7

Handling Overload Conditions in Real-Time Systems

Giorgio C. Buttazzo
Scuola Superiore Sant'Anna
Italy

1. Introduction

This chapter deals with the problem of handling overload conditions, that is, those critical situations in which the computational demand requested by the application exceeds the processor capacity (Buttazzo, 2011). If not properly handled, an overload can cause an abrupt performance degradation, or even a system crash. Therefore, a real-time system should be designed to anticipate and tolerate unexpected overload situations through specific kernel mechanisms.

Overload conditions can occur for different causes, including bad system design, simultaneous arrival of events, operating system exceptions, malfunctioning of input devices, and unpredicted variations of the environmental conditions.

In the following, we consider a set of n periodic or sporadic tasks, $\Gamma = \{\tau_1, \ldots, \tau_n\}$, each characterized by a worst-case execution time (WCET) C_i, a relative deadline D_i, and a period (or minimum inter-arrival time) T_i. Each task τ_i is initially activated at time Φ_i (denoted as the task phase) and generates an infinite sequence of jobs $\tau_{i,k}$ ($k = 1, 2, \ldots$). If a task τ_i is periodic, a generic job $\tau_{i,k}$ is regularly activated at time $r_{i,k} = \Phi_i + (k-1)T_i$. In general, the activation time of job $\tau_{i,k+1}$ is:

$$\begin{cases} r_{i,k+1} = r_{i,k} + T_i & \text{if } \tau_i \text{ is periodic} \\ r_{i,k+1} \geq r_{i,k} + T_i & \text{if } \tau_i \text{ is sporadic} \\ r_{i,k+1} > r_{i,k} & \text{if } \tau_i \text{ is aperiodic.} \end{cases}$$

Also, each job $\tau_{i,k}$ is characterized by an absolute deadline $d_{i,k} = r_{i,k} + D_i$. For a set of periodic tasks, the *hyperperiod* H denotes the minimum interval of time after which the schedule repeats itself. For a set of periodic tasks synchronously activated at time $t = 0$ ($\Phi_i = 0$, for all i), the hyperperiod is equal to the least common multiple of all the periods, that is $H = \text{lcm}(T_1, \ldots, T_n)$.

In a real-time system, the computational load depends on the temporal characteristics of the executing activities. For example, for a set of n periodic tasks, the system load is equivalent to the processor utilization factor (Liu & Layland, 1973):

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

A value $U > 1$ means that the total computation time requested by the task set in the *hyperperiod* exceeds the available time on the processor (i.e, the length H); 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 current time and the job deadlines. In general, if there are n active jobs at time t, with absolute deadlines $d_1, d_2, \ldots, d_n$, the *instantaneous load* $\rho(t)$ can be defined as follows (Buttazzo & Stankovic, 1995):

$$\rho(t) = \max_i \left\{ \frac{\sum_{d_k \le d_i} c_k(t)}{d_i - t} \right\}, \tag{2}$$

where $c_k(t)$ denotes the remaining worst-case computation time of the k-th job. Figure 1 shows how the instantaneous load varies as a function of time for a set of three real-time jobs $\{J_1, J_2, J_3\}$ having activation times (r_i: 3, 1, 2), computation times (C_i: 2, 3, 1), and relative deadlines (D_i: 3, 6, 7).

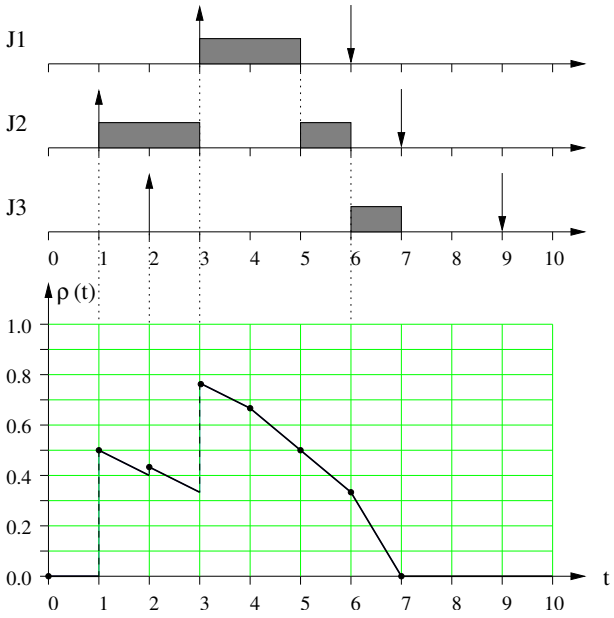

Fig. 1. Instantaneous load as a function of time for a set of three real-time jobs.

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

- 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.
- A task is said to experience an **overrun** when it exceeds 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*).

Note that, while the overload is a condition related to the processor, the overrun is a condition related to a single job. A job overrun does not necessarily cause an overload. However, a large unexpected overrun or a sequence of overruns on multiple jobs can cause very unpredictable effects on the system, if not properly handled.

In this chapter, two types of overload conditions will be analyzed:

- **Transient overload due to task overruns**. This type of overload is due to periodic or aperiodic tasks that sporadically execute (or are activated) more than expected. Under fixed priority scheduling, an overrun in a task τ_i does not affect tasks with higher priority, but any of the lower priority task could miss its deadline. Under the Earliest Deadline First (EDF) scheduling algorithm (Liu & Layland, 1973), a task overrun can potentially affect all the other tasks in the system. Figure 2 shows an example of an execution overrun under EDF scheduling. In this example, task τ_3 experiences an overrun of 7 units of time (shown in light gray), since its expected execution time was $C_3 = 3$.

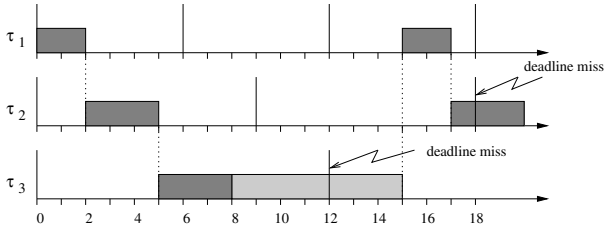

Fig. 2. Effect of an execution overrun in an EDF schedule.

- **Permanent overload in periodic task systems**. This type of overload occurs when the total utilization 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 because of some unexpected activation of new periodic tasks, or because some of the current tasks increased their activation rate to react to some change in the environment. In such a situation, tasks start accumulating in the system's queues (which tend to become longer and longer, if the overload persists), and their response times tend to increase indefinitely. Figure 3 shows the effect of a permanent overload condition in a Rate Monotonic schedule, where computation times are C_i: (2, 3, 2), and periods are T_i: (4, 6, 8). Note that, since $U_p = 1.25$, τ_2 misses its deadline and τ_3 can never execute.

2. Handling transient overloads

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.

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

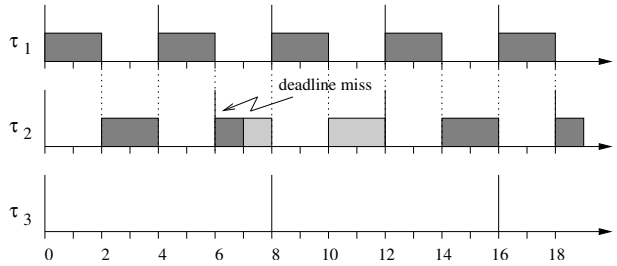

Fig. 3. Example of a permanent overload under Rate Monotonic: τ_2 misses its deadline and τ_3 can never execute.

other tasks can be tuned acting on the priority assigned to the "faulty" task for executing the remaining computation. Such a solution can be efficiently implemented through the resource reservation approach, which is a general kernel technique for limiting the inter-task interference and isolating the temporal behavior of a task subset.

2.1 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. 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 to consume 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 resource reservation technique for fixed priority scheduling was first presented by Mercer, Savage and Tokuda (Mercer et al., 1994). According to this method, a task τ_i is handled by a server, which is a kernel mechanism capable of controlling the execution of the task assigned to it through a pair of parameters (Q_s, P_s) (denoted as a CPU *capacity reserve*). The server enables τ_i to execute for Q_s units of time every P_s. In this case, the bandwidth reserved to the task is $U_s = Q_s/P_s$. When the task consumes its reserved quantum Q_s, 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_s 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, the execution of τ_i is *reshaped* to be more uniform along the timeline, so avoiding long intervals of time in which τ_i prevents other tasks to run.

Under EDF scheduling, resource reservation can be efficiently implemented through the Constant Bandwidth Server (CBS) (Abeni & Buttazzo, 1998; 2004), which is a service mechanism also controlled by two parameters, (Q_s, P_s), where Q_s is the *server maximum budget* and P_s is the *server period*. The ratio $U_s = Q_s/P_s$ is denoted as the *server bandwidth*. At each instant, two state variables are maintained: the server deadline d_s and the actual server budget q_s. Each job handled by a server is scheduled using the current server deadline and whenever the server executes a job, the budget q_s is decreased by the same amount. At the beginning $d_s = q_s = 0$. Since a job is not activated while the previous one is active, the CBS algorithm can be formally defined as follows:

1. When a job $\tau_{i,j}$ arrives, if $q_s \geq (d_s - r_{i,j})U_s$, it is assigned a server deadline $d_s = r_{i,j} + P_s$ and q_s is recharged at the maximum value Q_s, otherwise the job is served with the current deadline using the current budget.
2. When $q_s = 0$, the server budget is recharged at the maximum value Q_s and the server deadline is postponed at $d_s = d_s + P_s$. Note that there are no finite intervals of time in which the budget is equal to zero.

As shown in (Abeni & Buttazzo, 2004), if a task τ_i is handled by a CBS with bandwidth U_s, it will never demand more than U_s, independently of the actual execution time of its jobs. As a consequence, possible overruns occurring in the served task do not create extra interference in the other tasks, but only affect τ_i.

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 4 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) with utilization $U_p = 0.5$ 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 5.

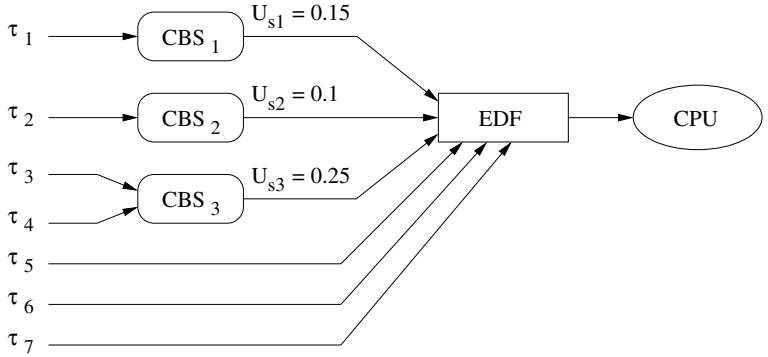

Fig. 4. Achieving temporal protection using the CBS mechanism.

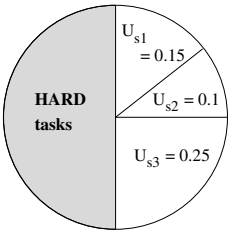

Fig. 5. Bandwidth allocation for a set of task.

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

$$U_p + U_{s_1} + U_{s_2} + U_{s_3} \leq 1. \tag{3}$$

Note that if condition (3) 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.

2.2 Schedulability analysis

Although a reservation R_k 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. (Mok et al., 2001) introduced the concept of *bounded delay partition* that describes a reservation R_k 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* (Lipari & Bini, 2003; Shin & Lee, 2003), representing the minimum amount of time the resource can provide in a given interval of time.

Definition 1. *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 (Feng & Mok, 2002; Mok et al., 2001), periodic servers (Lipari & Bini, 2003; Shin & Lee, 2003), or periodic servers with arbitrary deadline (Easwaran et al., 2007). Consider, for example, that processing time is provided only in the intervals [0,3], [6,8], and [9,10], 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 6.

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*:

$$\mathsf{sbf}_k(t) \stackrel{\text{def}}{=} \max\{0,\ \alpha_k(t - \Delta_k)\}. \tag{4}$$

represented by the dashed line in Figure 6. 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. 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:

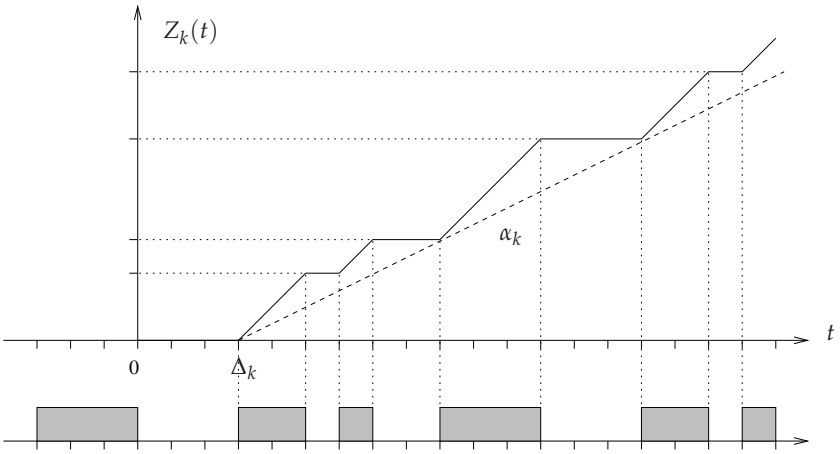

Fig. 6. A reservation implemented by a static partition of intervals.

$$\alpha_k = \lim_{t\to\infty} \frac{Z_k(t)}{t} \tag{5}$$

$$\Delta_k = \sup_{t\geq 0} \left\{ t - \frac{Z_k(t)}{\alpha_k} \right\}. \tag{6}$$

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 7, where

$$\alpha_k = Q_k/T_k \tag{7}$$

$$\Delta_k = 2(T_k - Q_k). \tag{8}$$

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 R_k is

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

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

$$Q_k = \frac{\alpha_k \Delta_k}{2(1-\alpha_k)}$$

$$P_k = \frac{\Delta_k}{2(1-\alpha_k)}.$$

Hence,

$$\alpha_k^{\text{eff}} = \alpha_k + \frac{2\sigma(1-\alpha_k)}{\Delta_k}. \tag{9}$$

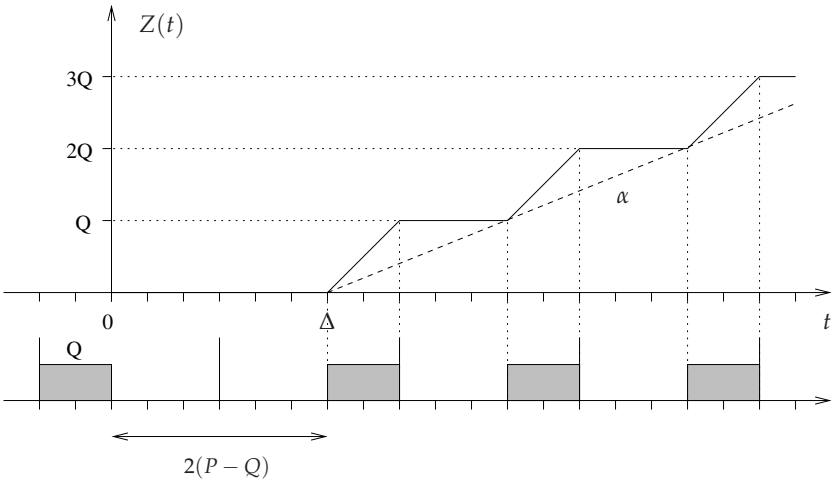

Fig. 7. A reservation implemented by a periodic server.

Within a reservation, the schedulability analysis of a task set under fixed priorities can be performed through the following Theorem (Bini et al., 2009):

Theorem 1 (Bini et al., 2009)**.** *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*

$$\forall i = 1, \ldots, n \quad \exists t \in (0, D_i] : W_i(t) \leq Z_k(t). \tag{10}$$

where $W_i(t)$ represents the Level-i workload, computed as follows:

$$W_i(t) = C_i + \sum_{h:P_h > P_i} \left\lceil \frac{t}{T_h} \right\rceil C_h. \tag{11}$$

Similarly, the schedulability analysis of a task set under EDF can be performed using the following theorem (Bini et al., 2009):

Theorem 2 (Bini et al., 2009)**.** *A set of preemptive periodic tasks with utilization U_p and 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 $U_p < \alpha_k$ and*

$$\forall t > 0 \quad \mathsf{dbf}(t) \leq Z_k(t). \tag{12}$$

where $\mathsf{dbf}(t)$ is the Demand Bound Function (Baruah et al., 1990) defined as

$$\mathsf{dbf}(t) \stackrel{\text{def}}{=} \sum_{i=1}^{n} \left\lfloor \frac{t + T_i - D_i}{T_i} \right\rfloor C_i. \tag{13}$$

In the specific case in which $Z_k(t)$ is lower bounded by the supply bound function, the test becomes only sufficient and the set of testing points can be better restricted as stated in the following theorem (Bertogna et al., 2009):

Theorem 3 (Bertogna et al., 2009). *A set of preemptive periodic tasks with utilization U_p and 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 $U_p < \alpha_k$ and*

$$\forall t \in \mathcal{D} \quad \mathrm{dbf}(t) \leq \max[0, \alpha_k(t - \Delta_k)]. \tag{14}$$

where

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

and

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

2.3 Handling wrong reservations

Although resource reservation is essential for achieving predictability in the presence of tasks with variable execution times, the overall system performance becomes quite dependent on a correct bandwidth allocation. In fact, 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. 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 (Bernat et al., December 5-8, 2004; Bernat & Burns, 2002) and dynamic priority servers (Caccamo et al., 2000). For example, the CASH algorithm (Caccamo et al., 2005) 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 (Lamastra et al., December 3-6, 2001) 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 (Marzario et al., 2004) enhances CBS with fairer slack reclaiming, so slack is not reclaimed until all current jobs have been serviced and the processor is idle. BACKSLASH (Lin & Brandt, December 5Ű8, 2005) 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 (Abeni & Buttazzo, May 30 - June 1, 2001) and by Palopoli et al. (Palopoli et al., December 3-5, 2002).

3. 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.

3.1 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 8 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.

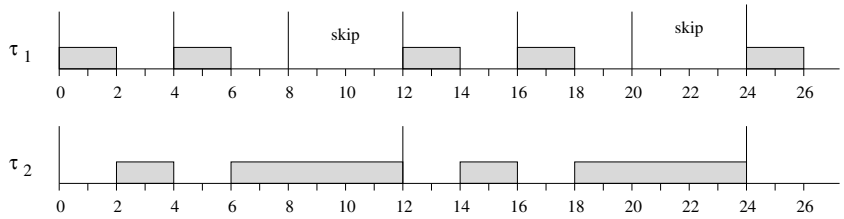

Fig. 8. Overload condition resolved by skipping one job every three in task τ_1.

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 (Koren & Shasha, 1995) proposed a new task model (known as the *firm* periodic model) suited to be handled by this technique. 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, $2 \leq S_i \leq \infty$, 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_i, the better the quality of service).

Using the terminology introduced by Koren and Shasha (Koren & Shasha, 1995), 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 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.*

Koren and Shasha showed that the worst case for a periodic skippable task set occurs when tasks are deeply-red. For this reason, the feasibility tests are derived under this condition, so that, if a task set is schedulable under the deeply-red condition, it is also schedulable in any other situation.

3.1.1 Schedulability analysis

The feasibility analysis of a set of firm tasks can be performed through the Processor Demand Criterion (Baruah et al., 1990) under the deeply-red condition, assuming that in the worst case all blue jobs are aborted. In such a worst-case scenario, the processor demand of τ_i due to the red jobs in an interval $[0, t]$ can be obtained as the difference between the demand of all the jobs and the demand of the blue jobs:

$$\mathsf{dbf}_i^{skip}(t) = \left(\left\lfloor \frac{t}{T_i} \right\rfloor - \left\lfloor \frac{t}{T_i S_i} \right\rfloor \right) C_i. \tag{15}$$

Hence, the feasibility of the task set can be verified through the following theorem.

Theorem 4 (Koren and Shasha, 1995)**.** *A set of firm periodic tasks is schedulable by EDF if*

$$\forall t \geq 0 \qquad \sum_{i=1}^{n} \left(\left\lfloor \frac{t}{T_i} \right\rfloor - \left\lfloor \frac{t}{T_i S_i} \right\rfloor \right) C_i \;\leq\; t. \tag{16}$$

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.

Theorem 5 (Koren and Shasha, 1995)**.** *Necessary condition for the schedulability of a set of firm periodic tasks is that*

$$\sum_{i=1}^{n} \frac{C_i(S_i - 1)}{T_i S_i} \;\leq\; 1. \tag{17}$$

To better clarify the concepts mentioned above, consider the task set shown in Figure 9 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 (16) and (17) are satisfied.

Task	C_i	T_i	D_i	S_i
τ_1	1	3	3	4
τ_2	2	4	4	3
τ_3	5	12	12	∞

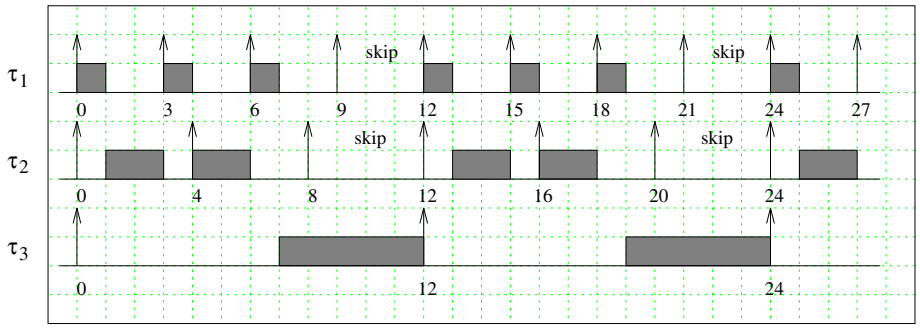

Fig. 9. A set of firm periodic tasks schedulable by EDF.

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 (Caccamo & Buttazzo, 1997) identified the amount of bandwidth saved by skips using a simple parameter, the *equivalent utilization factor* U_p^{skip}, which can be defined as

$$U_p^{skip} = \max_{t \geq 0} \left\{ \frac{\sum_i \mathsf{dbf}_i^{skip}(t)}{t} \right\} \tag{18}$$

where $\mathsf{dbf}_i^{skip}(t)$ is given in Equation (15).

Using this definition, the schedulability of a deeply-red skippable task set can be also verified using the following theorem (Caccamo & Buttazzo, 1997):

Theorem 6 (Caccamo and Buttazzo, 1997)**.** *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

(18) (setting $S_i = \infty$), U_p can also be defined as

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

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

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

The bandwidth saved by skips can also be exploited by an aperiodic server to advance the execution of aperiodic tasks.

3.2 Period adaptation

There are several real-time applications in which timing constraints are not rigid, but depend on the system state. 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, if the total utilization of the task set is greater than one, the system could reduce the utilizations of some tasks (by increasing their periods in a controlled fashion) to decrease the total load.

The elastic model presented in this section (originally introduced Buttazzo et al. (Buttazzo et al., 1998) and later extended by the same authors to deal with resource constraints (Buttazzo et al., 2002)), provides a novel theoretical framework for flexible workload management in real-time applications.

3.2.1 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 minimum period T_i^{min}, 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

$$\tau_i(C_i, T_i^{min}, T_i^{max}, E_i).$$

In the following, T_i denotes the actual period of task τ_i, which is constrained to be in the range $[T_i^{min}, T_i^{max}]$. 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.

It is worth noting that the elastic model is more general than the classical Liu and Layland's task model (Liu & Layland, 1973), so it does not prevent a user from defining hard real-time tasks. In fact, a task having $T_i^{max} = T_i^{min}$ 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.

Under the elastic model, given a set of n periodic tasks with utilization $U_p > 1$, the objective of the elastic guarantee is to compress tasks' utilization factors to achieve a new desired utilization $U_d \leq 1$ such that all the periods are within their ranges.

The following definitions are also used in this section:

$$U_i^{min} = C_i / T_i^{max};$$
$$U_{min} = \sum_{i=1}^{n} U_i^{min};$$
$$U_i^{max} = C_i / T_i^{min};$$
$$U_{max} = \sum_{i=1}^{n} U_i^{max}.$$

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

To understand how an elastic guarantee is performed in this model, it is convenient to compare an elastic task τ_i having utilization U_i and elasticity E_i with a linear spring S_i characterized by a length x_i and a rigidity coefficient k_i, equivalent to the inverse of the task's elasticity ($k_i = 1/E_i$). In this comparison, the nominal length x_{i_0} of the spring is equivalent to U_i^{max}, whereas the minimum length x_i^{min} is equivalent to U_i^{min}. 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 special case in which $U_i^{min} = 0$ for all tasks, the compressed task utilizations can be derived by solving a set of n spring linear equations, under the constraint that $\sum_{i=1}^{n} U_i = U_d$. The resulting expression is:

$$\forall i \quad U_i = U_i^{max} - (U_{max} - U_d) \frac{E_i}{E_s}. \tag{19}$$

where $E_s = \sum_{i=1}^{n} E_i$.

If each spring has a length constraint, in the sense that its length cannot be less than a minimum value x_i^{min}, 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 10.

Thus, at each instant, the set Γ can be divided into two subsets: a set Γ_f of fixed springs having minimum length (equivalent to tasks that reached their minimum utilization with the maximum period), and a set Γ_v of variable springs that can still be compressed. If U_v^{max} is the sum of the maximum utilizations of tasks in Γ_v, and U_f is the total utilization factor of tasks in Γ_f, then, to achieve a desired utilization $U_d \leq 1$, each task has to be compressed up to the following utilization:

$$\forall \tau_i \in \Gamma_v \quad U_i = U_i^{max} - (U_v^{max} - U_d + U_f) \frac{E_i}{E_v} \tag{20}$$

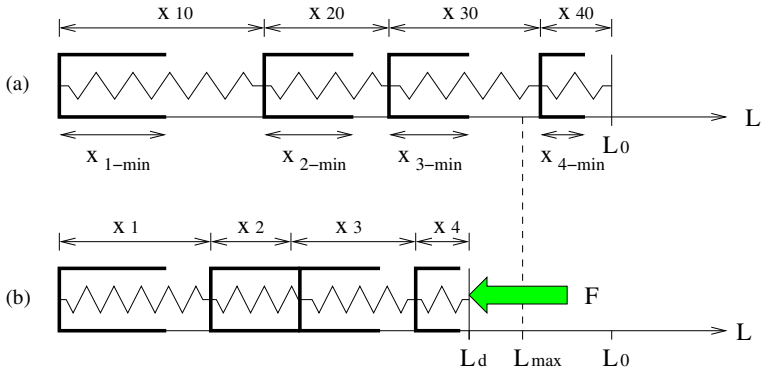

Fig. 10. Springs with minimum length constraints (a); during compression, spring S_2 reaches its minimum length and cannot be compressed any further (b).

where

$$U_v^{max} = \sum_{\tau_i \in \Gamma_v} U_i^{max} \tag{21}$$

$$U_f = \sum_{\tau_i \in \Gamma_f} U_i^{min} \tag{22}$$

$$E_v = \sum_{\tau_i \in \Gamma_v} E_i. \tag{23}$$

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 (20) applied again to the tasks in Γ_v. If there is a feasible solution, that is, if $U_{min} \leq U_d$, the iterative process ends when each value U_i computed by Equation (20) is greater than or equal to its corresponding minimum U_i^{min}. The algorithm for compressing a set Γ of n elastic tasks up to a desired utilization U_d is shown in Figure 11.

All tasks' utilizations that have been compressed to cope with an overload situation can return toward their nominal values when the overload is over.

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. In particular, the period of a task τ_i can be increased at any time, but can only be reduced at the next job activation. 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. (Buttazzo et al., 2002) and later improved by Guangming (Guangming, 2009).

3.2.2 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

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

begin

(1)  U_min := Σ_{i=1}^{n} C_i / T_i^max;
(2)  if (U_d < U_min) return(INFEASIBLE);
(3)  for (i := 1 to n) U_i^max := C_i / T_i^min;

(4)  do
(5)       U_f := 0; U_v^max := 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^max := U_v^max + U_i^max;
(12)           end
(13)      end

(14)      ok := 1;
(15)      for (each τ_i ∈ Γ_v) do
(16)           if ((E_i > 0) and (T_i < T_i^max)) then
(17)                U_i := U_i^max − (U_v^max − 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
```

Fig. 11. Algorithm for compressing a set of elastic tasks.

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 $U_v^{max} = U_{max}$. Under this assumption, by setting $E_i = U_i^{max}$, Equation (20) becomes:

$$\forall i \quad U_i = U_i^{max} - (U_{max} - U_d)\frac{U_i^{max}}{U_{max}} = \frac{U_i^{max}}{U_{max}}U_d$$

from which we have

$$T_i = T_i^{min}\frac{U_{max}}{U_d}. \tag{24}$$

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

$$\eta = \frac{U_{max}}{U_d}.$$

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

$$U = \sum_{i=1}^{n}\frac{C_i}{T_i} = \sum_{i=1}^{n}\frac{C_i}{\eta T_i^{min}} = \frac{1}{\eta}U_{max} = \frac{U_d}{U_{max}}U_{max} = U_d$$

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

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

By deciding to apply period rescaling, we lose the freedom of choosing the elastic coefficients, since they must be set equal to task maximum utilizations ($E_i = U_i^{max}$). 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.

3.3 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 (Lin et al., 1987; Liu et al., 1987; 1991; 1994;

Natarajan, 1995; Shih et al., 1991) 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 (Shih et al., 1989). Both subtasks have the same activation time r_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.

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. (Aydin et al., 2001), 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'.

4. Two case studies

This section describes two real-world applications to illustrate how the presented techniques can be used to prevent the negative effects of the overload. The first example considers a multimedia system and shows how resource reservation can isolate the timing behavior of concurrent applications characterized by highly variable execution times, preventing a performance degradation due to a reciprocal interference. The second example illustrates how to handle a permanent overload in a robot system that activates a new task to cope with obstacle avoidance.

4.1 Resource reservation in multimedia systems

Let us consider a multimedia device, like a cell phone, in which a phone call, a video player, and a web browser can be concurrently executed, so that a user can simultaneously make a phone call while watching a video and downloading a file from the web. These applications are characterized by a highly variable computational demand and share common resources (e.g., processor, memory, touch screen, audio codec, and graphic display). They are briefly described below.

- **Phone call** (A_1). This application consists at least of two periodic activities, executed with a period of 20 ms. A task is in charge of receiving the incoming audio signal, decoding it, and transferring the packets to the speaker buffer for reproduction. The processing time of this task can vary from 1 ms (during silence) up to 3 ms. The second task is responsible for sampling the voice from the microphone, performing data encoding, speech enhancement, and packet transmission through the modem. The processing time of this task can vary from 5 ms (during silence) up to 10 ms. Hence, overall, this application requires a processor bandwidth that can vary from 30% to 65%.
- **Video player** (A_2). The MPEG standard adopted for video compression is characterized by highly variable execution times. For instance, Figure 12 shows the typical distribution of frame decoding times for an MPEG player decoding a specific video on a given platform (Abeni & Buttazzo, 2000; Isovic et al., 2005). Note that the processing time of this task can vary from 6 ms to 30 ms, with an average decoding time of about 12 ms. If using the PAL standard, the frame rate is set to 25 frames per second, meaning that each frame has to be processed every 40 ms. Therefore, running an MPEG player requires an average processor bandwidth of 30%, which can reach 75% in peak load conditions.
- **Web browser** (A_3). This activity is also characterized by high load variations. In fact, when loading a web page, there is a peak processing load to parse the input, taking about one second. Then, a request is submitted to the server for sending a separate content, like images and layout files. At this time, the processing pauses for about another second while waiting for the new input. When full data arrive, there is another peak load, since they should be processed (decoded/parsed and rendered) as quickly as possible (this phase takes a few seconds depending on the content).

Note that, each of the considered applications is typically implemented as a set of tasks with different priorities. Hence, when they are concurrently executed on the same processor,

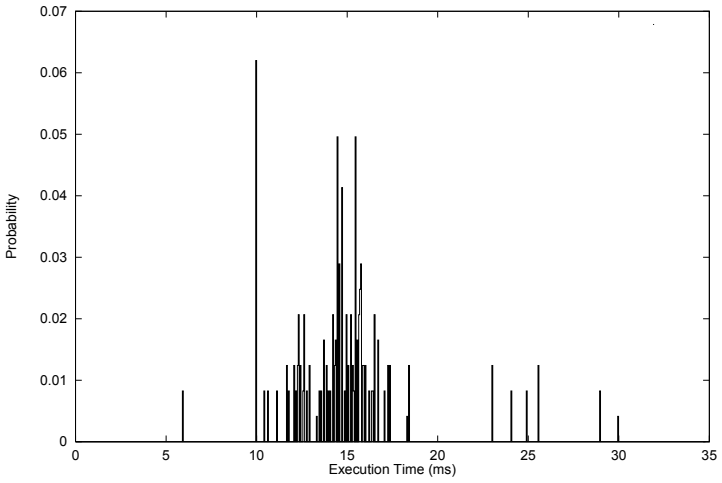

Fig. 12. Distribution of frame decoding times for an MPEG video player.

tasks are subject to reciprocal interference and can experience long blocking delays and jerky behavior. As a consequence, the user could experience a temporary motion stop on the movie, or perceive an annoying jitter in the sound. Although these applications are not safety critical, the unpleasant effects of such interferences on the user perception are taken in a serious consideration by the developers, since they can make a difference with another device produced by a competitor.

Resource reservation can be effectively used in this system to isolate the temporal behavior of the applications and limit their reciprocal interference (Bini et al., 2011). To do that, the processor should be partitioned into three reservations, with bandwidth U_{s_1}, U_{s_2}, and U_{s_3}, each behaving as a slower processor running at speed U_{s_i}. The advantage of this approach is that an overrun occurring in an application does not affect the other applications, but has only the effect of postponing the execution of those tasks in which the overrun is generated. Moreover, an application can be designed and analyzed independently of the others, because its execution behavior only depends on its own computational demand and the allocated bandwidth.

In our system, allocating each bandwidth for satisfying the worst-case processing demand of the application would waste resources and would lead to an infeasible schedule. For instance, the maximum bandwidth requirements of the first two applications already exceed the full processor capacity. To achieve a feasible schedule, the bandwidth can be reserved to satisfy a processing demand slightly higher than the average value, handling sporadic overruns through resource reservation. In the considered example, 40% of the processor can be reserved to the phone call, 50% to the video player, and, the remaining 10% to the web browser, which has less stringent timing constraints.

When the amount of allocated bandwidth results to be quite different from the real application needs, adaptive approaches based on feedback mechanisms can be applied at runtime to adjust the allocated bandwidth to the real resource needs (Abeni & Buttazzo, 1999; Bini et al., 2011).

4.2 Overload handing in robot control systems

Let us consider a mobile robot system whose goal is to explore an unknown environment to localize given targets through a dedicated sensor, while avoiding obstacles along the path using proximity sensing. Note that, by equipping the robot with suitable sensors, the system could be used for very different applications; for instance, to discover electrical sockets in a room, using a video camera, or to localize unexploded mines in a field, using a georadar. For the purpose of this chapter, we consider a robot equipped with two motors to move in two directions on a flat surface, two encoders to measure the angular rotation of the wheels and reconstruct the traveled path, a sensor to detect the target (target sensor), a proximity sensor (e.g., based on ultrasound transducers) to detect the distance from possible obstacles along the path, and an electronic compass to orient the trajectory in desired directions.

From the software point of view, the application consists of the following periodic tasks:

- **Motor Control Task** (MCT or τ_1): it performs the low-level motor control loop to drive the robot in a desired direction (θ) at a given speed (v);
- **Obstacle Detection Task** (ODT or τ_2): it reads the proximity sensor to detect a possible obstacle along the path;
- **Target Detection Task** (TDT or τ_3): it reads the target sensor and stores the target location in a buffer when it is detected;
- **Exploration Task** (EXT or τ_4): it generates the proper set points (exploring direction and speed) for the Motor Control Task;
- **Obstacle Avoidance Task** (OAT or τ_5): it is activated when an obstacle is detected and computes a sequence of set points to be followed to avoid the obstacle and return to the planned path.

To illustrate the use of an overload management technique, we assume that in normal operating conditions (i.e., in the absence of obstacles) the first four tasks ($\tau_1, \ldots, \tau_4$) generate a load equal to $U_{norm} = 0.9$, while the fifth task has a utilization $U_5 = 0.3$. Hence, when τ_5 is activated together with the other tasks, the total system utilization becomes $U_{max} = 1.2$. In this example, the elastic approach is applied to bring the load back to a desired value $U_d = 0.9$ (equal to the normal load condition).

The tasks are organized as shown in Figure 13, where hardware components are represented by rounded boxes, tasks by circles, and shared buffers by rectangles.

To apply the elastic method, each task τ_i must specify a range of valid periods $[T_i^{min}, T_i^{max}]$ and an elastic coefficient E_i. Task parameters are reported in Table 1 and are expressed in milliseconds. Note that, when the OAT task is not active, the utilization of the task set is

Task name	Task ID	C_i	T_i^{min}	T_i^{max}	E_i
MCT	τ_1	3	10	10	0
ODT	τ_2	6	20	30	1
TDT	τ_3	20	100	200	2
EXT	τ_4	20	200	500	1
OAT	τ_5	6	20	40	2

Table 1. Task parameters of the robot application.

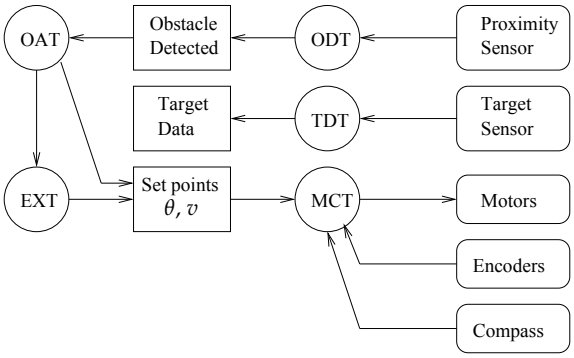

Fig. 13. Task interaction in the robot application.

$$U_{norm} = \sum_{i=1}^{4} \frac{C_i}{T_i} = 0.9$$

whereas, when OAT is active, the system becomes overloaded, being

$$U_{max} = \sum_{i=1}^{5} \frac{C_i}{T_i} = 1.2.$$

By applying the elastic approach, with a desired utilization $U_d = 0.9$ (to keep a safety margin), tasks utilizations are compressed according to Equation (19) and then enforced by re-computing the periods as $T_i = C_i/U_i$. The new tasks utilizations and periods derived by the elastic algorithm are reported in Table 2.

Task name	Task ID	U_i	T_i
MCT	τ_1	3/10	10
ODT	τ_2	1/4	24
TDT	τ_3	1/10	200
EXT	τ_4	1/20	400
OAT	τ_5	1/5	30

Table 2. Task utilizations and periods derived by the elastic compression, with a desired utilization $U_d = 0.9$.

When the obstacle is overcome, task OAT can be suspended and the remaining tasks can return to their original condition, running with their minimum periods.

5. References

Abeni, L. & Buttazzo, G. (1998). Integrating multimedia applications in hard real-time systems, *Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS'98)*, Madrid, Spain.

Abeni, L. & Buttazzo, G. (1999). Adaptive bandwidth reservation for multimedia computing, *Proceedings of the 6th IEEE International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*, Hong Kong, China, pp. 70–77.

Abeni, L. & Buttazzo, G. (2000). Support for dynamic qos in the hartik kernel, *IEEE Proceedings of the 7th International Conference on Real-Time Computing Systems and Applications (RTCSA'00)*, Cheju Island, South Korea.

Abeni, L. & Buttazzo, G. (2004). Resource reservations in dynamic real-time systems, *Real-Time Systems* 27(2): 123–165.

Abeni, L. & Buttazzo, G. (May 30 - June 1, 2001). Hierarchical qos management for time sensitive applications, *Proceedings of the IEEE Real-Time Technology and Applications Symposium (RTAS'01)*, Taipei, Taiwan.

Aydin, H., Melhem, R., Mossé, D. & Alvarez, P. M. (2001). Optimal reward-based scheduling for periodic real-time tasks, *IEEE Transactions on Computers* 50(2): 111–130.

Baruah, S., Rosier, L. & Howell, R. (1990). Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor, *Journal of Real-Time Systems* 2.

Bernat, G., Broster, I. & Burns, A. (December 5-8, 2004). Rewriting history to exploit gain time, *Proceedings of the 25th IEEE Real-Time Systems Symposium (RTSS'04)*, Lisbon, Portugal.

Bernat, G. & Burns, A. (2002). Multiple servers and capacity sharing for implementing flexible scheduling, *Real-Time Systems* 22: 49–75.

Bertogna, M., Fisher, N. & Baruah, S. (2009). Resource-sharing servers for open environments, *IEEE Transactions on Industrial Informatics* 5(3): 202–220.

Bini, E., Buttazzo, G., Eker, J., Schorr, S., Guerra, R., Fohler, G., Arzen, K.-E., Segovia, V. R. & Scordino, C. (2011). Resource management on multicore systems: The ACTORS approach, *IEEE Micro* 31(3): 72–81.

Bini, E., Buttazzo, G. & Lipari, G. (2009). Minimizing cpu energy in real-time systems with discrete speed management, *ACM Transactions on Embedded Computing Systems* 8(4): 31:1–31:23.

Buttazzo, G. (2011). *Hard Real-Time Computing Systems and Applications - Third Edition*, Springer, pp. 287–292.

Buttazzo, G., Abeni, L. & Lipari, G. (1998). Elastic task model for adaptive rate control, *IEEE Real Time System Symposium*, Madrid, Spain.

Buttazzo, G., Lipari, G., Caccamo, M. & Abeni, L. (2002). Elastic scheduling for flexible workload management, *IEEE Transactions on Computers* 51(3): 289–302.

Buttazzo, G. & Stankovic, J. (1995). Adding robustness in dynamic preemptive scheduling, *in* D. Fussel & M. Malek (eds), *Responsive Computer Systems: Steps Toward Fault-Tolerant Real-Time Systems*, Kluwer Academic Publishers.

Caccamo, M. & Buttazzo, G. (1997). Exploiting skips in periodic tasks for enhancing aperiodic responsiveness, *IEEE Real-Time Systems Symposium*, San Francisco, California, USA, pp. 330–339.

Caccamo, M., Buttazzo, G. & Sha, L. (2000). Capacity sharing for overrun control, *Proceedings of the IEEE Real-Time Systems Symposium*, Orlando, Florida, USA.

Caccamo, M., Buttazzo, G. & Thomas, D. (2005). Efficient reclaiming in reservation-based real-time systems with variable execution times, *IEEE Transactions on Computers* 54(2): 198–213.

Easwaran, A., Anand, M. & Lee, I. (2007). Compositional analysis framework using EDP resource models, *Proceedings of the* 28th *IEEE International Real-Time Systems Symposium*, Tucson, AZ, USA, pp. 129–138.

Feng, X. & Mok, A. K. (2002). A model of hierarchical real-time virtual resources, *Proceedings of the* 23rd *IEEE Real-Time Systems Symposium*, Austin, TX, USA, pp. 26–35.

Guangming, Q. (2009). An earlier time for inserting and/or accelerating tasks, *Real-Time Systems* 41(3): 181–194.

Isovic, D., Fohler, G. & Steffens, L. F. (2005). Real-time issues of mpeg-2 playout in resourse constrained systems, *Journal of Embedded Computing* 1: 239–256.

Koren, G. & Shasha, D. (1995). Skip-over: Algorithms and complexity for overloaded systems that allow skips, *Proceedings of the IEEE Real-Time Systems Symposium*.

Lamastra, G., Lipari, G. & Abeni, L. (December 3-6, 2001). A bandwidth inheritance algorithm for real-time task synchronization in open systems, *IEEE Proceedings of the 22nd Real-Time Systems Symposium (RTSS'01)*, London, UK.

Lin, C. & Brandt, S. A. (December 5Ű8, 2005). Improving soft real-time performance through better slack management, *Proc. of the IEEE Real-Time Systems Symposium (RTSS 2005)*, Miami, Florida, USA.

Lin, K., Natarajan, S. & Liu, J. (1987). Concord: a system of imprecise computation, *Proceedings of the 1987 IEEE Compsac*.

Lipari, G. & Bini, E. (2003). Resource partitioning among real-time applications, *Proceedings of the* 15th *Euromicro Conference on Real-Time Systems (ECRTS'03)*, Porto, Portugal, pp. 151–158.

Liu, C. & Layland, J. (1973). Scheduling algorithms for multiprogramming in a hard-real-time environment, *Journal of the Association for Computing Machinery* 20(1).

Liu, J., Lin, K. & Natarajan, S. (1987). Scheduling real-time, periodic jobs using imprecise results, *Proceedings of the IEEE Real-Time System Symposium*.

Liu, J., Lin, K., Shih, W., Yu, A., Chung, C., Yao, J. & Zhao, W. (1991). Algorithms for scheduling imprecise computations, *IEEE Computer* 24(5): 58–68.

Liu, J., Shih, W. K., Lin, K. J., Bettati, R. & Chung, J. Y. (1994). Imprecise computations, *Proceedings of the IEEE* 82(1): 83–94.

Marzario, L., Lipari, G., Balbastre, P. & Crespo, A. (2004). Iris: A new reclaiming algorithm for server-based real-time systems, *Proc. of the IEEE Real-Time and Embedded Technology and Applications Symposium*, Toronto, Canada.

Mercer, C. W., Savage, S. & Tokuda, H. (1994). Processor capacity reserves for multimedia operating systems, *Proceedings of IEEE international conference on Multimedia Computing and System*.

Mok, A. K., Feng, X. & Chen, D. (2001). Resource partition for real-time systems, *Proceedings of the* 7th *IEEE Real-Time Technology and Applications Symposium*, Taipei, Taiwan, pp. 75–84.

Natarajan, S. (ed.) (1995). *Imprecise and Approximate Computation*, Kluwer Academic Publishers.

Palopoli, L., Abeni, L., Lipari, G. & Walpole, J. (December 3-5, 2002). Analysis of a reservation-based feedback scheduler, *Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS'02)*, Austin-Texas.

Shih, W., Liu, W. & Chung, J. (1991). Algorithms for scheduling imprecise computations with timing constraints, *SIAM Journal of Computing* 20(3): 537–552.

Shih, W., Liu, W., Chung, J. & Gillies, D. (1989). Scheduling tasks with ready times and deadlines to minimize average error, *Operating System Review* 23(3).

Shin, I. & Lee, I. (2003). Periodic resource model for compositional real-time guarantees, *Proceedings of the* 24th *Real-Time Systems Symposium*, Cancun, Mexico, pp. 2–13.

Real-Time Systems, Architecture, Scheduling, and Application
Edited by Dr. Seyed Morteza Babamir

ISBN 978-953-51-0510-7
Hard cover, 334 pages
Publisher InTech
Published online 11, April, 2012
Published in print edition April, 2012

This book is a rich text for introducing diverse aspects of real-time systems including architecture, specification and verification, scheduling and real world applications. It is useful for advanced graduate students and researchers in a wide range of disciplines impacted by embedded computing and software. Since the book covers the most recent advances in real-time systems and communications networks, it serves as a vehicle for technology transition within the real-time systems community of systems architects, designers, technologists, and system analysts. Real-time applications are used in daily operations, such as engine and break mechanisms in cars, traffic light and air-traffic control and heart beat and blood pressure monitoring. This book includes 15 chapters arranged in 4 sections, Architecture (chapters 1-4), Specification and Verification (chapters 5-6), Scheduling (chapters 7-9) and Real word applications (chapters 10-15).

How to reference
In order to correctly reference this scholarly work, feel free to copy and paste the following:

Giorgio C. Buttazzo (2012). Handling Overload Conditions in Real-Time Systems, Real-Time Systems, Architecture, Scheduling, and Application, Dr. Seyed Morteza Babamir (Ed.), ISBN: 978-953-51-0510-7, InTech, Available from: http://www.intechopen.com/books/real-time-systems-architecture-scheduling-and-application/overload-handling

INTECH
open science | open minds

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821