2

OVERLOAD MANAGEMENT

2.1 INTRODUCTION

A system is said to be in overload when the computational demand of the task set exceeds the available processing time. In a real-time system, an overload condition causes one or more tasks to miss their deadline and, if not properly handled, it may cause abrupt degradations of system performance.

Even when the system is properly designed, an overload can occur for different reasons, such as a new task activation, a system mode change, the simultaneous arrival of asynchronous events, a fault in a peripheral device, or the execution of system exceptions.

If the operating system is not conceived to handle overloads, the effect of a transient overload can be catastrophic. 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 2.1.

Figure 2.1a shows a feasible schedule of a task set executed under EDF. However, if at time t_0 task τ_0 is executed, all the previous tasks miss their deadlines (see Figure 2.1b). In general, under EDF, accepting a new task with deadline d^* causes all tasks with deadline longer than d^* to be delayed. Similarly, under fixed priority scheduling, the activation of a task τ_i with priority P_i delays all tasks with lower priority. In order to avoid domino effects, the operating system and the scheduling algorithm must be explicitly designed to handle transient overloads in a controlled fashion, so that the damage due to a deadline miss can be minimized.

In the real-time literature, several scheduling algorithms have been proposed to deal with overloads. In 1984, Ramamritham and Stankovic [RS84] used EDF to dynamically guarantee incoming work via on-line planning, and, if a newly arriving task could not

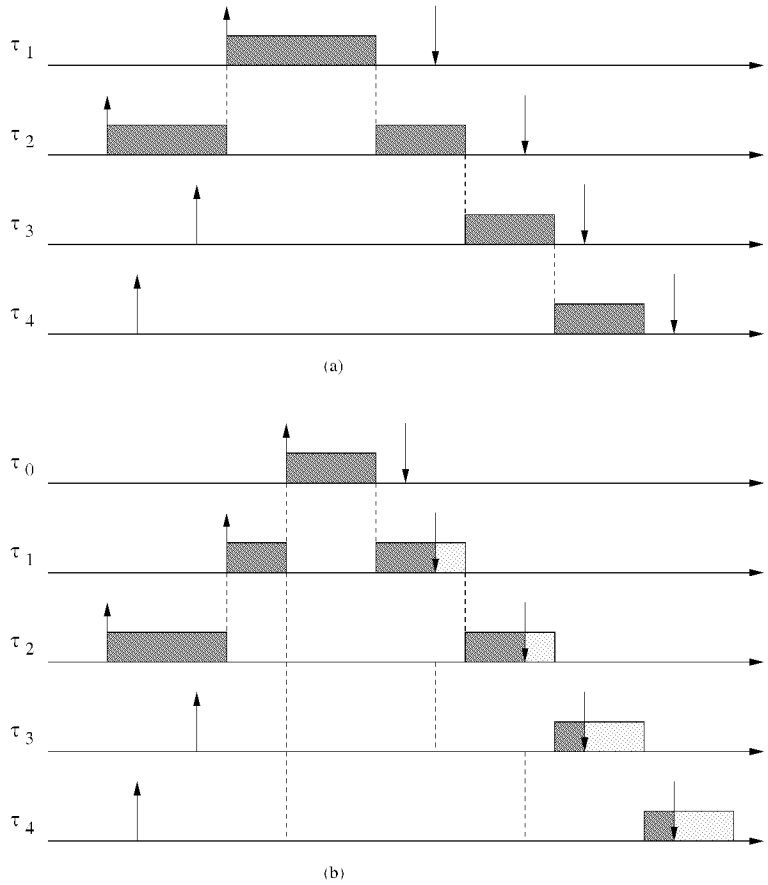


Figure 2.1 **a.** Feasible schedule with Earliest Deadline First, in normal load condition. **b.** Overload with domino effect due to the arrival of task τ_0.

be guaranteed, the task was either dropped or distributed scheduling was attempted. The dynamic guarantee performed in this approach had the effect of avoiding the catastrophic effects of overload on EDF.

In 1986, Locke [Loc86] developed an algorithm that makes a best effort at scheduling tasks based on earliest deadline with a rejection policy based on removing tasks with the minimum value density. He also suggested that removed tasks remain in the system until their deadline has passed. The algorithm computes the variance of the total slack

time in order to find the probability that the available slack time is less than zero. The calculated probability is used to detect a system overload. If it is less than the user prespecified threshold, the algorithm removes the tasks in increasing value density order.

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

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

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

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

Zlokapa, Stankovic, and Ramamritham [Zlo93] proposed an approach called *well-time scheduling*, which focuses on reducing the guarantee overhead in heavily loaded systems by delaying the guarantee. Various properties of the approach were developed via queueing theoretic arguments, and the results were a multilevel queue (based on an analytical derivation), similar to that found in [HLC91] (based on simulation).

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

2.2 LOAD DEFINITIONS

In Chapter 1, the processor workload in an interval of time $[t_1, t_2]$ has been defined as the ratio of the processor demand in that interval and the length of the interval:

$$\rho(t_1, t_2) = \frac{g(t_1, t_2)}{t_2 - t_1}.$$

Hence, the system loadin a given schedule is given by

$$\rho = \max_{t_1, t_2} \frac{g(t_1, t_2)}{t_2 - t_1}.$$

Computing the load using the previous definition, however, may not be practical, because the number of intervals $[t_1, t_2]$ can be very large. When the task set consists only of aperiodic activities, then a more effective method is to compute the instantaneous load $\rho(t)$, originally introduced by Buttazzo and Stankovic in [BS95]. According to this method, the load is computed at time t, based on the current set of active aperiodic tasks, each characterized by a remaining computation time $c_i(t)$ and a deadline d_i. In particular, the load at time t is computed as

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

where

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

Figure 2.2a shows an example of load calculation for a set of three real-time aperiodic tasks. At time $t = 6$, when τ_1 arrives, the loading factor $\rho_i(t)$ of each task is shown on the right of the timeline, so the instantaneous load at time 6 is $\rho(6) = 0.833$. Figure 2.2b shows the load as a function of time.

For a set of synchronous periodic tasks with deadlines less than or equal to periods, the processor demand can be computed from time $t = 0$ in an interval of length L as

$$g(0, L) = \sum_{i=1}^{n} \left\lfloor \frac{L + T_i - D_i}{T_i} \right\rfloor C_i.$$

Hence, the total processor workload can be computed as

$$\rho = \max_{L > 0} \frac{g(0, L)}{L}.$$

It is worth noticing that Baruah et al. [BMR90] showed that the maximum can be computed for L equal to task deadlines, up to a value $L_{max} = \min(H, L*)$, where H

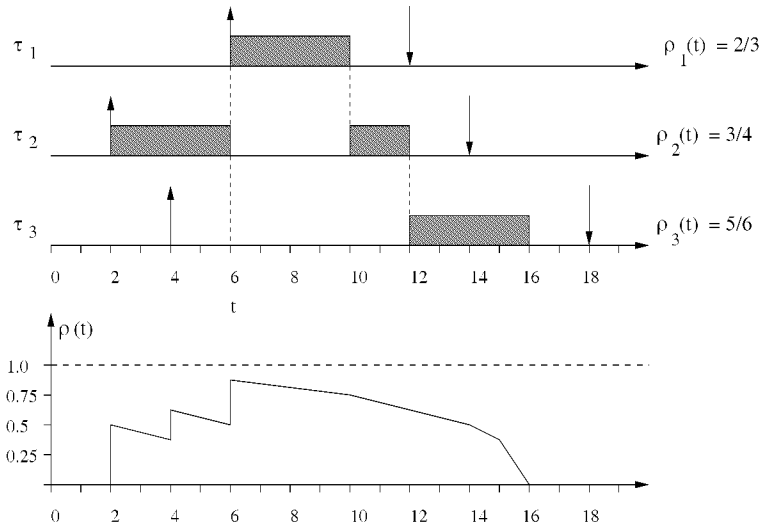


Figure 2.2 a. Load calculation for t = 6 in a set of three real-time tasks. b. Load as a function of time.

is the hyperperiod (i.e., the minimum common multiple of task periods) and

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

The methods proposed in the literature for dealing with permanent overload conditions can be grouped in two main categories:

1. **Admission control**. According to this method, each task is assigned an importance value. Then, in the presence of an overload, the least important tasks are rejected to keep the load under a desired threshold, whereas the other tasks receive full service.
2. **Performance degradation**. According to this method, no task is rejected, but the tasks are executed with reduced performance requirements.

2.3 ADMISSION CONTROL METHODS

When a real-time system is underloaded and dynamic activation of tasks is not allowed, there is no need to consider task importance in the scheduling policy, since there exist

optimal scheduling algorithms that can guarantee a feasible schedule under a set of assumptions. For example, Dertouzos [Der74] proved that EDF is an optimal algorithm for preemptive, independent tasks when there is no overload.

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

2.3.1 DEFINING VALUES

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 a utility function. Figure 2.3 illustrates some utility functions that can be associated with tasks in order to describe their importance. According to this view, a non-real-time task, which has no time constraints, has a low constant value, since it always contributes to the system value whenever it 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 in many situations. A task with a soft deadline, instead, can still give a value to the system if executed after its deadline, although this value may decrease with time. Then, there can be real-time activities, so-called *firm*, that do not jeopardize the system but give zero value if completed after their deadline.

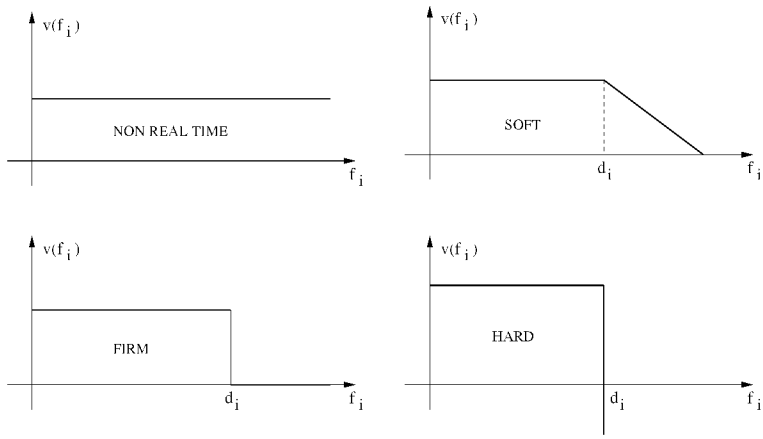


Figure 2.3 Utility functions that can be associated to a task to describe its importance.

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, the *cumulative value* achieved by a scheduling algorithm A is defined as follows:

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

Notice that if a hard task misses its deadline, the cumulative value achieved by the algorithm is minus infinity, even though all other tasks completed before their deadlines. For this reason, all activities with hard timing constraints should be guaranteed a priori by assigning them dedicated resources (included processors). If all hard tasks are guaranteed a priori, the objective of a real-time scheduling algorithm should be to guarantee a feasible schedule in underload conditions and maximize the cumulative value of soft and firm tasks during transient overloads.

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

It is easy to show that no optimal on-line algorithms exist in overloads. Without an a priori knowledge of the task arrival times, no on-line algorithm can guarantee the maximum cumulative value Γ^*. This value can only be achieved by an ideal clairvoyant scheduling algorithm that knows the future arrival time of any task.

A parameter that measures the worst-case performance of a scheduling algorithm in overload condition is the *competitive factor*, introduced by Baruah et al. in [BKM+92].

Definition 2.1 *A scheduling algorithm A has a* competitive factor φ_A *if and only if it can guarantee a cumulative value*

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

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

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

If the overload has an infinite duration, then no on-line algorithm can guarantee a competitive factor greater than zero. In real situations, however, overloads are intermittent and usually have a short duration; hence, it is desirable to use scheduling algorithms with a high competitive factor. An important theoretical result found in [BKM+92] is that there exists an upper bound on the competitive factor of any on-line algorithm. This is stated by the following theorem.

Theorem 2.1 (Baruah et al.) *In systems where the loading factor is greater than 2 ($\rho > 2$) and tasks' values are proportional to their computation times, no on-line algorithm can guarantee a competitive factor greater than* 0.25.

In general, the bound on the competitive factor as a function of the load has been computed in [BR91] and it is shown in Figure 2.4.

With respect to the strategy used to predict and handle overloads, most of the scheduling methods proposed in the literature can be divided into three main classes, illustrated in Figure 2.5:

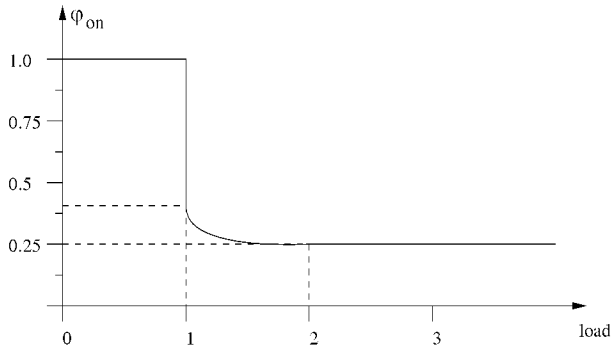


Figure 2.4 Bound of the competitive factor of an on-line scheduling algorithm as a function of the load.

- **Best Effort Scheduling**. 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.
- **Simple Admission Control**. This class includes those algorithms in which the load on the processor is controlled by an acceptance test executed at each task arrival. Typically, whenever a new task enters the system, a guarantee routine verifies the schedulability of the task set based on worst-case assumptions. If the task set is found schedulable, the new task is accepted in the ready queue; otherwise, it is rejected.
- **Robust Scheduling**. 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; otherwise, one or more tasks are rejected based on a different policy.

Notice that the simple admission control 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). This scheme, 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.

In the best effort scheme, the cumulative value can be increased using suitable heuristics for scheduling the tasks. For example, in the Spring kernel [SR87], Stankovic and Ramamritham proposed to schedule tasks by an appropriate heuristic function that can balance timing properties and importance values.

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

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

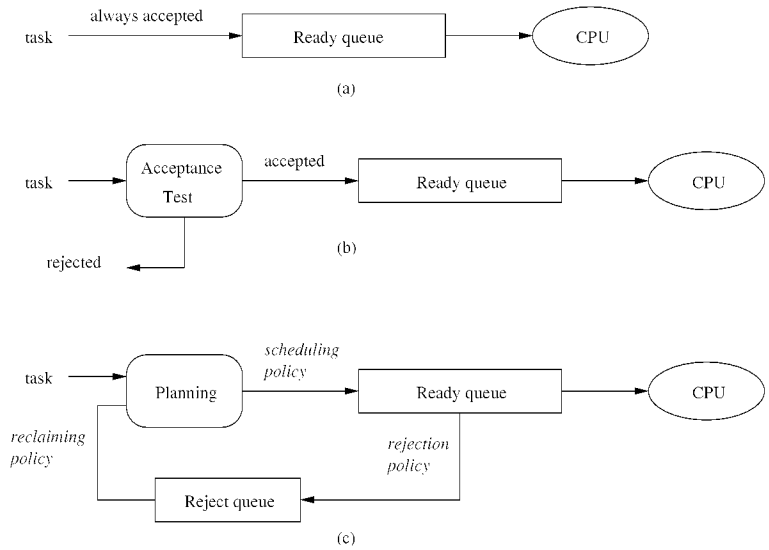


Figure 2.5 Scheduling schemes for handling overload situations. **a.** Best Effort scheduling. **b.** Admission control. **c.** Robust Scheduling.

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

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

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

The guarantee test performed in RED is formulated in terms of residual laxity L_i, defined as the interval between its estimated finishing time (f_i) and its primary (absolute) deadline (d_i). All residual laxities can be efficiently computed in $O(n)$, 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 executes after its secondary deadline:

$$E_i = \max(0, -(L_i + M_i)). \tag{2.1}$$

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 to plan an action to recover from the overload condition. Many recovering strategies can be used to solve this problem. The simplest one is to reject the least-value task that can remove the overload situation. In general, we assume that, whenever an overload is detected, some rejection policy will search for a subset J^* of least-value tasks that will be rejected to maximize the cumulative value of the remaining subset. The RED acceptance test is shown in Figure 2.6.

RED_acceptance_test(J, J_{new}) {

$E = 0$; /* Maximum Exceeding Time */
$L_0 = 0$;
$d_0 = current\_time()$;

$J' = J \cup \{J_{new}\}$;
k = <position of J_{new} in the task set J'>;

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

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

Figure 2.6 The RED acceptance test.

In RED, a resource reclaiming mechanism is used to take advantage of those tasks that complete before their worst-case finishing time. To reclaim the spare time, rejected tasks are not removed forever but temporarily parked in a queue, called *Reject Queue*, ordered by decreasing values. Whenever a running task completes its execution before its worst-case finishing time, the algorithm tries to reaccept the highest-value tasks in the Reject Queue having positive laxity. Tasks with negative laxity are removed from the system.

2.3.3 D_{OVER}: A COMPETITIVE ALGORITHM

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

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

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

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

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

It worth to observe that having the best competitive factor among all on-line algorithms does not mean having the best performance in *any* load condition. In fact, in order to guarantee the best competitive factor, D^{over} may reject tasks with values higher than the current task but not higher than the threshold that guarantees optimality. In other words, to cope with worst-case sequences, D^{over} does not take advantage of lucky sequences and may reject more value than it is necessary. In Section 2.3.4, the performance of D_{over} is tested for random task sets and compared with the one of other scheduling algorithms.

2.3.4 PERFORMANCE EVALUATION

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

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

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

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

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

In the graphs reported in Figure 2.7, 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.

In Figure 2.7, the GED curve refers to the guaranteed EDF scheme implemented with a simple admission control, while the RED curve refers to the robust EDF algorithm. For small values of β, that is, when tasks execute for almost their maximum computation

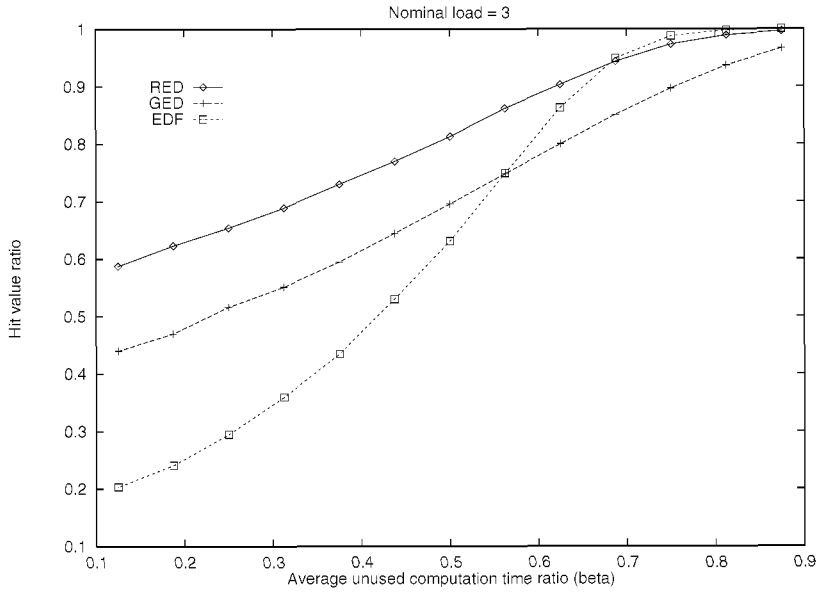


Figure 2.7 Performance of various EDF scheduling schemes: best-effort (EDF), guaranteed (GED) and robust (RED).

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

In the second experiment, D_{over} is compared against two robust algorithms: RED (Robust Earliest Deadline) and RHD (Robust High Density). In RHD, the task with the highest value density (v_i/C_i) is scheduled first, regardless of its deadline. Performance results are shown in Figure 2.8.

Notice that in underload conditions D_{over} and RED exhibit optimal behavior ($HVR = 1$), whereas RHD is not able to achieve the total cumulative value, since it does not take

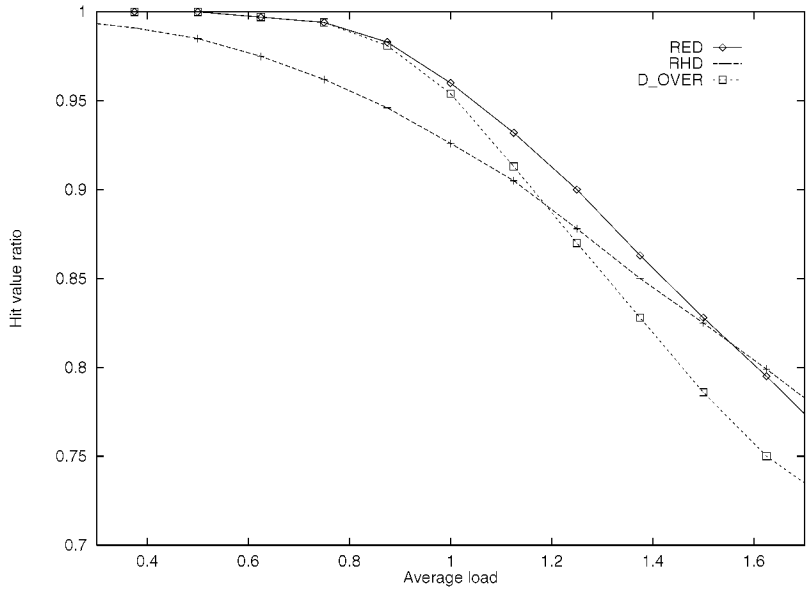


Figure 2.8 Performance of D_{over} against RED and RHD.

deadlines into account. However, for high load conditions ($\rho > 1.5$), RHD performs even better than RED and D_{over}.

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

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

2.4 PERFORMANCE DEGRADATION METHODS

According to this approach, the incoming activities that cause an overload are not rejected to preserve the active tasks; rather, the system load is reduced to accommodate them. Load reduction can be achieved at the price of a performance degradation through the following methods:

1. **Service adaptation**. The load is controlled by reducing the computation times of application tasks, thus affecting the quality of results.
2. **Job skipping**. The load is reduced by aborting entire task instances, but still guaranteeing that a minimum percentage of jobs is executed within the specified constraints.
3. **Period adaptation**. The load is reduced by relaxing timing constraints, thus allowing tasks to specify a range of values.

2.5 SERVICE ADAPTATION

A possible method for reducing the service time in overload conditions is to trade precision with computation time. The concept of imprecise and approximate computation has emerged as a new approach to increasing flexibility in dynamic scheduling by trading off computation accuracy with timing requirements [Nat95, LNL87, LLN87, LLS+91, LSL+94]. In dynamic situations, where time and resources are not enough for computations to complete within the deadline, there may still be enough resources to produce approximate results. The idea of using partial results when exact results cannot be produced within the deadline has been used for many years. Recently, however, this concept has been formalized, and specific techniques have been developed for designing programs that can produce partial results. Examples of applications that can exploit this technique include optimization methods (e.g., simulated annealing), cost-based heuristic searches, neural learning, and graphics activities.

In a real-time system that supports imprecise computation, every task J_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 J_i; however, O_i becomes ready for execution when M_i is completed. If C_i is the computation time associated with J_i, 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 noticing 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 J_i (or simply the error of J_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 J_i is equal to

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

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

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

where w_i is the relative importance of J_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 J_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 in the interval $[a_i, d_i]$. 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 in the interval $[a_i, d_i]$.

As an illustrative example, let us consider the task set shown in Figure 2.9a. Notice 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 2.9b. 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 $\overline{\epsilon} = 4$.

When an application task cannot be split into a mandatory and an optional part that can be aborted at any time, service adaptation can still be performed, at a coarse granularity, if multiple versions are provided for the task, each having different length and quality (the longer the task, the higher the quality). In this case, to cope with an overload condition, a high-quality version of a task may be replaced with a shorter one with lower quality. If all the tasks follow such a model, the objective of the system would be to maximize the overall quality under feasibility constraints.

Task	r_i	d_i	C_i	m_i	o_i
τ_1	0	6	4	2	2
τ_2	2	7	4	1	3
τ_3	4	10	5	2	3
τ_4	12	15	3	1	2
τ_5	6	20	8	5	3

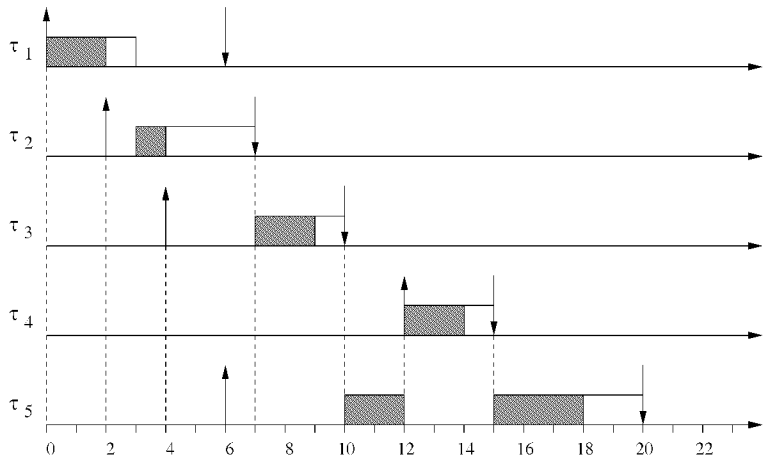


Figure 2.9 An example of an imprecise schedule.

2.6 JOB SKIPPING

Classical real-time scheduling theory ([LL73, SRL90, Bak91]) assumes that periodic tasks have hard timing constraints, meaning that all the instances of a periodic task must be guaranteed to complete within their deadlines. While such task model is suitable for critical control applications, it can be too restrictive for other applications: for example, in multimedia communication systems, skipping a video frame once in a while is acceptable to cope with transient overload situations. Even in more critical control applications, hard real-time tasks usually coexist with firm and soft real-time activities, which need to be treated in a different manner in order to optimize the available resources.

Permitting skips in periodic tasks increases system flexibility, since it allows to make a better use of resources and to schedule systems that would otherwise be overloaded. Consider for example two periodic tasks, τ_1 and τ_2, with periods $p_1 = 10$, $p_2 = 100$, and execution times $C_1 = 5$ and $C_2 = 55$, where τ_1 can skip an instance every 10 periods, whereas τ_2 is hard (i.e., no instances can be skipped). Clearly, the two tasks cannot be both scheduled as hard tasks, because the processor utilization factor is $U = 5/10 + 55/100 = 1.05 > 1$. However, if τ_1 is permitted to skip one instance every 10 periods, the spare time can be used to accommodate the execution of τ_2.

The *job skipping* model has been originally proposed by Koren and Shasha [KS95]. In their work, the authors showed that making optimal use of skips is NP-hard and presented two algorithms, called Skip-Over Algorithms (one a variant of rate monotonic scheduling and one of earliest deadline first) that exploit skips to increase the feasible periodic load and schedule slightly overloaded systems. According to the job skipping model, the maximum number of skips for each task is controlled by a specific parameter associated with the task. In particular, each periodic task τ_i is characterized by a worst-case computation time c_i, a period p_i, a relative deadline equal to its period, and a skip parameter s_i, $2 \leq s_i \leq \infty$, which gives 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 in [KS95], every instance of a periodic task with skips can be *red* or *blue*. A red instance must complete before its deadline; a blue instance can be aborted at any time. When a blue instance is aborted, we say that it was *skipped*. The fact that $s \geq 2$ implies that, if a blue instance is skipped, then the next $s - 1$ instances must be red. On the other hand, if a blue instance completes successfully, the next task instance is also blue.

Two on-line scheduling algorithms were proposed in [KS95] to handle tasks with skips under EDF.

1. The first algorithm, called *Red Tasks Only* (RTO), always rejects the blue instances, whereas the red ones are scheduled according to EDF.
2. The second algorithm, called *Blue When Possible* (BWP), is more flexible than RTO and schedules blue instances whenever there are no ready red jobs to execute. Red instances are scheduled according to EDF.

It is easy to find examples that show that BWP improves RTO in the sense that it is able to schedule task sets that RTO cannot schedule. In the general case, the above algorithms are not optimal, but they become optimal under a particular task model, called the *deeply-red* model.

Definition 2.2 *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.*

In the same paper, Koren and Shasha showed that the worst case for a periodic skippable task set occurs when tasks are deeply-red, hence all the results are derived under this assumption. This means that, if a task set is schedulable under the deeply-red model, it is also schedulable without this assumption.

In the hard periodic model in which all task instances are red (i.e., no skips are permitted), the schedulability of a periodic task set can be tested using a simple necessary and sufficient condition based upon cumulative processor utilization. In particular, Liu and Layland [LL73] showed that a periodic task set is schedulable by EDF if and only if its cumulative processor utilization is no greater than 1. Analyzing the feasibility of firm periodic tasks is not equally easy. Koren and Shasha proved that determining whether a set of skippable periodic tasks is schedulable is NP-hard. They also found that, given a set $\Gamma = \{T_i(p_i, c_i, s_i)\}$ of firm periodic tasks that allow skips, then

$$\sum_{i=1}^{n} \frac{c_i(s_i - 1)}{p_i s_i} \leq 1 \tag{2.2}$$

is a necessary condition for the feasibility of Γ, since it represents the utilization based on the computation that must take place.

To better clarify the concepts mentioned above consider the task set shown in Figure 2.10 and the corresponding feasible schedule, obtained by EDF. Notice that the

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

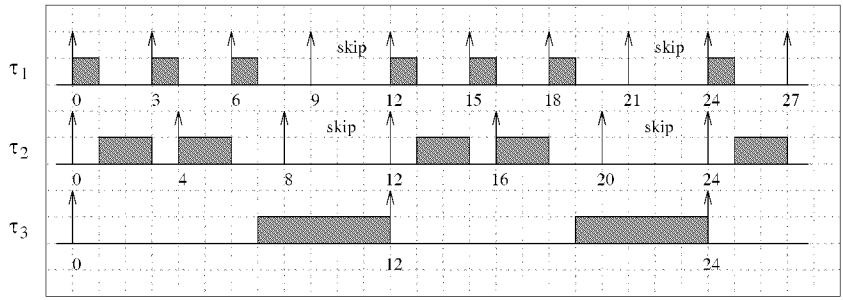


Figure 2.10 A schedulable set of firm periodic tasks.

processor utilization factor is greater than 1 ($U_p = 1.25$), but condition (2.2) is satisfied.

In the same work, Koren and Shasha proved the following theorem, which provides a sufficient condition for guaranteeing a set of skippable periodic tasks under EDF.

Theorem 2.2 *A set of firm (i.e., skippable) periodic tasks is schedulable if*

$$\forall L \geq 0 \qquad L \geq \sum_{i=1}^{n} D(i, [0, L]) \tag{2.3}$$

where

$$D(i, [0, L]) = \left(\left\lfloor \frac{L}{p_i} \right\rfloor - \left\lfloor \frac{L}{p_i s_i} \right\rfloor \right) c_i. \tag{2.4}$$

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. In [CB97], Caccamo and Buttazzo generalized the results of Theorem 2.2 by identifying the amount of bandwidth saving achieved by skips. To express this fact with a simple parameter, they defined an *equivalent utilization factor* U_p^* for periodic tasks with skips.

Definition 2.3 *Given a set* $\Gamma = \{T_i(p_i, c_i, s_i)\}$ *of* n *periodic tasks that allow skips, an* equivalent processor utilization factor *can be defined as:*

$$U_p^* = \max_{L \geq 0} \left\{ \frac{\sum_i D(i, [0, L])}{L} \right\} \tag{2.5}$$

where

$$D(i, [0, L]) = \left(\left\lfloor \frac{L}{p_i} \right\rfloor - \left\lfloor \frac{L}{p_i s_i} \right\rfloor \right) c_i.$$

The following theorem ([CB97]) states the schedulability condition for a set of deeply-red skippable tasks.

Theorem 2.3 *A set* Γ *of skippable periodic tasks, which are deeply-red, is schedulable if and only if*

$$U_p^* \leq 1.$$

The bandwidth saved by skipping blue instances can easily be exploited by an aperiodic server (like CBS [AB98a] described in Chapter 3) to advance the execution of aperiodic tasks. The following theorem ([CB97]) provides a sufficient condition for guaranteeing a feasible schedule of a hybrid task set consisting of n firm periodic tasks and a number of soft aperiodic tasks handled by a server with bandwidth U_s.

Theorem 2.4 *Given a set of periodic tasks that allow skip with equivalent utilization* U_p^* *and a set of soft aperiodic tasks handled by a server with utilization factor* U_s*, the hybrid set is schedulable by RTO or BWP if:*

$$U_p^* + U_s \leq 1. \tag{2.6}$$

The sufficient condition of Theorem 2.4 is a 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 ($U_{skip} = U_p - U_p^*$) available for aperiodic service. The remaining portion of the spare time saved

by skips is discontinuous, and creates a kind of "holes" in the schedule, which cannot be used at any time, unfortunately. Whenever an aperiodic request falls into some hole, it can exploit a bandwidth greater than $1 - U_p^*$. Indeed, it is easy to find examples in which a periodic task set is schedulable by assigning the aperiodic server a bandwidth U_s greater than $1 - U_p^*$. The following theorem ([CB97]) gives a maximum bandwidth $U_{s_{max}}$ above which the schedule is certainly not feasible.

Theorem 2.5 *Given a set $\Gamma = \{T_i(p_i, c_i, s_i)\}$ 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

$$U_{s_{max}} = 1 - U_p + \sum_{i=1}^{n} \frac{c_i}{p_i s_i}. \tag{2.7}$$

AN EXAMPLE

As an illustrative example, let us consider the task set shown in Table2.1. The task set consists of two periodic tasks, τ_1 and τ_2, with periods 3 and 5, computation times 2 and 2, and skip parameters 2 and ∞ respectively. The equivalent utilization factor of the periodic task set is $U_p^* = 4/5$ while $U_{s_{max}} = 0.27$, leaving a bandwidth of $U_s = 1/5$ 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^*	0.8	
$1 - U_p^*$	0.2	
$U_{s_{max}}$	0.27	

Table 2.1 A schedulable task set.

Supposing that the aperiodic activities are scheduled by a CBS server with maximum budget $Q^s = 1$ and server period $P^s = 5$, Figure 2.11 shows the resulting schedule

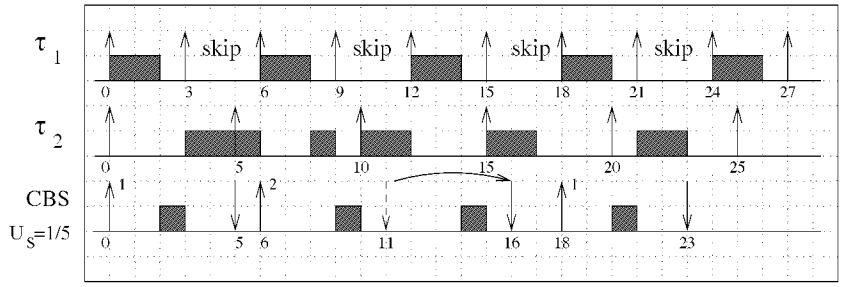


Figure 2.11 Schedule produced by RTO+CBS for the task set shown in Table 2.1.

by using RTO+CBS. Notice that J_2 has a deadline postponement (according to CBS rules) at time $t = 10$ with new server deadline $d^s_{new} = d^s_{old} + P^s = 11 + 5 = 16$. According to the sufficient schedulability test provided by Theorem 2.4, the task set is schedulable when the aperiodic server has assigned a bandwidth $U_s = 1 - U^*_p$.

2.7 PERIOD ADAPTATION

In a periodic task set, processor utilization can also be changed by varying task periods. Whenever the system load becomes greater than a maximum threshold, the periods can be enlarged in a proper way to bring the system load back to the desired value. Today, there are many real-time applications in which timing constraints are not rigid, but can be varied to better react to transient overload conditions.

For example, in multimedia systems, activities such as voice sampling, image acquisition, sound generation, data compression, and video playing, are performed periodically, but their execution rates are not as rigid as in control applications. Missing a deadline while displaying an MPEG video may decrease the quality of service (QoS), but does not cause critical system faults. Depending on the requested QoS, tasks may increase or decrease their execution rate to accommodate the requirements of other concurrent activities.

Even in some control application, there are situations in which periodic tasks could be executed at different rates in different operating conditions. 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 robotic applications in which robots have to work in unknown environments

where trajectories are planned based on the current sensory information. If a robot is equipped with proximity sensors, in order to maintain a desired performance, the acquisition rate of the sensors must increase whenever the robot is approaching an obstacle.

In other situations, the possibility of varying tasks' rates 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 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.

Unfortunately, there is no uniform approach for dealing with these situations. 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. In [NT94], Nakajima and Tezuka 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. In [SLSS97], Seto et al. 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 on-line load adjustment. In [LRM96], Lee, Rajkumar and Mercer propose a number of policies to dynamically adjust the tasks' rates in overload conditions. In [AAS97], Abdelzaher, Atkins, and Shin 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. In [Nak98a, Nak98b], Nakajima 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. In [BCRZ99], Beccari et al. 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 was originally introduced by Buttazzo et al. [BAL98] and then extended to a more general case [BLCA02]). It provides a novel theoretical framework for flexible workload management in real-time applications. In particular, the elastic approach provides the following advantages with respect to the classical "fixed-rate" approach.

- it allows tasks to intentionally change their execution rate to provide different quality of service;
- it can handle overload situations in a more flexible fashion;
- it provides a simple and efficient method for controlling the system's performance as a function of the current workload.

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

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

Now, suppose that a new periodic 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

$$U_p = \frac{10}{20} + \frac{10}{40} + \frac{15}{50} + \frac{5}{30} = 1.131 > 1.$$

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 task τ_3 reduces its period to 50, no feasible schedule exists, since the utilization would be greater than 1:

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

However, notice that a feasible schedule exists ($U_p = 0.977$) 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, notice 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 which 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 the all possible solutions.

2.7.1 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 exists a feasible schedule in which all the other periods are within their range.

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 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 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 $L = \sum_{i=1}^{n} x_i$.

Under the elastic model, given set of n periodic tasks with $U_p > U_{max}$, the objective of the guarantee is to compress tasks' utilization factors in order to achieve a new desired utilization $U_d \leq U_{max}$ such that all the periods are within their ranges. In the linear spring system, this is equivalent of 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 $L_0 = \sum_{i=1}^{n} x_{i_0} > L_{max}$, find a set of new lengths x_i such that $x_i \geq x_{i_{min}}$ and $L = L_d$, where L_d is any arbitrary desired length such that $L_d < L_{max}$.

For the sake of clarity, we first solve the problem for a spring system without length constraints, 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 2.12. 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:

$$L_d = \sum_{i=1}^{n} x_i. \tag{2.8}$$

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

$$\forall i \quad x_i = x_{i_0} - \frac{F}{k_i}. \tag{2.9}$$

By summing equations (2.9) we have:

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

Thus, force F can be expressed as

$$F = K_p(L_0 - L_d), \tag{2.10}$$

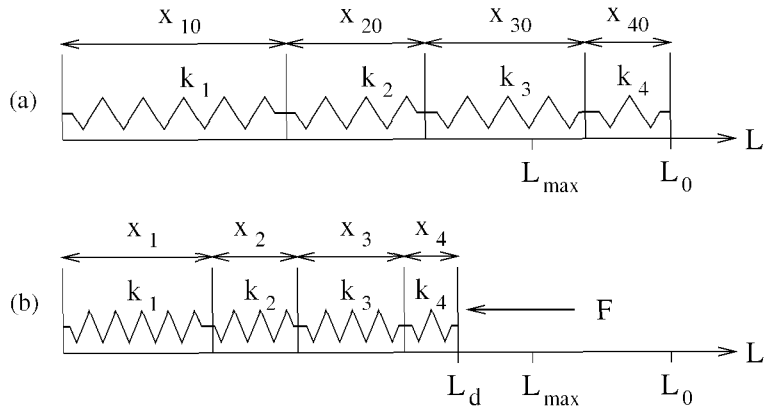


Figure 2.12 A linear spring system: the total length is L_0 when springs are uncompressed (a); and $L_d < L_0$ when springs are compressed by applying a force F (b).

where

$$K_p = \frac{1}{\sum_{i=1}^{n} \frac{1}{k_i}}. \tag{2.11}$$

Substituting expression (2.10) into Equations (2.9) we finally achieve:

$$\forall i \quad x_i = x_{i_0} - (L_0 - L_d)\frac{K_p}{k_i}. \tag{2.12}$$

Equation (2.12) allows us to compute how each spring has to be compressed in order to have a desired total length L_d.

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. 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 (2.12) to the set Γ_v of variable springs, we have

$$\forall S_i \in \Gamma_v \quad x_i = x_{i_0} - (L_{v_0} - L_d + L_f)\frac{K_v}{k_i} \tag{2.13}$$

where

$$L_{v_0} = \sum_{S_i \in \Gamma_v} x_{i_0} \tag{2.14}$$

$$L_f = \sum_{S_i \in \Gamma_f} x_{i_{min}} \tag{2.15}$$

$$K_v = \frac{1}{\sum_{S_i \in \Gamma_v} \frac{1}{k_i}}. \tag{2.16}$$

Whenever there exists some spring for which equation (2.13) 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 (2.13), (2.14), (2.15) and (2.16) recomputed for the new set Γ_v. If there exists a feasible solution, that is, if the desired final length L_d is greater than or equal to the minimum possible length of the array $L_{min} = \sum_{i=1}^{n} x_{i_{min}}$, the iterative process ends when each value computed by equations (2.13) is greater than or equal to its corresponding minimum $x_{i_{min}}$. The complete algorithm for compressing a set Γ of n springs with length constraints up to a desired length L_d is shown in Figure 2.13.

COMPRESSING TASKS' UTILIZATIONS

When dealing with a set of elastic tasks, equations (2.13), (2.14), (2.15) and (2.16) 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 of periodic tasks 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:

$$\forall \tau_i \in \Gamma_v \quad U_i = U_{i_0} - (U_{v_0} - U_d + U_f)\frac{E_i}{E_v} \tag{2.17}$$

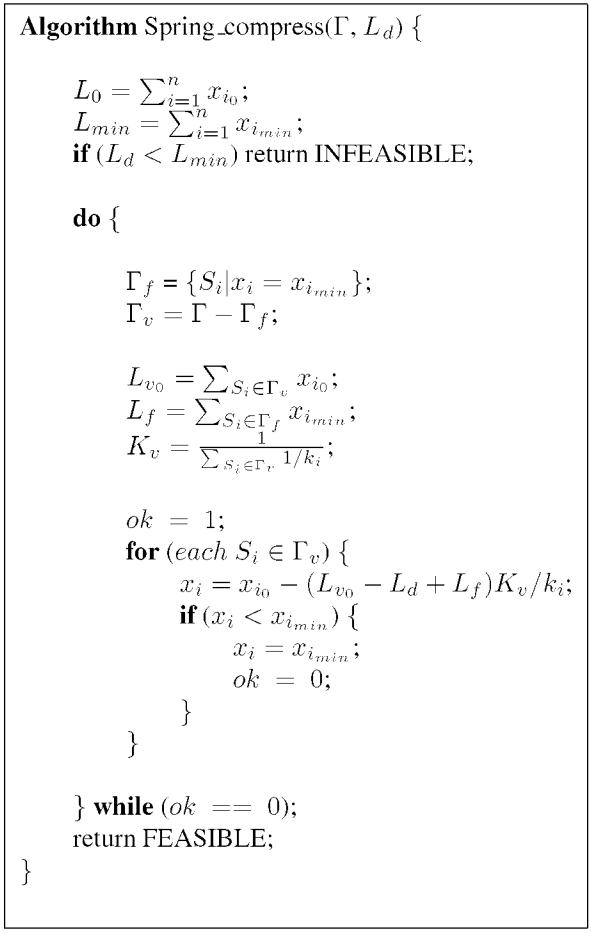
where

$$U_{v_0} = \sum_{\tau_i \in \Gamma_v} U_{i_0} \tag{2.18}$$

$$U_f = \sum_{\tau_i \in \Gamma_f} U_{i_{min}} \tag{2.19}$$

$$E_v = \sum_{\tau_i \in \Gamma_v} E_i. \tag{2.20}$$

If there exist 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 (2.17) applied again to the tasks in



Algorithm Spring_compress(Γ, L_d) {

$L_0 = \sum_{i=1}^{n} x_{i_0}$;
$L_{min} = \sum_{i=1}^{n} x_{i_{min}}$;
if ($L_d < L_{min}$) return INFEASIBLE;

do {

$\Gamma_f = \{S_i | x_i = x_{i_{min}}\}$;
$\Gamma_v = \Gamma - \Gamma_f$;

$L_{v_0} = \sum_{S_i \in \Gamma_v} x_{i_0}$;
$L_f = \sum_{S_i \in \Gamma_f} x_{i_{min}}$;
$K_v = \frac{1}{\sum_{S_i \in \Gamma_v} 1/k_i}$;

$ok = 1$;
for ($each\ S_i \in \Gamma_v$) {
$x_i = x_{i_0} - (L_{v_0} - L_d + L_f) K_v / k_i$;
if ($x_i < x_{i_{min}}$) {
$x_i = x_{i_{min}}$;
$ok = 0$;
}
}

} **while** ($ok == 0$);
return FEASIBLE;
}

Figure 2.13 Algorithm for compressing a set of springs with length constraints.

Γ_v. If there exists 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 (2.17) is greater than or equal to its corresponding minimum $U_{i_{min}}$. The algorithm[1] for compressing a set Γ of n elastic tasks up to a desired utilization U_d is shown in Figure 2.14.

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_{c_0} be the total utilization of tasks in Γ_c at their nominal periods. It can easily be seen that if $U_{c_0} + U_a \leq U_{lub}$ all tasks in Γ_c can return to their nominal periods. On the other hand, if $U_{c_0} + U_a > U_{lub}$, 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 (2.17), (2.19) and (2.20) and the algorithm presented in Figure 2.14.

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 $E_i = U_{i_0}$, equations (2.17) become:

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

from which we have that

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

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

```
Algorithm Task_compress(Γ, U_d) {

  U_0 = Σ_{i=1}^{n} C_i/T_{i_0};
  U_min = Σ_{i=1}^{n} C_i/T_{i_max};
  if (U_d < U_min) return INFEASIBLE;

  do {

      U_f = U_{v_0} = E_v = 0;
      for (each τ_i) {
          if ((E_i == 0) or (T_i == T_{i_max}))
              U_f = U_f + U_{i_min};
          else {
              E_v = E_v + E_i;
              U_{v_0} = U_{v_0} + U_{i_0}
          }
      }

      ok = 1;
      for (each τ_i ∈ Γ_v) {
          if ((E_i > 0) and (T_i < T_{i_max})) {
              U_i = U_{i_0} − (U_{v_0} − U_d + U_f)E_i/E_v;
              T_i = C_i/U_i;
              if (T_i > T_{i_max}) {
                  T_i = T_{i_max};
                  ok = 0;
              }
          }
      }

  } while (ok == 0);
  return FEASIBLE;
}
```

Figure 2.14 Algorithm for compressing a set of elastic tasks.

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

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

Notice 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 on-line 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 loose 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.

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 (for example, based on task importance). Each task is varied based on its current elastic status and a feasible configuration is found, if there exists one.

The elastic model is extremely 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 [But93a, LLB+97].

It is worth observing that the elastic approach is not limited to task scheduling. Rather, it represents a general resource allocation methodology which 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 in [PGBA02].

Another interesting application of the elastic approach is to automatically adapt the task rates to the current load, without specifying the worst-case execution times of the tasks. 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 grater than one and some deadline misses are detected, task rates can be reduced to bring the processor utilization at a desired safe value.

The elastic model has also been extended in [BLCA02] to deal with resource constraints, 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].