Analysis of a Fault-Tolerant Multiprocessor Scheduling Algorithm*

Daniel Mossé, Rami Melhem, and Sunondo Ghosh
Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260
{mosse,melhem,ghosh}@cs.pitt.edu

Keywords: Fault Tolerance, Real Time, Operating Systems, Primary/Backup, Redundancy

Abstract

Fault tolerance is an important aspect of real-time computer systems, since timing constraints must not be violated. When dealing with multiprocessor systems, fault tolerance becomes an even greater requirement, since there are more components that can fail. In this paper, we present the analysis of a fault-tolerant scheduling algorithm for real-time applications on multiprocessors. Our algorithm is based on the principles of *primary/backup tasks, backup overloading* (i.e., scheduling more than a single backup in the same time interval), and *backup deallocation* (i.e., reclaiming the resources unused by backup tasks in case of fault-free operation). A theoretical model is developed to study a particular class of applications and certain backup and overloading strategies.

The proposed scheme can tolerate a single fault of any processor at any time, be it transient or permanent. Simulation results offer evidence of little loss of schedulability due to the addition of the fault tolerance capability. Simulation is also used to study the length of time needed for the system to recover from a fault (i.e., the time when the system is again able to tolerate any fault).

1 Introduction

In critical real-time applications, a reliable environment is required to guarantee that jobs will meet their deadlines despite the presence of faults. Avoiding faults is not always possible, since the system designer does not know when a fault will occur or what the fault will be. On the other hand, if a system takes precautionary measures to handle faults, then it can tolerate faults and still yield functionally and temporally correct results, thus providing the desired reliable environment.

Among the basic mechanisms for fault tolerance are checkpointing and component redundancy. The more complex the underlying computing system, the more difficult it becomes to design and implement such mechanisms. For instance, the number of faults likely to occur increases with the number of hardware and software components in the overall system. Moreover, in real-time applications, handling of faults has to take place within a limited interval of time and thus it is more difficult than in applications where time is not critical.

Fault-tolerant systems developed for real-time distributed applications are dedicated to one specific application, suffer substantial performance overheads, or require special hardware. In this paper, we study a general purpose fault-tolerant scheduling scheme for real-time distributed applications, which does not require special hardware. The general problem of optimal fault-tolerant scheduling of tasks is NP-complete [3], and therefore different heuristics have been used to schedule real-time tasks maximizing some criteria [2, 7, 15].

Our scheme is based on the primary/backup technique and proves to be best suited for low failure rates and for applications that have large laxity. The *primary/backup* (PB) approach allows multiple copies of a task to be scheduled on different processors [13]. One or more of these copies can be run to make sure that the task completes before its deadline. In the PB approach, the backup task is activated only if a fault occurs while executing the primary task. As a special case of the PB approach, a fault tolerant scheduling algorithm for periodic tasks is proposed in [6] to handle transient faults in a uniprocessor system. One of the restrictions of this approach is that the period of any task should be a multiple of the period of its preceding tasks.

Son and Oh describe a PB scheduling strategy for periodic tasks on multiprocessor systems [11, 12]. 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. In [12] the number of processors required to provide a schedule to tolerate a single failure is double the number of the non-fault-tolerant schedule.

Two other works have studied fault-tolerant PB scheduling [1, 5]. In [1], there is a description of a primary/standby approach, where the standby has execution time smaller than the primary tasks (as in [6]). Both primary and standby start execution simultane-

*This work is supported in part by NSF grant CCR-9308886 to the Department of Computer Science, University of Pittsburgh.

ously and if a fault affects the primary, the standby will send its results. On the other hand, [5] presents theoretical results assuming that an optimal schedule exists and enhancing that schedule with the addition of standby tasks. Not all schedules permit such additions.

Another approach for providing single fault tolerance in real-time systems is to dedicate a standby processor as a spare. A major disadvantage of such approach is that no tasks are scheduled on the dedicated spare processor.

In [4], we present a general approach in which application's software modules (tasks) are replicated across processors in a way that guarantees that all the tasks are executed within their deadlines despite some fixed, user-specified number of faults. The novel ideas in that approach are ***backup overloading*** and ***backup de-allocation***. In this paper we extend the scheme by introducing the notion of ***weighed overloading***, and we study the effect of this notion on the schedulability and resiliency of the system. We also present a Markov model to analyze the schedulability when tasks have unit execution time. For more general tasks, we evaluate our scheme by a suite of simulated tests. In these simulations we generate a synthetic workload, with deadlines, worst case computation times, etc, and simulate the scheduler and the execution. We then use several metrics to evaluate the performance of the algorithm, such as the rejection rate of our scheduler, the average time that a second fault may be tolerated, and the schedulability for different system parameters.

This paper is organized as follows: in Section 2 we give an overview of our fault-tolerant scheduling algorithm, which is fully described in [4]. In that section we also present some results about the fault-tolerant capability of this scheme. In Section 3, we describe the model for systems of tasks that have unit length, and present solutions for that model. In Section 4, we show the details of our fault tolerance approach and discuss the cost and trade-offs involved using simulation results. Future work and conclusions are presented in Section 5.

2 The Scheduling Scheme

We consider a system which consists of n interconnected identical processors and we assume that there is a task scheduling processor that maintains a global schedule. In our model, a task is 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). The tasks in our model are aperiodic and independent, that is, have no precedence constraints. We assume that $a_i = r_i$ and that the ***window*** of a task ($w_i = d_i - r_i$) is at least twice as large as the computation time, in order to make it possible to schedule both the task and its backup[1] within its time constraints. Tasks arrive dynamically in the system.

[1] Each backup task is a copy of the primary task, that is, has the same execution time and yields the same results.

We assume that only one processor can fail at any instant of time and that a second processor cannot fail before the system recovers from the first failure. We also assume that there exists some fault detection mechanism that detects a site crashes (e.g., fail-signal processors [10]). Both permanent and transient faults can be handled by our approach. We do not consider the problem of software faults or correlated component failures.

When a task arrives, if a schedule cannot be found for the task and its backup, it is rejected (i.e., the system does not try to schedule the task at a later time). To achieve high schedulability while providing fault-tolerance, our method applies two techniques while scheduling the primary and backup copies of the tasks:

- ***backup overloading*** which is scheduling backups for multiple primary tasks during the same time period in order to make efficient utilization of available processor time, and
- ***de-allocation*** of resources reserved for backup tasks when the corresponding primaries complete successfully.

The primary and backup copies of a task 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 the backup copies of more than one task are scheduled to run in the same time slot, 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.

For fault-tolerant purposes, reservations of resources for backup copies must be guaranteed. Backup copies, however, can have a different scheduling strategy than primaries. Note, however, that scheduling primary as well as backup tasks does not significantly increase the running time the scheduling algorithm. This time is proportional to the task window and the average execution time of tasks. According to [8, 14], for many applications the ratio window/computation of tasks is not more than 11, which leads us to believe that a good implementation of the scheme will not be costly. Furthermore, in terms of algorithm complexity, the cost of searching for a schedule for the backup or not searching is the same (since we search at most all tasks once).

In later sections of this paper, we will show that our scheduling approach achieves higher schedulability than the approach in which one of the processors in the system is dedicated as a spare processor. This is mainly because in the dedicated spare approach, the spare processor is not used by any executing tasks during the 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 if needed.

Another disadvantage of the spare approach is that, in real-time systems, tasks must be memory resident at the time of execution. The spare approach, thus, requires that all tasks be loaded in the local spare processor memory prior to the execution of the tasks. That implies that either the scheduling algorithm must take into consideration the total memory requirements by all tasks, or it must consider the time to load tasks that are to be executed in case of failures. Currently, none of the systems that we are aware of take into consideration the loading time. Therefore, it would be a strain on the system, if all tasks needed to be loaded up on one processor's memory.

Fault-Tolerant Scheduling Results

From the problem definition above it is easy to derive the following propositions. Note that Pr_i and Bk_i should be scheduled within r_i and d_i. The nomenclature we use here is as follows: $P(T)$ is the processor on which T is scheduled to execute where T is either Pr_i or Bk_i, $beg(T)$ is the scheduled begin time of T, $end(T)$ is the scheduled end time of T, and $S(T)$ is the time interval (slot) in which T is scheduled (i.e., $S(T) = [beg(T), end(T)]$).

Proposition 1 *For a primary/backup scheme, a task T_i is guaranteed to execute in the presence of one permanent fault if and only if $P(Pr_i) \neq P(Bk_i)$.*

Proof: If $P(Pr_i)$ fails, then $P(Bk_i)$ cannot fail and therefore $P(Bk_i)$ will execute successfully. However, if $P(Pr_i) = P(Bk_i) = p$, and p fails, then neither Pr_i nor Bk_i can execute successfully. □

For transient faults the proposition above is overly conservative. The only restriction on the system architecture for transient failures is to have the window large enough to accommodate both the primary and the backup. On the other hand, we must leave enough time in the window for the primary, backup and extra work, as follows:

Proposition 2 *To allow T_i to execute in the presence of any single fault, $beg(Bk_i)$ should be larger than or equal to $end(Pr_i) + \delta_i$, where δ_i is the time it takes to detect and communicate an error occurring while executing task T_i.*

Proof: By contradiction: Assume $end(Bk_i) = d_i$ and $end(Pr_i) = beg(Pr_j)$. Let a fault occur at time $end(Pr_i) - \epsilon$. To start Bk_i, it would take $\delta > \epsilon$ units of time. That means that the backup would end at time $end(Pr_i) - \epsilon + \delta_i + c_i > d_i$. □

From the following result we can see that there are some restrictions on the processors on which primary and backup tasks can be scheduled.

Theorem 1 *To allow any single fault to be tolerated, only backups of tasks scheduled on different processors can overlap. That is, $S(Bk_i) \cap S(Bk_j) \neq \emptyset$ implies $P(Pr_i) \neq P(Pr_j)$.*

Proof: By contradiction: Assume $P(Pr_i) = P(Pr_j)$ and let $end(Pr_i) = beg(Pr_j)$. Since the backups overlap, the backup slot is smaller than the sum of the two computation times: $end(Bk_j) - beg(Bk_i) < c_i + c_j$. Let a fault occur at time $end(Pr_i) - \epsilon$. Then $beg(S(Bk_i)) + c_i + c_j > d_j$. That means that $P(Pr_i) \neq P(Pr_j)$. □

Another result refers to the time it takes for the system to become fully resilient after a fault occurs in processor P_i. We will call this latency time the *time to second fault* (TTSF). This time is essentially the maximum between the end time of backups for tasks scheduled on P_i, and the end time of the primary tasks on other processors with backups on P_i.

Theorem 2 *If a permanent fault occurs at time t in processor P_i, the system will be able to tolerate another fault that occur 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: When a fault occurs at time t in P_i, any task arriving later than t will be scheduled (primary and backup) on the $n-1$ non-faulty processors. Thus, such a task is guaranteed to complete even if a second fault occurs. If a task, T_j, is already scheduled when the first fault occurs and $P(Pr_j) \neq P_i$ and $P(Bk_j) \neq P_i$, then such a task is also guaranteed to complete even if a second fault occurs. Finally, if either $P(Pr_j) = P_i$ or $P(Bk_j) = P_i$, then the restriction on t' guarantees that Bk_j or Pr_j, respectively, will successfully execute before a second fault occurs. □

The propositions and theorems above will guide the scheduling algorithm presented in the following sections.

3 Scheduling Uniform Tasks

We start the presentation of our solution with a restricted form of the general model presented in Section 2. Specifically, we restrict the tasks to have a uniform worst case execution time, $c_i = 1$. With unit length tasks, backup slots may be easily overloaded since all backup tasks are of the same length. In fact, a simple backup pre-allocation policy is to reserve a slot for backup every n time slots on each processor. Backup slots on the n processors can be staggered (Figure 1), that is, if a backup slot is pre-allocated at time t on processor P_i, then a backup slot is pre-allocated at time $t+1$ on processor $P_{(i+1) mod n}$. This pre-allocation allows for a simple assignment of backups to tasks in a way that satisfies Theorem 1 . Specifically, if a backup slot is pre-allocated 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 (Proposition 1), then this task can use the backup slot at time $t+1$, which is on $P_{(i+1) mod n}$. In other words, for a task T_i, Bk_i is scheduled immediately after Pr_i with probability (n-2)/(n-1) and is scheduled two slots later than Pr_i with probability 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, P_i have their backups on different processors.

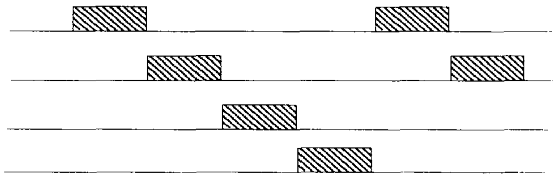

Figure 1: Staggered backup slots in a multiprocessor system

The slot pre-allocation for backups described above may decrease the schedulability of tasks since primary tasks may not be scheduled on slots reserved for backups. In order to estimate the loss of schedulability caused by the addition of the fault tolerance capability (the backup slots), we consider the simple FIFO (first come first served) scheduling of the primary tasks. We assume that A_{av} is the average rate of task arrival in the system with the number of tasks arriving at any time t being uniformly distributed between 0 and $2A_{av}$. In other words, if $P_{ar}(k)$ is the probability of k tasks arriving at a given time t, then $P_{ar}(k) = \frac{1}{2A_{av}+1}$. We also assume that the window sizes of the arriving tasks have a uniform distribution with a minimum of 3 and a maximum of W_{max}. That is, if $P_{win}(w)$ is the probability that an arriving task has a window w, then $P_{win}(w) = \frac{1}{W_{max}-3}$ for $w = 3, \ldots, W_{max}$. Note that the assumption that P_{ar} and P_{win} are uniformly distributed is not essential for the analysis technique described next. The technique may be applied to other distributions of P_{ar} and P_{win}.

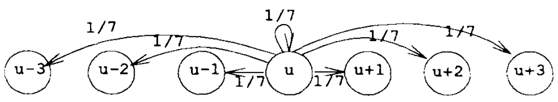

Figure 2: Transitions out of state S_u for a linear chain

FIFO scheduling 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[2], 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 k^{th} task in Q, then T_i is scheduled to execute at time slot $t + \lfloor \frac{k}{n-1} \rfloor$.

When a task T_i arrives at time t, its schedulability depends on the length of Q and on the window of the task, w_i. If T_i is appended at position q of Q and $w_i \geq \lfloor \frac{q}{n-1} \rfloor$, 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 $w_i \geq \lfloor \frac{q}{n-1} \rfloor + 2$, then Bk_i is also guaranteed to execute before $t + w_i$.

[2] One slot is reserved for backups.

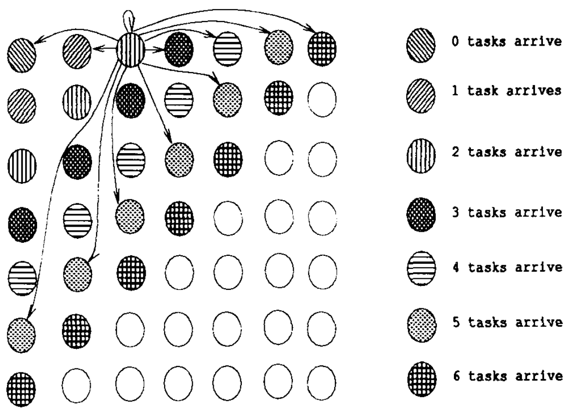

Figure 3: Transitions out of state $S_{2,0}$; $A_{av} = n = 3$.

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. 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 $k = 0, \ldots, 2A_{av}$. 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, Figure 2 shows the transitions in the Markov chain assuming that $A_{av} = n$ and $n = 3$.

When the k arriving tasks have finite window sizes, some of these tasks may be rejected. Let $p_{q,k}$ be the probability that one of the k tasks are rejected when the queue size is q. The value of $p_{q,k}$ is the probability that the window of the task is smaller than $\lfloor \frac{q+k/2}{n-1} \rfloor + \delta$, where δ is the extra time needed to schedule the backup and is equal to 1 or 2 with probability $1/(n-1)$ and $(n-2)/(n-1)$, respectively. Hence, when the queue size is q, the probability, $P_{rej}(r,k,q)$, that r out the k arriving tasks are rejected is $P_{rej}(r,k,q) = C^k_r (p_{q,k})^r (1-p_{q,k})^{k-r}$, where C^k_r is the number of possible ways to select r out of k objects.

In order to keep track of the number of rejected tasks, each state S_u is divided into $2A_{av}+1$ states, $S_{u,r}$, $r = 0, \ldots, 2A_{av}$, where $2A_{av}$ is the maximum number of tasks arriving, and possibly rejected, in each time unit. A transition into state $S_{u,r}$ indicates that r tasks are rejected. With this, a transition from S_u to S_v is now broken down to transitions to states $S_{v-r,r}$. Specifically, given that a state $S_{u,i}$, $0 \leq i \leq 2A_{av}$, represents a queue length of u, then if $u \geq n$, the probability of transition from $S_{u,i}$ to $S_{u-(n-1)+k-r,r}$ is

$$P[S_{u,i} \rightarrow S_{u-(n-1)+k-r,r}] = P_{ar}(k)P_{rej}(r,k,u-n+1) \quad (1)$$

$$for\ k = 0,\ldots,2A_{av}\ and\ r = 0,\ldots,k$$

If $u < n$, then

$$P[S_{u,i} \rightarrow S_{k-r,r}] = P_{ar}(k)P_{rej}(r,k,0) \quad (2)$$

$$for\ k = 0,\ldots,2A_{av}\ and\ r = 0,\ldots,k$$

In Figure 3, we illustrate the transitions out of state $S_{2,0}$, when $n = 3$ and 5 new tasks arrive. Specifically, $S_{2,0} \rightarrow S_{5-r,5}, r = 0,1,\ldots,5$, correspond to r of the 5 tasks being rejected. The breakdown of the transitions out of $S_{2,0}$ are not shown, but the target states are shaded according to the number of tasks that arrive. The top row of Figure 3 is when all tasks are accepted, while row r represents r tasks being rejected. Note that the number of states in each row of the Markov chain is $(n-1)W_{max}$ since the length of Q may never exceed this length.

By computing the steady state probabilities of being in the rejection states, it is possible to compute the expected value of the number of rejected task, R per time unit. Namely, if $P_{ss}(u,v)$ is the steady state probability of being in $S_{u,v}$, then

$$R = \sum_{u=0}^{(n-1)W_{max}} \sum_{v=1}^{2A_{av}} (vP_{ss}(u,v))$$

The rate of task rejection is then computed by dividing R by the average number of arriving tasks, A_{av}.

If no faults occur, the time interval used by the backups can be re-utilized (*backup de-allocation*). Backup deallocation means that if at time t no fault has occurred, then the backup pre-allocated during 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. The effect of backup deallocation may be analyzed by changing the transition probabilities in the above Markov chain. More specifically, the change of the transition probabilities in equation (1) is as follows:

$$P[S_{u,i} \rightarrow S_{u-(n-1),0}] = P_{ar}(0)$$

$$P[S_{u,i} \rightarrow S_{u-(n-1)+k-1-r,r}] = P_{ar}(k)P_{rej}(r,k-1,u-n+1)$$

$$for\ k = 1,\ldots,2A_{av}\ and\ r = 0,\ldots,k-1$$

The probabilities in equation (2) are changed similarly.

In Figure 4, we plot the rate of task rejection as a function of the number of processors, n, for the case $A_{av} = n$ with and without backup deallocation. The decrease in rejection rate 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, Figure 4 can be also looked at as a comparison between our strategy and the strategy of dedicating one spare as a backup.

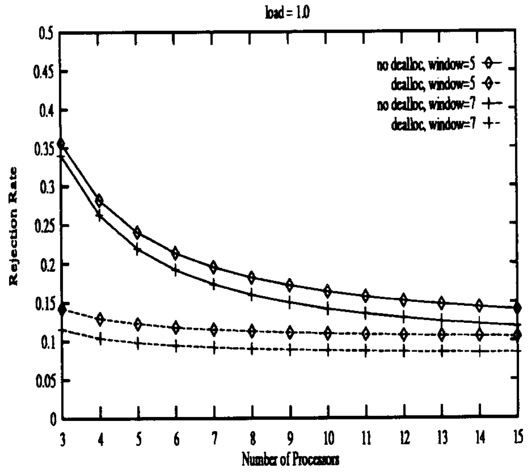

Figure 4: Rejection rate as a function of the number of processors

4 Scheduling Non-Uniform Tasks

4.1 Task Model and Scheduling

In this section we remove the restriction of the previous section about the execution times of the tasks. Here the execution time is modeled as in Section 2 by a parameter c_i.

Since the execution times of tasks are not fixed, we cannot pre-allocate the backup tasks as in Section 3. Also, the analysis of such systems becomes more complex. Therefore we will present a heuristic for the scheduling of tasks arriving dynamically, with different computation times and analyze this heuristic using simulations.

A *free slot* is the time not used by primaries and backups. *Forward Slack* is the maximum amount of time a slot can be postponed without violating any tasks' timing constraints. Forward slack will be called *slack*.

While scheduling the tasks and their backups, we maintain a list of the existing slots. Whenever a new task is received, the primary is scheduled as early as possible. As for scheduling backups, deallocation and overloading imply possibly conflicting heuristics. On the one hand, the chance of reutilizing the deallocated backup for scheduling other tasks increases if backups are scheduled as late as possible. On the other hand, maximizing the overloading provides an efficient utilization of the resources but may not result in the backups being scheduled as late as possible. In [4] we simulated an algorithm in which overloading is always favored. In this section, we study the relative merits of overloading and late backup scheduling. Specifically, we consider a parametric cost in which a parameter Ω represents the importance of overloading versus late scheduling. We call the scheme that uses such a parametric scheduling algorithm ***weighted overloading***.

Clearly, the scheduling policy for the backup will influence the schedulability of the tasks submitted to the system. When faced with the choice between overloading a backup, Bk_i, on a previously scheduled backup slot and scheduling Bk_i as late as possible, we use the user-defined parameter Ω to maximize the quantity defined by

$$cost = backup\_end + \Omega \times overlap\_length \qquad (3)$$

where *overlap_length* is the length of Bk_i which overlaps with a previously scheduled backup. That means that if $\Omega = 0$, we will schedule the backup as late as possible and if $\Omega = \infty$, we will attempt to overload as much as possible. This allows for tuning the parameter according to the system state.

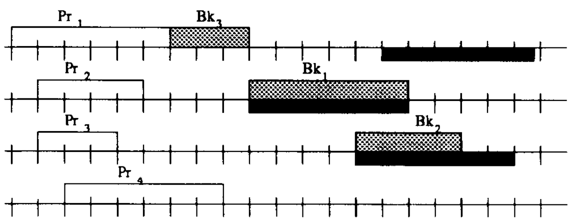

Figure 5: Scheduling 4 tasks on 4 processors. The black box represents the possible schedules of Bk_4

An Example

A simple schedule for four tasks is shown in Figure 5. The first three tasks[3] ($T_1 = \langle 0, 0, 6, 15\rangle, T_2 = \langle 1, 1, 4, 17\rangle, T_3 = \langle 1, 1, 3, 9\rangle$) are scheduled in 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 lastest 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.

Note that the backup can be scheduled in processors P_1, P_2 and P_3. That is because the primary is in processor P_4, and therefore the backup cannot is placed in the same processor. 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 Equation (3). The results of the tasks and their value t 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 highlighted cost is chosen.

proc	cost (from Eq 3)	wt=0	wt=0.5	wt=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

Table 1: Cost of scheduling in different positions of different processors, with different values of Ω

Notice that for each value of the weight, the location of the backup would be 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 $\Omega = 1$ in processor P_3, scheduling Bk_4 either at times 13 or 14 would yield the same cost.

4.2 Steps to Schedule the Task

Since we are interested in studying the effect of de-allocation and overloading on the schedulability of tasks, we consider a low-complexity scheduler. Simple schedulers are likely to be used in actual systems due to the low overhead and ease of implementation.

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 there is a free slot larger than c_i on a processor P between r_i and d_i, then we know Pr_i can be scheduled on P. If Pr_i cannot be scheduled without overlapping another time interval $slot_j$, then we have to check if we can reschedule $slot_j$. This can be done by checking the slack of $slot_j$. 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.

The backup for task T_i is scheduled as follows: Let the earliest schedule of primary Pr_i be on processor P_j. We look at the possible schedules for the backup on processors other than P_j. We try to schedule the backup on a free slot or overload it on an existing backup slot, according to the Equation (3). It is clear that the information about the forward slacks of primary slots need to be maintained so that they can be moved forward if necessary. However, to keep the complexity and running time of the algorithm low, our algorithm does not move backup slots. This 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 (and probably will) have a cascading effect and each time a backup slot is moved, it will be very costly to recalculate the slacks. This cascading effect can ripple through all tasks in the system, extending to processors other than the one that the backup is scheduled on. In contrast, shifting primaries will affect only tasks on a single processor. Since the scheduling algorithm only uses forward slacks, only the two processors on which the new slots are scheduled are affected.

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

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

that the task will be completed before its deadline even in the presence of a single fault.

4.3 Simulation and results

To study the scheduling algorithm presented in Section 4, we have performed a number of simulations. To the best of our knowledge, no simulation studies have been done for fault-tolerant scheduling of aperiodic tasks in dynamic real-time multiprocessor systems.

The schedule generated by the algorithm described in Section 4.2 can tolerate any single transient or permanent fault. In the case of a transient failure, only the failed task is run on the backup while in the case of a permanent failure, all tasks on the failed processor are rescheduled on their respective backups. This fault tolerance capability, however, comes at the cost of increasing the number of rejected tasks. The first goal of our simulation is to estimate this cost for the varying parameters as discussed below.

As we have mentioned, another approach for providing single fault tolerance in real-time systems is to dedicate a standby processor as a spare. The second goal of our simulation is to compare the schedulability of our scheme with that of the single spare processor scheme (or simply the spare scheme).

In the simulation, we measured the rejection rate of tasks as a function of the load, the window size, the number of processors, the time it takes for the system to tolerate a second fault (TTSF), and the weight.

4.3.1 Simulator and Simulation Parameters

Our simulator is 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. The interface with the visualisation tool [9] can be toggled at simulation dispatch time, and the simulator reads the simulation parameters from files and runs the simulation for each set of parameters. An output file is generated, from which several scripts extract the relevant information.

We generated task sets for the computation of the schedules and ran each policy on the same task set. The simulation parameters that can be controlled are:

- **number of processors** n: this is a fixed user input.
- **the average computation time**, c: the computation time of the arriving tasks is assumed to be uniformly distributed with mean c.
- **the 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 leads to larger task inter-arrival time. Specifically, the inter-arrival time of tasks is assumed to be uniformly distributed with mean $\alpha = c/(\gamma * n)$.
- a parameter β: it controls the window size, which is uniformly distributed with mean $c \times \beta$.
- **the overload weight**, Ω: it regulates the importance of overloading in contrast to scheduling the backup as late as possible.

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. Consistent with the model, we assume $\forall i, r_i = a_i$, yielding a dynamic system. Formally, $a_1 = r_1 = 0$ and $r_i = r_{i-1} + \alpha_i$, where α_i is the interarrival time. The load ranges from zero to one (i.e., $0 < \gamma \leq 1$). For example if $\gamma = 1$, $P = 4$, and $c = 4$, then the task inter-arrival rate is 1. This means that, on an average, one task arrives in the system every unit of time and thus the load on each of the 4 processors is 1. These parameters are summarised in Table 2.

To compute the TTSF, a fault is injected at a specific (and arbitrarily chosen) time instant, t. Theorem 2 is then applied to compute t'. We repeat this experiment 1000 times and average the results to obtain the mean TTSF.

4.3.2 Analysis of results

We start by analyzing the effect of the weight on the schedulability and on the time to second fault (TTSF). The results are shown in Figures 6 through 8. Then we show the results for the comparison of different schemes in Figure 9, the effect of each technique in Figure 10, and the effect of different window sizes in Figure 11.

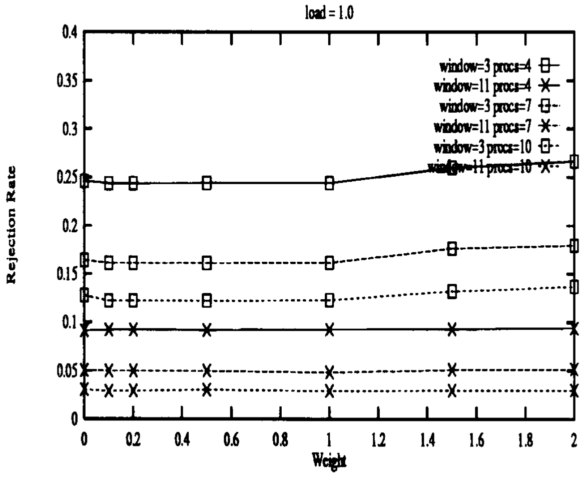

Figure 6: Effect of the parameter weight in the rejection rate

In Figure 6 we show the variation on the schedulability for different weights. For small window sizes, the optimal weight is between 0 and 1 (see Figure 7), while for larger windows, the maximum schedulability is obtained when $\Omega = 0$. The variation, however, for

parameter	name	distribution	values assumed
number of tasks	T	fixed	$1,000$
number of processors	n	fixed	$3,4,\ldots,10$
computation time	c	uniform	mean=5
load	γ	uniform	mean=$0.5, 0.6, \ldots, 1.0$
inter-arrival time	α	uniform	mean=$c/(\gamma * n)$
window size	β	uniform	mean=$c\beta$
weight	Ω	fixed	$0, 0.1, 0.2, 0.5, 1, 2, 5, 10, 15, 20$

Table 2: Parameters for Simulations

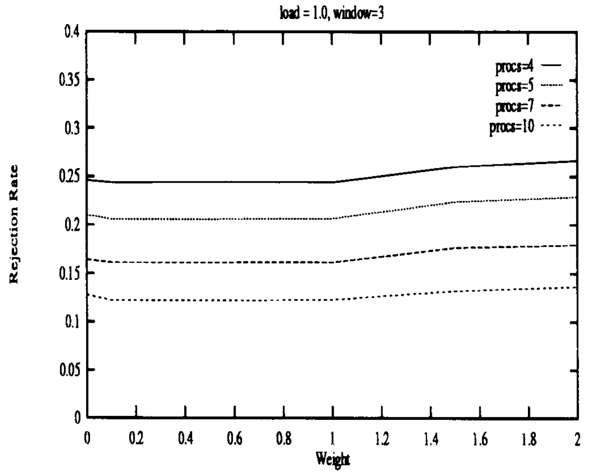

Figure 7: Rejection rate for window = 3

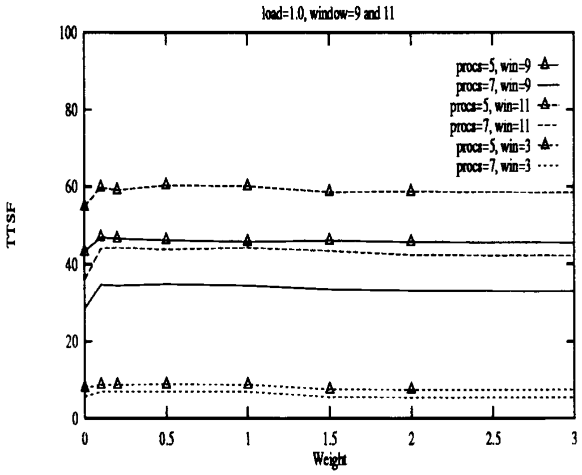

Figure 8: Schedulability versus weight for different windows and number of processors

different weights is relatively small. For instance, for window equal to 3 and load equal to 1.0, the rejection rate for 4 processors, varies from 0.2461 (at $\Omega = 0$) to 0.2814 (at $\Omega = 20$).

When we consider the weight in conjunction with the Time To Second Fault (TTSF in Figure 8), we can see that, although the weight has little influence on TTSF, the larger the window size, the larger the TTSF is. This is expected, since the larger window increases the length of the schedule.

The small sensitivity of the rejection rate and the TTSF to the weight indicates that the position of the backup task has little influence on the schedulability of tasks. This is because the processor time reserved for backups is reclaimed through de-allocation. This means that a simple strategy for scheduling backups may be preferred to a complex policy that increases the run time of the scheduling algorithm.

In Figure 9, we compare the rejection rate of three schemes, namely the spare scheme, our scheme, and the no-fault-tolerance (NOFT) scheme for different number of processors. In the NOFT 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 size.

In Figure 10, we can see the effect of using overloading and de-allocation in the schedulability of task sets plotted against the number of processors. The graph in that figure shows the different rejection rates for when we use overloading, de-allocation, both, and neither techniques. Note that whenever overloading is present ("both" and "overloading only"), we compare the results for *weight = 0* and *weight = 100.* An interesting result of these simulations is that the effect of using deallocation reverses the effect of using overloading. In other words, when only overloading is used, higher *weight* (more overloading) decreases the rejection rate; when de-allocation is also used, a higher *weight* has the reverse effect, since de-allocation is more beneficial when backups are scheduled as late as possible ($\Omega = 0$).

It is clear that the use of either method significantly

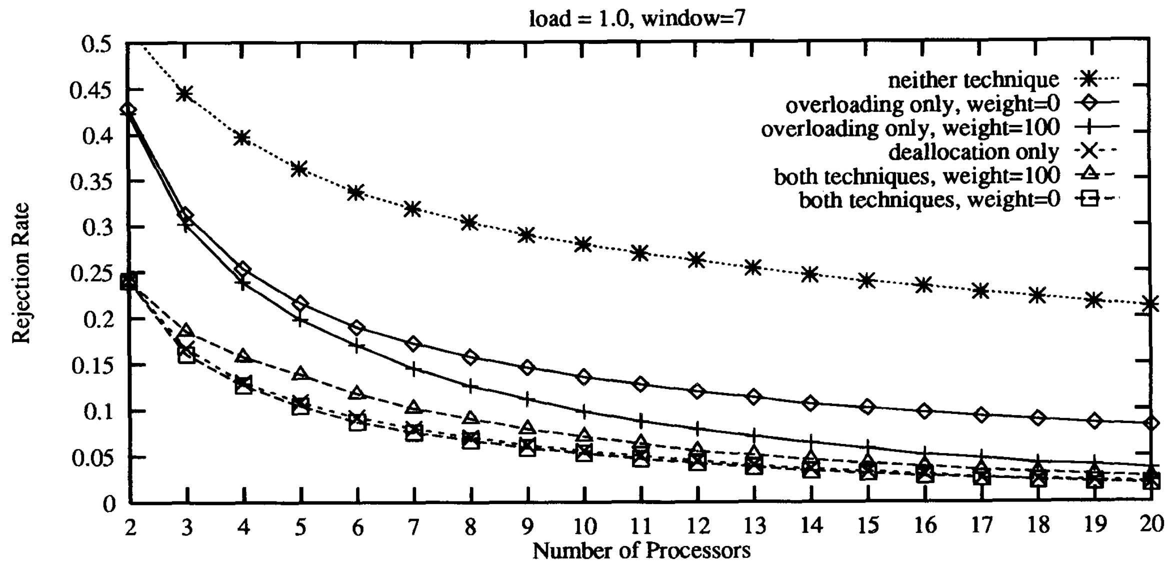

Figure 10: Rejection rate as a function of number of processors, using different techniques

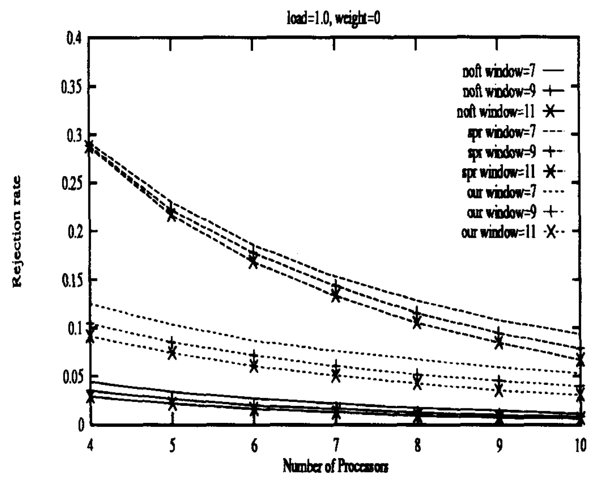

Figure 9: Comparison of rejection rates, as a function of load, for spare, our, and NOFT schemes

improves the schedulability of task sets, but the difference from one to both techniques shows a less drastic improvement. This leads us to believe that using either overloading or de-allocation in the scheduling of real-time fault-tolerant tasks will enhance the schedulability, but not much is gained from doing both. Although we do not show the results here, similar behavior is exhibited when the load is smaller or larger than 1.0.

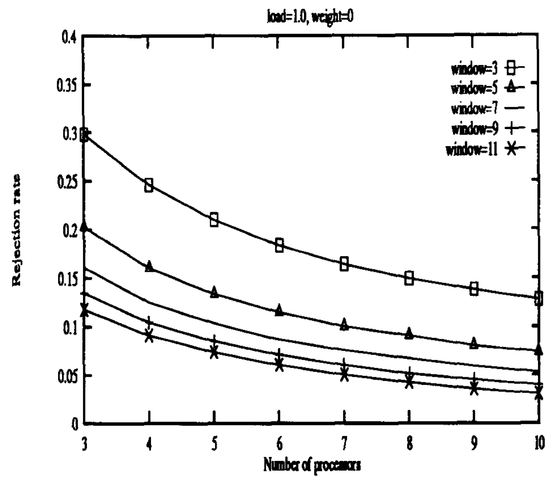

Figure 11: Rejection rate as a function of number of processors, for different windows

In Figure 11, we can see the effect of having different window sizes on the schedulability (plotting the rejection rate versus the number of processors). Note that the schedulability of our method is higher when the window is large and when there are more processors. Again, although we do not show the results here, similar behavior is exhibited when the load is smaller or larger than 1.

5 Concluding Remarks and Future Work

In this paper, we study a fault-tolerant scheduling method that tolerates processor faults, be them transient or permanent. We show that the overloading and de-allocation of backup slots provide an efficient utilization of the resources. Our results show positive correlation between the schedulability of task sets and the load on the system, as well as between the schedulability and the length of the average task window size. Both theoretical and simulation results indicate that the reclaiming of resources reserved for backup tasks (de-allocation) is the most important factor when scheduling tasks in a primary/backup fashion. With backup de-allocation, elaborate methods for increasing the overloading of backups seem to have only a small effect on schedulability 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 sufficient 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 multiples 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 non-faulty processors and thus can handle a second fault.

To handle more than a single simultaneous fault, we can schedule more than one backup copies 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.

The next step in this research is to further improve the scheduling algorithm by allowing backups for more than one primary from the same processor to be scheduled in the same backup slot (see Theorem 1). This is only possible if the actual computation times of tasks have high variances. We also intend to improve the simulator to measure the scheduling overhead, and to determine how effective this scheme is when coupled with a resource allocation scheme for distributed systems.

References

[1] S. Balaji, Lawrence Jenkins, L.M̃. Patnaik, and P.S̃. Goel. Workload Redistribution for Fault Tolerance in a Hard Real-Time Distributed Computing System. In *IEEE Fault Tolerance Computing Symposium (FTCS-19)*, pages 366–373, 1989.

[2] Ben A. Blake and Karsten Schwan. Experimental Evaluation of a Real-Time Scheduler for a Multiprocessor System. *IEEE Trans. on Soft. Eng.*, SE-17(1):34–44, Jan. 1991.

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

[4] Sunondo Ghosh, Rami Melhem, and Daniel Mossé. Fault-Tolerant Scheduling on a Hard Real-Time Multiprocessor System . In *International Parallel Processing Symposium*, April 1994.

[5] C.M̃. Krishna and Kang G. Shin. On Scheduling Tasks with a Quick Recovery from Failure. *IEEE Trans on Computers*, 35(5):448–455, May 1986.

[6] A.L. Liestman and R.H. Campbell. A Fault-tolerant Scheduling Problem. *Trans Software Engineering*, SE-12(11):1089–1095, Nov 1988.

[7] C. L. Liu and J. W.Layland. Scheduling Algorithms for Multiprogramming in Hard Real-Time Environment. *jacm*, pages 46–61, January 1973.

[8] J. J. Molini, S. K. Maimon, and P. H. Watson. Real-Time System Scenarios. In *11th Real-Time Systems Symposium*, pages 214–225, Lake Buena Vista, FL, Dec 1990. IEEE.

[9] Daniel Mossé. Tools for Visualizing Scheduling Algorithms. In *Computers in University Education Working Conference*, Irvine, CA, Jul 1993. IFIP.

[10] Sam K. Oh and Glenn MacEwen. Toward Fault-tolerant Adaptive Real-Time Distributed Systems. External Technical Report 92-325, Department of Computing and Information Science, Queen's University, Kingston, Ontario, Canada, January 1992.

[11] Yingfeng Oh and Sang Son. Multiprocessor Support for Real-Time Fault-Tolerant Scheduling. In *IEEE 1991 Workshop on Architectural Aspects of Real-Time Systems*, pages 76–80, San Antonio, TX, Dec 1991.

[12] Yingfeng Oh and Sang Son. Fault-Tolerant Real-Time Multiprocessor Scheduling. Technical Report TR-92-09, University of Virginia, April 1992.

[13] D.K. Pradhan. *Fault Tolerant Computing: Theory and Techniques*. Prentice-Hall, NJ, 1986.

[14] Robert L. Sedlmeyer and David J. Thuente. The Application of the Rate-Monotonic Algorithm to Signal Processing Systems. *Real-Time Systems Symposium*, Development Sessions, 1991.

[15] J. Xu and D. L. Parnas. Scheduling processes with release times, deadlines, precedence, and exclusion relations. *IEEE Trans. on Soft. Eng.*, SE-16(3):360–369, March 1990.