Handling Execution Overruns in Hard Real-Time Control Systems

Marco Caccamo, Giorgio Buttazzo, *Member*, *IEEE*, and Lui Sha, *Fellow*, *IEEE*

Abstract—In many real-time control applications, the task periods are typically fixed and worst-case execution times are used in schedulability analysis. With the advancement of robotics, flexible visual sensing using cameras has become a popular alternative to the use of embedded sensors. Unfortunately, the execution time of visual tracking varies greatly. In such environments, control tasks have a normally short computation time, but also an occasional long computation time; therefore, the use of worst-case execution time is inefficient for control performance optimization. Nevertheless, to maintain the control stability, we still need to guarantee the schedulability of the task set, even if the worst case arises. In this paper, we propose an integrated approach to control performance optimization and task scheduling for control applications where the execution time of each task can vary greatly. We present an innovative approach to overrun management that allows us to fully utilize the processor for optimizing the control performance and yet guaranteeing the schedulability of all tasks under worst-case conditions.

Index Terms—Overrun management, rate adaptation, real-time scheduling.

1 Introduction

1.1 Motivation

In many real-time control applications, task periods are typically fixed and worst-case execution times are used in the schedulability analysis. This model is fine for many classical control applications where the execution time variations are small. However, with the advancement of robotics, flexible visual sensing using cameras has become a popular alternative to the use of embedded sensors. Unfortunately, the execution time of visual tracking varies greatly.

For example, consider a ball and plate control system based on visual feedback, where a ball has to follow a specified trajectory on the plate, which can be controlled by acting on roll and pitch rotations. To speed up the visual tracking process, predictive techniques are typically used to search for the ball in a small mobile window centered in the estimated ball position rather than searching the whole image. Normally, the ball is found in the small window and its position can be computed quickly. However, in most control applications there are occasional disturbances. If the disturbance makes the ball move outside of the predicted window, searching has to be extended in a larger area. This process may continue until, eventually, the entire plate is scanned.

In this example, the visual tracking task's computation time can be modeled as having a constant normal execution time, due to the searching operation in the small window. In addition, it has a bounded probabilistic execution time during tracking exceptions. Similar situations can be found in radar tracking where a search window is centered on the predicted location of the target.

It is worth noting that the search time is part of the control loop since the position information is needed in control computations. With a normally short (control loop) computation time, but an occasional long computation time, the use of worst-case computation times is inefficient. Nevertheless, to guarantee the control stability, we still need to close the control loop in time, even if the worst case arises.

If we reserve the processor for the worst-case condition, then most of the time there is a large amount of reserved but unused execution time budget. Although such an unused reserved time can be reclaimed for soft real-time aperiodic applications [18], [19], [20], such an aperiodic application may or may not exist in a given application environment. An alternative solution is to fully utilize the reserved budget to optimize the control performance in spite of the variation in computation time.

In digital control, the system performance is a function of the sampling rate. For a given controller design method, faster sampling permits, up to a limit, a better control performance. So, the idea is to increase the control loop frequency when the loop computation time is short and slow down the frequency when the worst-case situation arises. However, there is a lower bound on the frequency for each task, f_i^{min}, the minimum frequency for each task τ_i, below which the performance is unacceptable or worse: The control becomes unstable. In this formulation, $1/f_i^{min}$ represents the hard deadline that each instance of τ_i has to honor.

The frequency adjustment is particularly easy when the common method of digitized analog design is used since this method does not require the change of control gains. The scheduler design method in this paper is also applicable to other control design methods. In this case, it

- *M. Caccamo is with Scuola Superiore S. Anna, Pisa, Italy. E-mail: caccamo@sssup.it.*
- *G. Buttazzo is with the INFM Research Unit, University of Pavia, Italy. E-mail: giorgio@sssup.it.*
- *L. Sha is with the Department of Computer Science, University of Illinois, Urbana, IL 61801. E-mail: lrs@cs.uiuc.edu.*

Manuscript received 7 July 2000; revised 2 July 2001; accepted 27 Nov. 2001. For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 112417.

TABLE 1
Task Set Parameters

Task	$WCET_i$ (ms)	c_i^n (ms)	f_i^{min} (Hz)
τ_1	25	20	9.9
τ_2	25	20	20

involves the design of control gain scheduling in conjunction with scheduler design. However, application of this approach requires more work on the analysis of system stability. Since the focus of this paper is on the scheduler design, we shall assume that a digitized analog controller design method is used for the system.

We refer the idea of dynamically adjusting the rates to optimize the control performance as "rate adaptation method". Notice that a simple-minded implementation of this idea would not work. Consider the following example.

Example 1. A simple system is composed of two tasks whose parameters are shown in Table 1, where $WCET_i$ is the worst-case execution time in msec, c_i^n is the normal execution time in msec, and f_i^{min} is the minimal frequency in Hz. Suppose that when both tasks execute normally, the frequency assignment for τ_1 is assumed to be $f_1 = 9.9$ Hz, the same as the minimal frequency ($f_1 = f_1^{min}$). Suppose also that the frequency assignment for task τ_2 is $f_2 = 40$ Hz, twice that of the minimal frequency of 20 Hz. Under this arrangement, these two tasks use 20 percent and 80 percent of the CPU, respectively. The idea is that, should the worst case arise, task τ_2 could slow down from 40 Hz to 20 Hz to avoid overload. Unfortunately, this idea would not work.

As shown in Fig. 1, both tasks start at time $t = 0$. At time $t = 100$, task τ_1 requests to execute 25 instead of 20 units of time. Unfortunately, at this point, there is nothing task τ_2 can do to assist task τ_1. As a result, τ_1 will miss its deadline at $t_{ov} = 101$.

To make the idea of rate adaptation work, the challenge is to develop a method that will adjust the task frequencies to produce a near optimal system control performance, subject to the constraint of guaranteeing that when the worst case comes, no deadlines will be missed.

1.2 Related Work

The problem of handling execution overruns in real-time systems has been recently addressed in the literature using various approaches. In [11], Gardner and Liu compared the behavior of three classes of scheduling algorithms for scheduling real-time systems in which jobs may overrun their allocated processor time, potentially causing the system to be overloaded. In their work, each task is characterized by a guaranteed execution time, which is zero for non-real-time tasks, equal to the worst-case execution time for hard real-time tasks, and equal to some intermediate value for soft real-time tasks. They introduced the Overrun-Server Method (OSM) and the Isolation-Server Method (ISM) as scheduling algorithms. Under OSM, a job is released for execution and scheduled in the same manner as it would be according to Deadline Monotonic (DM) [13] or Earliest Deadline First (EDF) [14] (in a fixed priority or dynamic priority environment, respectively). At the time of exception, the execution of the job is interrupted and its remaining execution time is released as an aperiodic request to a server. Using the ISM technique, jobs are submitted as aperiodic requests to the server assigned to their task at the time of their release and execute completely under server control. These scheduling strategies guarantee a feasible schedule for those jobs which do not generate exceptions, but they are unable to guarantee a maximum response time for those which generate exceptions.

In [21], [22], Stankovic et al. described another approach to increase the performance of a scheduling algorithm in unpredictable dynamic systems whose workloads cannot be accurately modeled. In such situations, a design based on worst-case assumptions would result in a highly underutilized system. In these papers, the authors proposed a new scheduling paradigm, called feedback control real-time scheduling, which defines error terms for schedules, monitors the amount of error, and continuously adjusts the schedules to maintain satisfactory performance. Using control theory methodology, they defined the percentage of tasks that miss their deadlines as the controlled variable and the requested CPU utilization as the manipulated variable. By doing this, the deadline miss ratio can be controlled by varying the admission strategy of tasks online. This technique, however, cannot be used to handle hard real-time tasks since jobs may be rejected to keep the system not fully loaded. Moreover, the method is unable to isolate

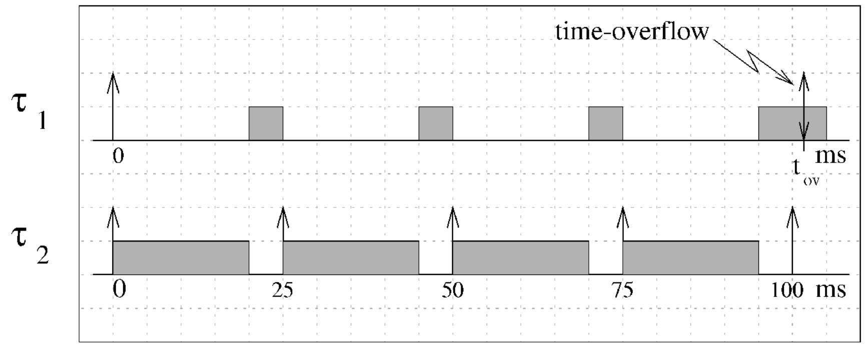


Fig. 1. Example of time-overflow due to an exception.

tasks from reciprocal interference; hence, a task overrun would affect the performance of all the other tasks.

In [1], Abdelzaher et al. proposed a mechanism for QoS (re)negotiation as a way to ensure graceful degradation in a distributed pool of shared computing resources. Although the method provides flexibility in handling load variations caused by new task arrivals or hardware failures, transient execution overruns due to variable computation times cannot be handled locally without affecting the whole application.

In [7], Buttazzo et al. presented a novel scheduling methodology for managing overload conditions in real-time control applications. According to their elastic approach, task utilizations are treated as springs that can be compressed through period variations to conform with a given workload. A slightly different method was proposed by Beccari et al. [6] in the context of soft real-time robotic applications. In both techniques, however, tasks execution times are assumed to be constant.

In [17], Shin and Meissner described a resource adaptation method in multiprocessor real-time control systems. Periodic utilization is adapted by changing task periods, whose values are computed to maximize a performance index. Their method is able to handle system overloads due to new task arrivals or application changes; however, periodic tasks have the restriction that periods must be harmonically related, that is each task period must be a multiple of all shorted periods.

Finally, in [15], Ryu and Hong described a design methodology to synthesizing schedulable timing constraints for real-time control systems, however no runtime mechanisms are described to handle online variations of tasks' execution times.

The problem of selecting a set of control task frequencies to optimize the system control performance subject to schedulability constraints was addressed by Seto et al. [16]. In this formulation, each control task τ_i is characterized by a *Performance Loss Index* (PLI[1]), which measures the difference between a digital and a continuous control as a function of the sampling frequency. In particular, if J_c and $J_d(f)$ are the performance indices generated by a continuous-time control and its digital implementation at a sampling frequency f, the PLI is defined as $\Delta J(f) = |J_d(f) - J_c|$, which is convex and monotonically decreasing with the frequency. In [16], for each control task τ_i, $\Delta J_i(f_i)$ is approximated by the following exponential function:

$$\Delta J_i(f_i) = \alpha_i e^{-\beta_i f_i}$$

where f_i is the frequency of τ_i, α_i is a magnitude coefficient, and β_i is the decay rate. A typical PLI is illustrated in Fig. 2, where f_m is the lower bound on the sampling frequency.

The performance loss index of the overall system $\Delta J(f_1, \ldots, f_n)$ is defined in [16] as follows:

$$\Delta J(f_1, \ldots, f_n) = \sum_i w_i \Delta J_i(f_i), \qquad (1)$$

1. In the original formulation, the performance loss index was simply called performance index or PI. In the following, it will be called PLI for more clarity.

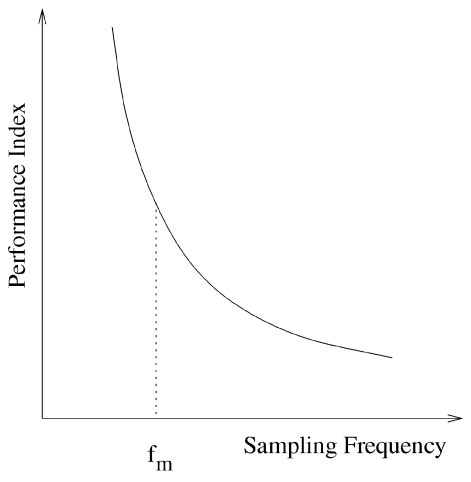


Fig. 2. Control system performance index versus sample frequency.

where w_i is a design parameter determined from the application. For instance, it can be the relative importance of the task in the control system with respect to the others.

Given the available bandwidth (A), the minimum permitted frequency (f_i^{min}), the worst-case execution time ($WCET_i$) and the weighed PLI ($w_i \Delta J_i(f_i)$) of each task τ_i as input parameters, Seto et al. [16] provided an algorithm, from now on referred to as the SLSS algorithm, to compute the frequencies f_i^{opt} which minimize the PLI of the system while guaranteeing the schedulability constraints (i.e., ensuring each task will meet its deadlines). Notice that each task frequency f_i^{opt} computed by the SLSS algorithm is always greater than or equal to the corresponding minimum frequency f_i^{min}.

However, in [16], f_i^{opt} were computed based on $WCET$s. If the normal computation times c_i^n are much less than $WCET_i$, then f_i^{opt} can be too low.

In this paper, we propose an integrated approach to control performance optimization and task scheduling for control applications where tasks' execution times have large variations. To improve a system's performance in normal conditions, optimal frequencies are computed using the SLSS algorithm based on normal computation times; however, schedulability is guaranteed under worst-case conditions, by a proper overrun management algorithm.

The rest of the paper is organized as follows: Section 2 introduces some of our terminology and basic assumptions. Section 3 presents a local approach for handling overruns and shows an example to illustrate the performance improvement that can be achieved using the proposed approach. Section 4 briefly recalls the Constant Bandwidth Server (CBS) algorithm and introduces an extension of it (CBS^{hd}) to efficiently schedule periodic tasks' overruns. Section 5 extends the rate adaptation method to work in the presence of resource constraints. Section 6 illustrates some experimental results. Section 7 addresses some engineering issues for reducing the runtime overhead and simplifying the implementation of the proposed approach. Finally, Section 8 presents our conclusions and future work.

2 Terminology and Assumptions

In this section, we introduce the rate adaptation method to improve the performance of real-time control systems with large variations of computation times. We will assume that the task set is scheduled by the Earliest Deadline First (EDF) algorithm [14], which assigns higher priorities to tasks with earlier deadlines.

Each periodic task is described by

$$\tau_i(WCET_i, c_i^n, f_i^{min}, \Delta J_i(f_i)),$$

where $WCET_i$ is the worst-case execution time, c_i^n is the normal computation time typically demanded by the task ($c_i^n \leq WCET_i$), f_i^{min} is the minimum frequency τ_i can execute at, and $\Delta J_i(f_i)$ is the Performance Loss Index (PLI) of τ_i.

Each task τ_i is considered as a sequence of jobs $\tau_{i,j}$ ($j = 1, 2, \ldots$), each characterized by a release time $r_{i,j}$, an execution time $c_{i,j}$, and a dynamic absolute deadline $d_{i,j}$. We require that each task must complete at or before its hard deadline $D_i^{hd} = 1/f_i^{min}$. Notice, however, that task τ_i will be normally scheduled using a dynamic deadline D_i less than or equal to D_i^{hd}.

The *computational demand* $g_i(t_1, t_2)$ of task τ_i is defined as the total computation time requested by those jobs $\tau_{i,j}$ whose arrival times and deadlines are within [t_1, t_2] (that is, $t_1 \leq r_{i,j} \leq d_{i,j} \leq t_2$).

A task τ_i is said to have a *bandwidth utilization* U_i if, in any interval of time [t_1, t_2], its computational demand $g_i(t_1, t_2)$ never exceeds $(t_2 - t_1)U_i$, and there exists an interval [t_a, t_b] such that $g_i(t_a, t_b) = (t_b - t_a)U_i$.

The goal of this approach is to determine the optimal frequencies f_i^{opt} (computed by the SLSS algorithm using c_i^n rather than $WCET_i$) which minimize the overall PLI, while guaranteeing that each task will never miss its hard deadline D_i^{hd}. To achieve isolation, each task is handled by a dedicated Constant Bandwidth Server (CBS) [2], which assigns it a fraction of the processor (bandwidth) and schedules each job by a suitable dynamic deadline computed according to the allocated bandwidth and the actual computational requirements. It will be briefly recalled in Section 4. The advantage of this approach is that execution overruns can be handled locally to each task; that is, whenever an overrun occurs on job $\tau_{i,j}$ (because $c_{i,j} > c_i^n$), its deadline is postponed to delay its execution, so that the frequencies of the other tasks are not affected.

Thus, to handle overruns, a task can dynamically change its frequency depending on its actual computational demand. If $d_{i,j}^{last}$ is the last deadline used by the server to schedule job $\tau_{i,j}$, the next job $\tau_{i,j+1}$ will be released at time

$$r_{i,j+1} = d_{i,j}^{last}. \tag{2}$$

Hence, each job $\tau_{i,j}$ has a variable period $T_{i,j} = r_{i,j+1} - r_{i,j}$. Using this formalism, we need to guarantee that

$$\forall\, i, j \quad T_{i,j} \leq \frac{1}{f_i^{min}}. \tag{3}$$

In the next section we will show how to handle overruns correctly and how to perform an offline guarantee of the task set. Notice that no restrictions are assumed on the number of overruns that a task can generate.

3 A Local Approach for Handling Overruns

In this section, we introduce a simple policy which allows each task to handle its overruns locally, that is, without affecting the frequencies of the other tasks. Using this approach, overruns can be handled efficiently, with a very small overhead.

In the following, we assume that the optimal frequency f_i^{opt} for each task τ_i has already been computed, based on c_i^n, using the SLSS algorithm proposed in [16]. Then, each task τ_i is reserved a bandwidth

$$U_i = f_i^{opt} c_i^n. \tag{4}$$

To schedule tasks with a given bandwidth U_i we can use the deadline assignment rule adopted by the Total Bandwidth Server [20]. According to this rule, a job $\tau_{i,j}$ with computation time c_i^n is assigned a deadline

$$d_{i,j}^0 = r_{i,j} + \frac{c_i^n}{U_i} = r_{i,j} + \frac{1}{f_i^{opt}}.$$

If $\tau_{i,j}$ tries to execute more than c_i^n, its deadline can be safely postponed at

$$d_{i,j}^{last} = d_{i,j}^1 = r_{i,j} + \frac{WCET_i}{U_i} = d_{i,j}^0 + \frac{WCET_i - c_i^n}{U_i}. \tag{5}$$

In this way, the requested bandwidth is not exceeded even though the job will run for its whole $WCET_i$. The "safety" of the deadline postponement rule relies on the correct assignment of bandwidth U_i, which will be computed to satisfy the constraints provided by Theorem 1 below.

Notice that the new job deadline is computed by assuming a maximum overrun, equal to $WCET - c^n$. In general, however, the overrun might be less than the maximum value, so the deadline postponement rule is pessimistic. This problem will be addressed in Section 4, where a more efficient scheduling strategy will be described to better exploit the available bandwidth.

Fig. 3 illustrates how an overrun is handled by the proposed method. The task set consists of two periodic tasks, τ_1 and τ_2, with minimum frequencies $1/20$ and $1/16$, worst-case execution times 5 and 8, and normal execution times 3 and 2, respectively. Moreover, let us suppose that the optimal frequencies computed by the SLSS algorithm are $f_1^{opt} = 1/6$ and $f_2^{opt} = 1/4$. Therefore, each task is assigned a bandwidth $U_1 = U_2 = 0.5$.

Initially, job $\tau_{2,2}$ is assigned a deadline $d_{2,2}^0 = 8$. At time $t = 7$, an overrun occurs and the job deadline is postponed at $d_{2,2}^1 = d_{2,2}^0 + (WCET_2 - c_2^n)/U_2 = 20$. After handling the overrun, the next job $\tau_{2,3}$ of task τ_2 will be released at time $r_{2,3} = 20$ and it will be executed again at its optimal frequency.

The following theorem gives a necessary and sufficient condition to guarantee the worst-case scenario in the presence of overruns handled with the local approach.

Theorem 1. *Let Γ be a set of periodic tasks*

$$\tau_i(WCET_i, c_i^n, f_i^{min}, \Delta J_i(f_i)),$$

such that the bandwidth utilization of each task is bounded by U_i, and $\sum_{i=1}^{n} U_i \leq 1$. Then, all tasks are schedulable by the algorithm given above with a frequency $f_i \geq f_i^{min}$ if and only if:

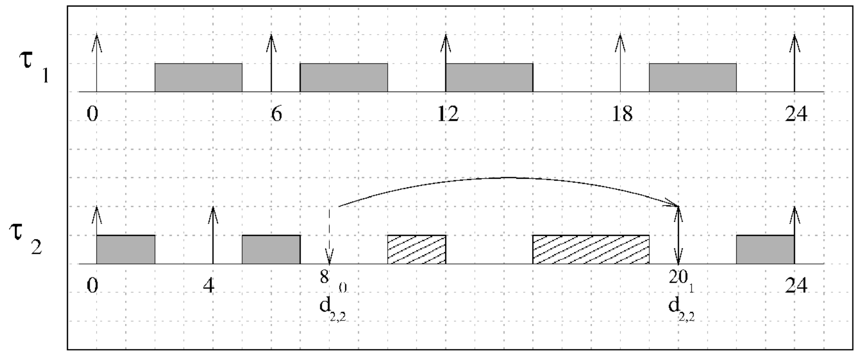


Fig. 3. Example of overrun handled locally.

$$\forall\, \tau_i \quad U_i \geq f_i^{min} WCET_i. \tag{6}$$

Proof. *If.* Suppose that (6) holds and that the jth job of task τ_k has an overrun. We notice that the worst-case condition for the completion time of task τ_k occurs when τ_k is scheduled with a bandwidth $U_k = f_k^{min} WCET_k$ and it raises an overrun at the end of its period, requiring a computation time equal to its $WCET_k$. Hence, if $r_{k,j}$ is the release time of the job, the worst case for its completion time occurs when an overrun equal to $WCET_k - c_k^n$ is detected at time $t_{ov} = r_{k,j} + c_k^n / U_k$. To check the feasibility of τ_k, we compute the response time R_k and verify that it is less than or equal to its hard relative deadline $D_k^{hd} = 1/f_k^{min}$. Since $\sum_{i=1}^{n} U_i \leq 1$, the schedule produced by EDF is feasible and all tasks are guaranteed to complete within their deadlines, that is

$$R_k \leq d_{k,j}^{last} - r_{k,j}.$$

Hence, substituting $d_{k,j}^{last}$ with the expression given in (5) we have:

$$R_k \leq d_{k,j}^{last} - r_{k,j} = \frac{WCET_k}{U_k} \leq \frac{WCET_k}{f_k^{min} WCET_k} = \frac{1}{f_k^{min}}.$$

Only if. By contradiction. Suppose that a task τ_k exists such that (6) is false, that is:

$$\exists\, \tau_k \;\mid\; U_k < f_k^{min} WCET_k,$$

but still each job ($\tau_{k,j}$) has a period $T_{k,j} \leq 1/f_k^{min}$. If τ_k requests a computation time equal to its $WCET_k$, then the actual frequency f_k of τ_k becomes

$$f_k = \frac{U_k}{WCET_k} < f_k^{min},$$

which is a contradiction. □

Using the result of Theorem 1, we can compute the optimal frequencies f_i^{opt} and guarantee a minimum frequency f_i^{min} for each task τ_i, even in the presence of overruns. Like in the classical SLSS algorithm, in this approach a task set is guaranteed if and only if:

$$\sum_i f_i^{min} WCET_i \leq 1.$$

As previously described in Section 1.2, the SLSS algorithm takes as input parameters the available bandwidth (A), the minimum permitted frequency (f_i^{min}), the worst-case execution time ($WCET_i$) and the weighed PLI ($w_i \Delta J_i(f_i)$) of each task to compute the optimal frequencies f_i^{opt} such that ($\forall i\;,\; f_i^{opt} \geq f_i^{min}$).

To implement the idea of local overrun on the SLSS algorithm, the additional constraint given in (6) must be satisfied by each task. Since each task is reserved a bandwidth $U_i = f_i^{opt} c_i^n$, the condition given in (6) can be expressed as a constraint on the optimal frequency:

$$\forall\, \tau_i \quad f_i^{opt} \geq \frac{f_i^{min} WCET_i}{c_i^n}. \tag{7}$$

Being $WCET_i \geq c_i^n$, (7) poses a new lower bound on the minimum frequency allowed for each task (called $\tilde{f}_i^{min}$), which must be used in place of f_i^{min} as input in the SLSS algorithm in order to take overruns into account:

$$\forall\, \tau_i \quad \tilde{f}_i^{min} = \frac{f_i^{min} WCET_i}{c_i^n}. \tag{8}$$

In summary, overruns can be taken into account in the SLSS algorithm by computing the optimal frequencies using c_i^n as computation time (instead of $WCET_i$) and $\tilde{f}_i^{min}$ as minimum frequency (instead of f_i^{min}).

Note that the SLSS algorithm is optimal among the algorithms which handle overruns locally (that is, without affecting the other tasks performance) even though the value of $\tilde{f}_i^{min}$ is used as minimum frequency instead of f_i^{min}. This is easy to prove, because the value $\tilde{f}_i^{min}$ represents the minimum frequency permitted to each task τ_i in order to handle overruns locally. Hence, in this model, each optimal frequency f_i^{opt} must be greater than or equal to $\tilde{f}_i^{min}$.

The performance cost of local handling provides us with the opening to consider global handling of overruns; that is, when an overrun occurs, instantaneously each task could decrease its frequency in order to free bandwidth for handling the overrun. However, this different approach is out of the scope of this paper.

3.1 An Example

We illustrate the effect of the proposed technique on a bubble control system, which is a simplified model designed to study diving control in submarines. The same system was described by Seto et al. in [16]. Here, we describe a modified version of it, to emphasize the advantages achievable using our technique.

TABLE 2
Task Parameters for the Bubble Control System

Task	α_i	β_i	$WCET_i$ *(ms)*	f_i^{min} *(Hz)*	w_i
b_1	1	0.4	25	10	2
b_2	1	0.1	25	20	1

The bubble control system considered here consists of a tank filled with air and immersed in the water. Depth control of the diver is achieved by adjusting the piston connected to the air bubble. In this example, a camera monitors the diver as sensor for getting its position.

Now, suppose that two such systems with different physical dimensions are installed on an underwater vehicle to control the depth and orientation of the vehicle, and assume they are controlled by one on-board processor. Hence, each control task is characterized by two different functions; the first one reads the image memory filled by the camera frame grabber and determines the actual position of diver, and the second one computes the next value of the control variable defined as the piston velocity. The first function of each control task is characterized by a variable computation time which depends on the current position of each diver; hence, we can assume that each task is characterized by a worst-case execution time ($WCET_i$) and by a normal computation time c_i^n. The task set parameters are shown in Table 2, where, for each bubble control system i, $WCET_i$ (ms) is the control task worst-case execution time in each sampling period, f_i^{min} (Hz) is the lower bound on sampling frequency, and w_i is the weight assigned to system i.

The following data are given for the control design and scheduling problem: $\Delta J_i = \alpha_i e^{-\beta_i f_i}, \ i = 1, 2$, where the frequencies f_i must be determined.

A simple computation shows that the total CPU utilization of the overall bubble system is 75 percent when the minimum task frequencies are assigned. Supposing the total CPU utilization available for the bubble systems is 100 percent, Table 3 shows, at different values of c^n, the optimal frequencies computed from the SLSS algorithm and the resulting performance loss index of the overall system. Note that system performance increases as the performance loss index decreases. Moreover, $c^n = kWCET$ means that all normal computation times are reduced by that fraction.

TABLE 3
Optimal Frequencies and Corresponding ΔJ for Different Values of c^n

c^n	f_1^{opt} *(Hz)*	f_2^{opt} *(Hz)*	ΔJ
$WCET$	12.16	27.84	0.0772
$0.9WCET$	13.05	31.40	0.0541
$0.8WCET$	14.16	35.84	0.0347
$0.7WCET$	15.59	41.56	0.0196
$0.6WCET$	17.49	49.17	0.0091
$0.5WCET$	20.16	59.84	0.0031

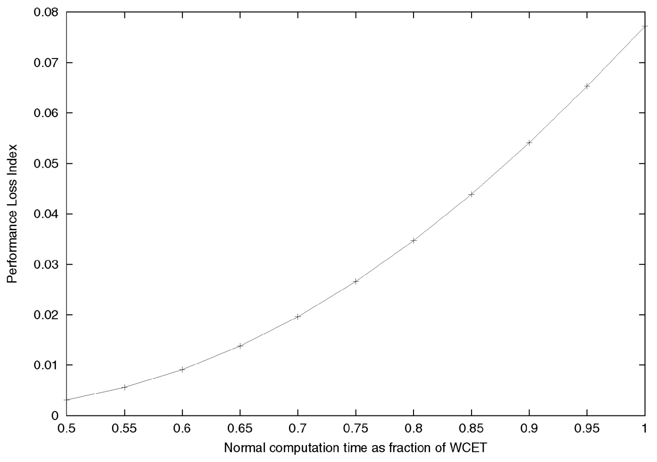


Fig. 4. ΔJ versus normal computation time c^n.

So, for instance, $c^n = 0.9WCET$ means that $c_1^n = 0.9WCET_1$ and $c_2^n = 0.9WCET_2$.

The results reported in Table 3 demonstrate that the control system performance significantly improves as the normal computation time is decreased with respect to the WCET, because tasks can run at higher frequencies. Fig. 4 illustrates the relation between the performance loss index ΔJ and the value of c^n. In the graph, the normal computation time on the x-axis is expressed as a fraction of WCET. Note that a little difference between c^n and WCET gives a significant gain in performance; for example, the performance loss index halves its value when each task has the normal computation time equal to 80 percent of its WCET.

3.2 A Problem with Partial Overruns

The simple rule described in Section 3 for handling overruns can be improved. In fact, assuming that overruns have always a maximum duration ($WCET - c^n$) is not efficient since it causes the deadline to be postponed too far away, even when the overrun has a small duration. To focus this problem, Fig. 5 shows the same task set of Fig. 3, but assuming that task τ_2 has an overrun of four units of time which is less than the maximum value ($WCET_2 - c_2^n = 6$).

The solution given by (5) is quite inefficient since the next deadline is computed assuming that a maximum overrun has to be handled. Clearly, this is not always true and the overrun size is usually variable. This problem is solved by cutting each overrun in small chunks, each one characterized by a shorter deadline. In this way, small overruns are handled as efficiently as large overruns.

The chunks' size can be determined by trading off the performance against the runtime overhead. In general, a small chunk size allows a more precise estimation of actual execution times, so tasks are assigned higher frequencies and the system is able to achieve a better performance. On the other hand, however, a small chunk size causes frequent overruns and deadline postponements, hence a larger runtime overhead. Experimental results reported in Section 6 suggest that a good compromise consists in setting the chunk size equal to the task normal computation time.

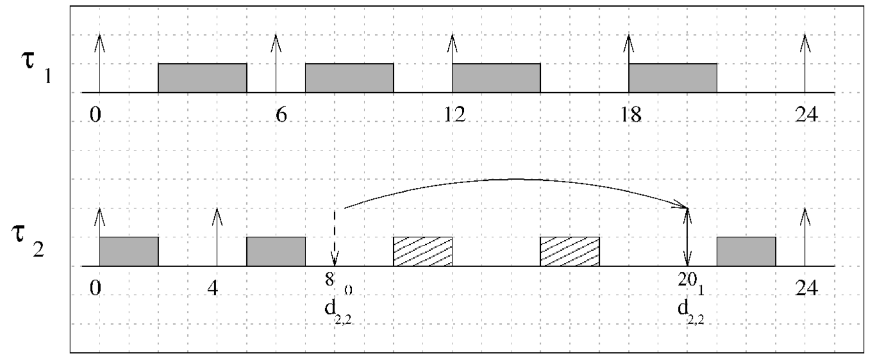


Fig. 5. Example of short overrun handled locally.

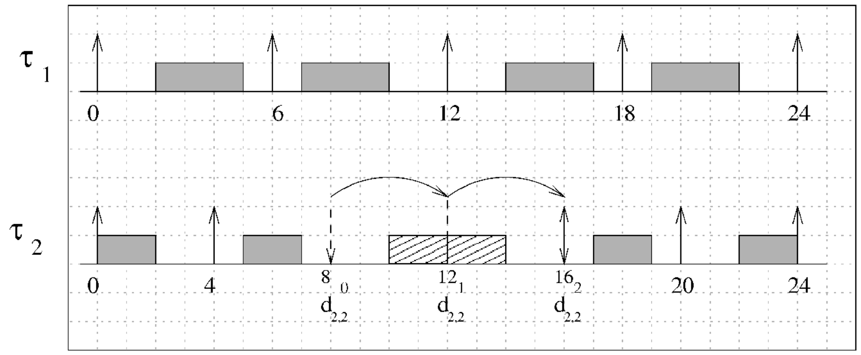


Fig. 6. Example of short overrun handled by CBS server.

Notice that, the correct value to be assigned to the normal computation time of a task is its average execution time, whose value can be derived statistically from previous measurements of task's execution times.

The proposed solution can be implemented by handling each task with a dedicated server. In the following section, we describe how to extend the Constant Bandwidth Server (CBS) in order to manage hard real-time periodic tasks.

4 Handling Overruns with a Server

In [2], Abeni and Buttazzo proposed a scheduling methodology, the *Constant Bandwidth Server* (CBS), devised for handling real-time tasks under a *temporal protection* mechanism. In order to provide task isolation, each task is assigned a fraction of the processor (a fixed *bandwidth*) and it is scheduled in such a way that it will never exceed its specified bandwidth, independently of its actual requests. This is achieved by assigning each task a suitable (dynamic) deadline, computed as a function of the reserved bandwidth and its actual requests. If a task needs to execute more than its expected computation time, its deadline is postponed so that its bandwidth is not exceeded. As a consequence, overruns occurring on task τ_a will only delay task τ_a, but will not steal the bandwidth assigned to the other tasks, which are then isolated and protected from reciprocal interference. In [3], the same authors present a statistical analysis for performing a probabilistic guarantee of soft tasks handled by the CBS algorithm. The CBS algorithm and its main properties are briefly recalled in Appendix A.1.

To show the advantages achievable by using a server mechanism, like the CBS, Fig. 6 illustrates the same task set of Fig. 5, where now each task is scheduled by a CBS server. In particular, task τ_1 is scheduled by a CBS with $Q_{s_1} = 3$ and $T_{s_1} = 6$, and τ_2 by a CBS with $Q_{s_2} = 2$ and $T_{s_2} = 4$. Notice that, using the CBS scheduling rules, job $\tau_{2,2}$ can improve its response time.

According to this new approach, each hard task τ_i must be scheduled with a reserved bandwidth U_i in order to isolate the effects of task overruns; however, we also require guaranteeing that each task τ_i always has a frequency greater than or equal to f_i^{min}. Under these requirements, the plain CBS server cannot be directly used for scheduling tasks with hard deadlines. In fact, if the budget is not properly assigned, the last server deadline assigned to the task could exceed the hard deadline imposed by the minimum frequency.

The following example illustrates how, by using a plain CBS, a task can miss its hard deadline. The task set consists of two periodic tasks, τ_1 and τ_2, with minimum frequencies $1/20$ and $1/14$, worst-case execution times 5 and 7, normal execution times 4 and 3, respectively. Moreover, let us suppose that the optimal frequencies are $f_1^{opt} = 1/8$ and $f_2^{opt} = 1/6$ and the server parameters are $Q_{s_1} = 4$, $T_{s_1} = 8$ for task τ_1, and $Q_{s_2} = 3$ and $T_{s_2} = 6$ for τ_2. Therefore, each server is assigned a bandwidth $U_1 = U_2 = 0.5$.

Fig. 7 shows the task schedule supposing that the first instance of τ_2 has an overrun of four units of time. Notice that task τ_2 misses its hard deadline at time $t = 14$ due to the used server mechanism. To solve this problem, the deadline assignment rule needs to be modified in order to meet the

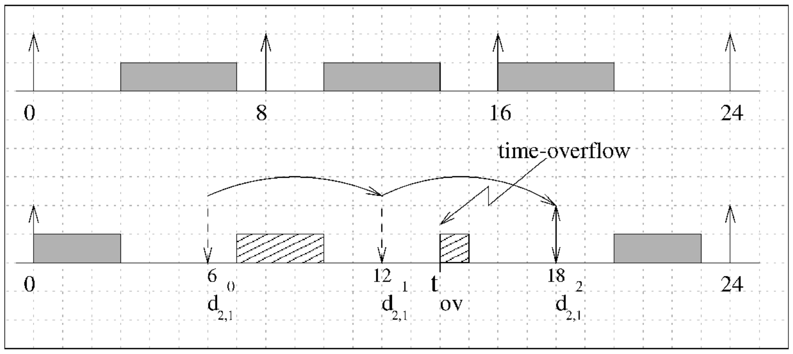


Fig. 7. Example of time-overflow due to the plan CBS server.

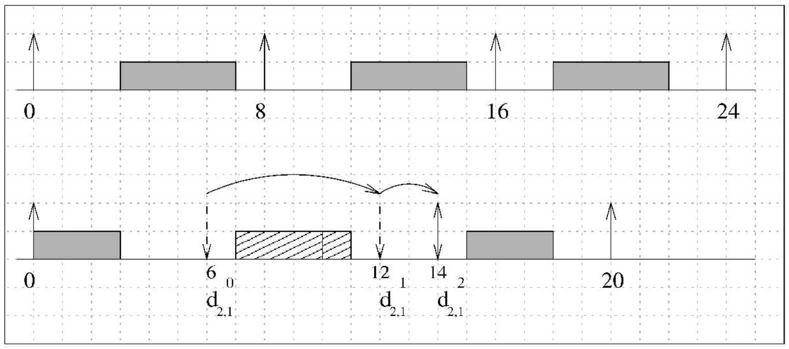


Fig. 8. Example of overrun handled by CBS^{hd} server.

hard timing constraints. In the next section, we propose a new version of the CBS, called the CBS^{hd}.

4.1 The CBS^{hd} Algorithm

The CBS^{hd} maintains all the CBS properties, but allows us to compute the dynamic deadline of each task in a more flexible way. The problem with the plain CBS in handling overruns is that, each time an overrun occurs, the budget is recharged to its maximum value and the task deadline is postponed by a fixed amount, equal to the server period. In this way, however, if the maximum budget is greater than the maximum remaining overrun, the current task deadline would be postponed too far away. In order to bound the task delay introduced by the overrun, the budget can be recharged in a more fit way.

Whenever the server budget is exhausted and the remaining job computation time[2] $c^r_{i,j}$ is less than the maximum server budget Q_{s_i}, the budget is set equal to the remaining computation time ($c_{s_i} = c^r_{i,j}$) and the deadline is postponed accordingly, by $c^r_{i,j}/U_i$, to demand the same bandwidth U_i. Using this simple rule, each job overrun can be safely handled within the hard deadlines.

Fig. 8 shows how the overrun of task τ_2 illustrated in previous example (see Fig. 7) can be safely executed within the hard deadline of $\tau_{2,1}(d^{hd}_{2,1} = r_{2,1} + 1/f^{min}_2 = 14)$. In fact, at time $t = 10$, the budget c_{s_2} is exhausted and the remaining computation time of τ_2 (1 unit of time) is less than the maximum budget $Q_{s_2} = 3$. Hence, the budget is recharged to one ($c_{s_2} = c^r_{2,1} = 1$) and the deadline is postponed accordingly, by two units of time. As a result, task τ_2 completes at time $t = 11$, which is less than its hard deadline.

2. An estimate of the remaining computation time can be computed as a difference between the job worst-case execution time and the amount of time the job has already executed.

More formally, whenever the budget is exhausted, the following rule must be applied for updating the server variables:

```
if $(c^r_{i,j} \geq Q_s)$ {
        $c_s = Q_s$;
        $d_{s,k+1} = d_{s,k} + T_s$;
}
else {
        $c_s = c^r_{i,j}$;
        $d_{s,k+1} = d_{s,k} + c^r_{i,j}/U_i$;
}
```

In the following, we will assume to schedule each task τ_i by a dedicated CBS^{hd}, with a reserved bandwidth $U_i = f^{opt}_i c^n_i$. The server parameters Q_{s_i} and T_{s_i} have to be assigned according to the reserved bandwidth. In particular, after setting the maximum budget, the server period must be set equal to $T_{s_i} = Q_{s_i}/U_i$. A possible solution, which is a good trade-off between efficiency and complexity, consists of assigning Q_{s_i} equal to c^n_i and T_{s_i} equal to $1/f^{opt}_i$.

4.1.1 *Adding Resource Reclaiming*

The behavior of the CBS^{hd} strictly depends on the runtime estimate of the remaining computation time $c^r_{i,j}$ of the current served job $\tau_{i,j}$ since each new server deadline is assigned based on this value. This may be a drawback if the value is overestimated. In this case, the deadline assigned by the server to the current job will be farther than necessary, and this may delay the release of the next job (computed by (2)). This problem can be solved in two different ways: either increasing the number of chunks a job is cut in, or introducing a resource reclaiming technique. Using the first approach, the size of each job chunk becomes smaller, hence the effect of an overestimation of the remaining computation time $c^r_{i,j}$ becomes negligible. Some considerations about the runtime overhead as well as a performance evaluation related to chunk size will be addressed later in Section 6. In this section, we introduce a resource reclaiming mechanism for advancing the release of the next job, when the current job completes earlier than expected. The method is based on that one described by Spuri et al. in [19].

The main idea of such a reclaiming technique is to keep track of the actual processor bandwidth taken by the served task and correct the assigned deadline accordingly. In particular, the actual execution time is used to compute the server deadline that could have been assigned to it if its execution time had been known in advance. This value is then used to compute the release time of the next job and the new server deadline.

If $d_{i,j}$ is the deadline assigned by the server to the current job $\tau_{i,j}$ based on the overrun estimate, and $\Delta_{i,j}$ is the time saved by the job, the corrected deadline $\bar{d}_{i,j}$ can be computed as follows:

$$\bar{d}_{i,j} = d_{i,j} - \frac{\Delta_{i,j}}{U_i};$$

hence, the release of the next job $\tau_{i,j+1}$ can be advanced at time

$$r_{i,j+1} = max(r_{i,j} + \frac{1}{f_i^{opt}}, \bar{d}_{i,j}, f_{i,j}),$$

where $f_{i,j}$ is the actual completion time of job $\tau_{i,j}$. When the new job $\tau_{i,j+1}$ arrives, the server budget is recharged at the maximum value Q_s and a new server deadline is generated at time $r_{i,j+1} + T_s$.

5 Handling Resource Constraints

We now extend the rate adaptation method to deal with resource constraints, thus allowing tasks to interact through shared memory locations. In order to estimate maximum blocking times due to mutual exclusion and analyze task schedulability, we assume that critical sections are accessed through the Stack Resource Policy (SRP) [4], but any protocol can be used for the purpose. For the sake of completeness, the features and the main properties of this protocol are briefly recalled in Appendix A.2.

5.1 Notes on Resource Sharing

When shared resources are accessed in mutual exclusion by tasks handled by a capacity-based server, an additional problem arises if the server exhausts its budget when a served task is inside a critical section. In order to prevent long blocking delays due to the budget replenishment rule, the task which exhausted its budget is allowed to continue executing with the same deadline (using extra budget) until it leaves the critical section. At this time, the budget can be replenished at its full value and the deadline postponed.

Since the server execution can be prolonged by an extra budget to allow the served task to complete the critical section, the server utilization has to be computed to take such a budget overrun into account. The maximum interference created by the budget overrun mechanism occurs when the server exhausts its budget immediately after the task entered its longest critical section. Thus, if ξ_i is the duration of the longest critical section of task τ_i handled by server S_i, the bandwidth demanded by the server becomes $\frac{Q_{s_i}+\xi_i}{T_{s_i}}$. Hence, a set of tasks in the presence of resource constraints can be guaranteed by verifying the following condition, which directly derives from the guarantee test proposed by Baker and reported in (16) in Appendix A.2:

$$\forall i,\ 1 \le i \le n \quad \sum_{k=1}^{i} \frac{Q_{s_k} + \xi_k}{T_{s_k}^{max}} + \frac{B_i}{T_{s_i}^{max}} \le 1, \tag{9}$$

where Q_{s_k} is the maximum budget of the k-th server, B_i is the blocking time computed assuming that each task τ_i is scheduled with a relative deadline equal to $T_{s_i}^{max} = Q_{s_i}/(\tilde{f}_i^{min} c_i^n)$, which is the server period computed when each server is assigned the minimum permitted bandwidth. A similar approach was first presented by Ghazalie and Baker in [12] for accounting for the blocking effects of critical sections in a number of dynamic aperiodic servers.

5.2 Computing Frequencies under Resource Constraints

In the presence of resource constraints, the SLSS algorithm must be extended to take blocking terms into account. In the proposed approach, we will use the following schedulability test:

$$\sum_{k=1}^{n} \frac{Q_{s_k} + \xi_k}{T_{s_k}} + \max_i \left(\frac{B_i}{T_{s_i}}\right) \le 1,$$

which is a simpler, but less tight, sufficient condition derived from condition (9). Assuming that relative deadlines are equal to the server periods, the extended SLSS algorithm must compute the optimal tasks' frequencies under the following optimization problem:

$$\min_{(f_1,\dots,f_n)} \Delta J = \sum_i w_i \Delta J_i(f_i), \tag{10}$$

subject to:

$$\sum_i f_i c_i^n + U_b \le A, \quad 0 < A \le 1, \tag{11}$$

$$f_i \ge \tilde{f}_i^{min}, \quad i = 1, \dots, n,$$

where $U_b = \sum_i \frac{f_i c_i^n \xi_i}{Q_{s_i}} + \max_k \frac{B_k}{T_{s_k}}$ and B_k is the blocking time computed assuming that each task τ_i is scheduled with a relative deadline equal to its server period $T_{s_k} = Q_{s_k}/(f_k c_k^n)$. Notice that each task τ_i is assigned a preemption level inversely proportional to its server period T_{s_i}; therefore, in order to assign each job chunk the same preemption level, the maximum server budget should be a submultiple of the worst-case execution time of the served task.

It is worth noting that, in the presence of resource sharing, a recursive constraint appears in the optimization problem stated above. In particular, (11) has a new term U_b where blocking times B_k are not constant, but they depend on the reciprocal period relations. As a consequence, the SLSS algorithm becomes an iterative algorithm. However, such an increase in complexity does not affect the runtime overhead, because the SLSS algorithm is executed offline.

In order to ensure that the SLSS algorithm will converge to a solution in a finite number of steps, the U_b term can be overestimated in such a way it becomes a monotonic nondecreasing function of the available bandwidth $U^{av} = A - U_b$. Let B_k^* denote the maximum blocking time of τ_k, computed as follows:

$$B_i^* = \max_{j,h}\{s_{jh} \mid (D_i < D_j^{max}) \ \wedge \ \pi_i^{min} \leq ceil(\rho_{jh})\}, \qquad (12)$$

where

$$D_i^{max} = T_{s_i}^{max} = \frac{Q_{s_i}}{f_i^{min} WCET_i}, \qquad \pi_i^{min} = \frac{1}{D_i^{max}}.$$

Then, we have that:

$$\forall i \ \ B_i^* \geq B_i.$$

Notice that the maximum blocking time of each task is a function of the available bandwidth U^{av}. In fact, if the available bandwidth increases, the SLSS algorithm will consequently increase the frequency of each task; hence, each task is assigned a shorter relative deadline (D_i) and resource ceilings might augment their value. From considerations above, it follows that the B_i^* term is a monotonic nondecreasing function of the available bandwidth, that is:

$$\forall i \ \ B_i^*(U_1^{av}) \leq B_i^*(U_2^{av}) \quad \Leftrightarrow \quad U_1^{av} < U_2^{av}. \qquad (13)$$

Using this approach, the term $U_b^* = \sum_i \frac{f_i c_i^n \xi_i}{Q_{s_i}} + \max_k \frac{B_k^*}{T_{s_k}}$ has the following property:

$$U_b^* \geq U_b.$$

Finally, from (13) and the definition of U_b^* term, it follows that:

$$U_b^*(U_1^{av}) \leq U_b^*(U_2^{av}) \quad \Leftrightarrow \quad U_1^{av} < U_2^{av}.$$

The property above ensures that in the presence of resource constraints the extended SLSS algorithm converges to a solution. The iterative version of the SLSS algorithm which takes resource constraints into account is shown in Fig. 9. The proposed algorithm is based on a binary search; at each step, the available bandwidth U^{av} is set equal to the average of the maximum permitted bandwidth U^{sup} and the actual required bandwidth U^{inf}. The bandwidth U^{av} is used to compute the tasks' frequencies with which the guarantee test is performed. If the test succeeds, the bandwidth U^{av} can be safely assigned to the task set, the actual required bandwidth U^{inf} is set equal to U^{av} and a new step is performed until the required precision is reached. If the test fails, the new maximum permitted bandwidth U^{sup} is set equal to U^{av} and the algorithm will try to assign less bandwidth for computing the task frequencies. The algorithm stops when the actual required bandwidth is maximized (within the feasibility constraint) by returning the set of frequencies. The algorithm precision depends on the value of the ERR constant, which must be carefully chosen to balance precision against computational complexity. The algorithm complexity is:

$$\mathcal{O}\left(max(mn, nlogn)log\left(\frac{U_s^{sup}(0) - U_s^{inf}(0)}{ERR}\right)\right),$$

where n is the number of tasks and m is the number of resources. The consequence of overestimating the U_b term is that the extended SLSS algorithm cannot exploit all the available bandwidth. However, if the feasibility test is satisfied, the algorithm always gives a solution which, in the worst case, consists in setting each task period equal to its maximum value.

```
**Frequency Computation Algorithm**
{
    $x = 0$;
    $\forall i \quad f_i = freq_i = \tilde{f}_i^{min}$;
    $U^{inf}(x) = \sum_i f_i c_i^n + U_b^*$;
    $U^{sup}(x) = A$;
    if ($U^{inf}(x) \geq U^{sup}(x)$)
        return(set of $f_i$);
    while ($U^{sup}(x) - U^{inf}(x) \geq$ ERR) {
        $U^{av}(x) = \frac{U^{sup}(x)+U^{inf}(x)}{2}$;
        for (each task $\tau_i$) compute frequencies $f_i$ with
            SLSS algorithm such that $\sum_i f_i c_i^n \leq U^{av}(x)$;
        for (each resource $\rho$) compute ceil($\rho$);
        for (each task $\tau_i$) compute $B_i^*$;
        if ($\sum_i f_i c_i^n + U_b^* \leq A$) {
            for (each task) $freq_i = f_i$;
            $U^{sup}(x+1) = U^{sup}(x)$;
            $U^{inf}(x+1) = U^{av}(x)$;
        }
        else {
            $U^{sup}(x+1) = U^{av}(x)$;
            $U^{inf}(x+1) = U^{inf}(x)$;
        }
        $x = x + 1$;
    }
    return(set of $f_i$)
}
```

Fig. 9. Iterative algorithm for computing each task frequency.

6 Performance Evaluation

The proposed algorithm has been implemented in the SHARK kernel [10] to verify the results predicted by the theory. In particular, a set of experiments has been performed to test the effectiveness of our method in enhancing the PLI in a set of periodic control tasks. Table 4 shows the parameters of the task set selected for this experiment. Each task τ_i has a normal computation time equal to its average computation time such that $\forall i \ \ c_i^n = C_i^{avg} = 0.7 WCET_i$. The optimal frequency f_i^{opt} represents the frequency computed by the Seto et al. algorithm assuming that each task has the same weight

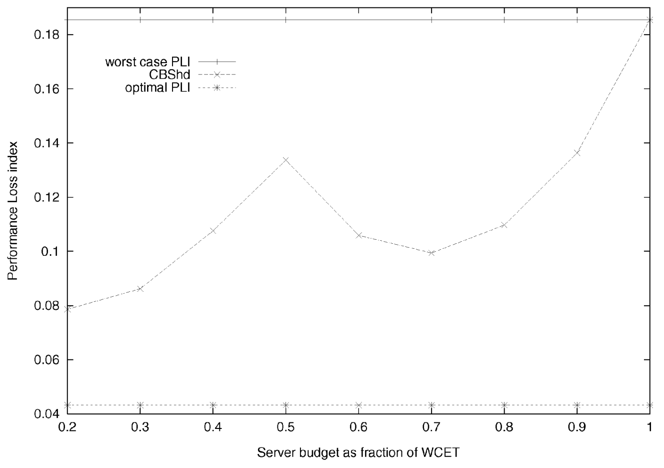


Fig. 10. PLI of a task set with $C^{avg} = 0.7WCET$.

$w_i = 1$, the same magnitude coefficient $\alpha_i = 1$ and the same decay rate $\beta_i = 0.4$.

The f_i^{min} parameters have been chosen to have the same value for each task just to simplify the interpretation of the optimal frequencies computed by the algorithm as a function of the nominal and worst-case computation times. Different values do not significantly affect the shape of the resulting graphs.

The minimum value of the PLI computed by the optimization algorithm is $\Delta J^{opt} = 0.0432$. Such a theoretical value, however, can be reached only if every instance of each task τ_i executes exactly for C_i^{avg}.

The performance of the algorithm was measured by computing the PLI of the task set as a function of the budget assigned to each server. For instance, a value of 0.9 on the x-axis means that each server has a maximum budget $Q_{s_i} = 0.9WCET_i$. Whenever the maximum budget is changed (on the x-axis), the server period is set according to the assigned bandwidth. Computation times have a uniform distribution and each computation time is obtained by splitting the whole execution time in a fixed part (C^{fix}) plus a random part (C^{rand}), where $C^{fix} = 2C^{avg} - WCET$ and C^{rand} is obtained by a uniform distribution in the interval $[0, WCET - C^{fix}]$.

Fig. 10 compares the CBS^{hd} with the optimal PLI (theoretical value) which is achieved in the ideal case in which all jobs execute for their average computation time. The worst-case PLI is also drawn as a reference value, which is obtained when every instance executes exactly for its $WCET$. The graph shows that no gain is obtained by the CBS^{hd} algorithm when the server budget is set equal to the task $WCET$. However, as the server budget decreases, the CBS^{hd} algorithm becomes more effective, improving the PLI. It is worth noting that the PLI has a peak for $Q_{s_i} = 0.5WCET_i$. This strange behavior is a direct consequence of the CBS^{hd} deadline postponement rule and can be explained as follows:

Whenever a deadline is postponed by the CBS^{hd}, the new deadline is increased by a server period. Let us focus, for instance, on task τ_5, when $Q_{s_5} = 0.5WCET_5 = 5ms$ and $T_5 = 50ms$ (being $U_5 = 0.1$). Since $C_5^{avg} = 0.7WCET_5 = 7ms$, during overruns the task period is increased up to $100ms$.

TABLE 4
Task Set Parameters

Task	$WCET_i$ (ms)	c_i^n (ms)	f_i^{opt} (Hz)	f_i^{min}
τ_1	25	17.5	11.85	5
τ_2	12.5	8.75	13.58	5
τ_3	38	26.6	10.8	5
τ_4	38	26.6	10.8	5
τ_5	10	7	14.14	5

This value is much greater than the average period of the task deriving from the allocated bandwidth ($P_5^{avg} = C_5^{avg}/U_5 = 70ms$). Such an effect becomes less significant for smaller values of the server budget. For example, when $Q_{s_5} = 0.4WCET_5 = 4ms$ and $T_5 = 40ms$, during overruns the task period becomes $80ms$, which is closer to the average period. The same consideration holds for the other tasks.

In conclusion, this experiment shows that, using the CBS^{hd} algorithm, the PLI can be improved by setting the server budget as a small fraction of the WCET. However, the algorithm yields good results also for server budgets equal to the task average computation times.

6.1 Considerations on Runtime Overhead

A final set of experiments has also been conducted to estimate the runtime overhead introduced by the CBS^{hd} algorithm. A quantitative analysis of the overhead is useful to provide a criterion for setting the servers' budgets since, as shown in Fig. 10, a small budget allows to improve the PLI, but increases the number of deadline postponements.

Our experiments have been performed on a Pentium 133 MHz using the task set shown in Table 4. Since in the SHARK kernel the scheduling algorithm executes in the context of the running task, the scheduling overhead has the effect of increasing the actual execution time of each task. Therefore, the overhead due to the CBS^{hd} algorithm has been measured as a difference between the average computation time $\overline{c}''$ computed with a budget equal to a small fraction of the WCET, so that the server deadline is postponed n times, in the average, and the average computation time $\overline{c}'$ of the longest task served with a budget equal to its WCET (so that no deadline is postponed). Hence, the overhead ω due to a single deadline postponement performed by the CBS^{hd} algorithm, resulted to be

$$\omega = \frac{\overline{c}'' - \overline{c}'}{n} = 42\mu s.$$

Notice that ω does not include the overhead due to preemption. To investigate the effects of the algorithm overhead on the PLI, the guarantee test has been modified to take the overhead into account. If Q_{s_i} is the budget assigned to server S_i, the net budget used by the task is $Q_{s_i}^{net} = Q_{s_i} - \omega$. Hence, the guarantee test can be rewritten as:

$$\sum_{i=1}^{n} \frac{Q_{s_i}^{net}}{T_i} \leq 1 - \omega \sum_{i=1}^{n} \frac{1}{T_i},$$

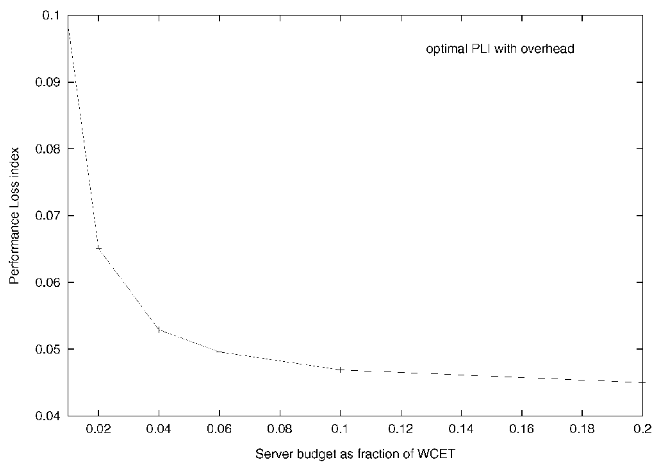


Fig. 11. Optimal PLI taking the scheduling overhead into account.

where $Q^{net}_{s_i}/T_i$ is bandwidth that must be assigned to task τ_i according to the Seto et al. algorithm to minimize the PLI. Since the overhead reduces the available bandwidth, the optimal PLI increases its value as the server budget is decreased.

Fig. 11 shows the optimal PLI (with overhead included) as a function of the budget assigned to each server. It is worth noting that the overhead effect is negligible up to $Q_s = 0.1WCET$; therefore, the server budget can be set equal to $0.2WCET$ for the task set of Table 4, obtaining a PLI very close to the optimal one (see Fig. 10).

As a final remark, we note that lower values of the budget could slightly improve the PLI; however, they cannot be easily assigned if the server budget becomes comparable with the time granularity of the kernel.

7 Engineering Considerations

In Section 3, it was explained how an overrun is handled when it occurs and how local handling can be guaranteed. The goal of the proposed method is to try to execute each task at its optimal frequency while avoiding the occurrence of too many overruns. In order to limit the number of overruns, it is important to use a good estimation of the normal computation times c^n_i. The system will be more flexible if we suppose that the new optimal frequencies are recomputed if some parameter changes online; in fact, if we suppose that a lower bound of c^n_i is provided by the user for each task when the system starts, then the number of overruns can be monitored online. Therefore, if a task τ_i has too many overruns, then its c^n_i can be increased according to a specific rule until the correct value is reached.

A heuristic approach defines a maximum number of overruns N^{max} permitted on a jobs' window W, and an increment factor Δ. Whenever the same task τ_i generates more than N^{max} overruns in a window of W jobs, its normal computation time c^n_i is increased by Δ and new optimal frequencies are computed by the SLSS algorithm.

In order to maintain task set schedulability during reconfiguration, the rate of each task has to change according to specific rules. In [7], Buttazzo et al. proposed a model where periodic tasks can intentionally change their execution rate to provide different quality of service and the other tasks can automatically adapt their periods to keep the system underloaded. They derived some theoretical results which permit changing the tasks' rate online. In fact, if we suppose that the total system utilization will change from U_p to U'_p and both utilization factors are less than or equal to one, the task set will remain schedulable and no deadline will be missed if each task will increase its frequency only at its next release time. They also proved that a task can decrease its frequency immediately.

Hence, the rate of each task can be changed online according to the previous rules; that is, each task τ_i will change its frequency f^{opt}_i and its normal computation time c^n_i at its next release time. By doing this, when the next job is released, the CBS^{hd} parameters (Q_s, T_s) will also change.

Another way to increase the system flexibility involves defining a QoS manager to provide different quality of service and handle overload situations in a more flexible way. In a situation where we need to schedule a task set composed of hard control tasks together with soft tasks (for instance monitoring tasks), a more general admission control mechanism can be used whenever a new soft task cannot be guaranteed by the system. Hence, instead of rejecting the soft task, the system can try to reduce the utilization of control tasks (by decreasing their frequencies in a controlled fashion) to reduce the total load and accommodate the new request. This approach is analogous to that one described by Buttazzo et al. in their elastic model [7]; the major difference is using the SLSS algorithm instead of their elastic algorithm in order to compute the new frequency of each task.

7.1 Application Level Implementation

The model described in the previous sections needs a real-time kernel able to monitor the computation time of each task in order to supply the information to a server like CBS^{hd}. In fact, whenever an overrun occurs, it must be detected by the kernel which also will compute a new deadline according to CBS^{hd} rules. However, some application classes do not need a kernel with such a feature, and the task itself can monitor its computation time and change its deadline by using suitable system calls. In order to use this approach, each task job is divided into chunks, and each chunk is characterized by a own computation time. Hence, at the end of each chunk, the task itself can test whether the computation has finished, or if another chunk has to execute in order to provide the result. Whenever another chunk has to start, the previous one computes the new chunk deadline before ending.

A control system which uses a camera as sensor to monitor the controlled environment can be implemented in this way. In fact, the task that reads the image memory filled by the camera frame grabber and determines the actual position of controlled object can be divided into chunks, each one scanning a different portion of the image. Each task job finishes when the object is found, and the whole image has to be scanned in the worst case.

More formally, according to this method, a task can be described by:

$$\tau_i(wcet_{i,1}, .., wcet_{i,N_i}, N_i, f_i^{min}, \Delta J_i(f_i)),$$

where $wcet_{i,k}$ is the worst-case execution time of kth chunk, N_i is the number of chunks the task is divided in, f_i^{min} is the minimum frequency τ_i can execute at, and $\Delta J_i(f_i)$ is the PLI of τ_i. Note that the worst-case execution time of each task τ_i becomes $WCET_i = \sum_{k=1}^{N_i} wcet_{i,k}$. In this simplified model, the guarantee test does not change; hence, the task set is guaranteed if and only if $\sum_i f_i^{min} WCET_i \leq 1$. By doing as described in Section 3, if the task set is feasible, a new lower-bound $\tilde{f}_i^{min}$ of frequency can be computed for each task τ_i in order to take overruns into account. Now, the parameter $\tilde{f}_i^{min}$ is defined as follows:

$$\forall\ \tau_i \quad \tilde{f}_i^{min} = \frac{f_i^{min} WCET_i}{wcet_{i,1}}.$$

Finally, the SLSS algorithm will be used to compute the optimal frequencies using $wcet_{i,1}$ as computation time (instead of $WCET_i$) and $\tilde{f}_i^{min}$ as minimum frequency (instead of f_i^{min}) for each task τ_i.

Note that each task τ_i has an utilization factor $U_i = f_i^{opt} wcet_{i,1}$. To simplify the notation, we will indicate all the chunks of a job with an increasing index k (hence, $H'_{i,j} = H_1$, $H''_{i,j} = H_2$ and so on). Supposing that H_k is the kth chunk of job $\tau_{i,j}$, d_k is the deadline of H_k and $a_{i,j}$ is the arrival time of $\tau_{i,j}$, the deadline of the first chunk H_1 is $d_1 = a_{i,j} + 1/f_i^{opt}$. Every time a job $\tau_{i,j}$ needs more than one chunk to finishing its computation, each next chunk H_k has a deadline d_k computed as follows:

$$\forall\ k, i \ \mid\ 2 \leq k \leq N_i \quad d_k = d_{k-1} + \frac{wcet_{i,k}}{U_i}.$$

Hence, the previous chunk can compute the new deadline and notify it to the kernel before ending. The major difference between this approach and the previous one described in Section 3, consists in handling the overrun at user level instead of kernel level.

8 Conclusions

In this paper, we presented a novel approach for increasing the efficiency of digital control systems in which the computation times of periodic activities have significant variations. The proposed method was proven to be particularly effective for those control activities, as visual tracking tasks, in which the worst-case computation time is much greater than the typical computation time required in normal operations.

The work presented in the paper integrates and extends two recent advances in real-time computing—the optimization of control performance subject to schedulability analysis and the Constant Bandwidth Server algorithm—to create an innovative approach to rate adaptation that allows us to fully utilize the processor to optimize the control performance and yet guarantee the schedulability of all tasks under worst-case conditions.

The rate adaptation method has been implemented in the SHARK kernel in order to evaluate its performance and validate our theoretical results. The experiments showed the effectiveness of the proposed method in enhancing the performance loss index through an increase of tasks' frequencies with respect to the classical worst-case design. It has been shown that using the CBS^{hd} algorithm, the PLI can be improved by setting the server budget as a small fraction of the WCET. However, the algorithm yields good results also for server budgets equal to the task average computation times. Specific tests also showed that the overhead introduced by the algorithm does not limit its use in real applications.

As a future work, we plan to investigate a technique for handling overruns globally, so that, when an overrun occurs, each task can decrease its frequency in order to create free bandwidth for handling the overrun.

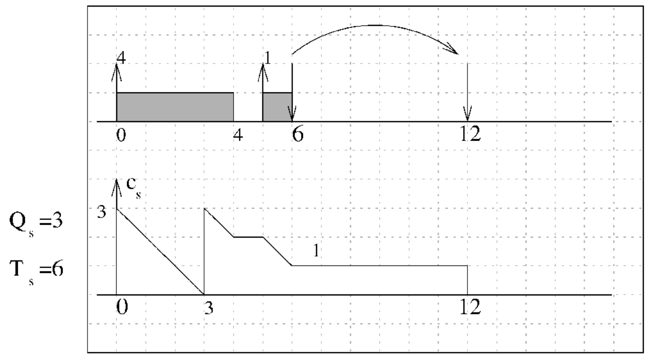


Fig. 12. Example of a CBS server.

Appendix A

A.1 The CBS Algorithm

A CBS is characterized by an ordered pair (Q_s, T_s), where Q_s is the maximum budget and T_s is the period of the server. The ratio $U_s = Q_s/T_s$ is denoted as the server bandwidth.

At each instant, a fixed deadline $d_{s,k}$ and a budget c_s is associated with the server. Every time a new job $J_{i,j}$ has to be served, it is assigned a dynamic deadline $d_{i,j}$ equal to the current server deadline $d_{s,k}$. The current budget c_s represents the amount of computation time schedulable by the CBS using the current server deadline. Whenever a served job executes, the budget c_s is decreased by the same amount and, every time $c_s = 0$, the server budget is recharged to the maximum value Q_s and a new server deadline is generated as $d_{s,k+1} = d_{s,k} + T_s$.

Fig. 12 illustrates an example in which two jobs J_1 and J_2 are served by a CBS having a budget $Q_s = 3$ and a period $T_s = 6$. The first job arrives at time $r_1 = 0$ and it is assigned a deadline $d_{s,1} = r_1 + T_s = 6$. Initially, c_s is equal to $Q_s = 3$; at time $t = 3$, the budget is exhausted, so a new deadline $d_{s,2} = d_{s,1} + T_s = 12$ is generated and c_s is replenished. At time $t = 4$, J_1 finishes and the server budget c_s is equal to two units; hence, the CBS is still available to schedule two units of computation time using the same server deadline $d_{s,2}$. At time $r_2 = 5$, the second job arrives and is served with the actual server deadline $(d_{s,2} = 12)$.

A.2 The Stack Resource Policy

The Stack Resource Policy (SRP) is a concurrency control protocol proposed by Baker [4] to bound the priority inversion phenomenon in static as well as dynamic priority systems. Under the EDF scheduling algorithm, each task τ_i is assigned a dynamic priority p_i inversely proportional to its absolute deadline d_i and a static *preemption level* π_i, such that the following property holds:

Property 1. *Task τ_i is not allowed to preempt task τ_j, unless $\pi_i > \pi_j$.*

Under EDF, Property 1 is verified if each periodic task is assigned a preemption level inversely proportional to its relative deadline D_i. That is,

$$\pi_i \propto \frac{1}{D_i}.$$

In addition, every resource R_k is assigned a static[3] *ceiling* defined as

$$ceil(R_k) = \max_i\{\pi_i \mid \tau_i \;\; needs \;\; R_k\}, \tag{14}$$

and a dynamic *system ceiling* is defined as

$$\Pi_s(t) = \max[\{ceil(R_k) \mid R_k \text{ is currently busy}\} \cup \{0\}].$$

Then, the SRP scheduling rule states that

A task is not allowed to preempt until its priority is the highest among those of the active tasks and its preemption level is greater than the system ceiling.

Such a protocol guarantees that each task can be blocked for at most the duration of one critical section. Moreover, it ensures that, once a task is started, it will never block until completion; it can only be preempted by higher priority tasks. As a consequence, the blocking time B_i considered in the schedulability analysis refers to the time in which task τ_i is kept in the ready queue by the preemption test, waiting for tasks with lower preemption levels to free shared resources. Blocking at preemption time also allows tasks to share a single stack, so reducing the total stack size when more tasks have the same preemption level. Finally, the SRP implementation is straightforward and there is no need to implement semaphore queues since a task never blocks during execution, but simply cannot preempt if its preemption level is not high enough.

Using SRP, the maximum blocking time for a task τ_i is bounded by the duration of the longest critical section among those that can block τ_i; that is, those with a ceiling greater than or equal to π_i belonging to tasks with a relative deadline greater than D_i:

$$B_i = \max_{j,h}\{s_{jh} \mid (D_i < D_j) \;\; \wedge \;\; \pi_i \le ceil(\rho_{jh})\}, \tag{15}$$

where s_{jh} is the worst-case execution time of the hth critical section of task τ_j and ρ_{jh} is the resource accessed in the critical section s_{jh}. Then, the feasibility of a task set with resource constraints can be tested by the following sufficient condition [4]:

$$\forall i, \;\; 1 \le i \le n \quad \sum_{k=1}^{i} \frac{C_k}{D_k} + \frac{B_i}{D_i} \;\le\; 1, \tag{16}$$

where it is assumed that tasks are sorted by decreasing preemption levels, so that $\pi_i \ge \pi_j$ only if $i < j$.

A simpler, but less tight, sufficient condition to verify the schedulability of a task set in the presence of resource constraints can be derived from condition (16):

$$\sum_{i=1}^{n} \frac{C_i}{D_i} + \max_k\left(\frac{B_k}{D_k}\right) \;\le\; 1. \tag{17}$$

A tighter test can be performed (in pseudopolynomial time) using a processor demand criterion [5]. However, in order to keep the overhead low, we decided to perform the SLSS algorithm using the simplest test given by (17).

3. In the case of multiunits resources, the ceiling of each resource is dynamic as it depends on the number of units actually free.

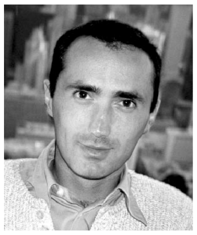
References

[1] T.F. Abdelzaher, E.M. Atkins, and K.G. Shin, "QoS Negotiation in Real-Time Systems and Its Application to Automated Flight Control," *Proc. IEEE Real-Time Technology and Applications Symp.*, June 1997.
[2] L. Abeni and G. Buttazzo, "Integrating Multimedia Applications in Hard Real-Time Systems," *Proc. IEEE Real-Time Systems Symp.*, Dec. 1998.
[3] L. Abeni and G. Buttazzo, "QoS Guarantee Using Probabilistic Deadlines," *IEEE Proc. 11th Euromicro Conf. Real-Time Systems*, pp. 242-249, June 1999.
[4] T.P. Baker, "Stack-Based Scheduling of Real-Time Processes," *J. Real-Time Systems*, vol. 3, no. 1, pp. 67-100, 1991.
[5] S.K. Baruah, R.R. Howell, L.E. Rosier, "Algorithms and Complexity Concerning the Preemptive Scheduling of Periodic Real-Time Tasks on One Processor," *J. Real-Time Systems*, vol. 2, 1990.
[6] G. Beccari, S. Caselli, M. Reggiani, and F. Zanichelli, "Rate Modulation of Soft Real-Time Tasks in Autonomous Robot Control Systems," *IEEE Proc. 11th Euromicro Conf. Real-Time Systems*, June 1999.
[7] G. Buttazzo, G. Lipari, and L. Abeni, "Elastic Task Model for Adaptive Rate Control," *Proc. IEEE Real-Time Systems Symp.*, Dec. 1998.
[8] M. Caccamo, G. Lipari, and G. Buttazzo, "Sharing Resources among Periodic and Aperiodic Tasks With Dynamic Deadlines," *Proc. IEEE Real-Time Systems Symp.*, Dec. 1999.
[9] M. Caccamo, G. Buttazzo, and L. Sha, "Elastic Feedback Control," *IEEE Proc. 12th Euromicro Conf. Real-Time Systems*, pp. 121-128, June 2000.
[10] P. Gai, L. Abeni, M. Giorgi, and G. Buttazzo, "A New Kernel Approach for Modular Real-Time systems Development," *Proc. 13th IEEE Euromicro Conf. Real-Time Systems*, June 2001.
[11] M.K. Gardner and J.W.S. Liu, "Performance of Algorithms for Scheduling Real-time Systems with Overrun and Overload," *IEEE Proc. 11th Euromicro Conf. Real-Time Systems*, June 1999.
[12] T.M. Ghazalie and T.P. Baker, "Aperiodic Servers in a Deadline Scheduling Environment," *J. Real-Time System*, vol. 9, 1995.
[13] J. Leung and J. Whitehead, "On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks," *Performance Evaluation*, vol. 2, pp 237-250 1982.
[14] C.L. Liu and J.W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *J. ACM*, vol. 20, no. 1, pp. 40-61, 1973.
[15] M. Ryu and S. Hong, "Toward Automatic Synthesis of Schedulable Real-Time Controllers," *J. Integrated Computer Aided Eng.*, vol. 5, no. 3, pp. 261-277, 1998.
[16] D. Seto, J.P. Lehoczky, L. Sha, and K.G. Shin, "On Task Schedulability in Real-Time Control System," *Proc. IEEE Real-Time Systems Symp.*, Dec. 1996.
[17] K.G. Shin and C.L. Meissner, "Adaptation and Graceful Degradation of Control System Performance by Task Reallocation and Period Adjustment," *IEEE Proc. 11th Euromicro Conf. Real-Time Systems*, June 1999.
[18] M. Spuri and G.C. Buttazzo, "Efficient Aperiodic Service under Earliest Deadline Scheduling," *Proc. IEEE Real-Time Systems Symp.*, Dec. 1994.
[19] M. Spuri, G. Buttazzo, and F. Sensini, "Robust Aperiodic Scheduling Under Dynamic Priority Systems," *Proc. 16th IEEE Real-Time Systems Symp.*, Dec. 1995.

[20] M. Spuri and G.C. Buttazzo, "Scheduling Aperiodic Tasks in Dynamic Priority Systems," *J. Real-Time Systems,* vol. 10, no. 2, 1996.

[21] J.A. Stankovic, C. Lu, S. Son, and G. Tao, "The Case for Feedback Control Real-Time Scheduling," *IEEE Proc. 11th Euromicro Conf. Real-Time Systems,* June 1999.

[22] C. Lu, J.A. Stankovic, G. Tao, and S.H. Son, "Design and Evaluation of a Feedback Control EDF Scheduling Algorithm," *Proc. IEEE Real-Time Systems Symp.,* Dec. 1999.



Marco Caccamo is a PhD student in computer engineering at the Scuola Superiore S. Anna of Pisa, Italy. In 1997, he graduated with a degree in computer engineering at the University of Pisa. During 1999, he was a visiting scholar at the University of Illinois (UIUC), working with Professor Lui Sha on novel scheduling algorithms for real-time control applications. His reasearch activity is focused on the development and analysis of flexible scheduling algorithms for real-time systems. His research interests include real-time operating systems, scheduling algorithms, fault-tolerant systems, quality of service control, and multimedia applications.

Giorgio C. Buttazzo graduated with a degree in electronic engineering at the University of Pisa in 1985, received a master's degree in computer science at the University of Pennsylvania in 1987, and a PhD degree in computer engineering at the Scuola Superiore S. Anna of Pisa in 1991. He is an associate professor of computer engineering at the University of Pavia, Italy. His main research interests include real-time operating systems, dynamic scheduling algorithms, quality of service control, multimedia systems, advanced robotics applications, and neural networks. He is a member of the IEEE.

Lui Sha received the PhD degree from Carnegie-Mellon University (CMU) in 1985. He is professor of computer science at the University of Illinois at Urbana Champaign. He was the chair of IEEE Real Time Systems Technical Committee from 1999 to 2000. He was elected to be an IEEE Fellow in 1998 "for technical leadership and research contributions, which enabled the transformation of real-time computing practice from an ad hoc process to an engineering process based on analytic methods." He is interested all aspects of distributed real time systems.