Fault-tolerant real-time scheduling under execution time constraints

TIK-Report Nr. 76
June 1999

Martin Naedele *

Abstract

The primary/backup with deallocation approach of [2] is a strategy for the fault-tolerant online scheduling of hard real-time tasks. In this scheme, tasks are either rejected within a short time after the request or guaranteed to be executed even in case of a processor failure.

In this paper several heuristics for the guarantee algorithm are investigated. For the first time different processor selection strategies for guarantee algorithms with execution time constraints are compared. In addition, the concept of a decision deadline is introduced which then leads to an extension of the primary and backup checking routines. The thus modified checking routines are shown to achieve a lower rejection ratio for tight task deadlines and constrained scheduler execution times than the modification making use of task slack suggested in [2].

*ETH Zurich, Computer Engineering and Networks Laboratory, CH-8092 Zurich, email: naedele@tik.ee.ethz.ch

Contents

1 Introduction **3**

2 System and task model **4**

3 Guarantee algorithms **5**
3.1 Processor selection strategies 5
3.1.1 Sequential search 5
3.1.2 Load-based selection 6
3.1.3 Random selection 6
3.2 Budgeting 6
3.3 Basic checking routines 6
3.3.1 Checking routine for primary candidate 6
3.3.2 Checking routine for backup candidate 7
3.4 Modifications of the checking routines 7
3.4.1 Using primary slack 7
3.4.2 Using the decision deadline of a task request 7

4 Simulation Experiments **9**
4.1 Assumptions 9
4.2 Task Set Parameters 10
4.3 Results 11

5 Conclusion **11**

1 Introduction

In systems where aperiodic invocations of hard real-time tasks occur due to interactions with the environment, such as avionics applications or patient monitoring in an intensive care unit [2], it is often necessary to provide fault-tolerance for the tasks as well.

Recently there has been some interest in the fault-tolerant scheduling of non-preemptive, aperiodic real-time tasks on a multiprocessor system using a primary/backup approach. Tasks are either rejected within a short time after the request so that appropriate emergency measures can be taken or they are guaranteed to be executed even in case of a processor failure.

In [2] two identical copies of a task, primary and backup, are scheduled on different processors for fault-tolerance. The primary is scheduled as soon as possible and the backup as late as possible. With this scheme, the backup needs not be executed (it can be *deallocated*) if the primary completes succesfully. Also, backups of primaries on different processors may overlap, as all except for one are guaranteed to be deallocated in time.

In [5] the algorithm of [2] is extended for the case of tight deadlines. In this case the backup is divided into two parts, the first of which is executed concurrently with the primary and the second is scheduled behind the primary and may be deallocated as above in case the primary completes successfully.

The work of [1] is also concerned with tasks with tight deadlines. Here passive replicas and checkpointing are used. This approach even allows to overload primaries and backups. The main concern of that work are ways to deal with the *domino effect* that occurs once a backup needs to be activated which is overloaded with a primary which is then also to considered to have failed, and so on.

[3] proposes a general algorithm for the fault-tolerance strategies of triple modular redundancy, primary/backup, and imprecise computation. In order to reclaim unused computation time of tasks that did not need their worst-case excution time the algorithm rebuilds the complete schedule with each newly arriving task using a heuristic search algorithm with a constraint on the number of branches that may be taken.

As the scheduling of tasks with arbitrary execution times on a multiprocessor system is NP-complete [4] good heuristics are needed for the scheduling of primaries and backups. Such heuristics must fulfil two requirements: they must be able to guarantee a high number of requests and they must be able to make the guarantee decision within a limited and short period of time after a request [5]. Even though the above works concede the importance of simple and effective scheduling heuristics, they do not give much detail of the actual algorithms used and do not investigate the effectivity/execution time trade-off of their heuristics.

In this paper several guarantee heuristics to be used with the primary/backup scheme are investigated. In particular, for the first time different processor selection strategies for guarantee algorithms with execution time constraints are compared. In addition, the concept of a decision deadline is introduced which then leads to an extension of the primary and backup checking routines. The thus modified checking routines are shown to achieve a lower rejection ratio for tight task deadlines and constrained scheduler execution times than the basic checking routines and even the modification making use of task slack suggested in [2].

2 System and task model

This paper assumes the same system and task model as [2]. *Aperiodic hard real-time* tasks are to be scheduled *online* on a *multiprocessor* system with *identical processors*. The tasks are *independent* and *do not share resources*. Tasks are either rejected within a specified period after the request or *guaranteed* to be executed within their deadline even in case of permanent processor failure, possibly tolerating more failures after a certain recovery time. For simplicity, it is assumed that the guarantee algorithm is executed on a *single global scheduling processor* with appropriate fault-tolerance mechanisms for itself. Various ways to detect task and processor failures are mentioned in [2].

A *task request* is specified by its *arrival time* t_a, its *worst-case execution time* c, its *relative deadline* d_{rel}, its *absolute deadline* $d_{abs} = t_a + d_{rel}$, and its *decision deadline* d_d (which will be explained in section 3.4.2). The relative deadline is assumed to be at least twice as large as the worst-case execution time.

A *slot* is the actual time interval on a processor's schedule which is allocated to a certain task instance.

Fault-tolerance is achieved by the *primary/backup* approach of [2]: for each task request two identical instances are scheduled on different processors within the relative deadline of the request. The instance in the earlier slot is called the primary (Pr(i)). It is executed if no failure occurs. The slot of the backup (Bk(i)) starts at some time after the end of the slot of the primary. The backup is only executed if the primary fails, otherwise it is deallocated - its slot is removed from the schedule - after the successful completion of the primary so that the capacity of the backup slot is again available for new requests.

As stated in [2], the relations between the slots for primary and backup necessary for tolerating a single fault are:

1. Both primary and backup must be scheduled within the task's deadline, the backup after the primary.

2. Backup and primary must be scheduled on different processors.

3. If the primaries of two tasks are scheduled on the same processor their backups must not overlap on their processor.

4. A primary and a backup must never overlap (not explicitely stated).

Fig. 1 demonstrates the various task parameters and their relations. The backup is placed ALAP so that the actual relative deadline for the primary is reduced to $d_{rel,prim} = d_{rel} - c$. Also, there is a time span within which the guarantee algorithm needs to make its decision. The earliest time the guarantee algorithm may schedule the primary is thus not t_a but t_s which it can calculate from its knowledge about t_a and the worst case execution time of the guarantee algorithm.

To evaluate the performance of different guarantee algorithms the rejection ratio produced by each of these algorithms without any fault occuring is measured. Both the rejection ratio with respect to the total number of requested tasks ($rr_n = \frac{number\ of\ rejected\ tasks}{total\ number\ of\ task\ requests}$) and with respect to the total executable computation load ($rr_c = \frac{sum\ of\ c\ for\ all\ rejected\ tasks}{sum\ of\ c\ for\ all\ task\ requests}$) are measured.

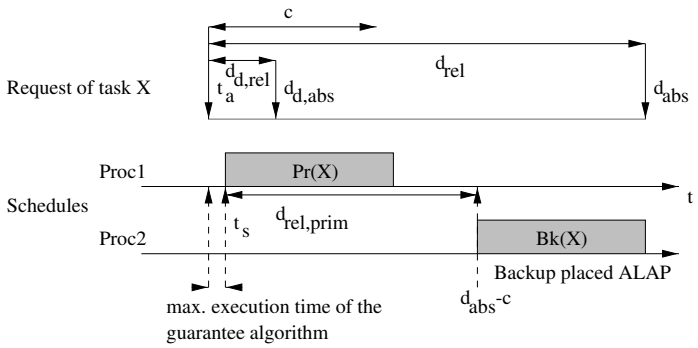

Figure 1: Task and scheduling parameters.

3 Guarantee algorithms

The algorithms which are evaluated in this work consist of three parts each: the *processor allocation strategy* which is different for each algorithm, and the *primary* as well as *backup checking* routines which are the same for each algorithm. The latter are almost completely determined by the scheduling conditions stated in section 2, and the two modifications which will be discussed later.

3.1 Processor selection strategies

If the number of comparisons the scheduler may make is limited, then it is very important that the heuristics which determines the candidate processors is effective, that means, that it selects first the processor on which the primary or backup checking routines are most likely to succeed.

In this work, three different processor selection heuristics were evaluated: *sequential search*, *load-based selection*, and *random candidate selection* as baseline.

Following the suggestion in [2], the number of primary candidates is limited to two and the primary candidates are chosen and checked before the backup.

The processor selection strategy in [2] is a simple sequential search. As the paper is not concerned with finding an effective selection heuristics, it proposes a selection strategy which simply tries each processor, one after the other, for both primary and backup. This strategy was not included in the comparison but a modified version was investigated instead.

3.1.1 Sequential search

The main difference to the simple sequential search of [2] is to advantageously choose the starting processor for both primary and backup search. Experiments indicate that a good starting processor for the primary search is the one after the processor on which the previous primary had been successfully placed. In order to reduce collisions between primary and backup candidate selections, the backup selection sequence runs in the opposite direction of the primary selection sequence, starting with the processor previous to the processor on which the previous primary had been successfully placed. Each search continues until a valid result is found, all processors have been examined, or the comparison budget has been exhausted.

3.1.2 Load-based selection

The load-based selection strategy keeps track of the load of each processor. The load is the sum of the worst-case computation times of the tasks currently scheduled on that processor. Backups are weighted with a factor of 0.3 because they are deallocated in the case of no fault occurring. This value has been determined by the observation that in a simple search strategy processors accommodate about triple the backup load compared to the primary load.

After the processors are sorted according to their load, the least loaded processor is selected as candidate for the first primary, the third least loaded for the second primary and the second least loaded for the backup.

3.1.3 Random selection

In order to have a baseline for algorithm comparison simple random processor selection was chosen as the third strategy to be evaluated.

3.2 Budgeting

A backup check fails for a primary candidate if the backup slot overlaps with another backup for a primary on the same processor as the primary candidate (see section 2). In this case this primary candidate is dropped, but the backup slot may still be valid for a second primary candidate. The backup check algorithm deals with all primary candidates in parallel, but would need to be rerun completely to check for each new primary candidate. As the backup check always has to examine each currently allocated slot on the backup processor, its effort is usually higher than that of a primary check.

After having found the first and necessary primary candidate it is thus necessary to decide how the remaining number of comparisons is most effectively distributed between the checking of another primary and the backup check. For this trade-off a concept of *budgeting* is introduced: The total budget of comparisons is divided into a primary and a backup budget. If the checking of the first primary candidate uses less than the primary budget, the remaining primary budget is used for the checking of a second primary, otherwise the backup check is started directly. Through experiments it has been found that this policy works well if the primary and backup budgets are each half of the total budget.

3.3 Basic checking routines

The basic checking routines used in this work are the same as in [2], except for the fact that in that paper the primary checking routine always uses uses a primay's slack (see section 3.4.1).

3.3.1 Checking routine for primary candidate

The primary check algorithm checks for each gap between a specified starting time[1] and the time $d_{abs} - c$ whether a primary of size c may be placed into it. The check terminates with the first successful fit.

[1] The later of either worst case completion time of the guarantee algorithm or the end of the currently executing task.

3.3.2 Checking routine for backup candidate

One result in [2] is that the ALAP strategy for backup positioning can only be marginally improved by investing effort to force increased overloading of backups. The backup check algorithm thus places the backup in the slot $[d_{abs} - c, d_{abs}]$ and makes no special effort to overload backups.

For the backup candidate processor, each slot is examined on whether it overlaps the backup candidate position. If the overlapping slot is a primary, the backup check has failed. If the overlapping slot is another backup the primary of which is on the same processor as one of the current primary candidates, then the respective primary candidate is dropped. If no primary candidates are left, the backup check has failed.

3.4 Modifications of the checking routines

3.4.1 Using primary slack

In [2] it has been suggested to make use of the slack of primaries to increase the likelyhood of succesful primary placement: If the primary check algorithm finds that a gap in the schedule is not big enough to fit in the new primary and if the limiting slot is a primary, an attempt is made to shift this limiting slot (and possibly others in front of it) forward as much as possible. In [2] the actual improvement gained by this scheme has not been evaluated.

This work investigates the utility of a slack-testing scheme which has been modified with respect to the one described in [2] as follows:

1. Slack is local, that is, at most one slot is shifted. Experiments have shown that the shifting of multiple slots only marginally contributes to a decrease in the rejection rate.

2. The slack test is done for each gap immediately, if necessary, not only after all gaps on all processors have been tried without it.

3. The slack is also used in the backup check algorithm to move primaries that overlap the candidate backup slot out of the way.

3.4.2 Using the decision deadline of a task request

While the slack test may be useful whenever a primary is the obstacle, it does not help with respect to backups. For this situation a novel strategy is suggested, which allows to get around conditions 3 and 4 stated in section 2.

The use of a guarantee algorithm always implicitely assumes that there is a time period after the task request in which the decision about the acceptance or rejection of the task is made, so that in the case of rejection appropriate measures can be taken. When designing a system it will thus be necessary to specify a *decision deadline* d_d for each task request which constrains the useful worst-case execution time of the guarantee algorithm (see also the discussion in section 4.1).

This decision deadline can actually be used to improve the basic checking routines for both, primaries and backups, by allowing to overload certain backup slots which must not be overloaded according to the original conditions:

- Assume the *primary check* finds that a certain gap is too small to schedule the primary and the limiting slot at the end of the gap is a backup (see Fig. 2). According

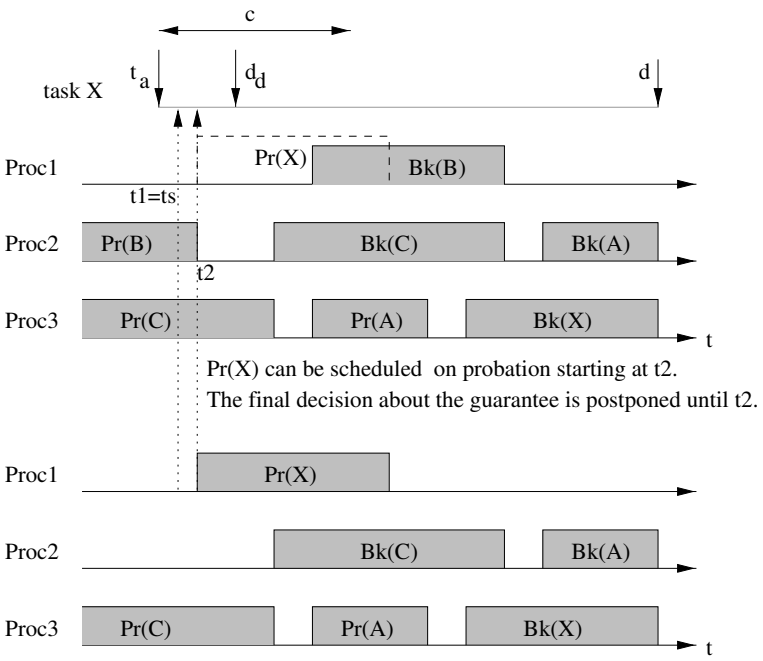

Figure 2: Using the decision deadline to overlap primary and backup.

to the conditions stated above, the primary Pr(X) must not not overlap the backup Bk(B) and can thus not be scheduled into this gap based on the system state in the time between t_a and t_s when the guarantee algorithm executes. However, if the primary (Pr(B))of this backup is scheduled to complete before the absolute decision deadline of the primary under consideration, then the final guarantee decision may be postponed until this point (t_2) and the new primary may be scheduled on probation. The implementation has to assure that in the case of a failure of the primary the backup of which is overlapped both the probationary primary and its backup get deallocated and the rejection is made known to the task requester. As there is no other gap for the primary it is clear that in the example of Fig. 2 the use of the decision deadline improves the rejection ratio.

- Assume the *backup check* finds that the candidate backup slot overlaps with the backup (Bk(A)) of a primary on the same processor as a primary candidate (see Fig. 3). According to the conditions stated above, backups with primaries on the same processor must not overlap, and the backup check would fail based on the system state in the time between t_a and t_s when the guarantee algorithm executes. However, as in the previous case this requirement is no longer necessary if the first primary is scheduled to complete within the decision deadline of the primary candidate. The primary candidate then has to be marked as being on probation and dependent on the first primary. By using the decision deadline and postponing the guarantee decision until t_2 in the example of Fig. 3 task X can be guaranteed while it would have been rejected otherwise.

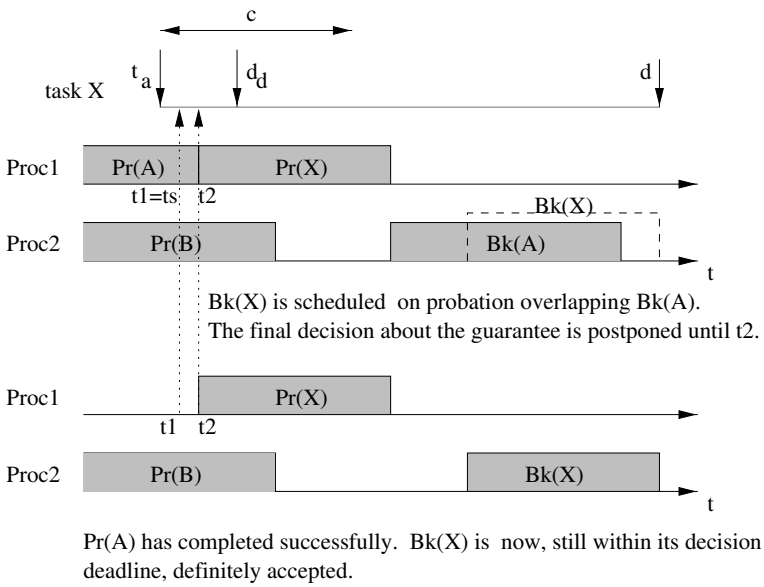

Figure 3: Using the decision deadline to overlap two backups with primaries on the same processor.

If one wanted to use both the slack and the decision deadline modifications together, the implementation would need to assure that a primary (possibly more than one in case of the backup check) on which a probationary slot depends must not be shifted anymore later. This can be most easily done by setting the slack of the respective primary to zero. Experimental results have shown, however, that the combination of both strategies results in a higher rejection ratio than the better of the two strategies alone.

4 Simulation Experiments

The performance of the guarantee algorithms presented above has been investigated by simulations. Both the performance of the algorithms relative to each other as well as the influence of the slack and decision deadline modifications have been studied. The measured variable is the rejection ratio, both with respect to the total number of requested tasks (rr_n) and the total amount of executable task size (rr_c). The independent variable for the simulations is the upper limit on the number of checking comparisons the algorithms are allowed to make, as a processor independent metric for the execution time of the guarantee algorithm. Further variable simulation parameters are the average task deadline, the number of processors, and the average task computation time.

4.1 Assumptions

The actual rejection ratio of a system which uses a guarantee algorithm consists of three components:

1. The ratio of requests which could not even be guaranteed by an ideal algorithm with the same available information.

parameter	name	distribution	values
number of processors	p		5
worst case execution time	c	uniform	c_{mean}=5,10,20,30,50
system load	l		5.0
interarrival time	α	uniform	$\alpha_{mean} = c/l$
window ratio	w_r	uniform	$w_{r,min} = 2$, $w_{r,mean} = 3, 11$
comparison budget	b		20,30,40,50,100,1000

Tab. 4.2: Simulation parameters

2. The ratio of requests which are rejected because the guarantee algorithm does not create the ideal schedule or because it reaches the end of its alloted execution time before a positive guarantee decision can be made, even though the task could be guaranteed in principle.

3. The ratio of requests that are rejected because the guarantee algorithm can not be run for them, e.g. because the worst-case execution time budgeted for the guarantee algorithm is longer than the decision deadline of the request[2].

The objective of the simulation experiments conducted for this work was to compare the performance of the guarantee heuristics (the second component). The first component is also contained in the result, because removing it would have required to construct the ideal schedule for each task set used. As, by using the same task sets for each candidate algorithm, this component is independent of any scheduler strategy and execution time constraint and thus equally biases the performance results for each scheduler, including it does not influence the validity of the algorithm comparison. The third component is not simulated, as its influence is the same for each guarantee algorithm considered here but its actual numerical size depends on the processor speed, task request parameter values, etc.

Including this component in the simulation would only have unnecessarily complicated the interpretation of the results for the relative performance of the algorithms. The consequence of this is, however, that the rejection ratio results in the following sections decrease with increasing algorithm execution time budget which would suggest that the optimum algorithm would have unlimited runtime. This is obviously not true for real systems where the function for rejection rate over comparison budget would have a processor dependent minimum after which the third component would drive the rejection ratio up to 100%.

4.2 Task Set Parameters

In order to obtain as far as possible comparable results the same simulation parameters as in [2] have been chosen. A discrete event simulator has been used to simulate 100 sets of 1000 tasks for each parameter setting. The same task sets have been used for all experiments and the results of each run have been averaged. Table 4.2 list the simulation parameter values that were used. For the simulation it was assumed that no fault actually occurs so that all backups can be deallocated.

The window ratio is defined as $w_r = d_{rel}/c$. In the cases when a decision deadline is used, the decision deadline is set to the smaller value of one tenth of the relative deadline and the slack available for the task ($d_{d,rel} = \min(d_{rel} - 2 * c, 0.1 * d_{rel})$).

[2] Alternatively, a 'best effort' approach could be used which would run the guarantee algorithm anyway, hoping that it might be able to make a decision within the given time. However, this approach would be complicated by its need for an additional timer to abort the guarantee algorithm, and it would also have a negative effect on requests arriving during the 'best effort' run.

4.3 Results

The first four plots investigate the effects of the different modifications of the checking routines on the performance for task sets with tight and wide deadlines.

Fig. 4 shows the rejection ratio of the modified checking routines relative to that of the basic routine as a function of the maximum number of comparisons allowed within the checking routines of the guarantee algorithm for the sequential search processor selection strategy for window ratios of $w_r = 3$ and $w_r = 11$. Simulation results for both performance metrics are given.

As can be seen, for tight deadlines ($w_r = 3$) the modification making use of the decision deadline performs better than both the basic algorithm and the slack testing modification with respect to the number of rejected tasks in the range up to 100 comparisons (rr_n) and further (rr_c). With a window ratio of $w_r = 11$ the advantage of the decision deadline modification is not so pronounced in the low number of comparisons range and the slack modification performs much better as the permissible number of comparisons gets higher. This result can easily be explained: backups are always scheduled ALAP, thus for tight deadlines it is quite likely that a slot with a backup is the obstacle which can be overcome by the decision deadline modification, but not the slack test. For long deadlines, on the other hand, it is more likely that a primary is the obstacle. Primaries can be moved using their slack, while knowledge about the decision deadline does not help.

For tight deadlines the improvement gained by the decision deadline modification is bigger for the rr_c metric than for rr_n. One can also see from the results that under certain conditions the slack modification performs even worse than the basic algorithm.

The results for the load-based processor selection, shown in Fig. 5, exhibit the same basic behavior as those for the sequential search, but with respect to both metrics the decision deadline modification performs better upto even higher comparison budgets.

In Fig. 6 the different processor selection heuristics are compared with each other. As this paper is mainly interested in the behavior of guarantee algorithms with restricted execution time, only the curves for the decisions deadline modification are shown, which were found to offer best performance in this range as demonstrated above. The plots show that for execution time constrained guarantee algorithms load-based processor selection achieves the relatively lowest overall rejection ratio with respect to the number of rejected tasks (2-5% better than sequential search) while sequential search performs better (up to 10% for tight deadlines) with respect to the rejected load. Both are clearly superior to random processor selection.

Additional results of the simulation experiments were that the algorithm performance is not sensitive to a variation of the mean execution time of the tasks of a task set and that with an increase in the number of processors (and the system load correspondingly, to keep the ratio $l/p = 1.0$) in the range from 3 to 7 the relative performance of the decision deadline modification improved slightly relatively to the other algorithms.

5 Conclusion

Several strategies for processor selection for execution time constrained guarantee algorithms have been evaluted. Both the sequential search strategy and the load-based strategy perform much better than random selection. To take the decision deadline of a task request into account has been shown to considerably improve the algorithm performance for tight deadlines.

rejection ratio reference		deadlines			
		tight		wide	
		max. number of comparisons		max. number of comparisons	
		low	high	low	high
number	checking	dd	slack	≈dd	slack
	selection	load-based	seq. search	load-based	≈seq. search
load	checking	dd	dd	≈dd	slack
	selection	seq. search	≈seq. search	≈seq. search	≈seq. search

Tab. 4.3: Summary of findings: Entries indicate the algorithm configuration that performs relatively better for the respective system parameter setting and performance metric (dd=decision deadline modification, slack=slack modification,'≈' indicates that the difference is small).

The simulation results summarized in Tab. 4.3 suggest that the choice of the guarantee algorithm with the best performance depends on the task set and system parameters. For tight deadlines and very short scheduler execution times one would choose load-based processor selection with the decision deadline modification, for less restrictive scheduler execution times sequential search with the slack modification is appropriate. Sequential search also has advantages whenever the rejected load rather than the number of rejected tasks is to be optimized.

In some cases the modification using decision deadline performs 20% better than the modification using slack suggested in [2]. The latter sometimes even performs worse than the basic checking routine while the decision deadline modification gave better results than the basic routine for all the investigated parameter settings.

Further research in this area might explore how an adaptive scheduler which monitors the variation of task request parameters such as the average window ratio over time might self-adjust its guarantee strategy for best performance.

References

[1] KapDae Ahn, Jong Kim, and SungJe Hong. Fault-tolerant real-time scheduling using passive replicas. In *Proceedings Pacific Rim Intern. Symp. on Fault-Tolerant Systems*, 1997.

[2] Sunondo Ghosh, Rami Melhem, and Daniel Mosse. Fault-tolerance through scheduling of aperiodic tasks in hard real-time multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems*, 8(3):272–284, March 1997.

[3] G. Manimaran and C. Siva Ram Murthy. A new scheduling approach supporting different fault-tolerant techniques for real-time multiprocessor systems. *Microprocessors and Microsystems*, 21:163–173, 1997.

[4] John Stankovic, Marco Spuri, Marco Di Natale, and Giorgio Buttazzo. Implications of classical scheduling results for real-time scheduling. *IEEE Computer*, pages 16–25, June 1995.

[5] Tatsuhiro Tsuchiya, Yoshiaki Kakuda, and Tohru Kikuno. A new fault-tolerant scheduling technique for real-time multiprocessor systems. In *Proceedings 2nd International Workshop on Real-Time Comp. Systems and Applications*, 1995.

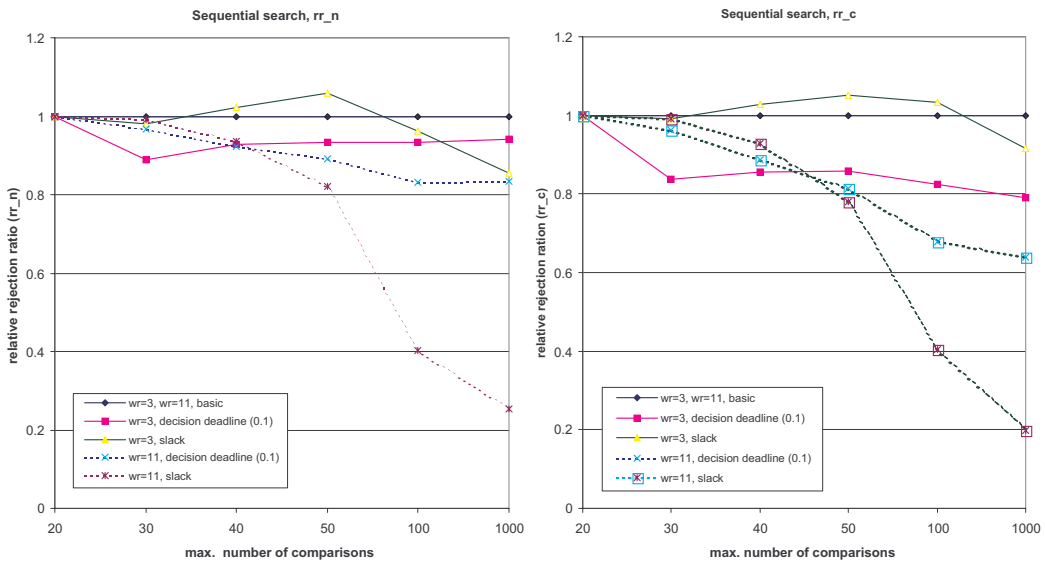

Figure 4: Relative rejection ratio (number and load) for sequential search.

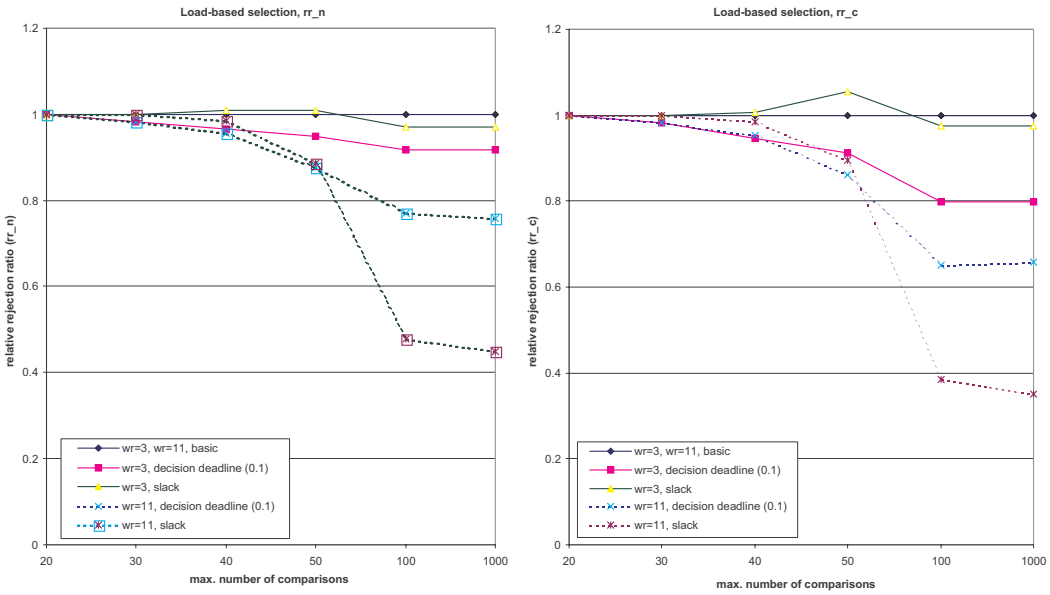

Figure 5: relative rejection ratio (number and load) for load-based selection.

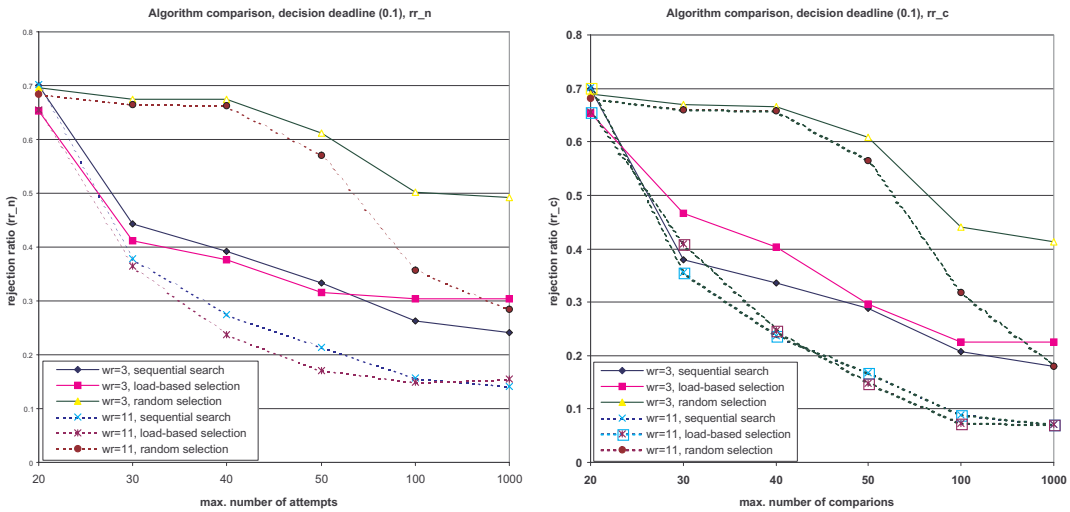

Figure 6: Comparison of the rejection ratios (comp. 1 and 2, see Sec. 4.1) of all three selection algorithms (decision deadline modification only) for window ratios of 3 and 11.