Fault-Tolerant Deadline-Monotonic Algorithm for Scheduling Hard-Real-Time Tasks [*]

Alan A. Bertossi
Dipartimento di Matematica
Università di Trento
Via Sommarive 14, I-38050 Povo (TN), Italy
bertossi@science.unitn.it

Andrea Fusiello
Dipartimento di Matematica e Informatica
Università di Udine
Via delle Scienze 206, I-33100 Udine, Italy
fusiello@dimi.uniud.it

Luigi Vincenzo Mancini
Dipartimento di Scienze dell'Informazione
Università di Roma "La Sapienza"
Via Salaria 113, I-00198 Roma, Italy
mancini@acm.org

Abstract

This paper presents a new fault-tolerant scheduling algorithm for multiprocessor hard-real-time systems. The so called partitioning *method is used to schedule a set of tasks in a multiprocessor system. Fault-tolerance is achieved by using a combined duplication technique where each task scheduled on a processor has either an active or a passive copy scheduled on a different processor. Simulation experiments reveal a saving of processors with respect to those needed by the usual approach of duplicating the schedule of the non-fault-tolerant case.*

1. Introduction

The demand for complex and sophisticated real-time computing systems continues to increase and fault-tolerance is one of the requirements that are playing a vital role in the design of new real-time systems. In particular, two schemes have been proposed to support fault-tolerance in multiprocessor systems:

- passive replication, where one or more passive copies of a task are allocated either to the same processor or to the backup processors. When a task fails, a passive copy of that task is started, that is a passive copy is executed only in the presence of a failure.
- active replication, where each task is always executed on two or more processors (active copies): if any task fails its active copies will continue to execute.

Many scheduling problems have been found to be NP-complete: most likely, there are no optimal polynomial-time algorithms for them [15]. In particular, scheduling periodic tasks with arbitrary deadlines is NP-complete, even if only a single processor is available [8, 3]. Several heuristic algorithms for scheduling periodic tasks on uniprocessor and multiprocessor systems have been proposed. Liu and Layland [10] proposed the Rate-Monotonic algorithm for uniprocessor systems for the special case that deadlines are equal to periods. The Deadline-Monotonic algorithm [9] generalised the Rate-Monotonic algorithm allowing deadlines less than periods. Subsequent works [6, 1] extended the original scheduling analysis technique by relaxing some constraints on task features.

As for the multiprocessor systems, the Rate-Monotonic algorithm was generalised by Dhall and Liu [5], who proposed the two heuristics called Rate Monotonic-First-Fit (RMFF) and Rate-Monotonic-Next-Fit (RMNF). Recently they were improved upon by Oh and Son [12]. The distributed algorithms proposed in [13] address the issue of dynamic scheduling.

As for fault-tolerant scheduling, some algorithms were presented in [2, 4, 7, 11, 14].

This paper presents the Fault-Tolerant Deadline-Monotonic (FTDM) algorithm which extends the RMFF algorithm to a fault-tolerant multiprocessor system by using a combined duplication technique where each task scheduled on a processor has either a passive or active copy scheduled

[*] This work was supported by the Progetto Speciale 1995, Dip. di Matematica, Università di Trento and by the Italian MURST.

on a different processor. Moreover, the algorithm is further extended by relaxing some constraints on the task features. In particular, tasks can have deadlines less than periods and release jitters. In order to tolerate faults, the FTDM algorithm also determines which kind of duplication (i.e. active or passive) is more suitable for each task. Task copies are assigned to processors following the First-Fit policy, a well-known and efficient heuristic for the Bin-Packing problem. The task copies assigned to a single processor are scheduled according to the Deadline-Monotonic algorithm.

In Section 2, some notations and basic assumptions are introduced. Section 3 deals with the task duplication scheme used to tolerate failures. Section 4 considers the assignment of task copies to processors so as to guarantee fault-tolerance. Section 5 presents an extension of the (one processor) schedulability test to deal with passive backup copies. Section 6 summarises the whole algorithm, where the Deadline-Monotonic policy is used to schedule tasks on the single processors and the First-Fit heuristic is employed for assigning tasks to processors. Section 7 shows some experimental results which reveal a remarkable saving of processors with respect to those needed by duplicating the schedule of the non-fault-tolerant case. Finally, Section 8 terminates the paper with final considerations and open questions.

2. Background and notation

A *periodic* task τ_i is identified by the quadruple (C_i, T_i, D_i, J_i), where:

- C_i is the (worst case) *computation time* of task τ_i;
- T_i is the *invocation period* of task τ_i;
- D_i is the *deadline* of task τ_i;
- J_i is the *release jitter* of task τ_i.

A periodic task gives raise to an infinite sequence of task instances. The $k-th$ instance of task τ_i is invoked at time $(k-1)T_i$ and, in order to meet its deadline, must complete its execution – that requires C_i time units – no later than time $(k-1)T_i + D_i$ (this time requirement is referred to as a *hard deadline*).

A task may not be ready to execute as soon as it is invoked. In fact it may experience a variable delay between its *invocation time* - when the task is logically able to run - and its *release time* - when it is placed into the run queue. The *release jitter* J_i is the worst case delay between τ_i's invocation time and τ_i's release time.

The *completion time* of a task instance is the absolute time when its execution is completed.

The *response time* of a task instance is given by the difference between its completion time and its invocation time. The *worst case response time* of task τ_i, W_i, is defined as the maximum possible response time among all its instances. It is worth noting here that, if $W_i \leq D_i$, then task τ_i is always guaranteed to meet its deadline.

For a given task set $\{\tau_1 \ldots \tau_n\}$, a *scheduling algorithm* establishes an order in which all the periodic task instances are to be executed and, for a multiprocessor system, the specific processor each task is allocated.

A given task set is said to be *schedulable* by the algorithm if all deadlines of all the task instances are met when that algorithm is applied.

From now on, the following assumptions on the task set are made:

- $C_i \leq D_i \leq T_i$ and $J_i \leq D_i - C_i$;
- all tasks are periodic;
- pre-emption is allowed;
- tasks are independent.

The last assumption implies that no precedence relation exists among tasks, and that no inter-process communication or synchronisation is permitted among tasks.

Moreover, the following failure characteristics of the hardware are assumed:

- processors fail in a *fail-stop* manner, that is a processor is either operational or ceases functioning;
- hardware provides fault isolation in the sense that a faulty processor cannot cause incorrect behaviour in a non-faulty processor;
- the failure of a processor is detected by the remaining ones within the closest completion time of a task scheduled on the faulty processor.

3. Duplication scheme

In this section a combined active/passive duplication scheme is presented which copes with one permanent processor failure. In order to achieve fault-tolerance, two copies of the same task, the *primary* and the *backup* copy, are used. Let $\tau_i = (C_i, T_i, D_i, J_i)$ be a primary copy and $\tau_{b(i)} = (C_{b(i)}, T_i, D_i, J_{b(i)})$ be the backup copy of τ_i. A backup copy may be *active* or *passive* depending on the fact that it is always executed or it is executed only in case of a failure, respectively. After assigning tasks to processors (as shown in Section 4), task scheduling is carried on according to the following rules, provided that primary and backup copies of the same task are not assigned to the same processor.

1. In the absence of failures, a processor must execute all primary copies and all active backup copies assigned to it.
2. When processor P_f fails, all the passive backup copies of all primary tasks assigned to P_f must be scheduled to run on the remaining processors while all the active backup copies of non-faulty tasks are no more executed.

In the Case 2 above, assume that the primary task τ_i, which has a passive backup copy $\tau_{b(i)}$, is assigned to processor P_f. If a failure of P_f is detected at time $\theta \in [(k-1)T_i, kT_i]$, then the passive backup copy $\tau_{b(i)}$ of τ_i must be released for the first time (i.e. it is placed in the assigned run queue). This happens either

- at time θ, if the execution of the primary copy τ_i was not completed by θ, namely if $\theta \leq (k-1)T_i + W_i$ in the worst case; or
- at the next invocation time kT_i, if the execution of τ_i was already completed before θ, namely if $\theta > (k-1)T_i + W_i$.

In any case, all the successive releases of $\tau_{b(i)}$ occur every period T_i. In the worst case, a passive copy $\tau_{b(i)}$ is released at time $(k-1)T_i + W_i$ and must complete its execution by time $(k-1)T_i + D_i$. Therefore, if $(D_i - W_i) \geq C_{b(i)}$, then $\tau_{b(i)}$ is scheduled as a passive copy, otherwise, $\tau_{b(i)}$ is scheduled as an active copy.

Backup tasks are characterised as follows:

Active backup. An active copy is always executed in the absence of faults. Thus it behaves as its primary copy, i.e. $\tau_{b(i)} = (C_{b(i)}, T_i, D_i, J_i)$;

Passive backup. A passive copy is assigned to a processor but is not executed until the primary copy fails. A clean way to model the behaviour of task $\tau_{b(i)}$ is to view it as having an invocation period T_i and a release jitter $J_{b(i)} = W_i$, where W_i is the worst-case response time of τ_i. This jitter takes into account the first delayed release, due to a failure, namely: $\tau_{b(i)} = (C_{b(i)}, T_i, D_i, W_i)$.

Note that the primary and backup copies of the same task may have different execution times.

4. Assignment of tasks to processors

In order to determine whether a task τ_i *fits* on a processor (i.e. whether it can be assigned to that processor or not), the schedulability of τ_i must be checked together with all the tasks already assigned to that processor, both in the no-failure case and in the failure case. Let us define:

- $primary(P_j)$ is the set of all the primary copies assigned to processor P_j;
- $backup(P_j)$ is the set of all the backup copies assigned to processor P_j;
- $backup|active(P_j)$ is the set of all the active backup copies assigned to processor P_j;
- $backup|passive(P_j)$ is the set of all the passive backup copies assigned to processor P_j;
- $recover(P_j, P_f)$ is the set of all the backup copies assigned to P_j such that their primary copies are all assigned to P_f, in symbols: $recover(P_j, P_f) = \{\tau_h \in backup(P_j) | P(\tau_h) = P_f\}$, where $P(\tau_h)$ denotes the processor where the primary copy of τ_h is assigned.

In order to assign a task to a processor, some of the following conditions must be satisfied. The conditions to be considered for each task depend on the *status* (i.e. primary, active backup or passive backup) of the task. Each condition requires to test the schedulability of a task set on a single processor, which can be accomplished by means of the *Completion Time Test* illustrated in Section 5. In the following, P_f denotes a failed processor.

Primary. To assign a primary task τ_i to P_j, the task set

$$\{\tau_i\} \cup primary(P_j) \cup backup|active(P_j)$$

must be schedulable (this is to take into account the normal situation i.e. the no-failure case), and the task set

$$\{\tau_i\} \cup primary(P_j) \cup recover(P_j, P_f) \quad \forall P_f \neq P_j$$

must be schedulable as well (which takes into account the possible failure of another processor).

Active backup. To assign an active backup task $\tau_{b(i)}$ to P_j, assuming that the primary copy τ_i is assigned to processor $P(\tau_i) \neq P_j$, the task set

$$\{\tau_{b(i)}\} \cup primary(P_j) \cup backup|active(P_j)$$

must be schedulable (no-failure case), and the set

$$\{\tau_{b(i)}\} \cup primary(P_j) \cup recover(P_j, P_f)$$

with $P_f = P(\tau_i)$ must be schedulable too (failure case). Note that, if $P_f \neq P(\tau_i)$, then $\tau_{b(i)}$ does not need to be executed anymore.

Passive backup. To assign a passive backup task $\tau_{b(i)}$ to P_j, assuming that the primary copy τ_i is assigned to $P(\tau_i) \neq P_j$, only the task set

$$\{\tau_{b(i)}\} \cup primary(P_j) \cup recover(P_j, P_f)$$

with $P_f = P(\tau_i)$ must be schedulable (failure case). Note again that, if $P_f \neq P(\tau_i)$, then $\tau_{b(i)}$ does not need to be executed anymore.

5. Schedulability test

This section presents a test needed to check whether a given task set is schedulable or not on a single processor by a certain algorithm. In particular, *priority-driven* scheduling algorithms will be considered. Such algorithms assign priorities to tasks according to some policy, and at each instant of time, the processor executes the highest priority task ready to run, suspending – if necessary – a lower priority task.

Joseph and Pandya [6] first derived an exact analysis to find the worst-case response time for a given periodic task on a single processor, assuming fixed priority, independent tasks and deadlines no greater than periods ($D_i \leq T_i \; \forall i$). Hereafter, the analysis found in [16] is used. According to the authors, the worst case response time W_i of a task τ_i satisfies the following equation:

$$W_i = W_i^* + J_j,$$

$$W_i^* = C_i + \sum_{j \in hp(i)} C_j \left\lceil \frac{W_i^* + J_j}{T_j} \right\rceil, \qquad (1)$$

where $hp(i)$ is the set of all tasks with higher priority than τ_i. The smallest value of W_i^* satisfying Equation (1) can be obtained by iteration:

$$W_i^*(k+1) = C_i + \sum_{j \in hp(i)} C_j \left\lceil \frac{W_i^*(k) + J_j}{T_j} \right\rceil \qquad (2)$$

starting from $W_i^*(0) = 0$. The iteration stops as soon as $W_i^*(k+1) = W_i^*(k)$. It is easy to see that each instance of τ_i will meet its deadline iff $W_i \leq D_i$. This is referred to as the *Completion Time Test*. Observe that the schedulability of lower priority tasks does not guarantee the schedulability of higher priority tasks. Therefore, in order to check the schedulability of a set of tasks, each task, starting from that with highest priority, must get through the Completion Time Test.

6. The algorithm

In order to schedule a set of periodic tasks in a multiprocessor system, the partitioning method [5] is used. Tasks are partitioned into groups so that each group of tasks can be feasibly scheduled on a single processor. This method involves two algorithms: an assignment algorithm of tasks to processors, and a scheduling algorithm of tasks assigned on each single processor.

Tasks are assigned to processors following the First-Fit policy. Tasks are picked one at a time and assigned to the first processor in which they fit. When a task cannot fit in any processor, a new processor is added. The conditions that a task has to verify in order to fit in a processor are those stated in Section 4. It is worth noting that tasks cannot be picked in any order. Indeed, the release jitter of a passive backup copy is equal to the worst case response time of its primary copy. This response time is computed after the primary task copy is assigned to a processor and is fixed only if no higher priority task is assigned to that processor. Therefore, tasks must be picked by decreasing priority order and each passive copy of a task must follow the primary copy of the same task.

Tasks are scheduled on each processor according to the known *Deadline-Monotonic* [9] fixed priority assignment policy. Priorities are assigned inversely to task deadlines; hence, τ_i receives a higher priority than τ_j if $D_i < D_j$. Leung and Whitehead [9] proved that this is an optimal scheduling algorithm among the fixed-priority ones. This means that every task set that is schedulable by some fixed-priority policy is also schedulable by the Deadline-Monotonic algorithm. Schedulability on each processor is checked by means of the Completion Time Test.

7. Simulation results

To evaluate the behaviour of the proposed Fault-Tolerant Deadline-Monotonic (FTDM) algorithm, simulation experiments for large task sets with $100 \leq k \leq 500$ tasks are performed. The task periods T_i are selected to be uniformly distributed in the interval $2 \leq T_i \leq 500$. The execution times C_i of the tasks are also taken from a uniform distribution in the interval $1 \leq C_i \leq \alpha T_i$, where α is a parameter representing the maximum task utilisation (since C_i/T_i is the utilisation factor of task τ_i, $\alpha \geq \max_i C_i/T_i$). In each of the following experiments the maximum utilisation factor α is chosen equal to 0.2, 0.4, 0.8.

The performance metric in all experiments is the percentage of additional processors required by the FTDM algorithm. In particular the overhead for the provision of fault tolerance is given by the following ratio:

$$\mathrm{overhead} = \frac{N - M}{M} \qquad (3)$$

where N is the number of processors required by the FTDM algorithm to schedule a task set consisting of primary and backup copies, and M is the number of processors required to schedule a task set with identical primary copies and no backup copies. The result of each experiment is the average value of the above ratio over 30 independent trials.

Firstly, the behaviour of the proposed algorithm is evaluated when the deadlines are equal to the periods, the jitters

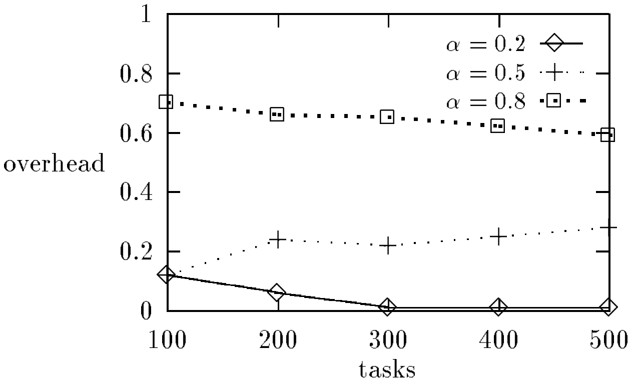

Figure 1. FTDM overhead with respect to RMFF for the primary copies. $D_i = T_i$**.**

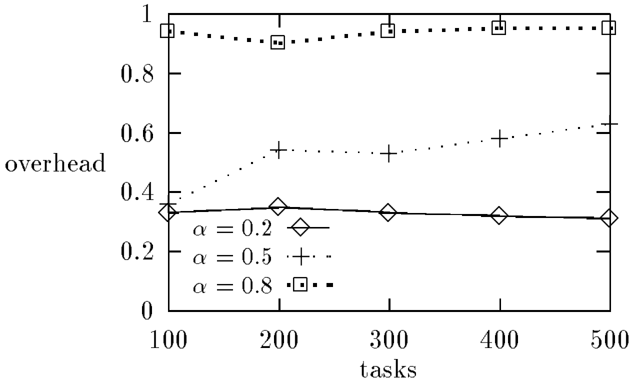

Figure 2. FTDM overhead with respect to RMFF with Completion Time Test. $D_i = T_i$**.**

are equal to zero, and the backup execution times are equal to the primary execution times. The outcome of the simulation experiments are shown in Figures 1 and 2. In Figure 1, the value M in the formula (3) is given by the widely-used RMFF algorithm [5]. The overhead introduced by algorithm FTDM grows with α. For small values of α the algorithm gains benefits from the passive duplication, and a small number of additional processors is required in order to guarantee fault-tolerance. As α increases (and C_i approaches T_i) more and more backup copies become active and the fault-tolerant schedule requires more additional processors. Summarising, Figure 1 suggests that the FTDM algorithm can provide fault-tolerant schedules which require from 40% to 99% less processors than those used by the active duplication of RMFF scheduling.

In order to have a pessimistic evaluation of the overhead it is useful to compare the FTDM schedule with the schedule for primary copies produced by the RMFF algorithm where the Completion Time Test is used on each processor instead of the $\ln 2$ bound used in [5]. The result of the experiment is shown in Figure 2. A comparison of Figure 1 and 2 shows that, as expected, much less processors are used in this case to schedule the primary tasks. However, the results of both experiments follow a similar pattern, namely, the overhead of the FTDM algorithm is low for smaller values of α, while as α increases the performance of the FTDM algorithm decreases.

The fault-tolerant algorithm proposed works also when deadlines are shorter than periods. In order to characterize the average-case performance of the FTDM algorithm in this case, the overhead defined in the formula (3) is evaluated computing M by applying the *Deadline Monotonic First Fit* (an obvious extension of RMFF) algorithm with the Completion Time Test. This is because in this way we consider a more pessimistic situation and hence the outcome of the experiments can be considered as a worst-case performance of FTDM algorithm.

The task periods T_i and the execution times C_i of the tasks are uniformly distributed in the same interval defined above. The jitters are equal to 0, the backup execution times are equal to the primary execution times, and the deadlines D_i are taken equal to $D_i = \min\{\beta C_i, T_i\}$, where β is a parameter determining the length of the deadline.

Figure 3 and 4 report the outcome of the simulation experiments for $\beta = 3$ and $\beta = 6$ respectively. It seems that the performance of the FTDM algorithm decreases when deadlines are much shorter than the periods. In particular, the experiments show that such degradation is more evident when α and D_i are small. For example, when $\alpha = 0.2$ and $\beta = 3$, see Figure 3, about 50% additional processors are required by the FTDM schedule against 30% in the case of $\alpha = 0.2$ and deadlines equal to periods.

However, the performance improves rapidly by increasing D_i. Indeed, starting by $\beta = 6$, see Figure 4, the performance obtained is comparable to that of Figure 1, where deadlines are equal to periods.

8. Concluding remarks

The present paper considered the problem of preemptively scheduling a set of independent periodic tasks on multiprocessor systems, assuming deadlines less than periods, positive release jitters and the presence of a processor failure.

As shown in Section 7, the cost of the fault-tolerance, in terms of the additional number of processors required, is low - the FTDM algorithm requires from 40% to 99% less processors than those used by the active duplication of RMFF schedules.

The fault-tolerant scheme presented tolerates a perma-

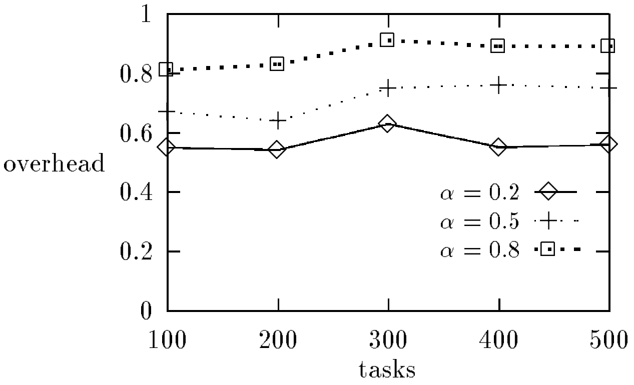

Figure 3. FTDM overhead. $\beta = 3$**.**

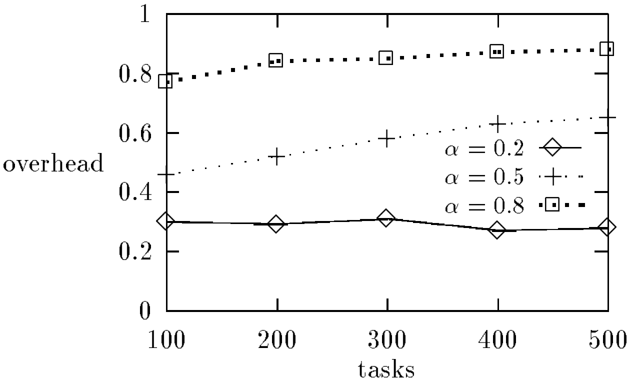

Figure 4. FTDM overhead. $\beta = 6$**.**

nent processor failure but it can be extended also to tolerate transient failures. In this case, if an acceptance test is used to detect transient failure, a passive backup copy and its primary copy could share the same processor.

Future work will be aimed at relaxing some constraints upon the task set. For instance, a relevant issue is the fault-tolerant scheduling of real-time tasks subject to precedence constraints and resource requirements.

References

[1] N. Audsley, A. Burns, M. Richardson, and A. Wellings. Hard-real-time scheduling: The deadline-monotonic approach. In *Proc. of the 8th Workshop on Real-Time Operating Systems and Software*, May 1991.

[2] J. Bannister and K. S. Trivedi. Tasks allocating in fault-tolerant distributed systems. *Acta Informatica*, 20:261–281, 1983.

[3] S. Baruah, L. Rosier, and R. Howell. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Real-Time Systems*, 2:301–324, 1990.

[4] A. Bertossi and L. Mancini. Scheduling algorithms for fault-tolerance in hard-real-time systems. *Real-Time Systems*, 7(3):229–245, 1994.

[5] S. Dhall and C. Liu. On a real time scheduling problem. *Operations Research*, 26(1):127–141, January-February 1978.

[6] M. Joseph and P. Pandya. Finding response times in a real-time system. *The Computer Journal*, 29(5):390–395, October 1986.

[7] C. Krishna and K. Shin. On scheduling tasks with a quick recovery from failure. *IEEE Transactions on Computers*, 35(5):448–454, May 1986.

[8] J.-T. Leung and M. Merril. A note on preemptive scheduling of periodic real-time tasks. *Information Processing Letters*, 11(3):115–118, 1980.

[9] J.-T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic real-time tasks. *Performance Evaluation*, 2(4):237–250, December 1982.

[10] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[11] Y. Oh and S. Son. Enhancing fault-tolerance in rate-monotonic scheduling. *Real-Time Systems*, 7(3):315–329, 1994.

[12] Y. Oh and S. Son. Allocating fixed-priority periodic tasks on multiprocessor systems. *Real-Time Systems*, 9:207–239, 1995.

[13] K. Ramamritham, J. Stankovic, and W. Zhao. Distributed scheduling of tasks with deadlines and resource requirements. *IEEE Transactions on Computers*, 38(8), August 1989.

[14] J. A. Stankovic. Decentralized decision making for task reallocation in hard-real-time systems. *IEEE Transactions on Computers*, 38(3):341–355, March 1989.

[15] J. A. Stankovic, M. Spuri, M. D. Natale, and G. Buttazzo. Implications of classical scheduling results for real-time systems. *IEEE Computer*, 28(6):16–25, June 1995.

[16] K. Tindell, A. Burns, and A. Wellings. An extendible approach for analysing fixed-priority hard-real-time tasks. *Real-Time Systems*, 6(2):133–151, March 1994.