Capacity Sharing for Overrun Control

Marco Caccamo
Scuola Superiore S. Anna
Pisa (Italy)
caccamo@sssup.it

Giorgio Buttazzo
University of Pavia (Italy)
INFM Research Unit
giorgio@sssup.it

Lui Sha
University of Illinois
Urbana, IL 61801
lrs@cs.uiuc.edu

Abstract

In this paper, we present a general scheduling methodology for managing overruns in a real-time environment, where tasks may have different criticality and flexible timing constraints. The proposed method achieves isolation among tasks through a resource reservation mechanism which bounds the effects of task interference, but also performs efficient reclaiming of the unused computation times to relax the utilization constraints imposed by isolation. The enhancements achieved by the proposed approach resulted to be very effective with respect to classical reservation schemes. The performance has been evaluated by implementing the algorithm on a real-time kernel. The runtime overhead introduced by the scheduling mechanism has also been investigated with specific experiments, in order to be taken into account in the schedulability analysis. However, it resulted to be negligible in most practical cases.

1. Introduction

In most real-time systems, predictability is achieved by enforcing timing constraints on application tasks, whose feasibility is guaranteed off line by means of proper schedulability tests based on worst-case execution time (WCET) estimations. Theoretically, such an approach works fine if all the tasks have a regular behavior and all WCETs are precisely estimated. In practical cases, however, a precise estimation of WCETs is very difficult to achieve, because several low level mechanisms present in modern computer architectures (such as interrupts, DMA, pipelining, caching, and prefetching) introduce a form of non deterministic behavior in tasks' execution, whose duration cannot be predicted in advance.

Even though a precise WCET estimation could be derived for each task, a worst-case feasibility analysis would be very inefficient when task execution times have a high variance. In this case, a classical off-line hard guarantee would waste the system's computational resources for preserving the task set feasibility under sporadic peak load situations, even though the average workload is much lower. Such a waste of resources (which increases the overall system's cost) can be justified for very critical applications (e.g., military defense systems or safety critical space missions), in which a single deadline miss may cause catastrophic consequences. However, it does not represent a good solution for those applications (the majority) in which several deadline misses can be tolerated by the system, as long as the average task rates are guaranteed off line. There are many soft real-time applications in which the worst-case duration of some tasks is rare but much longer than the average case. In multimedia systems, for instance, the time for decoding a video frame in MPEG players can vary significantly as a function of the data contained in the previous frames. As another example, consider a visual tracking system where, in order to increase responsiveness, the moving target is searched in a small window centered in a predicted position, rather than in the entire visual field. If the target is not found in the predicted area, the search has to be performed in a larger region until, eventually, the entire visual field is scanned in the worst-case. If the system is well designed, the target is found very quickly in the predicted area most of the times. Thus, the worst-case situation is very rare, but very expensive in terms of computational resources (computation time increases quadratically as a function of the number of trials). In this case, an off-line guarantee based on WCETs would drastically reduce the frequency of the tracking task, causing a severe performance degradation with respect to a soft guarantee based on the average execution time. On the other hand, uncontrolled

overruns[1] are very dangerous if not properly handled, since they may heavily interfere with the execution of other tasks, which could be more critical. Consider for example the task set given in Table 1, where two tasks, τ_1 and τ_2, have a constant execution time, whereas τ_3 has an average computation time ($C_3^{avg} = 3$) much lower than its worst-case value ($C_3^{max} = 10$). Here, if the schedulability analysis is performed using the average computation time C_3^{avg}, the total processor utilization becomes 0.92, meaning that the system is not overloaded; however, under the Earliest Deadline First (EDF) algorithm [9] the tasks can experience long delays during overruns, as illustrated in Figure 1. Similar examples can easily be found also under fixed priority assignments (e.g., under the Rate Monotonic algorithm [9]), when overruns occur in the high priority tasks.

Task	C_i^{avg}	C_i^{max}	T_i
τ_1	1	1	6
τ_2	5	5	10
τ_3	3	10	12

Table 1. Task set parameters.

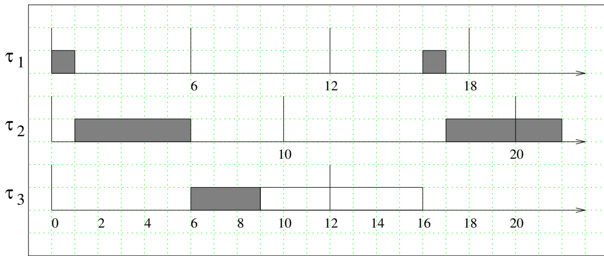

Figure 1. Negative effects of uncontrolled overruns.

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

[1] A task is said to overrun when it executes for more than its guaranteed execution time.

A general technique for limiting the effects of overruns is based on a resource reservation approach [10, 16, 1], according to which each task is assigned (off line) a fraction of the available resources and is handled by a dedicated server, which prevents the served task from demanding more than the reserved amount. Although such a method is essential for achieving predictability in the presence of tasks with variable execution times, the overall system's performance becomes quite dependent on a correct resource allocation. For example, if the CPU bandwidth allocated to a task is much less than its average requested value, the task may slow down too much, degrading the system's performance. On the other hand, if the allocated bandwidth is much greater than the actual needs, the system will run with low efficiency, wasting the available resources.

To overcome this problem, we propose a general scheduling methodology for managing overruns in a controlled fashion. In particular, the proposed technique allows to

- achieve isolation among tasks, through a resource reservation mechanism which bounds the effects of task overruns;
- perform efficient reclaiming of the unused computation times, through a global capacity sharing mechanism which allows to exploit early completions, in order to relax the bandwidth constraints enforced by isolation;
- handle tasks with different criticality and flexible timing constraints, to enhance the performance of those real-time applications which allow a certain degree of flexibility.

Although the idea of resource reclaiming and capacity sharing is not new in the literature, as discussed in Section 6 on related work, the peculiarity of our method is to increase resource utilization while preserving isolation, so that not only soft tasks, but also hard real-time tasks can benefit from our approach. Moreover, unlike other similar approaches, our method was not developed for enhancing aperiodic responsiveness of soft tasks, but to efficiently handle overruns in real-time (hard and soft) tasks, where some form of relaxed guarantee is required off line.

As a final remark, performance experiments on the algorithm (illustrated in Section 5) show that the run-time overhead introduced by the mechanism is negligible in most of practical cases and can be controlled through the amount of budget assigned to each server.

The rest of the paper is organized as follows: Section 2 describes the basic idea behind the proposed

approach; Section 3 illustrates the capacity sharing algorithm; Section 4 presents some theoretical results which validate the proposed model; Section 5 illustrates some experimental results; Section 6 presents the related work; and Section 7 contains our conclusions and future work.

2. Basic concepts

Throughout the paper, each task τ_i is considered as a stream of jobs (or task instances) $\tau_{i,j}$ $(j = 1, 2, \ldots)$, each characterized by a request time $r_{i,j}$, an execution time $c_{i,j}$, and a deadline $d_{i,j}$. In the following, P_i denotes the desired activation period of the task, C_i^{max} its maximum computation time, and C_i^{avg} its average computation time.

In the proposed approach, each task is handled by a dedicated *Constant Bandwidth Server* (CBS) [1], which provides isolation among tasks, and a capacity sharing mechanism allows tasks to reclaim the unused computations due to early completions. Due to the isolation mechanism introduced by the multiple server approach, there are no particular restrictions on the task model that can be handled by the proposed method. Hence, tasks can be hard, soft, periodic, or aperiodic. Although the method is built upon on the CBS, it can easily be generalized to be used with any capacity-based server.

In the following section we will briefly recall the CBS algorithm and its main properties.

2.1. 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 $\tau_{i,j}$ 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 $d_{s,k+1} = d_{s,k} + T_s$.

Figure 2 illustrates an example in which a task τ_1, with maximum computation time $C_1^{max} = 2$ and period $P_1 = 5$, is scheduled by EDF together with another task τ_2, served by a CBS having a budget $Q_s = 3$ and a period $T_s = 6$. Initially, $c_s = 0$ and $d_{s,0} = 0$. When job $\tau_{2,1}$ (requiring 5 units of computation) arrives at

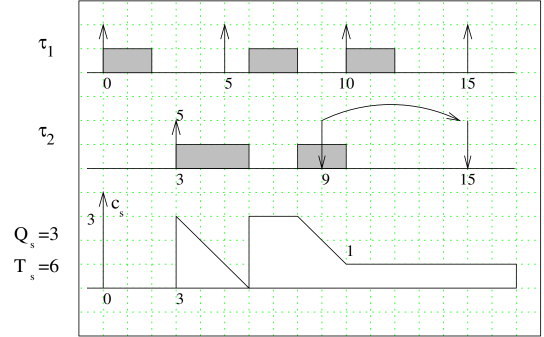

Figure 2. Example of a CBS server.

time $t = 3$, c_s is charged at the value $Q_s = 3$ and the job is assigned a deadline $d_{s,1} = t + T_s = 9$. At time $t = 6$, the budget is exhausted, so c_s is replenished and a new deadline $d_{s,2} = d_{s,1} + T_s = 15$ is generated by the server and assigned to job $\tau_{2,1}$.

In [1] it is proved that in any interval of time of length L a CBS with bandwidth U_s will never demand more than $U_s L$, independently from the actual task requests. Such a property allows us to use a bandwidth reservation strategy to allocate a fraction of the CPU time to soft tasks whose computation time cannot be easily bounded. The most important consequence of this result is that such tasks can be scheduled together with hard tasks without affecting the a priori guarantee, even in the case in which soft requests exceed the expected load.

2.2. The capacity sharing approach

The capacity sharing (CASH) mechanism proposed in this paper works in conjunction with the CBS. To illustrate the idea behind our approach, we present an example to show the potential improvements that can be achieved by a proper exploitation of the unused computation times coming from early completions.

Ideally, we would like to reserve a given bandwidth to each task, to achieve isolation; but we would also like to reclaim the unused time left by the other tasks as much as possible, so giving a chance to a task to handle its overruns without introducing long delays.

Consider the example shown in Figure 3, where three tasks are handled by three servers with budgets $Q_1 = 1$, $Q_2 = 5$, $Q_3 = 3$, and periods $T_1 = 4$, $T_2 = 10$, $T_3 = 12$, respectively. At time $t = 6$, job $\tau_{2,1}$ completes earlier with respect to the allocated budget, whereas job $\tau_{3,1}$ requires one extra unit of time. Figure 3a illustrates the case in which no reclaiming is used and tasks are served by the plain CBS algorithm. Notice

that, in spite of the budget saved by $\tau_{2,1}$, the third server is forced to postpone its current deadline when its budget is exhausted (it happens at time $t = 9$). As shown in Figure 3b, however, we observe that the spare capacity saved by $\tau_{2,1}$ can be used by $\tau_{3,1}$ to advance its execution and prevent the server from postponing its deadline.

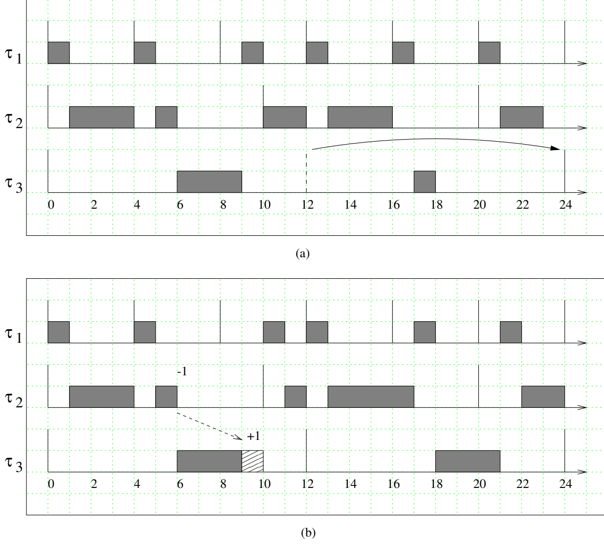

Figure 3. Overruns handled by a plain CBS (a) versus overruns handled by a CBS with the CASH reclaiming mechanism (b).

The reclaiming mechanism working with the CBS uses a global queue, the CASH queue, of spare capacities ordered by deadline. Whenever a task completes its execution and its server budget is greater than zero, the residual capacity can be used by any active task to advance its execution. When using a spare capacity, the task can be scheduled using the current deadline of the server which the spare capacity belongs to. In this way, each task can use its own capacity along with the residual capacities deriving by the other servers.

Whenever a new task instance is scheduled for execution, the server tries to use the residual capacities with deadlines less than or equal to the one assigned to the served instance; if these capacities are exhausted and the instance is not completed, the server starts using its own capacity. Every time a task ends its execution and the server becomes idle, the residual capacity (if any) is inserted with its deadline in the global queue of available capacities. Spare capacities are ordered by deadline and are consumed according to an EDF policy. The main benefit of the proposed reclaiming mechanism is to reduce the number of deadline shifts, so executing periodic activities with more stable frequencies.

This method, although developed for overrun control, can also be very effective in different contexts; for example, for improving the average response times of the served tasks, enhancing the performance of control applications, or increasing dependability in fault-tolerant real-time systems using recovery strategies under time redundancy. In such systems, in fact, an efficient reclaiming mechanism is important to exploit the unused computation time of backup copies whose primaries ended successfully.

3. The algorithm

In this section we formally describe the CASH algorithm assuming that each task τ_i is handled by a dedicated CBS server S_i running on a uniprocessor system. Capacity reclaiming is performed through the use of a global queue, called the CASH queue, containing all the residual capacities ordered by deadlines.

3.1. Algorithm rules

The CASH algorithm can be defined as follows:

1. Each server S_i is characterized by a budget c_i and by an ordered pair (Q_i, T_i), where Q_i is the maximum budget and T_i is the period of the server. The ratio $U_i = Q_i/T_i$ is denoted as the server bandwidth. At each instant, a fixed deadline $d_{i,k}$ is associated with the server. At the beginning $\forall i,\ d_{i,0} = 0$.
2. Each task instance $\tau_{i,j}$ handled by server S_i is assigned a dynamic deadline equal to the current server deadline $d_{i,k}$.
3. A server S_i is said to be active at time t if there are pending instances. A server is said to be idle at time t if it is not active.
4. When a task instance $\tau_{i,j}$ arrives and the server is idle, the server generates a new deadline $d_{i,k} = max(r_{i,j}, d_{i,k-1}) + T_i$ and c_i is recharged at the maximum value Q_i.
5. When a task instance $\tau_{i,j}$ arrives and the server is active the request is enqueued in a queue of pending jobs according to a given (arbitrary) discipline.
6. Whenever instance $\tau_{i,j}$ is scheduled for execution, the server S_i uses the capacity c_q in the CASH queue (if there is one) with the earliest deadline d_q, such that $d_q \le d_{i,k}$, otherwise its own capacity c_i is used.

7. Whenever job $\tau_{i,j}$ executes, the used budget c_q or c_i is decreased by the same amount. When c_q becomes equal to zero, it is extracted from the CASH queue and the next capacity in the queue with deadline less than or equal to $d_{i,k}$ can be used.

8. When the server is active and c_i becomes equal to zero, the server budget is recharged at the maximum value Q_i and a new server deadline is generated as $d_{i,k} = d_{i,k-1} + T_i$.

9. When a task instance finishes, the next pending instance, if any, is served using the current budget and deadline. If there are no pending jobs, the server becomes idle, the residual capacity $c_i > 0$ (if any) is inserted in the CASH queue with deadline equal to the server deadline, and c_i is set equal to zero.

10. Whenever the processor becomes idle for an interval of time Δ, the capacity c_q (if exists) with the earliest deadline in the CASH queue is decreased by the same amount of time until the CASH queue becomes empty.

3.2. An example

To better understand the proposed approach, we will describe a simple example which shows how our reclaiming algorithm works. Consider a task set consisting of two periodic tasks, τ_1 and τ_2, with periods $P_1 = 4$ and $P_2 = 8$, maximum execution times $C_1^{max} = 4$ and $C_2^{max} = 3$, and average execution times $C_1^{avg} = 3$ and $C_2^{avg} = 2$. Each task is scheduled by a dedicated CBS having a period equal to the task period and a budget equal to the average execution time. Hence, a task completing before its average execution time saves some budget, whereas it experiences an overrun if it completes after. A possible execution of the task set is reported in Figure 4, which also shows the capacity of each server and the residual capacities generated by each task. At time $t = 2$, task τ_1 has an early completion and a residual capacity equal to one with deadline equal to 4 becomes available. After that, τ_2 consumes the above residual capacity before starting to use its own capacity; hence, at time $t = 4$, a τ_2 overrun is handled without postponing its deadline. Notice that each task tries to use residual capacities before using its own capacity and that whenever an idle interval occurs (see interval [19, 20]), the residual capacity with the earliest deadline has to be discharged by the same amount in order to handle the residual capacities correctly.

The example above shows that overruns can be handled efficiently without postponing any deadline. A classical CBS instead, would postpone some deadlines in order to guarantee tasks isolation. Clearly, if all the tasks consume their allocated budget, no reclaiming can be done and our approach performs the same as a plain CBS. However, this situation is very rare, hence our approach helps in improving the average system's performance.

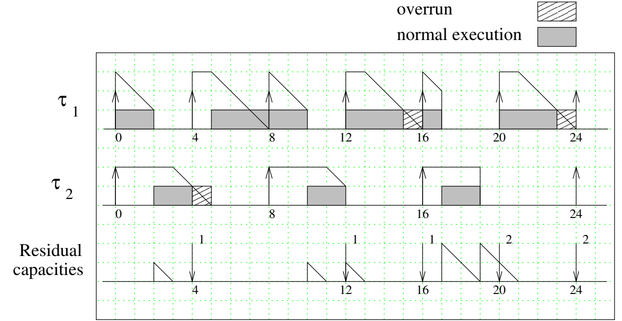

Figure 4. Example of global resource reclaiming.

4. Theoretical validation of the model

In this section we analyze the schedulability condition for a hybrid task set consisting of hard and soft periodic tasks. Each is scheduled using a dedicated CBS. If each hard periodic task is scheduled by a server with maximum budget equal to the task wcet and with period equal to the task period, it behaves like a standard hard task scheduled by EDF. The difference is that each task can gain and use extra capacities and yields its residual capacity to other tasks. This run-time exchange, however, does not affect schedulability; thus, the task set can be guaranteed using the classical Liu and Layland condition:

$$\sum_{i=1}^{n} \frac{Q_i}{T_i} \leq 1,$$

where Q_i is the maximum server budget and T_i is the server period. Before proving the schedulability condition, the following lemma will prove that all the generated capacities are exhausted before their respective deadlines.

Lemma 1 *Given a set Γ of capacity based servers along with the CASH algorithm, each capacity generated during the scheduling is exhausted before its dead-*

line if and only if:

$$\sum_{i=1}^{n} \frac{Q_i}{T_i} \leq 1, \quad (1)$$

where Q_i is the maximum server budget and T_i is the server period.

Proof.

If. Assume equation (1) holds and suppose that a capacity c^* is not exhausted at time t^*, when the corresponding deadline is reached. Let $t_a \geq 0$ be the last time before t^* at which no capacity is discharging; that is, the last instant before t^* during which the CPU is idle and the CASH queue is empty (if there is no such time, set $t_a = 0$). Let $t_b \geq 0$ be the last time before t^* at which a capacity with deadline after t^* is discharging (if there is no such time, set $t_b = 0$). If we take $t = max(t_a, t_b)$, time t has the property that only capacities created after t and with deadline less than or equal to t^* are used during $[t, t^*]$. Let $Q_T(t_1, t_2)$ be the sum of capacities created after t_1 and with deadline less than or equal to t_2; since a capacity misses its deadline at time t^*, it must be that:

$$Q_T(t, t^*) > (t^* - t)$$

In the interval $[t, t^*]$, we can write that:

$$(t^* - t) < Q_T(t, t^*) \leq \sum_{i=1}^{n} \left\lfloor \frac{t^* - t}{T_i} \right\rfloor Q_i \leq (t^* - t) \sum_{i=1}^{n} \frac{Q_i}{T_i},$$

which is a contradiction.

Only if. Suppose that $\sum_i \frac{Q_i}{T_i} > 1$. Then, we show there exists an interval $[t_1, t_2]$ in which $Q_T(t_1, t_2) > (t_2 - t_1)$. Assume that all the servers are activated at time 0; then, for $L = lcm(T_1, ..., T_n)$ we can write that:

$$Q_T(0, L) = \sum_{i=1}^{n} \left\lfloor \frac{L}{T_i} \right\rfloor Q_i = \sum_{i=1}^{n} \frac{L}{T_i} Q_i = L \sum_{i=1}^{n} \frac{Q_i}{T_i} > L,$$

hence, the "*only if* condition" follows. □

We now formally prove the schedulability condition with the following theorem:

Theorem 1 *Let $\mathcal{T}_h$ be a set of periodic hard tasks, where each task τ_i is scheduled by a dedicated server with $Q_i = C_i^{max}$ and $T_i = P_i$, and let $\mathcal{T}_s$ be a set of soft tasks scheduled by a group of servers with total utilization U^{soft}. Then, $\mathcal{T}_h$ is feasible if and only if*

$$\sum_{\tau_i \in \mathcal{T}_h} \frac{Q_i}{T_i} + U^{soft} \leq 1, \quad (2)$$

Proof.
The theorem follows immediately from Lemma 1; in fact, we can notice that each hard task instance has available at least its own capacity equal to the task WCET. Lemma 1 states that each capacity is always discharged before its deadline, hence it follows that each hard task instance has to finish by its deadline. □

It is worth noting that Theorem 1 also holds under a generic capacity-based server having a periodic behavior and a bandwidth U_s.

5. Performance evaluation

The CASH algorithm has been implemented in the HARTIK kernel [6] to measure the performance gain introduced by the capacity sharing mechanism and to verify the results predicted by the theory. In particular, we performed our experiments on a set of control tasks with the objective of minimizing a performance cost function under the schedulability constraints imposed by the system. Before illustrating the achieved results, we will briefly introduce some basic concepts useful for better understanding the experiments.

5.1. Background concepts

In a digital control system, the overall control performance is a function of the sampling rates used by the control tasks: the higher the rates, the better the performance. However, task rates are limited by the total processor utilization, which must be less than a maximum value in order to have a feasible schedule. On the other hand, each task τ_i is characterized by a minimum frequency f_i^{min}, below which the performance is unacceptable and the control becomes unstable. In this sense, $1/f_i^{min}$ represents a hard relative deadline for task τ_i.

A performance cost function can be defined using the concept of *Performance Loss Index* (PLI), originally introduced in [13], to measure the difference between a digital and continuous control as a function of the sampling frequency. In particular, if J and $J_D(f)$ are the performance indices generated by a continuous-time control and its digital implementation at a sampling frequency f, a PLI can be defined as $\Delta J(f) = |J_D(f) - J|$.

As noted in [12], $\Delta J(f)$ is convex and monotonically decreasing with the frequency, hence, for each control task, it can be 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. The PLI of the overall system $\Delta J(f_1, ..., f_n)$ can then be defined as $\Delta J(f_1, ..., f_n) = \sum_i w_i \Delta J_i(f_i)$, where w_i is a design parameter determined from the application (it can be considered a weight related to task's importance).

Ideally, we would like to minimize the overall PLI by increasing task rates as much as possible during normal load conditions, but we would also like to guarantee each task its minimum frequency, in the worst-case scenario.

To guarantee a minimum frequency to each task, we use the algorithm proposed in [12], able to determine a set of optimal rates f_i^{opt} which minimize a given PLI under a set of hard schedulability constraints.

To improve the average system's performance, a less pessimistic analysis was proposed in [3] to increase the task rates. According to this method, each task is assigned a normal computation time $c_i^n \leq C_i^{max}$ to perform a relaxed guarantee in normal load conditions (c_i^n can be set equal to C_i^{avg} or to a different value computed by analyzing the probabilistic distribution of the task computation time). Moreover, each task can dynamically change its frequency depending on the current load. If $d_{i,j}$ is the deadline used by the server to schedule job $\tau_{i,j}$, the next job $\tau_{i,j+1}$ will start at time

$$r_{i,j+1} = max(d_{i,j},\ r_{i,j} + \frac{1}{f_i^{opt}}), \qquad (3)$$

where f_i^{opt} is the optimal frequency computed by the Seto et al. algorithm [12] for task τ_i, assuming τ_i needs an execution time equal to its normal computation time. Hence, each job $\tau_{i,j}$ has a variable period $P_{i,j} = r_{i,j+1} - r_{i,j}$.

5.2. Experimental results

A set of experiments has been performed to verify the effectiveness of the CASH algorithm in enhancing the PLI in a set of periodic control tasks. Table 2 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 $\forall i \;\; c_i^n = C_i^{avg} = 0.7C_i^{max}$. The optimal frequency f_i^{opt} represents the frequency computed by the Seto et al. algorithm [12] assuming that each task has the same weight $w_i = 1$, the same magnitude coefficient $\alpha_i = 1$ and the same decay rate $\beta_i = 0.4$. The f_i^{min} value represents the minimum frequency each task can run at (following the elastic control model described in the above section). 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}. Notice that each task has a reserved bandwidth $U_i = c_i^n f_i^{opt}$.

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 $Q_i = 0.9C_i^{max}$. 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 $C^{fix} = 2C^{avg} - C^{max}$ and C^{rand} is obtained by a uniform distribution in interval $[0, C^{max} - C^{fix}]$.

Figure 5 compares the plain CBS with the CBS+CASH algorithm. The optimal PLI (theoretical value) is also drawn as a reference value along with the experimental results. The graph shows that no gain is obtained by the CASH algorithm when the server budget is set equal to the task C^{max}. However, as the server budget decreases, the CASH algorithm becomes more effective, improving the PLI with respect to the plain CBS. It is worth noting that the PLI has a peak for $Q_i = 0.5C_i^{max}$. This strange behaviour is a direct consequence of the CBS deadline postponement rule and can be explained as follows.

Whenever a deadline is postponed by the CBS, the new deadline is increased by a server period. Let us focus, for instance, on task τ_5, when $Q_5 = 0.5C_5^{max} = 5ms$ and $T_5 = 50ms$ (being $U_5 = 0.1$). Since $C_5^{avg} = 0.7C_5^{max} = 7ms$, during overruns the task period is increased up to $100ms$. 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 $Q_5 = 0.4C_5^{max} = 4ms$ 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 the CASH algorithm can achieve a PLI very close to the optimal value when the server budget is a small fraction of the average computation time, although the algorithm yields good results also for server budgets equal to the task average computation times.

5.3. Considerations on runtime overhead

A final set of experiments has also been conducted to estimate the runtime overhead introduced by the CBS+CASH algorithm. A quantitative analysis of the overhead is useful to provide a criterion for setting the

Task	C_i^{max} (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

Table 2. Task set parameters.

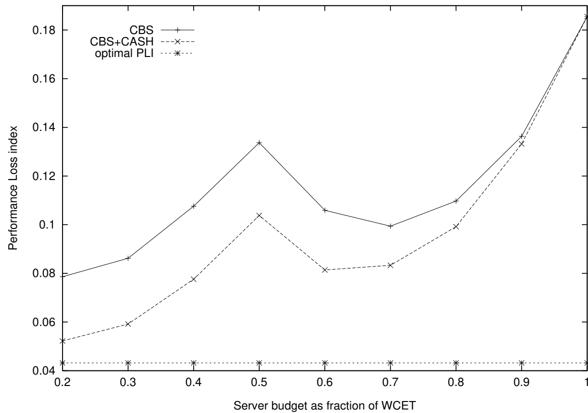

Figure 5. PLI of a task set with $C^{avg} = 0.7C^{max}$.

servers' budgets, since, as shown in Figure 5, 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 2. Since in the HARTIK 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+CASH algorithm has been measured as a difference between the average computation time $\overline{c}'$ of the longest task served with a budget equal to its WCET (so that no deadline is postponed) and 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. Hence, the overhead ω due to a single deadline postponement performed by the CBS+CASH 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_i is the budget assigned to server S_i, the net budget used by the task is $Q_i^{net} = Q_i - \omega$. Hence, the guarantee test can be rewritten as:

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

where Q_i^{net}/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.

Figure 6 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 $Q_s = 0.1C^{max}$; therefore, the server budget can be set equal to $0.2C^{max}$ for the task set of Table 2, obtaining a PLI very close to the optimal one (see Figure 5).

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.

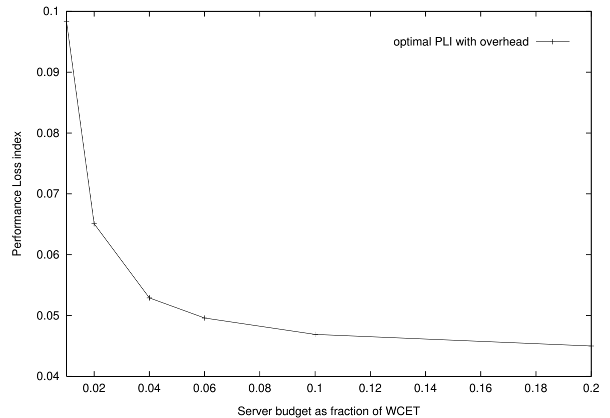

Figure 6. Optimal PLI taking the scheduling overhead into account.

6. Related work

Different approaches have been proposed in the literature to deal with overruns and variable execution times. In [15], the authors provide an upper bound of completion times of jobs chains with variable execution times and arbitrary release times. In [11], a guarantee is computed for tasks whose jobs are characterized by variable computation times and interarrival times, occurring with a cyclical pattern. In [10],

a capacity reservation technique is used to bound the computational demand of tasks with variable computation times, in a fixed priority environment. According to this approach, a fraction of the CPU bandwidth is reserved to each task to achieve temporal isolation. Although such a solution prevents unbounded interference, overruns are not handled efficiently. In fact, whenever a job consumes the reserved budget, its remaining portion is scheduled in background, so prolonging its completion for an unpredictable amount of time. In [16] the authors present a Transform-Task Method (TTM) according to which a task is split into two pieces, where the second piece (i.e., the exceeding computation time causing the overrun) is handled as a job served by a Sporadic Server [14]. Using this approach, a probabilistic guarantee is performed on tasks whose execution times have known distribution. In [5], the authors propose two approaches for handling overruns. The first approach, called the Overrun Server Method (OSM), extends the TTM method to combine a general baseline algorithm for scheduling normal periodic tasks with a generic aperiodic server for handling overruns. Although, this method performs better than handling overruns in background, it cannot ensure that the remaining portion of a task instance is always executed before the next one. The second approach, called the Isolation Server Method (ISM), can achieve isolation among tasks, but it cannot provide a priori guarantee.

A more efficient technique, namely the *Constant Bandwidth Server* (CBS), is proposed in [1] under a dynamic priority environment. As in [10], a fraction of the CPU bandwidth is reserved to each task, but tasks are scheduled by EDF using a suitable deadline, computed as a function of the reserved bandwidth and the actual requests. If a task requires to execute more than expected, its deadline is postponed and its budget replenished. This method allows to achieve isolation among tasks and overruns are handled efficiently based on their actual deadline. As mentioned in the introduction, although isolation mechanisms are essential for increasing system's reliability in the presence of tasks with variable execution times, the correct behavior of the system strongly depends on a correct reservation policy. Recently, this problem has been addressed by a number of authors, who proposed new techniques to reduce such a negative aspect of isolation.

In [7], the authors proposed the Bandwidth Sharing Server (BSS) to handle several multi-thread applications on a single processor, by allowing threads belonging to the same application to reclaim the spare time due to early completions. Although the algorithm provides isolation among applications, no isolation is guaranteed among tasks belonging to the same application. A multi-application environment is also treated in [4], where a two-level scheduling architecture is used to handle each application by a dedicated server. This approach is able to isolate the effect of overloads at the application level, rather than at the task level, but does not provide a global reclaiming mechanism to efficiently exploit the reserved bandwidths.

In [3], the authors proposed a methodology for improving the performance of hard control applications using a resource reservation approach combined with a suitable off-line analysis, based on the Seto et al. algorithm [12]. A less pessimistic analysis and a local reclaiming mechanism is used to increase the average task rates, while a proper overrun control mechanism is adopted to guarantee each task a minimum rate. However, since the reclaiming is local to each task (i.e., no capacity sharing is allowed), the improvement achieved over the Seto et al. algorithm is not so significant. In [8], the authors proposed an elegant technique for scheduling a set of real-time tasks on a single processor, so that each task runs as it is executing on a slower dedicated processor. The method achieves isolation and allows to reclaim most of the spare time unused by tasks. A critical parameter of this approach is the time granularity used in the algorithm; in fact, a small quantum reduces the scheduling error, but increases the overhead due to context switches. In [2], the authors propose a capacity sharing protocol for enhancing soft aperiodic responsiveness in a fixed priority environment, where each soft task is handled by a dedicated server. Although the basic idea of capacity sharing is the same as the one proposed in our paper, the main difference with our CASH algorithm is that in [2] each server can "steal" capacity from the other servers to advance the execution of the served task, thus loosing isolation among the served tasks (a low priority server could receive less bandwidth than requested). In our case, instead, a capacity is given only after a job is completed and a new replenishment is always performed (with a suitable deadline) when a new job arrives. These rules allow the algorithm to preserve the isolation property. Moreover, with respect to the capacity sharing protocol, the CASH algorithm is used to solve a different problem (overrun control) in a different context (dynamic deadline scheduling with resource reservation).

7. Conclusions

In this paper we presented a capacity sharing (CASH) mechanism which allows to achieve temporal protection on tasks' execution, while performing effi-

cient reclaiming of the unused computation times. The algorithm is able to handle tasks with soft, hard, as well as flexible timing constraints.

The work integrates and extends two recent advances in real-time computing: the elastic control approach [3], which allows to optimize the control performance under schedulability constraints, and the Constant Bandwidth Server [1], which provides isolation among application tasks.

The CASH algorithm has been implemented in the HARTIK kernel in order to evaluate its performance and validate our theoretical results. The experiments show the effectiveness of the reclaiming mechanism in enhancing the performance loss index through an increase of tasks' frequencies. The enhancement becomes more significant when the task computation times are characterized by a large variance. Specific tests on the reclaiming mechanism showed that the overhead introduced by the algorithm does not limit its use in real applications.

As a future work, we plan to apply this technique for handling fault-tolerant applications where, each task is composed by a primary and a backup copy.

References

[1] L. Abeni and G. Buttazzo, "Integrating Multimedia Applications in Hard Real-Time Systems", *Proc. of the IEEE Real-Time Systems Symposium*, Madrid, Spain, December 1998.

[2] G. Bernat and A. Burns, "Multiple Servers and Capacity Sharing for Implementing Flexible Scheduling," Technical Report, University of York, March 2000.

[3] M. Caccamo, G. Buttazzo, and L. Sha "Elastic Feedback Control", *IEEE Proceedings of the 12th Euromicro Conference on Real-Time Systems*, Stockholm, Sweden, June 2000.

[4] Z. Deng and J. W. S. Liu, "Scheduling Real-Time Applications in an Open Environment", *Proceedings of the IEEE Real-Time Systems Symposium*, San Francisco, December 1997.

[5] M. K. Gardner and J. W.S. Liu, "Performance of algorithms for scheduling real-time systems with overrun and overload", *IEEE Proceedings of the 11th Euromicro Conference on Real-Time Systems*, York, UK, June 1999.

[6] G. Lamastra, G. Lipari, G. Buttazzo, A. Casile, and F. Conticelli, "HARTIK 3.0: A Portable System for Developing Real-Time Applications," *Proceedings of the IEEE Real-Time Computing Systems and Applications*, Taipei, Taiwan, October 1997.

[7] G. Lipari and G. Buttazzo, "Scheduling Real-Time Multi-Task Applications in an Open System", IEEE Proceedings of the 11th Euromicro Conference on Real-Time Systems, York, England, pp. 234-241, June 1999.

[8] G. Lipari and S. Baruah, "Greedy reclaimation of unused bandwidth in constant-bandwidth servers," *IEEE Proceedings of the 12th Euromicro Conference on Real-Time Systems*, Stockholm, Sweden, June 2000.

[9] C.L. Liu and J.W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard real-Time Environment," *Journal of the ACM* 20(1), 1973, pp. 40–61.

[10] C. W. Mercer, S. Savage, and H. Tokuda, "Processor Capacity Reserves for Multimedia Operating Systems" *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, May 1994.

[11] A. K. Mok and D. Chen, "A multiframe model for real-time tasks," *Proceedings of IEEE Real-Time System Symposium*, Washington, December 1996.

[12] D. Seto, J.P. Lehoczky, L. Sha, and K.G. Shin, "On Task Schedulability in Real-Time Control Systems," *Proceedings of the IEEE Real-Time Systems Symposium*, December 1996.

[13] K.G. Shin, C.M. Krishna, and Y.-H. Lee, "A Unified Method for Evaluating Real-Time Computer Controllers and Its Application," IEEE Transactions on Automatic Control, pp. 357-365, April 1985.

[14] B. Sprunt, L. Sha, and J. P. Lehoczky, "Aperiodic scheduling for hard real-time system". *The Journal of Real-Time Systems*, 1, pp. 27-60, 1989.

[15] J. Sun and J.W.S. Liu, "Bounding Completion Times of Jobs with Arbitrary Release Times and Variable Execution Times", *Proceedings of IEEE Real-Time System Symposium*, December 1996.

[16] T.-S. Tia, Z. Deng, M. Shankar, M. Storch, J. Sun, L.-C. Wu, and J.W.-S. Liu, "Probabilistic Performance Guarantee for Real-Time Tasks with Varying Computation Times," *Proceedings of IEEE Real-Time Technology and Applications Symposium*, Chicago, Illinois, January 1995.