449, 1998.

[20] R. Melhem, S. Ghosh, and D. Mosse, "Enhancing Real-Time Schedules to Tolerate Transient Faults," *Proc. 16th IEEE Real-Time Systems Symp.*, pp. 120-129, Dec. 1995.

[21] H. Chetto and M. Chetto, "Some Results of the Earliest Deadline Scheduling Algorithm," *IEEE Trans. Software Eng.*, vol. 15, no. 10, pp. 1261-1269, Oct. 1989.

Ching-Chih (Jason) Han received the BS degree in electrical engineering from National Taiwan University, Taipei, Taiwan, in 1984, the MS degree in computer science from Purdue University, West Lafayette, Indiana, in 1988, and the PhD degree in computer science from the University of Illinois at Urbana-Champaign in 1992. From August 1992 to January 1994, he was an associate professor in the Department of Applied Mathematics at National Sun Yat-sen University, Kaohsiung, Taiwan. From February 1994 to July 1996, he was a visiting research scientist in the Real-Time Computing Laboratory at the University of Michigan, Ann Arbor. From August 1996 to July 1997, he was an assistant professor in the Department of Electrical Engineering at Ohio State University. Since August 1997, Dr. Han has joined several industry companies in the bay area of California, all of which are in the Internet/Web technology area. He also cofounded an Internet startup, CreOsys (now Oridus), Inc, which is now a leading Internet solution provider that integrates real-time, Web-based, distributed graphic information services for the engineering world. He is a member of the IEEE and the IEEE Computer Society.

Kang G. Shin (S'75-M'78-SM'83-F'92) received the BS degree in electronics engineering from Seoul National University, Seoul, Korea, in 1970 and the MS and PhD degrees in electrical engineering from Cornell University, Ithaca, New York, in 1976 and 1978, respectively. He is the Kevin and Nancy O'Connor Professor of Computer Science and Founding Director of the Real-Time Computing Laboratory in the Department of Electrical Engineering and Computer Science, the University of Michigan, Ann Arbor. His current research focuses on QoS-sensitive networking and computing as well as on embedded real-time OS, middleware, and applications, all with emphasis on timeliness and dependability. He has supervised the completion of 42 PhD theses, and authored/coauthored more than 500 technical papers and numerous book chapters in the areas of distributed real-time computing and control, computer networking, fault-tolerant computing, and intelligent manufacturing. He has coauthored (jointly with C.M. Krishna) a textbook *Real-Time Systems* (McGraw-Hill, 1997). From 1978 to 1982, he was on the faculty of Rensselaer Polytechnic Institute, Troy, New York. He has held visiting positions at the US Airforce Flight Dynamics Laboratory, AT&T Bell Laboratories, Computer Science Division within the Department of Electrical Engineering and Computer Science at the University of California Berkeley, and International Computer Science Institute, Berkeley, California, IBM T.J. Watson Research Center, and Software Engineering Institute at Carnegie Mellon University. He also chaired the Computer Science and Engineering Division, EECS Department, the University of Michigan for three years beginning in January 1991. He is a fellow of the IEEE and the ACM and member of the Korean Academy of Engineering, was a Distinguished Visitor of the Computer Society of the IEEE, an editor of the *IEEE Transactions on Parallel and Distributed Computing*, and an area editor of the *International Journal of Time-Critical Computing Systems*, *Computer Networks*, and *ACM Transactions on Embedded Systems*.

Jian Wu (S'02) received the BS and MS degrees in computer science from Tsinghua University, Beijing, China, in 1999 and 2001, respectively. Since August 2001, he has been with the Electrical Engineering and Computer Science Department, University of Michigan, Ann Arbor. His current research interests focus on the design of routing protocols to provide fault-tolerance for multicast communications. He is a student member of the IEEE and the IEEE Computer Society.

▷ **For more information on this or any computing topic, please visit our Digital Library at** http://computer.org/publications/dlib.