European Journal of Operational Research 96 (1997) 429–443

EUROPEAN JOURNAL OF OPERATIONAL RESEARCH

Invited Review

Rate-Monotonic scheduling for hard-real-time systems [1]

Alan A. Bertossi *, Andrea Fusiello

Dipartimento di Matematica, Università di Trento, Via Sommarive 14, 38050 Povo (Trento), Italy

Received 1 June 1996; accepted 1 July 1996

Abstract

Hard-real-time computing systems are widely used in our society, for example, in nuclear and industrial plants, telecommunications, avionics and robotics. In such systems, almost all tasks occur infinitely often and have time deadlines, namely, their correctness relies not only on their logical results, but also on the time at which the results are available. A scheduling algorithm specifies an order in which all the tasks are to be executed, in a way that all the time deadlines are met. This paper provides a review on deterministic scheduling algorithms for hard-real-time systems, focusing mainly on fixed priority, preemptive scheduling of periodic tasks on a single processor and, in particular, on the Rate-Monotonic algorithm. After presenting some basic results, several generalisations, aimed at relaxing some constraints and facing more realistic cases, are described. Issues covered include uniprocessor and multiprocessor systems, periodic and non-periodic tasks, restricted and arbitrary deadlines, fixed and dynamic priorities, independent and synchronised tasks, as well as fault-free and fault-tolerant systems.

Keywords: Scheduling theory; Hard-real-time computing; Multiprocessor systems; Packing; Fault-tolerance

1. Introduction and terminology

Real-time computing systems are widely used for monitoring and control functions in many industrial applications. Examples of such systems include the control of engines, traffic, nuclear power plants, time-critical packet communications, aircraft avionics and robotics. In this context, the term *task* is used to denote either a computer process or a single thread of control that has to be executed. In real-time systems, tasks usually have timing requirements that must be verified. Thus, the correctness of a task computation depends not only on its logical result, but also on when the result is available.

A common misconception about real-time computing is to think that it is equivalent to fast computing. Of course, minimising the task response times is helpful in satisfying the time requirements, but the most important principle of real-time systems is *predictability*, namely, the ability to determine whether the system is capable to meet all the time requirements of the tasks. In particular, since the tasks compete for the usage of shared resources, careful resource-management techniques must be used in order to prevent long waits that can lead to the violation of some time requirements. The measures of merit used to evaluate real-time systems may significantly differ from those typically used for other systems. In particular, such measures include:

* Corresponding author. e-mail: bertossi@science.unitn.it

[1] This work was supported by the Progetto Speciale 1995, Dipartimento di Matematica, Università di Trento, Italy.

PII S0377-2217(96)00220-2

schedulability – the degree of system loading below which the task timing requirements can be ensured;
responsiveness – the latency of the system in responding to external events; and
stability – the capability of the system to guarantee the time requirements of the most critical tasks in the cases where it is impossible to guarantee the time requirements of all the tasks (a typical case is the so called *transient overload* condition.)

Since tasks are used to perform control and monitoring functions, they often recur infinitely many times. There are three kinds of real-time tasks, depending upon their arrival pattern:
periodic – the task has a regular interarrival time, called *period*;
sporadic – there is a lower bound on the task interarrival time; and
irregular – the task can arrive at any time.

The time requirements of real-time tasks are called *deadlines*:

- if meeting a task deadline is critical for the system functionality, then the deadline is said to be *hard*; missing a hard deadline is considered a system failure, and can lead to catastrophic consequences;
- if it is desirable to meet a task deadline, but occasionally missing it can also be tolerated, then the deadline is said to be *soft*; a task with a soft deadline is expected to be completed as early as possible, in order to have a good responsiveness.

This paper will mainly deal with *hard-real-time systems* – the most common kind of real-time systems – where almost all the tasks are periodic and have hard deadlines.

There are two ways to guarantee that all the task deadlines are met in a hard-real-time system. One is to use semantic models to describe the system and prove its correctness, and the other is to use the scheduling theory to give an order in which the tasks have to be executed.

Although the theory of scheduling is born in the operations research milieu, scheduling of hard-real-time systems seldom appears in the operations research literature. The reason of this can be two-fold. First, queuing models and stochastic assumptions may be useless since they cannot always guarantee that hard deadlines are met. Second, deterministic scheduling deals mostly with tasks that have to be scheduled only once instead of infinitely many times.

The purpose of this paper is to provide a review on deterministic scheduling algorithms for hard-real-time systems which guarantee that all the periodic occurrences of the tasks will meet their dead-

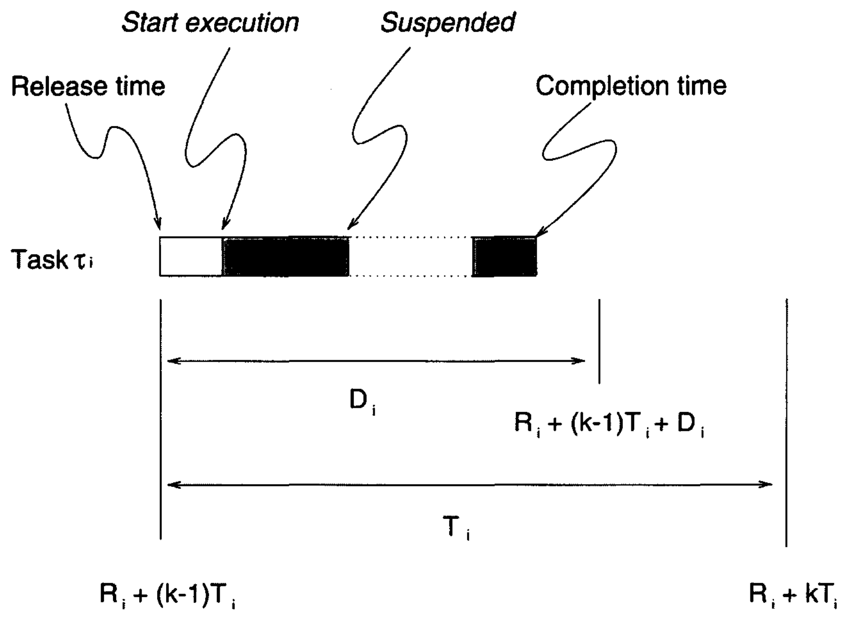

Fig. 1. Periodic task timing.

lines. The rest of the paper is structured as follows. The remaining part of this section is devoted to introduce some additional notions, like preemptions and priority-driven scheduling. Section 2 presents the basic Rate-Monotonic analysis for preemptively scheduling periodic independent tasks on a single processor. The analysis is extended in Section 3 in order to deal with task synchronisation and non-periodic tasks. Section 4 considers multiprocessor systems. In particular, partitioning algorithms for assigning tasks to processors are presented. Section 5 considers the fault-tolerance issue for multiprocessor systems, where hard deadlines of the tasks have to be met even in the presence of a processor failure. Final considerations terminate the paper in Section 6.

1.1. *Periodic tasks*

Periodic tasks are the most common in real-time systems. They are usually invoked in order to monitor a physical system with regularity.

A *periodic* task τ_i is characterised by the quadruple (C_i, T_i, D_i, R_i), where:
C_i is the (worst case) computation time of task τ_i.
T_i is the invocation (or arrival) period of task τ_i.
D_i is the deadline of task τ_i.
R_i is first invocation (or arrival) time of task τ_i.
A periodic task leads to an infinite sequence of task instances called *jobs*.

The k-th job of task τ_i is ready for execution at time $R_i + (k-1)T_i$ and, in order to meet its deadline, its execution – that requires C_i time units – must be completed no later than time $R_i + (k-1)T_i + D_i$.

The *release time* of a job is the time when it is ready to be executed (usually, this coincides with the beginning of a new period), its *completion time* is the time when its execution is completed, while the *response time* of the job is the difference between its completion time and its release time. An example to explain this terminology is provided in Fig. 1.

1.2. *Preemptions*

One dichotomy of scheduling algorithms is based on whether a running job can be suspended or not. *Preemptive* algorithms assume that any job can be suspended at any time, and can be resumed later from the point of suspension. The preemption of the job does not effect the behaviour of the job and the overhead due to the preemption can be considered small. Clearly, non-preemptive scheduling saves the overhead due to a context switch, but it is less powerful. Indeed, there are task sets which can be scheduled using preemptions but cannot be scheduled without preemptions. Moreover, there is no exact analysis for non-preemptive scheduling, while there is an exact analysis, which is reported in this paper, for preemptive scheduling.

1.3. *Priority-driven scheduling*

The algorithms used in practice for scheduling in hard-real-time systems are *priority-driven* preemptive algorithms. These algorithms assign priorities to jobs according to some policy. After priorities are assigned, dispatching proceeds as follows. At each instant of time, the processor is assigned to the highest priority job which is ready to run preempting – if necessary – a lower priority job.

For a given task set $\{\tau_1, \ldots, \tau_n\}$, a priority assignment is *feasible* if all the deadlines of all the jobs are met using such an assignment. In this case, the task set is said to be *schedulable* by the algorithm that produced the feasible priority assignment. A scheduling algorithm is *optimal* if *all* the task sets for which a feasible priority assignment exists are schedulable by that algorithm. Since there is nothing to optimise, this notion of optimality has a slight strange flavour. However, do not mistake an optimal scheduling algorithm for an optimisation scheduling algorithm, the latter being an algorithm that finds a feasible schedule having the best value of an objective function, e.g. see [18].

A priority-driven algorithm is characterised by the kind of its priority assignment. A scheduling algorithm is *static* if the priority of a task is fixed and cannot change in the time (i.e., all the jobs of the same task always have the same priority). A scheduling algorithm is *dynamic* if the priority of a task might change from invocation to invocation (i.e., different jobs of the same task may have different priorities.)

For example, a static scheduling algorithm is the *Rate-Monotonic* algorithm, where the task with shortest period has the highest priority. In contrast, a dynamic scheduling algorithm is the *Earliest Deadline First* algorithm, in which the ready job with the nearest deadline has the highest priority. Liu and Layland [28] proved that the Earliest Deadline First algorithm is an optimal priority-driven scheduling algorithm, and gave a necessary and sufficient condition for testing the schedulability of the tasks:

Theorem 1.1. *Given a set* $\{\tau_1, \ldots, \tau_n\}$ *of periodic tasks, with* $D_i = T_i$ *for all* i, *the Earliest Deadline First algorithm yields a feasible priority assignment iff*

$$\sum_{i=1}^{n} C_i/T_i \leqslant 1.$$

For a discussion about properties of the Earliest Deadline First algorithm see [28] and [12]. In the present review, only static algorithms are considered. In particular, the Rate-Monotonic algorithm will be treated in detail in the following sections. Although the Rate-Monotonic algorithm is not optimal (in fact, it is optimal among all *static* scheduling algorithms only) its predictability and stability under transient overload are so appreciated that it is becoming an industry standard.

1.4. Time complexity

Clearly, since tasks recur indefinitely in a hard-real-time system, their execution requires infinite time. However, deciding whether a feasible schedule exists usually requires *less* time! A *feasibility test* is an algorithm for checking conditions which are necessary and/or sufficient for a task set to be schedulable on a processor. In general, there are no polynomial-time testable conditions which are both necessary and sufficient for arbitrary task sets. Specifically, the problem of deciding whether an *arbitrary* task system can be scheduled on one processor was proven to be NP-complete [25]. The only known test for the general case consists in simulating the schedule over an interval equal to the least common multiple of the task periods, and such a test can run in exponential time. Anyway, for some particular cases, more efficient tests are known (for example Theorem 1.1 gives a polynomial time test when $D_i = T_i$, $\forall i$, while [5] gives a polynomial time test for non priority-driven scheduling). Leung and Whitehead [26] showed a pseudo-polynomial time test for fixed priority scheduling. Later, the so called *Completion Time Test* [20,3] was introduced. This test works for fixed priority schedules and requires pseudo-polynomial time, but behaves as it were polynomial for many practical cases. The Completion Time Test is described in more detail in the next section of the present paper. Other complexity results for special cases can be found in [4].

2. Basic Rate-Monotonic analysis

This section focuses on some early basic results concerning fixed priority, preemptive algorithms for scheduling a task set $\{\tau_1, \ldots, \tau_n\}$. The following assumptions are made:

- all tasks are periodic;
- $C_i \leqslant D_i \leqslant T_i$ for all i;
- tasks are independent, i.e., no inter-task communication or synchronisation is permitted;
- there is a single processor.

In practice, realistic models require some of these assumptions to be weakened, for example, by allowing both periodic and non-periodic tasks, synchronisations of tasks on shared resources, or more than one processor, as will be considered in the following sections.

From now on, for the sake of simplicity, tasks are assumed to be indexed so that τ_i has a higher priority than τ_j whenever $i < j$.

2.1. Basic results

Liu and Layland [28] proved the following important results, assuming periodic independent preemptable tasks, with $D_i = T_i$ for all i, and fixed priorities.

Theorem 2.1. *The longest response time for* any job *of a task* τ_i *occurs when it is invocated simultaneously with all higher priority tasks* (*i.e. when* $R_1 = R_2 = \cdots = R_i$).

The time when all the tasks are invoked simultaneously is called a *critical instant*. The important result regarding the feasibility of a fixed priority

Table 1
A task set with deadlines equal to periods

Task	T_i	D_i	C_i
τ_1	100	100	40
τ_2	150	150	40
τ_3	350	350	100

assignment is that only the first deadline of each task needs to be checked for feasibility.

Theorem 2.2. *A fixed priority assignment is feasible provided the deadline of the* first *job of each task starting from a critical instant is met.*

Due to the above results, all the first arrival times of the tasks are assumed to be 0 hereafter, i.e. $R_1 = R_2 = \cdots = R_n = 0$, since this assumption takes care of the worst possible case. In this way, only the first deadline of each task must be met, when the task is scheduled together with all higher priority tasks, in order for a fixed priority assignment to be feasible.

2.2. Rate-Monotonic scheduling

Liu and Layland [28] proposed a fixed-priority scheduling algorithm, called *Rate-Monotonic*, assuming that each task deadline coincides with the end of the period, that is when $D_i = T_i$ for all i. In their algorithm, priorities are assigned inversely to task periods – hence τ_i receives a higher priority than τ_j if $T_i < T_j$. They also proved that the Rate-Monotonic algorithm is optimal among all static scheduling algorithms (assuming $D_i = T_i$ for all i). As an example, consider the task set of Table 1. The schedule of such a task set, obtained with the Rate-Monotonic algorithm, is provided in Fig. 2.

2.2.1. Utilisation bound

Based on the notion of a critical instant, Liu and Layland derived a sufficient (but not necessary) schedulability test for the Rate-Monotonic algorithm:

Theorem 2.3. *Given a periodic task set $\{\tau_1, \ldots, \tau_n\}$, with $D_i = T_i$ for all i, the Rate-Monotonic algorithm yields a feasible priority assignment if*

$$\sum_{i=1}^{n} C_i/T_i < n(2^{1/n} - 1).$$

In the theorem above, the ratio C_i/T_i represents the *utilisation factor* of task τ_i, while the sum over all i represents the total utilisation of the task set (and hence of the processor). In practice, Theorem 2.3 states that there is a bound U on the total utilisation of the task set, below which the Rate-Monotonic policy always yields a feasible priority assignment. The bound U depends only on the number of tasks and for large task sets it is about 0.693, since

$$\lim_{n \to \infty} n(2^{1/n} - 1) = \ln 2 = 0.693.$$

A counterexample showing that the ln 2 bound is only sufficient is given by the task set of Table 1, where the processor utilisation is $\frac{40}{100} + \frac{40}{150} + \frac{100}{350} = 0.95$. A bound that in some cases can be higher than ln 2 is provided in [9].

2.2.2. Completion Time Test

Joseph and Pandya [20] derived an exact analysis to find the worst-case response time for a given task, assuming fixed priority, independent tasks and dead-

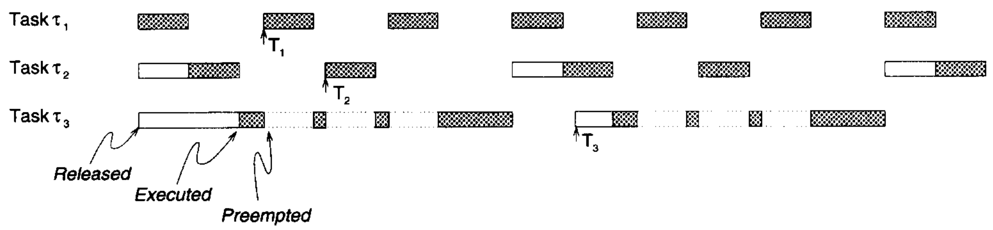

Fig. 2. Example of scheduling the task set of Table 1 according to the Rate-Monotonic algorithm.

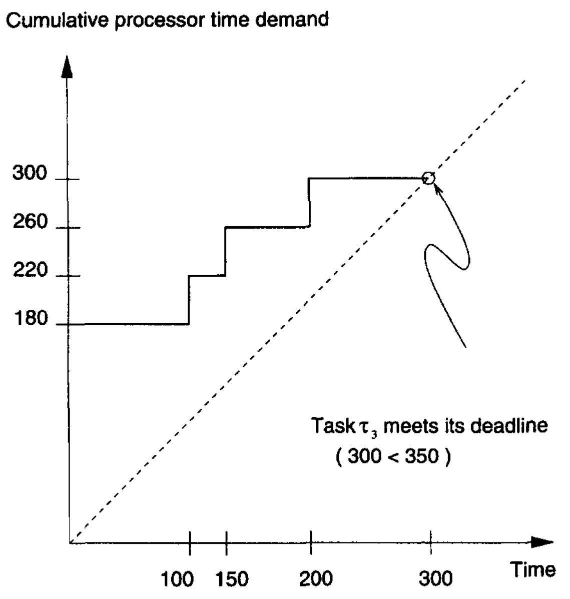

Fig. 3. Cumulative processor time demand versus time for the task set of Table 1. At time 50, the demand is $40\lceil\frac{50}{100}\rceil+40\lceil\frac{50}{150}\rceil+100\lceil\frac{50}{350}\rceil=180$. The worst case completion time W_3 of task τ_3 is computed as follows: $W_3(1)=100$, $W_3(2)=180$, $W_3(3)=260$, $W_3(4)=W_3(5)=300$. Since $W_3=300<350=D_3$, τ_3 meets its deadline.

lines less than periods (i.e., $D_i \leqslant T_i$ for all i). The analysis considers a critical instant and makes use of Theorem 2.2.

The following equation gives the worst case response time W_i of a task τ_i:

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

where hp(i) is the set of all tasks with higher priority than τ_i. The right side of Eq. (1) represents the *cumulative processor time demand* for tasks in hp(i) $\cup \{\tau_i\}$. Indeed, $\lceil t/T_j \rceil$ is the overall number of jobs of τ_j by time t and therefore $C_j\lceil t/T_j\rceil$ represents the processor time demand of task τ_j by time t. Thus, τ_i will complete its execution at time W_i iff the cumulative processor time demand of tasks in hp(i) $\cup \{\tau_i\}$ up to time W_i is exactly equal to W_i, i.e. when Eq. (1) holds. As depicted in Fig. 3, the worst case response time is the smallest W_i that satisfies the equation. This can be easily computed by iteration. Starting with $W_i(0)=0$, $W_i(k)$ is computed for $k = 1, 2, \ldots$ as follows:

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

stopping the iteration as soon as $W_i(k+1) = W_i(k)$. The convergence is guaranteed iff [41]

$$\sum_i C_i/T_i < 1.$$

A necessary and sufficient schedulability test can be readily derived from Eq. (1):

Theorem 2.4. *A fixed priority assignment for a task set* $\{\tau_1, \ldots, \tau_n\}$, *such that* $D_i \leqslant T_i$ *for each* i, *is feasible iff* $W_i \leqslant D_i$, $\forall i$.

This is referred to as the *Completion Time Test*.

It is worth to note that the condition $W_i \leqslant D_i$ of the above theorem must be verified for each i. Indeed, a common mistake is to think that if the lowest priority task will meet its deadline then all the tasks will meet their deadlines. This is false, since the schedulability of a task does not guarantee the schedulability of higher priority tasks. Consider, for

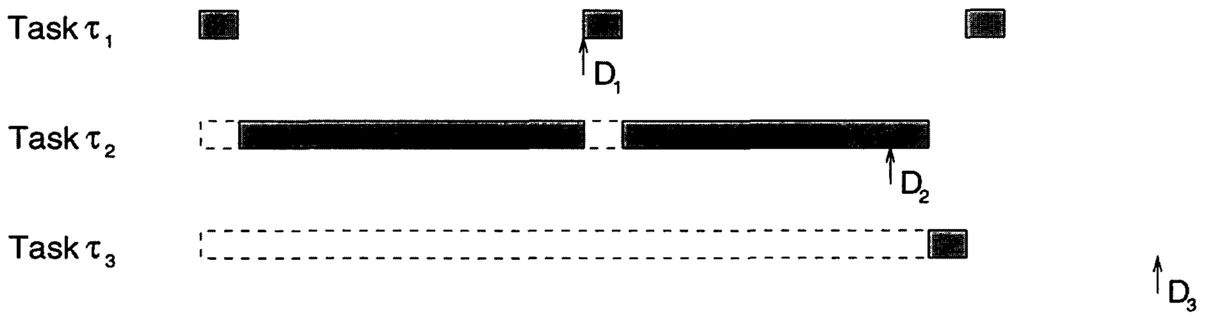

Fig. 4. The schedulability of the lowest priority task does not guarantee the schedulability of higher priority tasks.

Table 2
A task set with deadlines less than or equal to periods

Task	T_i	D_i	C_i
τ_1	100	100	10
τ_2	200	180	170
τ_3	250	250	10

example, the task set of Table 2, where tasks are indexed by decreasing priorities. The worst case completion time of task τ_1 is trivially 10. Computing the worst case completion time of task τ_2 by means of Eq. (1) yields: $W_2(1) = 170$, and $W_2(2) = W_2(3) = 190$. Thus the worst case completion time of τ_2 is 190 which is greater than its deadline, since $D_2 = 180$. However, τ_3, which is the lowest priority task, will meet its deadline: $W_3(1) = 10$, $W_3(2) = 190$, $W_3(3) = W_3(4) = 200$. Indeed, the worst case completion time of τ_3 is 200, which is less than 250, the deadline of τ_3. The schedule of τ_1, τ_2, and τ_3 is shown in Fig. 4.

2.3. Deadline-Monotonic scheduling

In the Rate-Monotonic algorithm, the task deadlines are assumed to coincide with the end of the task periods. In other words, the whole period of a task represents the time window within which a job must complete its execution. *Liu and Layland proved that giving higher priorities to tasks with narrower windows is optimal among fixed-priority algorithms.*

Relaxing the $D_i = T_i$ assumption into $D_i \leqslant T_i$ yields a time window narrower than the period. Leung and Whitehead [26] proved that, when the deadlines are less than or equal to the periods, the Rate-Monotonic priority assignment is no longer optimal. However, assigning higher priorities to tasks with narrower windows is still optimal among fixed-priority algorithms. Leung and Whitehead refer to this priority assignment policy as the *Deadline-Monotonic* scheduling algorithm. With this algorithm, the task having the smallest deadline is assigned the highest priority. In other words, τ_i has a higher priority than τ_j whenever D_i is smaller than D_j.

3. Generalised Rate-Monotonic analysis

Motivated by the need to face more general requirements of actual systems, much work has been

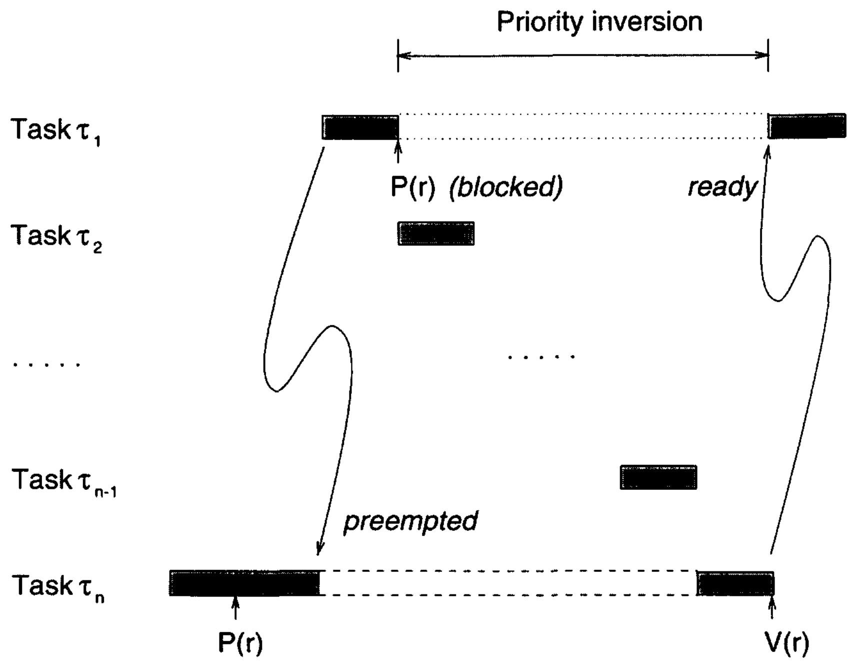

Fig. 5. Example of priority inversion. Tasks not involved with the critical section become the dominant factor causing the delay.

done upon the basic Rate-Monotonic analysis. This section deals with generalisations of the basic Rate-Monotonic analysis, which have weakened some assumptions and thus have widened its applicability.

3.1. Task synchronisation

In the previous section, independent tasks were assumed. In practice tasks interact, for example, to access a resource mutually exclusively. In this case, the application of task synchronisation primitives [40,37] may lead to the phenomenon of *priority inversion*, which occurs when a higher priority task is prevented from executing by a lower priority one. The duration of a priority inversion is a function of the task execution times and can be unbounded as shown in the example of Fig. 5. This example considers n tasks such that τ_1 and τ_n share a common resource r which must be accessed mutually exclusively. τ_n starts its execution, enters a critical section by locking r (e.g. by a P primitive) and thus accessing r. Then the highest priority task τ_1 preempts τ_n, attempts to lock r but remains blocked. Indeed, r is already accessed by another task, and thus τ_1 waits for the release of r. Before τ_n can complete its critical section and release r (e.g. by a V primitive), tasks $\tau_2, \ldots, \tau_{n-1}$, which do not use r, preempt τ_n. Thus task τ_1 is blocked by a lower priority task for an amount of time which is *a priori* unbounded. A good solution to overcome the above drawback is the so called *priority ceiling protocol*, which is briefly described in the following (for a more comprehensive review on resource control techniques for real-time systems see [1]).

Priority ceiling protocol. Priority inversion can be controlled by the *priority ceiling protocol* [34], which uses the concept of *priority inheritance* – a task executing a critical section and blocking a higher-priority task inherits that task priority for the duration of the critical section. The priority ceiling of a semaphore S is defined as the priority of the highest priority task that may lock S. Let $S^*(i)$ be the semaphore with the highest priority ceiling among all the semaphores currently locked by tasks different from τ_i. Task τ_i can lock a semaphore only if its priority is strictly higher than the priority ceiling of $S^*(i)$, otherwise it gets blocked on $S^*(i)$ (it will be awakened when the above condition becomes true).

Sha et al. [34] proved the following result:

Theorem 3.1. (i) *A task τ_i can be blocked (by lower priority tasks) for at most B_i time units, where B_i is the duration of the longest critical section executed by a task of lower priority than τ_i, guarded by a semaphore whose priority ceiling is greater than or equal to the priority of τ_i.* (ii) *The priority ceiling protocol is deadlock free.*

In order to take into account task synchronisation, Eq. (1) can be updated as follows:

$$W_i = C_i + B_i + \sum_{j \in \mathrm{hp}(i)} C_j \left\lceil \frac{W_i}{T_j} \right\rceil, \tag{2}$$

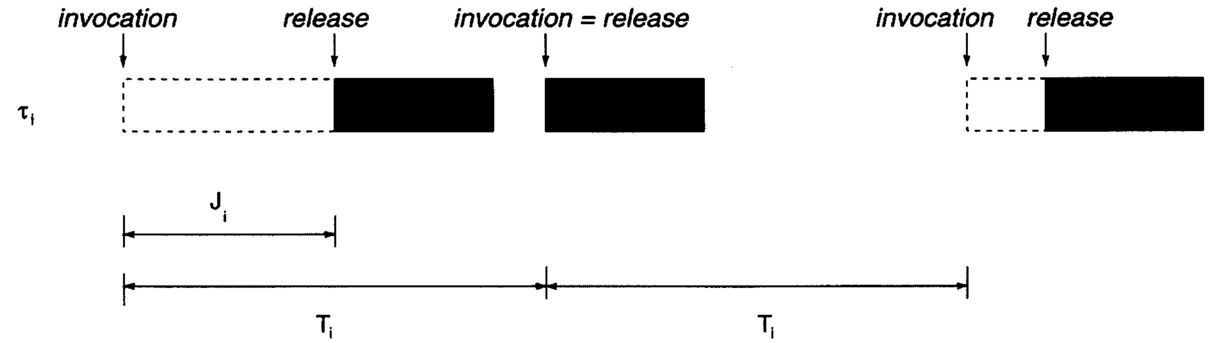

Fig. 6. The task release jitter. Note that the task inter-release time may be shorter than the period.

where B_i is the longest time duration in which task τ_i is blocked by lower priority tasks.

3.2. Release jitters

Until now each task was assumed to be released as soon as it arrives. However, it may happen that a task arrives but its release is delayed (for example owning to a scheduler that periodically polls task arrivals (*tick scheduling*)).

Thus there is a distinction between the *invocation time* – when a task is logically able to run – and the *release time* – when it is placed into the ready queue. The *release jitter* J_i is the worst case delay between the invocation time and the release time of τ_i, as shown in Fig. 6.

Response time with jitter. To take into account task release jitters, the previous analysis can be updated as follows [41]:

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

$$W_i = W_i^* + J_i.$$

3.3. Arbitrary deadlines

Another relaxation of the basic assumptions is to allow tasks to have arbitrary deadlines (i.e. deadlines greater than periods). In such a case, Liu and Layland's critical instant argument (Theorem 2.2) is no more valid – a task meeting its first deadline is not guaranteed to meet *all* its successive deadlines – and neither the Rate-Monotonic nor the Deadline-Monotonic algorithms are optimal anymore. Lehoczky [24] generalised Liu and Layland's bound for the case in which $D_i = kT_i$, with $k = 1, 2 \ldots$ (the *same constant* k for all tasks) by introducing the notion of *busy period.* Successively, Tindell and al. [41] extended the Completion Time Test, providing an exact test for tasks with arbitrary deadlines.

3.4. Scheduling non-periodic tasks

So far only periodic tasks with hard deadlines have been considered. This subsection describes some recent algorithms to schedule non-periodic tasks with both hard and soft deadlines. For non-periodic tasks with hard deadlines the goal is to guarantee that their deadlines will be always met, while for tasks with soft deadlines the goal is to provide low average response times.

3.4.1. Sporadic tasks

Sprunt et al. [39] showed how hard deadlines of sporadic tasks can be guaranteed using the so called *Sporadic Server.* This is a periodic task with an execution budget that handles sporadic requests at its assigned priority as long as the budget is available. When the budget is depleted, requests will be executed at background priority. The budget is preserved if no sporadic task is pending when the server is released. As long as the sporadic task is not released more frequently than the replenishment time of the Sporadic Server, its hard deadline can be guaranteed. Later, Audsley et al. [3] gave an exact analysis of sporadic tasks, showing that hard deadlines of sporadic tasks can be guaranteed without the overhead due to additional servers.

3.4.2. Sporadically periodic tasks

Some real-time systems have sporadically periodic tasks, i.e. tasks that arrive at some time, periodically execute for a bounded number of times (called *inner period*), and then do not re-arrive for a larger time (called *outer period*), as shown in Fig. 7. Tindell et al. [41] derived an exact analysis for tasks with this behaviour, showing how hard deadlines can be guaranteed. The schedulability analysis of the previous section can be updated to determine worst-case response times for sporadically periodic tasks. The details can be found in [41].

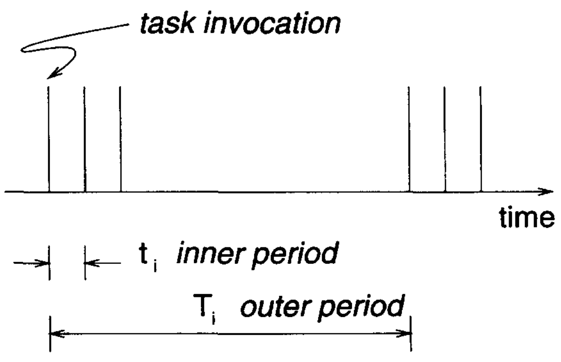

Fig. 7. The invocation pattern for a sporadically periodic task.

3.4.3. Irregular tasks

The invocation of an irregular task is essentially a random event. Therefore, a worst case analysis cannot be done and, as a result, hard deadlines of irregular tasks cannot be guaranteed in any way. Only soft deadlines for these tasks can be handled, by means of algorithms such as the Sporadic Server [39], the Extended Priority Exchange [38] or the *Dual Priority Scheduling* [15]. The latter represents the most elegant and efficient strategy to identify *spare* time – e.g. due to tasks which sometimes require less than their worst case computation times – for executing tasks with soft deadlines. Priorities are divided into three bands. Tasks with soft deadlines are assigned priorities in the middle band. Upon its release, each task with a hard deadline is assigned a priority in the lowest band. The priority of the task is promoted to the highest band after a fixed time from its release, which is the maximum delay that allows the deadline to be met in the worst case scenario. In [15], the authors also give an exact schedulability analysis based on the Completion Time Test.

3.5. Stability under transient overload

In many applications, the task execution times are stochastic. Thus considering worst execution times for the tasks can lower the processor utilisation, since worst execution times can be significatively larger than average execution times. On the other hand, scheduling a task set using average execution times may cause lower priority tasks to miss their deadlines under worst case conditions (in such a case, the system is said to be experiencing a *transient overload*). Since priorities are assigned according to task periods (or deadlines), this means that a critical task with a long period might miss its deadline. A technique to force critical tasks to have higher priorities (so that they will meet their deadlines under transient overload) is the *period transformation technique* [35], which consists in halving the task periods so as to increase their priorities.

3.6. Further reading

The interested reader could consult the following additional references:

- [2], whose authors provide an historical perspective on the development of fixed-priority preemptive scheduling up to the end of 1993;
- [21], which is a very clear introduction to the Rate-Monotonic analysis;
- [22], which is a book providing a comprehensive description of the Rate-Monotonic analysis and serves as a handbook for designing and analysing real-time systems.

Before concluding this section, it is worth mentioning one paper which is based on different assumptions and uses a different approach. In [43], Xu and Parnas introduce a static preemptive scheduling algorithm for hard-real-time systems with a single processor dealing with *exclusion relations*, *precedence constraints*, and *time constraints*. The algorithm uses a *branch & bound* strategy based on the Earliest Deadline First policy.

4. Multiprocessor scheduling

The Rate-Monotonic analysis, extended as outlined in the previous section to tackle more realistic requirements, is widely used for scheduling hard-real-time tasks on a single processor. How much of this analysis can be applied to multi-processor systems?

A major difficulty in scheduling on many processors is that the algorithm must specify not only an ordering of tasks on a single processor, but also an assignment of jobs to processors so as to *minimise* the number of processors.

There are two classes of priority-driven scheduling algorithms for multi-processor systems:
non-partitioning – processors are considered collectively as one entity and the dispatcher assigns the first ready task to the first free processor; in this way, different jobs of the same task may be executed on different processors;
partitioning – tasks are partitioned into groups so that each group of tasks can be feasibly scheduled on a single processor according to a given scheduling algorithm.

Since the Earliest Deadline First algorithm is optimal in the single processor case, it is tempting to think that it remains optimal also in the multiprocessor case. This is wrong: neither the Rate-

Table 3
Asymptotic worst-case performance ratio for task assignment heuristics. RMST requires that the maximum task utilisation factor is bounded by α

RMNF [16]	RMFF [16]	NF-M [13]	FFDUF [14]	RRMFF [31]	RMST [9]	RMGT [9]
2.67	2.33	2.28	2.0	2.0	$1/(1-\alpha)$	1.75

Monotonic nor the Earliest Deadline First algorithms are optimal for multiprocessors [16].

The most used approach for multiprocessor scheduling is the partitioning one, since tasks are allowed to be scheduled on each single processor according to the Rate-Monotonic (or Deadline-Monotonic) algorithm and partitioning of tasks among processors can be done by means of well-known heuristics.

4.1. Partitioning via Bin-Packing

The problem of assigning tasks to processors is similar to the *Bin-Packing* problem, where each task τ_i is a package of size equal to its utilisation factor C_i/T_i and each processor is a bin of size equal to one, if the Completion Time Test is used, or of size equal to ln 2, if Liu and Layland's utilisation bound is used.

The above problem has been widely studied and typical assignment schemes differ on the choice of the Bin-Packing heuristic. The most studied task model is Liu and Layland's one, where there are only periodic independent tasks without release jitters and with deadlines equal to periods. A schedulability test based on a utilisation bound – often the ln 2 bound – is used to constraint the assignment of tasks to processors.

The first assignment heuristics, called RMNF and RMFF, were proposed in [16], where tasks are picked by decreasing Rate-Monotonic priorities and assigned according to the *Next-Fit* and *First-Fit* Bin-Packing heuristics, respectively. More refined heuristics were successively introduced by Davari and Dhall [14,13] and Oh, Son et al. [31,9].

The performance of the above assignment heuristics is evaluated in terms of the asymptotic worst-case ratio $\lim_{N_0 \to \infty} N/N_0$, where N is the number of processors required by a given heuristic, and N_0 is the minimum number of processors needed. Table 3 summarises the performance of the most known heuristics.

It is worth noting that, as in the uniprocessor case, some of Liu and Layland's constraints on the task model can be relaxed by using the Completion Time Test instead of the utilisation bound. For example, task synchronisations can be handled by the Multiprocessor Priority Ceiling Protocol [32].

4.2. Partitioning without Bin-Packing

Before concluding this section, some papers are mentioned which are based on various task models and consider task partitioning without using Bin-Packing heuristics.

Tindell, Burns and Wellings [10] consider independent tasks with deadlines less than periods. The assignment of tasks to processors is obtained by *simulated annealing*, and tasks are scheduled on each processor according to the Deadline-Monotonic policy. Cheng and Agrawala [11] deal with no preemptable tasks with timing constraints on each job. Simulated annealing is used to compute a schedule, with length equal to the least common multiple of all task periods, which yields both the assignment of tasks to processors and the total order of job executions on all the processors. Fohler and Koza [17] and Verhoosel and Hammer [42] consider periodic fixed-priority tasks with resource requirements and use heuristics based on backtracking and implicit enumeration, respectively. Finally, Stankovic et al. [44] introduce dynamic scheduling heuristics for multiprocessors that take task resource requirements into account. Conflicts over resources are avoided in the scheduling phase, which allows mechanisms for mutual exclusion to be ignored.

5. Fault-tolerance

The purpose of a hard-real-time system is to provide time-critical services to its environment. Since the violation of a hard deadline can have catastrophic consequences, the system must be capable of providing a critical level of service even in the presence of one or more faults. A system can be designed to be *fault-tolerant* by incorporating additional components and algorithms which ensure that occurrences of erroneous states do not result in a failure of the whole system.

Fault-tolerant systems differ with respect to their behaviour in the presence of a fault. In some cases the aim is to continue to provide a full performance and all the functional capabilities of the system. In other cases, only a degraded performance or reduced functional capabilities are provided until the fault is removed [29].

Schemes for fault-tolerance also differ with respect to the types of faults which are to be tolerated. In particular, there are *hardware faults*, e.g. due to the incorrect behaviour of a processor, and *software faults*, e.g. due to an algorithmic error in a task design. Moreover, there is also a distinction between *transient faults*, which manifest themselves only temporarily, and *permanent faults*, which manifest themselves forever.

A variety of schemes have been proposed to support fault-tolerant computing in multi-processor systems. Such schemes can be partitioned into two broad classes.

● The *passive replication* technique prescribes that each task has one (or more) passive backup copies, that are executed only in the case of a fault – when a task fails, the passive copies of the task are started [23,27,19,7]. This technique is also called *temporal redundancy*, because it basically consists in reserving spare time for the reexecution of the faulty task (or of an alternate version of the task), and is usually employed in uniprocessor systems to tolerate software faults.

● The *active replication* technique prescribes that each task is replicated in two (or more) copies which are always executed on two (or more) processors – if any task fails, its active copy will continue to be executed [27,30]. This technique is also called *physical redundancy*, and is suited for multiprocessor systems to tolerate both hardware and software faults.

Only a few fault-tolerant scheduling algorithms for hard-real-time systems appeared in the literature, and only a very few of these deal with the Rate-Monotonic analysis. The most used approach for multiprocessors uses active replication and simply consists in duplicating on two sets of processors the schedule obtained for the fault-free case (e.g. by means of the Rate-Monotonic First-Fit policy [30]). Using such an active duplication approach, however, too many processors are required, since the number of processors used in the fault-free schedule is doubled. In the next subsections, an approach is described which combines both the passive and active task duplication in the same schedule and uses in practice less than twice the number of processors of the fault-free schedule [8].

5.1. Task duplication

For the sake of simplicity, only one permanent hardware fault is considered hereafter, but the duplication scheme to be presented can be extended to tolerate more than one (software or hardware) fault [8]. The following characteristics of the tasks are considered:

● $C_i \leqslant D_i \leqslant T_i$ and $J_i \leqslant D_i - C_i$ for all i;

● all tasks are periodic;

● all tasks are independent.

The following standard failure characteristics of the hardware are assumed:

● a processor is either non-faulty or ceases functioning, and a faulty processor cannot cause an incorrect behaviour in a non-faulty processor;

● the fault of a processor is detected by the remaining non-faulty processors after the fault, but within the closest completion time of a job that would have been scheduled on the faulty processor.

Two copies of the same task, the *primary* copy and the *backup* copy, are used, which are denoted, respectively,

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

and

$$\tau_{b(i)} = (C_{b(i)}, T_{b(i)}, D_{b(i)}, J_{b(i)}).$$

The primary and backup copies may have different execution times (for example, since they may correspond to different software implementations) and cannot be assigned to the same processor. The scheduling algorithm itself can determine whether a backup copy $\tau_{b(i)}$ must be *active* or can be *passive*, as soon as the primary copy τ_i is assigned to a processor and its worst-case completion time W_i is computed. If

$$D_i - W_i \geqslant C_{b(i)},$$

then there is enough time to execute $\tau_{b(i)}$ after τ_i within the same period and without violating the deadline, and thus $\tau_{b(i)}$ is scheduled as a passive copy, otherwise, $\tau_{b(i)}$ is scheduled as an active copy.

Backup copies are thus characterised as follows:

Active backup: An active copy is always executed in the absence of faults and 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, and thus $\tau_{b(i)}$ has the same period T_i of its primary copy, but has a release jitter $J_{b(i)}$ equal to the worst-case response time W_i of τ_i, as shown in Fig. 8. This jitter takes into account the first delayed release of $\tau_{b(i)}$, due to a failure of the processor running τ_i, namely: $\tau_{b(i)} = (C_{b(i)}, T_i, D_i, W_i)$.

5.2. Assignment and scheduling of tasks

Task copies are picked one at a time by decreasing Deadline-Monotonic priority order (each passive copy of a task immediately follows the primary copy of the same task) and assigned to a processor in which they fit, according to the partitioning policy (e.g. First-Fit, Best-Fit, etc.) used (the primary and backup copies of the same task are assigned to different processors). In order to determine whether a task copy can be assigned to a processor, the schedulability of the copy must be checked – by means of the Completion Time Test – together with all the task copies already assigned to that processor, both in the fault-free and faulty situations. All the copies assigned to each single processor are scheduled according to the Deadline-Monotonic algorithm.

In the absence of faults, the task set taking into account the fault-free situation is scheduled on each processor. This task set includes primary and active backup copies only. As soon as a processor failure is detected, the task set taking into account the faulty situation can replace at run time the old schedule in each non-faulty processor. This set includes primary copies, as before, but only the (active and passive) backup copies of primary tasks assigned to the failed processor. A passive copy is released in the next invocation period, if the execution of its primary copy was successfully completed by the failed processor before the fault was detected; it is released immediately, so as to be executed within the same invocation period, otherwise (see [8,6] for further details).

It is worth noting that primary copies scheduled on different processors can have their passive copies to share the same time on the same processor. This

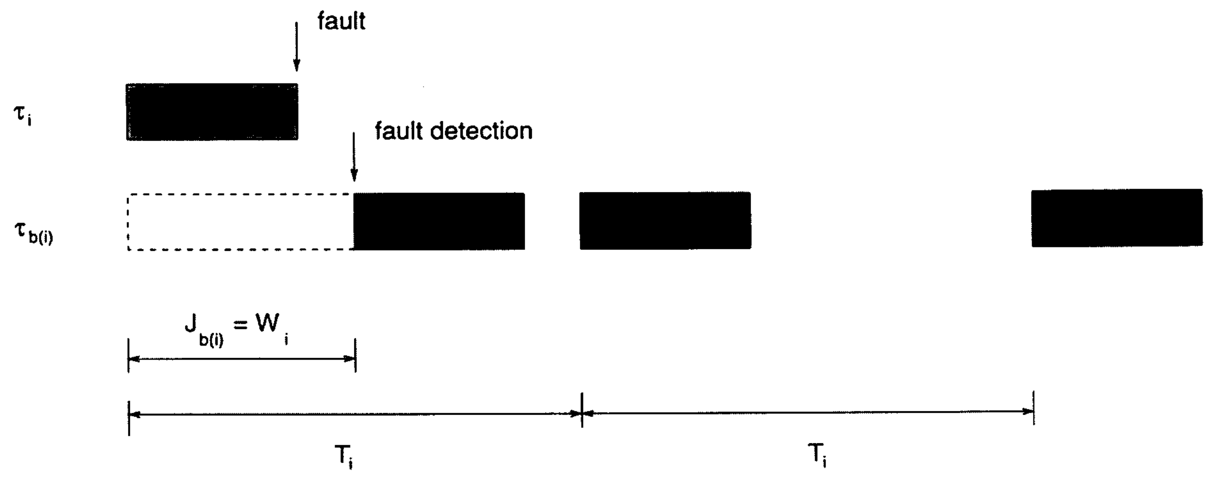

Fig. 8. The passive copy $\tau_{b(i)}$ is viewed as a task with period T_i (which is invocated together with its primary copy) that may experience a release jitter of W_i time units (that occurs only once, when it is invocated for the first time, owning to a failure of the processor executing τ_i).

allows the total number of processors to be considerably reduced with respect to those needed by the active duplication approach.

6. Conclusions

Hard-real-time systems are widely used in the industrialised society of today, and are expected to become larger and more complex in the society of tomorrow, due to the rapid advances in the computer hardware and communication networks. Since future systems are expected to be used for more and more critical applications, human, economical, and ecological catastrophes could follow if task deadlines will not be met.

In guaranteeing task deadlines, a *predictability/flexibility trade-off* arises. Indeed, predictability imposes that worst-case execution times and arrival rates are used to statically schedule the tasks, while flexibility requires that stochastic execution times and arrivals, hardware and software faults, as well as other system changes, are taken into account to dynamically make useful scheduling decisions. In this context, the Rate-Monotonic analysis represents a good compromise, since it guarantees predictability while being sufficiently flexible. However, further research should be focused on new scheduling approaches which could improve flexibility still guaranteeing predictability.

Future hard-real-time systems are expected to be distributed. Clearly, scheduling in a distributed network is different from scheduling in a centralised system [33,36]. Since there is no centralised scheduler, some resource requests of a node could be delayed or not satisfied at all, and scheduling decisions should be made by a node without having a complete knowledge of the decisions made by the other nodes in the network. Nevertheless, predictability should still be guaranteed in a distributed hard-real-time system.

References

[1] Audsley, N.C., "Resource control for hard-real-time systems: A review", Technical Report YCS 159, Department of Computer Science, University of York, August 1991.

[2] Audsley, N.C., Burns, A., Davies, R.I., Tindell, K.W., and Wellings, A.J., "Fixed priority preemptive scheduling: An historical perspective", *Real-Time Systems* 8 (1995) 173–198.

[3] Audsley, N.C., Burns, A., Richardson, M.F., and Wellings, A.J., "Hard-real-time scheduling: The deadline-monotonic approach", in: *Proceedings of the 8th Workshop on Real-Time Operating Systems and Software*, May 1991.

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

[5] Bertossi, A.A., and Bonuccelli, M.A., "A polynomial feasibility test for preemptive periodic scheduling of unrelated processors", *Discrete Applied Mathematics* 12 (1985) 195–201.

[6] Bertossi, A.A., and Fusiello, A., "Fault-tolerant deadline-monotonic algorithm for scheduling hard-real-time tasks", Technical Report UTM 483, Department of Mathematics, University of Trento, Italy, February 1996.

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

[8] Bertossi, A.A., Mancini, L.V., and Rossini, F., "A fault-tolerant rate-monotonic algorithm combining passive and active replication", Technical Report UTM 468, Department of Mathematics, University of Trento, Italy, June 1995.

[9] Burchard, A., Liebeherr, J., Oh, Y., and Son, S.H., "New strategies for assigning real-time tasks to multiprocessor systems", *IEEE Transactions on Computers* 44 (1995) 1429–1442.

[10] Burns, A., Tindell, K., and Wellings, A., "Allocating hard-real-time tasks: An NP-hard problem made easy", *Real-Time Systems* 4 (1992) 145–165.

[11] Cheng, S.T., and Agrawala, A.K., "Allocation and scheduling of real-time periodic tasks with relative timing constraints", Technical Report CS-TR-3402, Department of Computer Science, University of Maryland, College Park, MD, January 1995.

[12] Chetto, H., and Chetto, M., "Some results of the earliest deadline scheduling algorithm", *IEEE Transactions on Software Engineering* 15 (1989) 1261–1269.

[13] Davari, S., and Dhall, S., "On a periodic real-time task allocation problem", in: *Proceedings of the 19th Annual International Conference on System Sciences*, 1986, 133–141.

[14] Davari, S., and Dhall, S., "An on line algorithm for real-time task allocation", in: *Proceedings IEEE Real-Time Systems Symposium*, 1986, 194–200.

[15] Davies, R., and Wellings, A., "Dual priority scheduling", in: *Proceedings IEEE Real-Time Systems Symposium*, Pisa, Italy, December 1995.

[16] Dhall, S., and Liu, C.L., "On a real-time scheduling problem", *Operations Research* 26, (1978) 127–141.

[17] Fohler, G., and Koza, C., "Heuristic scheduling for distributed hard-real-time systems", Technical Report 12/1990, Institut für Technische Informatik, Technische Universität Wien, 1990.

[18] Garey, M.R., Graham, R.L., and Johnson, D.S., "Performance guarantees for scheduling algorithms", *Operations Research* 26 (1978) 3-21.

[19] Gosh, S., Melhem, R., and Mosse, D., "Enhancing real-time schedules to tolerate transient faults", in: *Proceedings IEEE Real-Time Systems Symposium*, Pisa, Italy, December 1995.

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

[21] Klein, M.H., Lehoczky, J.P., and Rajkumar, R., "Rate-monotonic analysis for real-time industrial computing", *IEEE Computer* 27 (1994) 24–33.

[22] Klein, M.H., Ralya, T., Pollak, B., Obenza, R., and Harbour, M.G., *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*, Kluwer Academic Publishers, Dordrecht, 1993.

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

[24] Lehoczky, J.P., "Real-time resource management techniques", in: J.J. Marciniak (ed.), *Encyclopedia of Software Engineering*, Wiley, New York, 1994, 1011–1020.

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

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

[27] Levi, S.-T., Mossé, D., and Agrawala, A.K., "Allocation of real-time computations under fault tolerant constraints", in: *Proceedings IEEE Real-Time Systems Symposium*, 1988, 161–170.

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

[29] Liu, J.W.S., Shih, W.-K., Lin, K.-J., Bettati, R., and Chung, J.-Y., "Imprecise computations", *Proceedings of the IEEE* 82 (1994) 83–93.

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

[31] Oh, Y., and Son, S.H., "Allocating fixed-priority periodic tasks on multiprocessor systems", *Real-Time Systems* 9 (1995) 207–239.

[32] Rajkumar, R., Sha, L., and Lehoczky, J.P., "Real-time synchronization protocols for multiprocessors", in: *Proceedings IEEE Real-Time Systems Symposium*, 1988, 259–269.

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

[34] Sha, L., Rajkumar, R., and Lehoczky, J.P., "Priority inheritance protocols: An approach to real-time synchronization", *IEEE Transactions on Computers* 39 (1990) 1175–1185.

[35] Sha, L., Rajkumar, R., and Sathaye, S.S., "Generalized rate-monotonic scheduling theory: A framework for developing real-time systems", *Proceedings of the IEEE* 82 (1994) 68–82.

[36] Sha, L., and Sathaye, S.S., "Distributed real-time systems design: Theoretical concepts and applications", Technical Report CMU/SEI-93-TR-2, Carnagie Mellon University, 1993.

[37] Silberschatz, A., and Galvin, P.B., *Operating System Concepts*, Addison-Wesley, Reading, MA, 1994.

[38] Sprunt, B., Lehoczky, J.P., and Sha, L., "Exploiting unused periodic time for aperiodic service using the extended priority exchange algorithm", in: *Proceedings IEEE Real-Time Systems Symposium*, 1988, 251–258.

[39] Sprunt, B., Sha, L., and Lehoczky, J.P., "Aperiodic task scheduling for hard-real-time systems", *The Journal of Real-Time Systems* 1 (1989) 27-60.

[40] Tanenbaum, A.S., *Modern Operating Systems*, Prentice-Hall, Englewood Cliffs, 1992.

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

[42] Verhoosel, J.P.C., Luit, E.J., Hammer, D.K., and Jansen, E., "A static scheduling algorithm for distributed hard-real-time systems", *Real-Time Systems* 3 (1991) 227-246.

[43] Xu, J., and Parnas, D.L., "Scheduling processes with release times, deadlines, precedence, and exclusion relations", *IEEE Transactions on Software Engineering* 16 (1990) 360–369.

[44] Zhao, W., Ramamritham, K., and Stankovic, J.A., "Preemptive scheduling under time and resource constraints," *IEEE Transactions on Computers* 36 (1987) 949-960.