Fault-Tolerance Through Scheduling of Aperiodic Tasks in Hard Real-Time Multiprocessor Systems

Sunondo Ghosh, *Student Member*, *IEEE Computer Society*,
Rami Melhem, *Member*, *IEEE Computer Society*,
and Daniel Mossé, *Member*, *IEEE Computer Society*

Abstract—Real-time systems are being increasingly used in several applications which are time critical in nature. Fault-tolerance is an important requirement of such systems, due to the catastrophic consequences of not tolerating faults. In this paper, we study a scheme that provides fault-tolerance through scheduling in real-time multiprocessor systems. We schedule multiple copies of dynamic, aperiodic, nonpreemptive tasks in the system, and use two techniques that we call *deallocation* and *overloading* to achieve high acceptance ratio (percentage of arriving tasks scheduled by the system). This paper compares the performance of our scheme with that of other fault-tolerant scheduling schemes, and determines how much each of deallocation and overloading affects the acceptance ratio of tasks. The paper also provides a technique that can help real-time system designers determine the number of processors required to provide fault-tolerance in dynamic systems. Lastly, a formal model is developed for the analysis of systems with uniform tasks.

Index Terms—Real-time scheduling, fault-tolerance, operating systems, primary/backup, reliability, redundancy.

1 Introduction

In recent years, computing systems have been used in several applications which have stringent timing constraints, such as autopilot systems and satellite and launch vehicle control. *Real-time systems* are systems whose correctness depends not only on their logical and functional behavior, but also on the temporal properties of this behavior. They can be classified as *hard* real-time systems, in which the consequences of missing a deadline may be catastrophic, and *soft* real-time systems, in which the consequences are relatively milder. Examples of hard real-time systems are space stations, radar for tracking missiles, systems for monitoring patients in critical condition, etc. An example of soft real-time is on-line transaction processes used in airline reservation systems. If the tasks on these systems are not completed within a deadline, a penalty is assessed.

In many real-time applications, fault-tolerance is also an important issue. A system is *fault-tolerant* if it produces correct results even in the presence of faults [9]. Due to the critical nature of tasks supported by many real-time systems, it is essential that tasks complete before their deadlines even in the presence of processor failures. This makes fault-tolerance an inherent requirement of hard real-time systems.

In a multiprocessor system, fault-tolerance can be provided by scheduling multiple copies of tasks on different processors [15], [20], [21]. The *primary/backup* (PB) approach and the *triple modular redundancy* (TMR) approach are two basic approaches that allow multiple copies of a task to be scheduled on different processors [23]. One or more of these copies can be run to ensure that the task completes before its deadline. In TMR, multiple copies are usually run to achieve error checking by comparing results after completion. In the PB approach, if incorrect results are generated from the primary task, the backup task is activated. The PB methodology has the advantage of small hardware resource requirements, but is not capable of masking faults. It should be noted that for both approaches, all copies of a task must be scheduled within its timing constraints so that the correct results are generated before the task's deadline even in the presence of faults.

In this paper, we study techniques for providing fault-tolerance for nonpreemptive, aperiodic, real-time tasks using the PB approach. *Aperiodic* tasks are those which are activated only when certain events occur [29] and *nonpreemptive* tasks are those that cannot be interrupted during execution by other tasks. A good motivation for nonpreemptive scheduling can be found in [8]. Since the arrival times of aperiodic tasks are not known a priori, they have to be scheduled *dynamically* as they arrive. This contrasts with *static* systems in which the schedules of all tasks are predetermined and remain fixed while the system is in operation.

The thrust of this paper is to study ways of providing fault-tolerance for dynamic real-time tasks, and not to develop an advanced heuristic for the scheduling of the tasks themselves. As proof of concept, we use a simple slack-based dynamic scheduling algorithm to schedule the real-

- *S. Ghosh is with Honeywell Technology Center, MN65-2600, 3600 Technology Drive, Minneapolis, MN 55418-1006.*
  E-mail: sghosh@htc.honeywell.com.
- *R. Melhem and D. Mossé are with the Department of Computer Science, University of Pittsburgh, Pittsburgh, PA 15260.*
  E-mail: {melhem, mosse}@cs.pitt.edu.

Manuscript received Oct. 24, 1994.
For information on obtaining reprints of this article, please send e-mail to: transpds@computer.org, and reference IEEECS Log Number D95251.

time tasks. This algorithm can be easily replaced by a more complicated algorithm for dynamic scheduling [16], [25], [35], [37]. The simplicity of our algorithm allows us to concentrate on studying and analyzing the incorporation of fault-tolerance while scheduling real-time tasks.

1.1 Motivation

Although many control applications are strictly periodic, there are several examples of systems in which aperiodic nonpreemptive hard real-time tasks need to be scheduled while providing fault-tolerance. Below we describe some of them.

Let us consider a multiprocessor system that monitors the condition of several patients in the Intensive Care Unit of a hospital. An action has to be taken as soon as the patient's condition changes. For example, if patient's heartbeat rate decreases beyond a certain threshold, a corrective action must be taken, such as injecting a drug in the patient's IV. This action must be taken within a certain hard deadline. Such a system should ensure that the task is executed within its deadline even if a fault occurs in one of the processors. As a second example, consider space applications, where transient faults may occur in the on-board computers due to electromagnetic interference. As a third example, in flight control systems, the controllers often activate tasks depending on what appears on their monitor. If a fault occurs, the system should be able to recover before the deadline. More examples of such safety critical applications can be found in [14].

To summarize, whenever real-time tasks can be generated due to events external to the system, we need a scheduling algorithm that can dynamically schedule the tasks. When the system is critical in nature, we also need fault-tolerance in the schedule.

For dynamic systems, it is not possible to guarantee optimal performance if the arrival times are not known a priori [4]. Static approaches such as those presented in [15], [13] may not be suitable for many real-time dynamic systems where predictability is also required [24]. So the question we need to answer is, how can fault-tolerance be provided to hard real-time tasks when their arrival times are not known?

One approach is to ensure that there are enough processors in the system to schedule multiple copies of each task even at peak load. By scheduling multiple copies, fault-tolerance can be provided. For this approach, the minimum number of processors required to meet peak load needs to be determined. Finding this number given the system load and the characteristics of arriving tasks is one of the goals of this paper.

A second approach is to reject tasks as soon as they arrive if they cannot be guaranteed fault-tolerance and, at the same time, inform the user that the tasks are being rejected. The user can take an appropriate action following the rejection of the tasks. For instance, in the hospital example described earlier, if the system cannot monitor one more patient in critical condition due to the additional tasks being generated, a nurse may be required to monitor the patient instead. Once the system accepts the monitoring tasks for a patient, all tasks have to be executed within their timing constraints even in presence of faults. Similarly, if an airplane running on autopilot is experiencing wind turbulence, and the additional tasks generated due to the disturbance cannot be executed while providing fault-tolerance, then the pilot has the option of taking over manual control of some or all functions of the airplane's navigational system.

Another possible approach is to let the user specify whether fault-tolerance is essential or not. If it is not essential, fault-tolerance will be provided to as many tasks arriving into the system as possible. For example, in the PB approach, even if the backup for a task cannot be scheduled, we may allow only the primary copy of the task to be scheduled in the system. When the task is scheduled, we inform the user whether both copies of the task or only the primary has been scheduled. The user can take necessary precaution if only one copy of the task has been scheduled.

The first approach is static in nature but can provide fault-tolerance to dynamic real-time tasks provided that the worst case combination of their arrivals is known. The two other approaches are dynamic in nature and can adapt to a change in the parameters of arriving tasks. This paper studies these approaches and, to our knowledge, is the first that guarantees the fault-tolerance of dynamic nonpreemptive real-time tasks.

The remainder of the paper is organized as follows. In Section 2, we describe representative previous work done in the area of real-time fault-tolerant scheduling and existing fault detection mechanisms. In Section 3, we state the problem. In Section 4, we discuss the general strategy that we use to solve the problem. In Section 5, we present a model for scheduling uniform tasks in the system with fault-tolerance. In Section 6, we extend the model to nonuniform tasks and present an algorithm for fault-tolerant scheduling of these tasks on a multiprocessor system. Simulation results are presented in Section 7. Finally, in Section 8, we discuss future work and provide some concluding remarks.

2 Related Work

When a fault occurs, extra time is required during task execution to handle fault detection and recovery. For real-time systems in particular, it is essential that the extra time be considered and accounted for prior to execution. Methods explicitly developed for fault-tolerance in real-time systems must take into consideration the number and type of faults, and ensure that the timing constraints are not violated. In this section, we describe some representative efforts in this direction.

Fault-tolerance has typically been approached from a hardware standpoint, with multiple replicas of essential applications running on separate hardware components [33]. More recently, a hybrid approach was proposed to integrate software checks at the end of hardware computation cycles [11]. To achieve fault masking of permanent hardware faults, redundant concurrent tasks may be used to carry out the computations (synchronously or asynchronously) [12], [33]. Some approaches use groups of processes executing sequentially [15], [27], while others have the replicas execute in parallel [3], [11], [12].

Another way of providing fault-tolerance is through scheduling. Several scheduling algorithms have been developed for real-time tasks in preemptive and nonpreemptive

systems. Since the general problem of optimal scheduling of tasks on a uniprocessor or a multiprocessor system is NP-complete [6], different heuristics have been used to schedule real-time tasks with the aim of maximizing performance measures such as acceptance ratio and processor utilization. Among these heuristics, only a few have attempted to provide fault-tolerance while scheduling real-time tasks.

In [15], a fault-tolerant scheduling algorithm is proposed to handle transient faults. The tasks are assumed to be periodic, and the short (backup) copies of all tasks are scheduled on a uniprocessor system to guarantee minimum performance for each task. Thereafter, the algorithm attempts to maximize the number of primaries that can be executed in the system. One of the restrictions of this approach is that task periods are multiples of each other.

In [13], processor failures are handled by maintaining contingency or backup schedules. These schedules are used in the event of a processor failure. To generate the backup schedule, it is assumed that an optimal schedule exists and the schedule is enhanced with the addition of "ghost" tasks, which function primarily as standby tasks. Since not all schedules will permit such additions, the scheme is optimistic.

In [1], a best effort approach to provide fault-tolerance has been discussed in hard real-time distributed computing systems. A primary/backup scheme is used in which both the primary and the backup start execution simultaneously and if a fault affects the primary, the results of the backup are used. The scheme also tries to balance the workload on each processor.

In the area of multiprocessor systems, a fault-tolerant scheduling strategy for periodic tasks is described in [20], [21]. In this strategy, a backup schedule is created for each task in the primary schedule. The tasks are then rotated such that the primary and backup schedules are on different processors and do not overlap. Thus, it is possible to tolerate up to one processor failure in the worst case. In [21], the number of processors required to provide a schedule to tolerate a single failure is double the number of the non-fault-tolerant schedule.

In the papers described above, the algorithms are static, that is, information about all tasks are known before scheduling them. Also, these studies generally concentrate on the analysis of algorithms and very little work has been done to provide any experimental or simulation results.

A dynamic procedure for the rescheduling of failed tasks has been described in [30]; such procedure is only invoked *after* the detection of a processor failure. This reallocation procedure involves local preemption, as well as bidding and focused addressing as described in [36]. If, after the processor crash is detected, no processor can be found to accommodate the failed tasks, the task is said to be *lost*. This work also presents simulation results.

In this paper, we concentrate only on the fault-tolerant scheduling of *nonpreemptive* tasks in real-time systems. For this reason, in the above short survey, we have not discussed systems that have been built with the aim of providing real-time fault-tolerance but do not study scheduling specifically [10], [12], [34]}. For the same reason, we have not discussed scheduling techniques that provide fault-tolerance for preemptive real-time tasks [2], [22], [26].

3 The Fault-Tolerant Scheduling Problem

In this section, we describe the system, fault, and task models used in this paper. We also introduce the approach that we have used to schedule dynamic real-time tasks with fault-tolerance.

We consider a system which consists of *n* identical processors and we assume that there is a task scheduling processor (central controller[1]) which maintains a global schedule, or that a global shared memory with test and set instructions is used. An example of such a system is a single node of the Spring system [31].

Because faults need to be identified before being tolerated, error detection is essential. Therefore, our approach will make use of the error detection mechanisms that have been developed for various fault models [9], [19], [23]:

- *Fail-signal* processors, which immediately notify other processors of a detected fault.
- *Alarms* or *watchdogs* for the detection of timing failures, assuming synchronized clocks.
- *Signatures* that can be used for detection of hardware or software faults.
- *Acceptance Tests* (AT) or AT-like checks that test results for hardware or software faults.

A task is modeled by a tuple $T_i = \langle a_i, r_i, d_i, c_i \rangle$, where a_i is the arrival time, r_i is the ready time (earliest start time of the task), d_i is the deadline, and c_i is maximum computation time (also called worst case execution time). We also assume that the *window* of a task ($w_i = d_i - r_i$) is at least twice as large as the computation time. Without this assumption, it would be impossible to schedule both the primary and its backup within the task's time constraints. We define the *window ratio* to be the ratio of the task's window to its computation time. That is, $wr_i = \frac{d_i - r_i}{c_i}$.

We assume that tasks arrive dynamically in the system and are scheduled nonpreemptively as they arrive. For ease of presentation, we assume that tasks are independent, that is, have no precedence constraints. However, tasks with precedence constraints can also be scheduled by transforming the precedence graph into independent nodes with new ready times and deadlines [5]. For example, if a group of tasks has precedence constraints, then the first among them is scheduled first (using the group's ready time and deadline, and its own computation time as the timing constraints). Once the first task is scheduled, the ready time of all the remaining tasks in the group is now equal to the end time of the first task. Then the tasks which immediately follow the first task are scheduled, and this process is continued until all tasks are scheduled, or one of them cannot be scheduled. If any one task cannot be scheduled, the whole group is rejected.

Tasks scheduled on this system are guaranteed to complete if a processor fails at any instant of time and if a second processor does not fail before the system recovers from the first failure. If a completion guarantee cannot be assured

1. This assumption is made to simplify the presentation since the scheduling strategy can also be implemented with distributed schedulers.

for a given task, the task is rejected (i.e., the system does not try to schedule the task at a later time). Both permanent and transient faults are handled by our approach. We do not consider the problem of software faults or correlated component failures.

Even though we do not deal with faults in the scheduling processor in this paper, there are several ways in which the failure of that processor can be handled. For example, the scheduling processor itself can be duplicated if it is prone to failures. Another solution is to schedule backups for the scheduling process on some other processor in the system.

We address the fault-tolerant scheduling problem by using a primary/backup approach. When a task arrives into the system, two copies,[2] a *primary* and a *backup*, are scheduled on two different processors within the task's window. The backup copy of a task executes only if a fault is detected in the primary. Since we assume dynamic systems, it is possible to release resources reserved for backup copies of tasks as soon as the primary copies finish executing. In this manner, we are able to better estimate the system state (e.g., available free time) when new tasks are scheduled.

To evaluate the performance of our scheme, we compare our results with the results of the *single spare* method, in which one processor is allocated to be used as a spare. If a nonspare develops a fault, all tasks scheduled on the faulty processor are executed on the spare. We also measure the overhead of providing fault-tolerance, by comparing our system to a system that has no provision for fault-tolerance, which we call the *no FT* method. In the case of real-time systems, not providing fault-tolerance is undesirable due to the possibility of catastrophic consequences if a fault occurs. Our comparison of these schemes is based on the performance of each technique in terms of *acceptance ratio* and *resiliency*. We define acceptance ratio as the percentage of arriving tasks accepted by the system, that is, those scheduled with fault-tolerance. *Rejection ratio* is defined as (1 – *acceptance ratio*) and is the percentage of arriving tasks rejected by the system. Note that each accepted task has resources reserved for itself once it is accepted by the system and hence is guaranteed to execute. Finally, *resiliency* is defined as the ability of the system to recover from a fault and resume normal operation before the next fault occurs.

4 The Scheduling Strategy

Our scheduling scheme is based on the following observations:

1) In real-time systems, tasks must be memory resident at the time of execution since the time taken to fetch the tasks from secondary storage is usually not predictable.
2) A processor functioning as a spare will be idle throughout the life-time of the system if no faults occur.
3) For hard real-time systems, reservation of resources for backup copies must be ensured, but backup copies can have a different scheduling strategy than primaries.

2. For simplicity, our description assumes that a backup is a copy of a primary.

The first and second observations steered us away from the single spare processor approach. To comply with real-time constraints, the spare processor would have to maintain in main memory all tasks in the system. This is a necessary condition for tasks to be able to execute in a timely fashion in case a processor fails. This implies that either the scheduling algorithm must take into consideration the total memory requirements of all tasks, or it must consider the time to load tasks that are to be executed in case of failures. Therefore, it would be a strain on a particular processor if all tasks must be loaded on that processor's memory.

We also notice that in the dedicated spare processor approach, the spare processor is not used by any executing tasks during intervals of fault-free operation. This wasted processor time could be used by some other task that could be executed concurrently, if we can guarantee that the backup tasks will be executed when needed.

The third observation, namely that resources reserved for backup copies could be reutilized, motivated us to apply the following two techniques to achieve high acceptance ratio while providing fault-tolerance with low overhead:

- *backup overloading*, which is the scheduling of more than one backup in the same time slot on the same processor (overlapping of multiple backups).
- *backup deallocation*, which is the reclamation of resources reserved for backup tasks when the corresponding primaries complete successfully.

The primary and backup copies of a task T_i will be referred to as simply the *primary* (PR_i), and the *backup* (BK_i). The time intervals on which the primary and the backup copies are scheduled are called the primary and backup time *slots*, respectively. If backup copies of more than one task are scheduled to run in the same time slot on the same processor, that backup slot is said to be *overloaded*. The backups of up to $n - 1$ tasks running on different processors can be overloaded on the same slot if at most one processor can fail at a time. The concept of overloading will be explained and studied further in the following sections.

The advantages of our scheme over the single spare scheme are as follows:

- As we will show later (see Section 7), the performance of our scheme is better than that of the single spare scheme in terms of acceptance ratio.
- If no faults occur, the time interval used by the backups can be reutilized (backup deallocation). For example, if it is known that a backup will not be used, then the task scheduled after the backup can be executed instead. Meanwhile, if new tasks arrive, they can be scheduled after the currently running task.
- As mentioned earlier, with our scheme, all processors have uniform memory requirements. Therefore, our scheme provides symmetry with respect to memory requirements which may aid if reconfigurations are needed or desired.

Although there are advantages over the spare scheme, the overhead must also be evaluated. Note that scheduling primary as well as backup tasks does not significantly increase the running time of the scheduling algorithm. This time is proportional to the task window and the average

execution time of tasks. According to [17], [28], for many applications the window ratio of tasks is around 11, which leads us to believe that a good implementation of our scheme will not be costly.

The nomenclature we use in this paper is as follows: $p(PR_i)$ is the processor on which PR_i is scheduled to execute, $beg(PR_i)$ is the scheduled begin time of PR_i, $end(PR_i)$ is the scheduled end time of PR_i, and $S(PR_i)$ is the slot in which PR_i is scheduled, that is, $S(PR_i) = [beg(PR_i), end(PR_i)]$. The same notation is used for a backup BK_i.

Using this notation, the necessary conditions for tolerating a single fault in real-time systems are described as follows:

[C1] Both PR_i and BK_i must be scheduled within the task's window w_i and the slot for BK_i should be later than the slot for PR_i:

$$r_i \leq beg(PR_i) < end(PR_i) \leq beg(BK_i) < end(BK_i) \leq d_i$$

[C2] PR_i and BK_i must be scheduled on different processors (to tolerate permanent faults):

$$p(PR_i) \neq p(BK_i)$$

[C3] If PR_i and PR_j are scheduled on the same processor p_i, BK_i and BK_j cannot overlap on the same processor p_j (to tolerate permanent faults):

$$p(BK_i) = p(BK_j) \wedge S(BK_i) \cap S(BK_j) \neq \emptyset \Rightarrow p(PR_i) \neq p(PR_j)$$

C1 is required because both primary and backup copies must satisfy the task's timing constraints and because it is assumed that the backup is executed only after a failure in the primary is detected (the failure is detected when the primary finishes executing). The second part of this condition can be relaxed if backups are executed in parallel with the primary, as in [32]. However, in a dynamic system, it is preferable to conserve as much of the processor time as possible for tasks which might arrive in the future, and thus we prefer not to execute backups until a primary fails. C2 ensures that both primary and backup copies are not scheduled on the same processor. Without C2, both copies might miss their deadlines if the processor on which they are scheduled fails. Finally, without C3, it would not be possible to execute backups of both tasks if a failure occurs in the processor on which the primaries are scheduled. Clearly, for transient faults, C2-C3 are overly conservative, since both primary and backup copies can be scheduled on the same processor.

The *resiliency* of the system (as defined in Section 3) can be measured in terms of the time it takes for the system to be able to tolerate a second fault after the first fault occurs in a processor p_i. We will call this latency time the *time to second fault* (TTSF). This time is the maximum between the end time of backups for primaries scheduled on p_i (the backups could be on any processor), and the end time of the primary tasks on other processors with backups on p_i. For example, in Fig. 1, primary PR_1 is scheduled on p_1 and its backup BK_1 on p_2. If a fault occurs on p_1 before PR_1 executes, then we can tolerate a fault on p_2 only after BK_1 completes. If a second fault occurs on $p(BK_1)$ before $end(BK_1)$, both copies of T_1 would be lost. Using a similar logic, we can tolerate a second fault on p_3 only after PR_2 completes. The maximum of all such combinations gives us the time at which we can guarantee that a second fault will be tolerated in the system. This argument is formalized in the following theorem:

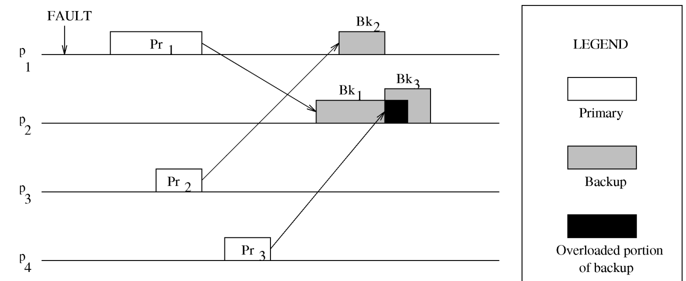


Fig. 1. Tolerating a second fault.

THEOREM 1. *If a permanent fault occurs at time t in processor p_i, the PB system will be able to tolerate another fault that occurs at a time t', where*

$$t' > max\{max_j\{end(BK_j) : p(PR_j) = p_i\}, max_j\{end(PR_j) : p(BK_j) = p_i\}\}$$

PROOF. If a permanent fault occurs at time t in p_i, any task arriving later than t will be scheduled (primary and backup) on the $n - 1$ nonfaulty processors. Thus, such tasks are guaranteed to complete even if a second fault occurs. If a task, T_j, is already scheduled when the first fault occurs then consider the following two cases:

1) Either $p(PR_j) = p_i$ or $p(BK_j) = p_i$: In this case, the restriction on t' guarantees that BK_j or PR_j, respectively, will successfully execute before a second fault occurs. The first part of the restriction on t' guarantees that the second fault can occur only after all backups with their primary on the faulty processor have executed. The second part ensures that the second fault occurs only after all primaries with their backups on the faulty processor have executed.
2) $p(PR_j) \neq p_i$ and $p(BK_j) \neq p_i$: In this case, T_j is guaranteed to complete even if a second fault occurs unless BK_j overlaps a backup BK_k whose primary PR_k is scheduled on p_i (for example, in Fig. 1, if $i = 1$, $j = 3$, and $k = 1$). Due to the first fault, BK_k is activated and hence BK_j cannot be used (since it overlaps BK_k). Therefore, a second fault cannot be tolerated on $p(PR_j)$ (p_4 in Fig. 1) until PR_j has executed. However, this case is covered by the second part of the restriction on t', according to which a second fault cannot be tolerated before BK_k has executed. Since BK_j and BK_k overlap, PR_j is scheduled earlier than $end(BK_k)$ in the system. This means that the second fault can be tolerated only after PR_j, which completes the proof for that case. □

In the next section, we present a uniform task model in which time is discretized and all tasks are of unit length. This model, which is amenable to a Markov analysis, is only an approximation of practical systems in which tasks

are nonuniform. In Section 6, we present a nonuniform task model in which time is continuous and all tasks have variable lengths. This model is not amenable to Markov analysis and thus we use simulations to evaluate it.

5 Scheduling Uniform Tasks

In this section, we assume that tasks have a uniform worst case execution time, $\forall i$, $c_i = 1$, and that all tasks that arrive within one unit of time are scheduled at the end of that time unit. With unit length tasks, backup slots may be easily overloaded since all backup tasks are of the same length. In fact, a simple backup preallocation policy is to reserve a slot for backup every n time slots on each processor, staggering backup slots on the n processors (Fig. 2). That is, if a backup slot is preallocated at time t on processor p_i, then a backup slot is preallocated at time $t + 1$ on processor $p_{(i+1) \bmod n}$. This preallocation allows for a simple assignment of backups to tasks in a way that satisfies Conditions C1-C3. Specifically, if a backup slot is preallocated at time t on p_i, then any task scheduled to run at time $t - 1$ on p_j, $j \neq i$, can use this slot as a backup. Because the task scheduled to run on p_i at time $t - 1$ cannot have its backup slot on the same processor (Condition C3), then this task can use the backup slot at time $t + 1$, which is on $p_{(t+1) \bmod n}$. In other words, for a task T_i, BK_i is scheduled in the time slot immediately after PR_i with probability $\frac{n-2}{n-1}$ and is scheduled two slots later than PR_i with probability $\frac{1}{n-1}$. Note that, in this scheme, n–1 backup tasks can potentially be overloaded on the same backup slot. Also, the $n - 1$ primary tasks that may be scheduled between two backup slots on a processor have their backups on different processors.

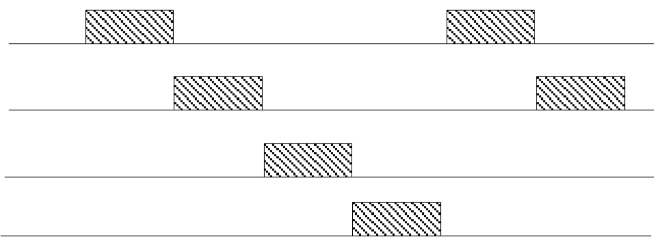


Fig. 2. Staggered backup slots in a multiprocessor system.

By staggering the backups, the equivalent of a spare processor is created using all n processors in the system. As mentioned in Section 4, the advantages of our scheme are: Memory requirements for processors are uniform, all processors are used leading to a higher acceptance ratio, and the task to be executed in lieu of a deallocated backup is known at scheduling time.

As an example of memory requirements in the uniform task case, let us consider a system which has four processors. Consider four consecutive unit time slots in the system. If one processor is used as a spare, that processor will have to maintain 12 tasks in its memory (for those four time units), while the other three processors will need four each. On the other hand, if backups are staggered, each processor will have to maintain six tasks in memory; three primary tasks, and three backup tasks.

The preallocation of backups described above may decrease the acceptance ratio of tasks since primary tasks may not be scheduled on slots reserved for backups. In order to estimate the loss of acceptance ratio caused by the addition of the fault-tolerance capability (the backup slots), we consider the simple first come first served scheduling of the primary tasks. Such a scheduling policy is equivalent to maintaining a task queue, Q, to which arriving tasks are appended. Given that $n - 1$ tasks can be scheduled on each time slot,[3] then the position of a task in Q indicates its scheduled execution time. If at the beginning of time slot t, a task T_i is the kth task in Q, then T_i is scheduled to execute at time slot $t + \left\lfloor \frac{k}{n-1} \right\rfloor$.

When a task T_i arrives at time t, its acceptance probability depends on the length of Q and on the window of the task, w_i. If T_i is appended at position u of Q and $w_i \geq \left\lfloor \frac{u}{n-1} \right\rfloor$, then the primary task, PR_i, is guaranteed to execute before time $t + w_i$. Otherwise, the task is not schedulable since it will miss its deadline. Moreover, if $w_i \geq \left\lfloor \frac{u}{n-1} \right\rfloor + 2$, BK_i is guaranteed to execute before $t + w_i$.

The dynamics of the above system may be modeled by a Markov process. For simplicity of presentation, we start by modeling a system without deadlines, that is, a system in which no tasks are rejected. Such a system may be modeled by a linear Markov chain in which each state represents the number of tasks in Q and each transition represents the change in the length of Q in one time unit.

We assume that a maximum of A_{max} tasks can arrive into the system every unit of time and that $P_{ar}(k)$ is the probability that k tasks arrive within a given unit of time. The probabilities of the different transitions may be calculated from the rate of task arrival. Specifically, if S_u represents the state in which Q contains u tasks and $u \geq n - 1$, then the probability of a transition from S_u to $S_{u_(n_1)+k}$ is $P_{ar}(k)$ for $k = 0, \ldots, A_{max}$. This is because during a time unit, $n - 1$ tasks are consumed from the queue and k new tasks arrive with probability $P_{ar}(k)$. If $u < n - 1$, then only u tasks can be consumed and $P_{ar}(k)$ becomes the probability of a transition from S_u to S_k. For example, Fig. 3 shows the transitions in the Markov chain assuming that $n = 4$, $A_{max} = 6$, and the arrivals are uniformly distributed.

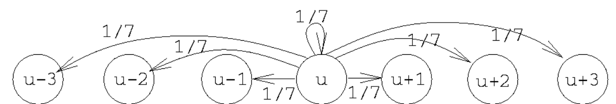


Fig. 3. Transitions out of state S_u for a linear chain.

Now we consider the case of tasks with deadlines. When the k arriving tasks have finite window sizes, some of these tasks may be rejected. Let $p_{u,k}$ be the probability that one of

3. One slot is reserved for backups.

the k tasks is rejected when the queue size is u. The value of $p_{u,k}$ is the probability that the window of the task is smaller than $\left\lfloor \frac{u+k/2}{n-1} \right\rfloor + \delta$, where δ is the extra time needed to schedule the backup and is equal to 1 or 2 with probability $\frac{n-2}{n-1}$ and $\frac{1}{n-1}$, respectively. Hence, when the queue size is u, the probability, $P_{rej}(r, k, u)$, that r out the k arriving tasks are rejected is $P_{rej}(r, k, u) = C_r^k (p_{u,k})^r (1 - p_{u,k})^{k-r}$, where C_r^k is the number of possible ways to select r out of k objects.

Transitions out of state S_u (current queue length is u) are to states $S_{u-n'}, \ldots, S_{u-1}, S_u, \ldots, S_{u+A_{max}-n'}$, where n' is the number of tasks consumed: $n' = n - 1$ if $u \geq n - 1$ and $n' = u$ otherwise. The probability of transition from S_u to $S_{u_n'}$ is the probability that no task arrives or all incoming tasks are rejected. This probability is equal to $\sum_{i=0}^{A_{max}} P_{rej}(i, i, u)$. In general, the transition from state S_u to state S_{u+i} occurs if j tasks arrive, $i + n' \leq j \leq A_{max}$ and $r = j - n' - i$ of these tasks are rejected thus adding $n' + i$ newly accepted tasks to the system. This increases the queue length by i since n' tasks are consumed from the queue. Hence the probability of transition from S_u to S_{u+i} is:

$$P[u \to u + i] = \sum_{j=i+n'}^{A_{max}} P_{ar}(j) \times P_{rej}(j - n' - i, j, u) \quad for\ i = -n', \ldots, A_{max} - n' \qquad (1)$$

Once the transition probabilities are known, we can calculate the steady state probabilities $P[u]$ of being in state u, $0 \leq u \leq M$, where $M = (n - 1)w_{max}$ is the largest possible value of the queue. When the steady state probabilities of being in each state are known, we can calculate the rate of rejection of tasks using the following equation:

$$Number\ of\ rejected\ tasks = \sum_{i=0}^{A_{max}} i \times P_{ar}(i) - \left(\sum_{u=0}^{n-1} u \times P[u] + \sum_{u=n}^{M} (n-1) \times P[u] \right) \qquad (2)$$

where the first part of the equation's right side is the average number of tasks arriving into the system per unit time, and the second part is the average number of tasks consumed (executed) by the system per unit time.

Up to now, we have not considered backup deallocation in the model. Backup deallocation means that if at time t no fault has occurred, then the backup preallocated at time slot $t + 1$ may be used to schedule a new task. In other words, if k tasks arrive during slot t, and $k > 0$, then one of these tasks can be scheduled in the deallocated backup slot, and the remaining $k - 1$ tasks can be treated as above. If we define an effective arrival rate P'_{ar} as $P'_{ar}(0) = P_{ar}(0)$ and $P'_{ar}(k) = P_{ar}(k + 1)$ for $k \neq 0$, then we can compute the transition probabilities from (1) with P_{ar} replaced by P'_{ar} and compute the number of rejected tasks from (2) with P_{ar} replaced by P'_{ar}.

In Fig. 4, we consider P_{ar} and P_{win} (probability that an arriving task has a window w) to be uniformly distributed, and we plot the rate of task rejection for window ratios of 5 and 7 as a function of the number of processors, n, for the case $A_{max} = 2n$ (uniformly distributed) with and without backup deallocation. The decrease in rejection ratio due to backup deallocation is clear. Note that, from a schedulability point of view, dedicating one of the n processors as a spare is equivalent to staggering the backup slots among the n processors when these slots are not deallocated. Hence, Fig. 4 can be also looked at as a comparison between our strategy and the spare method.

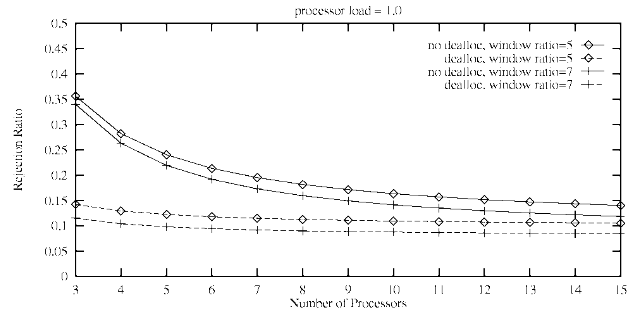


Fig. 4. Rejection ratio as a function of the number of processors.

6 Scheduling Nonuniform Tasks

6.1 Task Model and Scheduling

In this section, we enhance the model of the previous section with respect to the execution times of the tasks. We consider the case where execution times of tasks are variable and each task is scheduled as soon as it arrives. Since the execution times of tasks are not fixed, it is not possible to preallocate the backup tasks as in Section 5. Also, the analysis of such systems becomes more complex. Therefore, we will present a heuristic for the scheduling of tasks arriving dynamically, and evaluate this heuristic using simulations.

When a new task arrives, the primary is scheduled as early as possible within the task's window and the backup is scheduled after the primary but before the task's deadline. When a primary task completes successfully, its corresponding backup is deallocated. When the primary and backup are scheduled, or the primary finishes and the backup is deallocated, the list of existing slots is updated.

For example, consider the simple schedule for four tasks shown in Fig. 5. For these tasks it is assumed that $a_i = r_i$. Note that BK_1 and BK_3 partially overlap and BK_4 is not overloaded with BK_2 because d_4 is earlier than the begin time of BK_2, which was scheduled earlier when task 2 arrived. Moreover, the two backups should not overlap since their primary copies are scheduled on the same processor. PR_4 is scheduled on processor 2 because that is its earliest possible schedule.

A modification of the initial schedule is shown in Fig. 6, where the completion of tasks 2 and 1 (in that order) causes the deallocation of their respective backups. The arrival of tasks 5 and 6 (Fig. 7) causes further modifications in the schedule. BK_3 is overloaded with BK_5, and due to the deallocation of the BK_1, PR_6 can be scheduled on processor 2.

Our algorithm schedules a primary before scheduling its

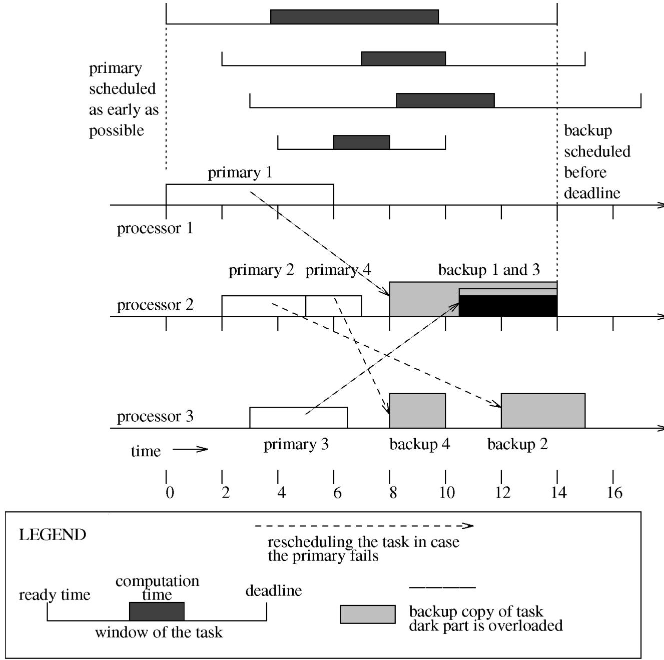


Fig. 5. Scheduling four tasks on three processors.

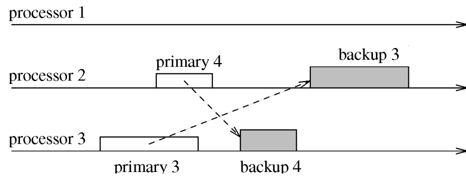


Fig. 6. After the completion of tasks and deallocation of respective backups.

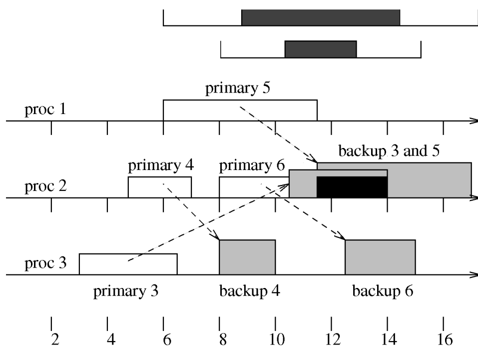


Fig. 7 The new schedule after the arrival of two more tasks.

corresponding backup because it is more difficult to schedule the primary than its backup and we want to minimize the number of constraints while scheduling the primary. Scheduling the backup is easier because we can overload it on any of the existing backups or simply schedule it on any available free slot. If the backup is scheduled first, we have two added constraints on the schedule for the primary: The end time of the primary has to be earlier than the begin time of the backup (as mentioned earlier) and there is one less processor for scheduling the primary (Condition C2).

The schedule that we choose for a primary may affect the acceptance probability of the backup for that task. For instance, let's assume that we find the earliest possible schedule of a primary on processor p_i and call this schedule s_1. It may not be possible to schedule a backup for the task using s_1 because there is space for the backup only on p_i. In order to solve this problem, we find the second earliest schedule for the same primary on another processor p_j and call this schedule s_2. If we cannot schedule the backup using s_1, we look at s_2. In s_2, we can schedule the backup on p_i (since the primary is now on p_j).

This solution, however, is not complete. Specifically, there are cases in which both s_1 and s_2 cannot be used to schedule the backup while the backup can be scheduled if the primary is scheduled on some processor other than p_i and p_j. To show that this solution is not complete, let us assume that s_1 and s_2 for PR_i have been found on processors p_1 and p_2, and the interval within which BK_i can be scheduled on all processors contains backups for primaries scheduled on p_1 and p_2. In this case, if we use s_1 or s_2 to schedule PR_i on p_1 or p_2, we will have to overload BK_i on one of the existing backups and violate Condition C3 to overload BK_i on any of the processors. However, we can schedule PR_i on p_3 and overload BK_i on either p_1 or p_2 without violating Condition C3.

The solution is complete only if we find a schedule for the primary on every processor and then try to schedule the backup for each primary schedule. Since keeping track of all possible schedules for the primary would increase the scheduling costs, we have considered in our simulation only the earliest and the second earliest schedules of the primary when trying to schedule the backup.

6.2 Steps to Schedule the Task

In this section, we present the scheduling algorithm in macrosteps.

The primary for task i is scheduled as follows: We look at each processor to find if PR_i can be scheduled between r_i and d_i. If PR_i cannot be scheduled without overlapping another time slot, $slot_j$, then we have to check if we can reschedule $slot_j$. This can be done by checking the *slack* of $slot_j$. The slack is defined as the maximum time by which the start of a task can be delayed so that it completes executing before its deadline. The function used for slack is:

$$Slack(PR_i) = min\,(Beg(S_{i+1}) + Slack(S_{i+1}),\, Beg(BK_i)) - End(PR_i)$$

$$Slack(BK_i) = 0$$

where S_{i+1} is the primary or backup slot following PR_i in p. If the slack of $slot_j$ added to the preceding free slot is greater than c_i, then PR_i can be scheduled after shifting $slot_j$ forward.

To schedule the backup, we may have two choices: Schedule it as late as possible or overload it as much as possible within the task's deadline. If we schedule the backup as late as possible, then we maximize the chances that the slot occupied by the backup in the schedule is reutilized. This is because the backup is deallocated as soon as the primary completes and since the backup is scheduled as far away from the primary as possible, there is more time available after it is deallocated, and new tasks arriving may be able to reutilize that space. In this situation, acceptance ratio increases. If the option of overloading is not available

(no previous backups have been already scheduled), then we always prefer to schedule as late as possible.

The other option is to overload the backup as much as possible. This also increases acceptance ratio since the processor time being reserved for backups is minimized and more time is available to schedule new tasks. However, if both options are available to us, we may not be able to do both at the same time. If we try to maximize overloading, then the backups may not be scheduled as late as possible and vice versa. To find out which factor (overloading or deallocation) is more effective in increasing acceptance ratio, we schedule each backup such that the following function is maximized:

$$\Phi = backup_end + \omega \times overlap_length \quad (3)$$

where *overlap_length* is the length of the intersection between the new backup and the backup(s) that are already scheduled. If there is no overloading, *overlap_length* is zero.

The user can choose the value of ω from 0 to infinity, which would yield scheduling disciplines ranging from scheduling the backups as late as possible to overloading as much as possible. At task scheduling time, the value of ω is fixed.

The backup for task i is scheduled as follows: Let the earliest schedule of primary i be on processor p_j. We examine the possible schedules for the backup on processors other than p_j (for permanent faults). Depending on the value of ω, we either overload the backup as much as possible, schedule it as late as possible, or follow a policy somewhere in the middle of these two extremes.

Although we maintain forward slacks of primary slots and we allow them to be moved forward if necessary, backup slots are not moved. That is because the backup slot may be supporting more than one primary and if the backup slot is moved, the slacks of all those primaries will change. This may have a cascading effect and each movement of a backup slot will be very costly in terms of time spent to recalculate slacks. If it is not possible to schedule the backup for task i's earliest schedule, we look at task i's second earliest schedule and again try to schedule the backup.

Finally, the task is committed as follows: Once the schedules for both the primary and the backup have been found, we commit the task. That is, we guarantee that the task will be completed before its deadline even in the presence of a single fault. To do this, we have to insert the primary and the backup in a global schedule being maintained by the central controller. Note that the slacks need to be recalculated only on the two processors on which the new slots were scheduled.

To summarize, the following steps are used to schedule the task:

1) Find a schedule for the primary as early as possible in the task's window.
2) Find a schedule for the backup corresponding to the primary schedule found in step 1 using the chosen heuristic (e.g., as late as possible or maximize overloading).
3) Commit the task (guarantee it will meet its deadline in the presence of a single fault).

6.3 An Example

A simple schedule for four tasks is shown in Fig. 8. The first three tasks[4] ($T_1 = \langle 0, 0, 6, 15\rangle$, $T_2 = \langle 1, 1, 4, 17\rangle$, $T_3 = \langle 1, 1, 3, 9\rangle$) are scheduled on p_1, p_2, and p_3, while their backups are scheduled in p_2, p_3, and p_1, respectively. Note that we chose to schedule these tasks according to their latest completion time, that is, their deadlines. Now consider a fourth task, $T_4 = \langle 2, 2, 6, 20\rangle$: BK_4 is scheduled in processor p_4, since it is the earliest possible schedule for that task. Thereafter, we attempt to schedule its backup, BK_4.

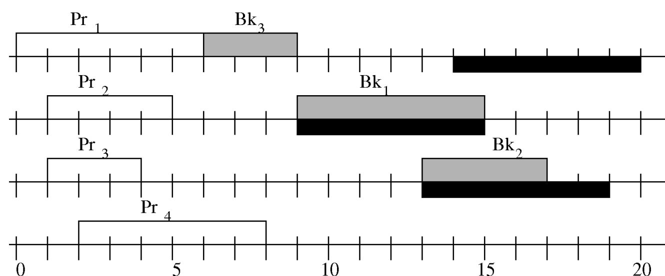


Fig. 8. Scheduling four tasks on four processors. The black boxes depict possible schedules of BK_4.

Note that the backup can be scheduled on processors p_1, p_2, and p_3. The question now is at what time, and in which processor to schedule the backup? We show that it depends on the parameter ω of the algorithm. We base our example on (3). The results of the tasks and their corresponding Φs are shown in Table 1. We select specific positions for BK_4 (complete, partial, and no overload) to illustrate the possibilities and how the heuristic works. In the table, the value of Φ in bold face is chosen.

TABLE 1
VALUE OF Φ WHEN SCHEDULING IN DIFFERENT POSITIONS OF DIFFERENT PROCESSORS, WITH DIFFERENT VALUES OF ω

proc	Φ (from (3))	$\omega = 0$	$\omega = 0.5$	$\omega = 10$
p_1	$14 + \omega \times 0$	**14**	14	14
p_2	$9 + \omega \times 6$	9	12	**69**
p_3	$13 + \omega \times 4$	13	**15**	53

Notice that for each value of ω, the location of the backup is different. There are ranges of values of weight that would not influence the scheduling of the backups. However, these situations depend on the task set and the current schedule. For example, when $\omega = 1$, scheduling BK_4 either at times 13 or 14 on processor p_3 would yield the same value of Φ.

In this algorithm, the maximum number of comparisons that have to be done for any new task depends on the number of tasks already scheduled in the system. In the average case, this depends on the average window ratio of the tasks in the system. The larger the window ratio, the more the number of tasks scheduled in the system, and hence the larger the number of potential comparisons. The average number of comparisons for each task is

4. Remember that a task is represented by the arrival time, ready time, computation time, and deadline.

TABLE 2
PARAMETERS FOR SIMULATIONS

parameter	name	distribution	values assumed
number of tasks	T	fixed	1,000
number of processors	n	fixed	3, 4, …, 20
computation time	c	uniform	mean = 5
processor load	γ	uniform	mean = 0.5, 0.6, …, 1.0
interarrival time	α	uniform	mean = $c/(\gamma * n)$
window ratio	wr	uniform	mean = 3, 5, 7, 9, 11
weight	ω	fixed	0, 0.1, 0.5, 1, 2, 5, 10, 20, 30

average window ratio of tasks $\times$ *number of processors*

In the worst case, we may have to try to fit the incoming task between each pair of existing slots already scheduled in the system. In this case, the total number of comparisons is equal to the number of tasks present in the system. This number is bounded if we assume that the rate of incoming and outgoing tasks (tasks that have completed execution) is the same on an average. The interval within which we need to compare the incoming task with scheduled tasks will shift forward with time, but the average average number of tasks in that interval will remain constant.

7 THE SIMULATION

To study the scheduling algorithm presented above, we have performed a number of simulations. The goals of our simulations are as follows:

- To compare the acceptance ratio of our fault-tolerant algorithm (described in Section 6.2) with that of the *no FT* method. The fault-tolerance capability in the schedule generated by our algorithm comes at the cost of increasing the number of rejected tasks. This comparison is used to measure the cost of increased rejection.
- To compare the acceptance ratio of our scheme with that of the *single spare* method. Note that the spare method is equivalent to having backup copies of all the primaries implicitly scheduled on the spare processor.
- To find the number of processors needed to schedule two copies of each task on the system, given the system load and task characteristics. This result is useful for statically determining the number of processors required for a given set of tasks in a real-time system.
- To measure the effect of weight ω on acceptance ratio. This tells us whether it is preferable to overload a backup more, or to schedule it as late as possible to increase the benefits of deallocation.
- To measure the time at which the system can tolerate a second fault. This tells us how quickly the system can completely recover from a fault, which is a measure of the resiliency of the system.

7.1 Simulator and Simulation Parameters

Our evaluation was done by implementing a discrete-event simulator where the events driving the simulation are the arrival, start, and completion of a task as well as occurrence of faults. We generated task sets for the computation of the schedules and ran the different policies on the same task sets. Note that, in the *spare* and the *no FT* methods, the tasks are scheduled using the same algorithm as the one used for primary copies in our method. Besides the number of processors n and the weight ω, the simulation parameters that can be controlled are:

- the **average computation time**, c: The computation time of the arriving tasks is assumed to be uniformly distributed with mean c.
- the **processor load**, γ: This parameter represents the average percentage of processor time that would be utilized if the tasks had no real-time constraints and no fault-tolerance requirements. Larger γ values lead to smaller task interarrival time. Specifically, the interarrival time of tasks is assumed to be uniformly distributed with mean $\alpha = c/(\gamma * n)$.
- the **average window ratio**, wr: This parameter (defined in Section 3) is uniformly distributed with a mean of wr and its minimum value is 2.

We ran simulations for task sets of 1,000 tasks. For each set of parameters we generated 100 task sets and calculated the average of the results generated. To be consistent with the model, we assumed $\forall i$, $r_i = a_i$, yielding a dynamic system. Formally, $a_1 = r_1 = 0$ and $r_i = r_{i_1} + \alpha_i$, where α_i is the interval between two successive arrivals. The processor load ranges from zero to one (i.e., $0 < \gamma \leq 1$). For example, if $\gamma = 1$, $n = 4$, and $c = 4$, then the task interarrival rate is 1. This means that, on an average, one task arrives in the system every unit of time and thus the processor load on each of the four processors is 1. These parameters are summarized in Table 2.

To compute the TTSF, a fault is injected at an arbitrarily chosen time instant, t. Theorem 1 is then applied to compute t'. We repeat this experiment 1,000 times and average the results to obtain the mean TTSF.

7.2 Results and Analysis

In Fig. 9, we plot the rejection ratio of tasks for our method as a function of processor load while keeping the number of processors in the system constant. As expected, the rejection ratio increases with the increase in processor load. We can also see that, due to the PB approach used and our deallocation technique, the rejection ratio decreases as the window ratio increases.

In Fig. 10, we compare the rejection ratio of three schemes, namely the spare scheme, our scheme, and the no-fault-tolerance (*no FT*) scheme for different number of processors. In the *no FT* scheme, no backups or spares are used; in the spare scheme, the primaries are scheduled by the same algorithm used in our scheme and backups are implicitly scheduled on the spare processor. Note that our scheme consistently performs better than the *single spare* scheme, for all values of window ratios. This is mainly because our algorithm uses deallocation. This tells us that for dynamic systems, it is less costly,

in terms of acceptance ratio, to provide fault-tolerance using the PB approach than to use the *single spare* approach.

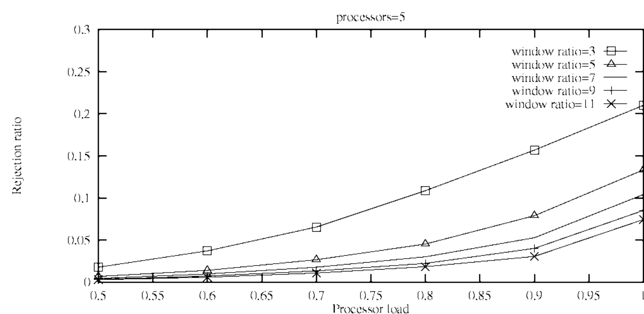


Fig. 9. Rejection ratio as a function of processor load, for different windows ratios.

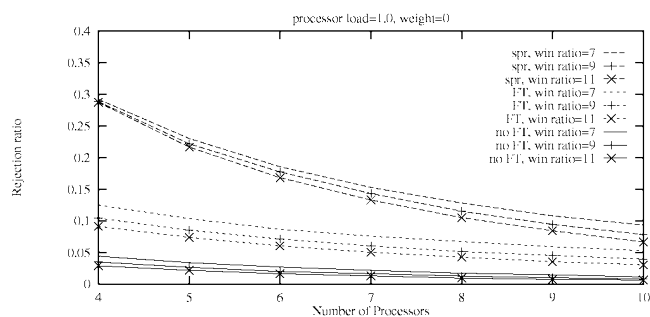


Fig. 10. Rejection ratio as a function of processor load, for spare, our, and no FT schemes.

The rejection ratio of our algorithm is very close to the rejection ratio of the *no FT* method. This means that the price paid to provide fault-tolerance is quite low. This low cost is due to the use of deallocation as explained in the previous paragraph, and overloading, which minimizes the amount of resources reserved for the backups.

We have also studied the effect of different values of ω (weight) on the acceptance ratio. For small window ratios, the optimal weight is between 0 and 1, while for larger window ratios, the maximum acceptance ratio is obtained when $\omega = 0$. However, the variation for different weights is relatively small. For instance, for window ratio equal to 3 and processor load equal to 1.0, the rejection ratio for four processors, varies from 0.2461 (at $\omega = 0$) to 0.2814 (at $\omega = 20$). For higher values of window ratio, the difference is even smaller.

When we consider the weight in conjunction with the Time To Second Fault (TTSF in Fig. 11), we can see that, although the weight has little influence on TTSF, the larger the window ratio, the larger the TTSF is. This is expected, since the larger window ratio increases the length of the schedule and therefore the recovery interval.

The small sensitivity of the rejection ratio and the TTSF to the weight indicates that the position of the backup task has little influence on the acceptance ratio of tasks. This means that a simple strategy for scheduling backups should be used rather than a complex policy that increases the run time of the scheduling algorithm. In fact, noting that the best results are obtained when $\omega = 0$, the simple policy of scheduling backups as late as possible is recommended.

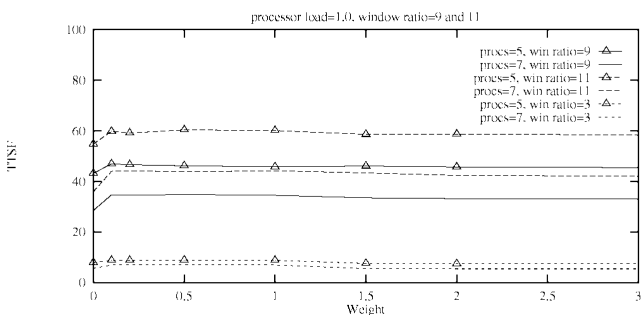


Fig. 11. TTSF versus weight for different window ratios and number of processors.

In the worst case, the TTSF will be equal to the maximum possible window of all tasks arriving into the system. If the TTSF is much smaller than the Mean Time To Failure (MTTF) of the processors, then we can safely say that we can handle multiple faults in the system.

Next we plot the rejection ratio for different values of *system load* (Fig. 12). The system load is a fixed parameter that is inversely proportional to the interarrival rate ($sysload = \frac{c}{\alpha}$). It can be computed as a product of the processor load and the number of processors ($sysload = \gamma \times n$). For example, if system load is 2, then the processor load is 1 on two processors, 0.67 on three processors, or 0.5 on four processors. This system load can be used to statically determine the number of processors needed to support dynamic real-time tasks while providing fault-tolerance. For example, in Fig. 12, we see that for a constant system load, the rejection ratio decreases as the number of processors increases. If the user requires a rejection ratio below 5% and the system load is 4, it can be seen from the graph that five processors are needed when the average window ratio is greater than 5, and six processors are needed when the average window ratio is 3. Similarly, the number of processors needed can be determined for any given rejection ratio and task characteristics.

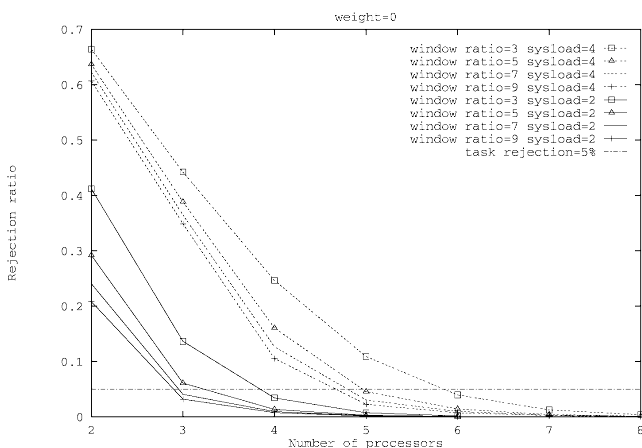


Fig. 12. Rejection ratio for different number of processors with varying system loads.

In Fig. 13, we show the effect of using overloading and deallocation in the acceptance ratio of task sets plotted

against the number of processors. The figure shows the varying rejection ratios when we use overloading, deallocation, both, and neither techniques. Note that whenever overloading is present ("both" and "overloading only"), we additionally compare the results for $\omega = 0$ and $\omega = 100$.

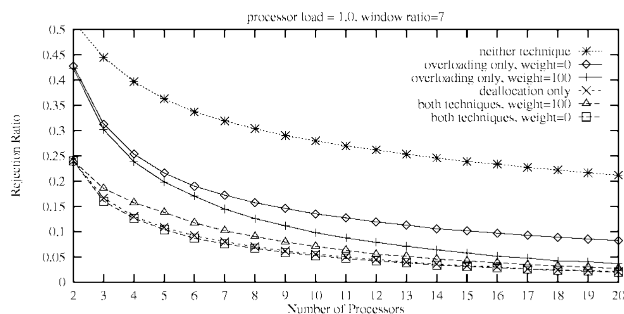


Fig. 13. Rejection ratio as a function of number of processors, using different techniques.

It is clear that the use of either method significantly improves the acceptance ratio of task sets, but the difference from using deallocation to using both techniques shows a less drastic improvement. This is because deallocation, in conjunction with scheduling the backups as late as possible, causes almost all the time reserved for backups to be reused. Thus, all processors are being utilized to their fullest extent possible while scheduling backups for each primary. Overloading cannot help much more in this case. This leads us to believe that using deallocation in the scheduling of real-time fault-tolerant tasks will enhance the acceptance ratio, but not much is gained from using overloading along with deallocation. Although we do not show the results here, similar behavior is exhibited when the processor load is smaller or larger than 1.0.

8 Concluding Remarks

In this paper, we have developed and analyzed a fault-tolerant scheduling method for real-time systems that tolerates processor faults. We show that the overloading and deallocation of backup slots provide efficient utilization of the resources. Our results show positive correlation between the acceptance ratio of task sets and the load on the system, as well as between the acceptance ratio and the average task window ratio. Both theoretical and simulation results indicate that the reclaiming of resources reserved for backup tasks (deallocation) is the most important factor when scheduling tasks in a primary/backup fashion. With backup deallocation, elaborate methods for increasing the overloading of backups seem to have only a small effect on acceptance ratio and resiliency. Thus, fast and simple scheduling algorithms should be used for backups.

Our method can tolerate more than one processor failure. As it stands, the scheme can tolerate successive permanent faults that are separated by a sufficiently large time interval. Once the time to second failure (TTSF) of the system is established, it is easy to relate the TTSF to the mean-time-to-failure (MTTF) and the reliability of the processors. The goal is to guarantee that within some fraction of MTTF, all tasks existing at the time of the failure complete. The new tasks arriving after the first failure will have their primary and backup copies scheduled on the nonfaulty processors and thus a second fault can be handled.

To handle multiple simultaneous faults, we can schedule more than one backup copy for each task. In this case, resiliency and overhead of the system will increase. Note that, although the scheduling policy for this case will be different from the one presented in this paper, the mechanisms we have developed remain the same.

We are currently extending this scheduling algorithm to distributed systems in which communication costs are taken into account. We plan to study a hybrid technique combining the PB approach and triple modular redundancy on multiprocessor systems. That is, scheduling more than two copies of each task in the system. We also plan to study the fault-tolerance scheduling of tasks that have precedence and resource constraints.

Acknowledgments

The basic concepts in this paper have been presented in [7] and a preliminary version of the analysis has appeared in [18]. Sunondo Ghosh was supported by U.S. National Science Foundation RIA grant CCR-9308886. Rami Melhem was supported in part by DARPA under contract DABT63-96-C-0044. Daniel Mossé was supported in part by U.S. National Science Foundation RIA grant CCR-9308886 and in part by DARPA under contract DABT63-96-C-0044.

References

[1] S. Balaji, L. Jenkins, L.M. Patnaik, and P.S. Goel, "Workload Redistribution for Fault-tolerance in a Hard Real-Time Distributed Computing System," *Proc. IEEE Fault-tolerance Computing Symp. (FTCS-19)*, pp. 366-383, 1989.

[2] A.A. Bertossi and L.V. Mancini, "Scheduling Algorithms for Fault-Tolerance in Hard-Real-Time Systems," Technical Report TR-15/91, Univ. of Pica, Corso, Italy, 1991.

[3] E. Cooper, "Replicated Distributed Programs," *Proc. 10th ACM Symp. Operating System Principles*, pp. 63-78, Dec. 1985.

[4] M.L. Dertouzos and A.K. Mok, "Multiprocessor On-Line Scheduling of Hard-Real-Time Tasks," *IEEE Trans. Software Eng.*, vol. 15, no. 12, pp. 1,497-1,506, Dec. 1989.

[5] M. DiNatale and J. Stankovic, "Dynamic End-to-End Guarantees in Distributed Real-Time Systems," *Proc. Real-Time Systems Symp.*, 1994.

[6] M.R. Garey and D.S. Johnson, *Computers and Intractability, a Guide to the Theory of Completeness.* San Francisco: W.H. Freeman, 1979.

[7] S. Ghosh, R. Melhem, and D. Mossé, "Fault-Tolerant Scheduling on a Hard Real-Time Multiprocessor System," *Proc. Int'l Parallel Processing Symp.*, Apr. 1994.

[8] K. Jeffay, D.F. Stanat, and C.U. Martel, "On Non-Preemptive Scheduling of Periodic and Sporadic Tasks," *Proc. IEEE Real-Time Systems Symp.*, pp. 129-139, Dec. 1991.

[9] B.W. Johnson, *Design and Analysis of Fault Tolerant Digital Systems.* Addison Wesley, 1989.

[10] R.M. Kieckhafer, C.J. Walter, A.M. Finn, and P.M. Thambidurai, "The MAFT Architecture for Distributed Fault-tolerance," *IEEE Trans. Computers*, vol. 37, no. 4, pp. 398-405, Apr. 1988.

[11] K.H. Kim and A. Damm, "Fault-tolerance Approaches in Two Experimental Real-Time Systems," *Proc. Seventh Workshop Real-Time Operating Systems and Software*, pp. 94-98, May 1990.

[12] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, and R. Zainlinger, "Distributed Fault-Tolerant Real-Time Systems: The MARS Approach," *IEEE Micro*, vol. 9, no. 1, pp. 25-40, Feb. 1989.

[13] C.M. Krishna and K.G. Shin, "On Scheduling Tasks with a Quick Recovery from Failure," *IEEE Trans. Computers*, vol. 35, no. 5, pp. 448-455, May 1986.

[14] J.H. Lala and R.E. Harper, "Architectural Principles for Safety-Critical Real-Time Applications," *Proc. IEEE*, vol. 82, no. 1, pp. 25-40, Jan. 1994.

[15] A.L. Liestman and R.H. Campbell, "A Fault-Tolerant Scheduling Problem," *IEEE Trans. Software Eng.*, vol. 12, no. 11, pp. 1,089-1,095, Nov. 1988.

[16] A.K. Mok and M.L. Dertouzos, "Multiprocessor Scheduling in a Hard Real-Time Environment," *Proc. Texas Conf. Computing Systems*, 1978.

[17] J.J. Molini, S.K. Maimon, and P.H. Watson, "Real-Time System Scenarios," *Proc. 11th Real-Time Systems Symp.*, pp. 214-225, Lake Buena Vista, Fla., Dec. 1990.

[18] D. Mossé, R. Melhem, and S. Ghosh, "Analysis of a Fault-Tolerant Multiprocessor Scheduling Algorithm," *Proc. 24th Int'l Symp. Fault-Tolerant Computing*, Austin, Tex., June 1994.

[19] S.K. Oh and G. MacEwen, "Toward Fault-Tolerant Adaptive Real-Time Distributed Systems," External Technical Report 92-325, Dept. of Computing and Information Science, Queen's Univ., Kingston, Ontario, Canada, Jan. 1992.

[20] Y. Oh and S. Son, "Multiprocessor Support for Real-Time Fault Tolerant Scheduling," *Proc. IEEE 1991 Workshop Architectural Aspects of Real-Time Systems*, pp. 76-80, San Antonio, Tex., Dec. 1991.

[21] Y. Oh and S. Son, "Fault-Tolerant Real-Time Multiprocessor Scheduling," Technical Report TR-92-09, Univ. of Virginia, Apr. 1992.

[22] M. Pandya and M. Malek, "Minimum Achievable Utilization for Fault-Tolerant Processing of Periodic Tasks," Technical Report TR 94-07, Univ. of Texas at Austin, Dept. of Computer Science, 1994.

[23] D.K. Pradhan, *Fault Tolerant Computing: Theory and Techniques.* Englewood Cliffs, N.J.: Prentice Hall, 1986.

[24] K. Ramamritham and J.A. Stankovic, "Scheduling Algorithms and Operating Systems Support for Real-Time Systems," *Proc. IEEE*, vol. 82, no. 1, pp. 55-67, Jan. 1994.

[25] K. Ramamritham and J.A. Stankovic, "Dynamic Task Scheduling in Hard Real-Time Distributed Systems," *IEEE Software*, pp. 65-75, July 1984.

[26] S. Ramos-Thuel and J.K. Strosnider, "The Transient Server Approach to Scheduling Time-Critical Recovery Operations," *Proc. Real-Time Systems Symp.*, pp. 286-295, San Antonio, Tex., Dec. 1991.

[27] B. Randell, "System Structure for Software Fault-tolerance," *IEEE Trans. Software Eng.*, vol. 1, no. 2, pp. 220-232, June 1975.

[28] R.L. Sedlmeyer and D.J. Thuente, "The Application of the Rate-Monotonic Algorithm to Signal Processing Systems," *Proc. Real-Time Systems Symp.*, Development Sessions, 1991.

[29] K.G. Shin and P. Ramanathan, "Real-Time Computing: A New Discipline of Computer Science and Engineering," *Proc. IEEE*, vol. 82, no. 1, pp. 6-24, Jan. 1994.

[30] J.A. Stankovic, "Decentralized Decision Making for Task Reallocation in a Hard Real-Time System," *IEEE Trans. Computers*, vol. 38, no. 3, pp. 341-355, Mar. 1989.

[31] J.A. Stankovic and K. Ramamritham, "The Spring Kernel: A New Paradigm for Real-Time Operating Systems," *ACM SIGOPS, Operating Systems Review*, vol. 23, no. 3, pp. 54-71, July 1989.

[32] T. Tsuchiya, Y. Kakuda, and T. Kikuno, "Fault-Tolerant Scheduling Algorithm for Distributed Real-Time Systems," *Proc. Third Workshop Parallel and Distributed Real-Time Systems*, 1995.

[33] M. Weber, "Operating Systems Enhancements for a Fault-Tolerant Dual Processor Structure for the Control of an Industrial Process," *Software Practice and Experience*, vol. 17, no. 5, pp. 345-350, May 1985.

[34] J.H. Wensley et al., "SIFT: Design and Analysis of a Fault Tolerant Computer for Aircraft Control," *Proc. IEEE*, pp. 1,240-1,255, Oct. 1978.

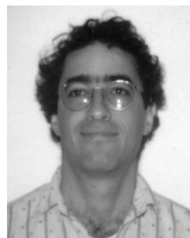
[35] C.M. Woodside and D.W. Craig, "Local Non-Preemptive Scheduling Policies for Hard Real-Time Distributed Systems," *Proc. Real-Time Systems Symp.*, pp. 12-16, 1987.

[36] W. Zhao and K. Ramamritham, "Distributed Scheduling Using Bidding and Focused Addressing," *Proc. IEEE Real-Time Systems Symp.*, pp. 103-111, Dec. 1985.

[37] W. Zhao, K. Ramamritham, and J.A. Stankovic, "Scheduling Tasks with Resource Requirements in a Hard Real-Time System," *IEEE Trans. Software Eng.*, vol. 13, no. 5, pp. 564-577, May 1987.

Sunondo Ghosh received a BTech degree in computer science and engineering from the Institute of Technology, Banaras Hindu University (IT-BHU), India, in 1991, and a PhD degree in computer science from the University of Pittsburgh in 1996. He is currently working in the Real-Time Execution Environments (RTEE) Group of Honeywell Inc. His interests include real-time systems and fault-tolerance. Dr. Ghosh is a member of the IEEE Computer Society.

Rami Melhem received a BE in electrical engineering from Cairo University in 1976, an MA degree in mathematics and an MS degree in computer science from the University of Pittsburgh in 1981, and a PhD degree in computer science from the University of Pittsburgh in December 1983. He is currently a professor of computer science at the University of Pittsburgh. Previously, he was an assistant professor at Purdue University and an assistant/associate professor at the University of Pittsburgh. He has published more than 100 papers in the areas of fault-tolerance, real-time systems, systolic architectures, parallel computing, and optical computing. He is the general chair of the Third International Conference on Massively Parallel Processing Using Optical Interconnections. He served on program committees of several conferences and workshops and he is on the editorial board of *IEEE Transactions on Computers*. He was a guest editor of a special issue of the *Journal of Parallel and Distributed Computing* on optical computing and interconnection systems. He is on the advisory boards of the IEEE Technical Committee on Parallel Processing and the IEEE Technical Committee on Computer Architecture. Dr. Melhem is a member of the IEEE Computer Society, the Association for Computing Machinery, and the International Society for Optical Engineering.



Daniel Mossé received his BS in math from the University of Brasilia, Brazil, in 1985, and an MS and PhD in computer science from the University of Maryland, College Park, in 1990 and 1993, respectively, and joined the faculty of the Department of Computer Science at the University of Pittsburgh in 1992. For his PhD, he was involved in the design and implementation of a distributed, real-time, fault-tolerant operating system prototype. His main research interests are in fault-tolerance, distributed resource allocation, real time, real-time communications, and multimedia. He is currently leading two groups in the implementation and installation of a fault-tolerant real-time operating system and a home-grown distributed reliable multimedia system.