8

HANDLING OVERLOAD CONDITIONS

8.1 INTRODUCTION

This chapter deals with the problem of scheduling real-time tasks in overload conditions; that is, in those critical situations in which the computational demand requested by the task set exceeds the time available on the processor, and hence not all tasks can complete within their deadlines.

In real-world applications, even when the system is properly designed and sized, a transient overload can occur for different reasons, such as changes in the environment, simultaneous arrivals of asynchronous events, faults of peripheral devices, or system exceptions. The major risk that could occur in these situations is that some critical task could miss its deadline, jeopardizing the correct behavior of the whole system.

If the operating system is not conceived to handle overloads, the effect of a transient overload can be catastrophic. Experiments carried out by Locke [Loc86] have shown that EDF can rapidly degrade its performance during overload intervals. This is due to the fact that EDF gives the highest priority to those processes that are close to missing their deadlines. There are cases in which the arrival of a new task can cause all the previous tasks to miss their deadlines. Such an undesirable phenomenon, called the *Domino effect*, is depicted in Figure 8.1.

Figure 8.1a shows a feasible schedule of a task set executed under EDF. However, if at time t_0 task J_0 is executed, all the previous tasks miss their deadlines (see Figure 8.1b). In such a situation, EDF does not provide any type of guarantee on which tasks meet their timing constraints. This is a very undesirable

behavior in those control applications in which a critical subset of tasks has to be guaranteed in all anticipated load conditions. In order to avoid domino effects, the operating system and the scheduling algorithm must be explicitly designed to handle transient overloads in a controlled fashion, so that the damage due to a deadline miss can be minimized.

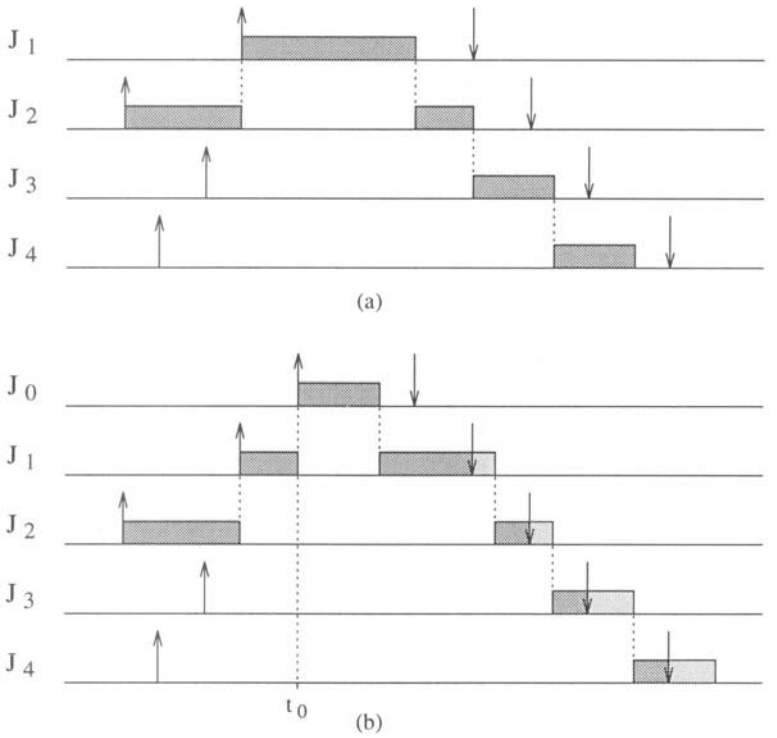

Figure 8.1 **a.** Feasible schedule with Earliest Deadline First, in normal load condition. **b.** Overload with domino effect due to the arrival of task J_0.

In the real-time literature, several scheduling algorithms have been proposed to deal with overloads. In 1984, Ramamritham and Stankovic [RS84] used EDF to dynamically guarantee incoming work via on-line planning, and, if a newly arriving task could not be guaranteed, the task was either dropped or distributed scheduling was attempted. The dynamic guarantee performed in this approach had the effect of avoiding the catastrophic effects of overload on EDF.

In 1986, Locke [Loc86] developed an algorithm that makes a best effort at scheduling tasks based on earliest deadline with a rejection policy based on removing tasks with the minimum value density. He also suggested that removed tasks remain in the system until their deadline has passed. The algorithm computes the variance of the total slack time in order to find the probability that the available slack time is less than zero. The calculated probability is used to detect a system overload. If it is less than the user prespecified threshold, the algorithm removes the tasks in increasing value density order.

In Biyabani et. al. [BSR88] the previous work of Ramamritham and Stankovic was extended to tasks with different values, and various policies were studied to decide which tasks should be dropped when a newly arriving task could not be guaranteed. This work used values of tasks such as in Locke's work but used an exact characterization of the first overload point rather than a probabilistic estimate that overload might occur.

Haritsa, Livny, and Carey [HLC91] presented the use of a feedback controlled EDF algorithm for use in real-time database systems. The purpose of their work was to obtain good average performance for transactions even in overload. Since they were working in a database environment, they assumed no knowledge of transaction characteristics, and they considered tasks with soft deadlines that are not guaranteed.

In real-time Mach [TWW87] tasks were ordered by EDF and overload was predicted using a statistical guess. If overload was predicted, tasks with least value were dropped.

Other general work on overload in real-time systems has also been done. For example, Sha [SLR88] showed that the Rate-Monotonic algorithm has poor properties in overload. Thambidurai and Trivedi [TT89] studied transient overloads in fault-tolerant real-time systems, building and analyzing a stochastic model for such systems. However, they provided no details on the scheduling algorithm itself. Schwan and Zhou [SZ92] did on-line guarantees based on keeping a slot list and searching for free-time intervals between slots. Once schedulability is determined in this fashion, tasks are actually dispatched using EDF. If a new task cannot be guaranteed, it is discarded.

Zlokapa, Stankovic, and Ramamritham [Zlo93] proposed an approach called *well-time scheduling*, which focuses on reducing the guarantee overhead in heavily loaded systems by delaying the guarantee. Various properties of the approach were developed via queueing theoretic arguments, and the results

were a multilevel queue (based on an analytical derivation), similar to that found in [HLC91] (based on simulation).

More recent approaches will be described in the following sections. Before presenting specific methods and theoretical results on overload, the concept of overload, and, in general, the meaning of computational load for real-time systems is defined in the next section.

8.2 LOAD DEFINITIONS

In a real-time system, the definition of computational workload depends on the temporal characteristics of the computational activities. For non-real-time or soft real-time tasks, a commonly accepted definition of workload refers to the standard queueing theory, according to which a load ρ, also called *traffic intensity*, represents the expected number of job arrivals per mean service time. If C is the mean service time and λ is the average interarrival rate of the jobs, the load can be computed as

$$\rho = \lambda C.$$

Notice that this definition does not take deadlines into account; hence, it is not particularly useful to describe real-time workloads. In a hard real-time environment, a system is overloaded when, based on worst-case assumptions, there is no feasible schedule for the current task set, so one or more tasks will miss their deadline.

If the task set consists of n independent preemptable periodic tasks, whose relative deadlines are equal to their period, then the system load ρ is equivalent to the processor utilization factor:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i},$$

where C_i and T_i are the computation time and the period of task τ_i, respectively. In this case, a load $\rho > 1$ means that the total computation time requested by the periodic activities in their *hyperperiod* exceeds the available time on the processor; therefore, the task set cannot be scheduled by any algorithm.

For a generic set of real-time jobs that can be dynamically activated, the system load varies at each job activation and it is a function of the jobs' deadlines. A general definition of load has been proposed by Baruah et al. [BKM+92], who

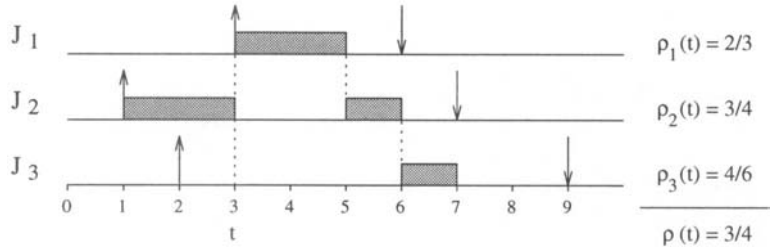

Figure 8.2 Load calculation for a set of three real-time tasks.

say that a hard real-time environment has a loading factor ρ if and only if it is guaranteed that there will be no interval of time $[t_x, t_y]$ such that the sum of the execution times of all jobs making requests and having deadlines within this interval is greater than $\rho(t_y - t_x)$. Although this definition is quite general and of theoretical value, it is of little practical use for load calculation, since the number of intervals $[t_x, t_y]$ can be very large.

A simpler method for calculating the load in a dynamic real-time environment has been proposed by Buttazzo and Stankovic in [BS95], where the load is computed at each job activation (r_i), and the number of intervals in which the computation in done is limited by the number of deadlines (d_i). The method for computing the load is based on the consideration that, for a single job J_i, the load is given by the ratio of its computation time C_i and its relative deadline $D_i = d_i - r_i$. For example, if $C_i = D_i$ (that is, the job does not have slack time), the load in the interval $[r_i, d_i]$ is one. When a new job arrives, the load can be computed from the last request time, which is also the current time t, and the longest deadline, say d_n. In this case, the intervals that need to be considered for the computation are $[t, d_1]$, $[t, d_2]$, ..., $[t, d_n]$. In general, the processor load in the interval $[t, d_i]$ is given by

$$\rho_i(t) = \frac{\sum_{d_k \leq d_i} c_k(t)}{(d_i - t)},$$

where $c_k(t)$ refers to the remaining execution time of job J_k with deadline less than or equal to d_i. Hence, the total load in the interval $[t, d_n]$ can be computed as the maximum among all $\rho_i(t)$; that is,

$$\rho = \max_i \rho_i(t).$$

Figure 8.2 shows an example of load calculation for a set of three real-time tasks.

8.3 PERFORMANCE METRICS

When a real-time system is underloaded and dynamic activation of tasks is not allowed, there is no need to consider task importance in the scheduling policy, since there exist optimal scheduling algorithms that can guarantee a feasible schedule under a set of assumptions. For example, Dertouzos [Der74] proved that EDF is an optimal algorithm for preemptive, independent tasks when there is no overload.

On the contrary, when tasks can be activated dynamically and an overload occurs, there are no algorithms that can guarantee a feasible schedule of the task set. Since one or more tasks will miss their deadlines, it is preferable that late tasks be the less important ones in order to achieve graceful degradation. Hence, in overload conditions, distinguishing between time constraints and importance is crucial for the system. In general, the importance of a task is not related to its deadline or its period; thus, a task with a long deadline could be much more important than another one with an earlier deadline. For example, in a chemical process, monitoring the temperature every ten seconds is certainly more important than updating the clock picture on the user console every second. This means that, during a transient overload, is better to skip one or more clock updates rather than miss the deadline of a temperature reading, since this could have a major impact on the controlled environment.

In order to specify importance, an additional parameter is usually associated with each task, its *value*, that can be used by the system to make scheduling decisions.

The value associated with a task reflects its importance with respect to the other tasks in the set. The specific assignment depends on the particular application. For instance, there are situations in which the value is set equal to the task computation time; in other cases, it is an arbitrary integer number in a given range; in other applications, it is set equal to the ratio of an arbitrary number (which reflects the importance of the task) and the task computation time; this ratio is referred to as the *value density.*

In a real-time system, however, the actual value of a task also depends on the time at which the task is completed; hence, the task importance can be better described by an utility function. Figure 8.3 illustrates some utility functions that can be associated with tasks in order to describe their importance. According to this view, a non-real-time task, which has no time constraints, has a low constant value, since it always contributes to the system value whenever it com-

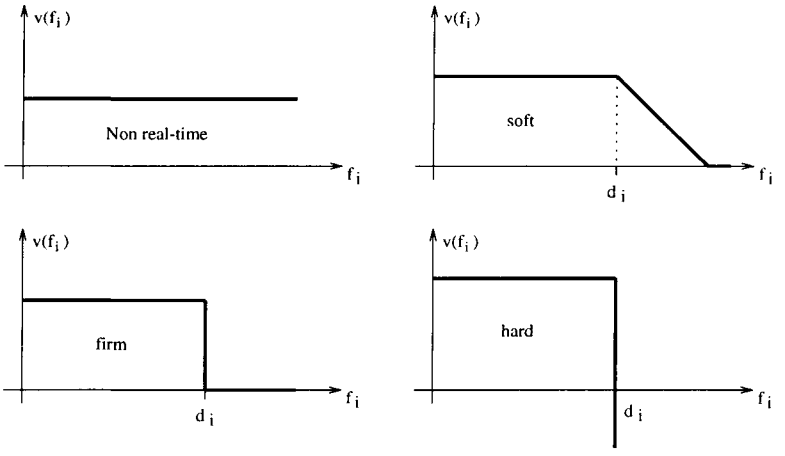

Figure 8.3 Utility functions that can be associated to a task to describe its importance.

pletes its execution. On the contrary, a hard task contributes to a value only if it completes within its deadline, and, since a deadline miss would jeopardize the behavior of the whole system, the value after its deadline can be considered minus infinity in many situations. A task with a soft deadline, instead, can still give a value to the system if executed after its deadline, although this value may decrease with time. Then, there can be real-time activities, so-called *firm*, that do not jeopardize the system but give zero value if completed after their deadline.

Once the importance of each task has been defined, the performance of a scheduling algorithm can be measured by accumulating the values of the task utility functions computed at their completion time. Specifically, we define as *cumulative value* of a scheduling algorithm A the following quantity:

$$\Gamma_A = \sum_{i=1}^{n} v(f_i).$$

Given this metric, a scheduling algorithm is optimal if it maximizes the cumulative value achievable on a task set.

Notice that if a hard task misses its deadline, the cumulative value achieved by the algorithm is minus infinity, even though all other tasks completed before their deadlines. For this reason, all activities with hard timing constraints should be guaranteed a priori by assigning them dedicated resources (included

processors). If all hard tasks are guaranteed a priori, the objective of a real-time scheduling algorithm should be to guarantee a feasible schedule in underload conditions and maximize the cumulative value of soft and firm tasks during transient overloads.

Given a set of n jobs $J_i(C_i, D_i, V_i)$, where C_i is the worst-case computation time, D_i is the relative deadline, and V_i is the importance value gained by the system when the task completes within its deadline, the maximum cumulative value achievable on the task set is clearly equal to the sum of all values V_i; that is, $\Gamma_{max} = \sum_{i=1}^{n} V_i$. In overload conditions, this value cannot be achieved, since one or more tasks will miss their deadlines. Hence, if Γ^* is the maximum cumulative value that can be achieved by any algorithm on a task set in overload conditions, the performance of a scheduling algorithm A can be measured by comparing the cumulative value Γ_A obtained by A with the maximum achievable value Γ^*.

8.3.1 On-line versus clairvoyant scheduling

Since dynamic environments require on-line scheduling, it is important to analyze the properties and the performance of on-line scheduling algorithms in overload conditions.

Although there are optimal on-line algorithms in underload conditions, it is easy to show that no optimal on-line algorithms exist in overloads. Consider for example the task set shown in Figure 8.4, consisting of three tasks $J_1(10, 11, 10)$, $J_2(6, 7, 6)$, $J_3(6, 7, 6)$.

Without loss of generality, we assume that the importance values associated to the tasks are proportional to their execution times ($V_i = C_i$) and that tasks are firm, so no value is accumulated if a task completes after its deadline. If J_1 and J_2 simultaneously arrive at time $t_0 = 0$, there is no way to maximize the cumulative value without knowing the arrival time of J_3. In fact, if J_3 arrives at time $t = 4$ or before, the maximum cumulative value is $\Gamma^* = 10$ and can be achieved by scheduling task J_1 (see Figure 8.4a). However, if J_3 arrives between time $t = 5$ and time $t = 8$, the maximum cumulative value is $\Gamma^* = 12$, achieved by scheduling task J_2 and J_3, and discarding J_1 (see Figure 8.4b). Notice that if J_3 arrives at time $t = 9$ or later (see Figure 8.4c), then the maximum cumulative value is $\Gamma^* = 16$ and can be accumulated by scheduling tasks J_1 and J_3. Hence, at time $t = 0$, without knowing the arrival time of

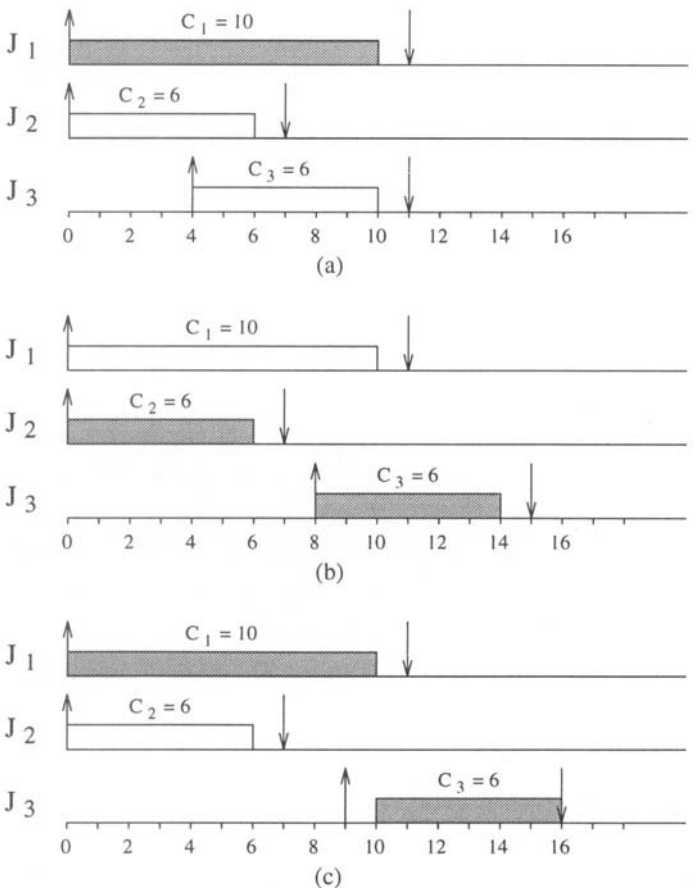

Figure 8.4 No optimal on-line optimal algorithms exist in overload conditions, since the schedule that maximizes Γ depends on the knowledge of future arrivals. **a.** $\Gamma_{max} = 10$. **b.** $\Gamma_{max} = 12$. **c.** $\Gamma_{max} = 16$.

J_3, no on-line algorithm can decide which task to schedule for maximizing the cumulative value.

What this example shows is that, without an a priori knowledge of the task arrival times, no on-line algorithm can guarantee the maximum cumulative value Γ^*. This value can only be achieved by an ideal clairvoyant scheduling algorithm that knows the future arrival time of any task. Although the optimal clairvoyant scheduler is a pure theoretical abstraction, it can be used as a reference model to evaluate the performance of on-line scheduling algorithms in overload conditions.

8.3.2 Competitive factor

The cumulative value obtained by a scheduling algorithm on a task set represents a measure of its performance for that particular task set. To characterize an algorithm with respect to worst-case conditions, however, the minimum cumulative value that can be achieved by the algorithm on any task set should be computed. A parameter that measures the worst-case performance of a scheduling algorithm is the *competitive factor*, introduced by Baruah et al. in [BKM+92].

Definition 8.1 *A scheduling algorithm A has a* competitive factor φ_A *if and only if it can guarantee a cumulative value*

$$\Gamma_A \geq \varphi_A \Gamma^*,$$

where Γ^* is the cumulative value achieved by the optimal clairvoyant scheduler.

From this definition, we can notice that the competitive factor is a real number $\varphi_A \in [0, 1]$. If an algorithm A has a competitive factor φ_A, it means that A can achieve a cumulative value Γ_A at least φ_A times the cumulative value achievable by the optimal clairvoyant scheduler on *any* task set.

If the overload has an infinite duration, then no on-line algorithm can guarantee a competitive factor greater than zero. In real situations, however, overloads are intermittent and usually have a short duration; hence, it is desirable to use scheduling algorithms with a high competitive factor.

Unfortunately, without any form of guarantee, the plain EDF algorithm has a zero competitive factor. To show this fact it is sufficient to find an overload

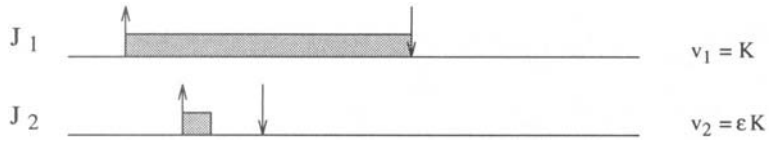

Figure 8.5 Situation in which EDF has an arbitrarily small competitive factor.

situation in which the cumulative value obtained by EDF can be arbitrarily reduced with respect to that one achieved by the clairvoyant scheduler. Consider the example shown in Figure 8.5, where tasks have a value proportional to their computation time. This is an overload condition because both tasks cannot be completed within their deadlines.

When task J_2 arrives, EDF preempts J_1 in favor of J_2, which has an earlier deadline, so it gains a cumulative value of C_2. On the other hand, the clairvoyant scheduler always gains $C_1 > C_2$. Since the ratio C_2/C_1 can be made arbitrarily small, it follows that the competitive factor of EDF is zero.

An important theoretical result found in [BKM+92] is that there exists an upper bound on the competitive factor of any on-line algorithm. This is stated by the following theorem.

Theorem 8.1 (Baruah et al.) *In systems where the loading factor is greater than 2 ($\rho > 2$) and tasks' values are proportional to their computation times, no on-line algorithm can guarantee a competitive factor greater than* 0.25.

The proof of this theorem is done by using an adversary argument, in which the on-line scheduling algorithm is identified as a player and the clairvoyant scheduler as the adversary. In order to propose worst-case conditions, the adversary dynamically generates the sequence of tasks depending on the player decisions, to minimize the ratio Γ_A/Γ^*. At the end of the game, the adversary shows its schedule and the two cumulative values are computed. Since the player tries to do his best in worst-case conditions, the ratio of the cumulative values gives the upper bound of the competitive factor for any on-line algorithm.

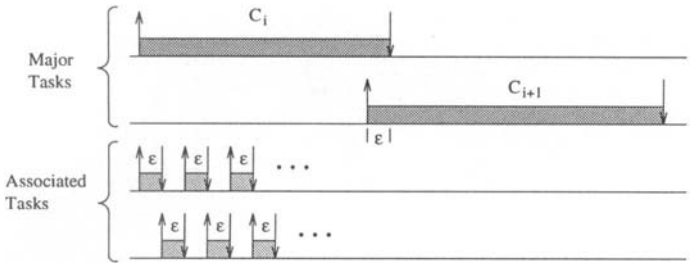

Figure 8.6 Task sequence generated by the adversary.

Task generation strategy

To create an overload condition and force the hand of the player, the adversary creates two types of tasks: *major* tasks, of length C_i, and *associated* tasks, of length ϵ arbitrarily small. These tasks are generated according to the following strategy (see Figure 8.6):

- All tasks have zero laxity; that is, the relative deadline of each task is exactly equal to its computation time.
- After releasing a major task J_i, the adversary releases the next major task J_{i+1} at time ϵ before the deadline of J_i; that is, $r_{i+1} = d_i - \epsilon$.
- For each major task J_i, the adversary may also create a sequence of associated tasks, in the interval $[r_i, d_i]$, such that each subsequent associated task is released at the deadline of the previous one in the sequence (see Figure 8.6). Note that the resulting load is $\rho = 2$. Moreover, any algorithm that schedules any one of the associated tasks cannot schedule J_i within its deadline.
- If the player chooses to abandon J_i in favor of an associated task, the adversary stops the sequence of associated tasks.
- If the player chooses to schedule a major task J_i, the sequence of tasks terminates with the release of J_{i+1}.
- Since the overload must have a finite duration, the sequence continues until the release of J_m, where m is a positive finite integer.

Notice that the sequence of tasks generated by the adversary is constructed in such a way that the player can schedule at most one task within its deadline

(either a major task or an associated task). Clearly, since task values are equal to their computation times, the player never abandons a major task for an associated task, since it would accumulate a negligible value; that is, ϵ. On the other hand, the values of the major tasks (that is, their computation times) are chosen by the adversary to minimize the resulting competitive factor. To find the worst-case sequence of values for the major tasks, let

$$J_0, J_1, J_2, \ldots, J_i, \ldots, J_m$$

be the longest sequence of major tasks that can be generated by the adversary and, without loss of generality, assume that the first task has a computation time equal to $C_0 = 1$. Now, consider the following three cases.

Case 0. If the player decides to schedule J_0, the sequence terminates with J_1. In this case, the cumulative value gained by the player is C_0, whereas the one obtained by the adversary is $(C_0 + C_1 - \epsilon)$. Notice that this value can be accumulated by the adversary either by executing all the associated tasks, or by executing J_0 and all associated tasks started after the release of J_1. Being ϵ arbitrarily small, it can be neglected in the cumulative value. Hence, the ratio among the two cumulative values is

$$\varphi_0 = \frac{C_0}{C_0 + C_1} = \frac{1}{1 + C_1} = \frac{1}{k}.$$

If $1/k$ is the value of this ratio ($k > 0$), then $C_1 = k - 1$.

Case 1. If the player decides to schedule J_1, the sequence terminates with J_2. In this case, the cumulative value gained by the player is C_1, whereas the one obtained by the adversary is $(C_0 + C_1 + C_2)$. Hence, the ratio among the two cumulative values is

$$\varphi_1 = \frac{C_1}{C_0 + C_1 + C_2} = \frac{k - 1}{k + C_2}.$$

In order not to lose with respect to the previous case, the adversary has to choose the value of C_2 so that $\varphi_1 \leq \varphi_0$; that is,

$$\frac{k - 1}{k + C_2} \leq \frac{1}{k},$$

which means

$$C_2 \geq k^2 - 2k.$$

However, observe that, if $\varphi_1 < \varphi_0$, the execution of J_0 would be more convenient for the player, thus the adversary decides to make $\varphi_1 = \varphi_0$; that is,

$$C_2 = k^2 - 2k.$$

Case i. If the player decides to schedule J_i, the sequence terminates with J_{i+1}. In this case, the cumulative value gained by the player is C_i, whereas the one obtained by the adversary is $(C_0 + C_1 + \ldots + C_{i+1})$. Hence, the ratio among the two cumulative values is

$$\varphi_i = \frac{C_i}{\sum_{j=0}^{i} C_j + C_{i+1}}.$$

As in the previous case, to prevent any advantage to the player, the adversary will choose tasks' values so that

$$\varphi_i = \varphi_{i-1} = \ldots = \varphi_0 = \frac{1}{k}.$$

Thus,

$$\varphi_i = \frac{C_i}{\sum_{j=0}^{i} C_j + C_{i+1}} = \frac{1}{k},$$

and hence

$$C_{i+1} = kC_i - \sum_{j=0}^{i} C_j.$$

Thus, the worst-case sequence for the player occurs when major tasks are generated with the following computation times:

$$\begin{cases} C_0 = 1 \\ C_{i+1} = kC_i - \sum_{j=0}^{i} C_j. \end{cases} \tag{8.1}$$

Proof of the bound

Whenever the player chooses to schedule a task J_i, the sequence stops with J_{i+1} and the ratio of the cumulative values is

$$\varphi_i = \frac{C_i}{\sum_{j=0}^{i} C_j + C_{i+1}} = \frac{1}{k}.$$

However, if the player chooses to schedule the last task J_m, the ratio of the cumulative values is

$$\varphi_m = \frac{C_m}{\sum_{j=0}^{m} C_j}.$$

Notice that if k and m can be chosen such that $\varphi_m \leq 1/k$; that is,

$$\frac{C_m}{\sum_{j=0}^{m} C_j} \leq \frac{1}{k}, \tag{8.2}$$

then we can conclude that, in the worst case, a player cannot achieve a cumulative value greater than $1/k$ times the adversary's value. Notice that

$$\frac{C_m}{\sum_{j=0}^{m} C_j} = \frac{C_m}{\sum_{j=0}^{m-1} C_j + C_m} = \frac{C_m}{\sum_{j=0}^{m-1} C_j + kC_{m-1} - \sum_{j=0}^{m-1} C_j} = \frac{C_m}{kC_{m-1}}.$$

Hence, if there exists an m which satisfies equation (8.2), it also satisfies the following equation:

$$C_m \leq C_{m-1}. \tag{8.3}$$

Thus, (8.3) is satisfied if and only if (8.2) is satisfied.

From (8.1) we can also write

$$\begin{aligned} C_{i+2} &= kC_{i+1} - \sum_{j=0}^{i+1} C_j \\ C_{i+1} &= kC_i - \sum_{j=0}^{i} C_j, \end{aligned}$$

and subtracting the second equation from the first one, we obtain

$$C_{i+2} - C_{i+1} = k(C_{i+1} - C_i) - C_{i+1}$$

that is,

$$C_{i+2} = k(C_{i+1} - C_i).$$

Hence, equation (8.1) is equivalent to

$$\begin{cases} C_0 &= 1 \\ C_1 &= k - 1 \\ C_{i+2} &= k(C_{i+1} - C_i). \end{cases} \tag{8.4}$$

From this result, we can say that the tightest bound on the competitive factor of an on-line algorithm is given by the smallest ratio $1/k$ (equivalently, the largest k) such that (8.4) satisfies (8.3). Equation (8.4) is a recurrence relation that can be solved by standard techniques [Sha85]. The characteristic equation of (8.4) is

$$x^2 - kx + k = 0,$$

which has roots

$$x_1 = \frac{k + \sqrt{k^2 - 4k}}{2} \quad \text{and} \quad x_2 = \frac{k - \sqrt{k^2 - 4k}}{2}.$$

When $k = 4$, we have

$$C_i = d_1 i 2^i + d_2 2^i, \tag{8.5}$$

and when $k \neq 4$ we have

$$C_i = d_1 (x_1)^i + d_2 (x_2)^i, \tag{8.6}$$

where values for d_1 and d_2 can be found from the boundary conditions expressed in (8.4). We now show that for $(k = 4)$ and $(k > 4)$ C_i will diverge, so equation (8.3) will not be satisfied, whereas for $(k < 4)$ C_i will satisfy (8.3).

Case $(k = 4)$. In this case, $C_i = d_1 i 2^i + d_2 2^i$ and, from the boundary conditions, we find $d_1 = 0.5$ and $d_2 = 1$. Thus,

$$C_i = (\frac{i}{2} + 1) 2^i,$$

which clearly diverges. Hence, for $k = 4$, equation (8.3) cannot be satisfied.

Case $(k > 4)$. In this case, $C_i = d_1 (x_1)^i + d_2 (x_2)^i$, where

$$x_1 = \frac{k + \sqrt{k^2 - 4k}}{2} \quad \text{and} \quad x_2 = \frac{k - \sqrt{k^2 - 4k}}{2}.$$

From the boundary conditions we find

$$\begin{cases} C_0 = d_1 + d_2 = 1 \\ C_1 = d_1 x_1 + d_2 x_2 = k - 1 \end{cases}$$

that is,

$$\begin{cases} d_1 = \frac{1}{2} + \frac{k-2}{2\sqrt{k^2-4k}} \\ d_2 = \frac{1}{2} - \frac{k-2}{2\sqrt{k^2-4k}}. \end{cases}$$

Since $(x_1 > x_2)$, $(x_1 > 2)$, and $(d_1 > 0)$, C_i will diverge, and hence, also for $k > 4$, equation (8.3) cannot be satisfied.

Case $(k < 4)$. In this case, since $(k^2 - 4k < 0)$, both the roots x_1, x_2 and the coefficients d_1, d_2 are complex conjugates, so they can be represented as follows:

$$\begin{cases} d_1 = s e^{j\theta} \\ d_2 = s e^{-j\theta} \end{cases} \qquad \begin{cases} x_1 = r e^{j\omega} \\ x_2 = r e^{-j\omega}, \end{cases}$$

where s and r are real numbers, $j = \sqrt{-1}$, and θ and ω are angles such that, $-\pi/2 < \theta < 0$, $0 < \omega < \pi/2$. Equation (8.6) may therefore be rewritten as

$$\begin{aligned} C_i &= se^{j\theta}r^i e^{ji\omega} + se^{-j\theta}r^i e^{-ji\omega} = \\ &= sr^i[e^{j(\theta+i\omega)} + e^{-j(\theta+i\omega)}] = \\ &= sr^i[\cos(\theta + i\omega) + j\sin(\theta + i\omega) + \cos(\theta + i\omega) - j\sin(\theta + i\omega)] = \\ &= 2sr^i \cos(\theta + i\omega). \end{aligned}$$

Being $\omega \neq 0$, $\cos(\theta + i\omega)$ is negative for some $i \in \mathbf{N}$, which implies that there exists a finite m that satisfies (8.3).

Since (8.3) is satisfied for $k < 4$, the largest k that determines the competitive factor of an on-line algorithm is certainly less than 4. Therefore, we can conclude that 1/4 is an upper bound on the competitive factor that can be achieved by any on-line scheduling algorithm in an overloaded environment. Hence, Theorem 8.1 follows.

Extensions

Theorem 8.1 establishes an upper bound on the competitive factor of on-line scheduling algorithms operating in heavy load conditions ($\rho > 2$). In lighter overload conditions ($1 < \rho \leq 2$), the bound is a little higher, and it is given by the following theorem [BR91].

Theorem 8.2 (Baruah et al.) *In real-time environments with a loading factor ρ, $1 < \rho \leq 2$, and task values equal to computation times, no on-line algorithm can guarantee a competitive factor greater than p, where p satisfies*

$$4[1 - (\rho - 1)p]^3 = 27p^2. \tag{8.7}$$

Notice that, for $\rho = 1 + \epsilon$, equation (8.7) is satisfied for $p = \sqrt{4/27} \simeq 0.385$, whereas, for $\rho = 2$, the same equation is satisfied for $p = 0.25$.

In summary, whenever the system load does not exceed one, the upper bound of the competitive factor is obviously one. As the load exceeds one, the bound immediately falls to 0.385, and as the load increases from one to two, it falls from 0.385 to 0.25. For loads higher than two, the competitive factor limitation remains at 0.25. The bound on the competitive factor as a function of the load is shown in Figure 8.7.

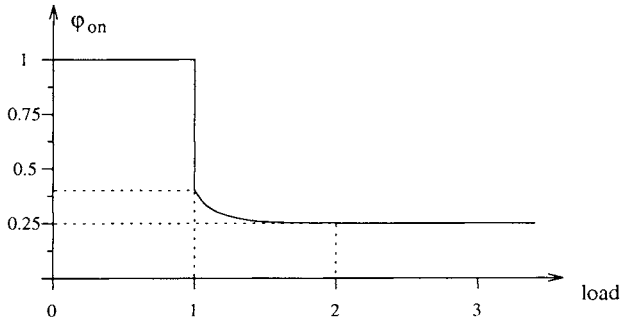

Figure 8.7 Bound of the competitive factor of an on-line scheduling algorithm as a function of the load.

Baruah et al. [BR91] also showed that, when using value density metrics (where the value density of a task is its value divided by its computation time), the best that an on-line algorithm can guarantee in environments with load $\rho > 2$ is

$$\frac{1}{(1+\sqrt{k})^2},$$

where k is the important ratio between the highest and the lowest value density task in the system.

In environments with a loading factor ρ, $1 < \rho \leq 2$, and an importance ratio k, two cases must be considered. Let $q = k(\rho - 1)$. If $q \geq 1$, then no on-line algorithm can achieve a competitive factor greater than

$$\frac{1}{(1+\sqrt{q})^2},$$

whereas, if $q < 1$, no on-line algorithm can achieve a competitive factor greater than p, where p satisfies

$$4(1-qp)^3 = 27p^2.$$

Before concluding the discussion on the competitive analysis, it is worth pointing out that all the above bounds are derived under very restrictive assumptions, such as all tasks have zero laxity, the overload can have an arbitrary (but finite) duration, and task's execution time can be arbitrarily small. In most real-world applications, however, tasks characteristics are much less restrictive; therefore, the 1/4th bound has only a theoretical validity, and more work is needed to derive other bounds based on more knowledge of the actual environmental load conditions. An analysis of on-line scheduling algorithms under different types of adversaries has been presented by Karp in [Kar92].

8.4 SCHEDULING SCHEMES FOR OVERLOAD

With respect to the strategy used to predict and handle overloads, most of the scheduling algorithms proposed in the literature can be divided into three main classes, illustrated in Figure 8.8:

- **Best Effort**. This class includes those algorithms with no prediction for overload conditions. At its arrival, a new task is always accepted into the ready queue, so the system performance can only be controlled through a proper priority assignment.
- **Guarantee**. This class includes those algorithms in which the load on the processor is controlled by an acceptance test executed at each task arrival. Typically, whenever a new task enters the system, a guarantee routine verifies the schedulability of the task set based on worst-case assumptions. If the task set is found schedulable, the new task is accepted in the ready queue; otherwise, it is rejected.
- **Robust**. This class includes those algorithms that separate timing constraints and importance by considering two different policies: one for task acceptance and one for task rejection. Typically, whenever a new task enters the system, an acceptance test verifies the schedulability of the new task set based on worst-case assumptions. If the task set is found schedulable, the new task is accepted in the ready queue; otherwise, one or more tasks are rejected based on a different policy.

In addition, an algorithm is said to be *competitive* if it has a competitive factor greater than zero.

Notice that the simple guarantee scheme is able to avoid domino effects by sacrificing the execution of the newly arrived task. Basically, the acceptance test acts as a filter that controls the load on the system and always keeps it less than one. Once a task is accepted, the algorithm guarantees that it will complete by its deadline (assuming that no task will exceed its estimated worst-case computation time). Guarantee algorithms, however, do not take task importance into account and, during transient overloads, always reject the newly arrived task, regardless of its value. In certain conditions (such as when tasks have very different importance levels), this scheduling strategy may exhibit poor performance in terms of cumulative value, whereas a robust algorithm can be much more effective.

In guarantee and robust algorithms, a reclaiming mechanism can be used to take advantage of those tasks that complete before their worst-case finishing time. To reclaim the spare time, rejected tasks will not be removed but temporarily parked in a queue, from which they can be possibly recovered whenever a task completes before its worst-case finishing time.

In the following sections we present a few examples of scheduling algorithms for handling overload situations and then compare their performance for different peak load conditions.

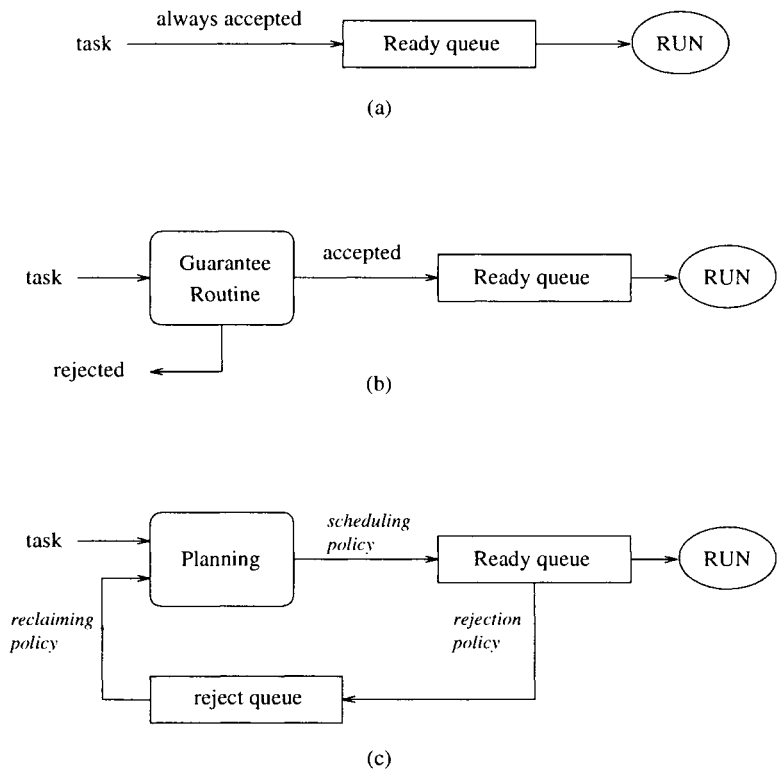

Figure 8.8 Scheduling schemes for handling overload situations. **a.** Best Effort. **b.** Guarantee. **c.** Robust.

8.4.1 The RED algorithm

RED (Robust Earliest Deadline) is a robust scheduling algorithm proposed by Buttazzo and Stankovic [BS93, BS95] for dealing with firm aperiodic tasks in overloaded environments. The algorithm synergistically combines many features including graceful degradation in overloads, deadline tolerance, and resource reclaiming. It operates in normal and overload conditions with excellent performance, and it is able to predict not only deadline misses but also the size of the overload, its duration, and its overall impact on the system.

In RED, each task $J_i(C_i, D_i, M_i, V_i)$ is characterized by four parameters: a worst-case execution time (C_i), a relative deadline (D_i), a deadline tolerance (M_i), and an importance value (V_i). The deadline tolerance is the amount of time by which a task is permitted to be late; that is, the amount of time that a task may execute after its deadline and still produce a valid result. This parameter can be useful in many real applications, such as robotics and multimedia systems, where the deadline timing semantics is more flexible than scheduling theory generally permits.

Deadline tolerances also provide a sort of compensation for the pessimistic evaluation of the worst-case execution time. For example, without tolerance, we could find that a task set is not feasibly schedulable and hence decide to reject a task. But, in reality, the system could have been scheduled within the tolerance levels. Another positive effect of tolerance is that various tasks could actually finish before their worst-case times, so a resource reclaiming mechanism could then compensate, and the tasks with tolerance could actually finish on time.

In RED, the primary deadline plus the deadline tolerance provides a sort of secondary deadline, used to run the acceptance test in overload conditions. Notice that having a tolerance greater than zero is different than having a longer deadline. In fact, tasks are scheduled based on their primary deadline but accepted based on their secondary deadline. In this framework, a schedule is said to be *strictly feasible* if all tasks complete before their primary deadline, whereas is said to be *tolerant* if there exists some task that executes after its primary deadline but completes within its secondary deadline.

The guarantee test performed in RED is formulated in terms of residual laxity. The residual laxity L_i of a task is defined as the interval between its estimated finishing time (f_i) and its primary (absolute) deadline (d_i). Each residual laxity can be efficiently computed using the result of the following lemma.

Lemma 8.1 *Given a set $J = \{J_1, J_2, \ldots, J_n\}$ of active aperiodic tasks ordered by increasing primary (absolute) deadline, the residual laxity L_i of each task J_i at time t can be computed as*

$$L_i = L_{i-1} + (d_i - d_{i-1}) - c_i(t), \tag{8.8}$$

where $L_0 = 0$, $d_0 = t$ (that is, the current time), and $c_i(t)$ is the remaining worst-case computation time of task J_i at time t.

Proof. By definition, a residual laxity is $L_i = d_i - f_i$. Since tasks in the set J are ordered by increasing deadlines, task J_1 is executing at time t, and its estimated finishing time is given by the current time plus its remaining execution time ($f_1 = t + c_1$). As a consequence, L_1 is given by

$$L_1 = d_1 - f_1 = d_1 - t - c_1.$$

Any other task J_i, with $i > 1$, will start as soon as J_{i-1} completes and will finish c_i units of time after its start ($f_i = f_{i-1} + c_i$). Hence, we have

$$\begin{aligned} L_i &= d_i - f_i = d_i - f_{i-1} - c_i = d_i - (d_{i-1} - L_{i-1}) - c_i = \\ &= L_{i-1} + (d_i - d_{i-1}) - c_i, \end{aligned}$$

and the lemma follows. □

Notice that if the current task set J is schedulable and a new task J_a arrives at time t, the feasibility test for the new task set $J' = J \cup \{J_a\}$ requires to compute only the residual laxity of task J_a and that one of those tasks J_i such that $d_i > d_a$. This is because the execution of J_a does not influence those tasks having deadline less than or equal to d_a, which are scheduled before J_a. It follows that, the acceptance test has $O(n)$ complexity in the worst case.

To simplify the description of the RED guarantee test, we define the *Exceeding time* E_i as the time that task J_i executes after its secondary deadline:[1]

$$E_i = \max(0, -(L_i + M_i)). \tag{8.9}$$

We also define the *Maximum Exceeding Time* E_{max} as the maximum among all E_i's in the tasks set; that is, $E_{max} = \max_i(E_i)$. Clearly, a schedule will be strictly feasible if and only if $L_i \geq 0$ for all tasks in the set, whereas it will be tolerant if and only if there exists some $L_i < 0$, but $E_{max} = 0$.

[1] If $M_i = 0$, the *Exceeding Time* is also called the *Tardiness*.

By this approach we can identify which tasks will miss their deadlines and compute the amount of processing time required above the capacity of the system – the maximum exceeding time. This global view allows to plan an action to recover from the overload condition. Many recovering strategies can be used to solve this problem. The simplest one is to reject the least-value task that can remove the overload situation. In general, we assume that, whenever an overload is detected, some rejection policy will search for a subset J^* of least-value tasks that will be rejected to maximize the cumulative value of the remaining subset. The RED acceptance test is shown in Figure 8.9.

```
RED_acceptance_test(J, J_new) {

    E = 0;                      /* Maximum Exceeding Time */
    L_0 = 0;
    d_0 = current_time();

    J' = J ∪ {J_new};
    k = <position of J_new in the task set J'>;

    for each task J'_i such that i ≥ k do {
        /* compute the maximum exceeding time */
        L_i = L_{i-1} + (d_i − d_{i-1}) − c_i;
        if (L_i + M_i < −E) then E = −(L_i + M_i);
    }

    if (E > 0) {
        <select a set J* of least-value tasks to be rejected>;
        <reject all task in J*>;
    }
}
```

Figure 8.9 The RED acceptance test.

In RED, a resource reclaiming mechanism is used to take advantage of those tasks that complete before their worst-case finishing time. To reclaim the spare time, rejected tasks are not removed forever but temporarily parked in a queue, called *Reject Queue*, ordered by decreasing values. Whenever a running task

completes its execution before its worst-case finishing time, the algorithm tries to reaccept the highest-value tasks in the Reject Queue having positive laxity. Tasks with negative laxity are removed from the system.

8.4.2 D_{over}: a competitive algorithm

Koren and Shasha [KS92] found an on-line scheduling algorithm, called D^{over}, which has been proved to be optimal, in the sense that it gives the best competitive factor achievable by any on-line algorithm (that is, 0.25).

As long as no overload is detected, D^{over} behaves like EDF. An overload is detected when a ready task reaches its *Latest Start Time (LST)*; that is, the time at which the task's remaining computation time is equal to the time remaining until its deadline. At this time, some task must be abandoned: either the task that reached its LST or some other task. In D_{over}, the set of ready tasks is partitioned in two disjoint sets: *privileged* tasks and *waiting* tasks. Whenever a task is preempted it becomes a *privileged* task. However, whenever some task is scheduled as the result of a LST, all the ready tasks (whether preempted or never executed) become *waiting* tasks.

When an overload is detected because a task J_z reaches its LST, then the value of J_z is compared against the total value V_{priv} of all the privileged tasks (including the value v_{curr} of the currently running task). If

$$v_z > (1 + \sqrt{k})(v_{curr} + V_{priv})$$

(where k is ratio of the highest value density and the lowest value density task in the system), then J_z is executed; otherwise, it is abandoned. If J_z is executed, all the privileged tasks become waiting tasks. Task J_z can in turn be abandoned in favor of another task J_x that reaches its LST, but only if $v_x > (1 + \sqrt{k})v_z$.

It worth to observe that having the best competitive factor among all on-line algorithms does not mean having the best performance in *any* load condition. In fact, in order to guarantee the best competitive factor, D^{over} may reject tasks with values higher than the current task but not higher than the threshold that guarantees optimality. In other words, to cope with worst-case sequences, D^{over} does not take advantage of lucky sequences and may reject more value than it is necessary. In Section 8.5, the performance of D_{over} is tested for random task sets and compared with the one of other scheduling algorithms.

8.5 PERFORMANCE EVALUATION

In this section, the performance of the scheduling algorithms described above is tested through simulation using a synthetic workload. Each plot on the graphs represents the average of a set of 100 independent simulations, the duration of each is chosen to be 300,000 time units long. The algorithms are executed on task sets consisting of 100 aperiodic tasks, whose parameters are generated as follows. The worst-case execution time C_i is chosen as a random variable with uniform distribution between 50 and 350 time units. The interarrival time T_i is modeled as a random variable with a Poisson distribution with average value equal to $T_i = NC_i/\rho$, where N is the total number of tasks and ρ is the average load. The laxity of a task is computed as a random value with uniform distribution from 150 and 1850 time units, and the relative deadline is computed as the sum of its worst-case execution time and its laxity. The task value is generated as a random variable with uniform distribution ranging from 150 to 1850 time units, as for the laxity.

The first experiment illustrates the effectiveness of the guarantee and robust scheduling paradigm with respect to the best-effort scheme, under the EDF priority assignment. In particular, it shows how the pessimistic assumptions made in the guarantee test affect the performance of the algorithms and how much a reclaiming mechanism can compensate for this degradation. In order to test these effects, tasks were generated with actual execution times less than their worst-case values. The specific parameter varied in the simulations was the average *Unused Computation Time Ratio*, defined as

$$\beta = 1 - \frac{\text{Actual Computation Time}}{\text{Worst-Case Computation Time}}.$$

Note that, if ρ_n is the *nominal* load estimated based on the worst-case computation times, the *actual* load ρ is given by

$$\rho = \rho_n(1 - \beta).$$

In the graphs reported in Figure 8.10, the task set was generated with a nominal load $\rho_n = 3$, while β was varied from 0.125 to 0.875. As a consequence, the actual mean load changed from a value of 2.635 to a value of 0.375, thus ranging over very different actual load conditions. The performance was measured by computing the *Hit Value Ratio (HVR)*; that is, the ratio of the cumulative value achieved by an algorithm and the total value of the task set. Hence, $HVR = 1$ means that all the tasks completed within their deadlines and no tasks were rejected.

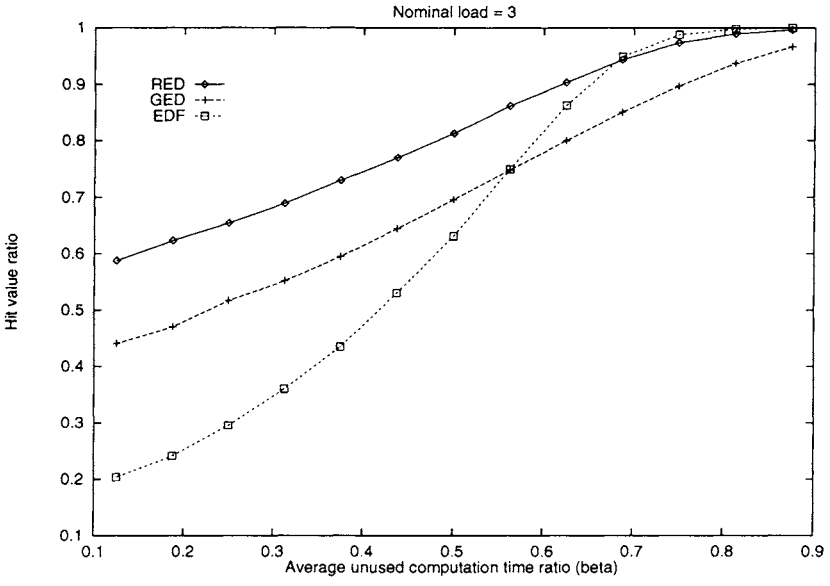

Figure 8.10 Performance of various EDF scheduling schemes: best-effort (EDF), guarantee (GED) and robust (RED).

For small values of β, that is, when tasks execute for almost their maximum computation time, the guarantee (GED) and robust (RED) versions are able to obtain a significant improvement compared to the plain EDF scheme. Increasing the unused computation time, however, the actual load falls down and the plain EDF performs better and better, reaching the optimality in underload conditions. Notice that as the system becomes underloaded ($\beta \simeq 0.7$) GED becomes less effective than EDF. This is due to the fact that GED performs a worst-case analysis, thus rejecting tasks that still have some chance to execute within their deadline. This phenomenon does not appear on RED, because the reclaiming mechanism implemented in the robust scheme is able to recover the rejected tasks whenever possible.

In the second experiment, D_{over} is compared against two robust algorithms: RED (Robust Earliest Deadline) and RHD (Robust High Density). In RHD, the task with the highest value density (v_i/C_i) is scheduled first, regardless of its deadline. Performance results are shown in Figure 8.11.

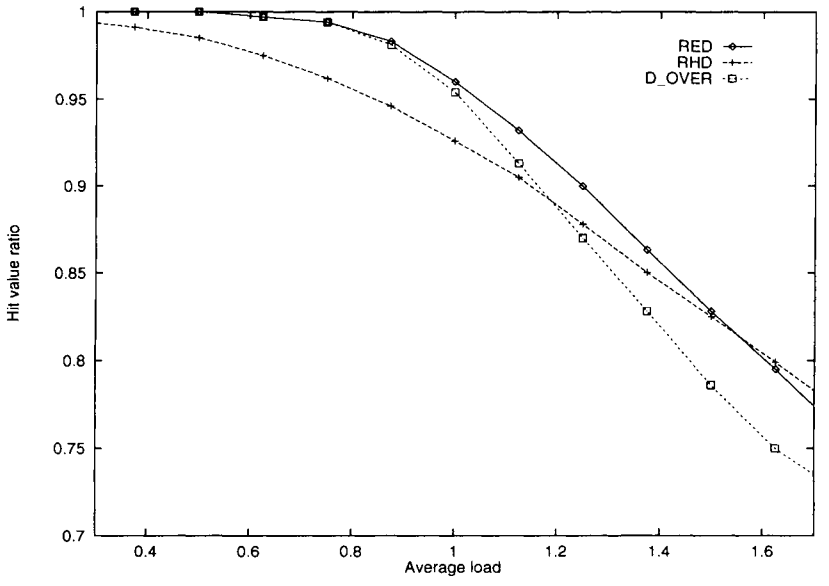

Figure 8.11 Performance of D_{over} against RED and RHD.

Notice that in underload conditions D_{over} and RED exhibit optimal behavior ($HVR = 1$), whereas RHD is not able to achieve the total cumulative value, since it does not take deadlines into account. However, for high load conditions ($\rho > 1.5$), RHD performs even better than RED and D_{over}.

In particular, for random task sets, D_{over} is less effective than RED and RHD for two reasons: first, it does not have a reclaiming mechanism for recovering rejected tasks in the case of early completions; second, the threshold value used in the rejection policy is set to reach the best competitive factor in a worst-case scenario. But this means that for random sequences D_{over} may reject tasks that could increase the cumulative value, if executed.

In conclusion, we can say that in overload conditions no on-line algorithm can achieve optimal performance in terms of cumulative value. Competitive algorithms are designed to guarantee a *minimum* performance in any load condition, but they cannot guarantee the best performance for all possible scenarios. For random task sets, robust scheduling schemes appear to be more appropriate.