Fault-Tolerant Scheduling on a Hard Real-Time Multiprocessor System

Sunondo Ghosh, Rami Melhem and Daniel Mossé*
Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260

Abstract

Fault-tolerance is an important issue in hard real-time systems due to the critical nature of the supported tasks. One way of providing fault-tolerance is to schedule multiple copies of a task on different processors. If the primary copy of a task cannot be completed due to a fault, the scheduled backup copy is run and the task is completed. In this paper, we propose a new algorithm for fault-tolerant scheduling on multiprocessor systems. The algorithm guarantees the completion of a scheduled task before its deadline in the presence of processor failures. Our algorithm schedules several backup tasks overlapping one another and dynamically deallocates the backups as soon as the original tasks complete executions, thus increasing the utilization of processors.

Simulation results show that our method achieves higher task schedulability compared to using a spare processor as a backup to be invoked in the event of a failure. Further, we show that the cost, in terms of schedulability, of guaranteeing fault tolerance for dynamic systems is quite low.

1 Introduction

Multiprocessor systems have been widely used for parallel applications. In the ideal case, an application can be divided into n independent tasks, each task can be run on one of the processors in the system, and hence the runtime can be decreased by a factor of n.

For multiprocessor systems, reliability is an important issue. A system is fault-tolerant if it produces correct results even in the presence of faults [4]. Fault-tolerance is particularly relevant for multiprocessor systems because a higher number of processors increases the chances of a fault in the system. Fault-tolerance becomes even more important when the multiprocessor system is used to run real-time applications for which timing constraints should be satisfied. *Real-time systems* are those in which the temporal correctness is as important as the functional correctness. 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 smaller. Examples of hard real-time systems are a space station, a radar for tracking missiles, a system for monitoring a patient in critical condition, etc. In these real-time systems, it is essential that tasks complete before their deadline even in the presence of processor failures. This makes fault tolerance an inherent requirement of hard real-time systems.

A crucial issue in real-time systems is the scheduling of tasks in a way that does not violate the timing constraints. The general problem of optimal scheduling of tasks on a uniprocessor or a multiprocessor system is NP-complete [3]. Therefore, different heuristics have been used to schedule real-time tasks to maximize schedulability and/or processor utilization [2, 7, 18].

In a real-time multiprocessor system, fault tolerance can be provided by scheduling multiple copies of tasks on different processors [6, 12, 13]. *Primary/backup* (PB) and *triple modular redundancy* (TMR) are two basic approaches that allow multiple copies of a task to be scheduled on different processors [14]. One or more of these copies can be run to make sure that the task completes before its deadline. In the PB approach, if incorrect results are generated from the primary task, the backup task is activated. In TMR, multiple copies are executed and error checking is achieved by comparing results after completion. In this scheme, the overhead is always on the order of the number of copies running simultaneously.

In [6], a fault-tolerant scheduling algorithm is proposed to handle transient faults. The tasks are assumed to be periodic and two instances of each task (a *primary* and a *backup*) are scheduled on 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. It also assumes that the execution time of the backup is shorter than that of the primary.

In the area of multiprocessor systems, Son and Oh describe a fault-tolerant scheduling strategy for periodic tasks [12, 13]. In this strategy, a backup schedule

*Authors' e-mail: {ghosh, melhem, mosse}@cs.pitt.edu

is created for each set of tasks 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 [13] 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 scheduling, but have not performed any type of simulation or obtained experimental results [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 simultaneously and if a fault affects the primary, the results of the standby are used. On the other hand, [5] presents theoretical results that assume the existence of an optimal schedule, and augments that schedule with the addition of "ghost" tasks, which are standby tasks. However, since not all schedules will permit such additions, the scheme does not offer firm guarantees in faulty environments.

The remainder of the paper is organized as follows. In Section 2 we describe the problem. In Section 3 we introduce an algorithm for fault-tolerant scheduling of tasks on a multiprocessor system. Simulation results are presented in Section 4. In Section 5 we discuss future work and provide some concluding remarks.

2 The Fault-Tolerant Scheduling Problem

We introduce a fault-tolerant scheduling approach for real-time multiprocessor systems. We consider a system which consists of n interconnected identical processors and we assume that there is a task scheduling processor (central controller) which maintains a global schedule. Although we consider a central controller, the basic scheduling strategy can be be implemented with distributed schedulers.

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 assume that tasks are independent, that is, have no precedence constraints. This assumption can be removed if we can transform the precedence graph into independent nodes with new ready times and deadlines [9]. 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. This makes it possible to schedule both the primary and its backup within its time constraints. Tasks arrive dynamically in the system and are scheduled as they arrive.

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. A fault detection mechanism (e.g., fail-signal processors [11]) is assumed to immediately detect a fault and its type. If a completion guarantee cannot be assured for a given task, the task is rejected (i.e., the system does not try to schedule the task at a later time). Safety-related tasks are examples of such types of tasks, where the tasks need to be guaranteed to execute with fault tolerance, or should not execute at all. Both permanent and transient faults are handled by our approach. We do not consider the problem of software faults or correlated component failures.

We address the fault-tolerant scheduling problem by using a primary/backup approach. When a task arrives, two copies, a *primary* and a *backup*, are scheduled on two different processors. The backup copy of a task executes only if the execution of the primary copy is not completed in time (i.e., if a fault is detected, the backup is activated). However, unlike the scheduling algorithm presented in [12, 13], we assume dynamic systems in which tasks are aperiodic. The dynamic property allows us to deallocate backup copies of tasks as soon as the primary copies finish executing. In this manner, we are able to better estimate the system state (available free time) when new tasks are scheduled. Note that, unlike some previous studies [6], we assume that the primary and backup execution times are equal.

To evaluate the performance of our scheme, we compare our results with the **spare method**, in which one processor is allocated to be used as a spare. If a non-spare develops a fault, all tasks scheduled on this processor are executed on the spare. We also estimate the overhead of providing fault tolerance, in comparison with a system that has no provision for fault tolerance, which we call the **no fault tolerance (no FT) method**. In the case of real-time systems, this option is not desirable due to the possibility of catastrophic consequences if a fault occurs.

3 The Scheduling Strategy

Our scheduling scheme is based on the following observations: (a) in real-time systems, tasks must be memory resident at the time of execution; (b) a processor functioning as a spare will be idle throughout the life-time of the system; and (c) reservations of resources for backup copies must be ensured, but backup copies can have a different scheduling strategy than primaries. 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. Although memory prices are relatively low, this approach is still spatially and financially impractical. We also notice that the spare processor would not be used by any executing tasks in the case of fault-free operation, thus lowering the schedulability of the system.

The third observation, namely that resources reserved for backup copies could be re-utilized, motivated us to introduce the notion of overloading. Overloading is the scheduling of more than one backup in

the same time slot. This notion will be explained further in the next section. If we can ensure that there will be at most a single fault in the multiprocessor system, we can overload a backup slot on a processor with backup tasks from the other $n-1$ processors. Note also that due to the dynamic nature of our system, we are able to reclaim resources in case of fault-free operations.

3.1 Rationale

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
- *deallocation* of backup tasks dynamically when the corresponding primaries complete successfully.

These techniques help us ensure high schedulability while providing fault-tolerance.

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 slots 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 from different processors are scheduled to run in the same time slot, that backup slot is said to be *overloaded.* 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.*

It is easy to see that the following *restrictions* lead to a schedule which satisfies the real time and the fault tolerance requirements stated above.

- Pr_i and Bk_i for task i have to be scheduled on different processors to allow any single fault to be tolerated. This restriction can be removed if only transient faults are to be tolerated.
- The begin time of Bk_i has to be greater than the end time of Pr_i so that Bk_i can execute if a fault is detected in the processor on which Pr_i is executing.
- Both Pr_i and Bk_i should be scheduled between r_i and d_i.
- two primary tasks that are scheduled on the same processor, their backups must not overlap. Otherwise it will not be possible to reschedule two primaries from the same processor p onto the backup slot if p fails.

A simple schedule for four tasks is shown in Figure 1. For these tasks it is assumed that $a_i = r_i$. Bk_1 and Bk_3 are overloaded on the same backup slot. Note that Bk_3 could have been scheduled later on processor 2, but that would decrease the overlap of backups and would also diminish the available free slot time. 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.

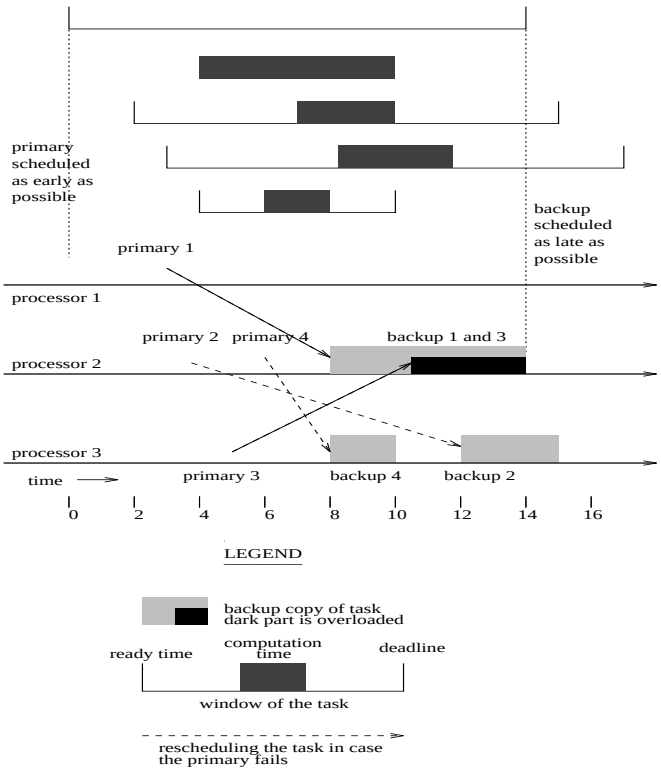

Figure 1: Scheduling 4 tasks on 3 processors

A modification of the initial schedule is shown in Figure 2, where the completion of tasks 2 and 1 (in that order) causes the de-allocation of their respective backups. The arrival of tasks 5 and 6 (Figure 3) causes further modifications in the schedule. Bk_3 is overloaded with Bk_5, and due to the de-allocation of the Bk_1, Pr_6 can be scheduled on processor 2.

3.2 Algorithm Principles

While scheduling the tasks and their backups, we maintain a list of the existing slots. Whenever a new task is received, the primary and backup have to be scheduled. We schedule the primary as early as possible and the backup as late as possible. When a primary task completes, its corresponding backup slot is deallocated. Since the deallocated slot is as far from the present time as possible, the probability of reusing the newly created free slot increases. This improves the utilization of the processors and provides fault-

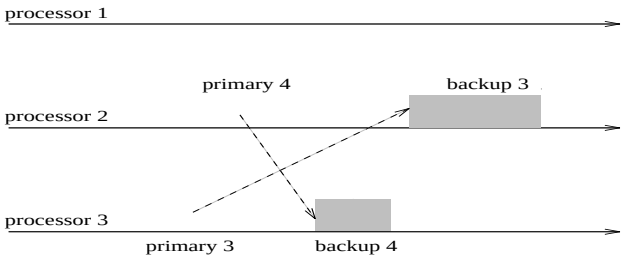

Figure 2: After the completion of tasks and deallocation of respective backups.

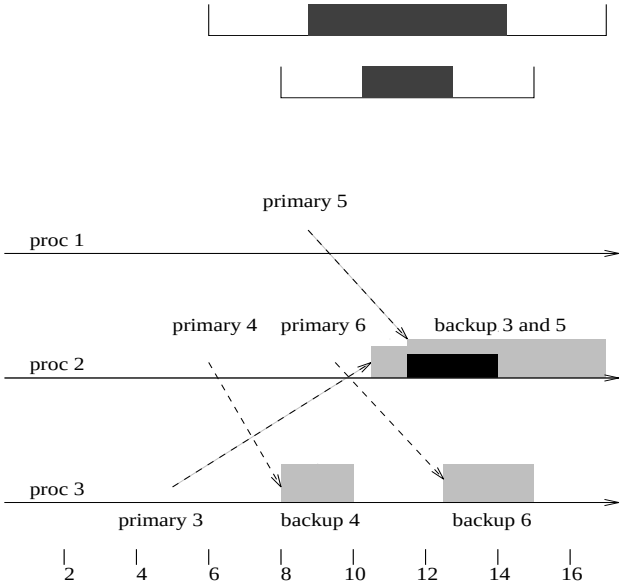

Figure 3: The new schedule after the arrival of two more tasks

tolerance with low overhead, in terms of schedulability.

Our algorithm schedules a primary before scheduling its corresponding backup, due to the following. 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. This is because we cannot schedule the primary and the backup on the same processor (see previous section). This constraint has less effect when we try to schedule the backup after the primary due to the possibility of overloading.

If the backup is scheduled first, many options exist, some of which have conflicting requirements. First, we have two choices, namely overloading an existing backup slot or scheduling the backup on free slots. Overloading is preferred because it minimizes the utilization of available processor time. However, we may reduce the time span in which the primary can be scheduled if the overload schedule for the backup is early in the task's window. On the other hand if we simply schedule the backup on a free slot, we leave a bigger span for the primary to be scheduled but we decrease the utilization of the processors.

If the primary is scheduled first, we remove the above conflicts. We schedule the primary as early as possible on some processor. This ensures that we have the maximum possible time span to schedule the backup. In a system with a central controller, the existence of a global schedule makes it easy to determine the earliest possible schedule for the primary. In this case we can simply try to overload the backup wherever possible and if no such schedule exists, we schedule it on a free slot.

The schedule that we choose for a primary may affect the schedulability 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, it may be shown that 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. The solution is complete if we find a schedule for the primary on every processor and then try to schedule the backup for each primary schedule. Since this 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. A similar approximation approach is suggested in the focussed addressing scheme [19].

3.3 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 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 slot, $slot_j$, then we have to check if we can reschedule $slot_j$. This can be done by checking the slack of $slot_j$. 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. We consider the earliest two tentative schedules on two different processors. For each of

these, we consider a different possibility of scheduling the backup.

The backup for task i is scheduled as follows: Let the earliest schedule of primary i be on processor p_j. We look at the possible schedules for the backup on processors other than p_j. Our first choice is to overload an existing backup slot as explained earlier. If there is no backup slot that can be overloaded, we schedule the backup on the latest possible free slot. We maintain forward slacks of primary slots and we allow them to be moved forward if necessary. We do not allow backup slots to be 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 will 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.

Therefore the algorithm for the fault-tolerant scheduling of task T_i is:

- Schedule Pr_i as early as possible.
- Try to overload Bk_i on an existing backup slot. If that is not possible, schedule it as late as possible on a free slot.
- If a schedule has been found for both Pr_i and Bk_i, commit the task. Otherwise reject it.

3.4 Runtime Behavior

As the tasks arrive in the system, two copies of the task (primary and backup) are scheduled on two different processors by the scheduling algorithm presented above. While the primary is in execution, if no faults occur in the processor on which the primary is scheduled, the primary can run to completion. In that case, the backup is deallocated and the newly generated free slot is made available for scheduling any task that arrives after the deallocation. This results in a remarkable increase in schedulability.

If there is a permanent fault in a processor, the backups of all primaries scheduled or running on that processor are executed on their respective backups. For any task that arrives after the fault, both the primary and backup copies have to be scheduled on the remaining non-faulty processors. A second fault can be tolerated in the system only after the last primary scheduled on the faulty processor has been run on its backup schedule and the last task which had a backup scheduled on the faulty processor has been executed to completion.

In case of a transient fault, only the backup copy of the currently executing primary is activated. All other tasks remain as scheduled.

4 The Simulation

To study the scheduling algorithm presented above, 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 3.3 can tolerate any single transient or permanent fault. 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 by comparing the schedulability in our fault-tolerant algorithm with that of the *no FT* method. The second goal of our simulation is to compare the schedulability of our scheme with that of the single spare processor scheme. In the **spare** and the **no FT** methods, the primary copies are scheduled using the same algorithm as the one used in our method. Note that the spare method is equivalent to having backup copies of all the primaries implicitly scheduled on the spare processor.

In the simulation, we measured the number of rejections in a set of tasks as a function of the load, the window size, and the number of processors. To aid in the development of the simulation, we used a visual tool [10] that displays the tasks and animates the behavior of the schedule. It shows the task to be scheduled, the running primaries, and the scheduled primaries and backups in different colors. It also shows the computation time of the incoming task and whether it was scheduled or rejected. We used this tool to verify that the generated schedule was not violating timing constraints and to find out bugs in the simulator.

4.1 Simulation Parameters

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** P: 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 $\alpha = c/\gamma * P$.

- a parameter β: this parameter controls the window size, which is uniformly distributed with mean $c \times \beta$.
- the **lead time**, δ: this parameter is the difference between the ready time and the arrival time. That is, $r_i = a_i + \delta$.

In the simulations, our task sets consisted of 1000 tasks. We generated 100 task sets for each set of parameters and calculated the average of the results generated. We also assumed that $\delta = 0$, which means that the arrival time and ready time of a task are the same. Thus, we have 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., $0 \leq \gamma \leq 1$). For example if $\gamma = 1$, $P = 4$, and $c = 4$, then the average task inter-arrival time is 1. This means that, on an average, one task will arrive in the system every unit of time and thus the load on each of the 4 processors will be 1. These parameters are summarized in Table 1.

As mentioned earlier, in our simulations we maintain only the earliest and the second earliest schedules of the primary before trying to schedule the backup. We noticed during our simulations that *the second earliest primary schedule is rarely used for a successfully scheduled task.* This indicates that the two earliest primary schedules are sufficient to find a feasible schedule for the task if one exists.

4.2 Analysis of results

The results of our simulations can be found in Figures 4 through 7. In Figure 4 we plot the rejection rate of tasks for our method as a function of load (using 5 processors). As expected, the rejection rate increases with an increase in load. We can also see that, due to the primary/standby approach used and our deallocation technique, the larger the window size, the smaller the rejection rate is. The schedulability of tasks does not suffer when the window size is 9. This window size is commonly used in applications such as signal processing systems, where the utilization[1] of different tasks is often less than 11% [15].

In Figure 5 we compare the three schemes, namely the spare scheme, our scheme, and the no FT scheme for different loads. Note that our scheme consistently performs better than the single spare scheme, for all values of window size.

In Figure 6 we can see the drastic improvement of having smaller loads when the window size is fixed (plotting the rejection rate versus the number of processors). Note that the schedulability of our method is better when the window is large, the load is small, and the number of processors is large. This combination of factors results in a higher degree of overloading, thus allowing more tasks to be scheduled in the multiprocessor system.

[1] Utilization can be seen as the inverse of the window size with respect to computation time.

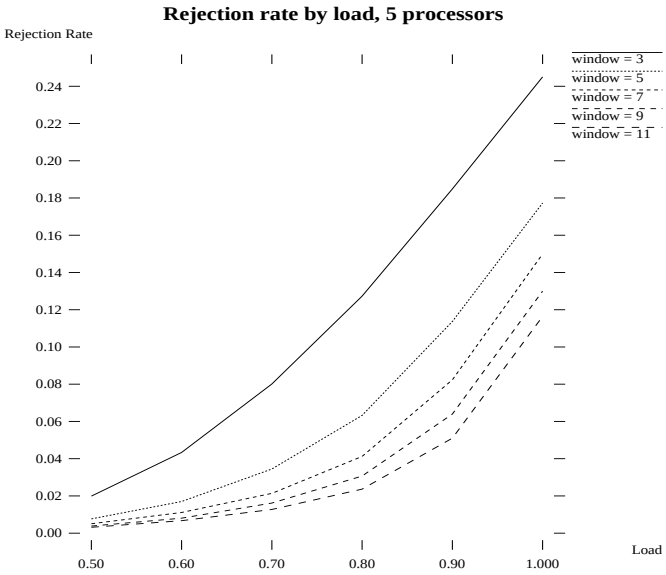

Figure 4: Rejection rate, as a function of load, for different window sizes, with 5 processors

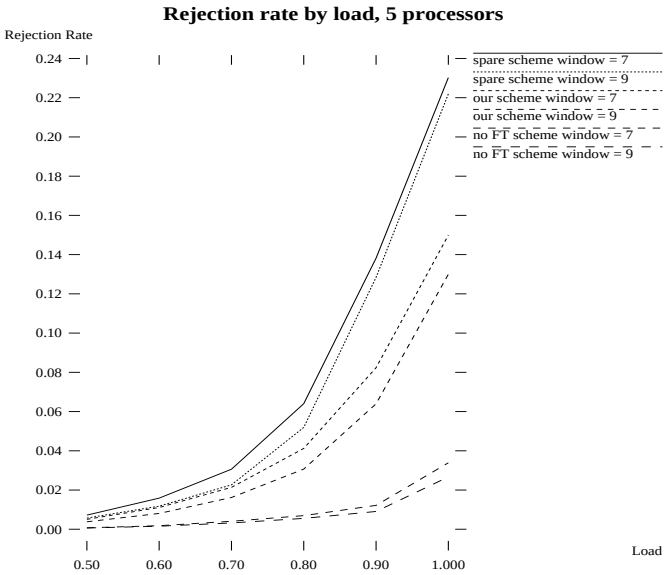

Figure 5: Comparison of rejection rates, as a function of load, for different loads

In Figure 7 we can see the rejection rate of the three schemes, as a function of the window size. It is clear that the schedulability in the no FT method is higher than in our method. However, notice that the gap is small when the window size is large. This, coupled with the window sizes shown in [15], leads us to believe in the high applicability of our scheme.

Although we have shown here only a few of the measurements we have done, it is important to note

parameter	name	distribution	values assumed
number of tasks	T	fixed	1000
number of processors	P	fixed	3, 4, 5
computation time	c	uniform	mean=5
load	γ	fixed	0.5, 0.6, 0.7, 0.8, 0.9, 1.0
inter-arrival time	α	uniform	mean=$c/(\gamma * P)$
window size	β	uniform	mean=$c\beta$

Table 1: Parameters for Simulations

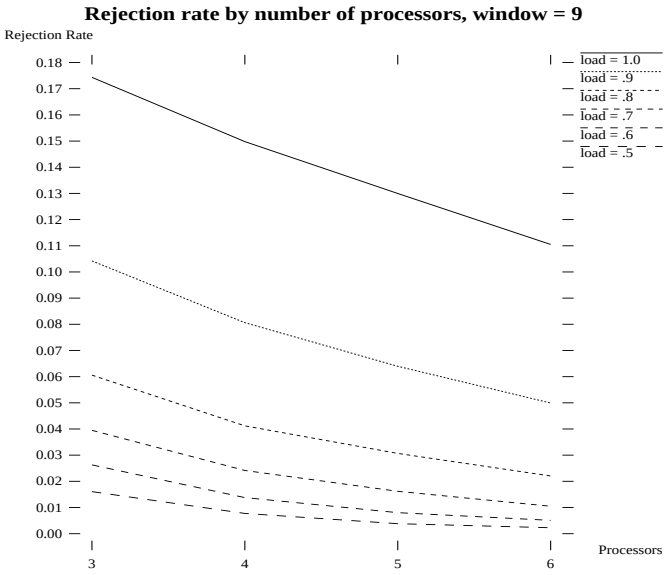

Figure 6: Rejection rate of overloading scheme, as a function of number of processors, for different loads

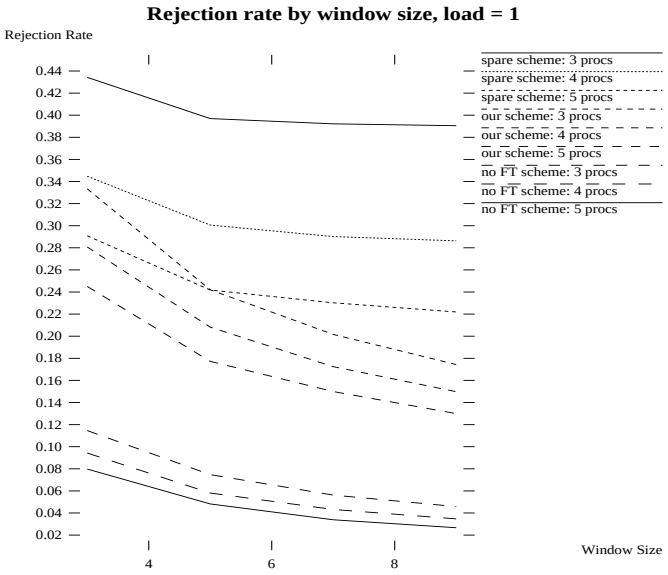

Figure 7: Rejection rate versus window sizes, for 3, 4, and 5 processors

that the experiments were done for various loads, including overload situations (load = 1.2). Also, the consistency of results emphasizes the appropriateness of our algorithm for a variety of parameters.

5 Concluding Remarks and Future Work

We have presented in this paper a fault-tolerant scheduling method, that allows processor faults, be them transient or permanent. We have shown that the overloading of backup slots provides an efficient utilization of the resource. Our preliminary results show positive correlation between the schedulability of task sets and the load on the system. We also observe that the schedulability increases with the average task window size.

One of the main contributions of this paper is the introduction of the idea of backup overloading which demands less processor time to provide fault-tolerance (i.e., schedule all backup tasks) hence increasing schedulability. Another important achievement is the successful combination of the idea of resource reclamation with the primary/backup approach. Resource reclamation has been used previously for tasks that finish execution earlier than their worst case execution time [17] but not for the primary/backup approach. The backup de-allocation is a combination of the two ideas. Overloading and deallocation increase the schedulability of real-time tasks on the multiprocessor system.

Our scheduling algorithm is different from other algorithms in that it deals with aperiodic tasks in dynamic real-time multiprocessor systems. The algorithm, which is easy to understand and implement, results in a high degree of schedulability, predictably, fast response to urgent events, and stability under transient overload which are important measures in real-time systems [16].

The type of results presented in this paper provides a tool for real-time system designers to compare three schemes, namely the spare processor, the no FT, and our fault-tolerant scheduling approach. Depending on the fault model, the memory availability, the schedu-

lability required, and other parameters such as the arrival rate, average window and computation time of tasks, the designer can determine which approach best matches the system's requirements.

In its current form, the algorithm can only tolerate more than one fault if they are separated by a sufficient amount of time. To handle two simultaneous faults, more than one backup copy should be scheduled for each task. Note that, although the scheduling policy for this case is different from the one presented in this paper, the mechanisms we have developed remain the same. Note also that it is not difficult to couple our scheduling algorithm with hierarchical resource allocation schemes that will accept local schedulers [8].

The next step in this research is to develop a scheduling algorithm which uses some weight to achieve a relative balance between backup overloading and late backup scheduling. The optimal values for the weight depend on system parameters.

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] Barry W. Johnson. *Design and Analysis of Fault Tolerant Digital Systems.* Addison Wesley Pub. Co., Inc, 1989.

[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] Daniel Mossé, Sam H. Noh, Bao Trinh, and Ashok K. Agrawala. Multiple Resource Allocation for Multiprocessor Distributed Real-Time Systems. In *Workshop on Parallel and Distributed Real-Time Systems (PDRTS), IEEE IPPS'93*, Newport Beach, CA, April 1992.

[9] Daniel Mossé. Extracting Task Timing Constraints from Applications. Technical Report, University of Pittsburgh, 1992.

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

[11] 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.

[12] 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.

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

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

[15] 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.

[16] Lui Sha and Shirish S. Sathaye. A Systematic Approach to Designing Distributed Real-Time systems. *Computer*, July, 1993.

[17] Chia Shen. *Resource Reclaiming in Multiprocessor Real-Time Systems.* PhD thesis, University of Massachussets, Amherst, 1992.

[18] 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.

[19] W. Zhao and K. Ramamritham. Distributed Scheduling Using Bidding and Focused Addressing. In *Proc. IEEE Real-Time Syst. Symp.*, pages 103–111, Dec. 1985.