Fault-Tolerant Real-Time Scheduling under Execution Time Constraints

Martin Naedele
Computer Engineering and Networks Laboratory (TIK)
Swiss Federal Institute of Technology Zurich
naedele@tik.ee.ethz.ch

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].

1 Introduction

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 successfully. Also, backups of primaries on different processors may overlap, as all except for one are guaranteed to be deallocated in time. In [6] 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. [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 [5], 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 [6]. 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*. 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 (see Section 3.4.2). 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.

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.

3.1 Processor selection strategies

If the execution time of the scheduler to make a decision is constrained, 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.

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. 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.

3.2 Budgeting

After having found the first 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.

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 (the later of either worst case completion time of the guarantee algorithm or the end of the currently executing task) 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.

3.3.2 Checking routine for backup candidate

The backup check algorithm places the backup in the slot $[d_{abs} - c, d_{abs}]$ and makes no special effort to overload backups [2].

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 Modified checking routines

3.4.1 Using primary slack

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: Slack is local, that is, at most one slot is shifted, as experiments have shown that the shifting of multiple slots only marginally contributes to a decrease in the rejection rate. The slack test is done for each gap immediately, if necessary, not only after all gaps on all processors have been tried without it. 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 request

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. 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. 1). According 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

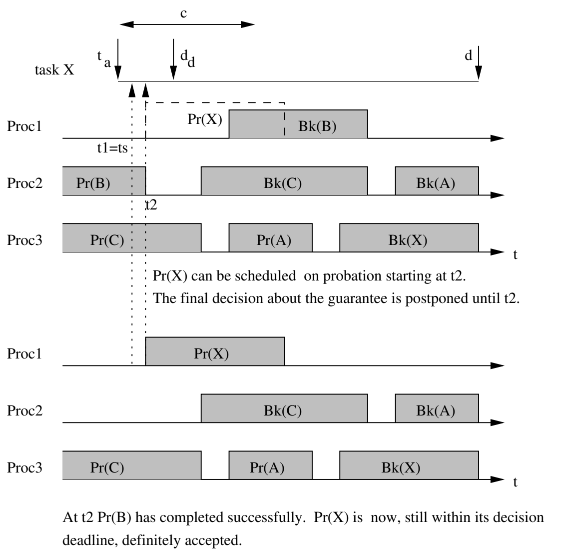

Figure 1. Using the decision deadline to overlap primary and backup.

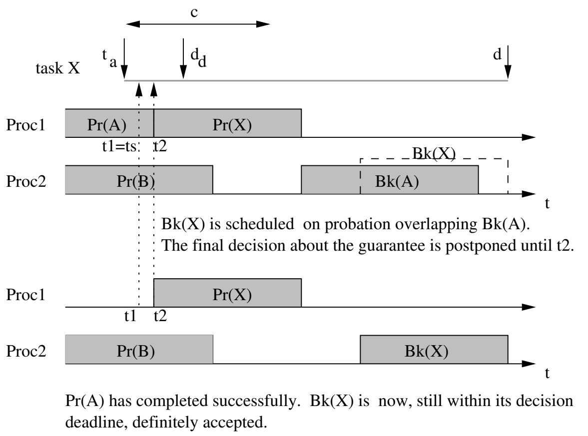

Figure 2. Using the decision deadline to overlap two backups with primaries on the same processor.

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.

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. 2). According to the scheduling rules of [2], 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 new task. The primary candidate then has to be marked as being on probation and dependent on the first primary.

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.

Table 1. Simulation parameters

parameter	name	distrib.	values
num. processors	p		5
w. c. 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

4.1 Task Set Parameters

In order to obtain as far as possible comparable results the same simulation parameters as in [2] have been chosen (see Table 4.1). 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. For the simulation it was assumed that no fault actually occurs so that all backups can be deallocated.

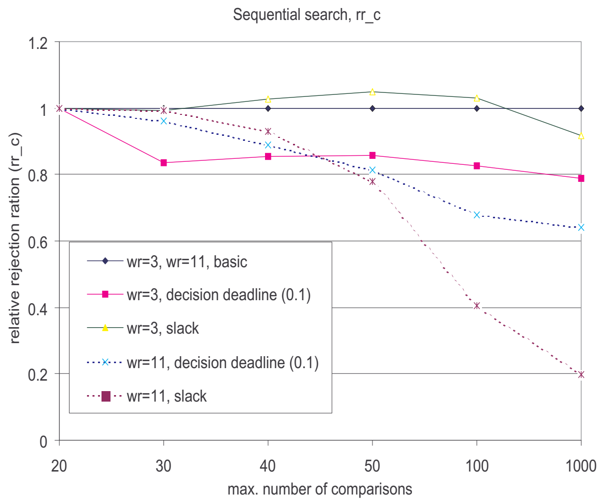

Figure 3. Relative load rejection ratio for sequential search.

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 ($d_{d,rel} = \min(d_{rel} - 2 * c, 0.1 * d_{rel})$).

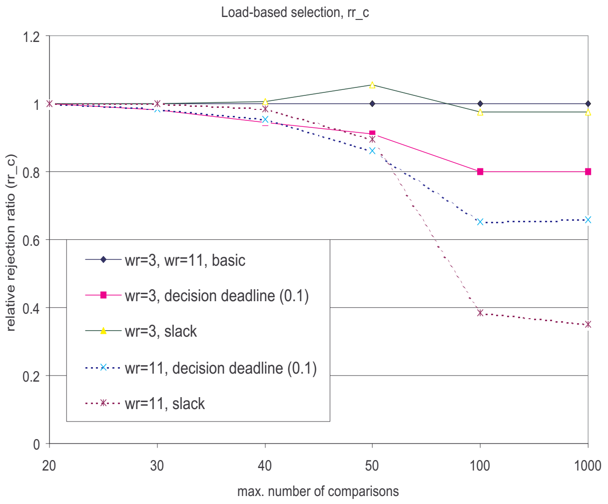

Figure 4. Relative load rejection ratio for load-based selection.

4.2 Results

The first two plots investigate the effects of the different modifications of the checking routines on the performance for task sets with tight and wide deadlines. Fig. 3 shows the load rejection ratio rr_c 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 $w_r = 11$. 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 rejected load. With a window ratio of $w_r = 11$ the advantage of the decision deadline modification is not so pronounced in the low number of comparions 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. 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. 4, exhibit the same basic behavior as those for the sequential search. In Fig. 5 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. 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, see [4]) while sequential search performs better (up to 10% for tight deadlines) with respect to the rejected load (Fig. 5). Both are clearly superior to random processor selection.

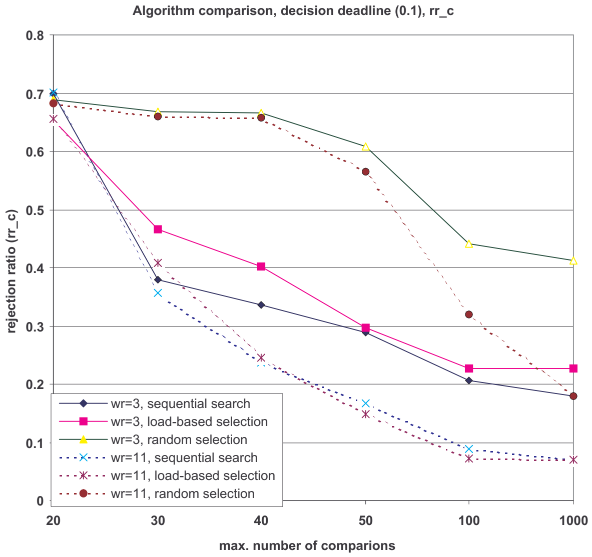

Figure 5. Comparison of the rejection ratios of all three selection algorithms (decision deadline modification only) for window ratios of 3 and 11.

5 Conclusion

Several strategies for processor selection for execution time constrained guarantee algorithms have been evaluted. To take the decision deadline of a task request into account has been shown to considerably improve the algorithm performance for tight deadlines. In some cases the modification using decision deadline performs 20% better than the modification using slack suggested in [2]. More details about our work can be found in [4]. Further research 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] K. Ahn, J. Kim, and S. Hong. Fault-tolerant real-time scheduling using passive replicas. In *Proceedings Pacific Rim Intern. Symp. on Fault-Tolerant Systems*, 1997.

[2] S. Ghosh, R. Melhem, and D. 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, Mar. 1997.

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

[4] M. Naedele. Fault-tolerant real-time scheduling under execution time constraints. TIK-Report 76, ETH Zurich, TIK, 1999.

[5] J. Stankovic, M. Spuri, M. D. Natale, and G. Buttazzo. Implications of classical scheduling results for real-time scheduling. *IEEE Computer*, pages 16–25, June 1995.

[6] T. Tsuchiya, Y. Kakuda, and T. 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.