Scheduling Hard-Real-Time Tasks with Backup Phasing Delay

Alan A. Bertossi[1] Luigi V. Mancini[2] Alessandra Menapace
[1] Dipartimento di Scienze dell'Informazione
Università di Bologna, Bologna, Italy
bertossi@cs.unibo.it
[2]Dipartimento di Informatica
Università di Roma "La Sapienza", Roma, Italy
lv.mancini@di.uniroma1.it

Abstract

This paper presents several fault-tolerant extensions of the Rate-Monotonic First-Fit multiprocessor scheduling algorithm handling both active and passive task copies. In particular, the technique of backup phasing delay is used to reduce the portions of active task copies that must be always executed and to deallocate active task copies as soon as their primary task copies have been successfully executed. It is also shown how to employ this technique while considering passive task duplication so as to over-book each processor with many passive task copies, assigning tasks to processors in such a way that tasks with equal or multiple periods have a high chance to be assigned to the same processor, and partitioning the processors into groups to avoid the mix of primary, active, and passive task copies on the same processor.

Extensive simulations reveal a remarkable saving of both the overall number of processors used and the total computation time of the schedulability test (achieved especially by two new algorithms, called ARR3 *and* S-PR-PASS*) with respect to previously proposed algorithms.*

Keywords: fault-tolerance, hard-real-time systems, multiprocessors, periodic tasks, Rate-Monotonic scheduling.

1 Introduction

Hard-real-time computing systems are widely used in our society, e.g. for periodically executing monitor and control functions. Such systems are characterized by periodically occurring tasks which have to be preemptively scheduled on identical processors in such a way that each task occurrence has to be completely executed by a hard deadline, which often coincides with the end of the task period. Since the purpose of a hard-real-time system is to provide time-critical services to its environment, the system must be capable of providing such a service even in the presence of failures. Thus fault-tolerance plays a vital role in the design of hard-real-time systems.

Since the hard-real-time scheduling problem is NP-hard, even if only a single processor is available [18], several heuristics for scheduling periodic tasks on uniprocessor and multiprocessor systems have been proposed. Liu and Layland [20] proposed the *Rate-Monotonic* (RM) algorithm, a fixed-priority, preemptive algorithm for a single processor, where the task with shortest period has the highest priority: the ready task with highest priority is executed on the processor suspending, if necessary, a running task with lower priority. The *Completion Time Test* (CTT), devised by Joseph and Pandya [14], is also used for checking schedulability of a set of fixed-priority tasks on a single processor. RM has been generalized to multiprocessor systems by Dhall and Liu [11], who proposed the *Rate-Monotonic First-Fit* (RMFF) heuristic, where tasks are considered by RM priority order and assigned to the first processor in which they fit.

A simple technique to achieve fault-tolerance in hard-real-time multiprocessor systems consists in replicating on at least two sets of processors the schedule obtained for the non-fault-tolerant case (i.e. by employing active duplication for all tasks), but this approach, studied by Oh and Son [21] for RM scheduling, presents the disadvantage of requiring many processors. They show that on the average 40% extra processors are required compared to the lower bound needed to schedule all the primary and active backup copies. In the case of multiprocessors, the use of passive task copies has the advantage of over-booking the processors: many passive copies of primary copies assigned to different processors can be scheduled on the same processor during the same time interval - under the assumption of a single processor failure, only one of such passive copies will be actually executed. Ghosh, Melhem and Mosse [12] studied this technique in the case of aperiodic non-preemptive tasks and achieved high acceptance ratio postponing as much as possible the execution of passive backup copies. A heuristic was introduced in [5], called *Fault-Tolerant Rate-Monotonic First-Fit* (FTRMFF), which extends the RMFF algorithm by combining in the same schedule both active and passive task duplication, thus exploiting the advantages of

both types of duplication.

The present paper considers the problem of preemptively scheduling a set of independent periodic tasks on a multiprocessor system. In particular, we extend the RMFF scheduling algorithm to tolerate permanent processor failures that can affect more tasks at a time. In other words when a processor fails, all the primary copies executed on that processor are considered to be failed. The fault-tolerance is provided by combining passive and active duplication, preferring passive duplication whenever possible. Furthermore, we consider the *phasing delay* technique to extend the FTRMFF heuristic and to improve the schedule of the backup copies. The *phasing delay* technique, introduced in [22, 1], allows the delay of a passive copy until the completion time of its primary copy so that the passive copy is executed only when the primary task fails. Moreover, the phasing delay enables to reduce the length of the worst case overlapping interval between a primary and its active copy, so that only a small fraction of the computation of the active copy must be executed in the absence of failure of the primary copy. Indeed, a disadvantage of active task duplication is an excess of redundant computation due to useless portions of active copies which are executed after their corresponding primary copies have been successfully completed. Therefore, one could deallocate the unexecuted portion of an active copy as soon as a primary copy has been successfully completed, thus reusing the processor for the execution of another task. This can be achieved by forcing the scheduling of the active copies as late as possible within their periods, thus reducing the initial portions of active copies that must be always executed before the completion of their corresponding primary copies. Note that although the techniques of postponing, over-booking and time deallocation of passive copies have been proposed in [12], their application to active copies has not been investigated to our knowledge so far. Indeed, a passive backup copy can be simply delayed until the completion time of its primary copy to maximize the over-booking and deallocation of passive copies. However, the over-booking and deallocation of active copies are more difficult since they require to establish exactly the computation overlap between an active copy and its primary copy. Such a computation overlap is determined by the unrelated preemptions caused by higher priority tasks. This paper considers the phasing delay, the over-booking and the time deallocation of active backup copies and generalizes these optimizations in a task set composed of both active and passive copies, thus combining together all the techniques proposed in [22, 5, 12].

In order to reduce further both the number of processors needed and the running time for task assignment, it is also shown how to combine the phasing delay with other known techniques. For instance, instead of assigning tasks to processors following the Rate-Monotonic order, one could consider other assignment orders, such as the S priority, proposed by Burchard, Liebeherr, Oh and Son [6], which has the effect of grouping together on the same processor those tasks whose periods are equal or multiple, since this could produce a more compact schedule and thus employs less processors. However, the computation time of the schedulability test is related to the number of invocations of CTT required to assign each task. Thus, the processors can be partitioned into groups so as to avoid the mix of primary, active, and passive copies on the same group and reduce the overall number of CTT invocations. Finally, when all tasks have low utilization factors, one could employ only passive duplication.

The remaining part of this paper is structured as follows. Section 2 gives the notation, the fault-tolerant system assumptions, and a formal definition of the scheduling problem. Section 3 deals with the *Active Resource Reclaiming* (ARR) strategy, which exploits the backup phasing delay to schedule the active copies as late as possible within their periods and to permit the deallocation of active copies as soon as their primary copies have been successfully executed. In order to reduce the number of CTT invocations, the active resource reclaiming technique is combined with a partitioning of the processors into at most three groups (algorithms ARR1, ARR2 and ARR3). Section 4 introduces the S-PRIORITY algorithm, where processors are also partitioned into groups but tasks are assigned to processors by following the S priority [6] in such a way that tasks with equal or multiple periods have a high chance to be assigned to the same processor. In Section 5, the PASSIVE algorithm is presented, which considers only passive task duplication, and then the S-PRIORITY and PASSIVE algorithms are combined to derive the S-PR-PASS algorithm. Section 6 reports extensive simulations where all the algorithms proposed in this paper are compared. The simulation results show a remarkable saving of both the number of processors used and the total computation times achieved especially by the ARR3 and S-PR-PASS algorithms.

2 Problem formulation and assumptions

This section gives a formal definition of the scheduling problem and a precise specification of the fault-tolerance model. A *periodic* task τ_i is characterized by the tuple:

$$\tau_i = (R_i, T_i, C_i, D_i)$$

where R_i is the *release* time, that is, the time of the first invocation, T_i is the invocation (or request, or arrival) *period*, C_i is the (worst case) *computation time*, and D_i is the *deadline*. The ratio $U_i = \frac{C_i}{T_i}$ is called the *load* (or *utilization*) of task τ_i and cannot be greater than 1 for the task to be schedulable. Each periodic task leads to an infinite sequence of occurrences. The $k-th$ occurrence of task τ_i is ready for

execution at time $R_i + (k-1)T_i$ and, in order to meet its deadline, must complete its execution — that requires C_i time units — no later than time $R_i + (k-1)T_i + D_i$.

To better understand the algorithms presented in this paper, two results due to Liu and Layland [20] are recalled about static priority-driven scheduling. Consider a set $\{\tau_1, \ldots, \tau_n\}$ of periodic tasks, indexed by decreasing priority, such that $D_i = T_i$ for $i = 1, \ldots, n$.

Theorem 2.1 *The longest response time (i.e. difference between completion and release time) for* any *occurrence of a task τ_i occurs when it is requested at a critical instant, that is, simultaneously with all higher priority tasks (e.g. when $R_1 = R_2 = \ldots = R_i$).*

Theorem 2.2 *A periodic task set can be scheduled by a fixed-priority preemptive algorithm provided that the deadline of the* first *request of each task starting from a critical instant is met.*

Due to the above results, in the following all the first arrival times of the tasks are assumed to be 0, i.e. $R_1 = R_2 = \ldots = R_n = 0$, since this assumption takes care of the worst possible case. As a consequence, to check the schedulability of any task τ_i, it is sufficient to check whether τ_i is schedulable within its first period $[0, T_i]$ by its first deadline D_i, when it is scheduled with all higher priority tasks $\{\tau_1, \ldots, \tau_{i-1}\}$. Moreover, we also assume, as in [11] and [20], that all the tasks are *independent*, namely, there is no precedence relation among them. For the sake of simplicity, we assume that all the task deadlines coincide with the end of the task periods, in symbols: $D_i = T_i$. However, the above results and the *Completion Time Test* (CTT) have been extended in [23] relaxing this assumption, hence the fault tolerant algorithms presented here could also be generalized when $D_i \leq T_i$.

Below, the basic formulation of CTT is reported which will be used to test whether a task τ_h can be assigned to a given processor. CTT determines the minimum W_h such that:

$$W_h = C_h + \sum_{j \in hp(h)} C_j \left\lceil \frac{W_h}{T_j} \right\rceil \qquad (1)$$

where $hp(h)$ denotes the subset of tasks with higher priority than τ_h already assigned to the same processor, and W_h denotes the *worst-case response time* of τ_h

Remark 2.3 *A task τ_h can be scheduled on a processor together with all higher priority tasks if and only if:*

$$W_h \leq T_h.$$

Given *n* periodic independent tasks $\{\tau_1, \ldots, \tau_n\}$, the fault-tolerant scheduling problem considered in the present paper consists in finding an order in which all the periodic task occurrences have to be executed on a set of identical processors (using preemption and backup copies, when necessary) so as to meet all the task deadlines, even in the presence of a processor failure, and to minimize the total number *m* of processors used.

As for the fault-tolerance model, we assume that the failure characteristics of the multiprocessor system are the following: (1) processors fail in a fail-stop manner, that is a processor is either operational (i.e. non-faulty) or ceases functioning; (2) all non-faulty processors can communicate with each other; (3) a faulty processor cannot cause incorrect behavior in a non-faulty processor (that is, processors are independent as regard to failures); and (4) the failure of a processor P_f is detected by the remaining non-faulty processors after the failure, but within the instant corresponding to the closest task completion time of a task scheduled on P_f. Each (primary) task is assumed to have a backup copy with the same parameters. In particular, for the sake of simplicity, it is assumed that each backup copy has the same computation time as its primary copy, and that a single permanent processor failure has to be tolerated, unless otherwise stated. However, the results could be generalized to tolerate many permanent processor failures even when the two copies of the same task have different computation times.

The algorithms presented here introduce fault-tolerance by extending RMFF in a natural way. Two copies for each task are used, a *primary* copy and a *backup* copy, such that each backup copy has the same RM priority as its corresponding primary copy. The task set thus becomes $\{\tau_1, \tau_2, \ldots, \tau_{2i+1}, \tau_{2i+2}, \ldots, \tau_{2n-1}, \tau_{2n}\}$, where τ_{2i+1} and τ_{2i+2} denote the primary copy and the backup copy, respectively, of the same task.

A primary copy τ_{2i+1} of a task is always executed, while its backup copy τ_{2i+2} is executed according to its status, which can be active or passive. If the status is active, then τ_{2i+2} is always executed, while if it is passive, then τ_{2i+2} is executed only when the primary copy fails. In other words, although both active and passive copies of the primary tasks are statically assigned to processors, passive copies are actually executed only when a failure of the corresponding primary copy occurs, see [5] for further details.

3 The Active Resource Reclaiming Algorithms

This section deals with the *Active Resource Reclaiming* (ARR) algorithms which use the phasing delay of active copies with the purpose of: (1) forcing the active copies to be scheduled as late as possible within their periods, so as to minimize their overlap with their corresponding primary copies; (2) permitting the deallocation of unfinished active

copies as soon as their corresponding primary copies have been successfully executed; (3) reusing the free processor time for the execution of other tasks.

The algorithms presented in the rest of this paper, while assigning the tasks to the processors, determine whether a backup copy has to be passive or active. Indeed, the CTT returns the worst-case response time W_{2i+1} of each primary copy τ_{2i+1} on the processor it is assigned to. If τ_{2i+1} fails, there are $T_{2i+1} - W_{2i+1}$ time units for recovering the task. Two cases may arise:

1. If $T_{2i+1} - W_{2i+1} \geq C_{2i+2}$, then the time interval between the finish time of the primary copy and the end of the period is large enough to completely execute the backup copy, and thus τ_{2i+2} is chosen to be passive.

2. If $T_{2i+1} - W_{2i+1} < C_{2i+2}$, the backup copy τ_{2i+2} is chosen to be active, since its execution must begin before the finish time of its primary copy.

We now introduce with the help of the example in Figure 1 the scheduling strategy used for active copies. In the best case, a primary copy τ_{2i+1} is executed with no preemption at the beginning of its period, while its active copy τ_{2i+2} is executed with no preemption at the end of its period as shown in the figure. Since τ_{2i+2} is active, $C_{2i+2} > T_{2i+1} - W_{2i+1}$ and thus a minimum computational overlap of $2C_{2i+2} - T_{2i+1} > 0$ time exists between τ_{2i+1} and τ_{2i+2}. Therefore, the active copy τ_{2i+2} can be executed either for $2C_{2i+2} - T_{2i+1}$ time, if the primary copy τ_{2i+1} is successfully completed, or for C_{2i+2} time, if the primary copy fails. Note that, in the absence of failures, the remaining $T_{2i+1} - C_{2i+1}$ time units of the unfinished active copy can be reused.

In general, the length of the computation overlap is larger than the minimum $2C_{2i+2} - T_{2i+1}$ and it is difficult to be determined exactly, due to the scheduling of higher priority tasks which cause unrelated preemptions of both τ_{2i+1} and τ_{2i+2}. For this reason, the *worst case overlapping interval* between τ_{2i+2} and its primary copy τ_{2i+1} is introduced as an upper bound. Such interval starts from the release time of the active copy τ_{2i+2} and ends at the worst case response time of the primary copy τ_{2i+1}. To reduce the length of the worst case overlapping interval, the only way consists in delaying the release time of the active copy τ_{2i+2}. This is the rule employed by the ARR strategy, which determines in advance during the schedulability test the time interval an active copy can be delayed and when it can be deallocated in the absence of failures of the primary copy. In order to determine the value of the *phasing delay* $dmax_{2i+2}$ of an active copy, consider the assignment of tasks τ_{2i+1} and τ_{2i+2}:

1. A primary copy τ_{2i+1} is assigned, say, to processor P_h and thus its worst-case response time W_{2i+1} is computed by CTT. Assume that $T_{2i+1} - W_{2i+1} < C_{2i+2}$ and thus the backup copy τ_{2i+2} is active.

2. The active copy τ_{2i+2} is assigned, say, to $P_k \neq P_h$ and its worst-case response time is W_{2i+2}. The parameter $dmax_{2i+2}$, which gives the maximum phasing delay applicable to τ_{2i+2} without τ_{2i+2} misses its deadline, is:
$$dmax_{2i+2} = T_{2i+1} - W_{2i+2}.$$

3. Let C'_{2i+2} denote the computation time of a portion of the active copy that has to be always executed. Such a time is equal to the minimum between C_{2i+2} and the worst case overlapping interval:
$$C'_{2i+2} =$$
$$min\{C_{2i+2}, W_{2i+1} - dmax_{2i+2}\} \geq 2C_{2i+2} - T_{2i+1}.$$

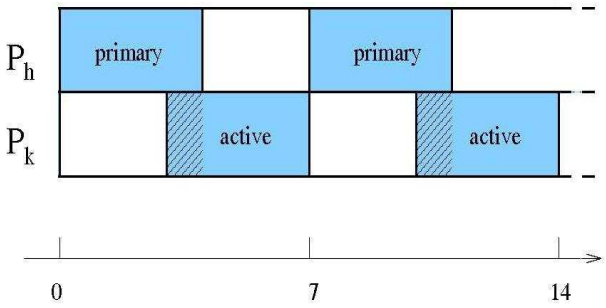

Figure 1. A schedule for the primary copy and the active copy of the same task having minimum computation overlap.

3.1 Maximum phasing delay of an active copy

In order to justify the introduction of the parameter $dmax_{2i+2}$ to delay the execution of an active copy τ_{2i+2}, assume that W_{2i+2} has been already computed for the worst possible case which, by Theorems 2.1 and 2.2, arises when the task copy τ_{2i+2} starts in phase with all the task copies already assigned to the same processor, say P_k, (see the next subsection for all the details on how W_{2i+2} is actually computed).

If $W_{2i+2} \leq T_{2i+1}$ then τ_{2i+2} can be scheduled on P_k and delayed by $dmax_{2i+2} = T_{2i+1} - W_{2i+2}$ units. Note that such a delay has the sole purpose of reducing the worst case overlapping interval between an active copy and its primary copy during their execution and cannot cause an incorrect schedule on P_k. Indeed, a phasing delay of $dmax_{2i+2} > 0$ for the first occurrence (and also the successive occurrences) of τ_{2i+2} can cause neither a response time greater than W_{2i+2}, which already includes the worst possible case, nor

a change in the task priorities. If the first invocation of τ_{2i+2} has a $dmax_{2i+2}$ delay (equal to $T_{2i+1} - W_{2i+2}$), then the first occurrence of τ_{2i+2} will be completed no later than:

$$dmax_{2i+2} + W_{2i+2} = (T_{2i+1} - W_{2i+2}) + W_{2i+2} = T_{2i+1}.$$

Thus the completion of the active copy τ_{2i+2} is guaranteed by the end of the period of its primary copy τ_{2i+1}, although the period of the active copy is delayed by $dmax_{2i+2}$ during the execution. Note that, during the schedulability test, the active copy τ_{2i+2} may be considered as a replica of τ_{2i+1}, characterized by the *same* parameters $(C_{2i+2}, T_{2i+2}, \ldots)$, and thus with no phase delay (i.e. $R_{2i+2} = 0$). In contrast, τ_{2i+2} is delayed by $R_{2i+2} = dmax_{2i+2}$ time during the actual task execution. Such a delay is determined only after that W_{2i+2} has been computed by CTT.

Given any phasing delay $dmax_{2i+2} > 0$, the following results hold.

Remark 3.1 *If the k-th occurrence of τ_{2i+2} is invoked at time $dmax_{2i+2} + (k-1)T_{2i+2}$, then it will be completed at most by time*

$$dmax_{2i+2} + (k-1)T_{2i+2} + W_{2i+2}.$$

Remark 3.2 *Let τ_{2i+2} and τ_{2i+1} be the active and primary copies of the same task, and let W_{2i+2} be the worst-case response time of τ_{2i+2}. If τ_{2i+2} has a phasing delay*

$$dmax_{2i+2} = T_{2i+1} - W_{2i+2}$$

with respect to τ_{2i+1} and $P_f = P(\tau_{2i+1})$ fails during the k-th occurrence of τ_{2i+1}, then τ_{2i+2} terminates by kT_{2i+1}.

3.2 Worst-case response time of an active copy

In order to determine the maximum phasing delay, as explained in the previous subsection, a particular care has to be taken while computing the worst-case response time W_{2i+2} of an active copy τ_{2i+2}. Indeed, the worst-case response time must be equal to:

$$W_{2i+2} = \max\{W_{2i+2}(NF), W_{2i+2}(OF)\}$$

where $W_{2i+2}(NF)$ and $W_{2i+2}(OF)$ represent the worst-case response time in the absence of failures and in the case of one failure, respectively.

3.3 Determining the portion of active copy to be always executed

In this subsection, the formula for computing C'_{2i+2}, the portion of active copy to be always executed, is considered in details. We know that τ_{2i+1} terminates within W_{2i+1} units after the beginning of the period, while τ_{2i+2} cannot start before $dmax_{2i+2}$ units after the beginning of the period.

Since τ_{2i+2} is active, the quantity $W_{2i+1} - dmax_{2i+2}$ represents the worst case overlapping interval of the active copy that may be executed before the primary copy τ_{2i+1} terminates. In order to determine whether τ_{2i+2} has to be completely executed or not, a final test is needed:

- if $dmax_{2i+2} + C_{2i+2} \leq W_{2i+1}$, then an occurrence of τ_{2i+2} may exist that terminates before the completion of τ_{2i+1}, since the worst case arises when τ_{2i+2} starts at time $dmax_{2i+2}$ and is executed with no preemption for C_{2i+2} time units, and hence $C'_{2i+2} = C_{2i+2}$;
- if $dmax_{2i+2} + C_{2i+2} > W_{2i+1}$, then the execution of τ_{2i+2} must necessarily continue beyond W_{2i+1}, and the maximum portion of the active copy that has to be executed before W_{2i+1} is indeed $C'_{2i+2} = W_{2i+1} - dmax_{2i+2}$.

3.4 Assignment of tasks to processors

Let $P(\tau_h)$ denote the processor to which the (primary or backup) copy τ_h is assigned. The following notation is useful:

$$\begin{aligned}
&\textit{primary}(P_j) = \{\tau_{2i+1} : P(\tau_{2i+1}) = P_j\}\\
&\textit{backup}(P_j) = \{\tau_{2i+2} : P(\tau_{2i+2}) = P_j\}\\
&\textit{active}(P_j) = \{\tau_{2i+2} \in \text{backup}(P_j) : \tau_{2i+2} \text{ is active}\}\\
&\textit{passRecover}(P_j, P_f) =\\
&\quad \{\tau_{2i+2} \in \text{backup}(P_j) : P(\tau_{2i+1}) = P_f, \tau_{2i+2} \text{ is passive}\}\\
&\textit{actRecover}(P_j, P_f) =\\
&\quad \{\tau_{2i+2} \in \text{backup}(P_j) : P(\tau_{2i+1}) = P_f, \tau_{2i+2} \text{ is active}\}\\
&\textit{recover}(P_j, P_f) =\\
&\quad \textit{passRecover}(P_j, P_f) \cup \text{actRecover}(P_j, P_f)
\end{aligned}$$

The sets $\textit{primary}(P_j)$ and $\textit{backup}(P_j)$ represent the primary and backup copies assigned to processor P_j. The set $\textit{active}(P_j)$ includes the active backup copies assigned to processor P_j. The set $\textit{passRecovery}(P_j, P_f)$ consists of the passive copies assigned to P_j such that their primary copies are assigned to P_f, namely, this set contains all the passive backup copies that processor P_j must start scheduling when a failure of processor P_f is detected. The set $\textit{actRecovery}(P_j, P_f)$ denotes the active copies assigned to P_j with primary copies assigned to P_f, namely, this set contains all the active backup copies that processor P_j must keep executing when P_f fails. Finally, $\textit{recover}(P_j, P_f)$ gives the union between the last two sets.

The following definition has to be added to those above, to take into account the active copies whose computation can be performed only partially because their corresponding primary copies successfully complete. Let

$$\begin{aligned}
&\textit{shortActive}(P_j, P_f) =\\
&\quad \{\tau_{2i+2} \in \text{backup}(P_j) : P(\tau_{2i+1}) \neq P_f, \tau_{2i+2} \text{ is active}\}.
\end{aligned}$$

Remark 3.3 *shortActive*(P_j, P_f) *denotes the active copies whose corresponding primary copies are not assigned to* P_f *and that could be partially executed. Note that, by definition: (1)* *shortActive*$(P_j, P_f) \cap$ actRecover$(P_j, P_f) = \emptyset$ *and* *shortActive*$(P_j, P_f) \cup$ actRecover$(P_j, P_f) =$ active(P_j). *That is the active copies on* P_j *are partitioned in two sets:* *shortActive*(P_j, P_f) *containing the active copies that could be partially executed and actRecover*(P_j, P_f) *containing the active copies that must be completely executed; and (2) the particular set* *shortActive*(P_j, P_j) *indicates that all the active copies in active*(P_j) *could be partially executed* (*no active copy on* P_j *can have its primary copy on the same processor* P_j, *and thus* *shortActive*$(P_j, P_j) =$ active(P_j)).

The task copies are considered by decreasing RM priorities and assigned to processors following the *First-Fit* heuristic. The schedulability test is the CTT executed on task sets determined according to the following considerations.

To assign a task copy τ_h to a processor P_j, two feasibility tests, *NoFaultCTT* and *OneFaultCTT*, have to be executed on proper sets, which depend on the characteristic (i.e. primary, active, or passive) of the task copy τ_h and on the potential failed processor P_f. The test *OneFaultCTT* is exactly the same as described in [5] while *NoFaultCTT* is obtained from that of [5] by replacing *active*(P_j) with *shortActive*(P_j, P_j).

In the following, we expand on the implementation of *NoFaultCTT* to cope with the ARR strategy. In the absence of failure, to check whether τ_h (either primary or active) can be scheduled on P_j, determine the minimum W_h satisfying

$$W_h = C_h + \sum_{\tau_{2k+1} \in hp(h)} C_{2k+1} \left\lceil \frac{W_h}{T_{2k+1}} \right\rceil + \sum_{\tau_{2k+2} \in hp(h)} E_{2k+2} \left\lceil \frac{W_h}{T_{2k+2}} \right\rceil$$

where,

$$E_{2k+2} = \begin{cases} C_{2k+2} & \text{if } \tau_h \text{ is active and } \tau_{2k+2} \\ & \in \textit{actRecover}(P_j, P(\tau_{h-1})) \\ C'_{2k+2} & \text{if } (\tau_h \text{ is primary and } \tau_{2k+2} \\ & \in \textit{shortActive}(P_j, P_j)) \text{ or} \\ & (\tau_h \text{ is active and } \tau_{2k+2} \\ & \in \textit{shortActive}(P_j, P(\tau_{h-1}))) \end{cases}$$

and check whether

$$W_h \leq T_h.$$

Note that the two alternatives for E_{2k+2} follow from the definition of *NoFaultCTT*. In particular, to schedule an active copy τ_h together with other active copies τ_{2k+2} on processor P_j, the entire computation time C_{2k+2} must be considered when τ_{2k+2} belongs to the active copies whose primary copies are assigned to $P(\tau_{h-1})$ (τ_h is an active copy, hence $P(\tau_{h-1})$ denotes the processor to which the primary copy τ_{h-1} is assigned). In the other case, τ_{2k+2} can be partially executed and the computation time C'_{2k+2} is considered.

A high-level description of the algorithm for assigning tasks to processors, called ARR1, is given below. Its correctness follows from Remarks 3.1 and 3.2 and from the correctness of CTT.

ARR1

(0) Let the task copies $\tau_1, \tau_2, \ldots, \tau_{2n-1}, \tau_{2n}$ be indexed by increasing periods and set to 1 the number *m* of processors used.

(1) Repeat the following steps for $i = 0, \ldots, n-1$:

(1.1) Assign the primary copy τ_{2i+1} to the first processor P_j for which NoFaultCTT(τ_{2i+1}, P_j) and OneFaultCTT(τ_{2i+1}, P_j, P_f) for all $P_f \neq P_j$ are satisfied. If there is no such processor, then set *m* to $m+1$ and assign τ_{2i+1} to P_m. Compute W_{2i+1}.

(1.2) If $T_{2i+1} - W_{2i+1} < C_{2i+2}$, then set τ_{2i+2} to active, otherwise set τ_{2i+2} to passive and $dmax_{2i+2}$ to W_{2i+1}.

(1.3) If τ_{2i+2} is active, then assign it to the first processor P_j for which NoFaultCTT(τ_{2i+2}, P_j) and OneFaultCTT$(\tau_{2i+2}, P_j, P(\tau_{2i+1}))$ are satisfied. If there is no such processor, then set *m* to $m+1$ and assign τ_{2i+2} to P_m.

(1.3.1) Consider $W_{2i+2} = \max\{W_{2i+2}(NF), W_{2i+2}(OF)\}$, where $W_{2i+2}(NF)$ is the worst-case response time computed by NoFaultCTT, while $W_{2i+2}(OF)$ is that computed by OneFaultCTT. Set $dmax_{2i+2} = T_{2i+1} - W_{2i+2}$.

(1.3.2) Compute the partial computation time:

$$C'_{2i+2} = \begin{cases} W_{2i+1} - dmax_{2i+2} & \text{if } dmax_{2i+2} + \\ & C_{2i+2} > W_{2i+1} \\ C_{2i+2} & \text{otherwise} \end{cases}$$

(1.4) If τ_{2i+2} is passive, then assign it to the first processor P_j for which OneFaultCTT$(\tau_{2i+2}, P_j, P(\tau_{2i+1}))$ is satisfied. If there is no such processor, then set *m* to $m+1$ and assign τ_{2i+2} to P_m.

(2) Return the number *m* of processors used and the schedule so found.

3.5 Recovery from a failure

Once the task copies are assigned, each processor P_j executes, in the absence of failures, the task copies in *primary*$(P_j) \cup$ shortActive(P_j, P_j) by means of the RM algorithm. As soon as a primary copy τ_{2i+1} completes its execution, a "successful completion" message is sent to the processor where the backup copy is allocated. If such a copy is passive, then no action is taken, while if it is active, then its execution is immediately suspended. If a failure of processor P_f is detected at time θ, e.g., the successful completion message of τ_{2i+1} is not received by $P(\tau_{2i+2})$ by time

θ which corresponds to the completion time of τ_{2i+1}, then a *Recovery* procedure is invoked. Such a *Recovery* procedure can be obtained from that in [5] by replacing $active(P_j)$ with $shortActive(P_j, P_j)$. Moreover, such procedure can be extended to tolerate more than one processor failure with a technique related to that described in [5].

3.6 Reducing the number of CTT invocations

In the ARR1 assignment procedure seen above, each task copy is assigned to the first processor to which it fits, thus mixing together primary, active, and passive copies on the same processor. In this way, however, many CTT's are required to assign each task, thus increasing the computation time of the schedulability test.

In order to reduce the number of CTT invocations, the active resource reclaiming technique can be combined with a partitioning of the processors into two or three groups so as to avoid to mix together primary, active, and passive task copies on the same processor.

The simplest version to be implemented is clearly that in which three groups of processors are used, denoted ARR3, where there is a first group of processors for the primary copies, a second group for the active copies, and a third one for the passive copies. However, two groups can also be used (version ARR2), the first group for both primary and active copies (which indeed are handled in a very similar way) and the second one for the passive copies. The ARR2 and ARR3 algorithms can be easily derived from the ARR1 algorithm explained above, and thus are not described here in details.

The advantage of ARR3 is that of requiring less CTT invocations than ARR1 and ARR2. Indeed, using ARR3, at most m CTT's are needed to assign a primary copy, for a total of $O(nm)$ CTT's, where n is the number of primary copies and m the number of processors employed. In contrast, when using ARR1 and ARR2 for assigning a primary copy to a processor P_j, besides to check for schedulability in the absence of failures, one needs to check for schedulability also in the case of a failure to any processor other than P_j, for a total of $O(nm^2)$ CTT's.

4 The S-PRIORITY algorithm

This section discusses a variant of the ARR strategy where tasks are assigned to processors without following the Rate-Monotonic priority. In particular, we consider another assignment priority, called here S-PRIORITY, which was introduced in [6] and has the effect of grouping together on the same processor those tasks whose periods are equal or multiple, thus producing a more compact schedule in many cases. It is worth noting that, once the tasks are assigned to processors, they are scheduled on each single processor by means of the usual RM algorithm.

In order to assign to the same processor those tasks whose periods are equal or multiple, the priority ordering given in [6] can be used. Consider k tasks $\{\tau_1, \ldots, \tau_k\}$ indexed by RM priorities, that is, with $T_1 \leq \ldots \leq T_k$. For each task τ_i compute

$$S_i = \log_2 T_i - \lfloor \log_2 T_i \rfloor \qquad i = 1, \ldots, k$$

and consider the permutation $(1, 2, \ldots, k) \rightarrow (j_1, j_2, \ldots, j_k)$ such that $\{\tau_{j_1}, \ldots, \tau_{j_k}\}$ are ordered by S priorities, namely $S_{j_1} \leq \ldots \leq S_{j_k}$. Then assign tasks to processors following such an ordering. Observe that, in this case, when assigning τ_{j_i} to any processor P_h, tasks with RM priority smaller than that of τ_{j_i} can be already assigned to P_h. Therefore, besides to test for schedulability of τ_{j_i} together with the tasks in $hp(j_i)$, a CTT must be performed again for each τ_{j_l}, already assigned to P_h, with $T_{j_l} > T_{j_i}$.

In this way, however, the worst-case response times of the primary copies are known only when *all* the primary copies have been assigned to the processors. Therefore, to assign a backup copy, its status (active/passive) can be determined only after that the assignment of all the primary copies is completed. In order to maintain unchanged the worst-case response times of the primary copies, the backup copies are assigned to a second group of processors, following the S priority order. Thus the primary copies $\tau_1, \tau_3, \ldots, \tau_{2n-1}$ are first assigned to a group of processors, and then the backup copies $\tau_2, \tau_4, \ldots, \tau_{2n}$ are successively assigned to the other group of processors. Remember, however, that the S priority is used only for assigning tasks to processors, since all the tasks assigned to the same processor are then scheduled by the usual RM priority.

5 The PASSIVE and S-PR-PASS algorithms

In an ideal situation, only passive duplication should be used, since in this case redundant computations are performed only when needed after a failure. In contrast, active copies require at least a partial computation even in the absence of failures. This section presents two algorithms that employ only passive duplication: whenever $T_{2i+1} - W_{2i+1} < C_{2i+2}$ a new processor is used to schedule τ_{2i+1}, hence only passive backup copies can be employed to tolerate failures. Clearly, this is not possible in all cases, but only when

$$C_{2i+1} + C_{2i+2} \leq T_{2i+1} \qquad i = 0, \ldots, n-1,$$

which, under our assumption $C_{2i+1} = C_{2i+2}$, becomes $U_{2i+1} = \frac{C_{2i+1}}{T_{2i+1}} \leq \frac{1}{2}$ for $i = 0, \ldots, n-1$, since in this case the period of each task is long enough to permit the execution of both a primary copy and its passive in the case of a failure.

The first algorithm, called PASSIVE, uses the RM priority both to assign tasks to processors and to schedule the tasks assigned to each processor. The PASSIVE algorithm can be described as follows.

PASSIVE

(0) Let the task copies $\tau_1, \tau_2, \ldots, \tau_{2n-1}, \tau_{2n}$ be indexed by increasing periods, set to 1 the number m of processors used, and set to passive all the backup copies $\tau_2, \tau_4, \ldots, \tau_{2n-2}, \tau_{2n}$

(1) Repeat the following steps for $i = 0, \ldots, n-1$:

(1.1) Assign the primary copy τ_{2i+1} to the first processor P_j for which NoFaultCTT(τ_{2i+1}, P_j), OneFaultCTT(τ_{2i+1}, P_j, P_f) for each $P_f \neq P_j$, and the condition $(T_{2i+1} - W_{2i+1} \geq C_{2i+2})$ are all satisfied. If there is no such processor, then set m to $m+1$ and assign τ_{2i+1} to P_m. Compute W_{2i+1}, and set $dmax_{2i+2}$ to W_{2i+1}.

(1.2) Assign τ_{2i+2} to the first processor P_j for which OneFaultCTT($\tau_{2i+2}, P_j, P(\tau_{2i+1})$) is satisfied. If there is no such processor, then set m to $m+1$ and assign τ_{2i+2} to P_m.

(2) Return the number m of processors used and the schedule so found.

The S-PR-PASS algorithm combines the PASSIVE and S-PRIORITY algorithms previously presented. As in the S-PRIORITY algorithm, all the primary copies are first assigned to a group of processors following the S priorities. The schedulability test of any primary copy τ_{2i+1} on any processor P_j consists in the following steps, where all the CTT's are executed on task sets containing only primary copies:

- a CTT on $\{\tau_{2i+1}\} \cup hp(2i+1)$;
- a CTT on $\{\tau_{2k+1}\} \cup \{\tau_{2i+1}\} \cup hp(2k+1)$ for each τ_{2k+1} already assigned to P_j such that $T_{2k+1} > T_{2i+1}$;
- a check to verify whether $T_{2i+1} - W_{2i+1} \geq C_{2i+2}$, in order to guarantee that there is enough time to schedule the passive copy within the same period (otherwise, τ_{2i+1} is assigned to a new processor).

During the assignment of the primary copies, no *OneFaultCTT* is needed, since primary and passive copies are assigned to two different groups of processors. Once all the primary copies are assigned, the S-PR-PASS algorithm assigns all the passive copies following again the S priorities. As in the PASSIVE algorithm, S-PR-PASS uses the RM priority to schedule the tasks assigned to each processor.

6 Simulation results

In the previous sections, the RMFF algorithm has been extended leading to six fault-tolerant algorithms: ARR1, ARR2, ARR3, PASSIVE, S-PRIORITY, and S-PR-PASS. In this section, simulation experiments are reported in order to evaluate the performance of the different algorithms.

As in [5, 6], large task sets with at most $n = 600$ tasks are generated. The parameters of each task τ_i are chosen as follows. The period T_i is an integer uniformly distributed in $[1, T_{max}]$, while the computation time C_i is an integer uniformly distributed in $[1, \alpha T_i]$, where $\alpha = \max_i \frac{C_i}{T_i}$ is an upper bound for the task load. T_{max} is fixed to 500, and each backup copy has the same period and computation time as its primary copy. Three values for α are chosen, namely, 0.2, 0.5, and 0.8. For the chosen n and α, the experiment is repeated 30 times. The performance metric in all the experiments is the number N of processors used by an algorithm to schedule both primary and backup copies. Another useful metric should be the ratio $\frac{N}{N_0}$, where N_0 is the minimum number of processors to schedule only the primary copies, since $\frac{N}{N_0} - 1$ gives the ratio of additional processors introduced to tolerate a processor failure. Since an optimal task assignment is hard to be found for large task sets, N_0 is replaced by its lower bound U, which is obtained by summing up all the loads of the primary copies.

For the chosen n and α, the $j-th$ experiment gives N_j, U_j, and $\frac{N_j}{U_j}$. Since the experiment is repeated 30 times, the average results are computed as follows: $N = (\sum_{j=1}^{30} N_j)/30$, $U = (\sum_{j=1}^{30} U_j)/30$, and $\frac{N}{U} = (\sum_{j=1}^{30} \frac{N_j}{U_j})/30$. All the algorithms were written in C and run on a Digital Alpha-Server 2100, Model 5/250. The outcome of the experiments is given in Figure 2 (for the sake of clarity, the results for ARR1 and ARR3, which are similar to those of ARR2, will be reported only in Figure 3 but not in Figure 2).

Figure 2 shows the ratio $\frac{N}{U}$ for the experiments, where U is the lower bound for scheduling only the primary copies, and N the number of processors required by the various algorithm, including also the FTRMFF algorithm presented in [5].

By observing Figure 2, one notes that, when $\alpha = 0.2$, PASSIVE and ARR2 behave as FTRMFF since in this case all the tasks have a low load and thus both FTRMFF and ARR2 employ almost exclusively the passive duplication. When $\alpha = 0.2$, the best performance is given by the S-PRIORITY and S-PR-PASS algorithms. When $\alpha = 0.5$ and $\alpha = 0.8$, FTRMFF is the worst algorithm. In the $\alpha = 0.5$ case, S-PR-PASS and ARR2 have the best performance, while in the $\alpha = 0.8$ case the best algorithm is ARR2. Indeed, in this latter case, almost all tasks have a high load and thus the active duplication is almost exclusively employed. Therefore, ARR2 gains benefits from the active resource reclaiming strategy where active copies are executed only partially. Observe also that for $\alpha = 0.2$ and $\alpha = 0.5$ the S-PR-PASS algorithm can tolerate a failure by using a number of processors close to the minimum number of processors required in the non fault-tolerant case, since the corresponding ratio $\frac{N}{U}$

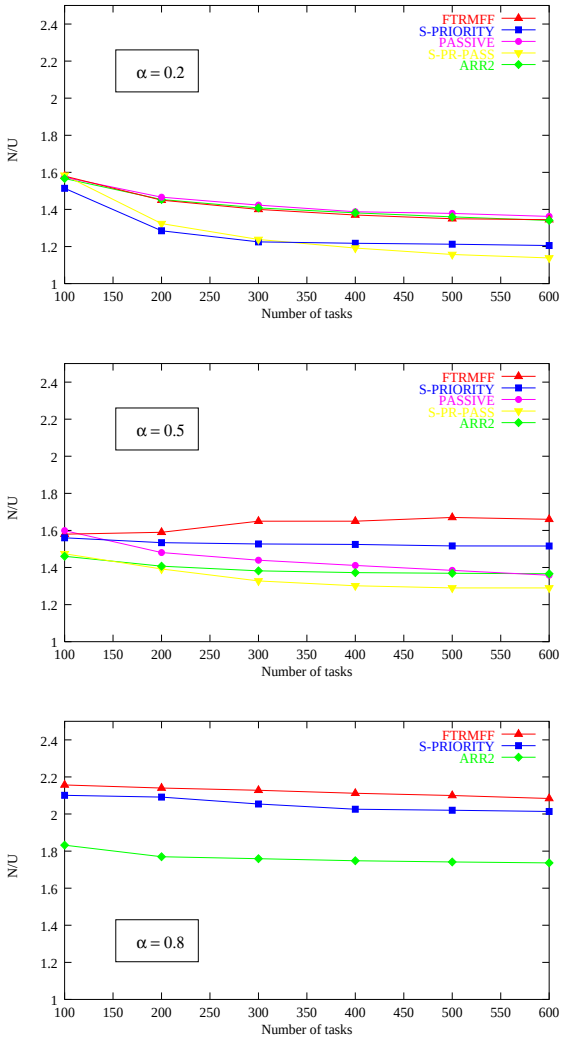

Figure 2. Ratio N/U of the number N of processors for scheduling both primary and backup copies and the lower bound U for scheduling only the primary copies. $N/U-1$ gives the ratio of additional processors introduced to tolerate a processor failure.

	α	% gain	Running time (seconds)		
		n= 400	n= 100	n= 300	n= 600
PASSIVE	0.2	0%	8.1	181.8	1569
	0.5	15%	15.0	240.5	1693
	0.8	–		–	
S-PRIORITY	0.2	12%	1.0	4.4	12.1
	0.5	8.5%	0.9	4.8	19.3
	0.8	3%	1.0	8.4	39.4
S-PR-PASS	0.2	13%	1.2	6.0	19.3
	0.5	21%	1.3	9.3	36.6
	0.8	–		–	
ARR1	0.2	0%	18.7	321.2	2792
	0.5	16%	13.4	105.9	424
	0.8	17%	17.8	158.9	715.3
ARR2	0.2	12%	3.5	33.6	188.1
	0.5	15%	8.4	87.2	400
	0.8	15%	16.3	160	698
ARR3	0.2	8%	1.4	8.4	28.6
	0.5	10%	1.7	12.3	56.4
	0.8	14%	2.5	21.6	114.1

Figure 3. Performance and running times of the proposed algorithms. The second column reports the percentage of gain introduced by the proposed algorithms with respect to the FTRMFF algorithm in [5].

is close to 1.

Oh and Son [21] report the performance of their best RM scheduling algorithm to achieve fault-tolerance considering multiple backup copies. They show that on the average 40% extra processors are required compared to the lower bound L which is equal to the sum of all the loads of the primary copies and backup copies. If each task has one backup copy with the same computation time as its primary then the lower bound L becomes twice the lower bound U considered here, and thus 180% extra processors are required by [21] with respect to U. In contrast, Figure 2 shows that the extra processors needed by our best algorithms range from 20% to 80% with respect to the lower bound U.

The performance of all the new algorithms introduced in this paper (including also ARR1 and ARR3) are summarized in Figure 3. In this figure, the second column reports the percentage gained by the proposed algorithms with respect to FTRMFF (for instance, the S-PR-PASS algorithm with $\alpha = 0.5$ employs 21% less processors than FTRMFF). These values are computed for $n = 400$ only, since for $n \geq 400$ the gain remains almost the same. Moreover, Figure 3 reports also the average computation time of the six assignment algorithms, for $n = 100$, $n = 300$, and $n = 600$. The average running times of FTRMFF are comparable to those of ARR1 and are not reported explicitly. A circle in the figure outlines the entries with the highest processor gains or the lowest running time.

By observing Figure 3, it is possible to choose the best algorithm depending on the characteristics of the task set, taking into account not only the number of processors used in the schedule, but also the running time of the assignment algorithm. Note that the running times in this figure are

those required by the assignment algorithm, which is performed off-line only once. The algorithms for the actual task scheduling and for recovering from a failure are performed on-line and are much faster. Indeed, all the schedulability tests and all the task sets to be scheduled on the processors in the case of a failure were previously computed off-line by the assignment algorithm.

One can note in Figure 3 that ARR2 has a good processor gain for all values of α, and thus ARR2 can handle task sets with different characteristics. The good number of processors found, however, may require a high running time. Disregarding the running time, the ARR1 algorithm is that using the lowest number of processors when $\alpha = 0.8$. Instead, S-PRIORITY has the smallest running time, but this is paid in terms of a higher number of processors. The trade-off processors/time suggests to choose the assignment algorithm as follows:

- S-PR-PASS, fast and effective for $\alpha = 0.2$ and $\alpha = 0.5$;
- ARR3, with low number of processors and running time for $\alpha = 0.8$.

7 Conclusions

In this paper, several fault-tolerant extensions to the RMFF algorithm have been presented that improve the performance, reducing both the number of processors needed for the schedule and the running time of the assignment algorithm, with respect to previously presented algorithms.

Several questions still remain to be explored. For example, a schedulability condition could be used which is only sufficient, such as those proposed in [20] and [6], but easier and faster to verify than the necessary and sufficient CTT. In addition, to reduce the number of processors needed, one could employ either a new task ordering different from both the RM and the S priority for the task assignment, or new heuristics different from the First-Fit heuristics.

However, it does not seem straightforward to find new heuristics which can lead to better performance. For instance, we tried to assign the primary copies to the less loaded processors, since one expects that such a criterium would simplify the scheduling of the passive copies, but the outcome of the resulting experiments was worse than ARR1.

References

[1] N. Audsley, K. Tindell, A. Burns. The end of line for static cyclic scheduling? *Proc. Euromicro Workshop on Real-Time Systems*, 36-41, June 1993.

[2] A.A. Bertossi, A. Fusiello. Rate-monotonic scheduling for hard-real-time systems. *European Journal of Operational Research* 96, 429-443, 1997.

[3] A.A. Bertossi, A. Fusiello, L.V. Mancini. Fault-tolerant deadline-monotonic algorithm for scheduling hard-real-time tasks. *Proc. 11th IEEE International Parallel Processing Symposium*, 133-138, Geneva, Switzerland, April 1997.

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

[5] A.A. Bertossi, L.V. Mancini, F. Rossini. Fault-tolerant rate-monotonic first-fit scheduling in hard-real-time systems. *IEEE Transactions on Parallel and Distributed Systems* 10, 934-945, September 1999.

[6] A. Burchard, J. Liebherr, Y. Oh, S.H. Son. New strategies for assigning real-time tasks to multiprocessor systems. *IEEE Transactions on Computers* 44, 1429-1442, 1995.

[7] A. Burns, R. Davis, S. Punnekkat. Feasibility analysis of fault-tolerant real-time task sets. *Proc. Euromicro Workshop on Real-Time Systems* 29-33, June 1996.

[8] G. Buttazzo. Hard Real-Time Computing Systems. *Real time systems series* Vol 23, 2nd Edition, ISBN 0-387-23137-4, Springer Verlag, 2005.

[9] M. Caccamo, G. Buttazzo. Optimal scheduling for fault-tolerant and firm real-time systems. *Proc. IEEE Conference on Real-Time Computing Systems and Applications*, Hiroshima, Japan, Oct. 1998.

[10] H. Chetto, M Chetto An adaptive scheduling algorithm for fault-tolerant real-time systems. *Software Engineering Journal*, 93-100, 1991.

[11] S. Dhall, C.L. Liu. On a real-time scheduling problem. *Operations Research* 26, 127-141, 1978.

[12] S. Ghosh, R. Melhem, D. Mosse. Fault-tolerance through scheduling of aperiodic tasks in hard-real-time systems. *IEEE Transactions on Parallel and Distributed Systems* 8, 272-284, 1997.

[13] S. Ghosh, R. Melhem, D. Mosse, J.S. Sarma. Fault-tolerant rate-monotonic scheduling. *Real Time Systems* 15, 149-181, 1998.

[14] M. Joseph, P. Pandya. Finding response times in a real-time system. *The Computer Journal* 29, 390-395, 1986.

[15] M.H. Klein, J.P. Lehoczky, R. Rajkumar. Rate-monotonic analysis for real-time industrial computing. *IEEE Computer*, 24-33, Jan. 1994.

[16] C.M. Krishna, K.G. Shin. On scheduling tasks with a quick recovery from failure. *IEEE Transactions on Computers* 35, 448-454, May 1986.

[17] S. Lauzac, R. Melhem, D. Mosse. An Improved Rate-monotonic Admission Control and its Applications *IEEE Transactions on Computers* 52(3), March 2003.

[18] J.Y. Leung, M.L. Merrill, A note on preemptive scheduling periodic real-time tasks. *Information Processing Letters* 11, 115-118, 1980.

[19] F. Liberato, R. Melhem, D. Mosse, Tolerance to Multiple Transient Faults for Aperiodic Tasks in Hard Real-time Systems. *IEEE Transactions on Computers* 49(9), Sep 2000.

[20] C.L. Liu, J.W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM* 20, 46-61, 1973.

[21] Y. Oh, S.H. Son. Enhancing fault-tolerance in rate-monotonic scheduling. *Real-Time Systems* 7, 315-329, 1994.

[22] K. Tindell. Adding time-offsets to schedulability analysis. *Technical Report YCS-221*, Dept. of Computer Science, University of York, 1994.

[23] K. Tindell, A. Burns, A.J. Wellings. An extendible approach for analyzing fixed-priority hard-real-time tasks. *Real-Time Systems* 6, 133-151, 1994.