9 HANDLING OVERLOAD CONDITIONS

9.1 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 processor capacity, and hence not all tasks can complete within their deadlines.

Overload conditions can occur for different causes, including the following:

- Bad system design. If a system is not designed or analyzed under pessimistic assumptions and worst-case load scenarios, it may work for most typical situations, but it can collapse in particular peak-load conditions, where too much computation is requested for the available computational resources.
- Simultaneous arrival of events. Even if the system is properly designed, the simultaneous arrival of several "unexpected" events could increase the load over the tolerated threshold.
- Malfunctioning of input devices. Sometimes hardware defects in the acquisition boards or in some sensors could generate anomalous sequences of interrupts, saturating the processor bandwidth or delaying the application tasks after their deadlines.
- Unpredicted variations of the environmental conditions could generate a computational demand higher than that manageable by the processor under the specified timing requirements.
- Operating system exceptions. In some cases, anomalous combination of data could raise exceptions in the kernel, triggering the execution of high-priority handling routines that would delay the execution of application tasks.

9.1.1 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 $\overline{C}$ is the mean service time and λ is the average interarrival rate of the jobs, the average load can be computed as

$$\rho = \lambda \overline{C}.$$

Note 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:

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

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. In general, the load in a given interval $[t_a, t_b]$ can be defined as

$$\rho(t_a, t_b) = \max_{t_1, t_2 \in [t_a, t_b]} \frac{g(t_1, t_2)}{t_2 - t_1} \tag{9.1}$$

where $g(t_1, t_2)$ is the processor demand in the generic interval $[t_1, t_2]$. Such a definition, however, is of little practical use for load calculation, since the number of intervals in which the maximum has to be computed can be very high. Moreover, it is not clear how large the interval $[t_a, t_b]$ should be to estimate the overall system load.

A more practical definition that can be used to estimate the current load in dynamic real-time systems is the *instantaneous load* $\rho(t)$, proposed by Buttazzo and Stankovic [BS95].

According to this method, the load is computed in all intervals from the current time t and each deadline (d_i) of the active jobs. Hence, the intervals that need to be considered for the computation are $[t, d_1]$, $[t, d_2]$, …, $[t, d_n]$. In each interval $[t, d_i]$, the partial load $\rho_i(t)$ due to the first i jobs is

$$\rho_i(t) \;=\; \frac{\sum_{d_k \leq d_i} c_k(t)}{(d_i - t)}, \tag{9.2}$$

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 at time t is

$$\rho(t) \;=\; \max_i \rho_i(t). \tag{9.3}$$

Figure 9.1 shows an example of load calculation, at time $t = 3$, for a set of three real-time jobs. Then, Figure 9.2 shows how the load varies as a function of time for the same set of jobs.

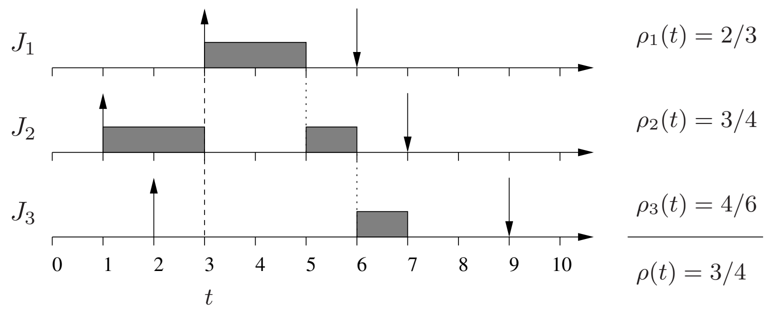


Figure 9.1 Instantaneous load at time $t = 3$ for a set of three real-time jobs.

9.1.2 TERMINOLOGY

When dealing with computational load, it is important to distinguish between *overload* and *overrun*.

Definition 9.1 *A computing system is said to experience an* ***overload*** *when the computation time demanded by the task set in a certain interval of time exceeds the available processing time in the same interval.*

Definition 9.2 *A task (or a job) is said to experience an* ***overrun*** *when exceeding its expected utilization. An overrun may occur either because the next job is activated before its expected arrival time (*activation overrun*), or because the job computation time exceeds its expected value (*execution overrun*).*

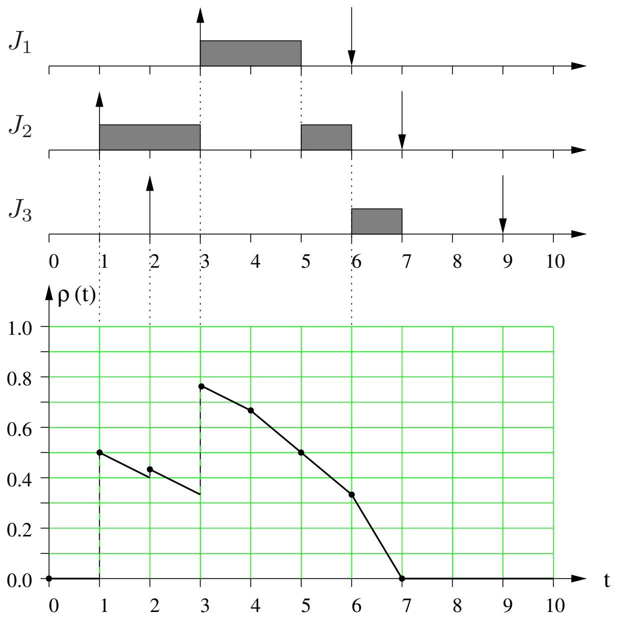


Figure 9.2 Instantaneous load as a function of time for a set of three real-time jobs.

Note that while the overload is a condition related to the processor, the overrun is a condition related to a task (or a single job). A task overrun does not necessarily cause an overload. However, a large unexpected overrun or a sequence of overruns can cause very unpredictable effects on the system, if not properly handled. In the following, we distinguish between two types of overload conditions:

- **Transient overload**: it is an overload condition occurring for a limited duration, in a system in which the average load is less than or equal to one ($\overline{\rho} \leq 1$), but the maximum load is greater than one ($\rho^{max} > 1$).
- **Permanent overload**: it is an overload condition occurring for an unpredictable duration, in a system in which the average load is higher than one ($\overline{\rho} > 1$).

In a real-time computing system, a transient overload can be caused by a sequence of overruns, or by a bursty arrival of aperiodic requests, whereas a permanent overload condition typically occurs in periodic task systems when the total processor utilization exceeds one.

In the rest of this chapter, the following types of overload conditions will be analyzed:

- **Transient overloads due to aperiodic jobs**. This type of overload is typical of event-triggered systems consisting of many aperiodic jobs activated by external events. If the operating system is not designed to cope with excessive event arrivals, the effects of an overload can be unpredictable and cause serious problems on the controlled system. Experiments carried out by Locke [Loc86] have shown that EDF can rapidly degrade its performance during overload intervals, and 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 9.3. Figure 9.3a 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 9.3b).

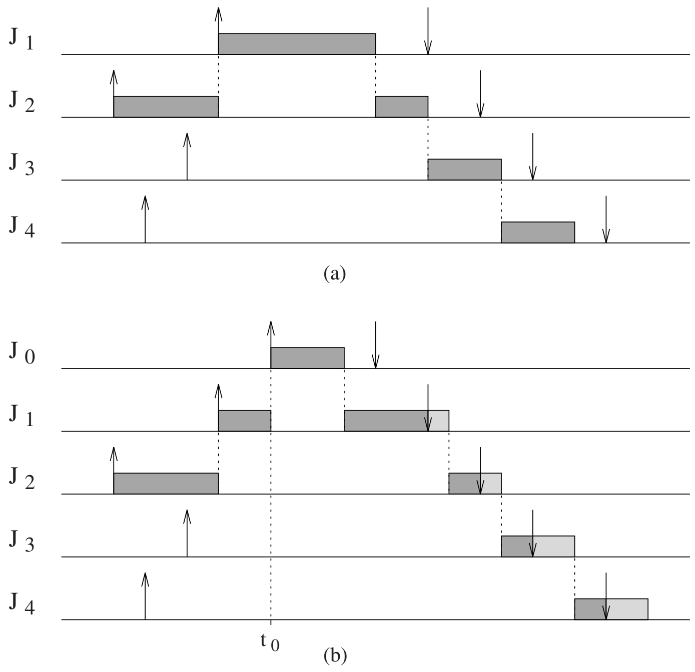


Figure 9.3 Feasible schedule with Earliest Deadline First, in normal load condition (a). Overload with Domino Effect due to the arrival of task J_0 (b).

- **Transient overloads due to task overruns**. This type of overload can occur both in event-triggered and time-triggered systems, and it is due to periodic or aperiodic tasks that sporadically execute (or are activated) more than expected. Under Rate Monotonic, an overrun in a task τ_i does not affect tasks with higher priority, but any of the lower priority tasks could miss their deadline. Under EDF, a task overrun can potentially affect all the other tasks in the system. Figure 9.4 shows an example of execution overrun in an EDF schedule.

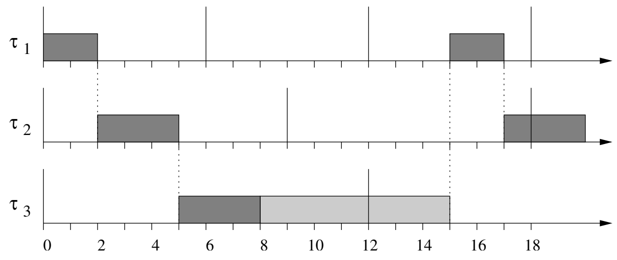


Figure 9.4 Effect of an execution overrun in an EDF schedule.

- **Permanent overloads in periodic task systems**. This type of overload occurs when the total utilization factor of the periodic task set is greater than one. This can happen either because the execution requirement of the task set was not correctly estimated, or some unexpected activation of new periodic tasks, or some of the current tasks increased their activation rate to react to some change in the environment. In such a situation, computational activities start accumulating in the system's queues (which tend to become longer and longer, if the overload persists), and tasks response times tend to increase indefinitely. Figure 9.5 shows the effect of a permanent overload condition in a Rate Monotonic schedule, where τ_2 misses its deadline and τ_3 can never execute.

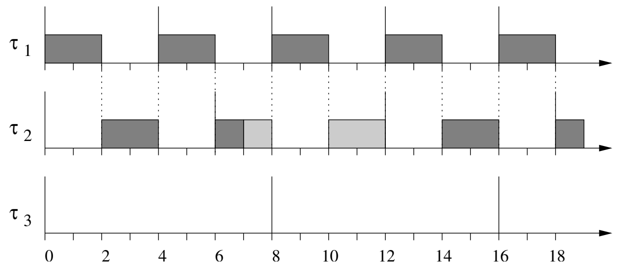


Figure 9.5 Example of a permanent overload under Rate Monotonic: τ_2 misses its deadline and τ_3 can never execute.

9.2 HANDLING APERIODIC OVERLOADS

In this section we consider event-driven systems where tasks arrive dynamically at unknown time instants. Each task consists of a single job, which is characterized by a fixed (known) computation time C_i and a relative deadline D_i. As a consequence, the overload in these systems can only be caused by the excessive number of tasks and can be detected at task activation times.

9.2.1 PERFORMANCE METRICS

When tasks are 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. For example, 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 completes 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

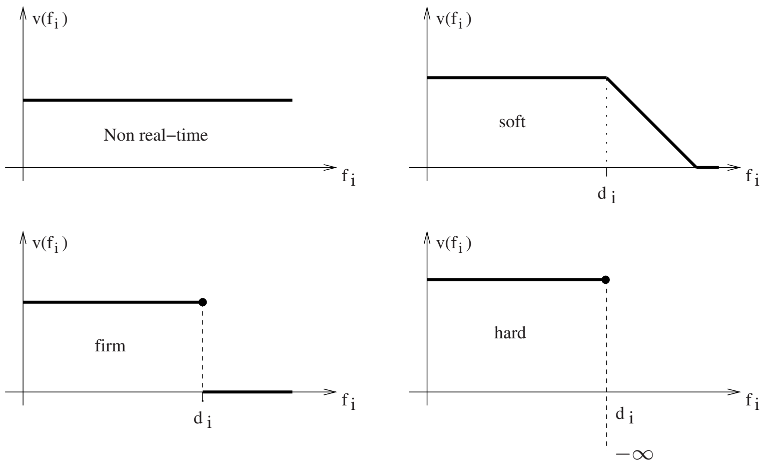


Figure 9.6 Utility functions that can be associated to a task to describe its importance.

in many situations. A soft task can still give a value to the system if competed after its deadline, although this value may decrease with time. There are also real-time activities, so-called *firm*, that do not jeopardize the system, but give a negligible contribution if completed after their deadline. Figure 9.6 illustrates the utility functions of four different types of tasks.

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.

Note 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 (including 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 normal load 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 its relative deadline, and V_i 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^*.

9.2.2 ON-LINE VERSUS CLAIRVOYANT SCHEDULING

Since dynamic environments require online scheduling, it is important to analyze the properties and the performance of online scheduling algorithms in overload conditions. Although there exist optimal online algorithms in normal load conditions, it is easy to show that no optimal on-line algorithms exist in overloads situations. Consider for example the task set shown in Figure 9.7, 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 9.7a). 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 9.7b). Note that if J_3 arrives at time $t = 9$ or later (see Figure 9.7c), 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 J_3, no online 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 online 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 online scheduling algorithms in overload conditions.

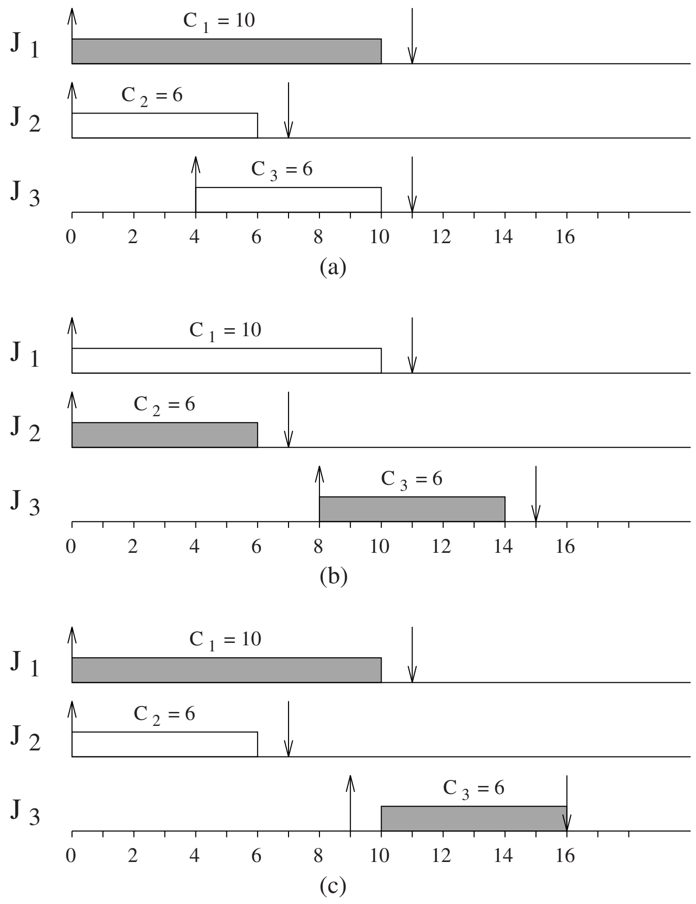


Figure 9.7 No optimal online algorithms exist in overload conditions, since the schedule that maximizes Γ depends on the knowledge of future arrivals: $\Gamma_{max} = 10$ in case (a), $\Gamma_{max} = 12$ in case (b), and $\Gamma_{max} = 16$ in case (c).

9.2.3 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. [BKM+92].

Definition 9.3 *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 note 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 online 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 result it is sufficient to find an overload situation in which the cumulative value obtained by EDF can be arbitrarily small with respect to that one achieved by the clairvoyant scheduler. Consider the example shown in Figure 9.8, where tasks have a value proportional to their computation time. This is an overload condition because both tasks cannot be completed within their deadlines.

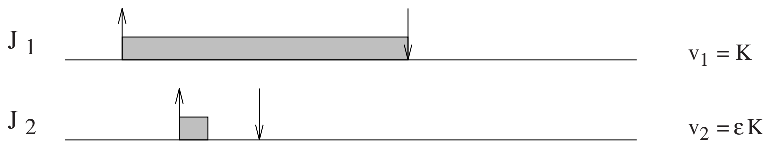


Figure 9.8 Situation in which EDF has an arbitrarily small competitive factor.

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 by Baruah et al. [BKM$^+$92] is that there is an upper bound on the competitive factor of any on-line algorithm. This is stated by the following theorem.

Theorem 9.1 (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 online 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 online algorithm.

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 9.9):

- 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 9.9). 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.

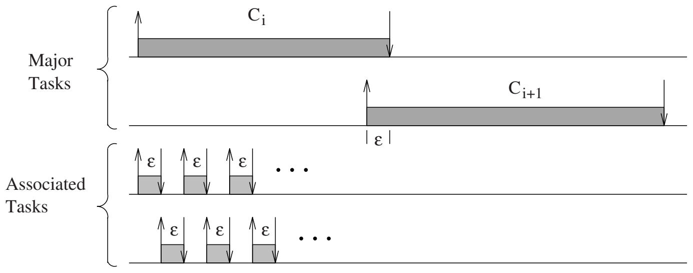


Figure 9.9 Task sequence generated by the adversary.

Note 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 because 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 $(C_0 + C_1 - \epsilon)$. Note 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:

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

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

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

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

$$\frac{C_m}{\sum_{j=0}^{m} C_j} \leq \frac{1}{k}, \tag{9.4}$$

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. Note 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 that satisfies Equation (9.4), it also satisfies the following equation:

$$C_m \leq C_{m-1}. \tag{9.5}$$

Thus, (9.5) is satisfied if and only if (9.4) is satisfied.

From (9.4) 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 (9.4) is equivalent to

$$\begin{cases} C_0 & = & 1 \\ C_1 & = & k-1 \\ C_{i+2} & = & k(C_{i+1}-C_i). \end{cases} \tag{9.6}$$

From this result, we can say that the tightest bound on the competitive factor of an online algorithm is given by the smallest ratio $1/k$ (equivalently, the largest k) such that (9.6) satisfies (9.5). Equation (9.6) is a recurrence relation that can be solved by standard techniques [Sha85]. The characteristic equation of (9.6) is

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

which has roots

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

When $k = 4$, we have

$$C_i \;=\; d_1 i 2^i + d_2 2^i, \tag{9.7}$$

and when $k \neq 4$ we have

$$C_i \;=\; d_1(x_1)^i + d_2(x_2)^i, \tag{9.8}$$

where values for d_1 and d_2 can be found from the boundary conditions expressed in (9.6). We now show that for ($k = 4$) and ($k > 4$) C_i will diverge, so Equation (9.5) will not be satisfied, whereas for ($k < 4$) C_i will satisfy (9.5).

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 (9.5) 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 (9.5) 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:

$$\begin{cases} d_1 &= se^{j\theta} \\ d_2 &= se^{-j\theta} \end{cases} \qquad \begin{cases} x_1 &= re^{j\omega} \\ x_2 &= re^{-j\omega}, \end{cases}$$

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 (9.8) 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 (9.5).

Since (9.5) is satisfied for $k < 4$, the largest k that determines the competitive factor of an online 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 online scheduling algorithm in an overloaded environment. Hence, Theorem 9.1 follows.

EXTENSIONS

Theorem 9.1 establishes an upper bound on the competitive factor of online 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 9.2 (Baruah et al.) *In systems with a loading factor ρ, $1 < \rho \leq 2$, and task values equal to computation times, no online algorithm can guarantee a competitive factor greater than p, where p satisfies*

$$4[1 - (\rho - 1)p]^3 = 27p^2. \tag{9.9}$$

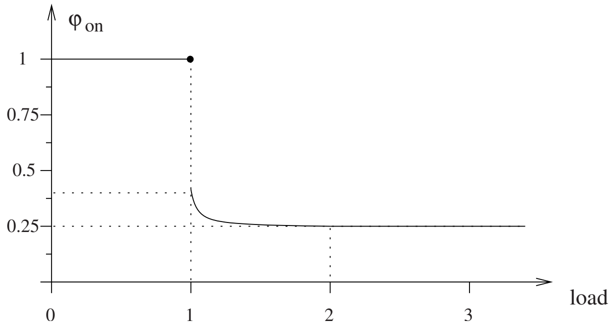


Figure 9.10 Bound of the competitive factor of an on-line scheduling algorithm as a function of the load.

Note that for $\rho = 1 + \epsilon$, Equation (9.9) 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 9.10.

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 online 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 online algorithm can achieve a competitive factor greater than

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

whereas, if $q < 1$, no online 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 0.25 bound has only a theoretical validity, and more work is needed to derive other bounds based on more realistic assumptions on the actual load conditions. An analysis of online scheduling algorithms under different types of adversaries has been presented by Karp [Kar92].

9.2.4 TYPICAL SCHEDULING SCHEMES

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 9.11:

- **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 that takes task values into account.

- **With acceptance test**. This class includes those algorithms with admission control, performing a guarantee test at every job activation. 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 system; 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, aimed at maximizing the cumulative value of the feasible tasks.

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

Note 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). The acceptance test, however, does not take task importance into account and, during transient overloads, always rejects 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.

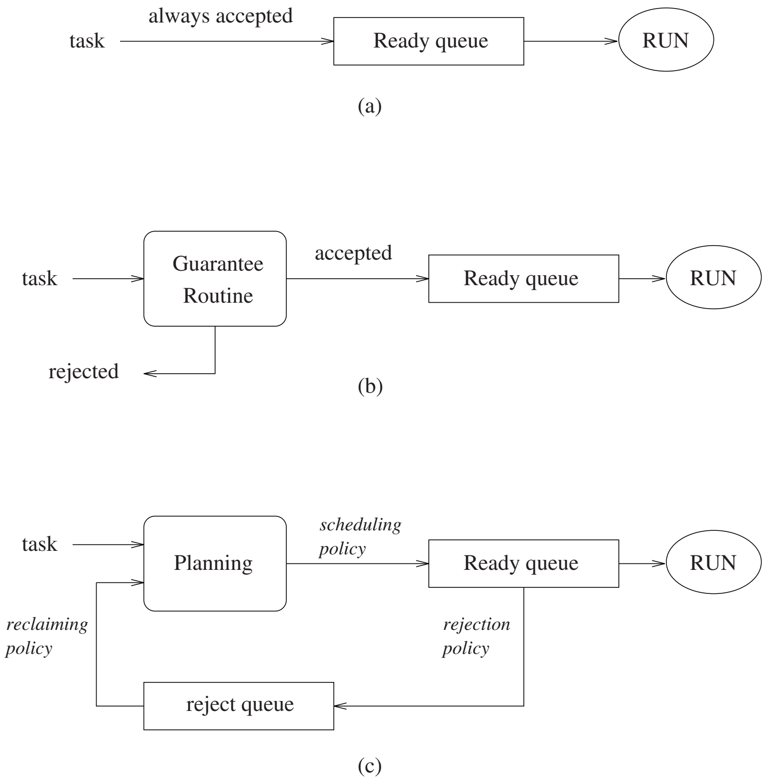


Figure 9.11 Scheduling schemes for handling overload situations: best effort (a), with acceptance test (b), and robust (c).

When the load is controlled by job rejection, 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 real-time literature, several scheduling algorithms have been proposed to deal with transient overloads in event triggered systems. Ramamritham and Stankovic [RS84] used EDF to dynamically guarantee incoming work via on-line planning. Locke [Loc86] proposed a best effort algorithm using EDF with a rejection policy based on tasks value density. Biyabani et. al. [BSR88] extended the work of Ramamritham and Stankovic 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. Haritsa, Livny, and Carey [HLC91] presented the use of a feedback-based EDF algorithm for real-time database systems.

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 results on overload in real-time systems were also derived. 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. Schwan and Zhou [SZ92] did online 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 by Haritsa et al. [HLC91] (based on simulation).

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

9.2.5 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, a task could be rejected, although the system could be scheduled within the tolerance levels.

In RED, the primary deadline plus the deadline tolerance provides a sort of secondary deadline, used to run the acceptance test in overload conditions. Note 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 9.1 *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*

$$L_i = L_{i-1} + (d_i - d_{i-1}) - c_i(t), \tag{9.10}$$

where $L_0 = 0$, $d_0 = t$ (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 are ordered by increasing deadlines, 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. □

Note 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]

$$E_i = \max(0, -(L_i + M_i)). \tag{9.11}$$

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 $E_{max} = 0$.

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 planning 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 9.12.

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

```
Algorithm: RED Acceptance Test
Input: A task set $\mathcal{J}$ with $\{C_i, D_i, V_i, M_i\}, \forall J_i \in \mathcal{J}$
Output: A schedulable task set
// Assumes deadlines are ordered by decreasing values

(1)  begin
(2)      $E = 0$;                // Maximum Exceeding Time
(3)      $L_0 = 0$;
(4)      $d_0 = current\_time()$;
(5)      $J' = J \cup \{J_{new}\}$;
(6)      $k$ = <position of $J_{new}$ in the task set $J'$>;
(7)      for (each task $J'_i$ such that $i \geq k$) do
(8)          $L_i = L_{i-1} + (d_i - d_{i-1}) - c_i$;
(9)          if $(L_i + M_i \ < \ -E)$ then          // compute $E_{max}$
(10)             $E = -(L_i + M_i)$;
(11)         end
(12)     end
(13)     if $(E > 0)$ then
(14)         <select a set $J^*$ of least-value tasks to be rejected>;
(15)         <reject all task in $J^*$>;
(16)     end
(17) end
```

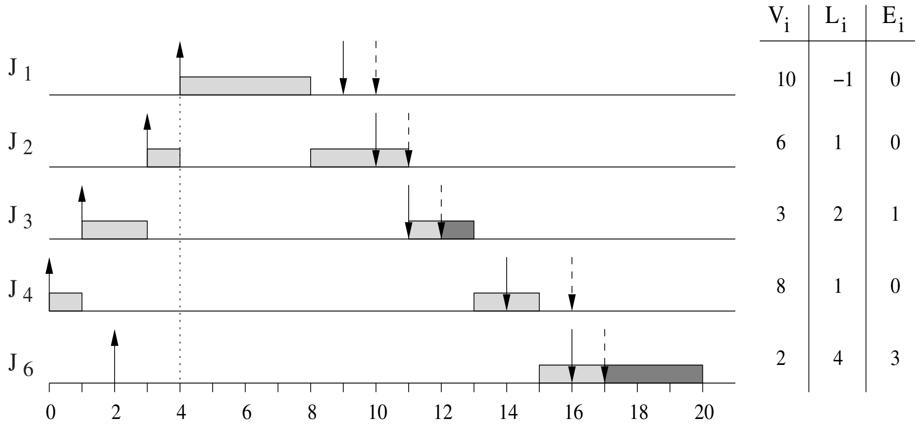
Figure 9.12 The RED acceptance test.

A simple rejection strategy consists in removing the task with the smallest value that resolves the overload. To quickly identify the task to be rejected, we can keep track of the *First Exceeding Task*, denoted as J_{FET}, which is the task with the earliest primary deadline that misses its secondary deadline. The FET index can be easily determined within the *for* loop in which residual each laxity is computed. Note that in order to resolve the overload, the task to be rejected must have a residual computation time greater than or equal to the maximum exceeding time and a primary deadline less than d_{FET}. Hence, the rejection strategy can be expressed as follows:

Reject the task J_r with the least value, such that

$$(r \leq \text{FET}) \ \text{ and } \ (c_r(t) \geq E_{max})$$

To better understand the rejection strategy, consider the example illustrated in Figure 9.13, where secondary deadlines are drawn with dashed arrows. As can be easily verified, before the arrival of J_1 the task set $\{J_2, J_3, J_4, J_5\}$ is strictly feasible.



	V_i	L_i	E_i
J 1	10	−1	0
J 2	6	1	0
J 3	3	2	1
J 4	8	1	0
J 6	2	4	3

Figure 9.13 Example of overload in a task set with deadline tolerances.

At time $t = 4$, when J_1 arrives, an overload occurs because both J_3 and J_5 would terminate after their secondary deadline. The least value task able to resolve the overload is J_2. In fact, J_5, that has the smallest value, cannot be rejected because, having a long primary deadline, it would not advance the execution of J_3. Also, rejecting J_3 would not solve the overload, since its residual computation time is not sufficient to advance the completion of J_5 before the deadline.

A more efficient rejection strategy could consider rejecting more than one task to minimize the cumulative value of the rejected tasks. For example, rejecting J_3 and J_5 is better than rejecting J_2. However, minimizing the value of the rejected tasks requires a combinatorial search that would be too expensive to be performed online for large task sets.

To take advantage of early completions and reduce the pessimism of the acceptance test, some algorithms use an online reclaiming mechanism that exploits the saved time to possibly recover previously rejected tasks. For example, in RED, a rejected task is not removed from the system, but it is temporarily parked in a *Reject Queue*, with the hope that it can be recovered due to some early completion. If δ is the time saved by the running task, then all the residual laxities will increase by δ, and some of the rejected tasks may be recovered based on their value.

9.2.6 D_{OVER}: A COMPETITIVE ALGORITHM

Koren and Shasha [KS92] found an online 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 online 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 is worth observing that having the best competitive factor among all online 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 9.2.7, the performance of D_{over} is tested for random task sets and compared with the one of other scheduling algorithms.

9.2.7 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 guaranteed (GED) and robust scheduling paradigm (RED) 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 shown in Figure 9.14, 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.

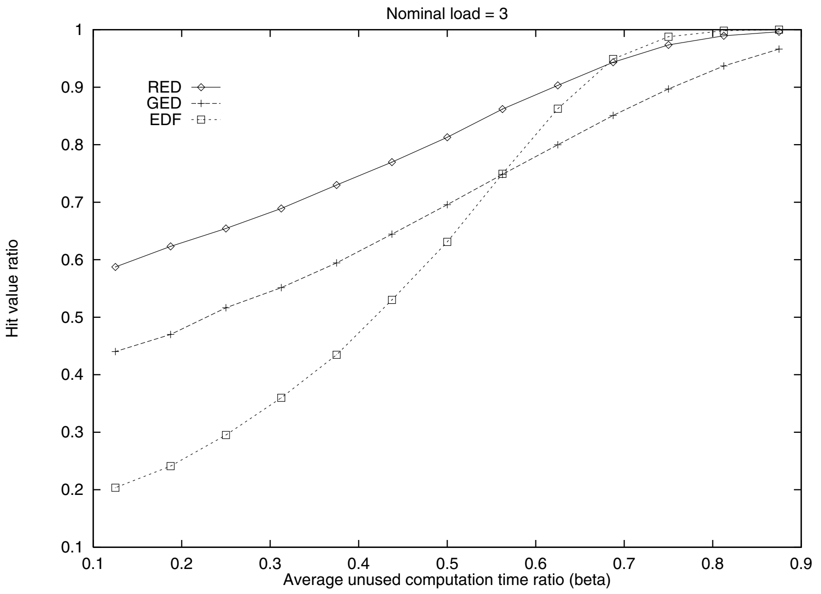


Figure 9.14 Performance of various EDF scheduling schemes: best-effort (EDF), guaranteed (GED) and robust (RED).

For small values of β, that is, when tasks execute for almost their maximum computation time, the guaranteed (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. Note 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 9.15.

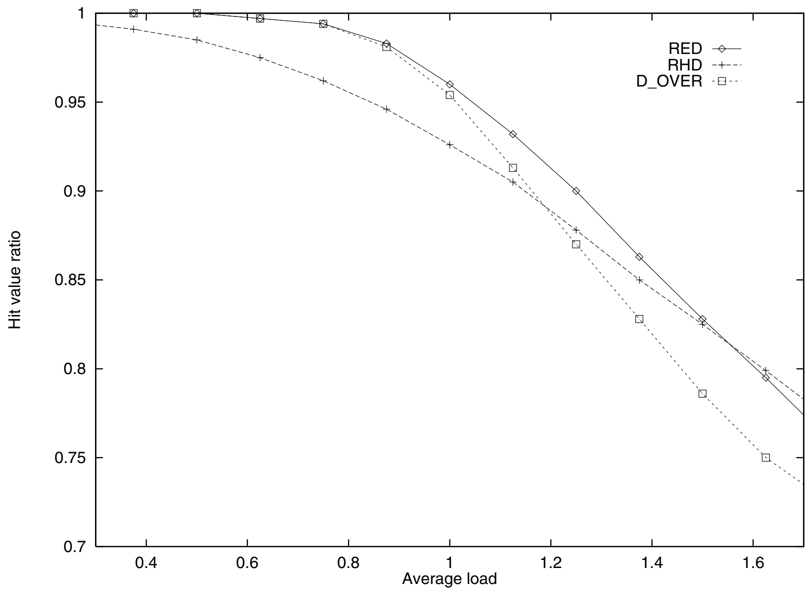


Figure 9.15 Performance of D_{over} against RED and RHD.

Note 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 online 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.

9.3 HANDLING OVERRUNS

This section presents some methodology for handling transient overload conditions caused by tasks that execute more than expected or are activated more frequently than expected. This could happen either because some task parameter was incorrectly estimated, or because the system was intentionally designed under less pessimistic assumptions for achieving a higher average utilization.

If not properly handled, task overruns can cause serious problems in the real-time system, jeopardizing the guarantee performed for the critical tasks and causing an abrupt performance degradation. An example of negative effects of an execution overrun in EDF was already illustrated in Figure 9.4.

To prevent an overrun to introducing unbounded delays on tasks' execution, the system could either decide to abort the current job experiencing the overrun or let it continue with a lower priority. The first solution is not safe, because the job could be in a critical section when aborted, thus leaving a shared resource with inconsistent data (very dangerous). The second solution is much more flexible, since the degree of interference caused by the overrun on the other tasks can be tuned acting on the priority assigned to the "faulty" task for executing the remaining computation. A general technique for implementing such a solution is the resource reservation approach.

9.3.1 RESOURCE RESERVATION

Resource reservation is a general technique used in real-time systems for limiting the effects of overruns in tasks with variable computation times [MST93, MST94b, AB98, AB04]. According to this method, each task is assigned a fraction of the processor bandwidth, just enough to satisfy its timing constraints. The kernel, however, must prevent each task from consuming more than the requested amount to protect the other tasks in the systems (temporal protection). In this way, a task receiving a fraction U_i of the total processor bandwidth behaves as it were executing alone on a slower processor with a speed equal to U_i times the full speed. The advantage of this method is that each task can be guaranteed in isolation, independently of the behavior of the other tasks.

A simple and effective mechanism for implementing resource reservation in a real-time system is to reserve each task τ_i a specified amount of CPU time Q_i in every interval P_i. Such a general approach can also be applied to other resources different than the CPU, but in this context we will mainly focus on the CPU, because CPU scheduling is the topic of this book.

Some authors [RJMO98] tend to distinguish between *hard* and *soft* reservations. According to such a taxonomy, a hard reservation allows the reserved task to execute *at most* for Q_i units of time every P_i, whereas a soft reservation guarantees that the task executes *at least* for Q_i time units every P_i, allowing it to execute more if there is some idle time available.

A resource reservation technique for fixed priority scheduling was first presented by Mercer, Savage, and Tokuda [MST94a]. According to this method, a task τ_i is first assigned a pair (Q_i, P_i) (denoted as a CPU *capacity reserve*) and then it is enabled to execute as a real-time task for Q_i units of time every P_i. When the task consumes its reserved quantum Q_i, it is blocked until the next period, if the reservation is hard, or it is scheduled in background as a non-real-time task, if the reservation is soft. If the task is not finished, it is assigned another time quantum Q_i at the beginning of the next period and it is scheduled as a real-time task until the budget expires, and so on.

In this way, a task is *reshaped* so that it behaves like a periodic real-time task with known parameters (Q_i, P_i) and can be properly scheduled by a classical real-time scheduler.

Although such a method is essential for achieving predictability in the presence of tasks with variable execution times, the overall system's performance becomes quite dependent from a correct resource allocation. For example, if the CPU bandwidth allocated to a task is much less than its average requested value, the task may slow down too much, degrading the system's performance. On the other hand, if the allocated bandwidth is much greater than the actual needs, the system will run with low efficiency, wasting the available resources.

A simple kernel mechanism to enforce temporal protection under EDF scheduling is the Constant Bandwidth Server (CBS) [AB98, AB04], described in Chapter 6. To properly implement temporal protection, however, each task τ_i with variable computation time should be handled by a dedicated CBS with bandwidth U_{s_i}, so that it cannot interfere with the rest of the tasks for more than U_{s_i}. Figure 9.16 illustrates an example in which two tasks (τ_1 and τ_2) are served by two dedicated CBSs with bandwidth $U_{s_1} = 0.15$ and $U_{s_2} = 0.1$, a group of two tasks (τ_3, τ_4) is handled by a single CBS with bandwidth $U_{s_3} = 0.25$, and three hard periodic tasks (τ_5, τ_6, τ_7) are directly scheduled by EDF, without server intercession, since their execution times are not subject to large variations. In this example the total processor bandwidth is shared among the tasks as shown in Figure 9.17.

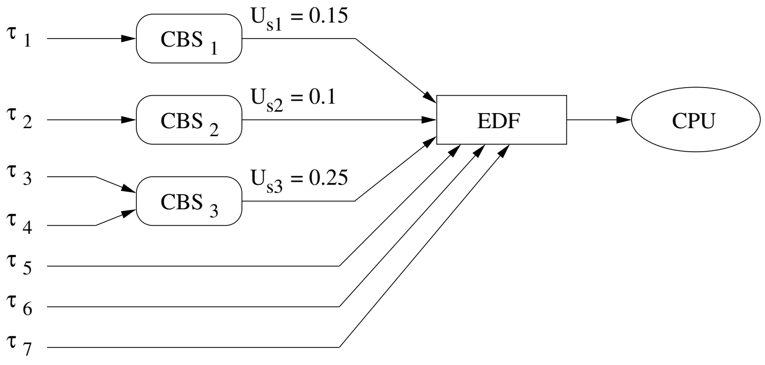


Figure 9.16 Achieving temporal protection using the CBS mechanism.

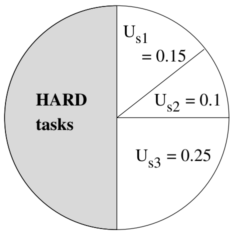


Figure 9.17 Bandwidth allocation for a set of task.

The properties of the CBS guarantee that the set of hard periodic tasks (with utilization U_p) is schedulable by EDF if and only if

$$U_p + U_{s_1} + U_{s_2} + U_{s_3} \leq 1. \tag{9.12}$$

Note that if condition (9.12) holds, the set of hard periodic tasks is always guaranteed to use 50% of the processor, independently of the execution times of the other tasks. Also observe that τ_3 and τ_4 are not isolated with respect to each other (i.e., one can steals processor time from the other), but they cannot interfere with the other tasks for more than one-fourth of the total processor bandwidth.

The CBS version presented in this book is meant for handling soft reservations. In fact, when the budget is exhausted, it is always replenished at its full value and the server deadline is postponed (i.e., the server is always active). As a consequence, a served task can execute more than Q_s in each period P_s, if there are no other tasks in the system. However, the CBS can be easily modified to enforce hard reservations, just by postponing the budget replenishment to the server deadline.

9.3.2 SCHEDULABILITY ANALYSIS

Although a reservation is typically implemented using a server characterized by a budget Q_k and a period T_k, there are cases in which temporal isolation can be achieved by executing tasks in a static partition of disjoint time slots.

To characterize a bandwidth reservation independently on the specific implementation, Mok et al. [MFC01] introduced the concept of *bounded delay partition* that describes a reservation by two parameters: a bandwidth α_k and a delay Δ_k. The bandwidth α_k measures the fraction of resource that is assigned to the served tasks, whereas the delay Δ_k represents the longest interval of time in which the resource is not available. In general, the minimum service provided by a resource can be precisely described by its *supply function* [LB03, SL03], representing the minimum amount of time the resource can provide in a given interval of time.

Definition 9.4 *Given a reservation, the* supply function $Z_k(t)$ *is the minimum amount of time provided by the reservation in every time interval of length* $t \geq 0$.

The supply function can be defined for many kinds of reservations, as static time partitions [MFC01, FM02], periodic servers [LB03, SL03], or periodic servers with arbitrary deadline [EAL07]. Consider, for example, that processing time is provided only in the intervals illustrated in Figure 9.18, with a period of 12 units. In this case, the minimum service occurs when the resource is requested at the beginning of the longest idle interval; hence, the supply function is the one depicted in Figure 9.19.

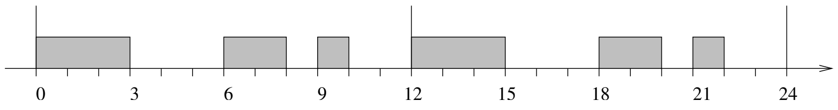


Figure 9.18 A reservation implemented by a static partition of intervals.

For this example we have $\alpha_k = 0.5$ and $\Delta_k = 3$. Once the bandwidth and the delay are computed, the supply function of a resource reservation can be lower bounded by the following *supply bound function*:

$$\mathsf{sbf}_k(t) \overset{\text{def}}{=} \max\{0,\ \alpha_k(t - \Delta_k)\}. \tag{9.13}$$

represented by the dashed line in Figure 9.19. The advantage of using such a lower bound instead of the exact $Z_k(t)$ is that a reservation can be expressed with just two parameters.

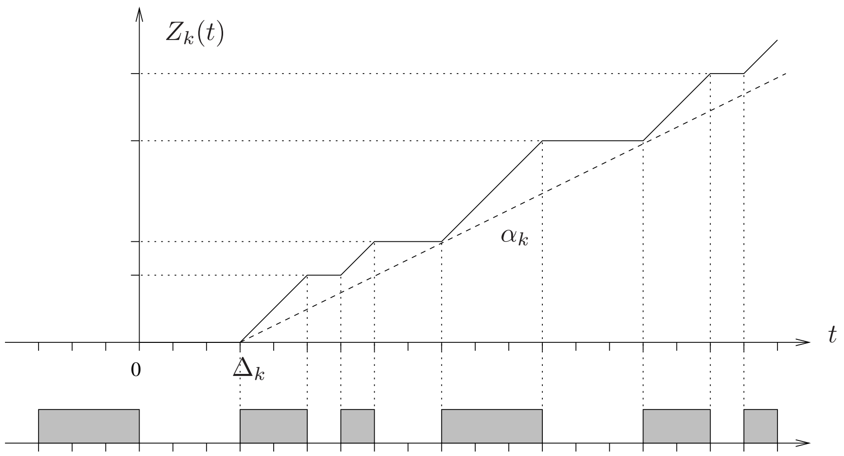


Figure 9.19 A reservation implemented by a static partition of intervals.

In general, for a given supply function $Z_k(t)$, the bandwidth α_k and the delay Δ_k can be formally defined as follows:

$$\alpha_k = \lim_{t\to\infty} \frac{Z_k(t)}{t} \tag{9.14}$$

$$\Delta_k = \sup_{t\geq 0} \left\{ t - \frac{Z_k(t)}{\alpha_k} \right\}. \tag{9.15}$$

If a reservation is implemented using a periodic server with unspecified priority that allocates a budget Q_k every period T_k, then the supply function is the one illustrated in Figure 9.20, where

$$\alpha_k = Q_k/T_k \tag{9.16}$$

$$\Delta_k = 2(T_k - Q_k). \tag{9.17}$$

It is worth observing that reservations with smaller delays are able to serve tasks with shorter deadlines, providing better responsiveness. However, small delays can only be achieved with servers with a small period, condition for which the context switch overhead cannot be neglected. If σ is the runtime overhead due to a context switch (subtracted from the budget every period), then the effective bandwidth of reservation is

$$\alpha_k^{\mathsf{eff}} = \frac{Q-\sigma}{T_k} = \alpha_k \left(1 - \frac{\sigma}{Q_k}\right).$$

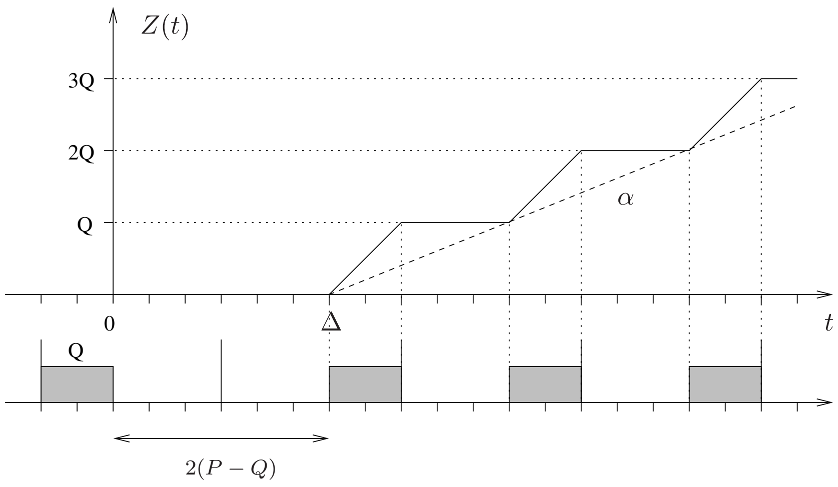


Figure 9.20 A reservation implemented by a static partition of intervals.

Expressing Q_k and T_k as a function of α_k and Δ_k we have

$$\begin{aligned} Q_k &= \frac{\alpha_k \Delta_k}{2(1-\alpha_k)} \\ P_k &= \frac{\Delta_k}{2(1-\alpha_k)}. \end{aligned}$$

Hence,

$$\alpha_k^{\text{eff}} = \alpha_k + \frac{2\sigma(1-\alpha_k)}{\Delta_k}. \tag{9.18}$$

Within a reservation, the schedulability analysis of a task set under fixed priorities can be performed by extending Theorem 4.4 as follows [BBL09]:

Theorem 9.3 *A set of preemptive periodic tasks with relative deadlines less than or equal to periods can be scheduled by a fixed priority algorithm, under a reservation characterized by a supply function $Z_k(t)$, if and only if*

$$\forall i = 1, \ldots, n \quad \exists t \in \mathcal{TS}_i : W_i(t) \leq Z_k(t). \tag{9.19}$$

where $W_i(t)$ is defined by Equation (4.19) *and $\mathcal{TS}_i$ by Equation* (4.21).

Similarly, the schedulability analysis of a task set under EDF can be performed by extending Theorem 4.6 as follows [BBL09]:

Theorem 9.4 *A set of preemptive periodic tasks with relative deadlines less than or equal to periods can be scheduled by EDF, under a reservation characterized by a supply function $Z_k(t)$, if and only if $U < \alpha_k$ and*

$$\forall t > 0 \quad \mathsf{dbf}(t) \leq Z_k(t). \tag{9.20}$$

In the specific case in which $Z_k(t)$ is lower bounded by the supply bound function, the test become only sufficient and the set of testing points can be better restricted as stated in the following theorem [BFB09]:

Theorem 9.5 *A set of preemptive periodic tasks with relative deadlines less than or equal to periods can be scheduled by EDF, under a reservation characterized by a supply function $Z_k(t) = \max[0, \alpha_k(t - \Delta_k)]$, if $U < \alpha_k$ and*

$$\forall t \in \mathcal{D} \quad \mathsf{dbf}(t) \leq \max[0, \alpha_k(t - \Delta_k)]. \tag{9.21}$$

where

$$\mathcal{D} = \{d_k \mid d_k \leq \min[H, \max(D_{max}, L^*)]\}$$

and

$$L^* = \frac{\alpha_k \Delta_k + \sum_{i=1}^{n}(T_i - D_i)U_i}{\alpha_k - U}.$$

9.3.3 HANDLING WRONG RESERVATIONS

As already mentioned, under resource reservations, the system performance heavily depends on a correct bandwidth allocation to the various activities. In fact, if the bandwidth is under allocated, the activities within that reservation will progress more slowly than expected, whereas an over-allocated bandwidth may waste the available resources. This problem can be solved by using capacity sharing mechanisms that can transfer unused budgets to the reservations that need more bandwidth.

Capacity sharing algorithms have been developed both under fixed priority servers [BB02, BBB04] and dynamic priority servers [CBS00]. For example, the CASH algorithm [CBT05] extends CBS to include a slack reclamation. When a server becomes idle with residual budget, the slack is inserted in a queue of spare budgets (CASH queue) ordered by server deadlines. Whenever a new server is scheduled for execution, it first uses any CASH budget whose deadline is less than or equal to its own.

The bandwidth inheritance (BWI) algorithm [LLA01] applies the idea of priority inheritance to CPU resources in CBS, allowing a blocking low-priority process to steal resources from a blocked higher priority process. IRIS [MLBC04] enhances CBS with fairer slack reclaiming, so slack is not reclaimed until all current jobs have been serviced and the processor is idle. BACKSLASH [LB05] is another algorithm that enhances the efficiency of the reclaiming mechanism under EDF.

Wrong reservations can also be handled through feedback scheduling. If the operating system is able to monitor the actual execution time $e_{i,k}$ of each task instance, the actual maximum computation time of a task τ_i can be estimated (in a moving window) as

$$\hat{C}_i = \max_k \{e_{i,k}\}$$

and the actual requested bandwidth as $\hat{U}_i = \hat{C}_i / T_i$. Hence, $\hat{U}_i$ can be used as a reference value in a feedback loop to adapt the reservation bandwidth allocated to the task according to the actual needs. If more reservations are adapted online, we must ensure that the overall allocated bandwidth does not exceed the processor utilization; hence, a form of global feedback adaptation is required to prevent an overload condition. Similar approaches to achieve adaptive reservations have been proposed by Abeni and Buttazzo [AB01] and by Palopoli et al. [PALW02].

9.3.4 RESOURCE SHARING

When critical sections are used by tasks handled within a reservation server, an additional problem occurs when the server budget is exhausted inside a region. In this case, the served task cannot continue the execution to prevent other tasks from missing their deadlines; thus an extra delay is added to the blocked tasks to wait until the next budget replenishment. Figure 9.21 illustrates a situation in which a high priority task τ_1 shares a resource with another task τ_2 handled by a reservation server (e.g., a Sporadic Server) with budget $Q_k = 4$ and period $T_k = 10$. At time $t = 3$, τ_1 preempts τ_2 within its critical section, and at time $t = 4$ it blocks on the locked resource. When τ_2 resumes, however, the residual budget is not sufficient to finish the critical section, and τ_2 must be suspended until the budget will be replenished at time $t = 10$, so introducing an extra delay [5,10] in the execution of τ_1. Two solutions have been proposed in the literature to prevent such an extra delay.

SOLUTION 1: BUDGET CHECK

When a task wants to enter a critical section, the current server budget is checked before granting the access to the resource; if the budget is sufficient, the task enters the critical section, otherwise the access to the resource is postponed until the next budget replenishment.

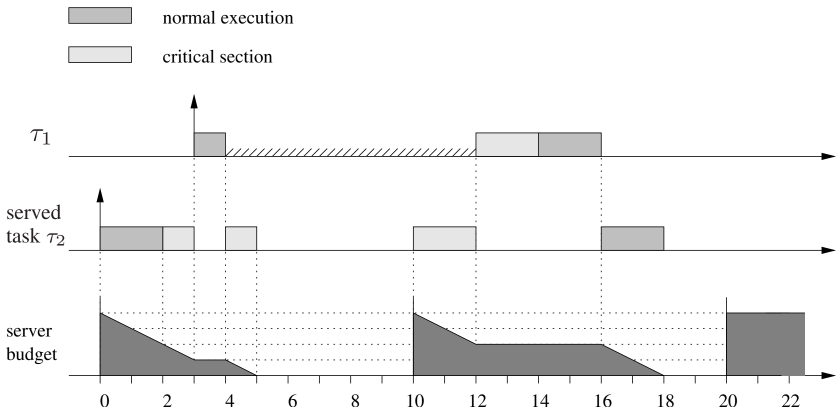


Figure 9.21 Example of extra blocking introduced when the budget is exhausted inside a critical section.

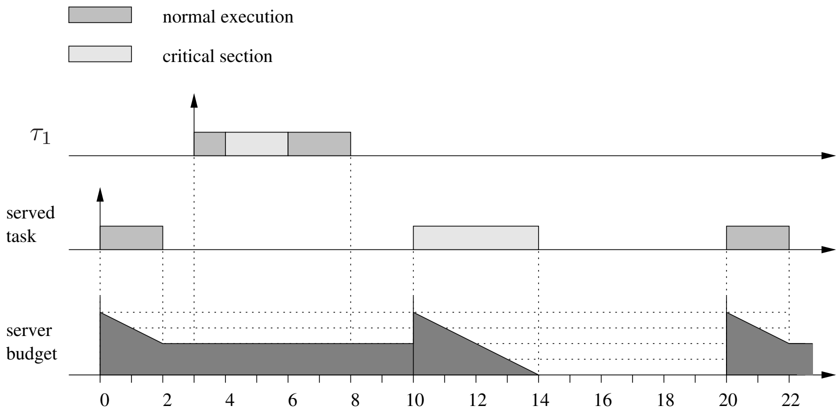


Figure 9.22 Example of budget check to allow resource sharing within reservations.

This mechanism is used in the SIRAP protocol, proposed by Behnam et al. [BSNN07, NSBS09] to share resources among reservations. An example of such a strategy is illustrated in Figure 9.22. In the example, since at time $t = 2$ the budget is $Q_k = 2$ and the critical section is 4 units long, the resource is not granted and τ_2 is suspended until time $t = 10$, when the budget is recharged. In this case, τ_1 is able to execute immediately, while τ_2 experiences a longer delay with respect to the absence of protocol.

SOLUTION 2: BUDGET OVERRUN

The second approach consists in entering a critical section without performing any budget check. When the budget is exhausted inside a resource, the server is allowed to consume some extra budget until the end of the critical section. In this case, the maximum extra budget must be estimated off-line and taken into account in the schedulability analysis. An example of such a strategy is illustrated in Figure 9.23. In this example, at time $t = 5$, when the budget is exhausted inside the resource, τ_2 is allowed to continue the execution until the end of the critical section, consuming 2 extra units of budget. In the worst case, the extra budget to be taken into account is equal to the longest critical section of the served task.

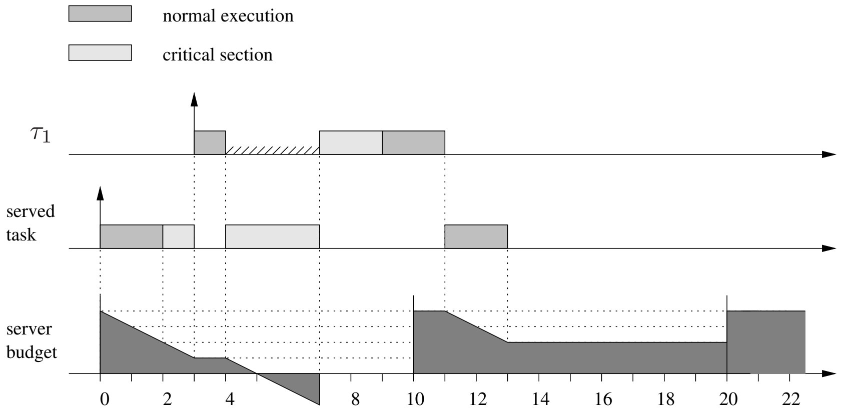


Figure 9.23 Example of budget overrun to allow resource sharing within reservations.

This approach was first proposed by Abeni and Buttazzo under EDF, using a Constant Bandwidth Server (CBS) [AB04]. Then, it was analyzed under fixed priority systems by Davis and Burns [DB06] and later extended under EDF by Behnam et al. [BSNN08, BNSS10]. Davis and Burns proposed two versions of this mechanism:

1. *overrun with payback*, where the server pays back in the next execution instant, in that the next budget replenishment is decreased by the overrun value;
2. *overrun without payback*, where no further action is taken after the overrun.

Note that the first solution (budget check) does not affect the execution of tasks in other reservations, but penalizes the response time of the served task. On the contrary, the second solution (budget overrun) does not increase the response time of the served task at the cost of a greater bandwidth requirement for the reservation.

9.4 HANDLING PERMANENT OVERLOADS

This section presents some methodologies for handling permanent overload conditions occurring in periodic task systems when the total processor utilization exceeds one. Basically, there are three methods to reduce the load:

- **Job skipping.** This method reduces the total load by properly skipping (i.e., aborting) some job execution in the periodic tasks, in such a way that a minimum number of jobs per task is guaranteed to execute within their timing constraints.
- **Period adaptation.** According to this approach, the load is reduced by enlarging task periods to suitable values, so that the total workload can be kept below a desired threshold.
- **Service adaptation.** According to this method, the load is reduced by decreasing the computational requirements of the tasks, trading predictability with quality of service.

9.4.1 JOB SKIPPING

The computational load of a set of periodic tasks can be reduced by properly *skipping* a few jobs in the task set, in such a way that the remaining jobs can be scheduled within their deadlines. This approach is suitable for real-time applications characterized by soft or firm deadlines, such as those typically found in multimedia systems, where skipping a video frame once in a while is better than processing it with a long delay. Even in certain control applications, the sporadic skip of some job can be tolerated when the controlled systems is characterized by a high inertia.

To understand how job skipping can make an overloaded system schedulable, consider the following example, consisting of two tasks, with computation times $C_1 = 2$ and $C_2 = 8$ and periods $T_1 = 4$ and $T_2 = 12$. Since the processor utilization factor is $U_p = 14/12 > 1$, the system is under a permanent overload, and the tasks cannot be scheduled within their deadlines. Nevertheless, Figure 9.24 shows that skipping a job every three in task τ_1 the overload can be resolved and all the remaining jobs can be scheduled within their deadlines.

In order to control the overall system load, it is important to derive the relation between the number of skips (i.e., the number of aborted jobs per task) and the total computational demand. In 1995, Koren and Shasha [KS95] proposed a new task model (known as the *firm* periodic model) suited to be handled by this technique.

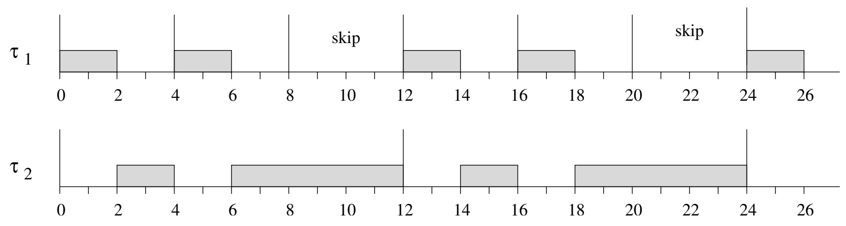


Figure 9.24 The overload condition resolved by skipping one job every three in task τ_1.

According to this model, each periodic task τ_i is characterized by the following parameters:

$$\tau_i(C_i, T_i, D_i, S_i)$$

where C_i is the worst-case computation time, T_i its period, D_i its relative deadline (assumed to be equal to the period), and S_i a skip parameter, $2 \le S_i \le \infty$, expressing the minimum distance between two consecutive skips. For example, if $S_i = 5$ the task can skip one instance every five. When $S_i = \infty$ no skips are allowed and τ_i is equivalent to a hard periodic task. The skip parameter can be viewed as a *Quality of Service* (QoS) metric (the higher S, the better the quality of service).

Using the terminology introduced by Koren and Shasha [KS95], every job of a periodic task can be *red* or *blue*: a red job must be completed within its deadline, whereas a blue job can be aborted at any time. To meet the constraint imposed by the skip parameter S_i, each scheduling algorithm must have the following characteristics:

- if a blue job is skipped, then the next $S_i - 1$ jobs must be red.
- if a blue job completes successfully, the next job is also blue.

The authors showed that making optimal use of skips is NP-hard and presented two algorithms (one working under Rate Monotonic and one under EDF) that exploit skips to schedule slightly overloaded systems. In general, these algorithms are not optimal, but they become optimal under a particular condition, called the *deeply-red* condition.

Definition 9.5 *A system is* deeply-red *if all tasks are synchronously activated and the first $S_i - 1$ instances of every task τ_i are red.*

Koren and Shasha showed that the worst case for a periodic skippable task set occurs when tasks are deeply-red. For this reason, all the results shown in this section are proved under this condition. This means that if a task set is schedulable under the deeply-red condition, it is also schedulable in any other situation.

SCHEDULABILITY ANALYSIS

The feasibility analysis of a set of firm tasks can be performed through the Processor Demand Criterion [BRH90] illustrated in Chapter 4, under the deeply-red condition, and assuming that in the worst case all blue jobs are aborted. In this worst-case scenario, the processor demand of τ_i due to the red jobs in an interval $[0, L]$ can be obtained as the difference between the demand of all the jobs and the demand of the blue jobs:

$$g_i^{skip}(0, L) = \left(\left\lfloor \frac{L}{T_i} \right\rfloor - \left\lfloor \frac{L}{T_i S_i} \right\rfloor \right) C_i. \tag{9.22}$$

Hence, the feasibility of the task set can be verified through the following test:

Sufficient condition

A set of firm periodic tasks is schedulable by EDF if

$$\forall L \geq 0 \qquad \sum_{i=1}^{n} \left(\left\lfloor \frac{L}{T_i} \right\rfloor - \left\lfloor \frac{L}{T_i S_i} \right\rfloor \right) C_i \;\leq\; L \tag{9.23}$$

A necessary condition can be easily derived by observing that a schedule is certainly infeasible when the utilization factor due to the red jobs is greater than one. That is,

Necessary condition

Necessary condition for the schedulability of a set of firm periodic tasks is that

$$\sum_{i=1}^{n} \frac{C_i(S_i - 1)}{T_i S_i} \;\leq\; 1 \tag{9.24}$$

EXAMPLE

To better clarify the concepts mentioned above, consider the task set shown in Figure 9.25 and the corresponding feasible schedule, obtained by EDF. Note that the processor utilization factor is greater than 1 ($U_p = 1.25$), but both conditions (9.23) and (9.24) are satisfied.

Task	**Task1**	**Task2**	**Task3**
Computation	1	2	5
Period	3	4	12
Skip Parameter	4	3	∞
U_p	1.25		

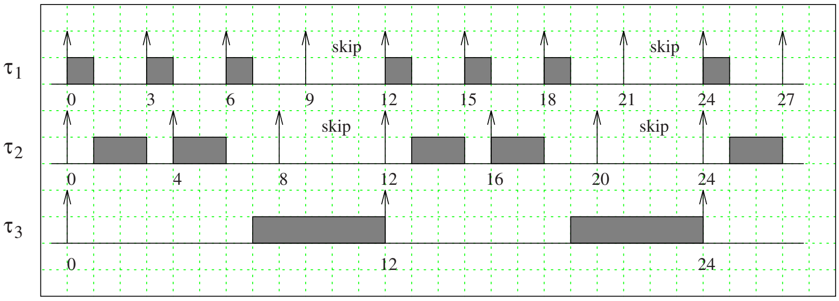


Figure 9.25 A schedulable set of firm periodic tasks.

SKIPS AND BANDWIDTH SAVING

If skips are permitted in the periodic task set, the spare time saved by rejecting the blue instances can be reallocated for other purposes. For example, for scheduling slightly overloaded systems or for advancing the execution of soft aperiodic requests.

Unfortunately, the spare time has a "granular" distribution and cannot be reclaimed at any time. Nevertheless, it can be shown that skipping blue instances still produces a bandwidth saving in the periodic schedule. Caccamo and Buttazzo [CB97] identified the amount of bandwidth saved by skips using a simple parameter, the *equivalent utilization factor* U_p^{skip}, which can be defined as

$$U_p^{skip} = \max_{L \geq 0} \left\{ \frac{\sum_i g_i^{skip}(0, L)}{L} \right\} \tag{9.25}$$

where $g_i^{skip}(0, L)$ is given in Equation (9.22).

Using this definition, the schedulability of a deeply-red skippable task set can be also verified using the following theorem ([CB97]):

Theorem 9.6 *A set Γ of deeply-red skippable periodic tasks is schedulable by EDF if*

$$U_p^{skip} \leq 1.$$

Note that the U_p^{skip} factor represents the net bandwidth really used by periodic tasks, under the deeply-red condition. It is easy to show that $U_p^{skip} \leq U_p$. In fact, according to Equation (9.25) (setting $S_i = \infty$), U_p can also be defined as

$$U_p = \max_{L \geq 0} \left\{ \frac{\sum_i \left\lfloor \frac{L}{T_i} \right\rfloor C_i}{L} \right\}.$$

Thus, $U_p^{skip} \leq U_p$ because

$$\left(\left\lfloor \frac{L}{T_i} \right\rfloor - \left\lfloor \frac{L}{T_i S_i} \right\rfloor \right) \leq \left\lfloor \frac{L}{T_i} \right\rfloor.$$

The bandwidth saved by skips can easily be exploited by an aperiodic server to advance the execution of aperiodic tasks. The following theorem ([CB97]) provides a sufficient condition for guaranteeing a hybrid (periodic and aperiodic) task set.

Theorem 9.7 *Given a set of periodic tasks that allow skip with equivalent utilization U_p^{skip} and a set of soft aperiodic tasks handled by a server with utilization factor U_s, the hybrid set is schedulable by EDF if*

$$U_p^{skip} + U_s \leq 1. \tag{9.26}$$

The fact that the condition of Theorem 9.7 is not necessary is a direct consequence of the "granular" distribution of the spare time produced by skips. In fact, a fraction of this spare time is uniformly distributed along the schedule and can be used as an additional free bandwidth ($U_p - U_p^{skip}$) available for aperiodic service. The remaining portion is discontinuous, and creates a kind of "holes" in the schedule, which can only be used in specific situations. Whenever an aperiodic request falls into some hole, it can exploit a bandwidth greater than $1 - U_p^{skip}$. Indeed, it is easy to find examples of feasible task sets with a server bandwidth $U_s > 1 - U_p^{skip}$. The following theorem ([CB97]) gives a maximum bandwidth U_s^{max} above which the schedule is certainly not feasible.

Theorem 9.8 *Given a set Γ of n periodic tasks that allow skips and an aperiodic server with bandwidth U_s, a necessary condition for the feasibility of Γ is*

$$U_s \leq U_s^{max}$$

where

$$U_s^{max} = 1 - U_p + \sum_{i=1}^{n} \frac{C_i}{T_i S_i}. \tag{9.27}$$

EXAMPLE

Consider the periodic task set shown in Table 9.1. The equivalent utilization factor of the periodic task set is $U_p^{skip} = 4/5$, while $U_s^{max} = 0.27$, leaving a bandwidth of $U_s = 1 - U_p^{skip} = 1/5$ for the aperiodic tasks. Three aperiodic jobs J_1, J_2, and J_3 are released at times $t_1 = 0$, $t_2 = 6$, and $t_3 = 18$; moreover, they have computation times $C_1^{ape} = 1$, $C_2^{ape} = 2$, and $C_3^{ape} = 1$, respectively.

Task	**Task1**	**Task2**
Computation	2	2
Period	3	5
Skip Parameter	2	∞
U_p	1.07	
U_p^{skip}	0.8	
$1 - U_p^{skip}$	0.2	
U_s^{max}	0.27	

Table 9.1 A schedulable task set.

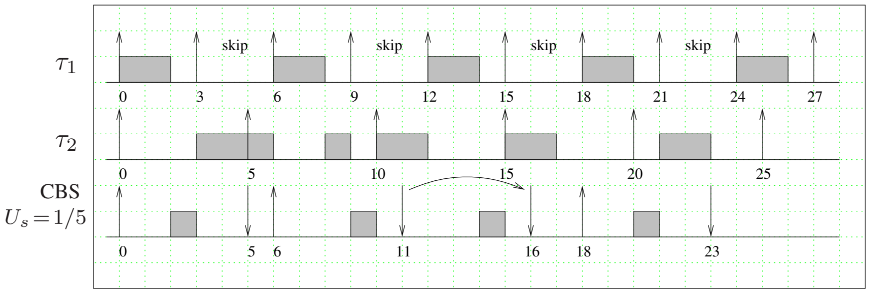


Figure 9.26 Schedule produced by EDF+CBS for the task set shown in Table 9.1.

Supposing aperiodic activities are scheduled by a CBS server with budget $Q^s = 1$ and period $T^s = 5$, Figure 9.26 shows the resulting schedule under EDF+CBS. Note that J_2 has a deadline postponement (according to CBS rules) at time $t = 10$ with new server deadline $d^s_{new} = d^s_{old} + T^s = 11 + 5 = 16$. According to the sufficient schedulability test provided by Theorem 9.7, the task set is schedulable when the CBS is assigned a bandwidth $U_s = 1 - U_p^{skip}$. However, this task set is also schedulable with a bandwidth $U_s = 0.25$, greater than $1 - U_p^{skip}$ but less than U_s^{max}, although this is not generally true.

9.4.2 PERIOD ADAPTATION

There are several real-time applications in which timing constraints are not rigid, but depend on the system state. For example, in a flight control system, the sampling rate of the altimeters is a function of the current altitude of the aircraft: the lower the altitude, the higher the sampling frequency. A similar need arises in mobile robots operating in unknown environments, where trajectories are planned based on the current sensory information. If a robot is equipped with proximity sensors, to maintain a desired performance, the acquisition rate of the sensors must increase whenever the robot is approaching an obstacle.

The possibility of varying tasks' rates also increases the flexibility of the system in handling overload conditions, providing a more general admission control mechanism. For example, whenever a new task cannot be guaranteed, instead of rejecting the task, the system can reduce the utilizations of the other tasks (by increasing their periods in a controlled fashion) to decrease the total load and accommodate the new request.

In the real-time literature, several approaches exist for dealing with an overload through a period adaptation. For example, Kuo and Mok [KM91] propose a load scaling technique to gracefully degrade the workload of a system by adjusting the periods of processes. In this work, tasks are assumed to be equally important and the objective is to minimize the number of fundamental frequencies to improve schedulability under static priority assignments. Nakajima and Tezuka [NT94] show how a real-time system can be used to support an adaptive application: whenever a deadline miss is detected, the period of the failed task is increased. Seto et al. [SLSS96] change tasks' periods within a specified range to minimize a performance index defined over the task set. This approach is effective at a design stage to optimize the performance of a discrete control system, but cannot be used for online load adjustment. Lee, Rajkumar and Mercer [LRM96] propose a number of policies to dynamically adjust the tasks' rates in overload conditions. Abdelzaher, Atkins, and Shin [AAS97] present a model for QoS negotiation to meet both predictability and graceful degradation requirements during overloads. In this model, the QoS is specified as a set of negotiation options in terms of rewards and rejection penalties. Nakajima [Nak98] shows how a multimedia activity can adapt its requirements during transient overloads by scaling down its rate or its computational demand. However, it is not clear how the QoS can be increased when the system is underloaded. Beccari et al. [BCRZ99] propose several policies for handling overload through period adjustment. The authors, however, do not address the problem of increasing the task rates when the processor is not fully utilized.

Although these approaches may lead to interesting results in specific applications, a more general framework can be used to avoid a proliferation of policies and treat different applications in a uniform fashion.

The elastic model presented in this section (originally proposed by Buttazzo, Abeni, and Lipari [BAL98] and later extended by Buttazzo, Lipari, Caccamo, and Abeni [BLCA02]), provides a novel theoretical framework for flexible workload management in real-time applications.

EXAMPLES

To better understand the idea behind the elastic model, consider a set of three periodic tasks, with computation times $C_1 = 10$, $C_2 = 10$, and $C_3 = 15$ and periods $T_1 = 20$, $T_2 = 40$, and $T_3 = 70$. Clearly, the task set is schedulable by EDF because

$$U_p = \frac{10}{20} + \frac{10}{40} + \frac{15}{70} = 0.964 < 1.$$

To allow a certain degree of flexibility, suppose that tasks are allowed to run with periods ranging within two values, reported in Table 9.2.

	$T_{i_{min}}$	$T_{i_{max}}$
τ_1	20	25
τ_2	40	50
τ_3	35	80

Table 9.2 Period ranges for the task set considered in the example.

Now, suppose that a new task τ_4, with computation time $C_4 = 5$ and period $T_4 = 30$, enters the system at time t. The total processor utilization of the new task set is

$$U_p = \frac{10}{20} + \frac{10}{40} + \frac{15}{70} + \frac{5}{30} = 1.131 > 1.$$

In a rigid scheduling framework, τ_4 should be rejected to preserve the timing behavior of the previously guaranteed tasks. However, τ_4 can be accepted if the periods of the other tasks can be increased in such a way that the total utilization is less than one. For example, if T_1 can be increased up to 23, the total utilization becomes $U_p = 0.989$, and hence τ_4 can be accepted.

As another example, if tasks are allowed to change their frequency and τ_3 reduces its period to 50, no feasible schedule exists, since the utilization would be greater than 1:

$$U_p = \frac{10}{20} + \frac{10}{40} + \frac{15}{50} = 1.05 > 1.$$

Note that a feasible schedule exists for $T_1 = 22$, $T_2 = 45$, and $T_3 = 50$. Hence, the system can accept the higher request rate of τ_3 by slightly decreasing the rates of τ_1 and τ_2. Task τ_3 can even run with a period $T_3 = 40$, since a feasible schedule exists with periods T_1 and T_2 within their range. In fact, when $T_1 = 24$, $T_2 = 50$, and $T_3 = 40$, $U_p = 0.992$. Finally, note that if τ_3 requires to run at its minimum period ($T_3 = 35$), there is no feasible schedule with periods T_1 and T_2 within their range, hence the request of τ_3 to execute with a period $T_3 = 35$ must be rejected.

Clearly, for a given value of T_3, there can be many different period configurations that lead to a feasible schedule; thus one of the possible feasible configurations must be selected. The elastic approach provides an efficient way for quickly selecting a feasible period configuration among all the possible solutions.

THE ELASTIC MODEL

The basic idea behind the elastic model is to consider each task as flexible as a spring with a given rigidity coefficient and length constraints. In particular, the utilization of a task is treated as an elastic parameter, whose value can be modified by changing the period within a specified range.

Each task is characterized by four parameters: a computation time C_i, a nominal period T_{i_0} (considered as the minimum period), a maximum period $T_{i_{max}}$, and an elastic coefficient $E_i \geq 0$, which specifies the flexibility of the task to vary its utilization for adapting the system to a new feasible rate configuration. The greater E_i, the more elastic the task. Thus, an elastic task is denoted as

$$\tau_i(C_i, T_{i_0}, T_{i_{max}}, E_i).$$

In the following, T_i will denote the actual period of task τ_i, which is constrained to be in the range $[T_{i_0}, T_{i_{max}}]$. Any task can vary its period according to its needs within the specified range. Any variation, however, is subject to an *elastic* guarantee and is accepted only if there is a feasible schedule in which all the other periods are within their range.

Under the elastic model, given a set of n periodic tasks with utilization $U_p > U_{max}$, the objective of the guarantee is to compress tasks' utilization factors to achieve a new desired utilization $U_d \leq U_{max}$ such that all the periods are within their ranges.

The following definitions are also used in this section:

$$\begin{aligned} U_{i_{min}} &= C_i/T_{i_{max}} \\ U_{min} &= \sum_{i=1}^{n} U_{i_{min}} \\ U_{i_0} &= C_i/T_{i_0} \\ U_0 &= \sum_{i=1}^{n} U_{i_0} \end{aligned}$$

Clearly, a solution can always be found if $U_{min} \leq U_d$; hence, this condition has to be verified a priori.

It is worth noting that the elastic model is more general than the classical Liu and Layland's task model, so it does not prevent a user from defining hard real-time tasks. In fact, a task having $T_{i_{max}} = T_{i_0}$ is equivalent to a hard real-time task with fixed period, independently of its elastic coefficient. A task with $E_i = 0$ can arbitrarily vary its period within its specified range, but it cannot be varied by the system during load reconfigurations.

To understand how an elastic guarantee is performed in this model, it is convenient to compare an elastic task τ_i with a linear spring S_i characterized by a rigidity coefficient k_i, a nominal length x_{i_0}, and a minimum length $x_{i_{min}}$. In the following, x_i will denote the actual length of spring S_i, which is constrained to be greater than or equal to $x_{i_{min}}$.

In this comparison, the length x_i of the spring is equivalent to the task's utilization factor $U_i = C_i/T_i$, and the rigidity coefficient k_i is equivalent to the inverse of the task's elasticity ($k_i = 1/E_i$). Hence, a set of n periodic tasks with total utilization factor $U_p = \sum_{i=1}^{n} U_i$ can be viewed as a sequence of n springs with total length $L = \sum_{i=1}^{n} x_i$.

In the linear spring system, this is equivalent to compressing the springs so that the new total length L_d is less than or equal to a given maximum length L_{max}. More formally, in the spring system the problem can be stated as follows.

> Given a set of n springs with known rigidity and length constraints, if the total length $L_0 = \sum_{i=1}^{n} x_{i_0} > L_{max}$, find a set of new lengths x_i such that $x_i \geq x_{i_{min}}$ and $\sum_{i=1}^{n} x_i = L_d$, where L_d is any arbitrary desired length such that $L_d < L_{max}$.

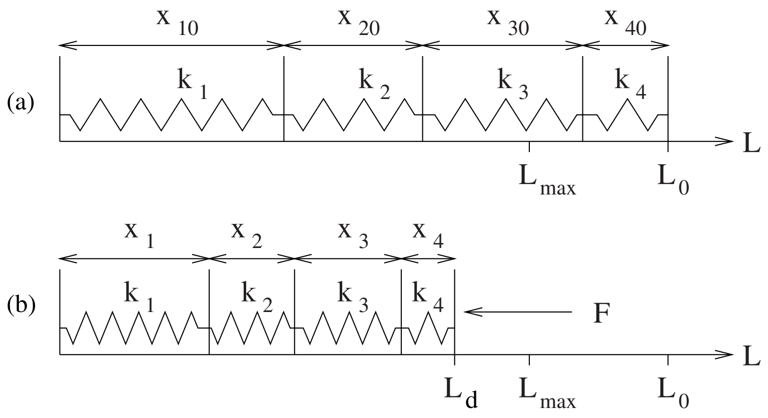


Figure 9.27 A linear spring system: (a) the total length is L_0 when springs are uncompressed; (b) the total length is $L_d < L_0$ when springs are compressed by a force F.

For the sake of clarity, we first solve the problem for a spring system without length constraints (i.e., $x_{i_{min}} = 0$), then we show how the solution can be modified by introducing length constraints, and finally we show how the solution can be adapted to the case of a task set.

SPRINGS WITH NO LENGTH CONSTRAINTS

Consider a set Γ of n springs with nominal length x_{i_0} and rigidity coefficient k_i positioned one after the other, as depicted in Figure 9.27. Let L_0 be the total length of the array; that is, the sum of the nominal lengths: $L_0 = \sum_{i=1}^{n} x_{i_0}$. If the array is compressed so that its total length is equal to a desired length L_d ($0 < L_d < L_0$), the first problem we want to solve is to find the new length x_i of each spring, assuming that for all i, $0 < x_i < x_{i_0}$ (i.e., $x_{i_{min}} = 0$).

Being L_d the total length of the compressed array of springs, we have

$$L_d = \sum_{i=1}^{n} x_i. \tag{9.28}$$

If F is the force that keeps a spring in its compressed state, then for the equilibrium of the system, it must be

$$\forall i \quad F = k_i(x_{i_0} - x_i),$$

from which we derive

$$\forall i \quad x_i = x_{i_0} - \frac{F}{k_i}. \tag{9.29}$$

By summing equations (9.29) we have

$$L_d = L_0 - F \sum_{i=1}^{n} \frac{1}{k_i}.$$

Thus, force F can be expressed as

$$F = K_p(L_0 - L_d), \tag{9.30}$$

where

$$K_p = \frac{1}{\sum_{i=1}^{n} \frac{1}{k_i}}. \tag{9.31}$$

Substituting expression (9.30) into Equation (9.29) we finally achieve

$$\forall i \quad x_i = x_{i_0} - (L_0 - L_d)\frac{K_p}{k_i}. \tag{9.32}$$

Equation (9.32) allows us to compute how each spring has to be compressed in order to have a desired total length L_d.

For a set of elastic tasks, Equation (9.32) can be translated as follows:

$$\forall i \quad U_i = U_{i_0} - (U_0 - U_d)\frac{E_i}{E_0}. \tag{9.33}$$

where $E_i = 1/k_i$ and $E_0 = \sum_{i=1}^{n} E_i$.

INTRODUCING LENGTH CONSTRAINTS

If each spring has a length constraint, in the sense that its length cannot be less than a minimum value $x_{i_{min}}$, then the problem of finding the values x_i requires an iterative solution. In fact, if during compression one or more springs reach their minimum length, the additional compression force will only deform the remaining springs. Such a situation is depicted in Figure 9.28.

Thus, at each instant, the set Γ can be divided into two subsets: a set Γ_f of fixed springs having minimum length, and a set Γ_v of variable springs that can still be compressed. Applying Equations (9.32) to the set Γ_v of variable springs, we have

$$\forall S_i \in \Gamma_v \quad x_i = x_{i_0} - (L_{v_0} - L_d + L_f)\frac{K_v}{k_i} \tag{9.34}$$

where

$$L_{v_0} = \sum_{S_i \in \Gamma_v} x_{i_0} \tag{9.35}$$

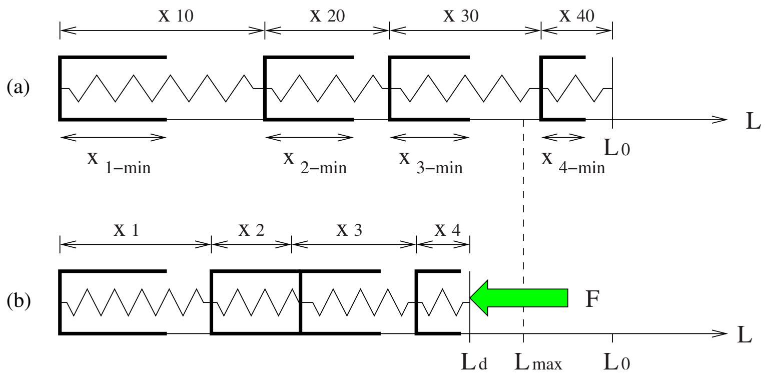


Figure 9.28 Springs with minimum length constraints (a); during compression, spring S_2 reaches its minimum length and cannot be compressed any further (b).

$$L_f = \sum_{S_i \in \Gamma_f} x_{i_{min}} \tag{9.36}$$

$$K_v = \frac{1}{\sum_{S_i \in \Gamma_v} \frac{1}{k_i}}. \tag{9.37}$$

Whenever there exists some spring for which Equation (9.34) gives $x_i < x_{i_{min}}$, the length of that spring has to be fixed at its minimum value, sets Γ_f and Γ_v must be updated, and Equations (9.34), (9.35), (9.36) and (9.37) recomputed for the new set Γ_v. If there is a feasible solution, that is, if $L_d \geq L_{min} = \sum_{i=1}^{n} x_{i_{min}}$, the iterative process ends when each value computed by Equations (9.34) is greater than or equal to its corresponding minimum $x_{i_{min}}$.

COMPRESSING TASKS' UTILIZATIONS

When dealing with a set of elastic tasks, Equations (9.34), (9.35), (9.36) and (9.37) can be rewritten by substituting all length parameters with the corresponding utilization factors, and the rigidity coefficients k_i and K_v with the corresponding elastic coefficients E_i and E_v. Similarly, at each instant, the set Γ can be divided into two subsets: a set Γ_f of fixed tasks having minimum utilization, and a set Γ_v of variable tasks that can still be compressed. Let $U_{i_0} = C_i/T_{i_0}$ be the nominal utilization of task τ_i, $U_0 = \sum_{i=1}^{n} U_{i_0}$ be the nominal utilization of the task set, U_{v_0} be the sum of the nominal utilizations of tasks in Γ_v, and U_f be the total utilization factor of tasks in Γ_f. Then, to achieve a desired utilization $U_d < U_0$ each task has to be compressed up to the following utilization:

$$\forall \tau_i \in \Gamma_v \quad U_i = U_{i_0} - (U_{v_0} - U_d + U_f)\frac{E_i}{E_v} \tag{9.38}$$

where

$$U_{v_0} = \sum_{\tau_i \in \Gamma_v} U_{i_0} \tag{9.39}$$

$$U_f = \sum_{\tau_i \in \Gamma_f} U_{i_{min}} \tag{9.40}$$

$$E_v = \sum_{\tau_i \in \Gamma_v} E_i. \tag{9.41}$$

If there are tasks for which $U_i < U_{i_{min}}$, then the period of those tasks has to be fixed at its maximum value $T_{i_{max}}$ (so that $U_i = U_{i_{min}}$), sets Γ_f and Γ_v must be updated (hence, U_f and E_v recomputed), and Equation (9.38) applied again to the tasks in Γ_v. If there is a feasible solution, that is, if the desired utilization U_d is greater than or equal to the minimum possible utilization $U_{min} = \sum_{i=1}^{n} \frac{C_i}{T_{i_{max}}}$, the iterative process ends when each value computed by Equation (9.38) is greater than or equal to its corresponding minimum $U_{i_{min}}$. The algorithm[2] for compressing a set Γ of n elastic tasks up to a desired utilization U_d is shown in Figure 9.29.

DECOMPRESSION

All tasks' utilizations that have been compressed to cope with an overload situation can return toward their nominal values when the overload is over. Let Γ_c be the subset of compressed tasks (that is, the set of tasks with $T_i > T_{i_0}$), let Γ_a be the set of remaining tasks in Γ (that is, the set of tasks with $T_i = T_{i_0}$), and let U_d be the current processor utilization of Γ. Whenever a task in Γ_a voluntarily increases its period, all tasks in Γ_c can expand their utilizations according to their elastic coefficients, so that the processor utilization is kept at the value of U_d.

Now, let U_c be the total utilization of Γ_c, let U_a be the total utilization of Γ_a after some task has increased its period, and let U_{c0} be the total utilization of tasks in Γ_c at their nominal periods. It can easily be seen that if $U_{c0} + U_a \leq U_{lub}$, all tasks in Γ_c can return to their nominal periods. On the other hand, if $U_{c0} + U_a > U_{lub}$, then the release operation of the tasks in Γ_c can be viewed as a compression, where $\Gamma_f = \Gamma_a$ and $\Gamma_v = \Gamma_c$. Hence, it can still be performed by using Equations (9.38), (9.40) and (9.41) and the algorithm presented in Figure 9.29.

[2]The actual implementation of the algorithm contains more checks on tasks' variables, which are not shown here in order to simplify its description.

```
Algorithm: Elastic_compression($\Gamma, U_d$)
Input: A task set $\Gamma$ and a desired utilization $U_d < 1$
Output: A task set with modified periods such that $U_p = U_d$

begin

(1)   $U_{min} := \sum_{i=1}^{n} C_i/T_{i_{max}}$;
(2)   if ($U_d < U_{min}$) return(INFEASIBLE);
(3)   for ($i := 1$ to $n$) $U_{i_0} := C_i/T_{i_0}$;

(4)   do
(5)       $U_f := 0; U_{v_0} := 0; E_v := 0$;
(6)       for ($i := 1$ to $n$) do
(7)           if (($E_i == 0$) or ($T_i == T_{i_{max}}$)) then
(8)               $U_f := U_f + U_{i_{min}}$;
(9)           else
(10)              $E_v := E_v + E_i$;
(11)              $U_{v_0} := U_{v_0} + U_{i_0}$;
(12)          end
(13)      end

(14)      $ok := 1$;
(15)      for ($each\ \tau_i \in \Gamma_v$) do
(16)          if (($E_i > 0$) and ($T_i < T_{i_{max}}$)) then
(17)              $U_i := U_{i_0} - (U_{v_0} - U_d + U_f)E_i/E_v$;
(18)              $T_i := C_i/U_i$;
(19)              if ($T_i > T_{i_{max}}$) then
(20)                  $T_i := T_{i_{max}}$;
(21)                  $ok := 0$;
(22)              end
(23)          end
(24)      end

(25)  while ($ok == 0$);
(26)  return(FEASIBLE);

end
```

Figure 9.29 Algorithm for compressing a set of elastic tasks.

9.4.3 IMPLEMENTATION ISSUES

The elastic compression algorithm can be efficiently implemented on top of a real-time kernel as a routine (elastic manager) that is activated every time a new task is created, terminated, or there is a request for a period change. When activated, the elastic manager computes the new periods according to the compression algorithm and modifies them atomically.

To avoid any deadline miss during the transition phase, it is crucial to ensure that all the periods are modified at opportune time instants, according to the following rule [BLCA02]:

> The period of a task τ_i can be increased at any time, but can only be reduced at the next job activation.

Figure 9.30 shows an example in which τ_1 misses its deadline when its period is reduced at time $t = 15$ (i.e., before its next activation time ($t = 20$). Note that the task set utilization is $U_p = 29/30$ before compression, and $U'_p = 28/30$ after compression. This means that although the system is schedulable by EDF in both steady state conditions, some deadline can be missed if a period is reduced too early.

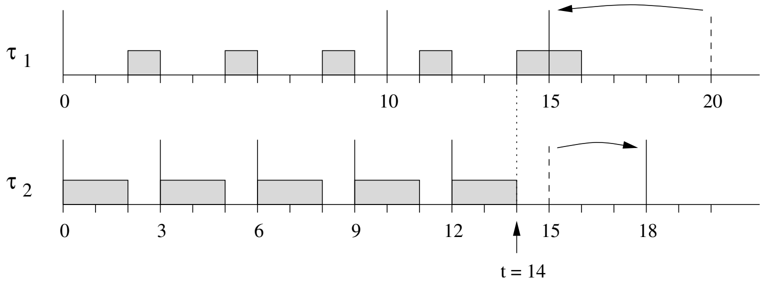


Figure 9.30 A task can miss its deadline if a period is reduced at an arbitrary time instant.

An earlier instant at which a period can be safely reduced without causing any deadline miss in the transition phase has been computed by Buttazzo et al. [BLCA02] and later improved by Guangming [Gua09].

PERIOD RESCALING

If the elastic coefficients are set equal to task nominal utilizations, elastic compression has the effect of a simple rescaling, where all the periods are increased by the same percentage. In order to work correctly, however, period rescaling must be uniformly applied to all the tasks, without restrictions on the maximum period. This means having $U_f = 0$ and $U_{v_0} = U_0$. Under this assumption, by setting $E_i = U_{i_0}$, Equation (9.38) becomes:

$$\forall i \quad U_i = U_{i_0} - (U_0 - U_d)\frac{U_{i_0}}{U_0} = \frac{U_{i_0}}{U_0}[U_0 - (U_0 - U_d)] = \frac{U_{i_0}}{U_0}U_d$$

from which we have

$$T_i = T_{i_0}\frac{U_0}{U_d}. \tag{9.42}$$

This means that in overload situations ($U_0 > 1$) the compression algorithm causes all task periods to be increased by a common scale factor

$$\eta = \frac{U_0}{U_d}.$$

Note that after compression is performed, the total processor utilization becomes

$$U = \sum_{i=1}^{n} \frac{C_i}{\eta T_{i_0}} = \frac{1}{\eta}U_0 = \frac{U_d}{U_0}U_0 = U_d$$

as desired.

If a maximum period needs to be defined for some task, an online guarantee test can easily be performed before compression to check whether all the new periods are less than or equal to the maximum value. This can be done in $O(n)$ by testing whether

$$\forall i = 1, \ldots, n \quad \eta T_{i_0} \leq T_i^{max}.$$

By deciding to apply period rescaling, we lose the freedom of choosing the elastic coefficients, since they must be set equal to task nominal utilizations. However, this technique has the advantage of leaving the task periods ordered as in the nominal configuration, which simplifies the compression algorithm in the presence of resource constraints and enables its usage in fixed priority systems, where priorities are typically assigned based on periods.

CONCLUDING REMARKS

The elastic model offers a flexible way to handle overload conditions. In fact, whenever a new task cannot be guaranteed by the system, instead of rejecting the task, the system can try to reduce the utilizations of the other tasks (by increasing their periods in a controlled fashion) to decrease the total load and accommodate the new request. As soon as a transient overload condition is over (because a task terminates or voluntarily increases its period) all the compressed tasks may expand up to their original utilization, eventually recovering their nominal periods.

The major advantage of the elastic method is that the policy for selecting a solution is implicitly encoded in the elastic coefficients provided by the user (e.g., based on task importance). Each task is varied based on its elastic status and a feasible configuration is found, if one exists. This is very useful for supporting both multimedia systems and control applications, in which the execution rates of some computational activities have to be dynamically tuned as a function of the current system state. Furthermore, the elastic mechanism can easily be implemented on top of classical real-time kernels, and can be used under fixed or dynamic priority scheduling algorithms.

It is worth observing that the elastic approach is not limited to task scheduling. Rather, it represents a general resource allocation methodology that can be applied whenever a resource has to be allocated to objects whose constraints allow a certain degree of flexibility. For example, in a distributed system, dynamic changes in node transmission rates over the network could be efficiently handled by assigning each channel an elastic bandwidth, which could be tuned based on the actual network traffic. An application of the elastic model to the network has been proposed by Pedreiras et al. [PGBA02].

Another interesting application of the elastic approach is to automatically adapt task rates to the current load, without specifying worst-case execution times. If the system is able to monitor the actual execution time of each job, such data can be used to compute the actual processor utilization. If this is less than one, task rates can be increased according to elastic coefficients to fully utilize the processor. On the other hand, if the actual processor utilization is a little greater than one and some deadline misses are detected, task rates can be reduced to bring the processor utilization to a desired safe value.

The elastic model has also been extended to deal with resource constraints [BLCA02], thus allowing tasks to interact through shared memory buffers. In order to estimate maximum blocking times due to mutual exclusion and analyze task schedulability, critical sections are assumed to be accessed through the Stack Resource Policy [Bak91].

9.4.4 SERVICE ADAPTATION

A third method for coping with a permanent overload condition is to reduce the load by decreasing the task computation times. This can be done only if the tasks have been originally designed to trade performance with computational requirements. When tasks use some incremental algorithm to produce approximated results, the precision of results is related to the number of iterations, and thus with the computation time. In this case, an overload condition can be handled by reducing the quality of results, aborting the remaining computation if the quality of the current results is acceptable.

The concept of imprecise and approximate computation has emerged as a new approach to increasing flexibility in dynamic scheduling by trading computation accuracy with timing requirements. If processing time is not enough to produce high-quality results within the deadlines, there could be enough time for producing approximate results with a lower quality. This concept has been formalized by many authors [LNL87, LLN87, LLS$^+$91, SLC91, LSL$^+$94, Nat95] and specific techniques have been developed for designing programs that can produce partial results.

In a real-time system that supports imprecise computation, every task τ_i is decomposed into a *mandatory* subtask M_i and an *optional* subtask O_i. The mandatory subtask is the portion of the computation that must be done in order to produce a result of acceptable quality, whereas the optional subtask refines this result [SLCG89]. Both subtasks have the same arrival time a_i and the same deadline d_i as the original task τ_i; however, O_i becomes ready for execution when M_i is completed. If C_i is the worst-case computation time associated with the task, subtasks M_i and O_i have computation times m_i and o_i, such that $m_i + o_i = C_i$. In order to guarantee a minimum level of performance, M_i must be completed within its deadline, whereas O_i can be left incomplete, if necessary, at the expense of the quality of the result produced by the task.

It is worth noting that the task model used in traditional real-time systems is a special case of the one adopted for imprecise computation. In fact, a hard task corresponds to a task with no optional part ($o_i = 0$), whereas a soft task is equivalent to a task with no mandatory part ($m_i = 0$).

In systems that support imprecise computation, the *error* ϵ_i in the result produced by τ_i (or simply the error of τ_i) is defined as the length of the portion of O_i discarded in the schedule. If σ_i is the total processor time assigned to O_i by the scheduler, the error of task τ_i is equal to

$$\epsilon_i = o_i - \sigma_i.$$

The *average error* $\overline{\epsilon}$ on the task set is defined as

$$\overline{\epsilon} = \sum_{i=1}^{n} w_i \epsilon_i,$$

where w_i is the relative importance of τ_i in the task set. An error $\epsilon_i > 0$ means that a portion of subtask O_i has been discarded in the schedule at the expense of the quality of the result produced by task τ_i, but for the benefit of other mandatory subtasks that can complete within their deadlines.

In this model, a schedule is said to be *feasible* if every mandatory subtask M_i is completed within its deadline. A schedule is said to be *precise* if the average error $\overline{\epsilon}$ on the task set is zero. In a precise schedule, all mandatory and optional subtasks are completed within their deadlines.

As an illustrative example, let us consider a set of jobs $\{J_1, \ldots, J_n\}$ shown in Figure 9.31a. Note that this task set cannot be precisely scheduled; however, a feasible schedule with an average error of $\overline{\epsilon} = 4$ can be found, and it is shown in Figure 9.31b. In fact, all mandatory subtasks finish within their deadlines, whereas not all optional subtasks are able to complete. In particular, a time unit of execution is subtracted from O_1, two units from O_3, and one unit from O_5. Hence, assuming that all tasks have an importance value equal to one ($w_i = 1$), the average error on the task set is $\overline{\epsilon} = 4$.

For a set of periodic tasks, the problem of deciding the best level of quality compatible with a given load condition can be solved by associating each optional part of a task a reward function $R_i(\sigma_i)$, which indicates the reward accrued by the task when it receives σ_i units of service beyond its mandatory portion. This problem has been addressed by Aydin et al. [AMMA01], who presented an optimal algorithm that maximizes the weighted average of the rewards over the task set.

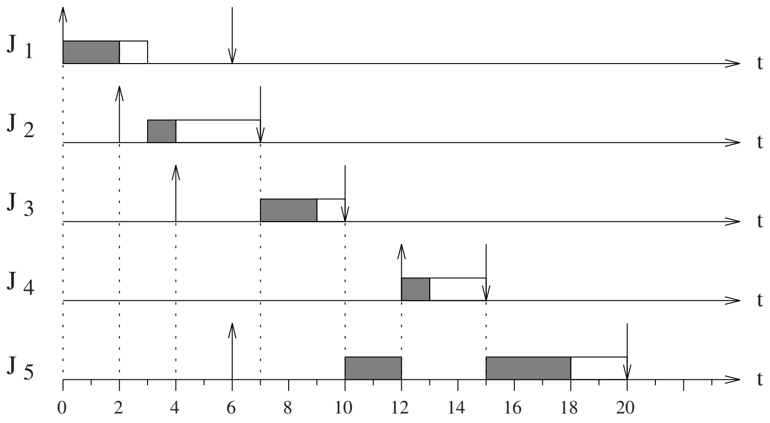
Note that in the absence of a reward function, the problem can easily be solved by using a compression algorithm like the elastic approach. In fact, once, the new task utilizations U'_i are computed, the new computation times C'_i that lead to a given desired load can easily be computed from the periods as

$$C'_i = T_i U'_i.$$

Finally, if an algorithm cannot be executed in an incremental fashion or it cannot be aborted at any time, a task can be provided with multiple versions, each characterized by a different quality of performance and execution time. Then, the value C'_i can be used to select the task version having the computation time closer to, but smaller than C'_i.

	a_i	d_i	C_i	m_i	o_i
J_1	0	6	4	2	2
J_2	2	7	4	1	3
J_3	4	10	5	2	3
J_4	12	15	3	1	2
J_5	6	20	8	5	3

(a)



(b)

Figure 9.31 An example of an imprecise schedule.

Exercises

9.1 For the set of two aperiodic jobs reported in the table, compute the instantaneous load for all instants in [0,8].

	r_i	C_i	D_i
J_1	3	3	5
J_2	0	5	10

9.2 Verify the schedulability under EDF of the set of skippable tasks illustrated in the table:

	C_i	T_i	S_i
τ_1	2	5	∞
τ_2	2	6	4
τ_3	4	8	5

9.3 A resource reservation mechanism achieves temporal isolation by providing service using the following periodic time partition, in every window of 10 units: {[0,2], [5,6], [8,9]}. This means that the service is only provided in the three intervals indicated in the set and repeats every 10 units. Illustrate the resulting supply function $Z(t)$ and compute the (α, Δ) parameters of the corresponding bounded delay function.

9.4 Consider the set of elastic tasks illustrated in the table, to be scheduled by EDF:

	C_i	T_i^{min}	T_i^{max}	E_i
τ_1	9	15	30	1
τ_2	16	20	40	3

Since the task set is not feasible with the minimum periods, compute the new periods T_i' that make the task set feasible with a total utilization $U_d = 1$.

9.5 Considering the same periodic task set of the previous exercise, compute the new periods T_i' resulting by applying a period rescaling (hence, not using the elastic coefficients).