Efficient Aperiodic Service under Earliest Deadline Scheduling

Marco Spuri* Giorgio C. Buttazzo

Scuola Superiore S.Anna
via Carducci, 40 - 56100 Pisa - Italy
spuri@pegasus.sssup.it, giorgio@sssup1.sssup.it

Abstract

In this paper we present four new on-line algorithms for servicing soft aperiodic requests in real-time systems, where a set of hard periodic tasks is scheduled using the Earliest Deadline First (EDF) algorithm. All the proposed solutions can achieve full processor utilization and enhance aperiodic responsiveness, still guaranteeing the execution of the periodic tasks. Operation of the algorithms, performance, schedulability analysis, and implementation complexity are discussed and compared with classical alternative solutions, such as background and polling service. Extensive simulations show that algorithms with contained run-time overhead present nearly optimal responsiveness.

A valuable contribution of this work is to provide the real-time system designer with a wide range of practical solutions which allow to balance efficiency against implementation complexity.

1 Introduction

Many complex control applications include tasks which have to be completed within strict time constraints, called deadlines. If meeting a given deadline is critical for the system operation, and may cause catastrophic consequences, that deadline is considered to be *hard*. If meeting time constraints is desirable, but missing a deadline does not cause any serious damage, then that deadline is considered to be *soft*. In addition to their criticalness, tasks that require regular activations are called *periodic*, whereas tasks which have irregular arrival times are called *aperiodic*.

The problem of scheduling a mixed set of hard periodic tasks and soft aperiodic tasks in a dynamic environment has been widely considered when periodic tasks are executed under the Rate Monotonic (RM) scheduling algorithm [11]. Lehoczky *et al.* [10] investigated server mechanisms (Deferrable Server and Priority Exchange) to enhance aperiodic responsiveness. Sprunt *et al.* [14] described a better service mechanism, called Sporadic Server (SS). Then, Lehoczky and Ramos-Thuel [8] found an optimal service method, called Slack Stealer, which is based on the idea of "stealing" all the possible processing time from the periodic tasks, without causing their deadlines to be missed. The same algorithm has been extended in [13] to handle hard aperiodic tasks, and in [6], to treat a more general class of scheduling problems.

All these methods assume that periodic tasks are scheduled by the RM algorithm. Although RM is an optimal algorithm, it is static and in the general case cannot achieve full processor utilization. In the worst case, the maximum processor utilization that can be achieved is about 69% [11], whereas in the average case, for a random task set, Lehoczky *et al.* [9] showed that it can be about 88%.

For certain applications requiring high processor workload, a 69% or an 88% utilization bound can represent a serious limitation. Processor utilization can be increased by using dynamic scheduling algorithms, such as the Earliest Deadline First (EDF) [11] or the Least Slack algorithm [12]. Both algorithms have been shown to be optimal and achieve full processor utilization, although EDF can run with smaller overhead.

Scheduling aperiodic tasks under the EDF algorithm has been investigated by Chetto and Chetto [4] and Chetto *et al.* [5]. These authors propose acceptance tests for guaranteeing single sporadic tasks, or group of precedence related aperiodic tasks. Although optimal from the processor utilization point of view, these acceptance tests present a quite large overhead to be practical in real-world applications.

Three server mechanisms under EDF have been recently proposed by Ghazalie and Baker in [7]. The authors describe a dynamic version of the known De-

*This work has been supported in part by the CNR of Italy under a research grant.

ferrable and Sporadic Servers [14], called Deadline Deferrable Server and Deadline Sporadic Server, respectively. Then, the latter is extended to obtain a simpler algorithm called Deadline Exchange Server.

The aim of our work is to provide more efficient algorithms for the joint scheduling of random soft aperiodic requests and hard periodic tasks under the EDF policy. Our proposal includes four algorithms having different implementation overheads and different performances. We first present an algorithm, called Dynamic Priority Exchange, which is an extension of previous work under Rate Monotonic (RM). Although much better than background and polling service, it does not offer the same improvement as the others. A completely new "bandwidth preserving algorithm", called Total Bandwidth Server, is also introduced. The algorithm significantly enhances the performance of the previous servers and can be easily implemented with very little overhead, thus showing the best performance/cost ratio. Finally, we present an optimal algorithm, the EDL Server, and a close approximation of it, the Improved Priority Exchange, which has much less run–time overhead. They are both based on off–line computations of the slack time of the periodic tasks. The proposed algorithms provide a useful framework to assist an HRT system designer in selecting the most appropriate method for his or her needs, by balancing efficiency with implementation overhead.

In the definition of our algorithms, we assume that all periodic tasks have hard deadlines coincident with the end of their periods, constant period T_i and constant worst case execution time C_i. All aperiodic tasks do not have deadlines and their arrival time is unknown.

For the sake of clarity, all properties of the proposed algorithms are proved under the above assumptions. However, they can easily be extended to handle periodic tasks whose deadlines differ from the end of the periods and that have non null phasing. In this case, the guarantee tests would only provide sufficient conditions for the feasibility of the schedule. Shared resources can also be included using the same approach found in [7], assuming an access protocol like the Stack Resource Policy [1] or the Dynamic Priority Ceiling [3]. The schedulability analysis would be consequently modified to take into account the blocking factors due to the mutually exclusive access to resources.

Due to lack of space, all proofs and some of the simulations have been omitted. See [15] for a complete description.

2 The Dynamic Priority Exchange Algorithm

In this section we introduce the Dynamic Priority Exchange server, DPE from now on. The main idea of the algorithm is to let the server trade its run–time with the run–time of lower priority periodic tasks (under EDF this means a longer deadline) in case there are no aperiodic requests pending. In this way, the server run–time is only exchanged with periodic tasks, but never wasted (unless there are idle times). It is simply preserved, even if at a lower priority, and it can be later reclaimed when aperiodic requests enter the system.

2.1 Definition of the DPE Server

The DPE server is an extension of the Priority Exchange server [10] adapted to work with the EDF algorithm. In the definition of the server we make use of *aperiodic capacities*, associated to the server itself and to each deadline of periodic task instances. They are updated by the algorithm we are going to describe and, when greater than zero, are considered by the scheduler as schedulable entities. When scheduled, they are used to service pending aperiodic requests.

The server has a specified period T_S and a capacity C_S. At the beginning of each period, the server's aperiodic capacity, C_S^d, where d is the deadline of the current server period, is set to C_S. Each deadline d associated to the instances (completed or not) of the i–th periodic task has an aperiodic capacity, $C_{S_i}^d$, initially set to 0. The aperiodic capacities (those greater than 0) receive priorities according to their deadlines and the EDF algorithm, like all the periodic task instances (ties are broken in favour of capacities, *i.e.*, aperiodics). Whenever the highest priority entity in the system is an aperiodic capacity of C units of time the following happens:

- if there are aperiodic requests in the system, these are served until they complete or the capacity is exhausted (each request consumes a capacity equal to its execution time);
- if there are no aperiodic requests pending, the periodic task having the shortest deadline is executed; a capacity equal to the length of the execution is added to the aperiodic capacity of the task deadline and is subtracted from C (*i.e.*, the deadlines of the highest priority capacity and the periodic task are exchanged);

- if neither aperiodic requests nor periodic task instances are pending, there is an idle time and the capacity C is consumed until, at most, it is exhausted.

In order to implement the algorithm, the only operations required in case of deadline exchange, are to update the values of two capacities and to check whether the "running" one is exhausted. Furthermore, the ready queue can be at most twice as long as without the server (there is at most one aperiodic capacity for each periodic task instance). From these simple observations we can conclude that whereas the implementation of a DPE server is not trivial, the run-time overhead does not significantly increase the typical overhead of a system using an EDF scheduler.

As far as the schedulability is concerned, the DPE server behaves like any other periodic task. The difference is that it can trade its run-time with the run-time of lower priority tasks. When a certain amount of time is traded, one or more lower priority tasks are run at a higher priority level, but their lower priority time is preserved for possible aperiodic requests. This run-time exchange does not affect the schedulability of the task set, as shown in the following Theorem.

Theorem 1 *Given a set of periodic tasks with processor utilization U_P and a DPE server with processor utilization U_S, the whole set is schedulable* if *and* only if

$$U_P + U_S \leq 1,$$

where U_P and U_S are the utilization factors of the periodic task set and the DPE server, respectively. □

2.2 Resource Reclaiming

In most typical real-time systems, the processor load of periodic activities, either statically or dynamically, is guaranteed *a-priori*. This means that the maximum possible load reachable by periodic tasks is taken into account. When this peak is not reached, that is, the actual execution times are lower than the worst case values, it is not always obvious how to reclaim the spare time for real-time activities (a trivial approach is to execute background tasks).

In a system with a DPE server is very simple to reclaim the spare time of periodic tasks for aperiodic requests. It is sufficient that when a periodic task completes, its spare time is added to the corresponding aperiodic capacity. An example of this behaviour is depicted in Figure 1. When the first aperiodic request enters the system at time $t = 4$, one unit of time is available with deadline 8, and three units are available

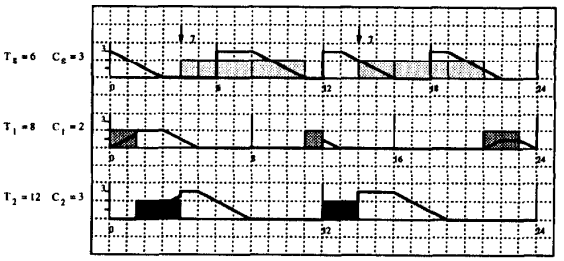

Figure 1: DPE server resource reclaiming.

with deadline 12. The aperiodic request can thus be serviced immediately for all the seven units of time required, as shown in the schedule.

Without the reclaiming described, at time $t = 4$ there would be a half unit of time available with deadline 8 and two and a half units available with deadline 12. The request would be serviced immediately for six units of time, but the last unit would be delayed until time $t = 11$, when it would be serviced in background (neither periodic tasks nor aperiodic capacities would be ready at that time).

Note that reclaiming the spare time of periodic tasks as aperiodic capacities does not affect the schedulability of the system. It is sufficient to observe that, when a periodic task has spare time, this time has been already "allocated" to a priority level corresponding to its deadline when the task set has been guaranteed. That is, the spare time can be safely used if requested with the same deadline. But this is exactly the same as adding it to the task corresponding aperiodic capacity.

3 The Total Bandwidth Algorithm

A different approach that we can follow to improve the aperiodic response times is to assign a possible short deadline to each aperiodic request. The assignment must be done in such a way that the overall processor utilisation of the aperiodic load never exceeds a specified maximum value U_S.

This approach is the main idea behind the Total Bandwidth Server (TBS), which we define in the following section. The name of the server comes from the fact that, each time an aperiodic request enters the system, the total bandwidth of the server, whenever possible, is immediately assigned to it.

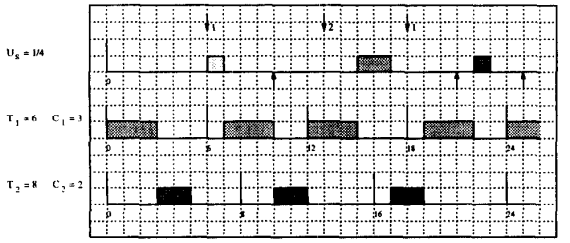

Figure 2: Total Bandwidth server example.

3.1 Definition of the TB Server

The definition of the TB server is very simple. When the k-th aperiodic request arrives at time $t = r_k$, it receives a deadline

$$d_k = \max(r_k, d_{k-1}) + \frac{C_k}{U_S},$$

where C_k is the execution time of the request and U_S is the server utilization factor (*i.e.*, its bandwidth). By definition $d_0 = 0$. The request is then inserted into the ready queue of the system and scheduled by EDF, as any other periodic instance or aperiodic request already present in the system.

Note that we can keep track of the bandwidth already assigned to other requests by simply taking the maximum between r_k and d_{k-1}. Intuitively, as it is stated in Lemma 1, the assignment of the deadlines is such that in each interval of time the ratio allocated by EDF to the aperiodic requests never exceeds the server utilization U_S, that is, the processor utilization of the aperiodic tasks is at most U_S.

In Figure 2, an example of schedule produced by the TB server is depicted. The first aperiodic request, arrived at time $t = 6$, is serviced (*i.e.*, scheduled) with deadline $d_1 = r_1 + \frac{C_1}{U_S} = 6 + \frac{1}{0.25} = 10$. 10 being the earliest deadline in the system, the aperiodic activity is executed immediately. Similarly, the second request receives the deadline $d_2 = r_2 + \frac{C_2}{U_S} = 21$, but it is not serviced immediately, since at time $t = 13$ there is an active periodic task with a shorter deadline (18). Finally, the third aperiodic request, arrived at time $t = 18$, receives the deadline $d_3 = \max(r_3, d_2) + \frac{C_2}{U_S} = 21 + \frac{1}{0.25} = 25$ and is serviced at time $t = 22$.

To show that full processor utilization can be achieved with a TB server, too, we have first proved that the aperiodic processor utilization does not actually exceeds U_S.

Lemma 1 *In each interval of time $[t_1, t_2]$, if C_{ape} is the total execution time demanded by aperiodic requests arrived at t_1 or later and served with deadlines less than or equal to t_2, then*

$$C_{ape} \leq (t_2 - t_1)U_S.$$

□

Now the following Theorem holds.

Theorem 2 *Given a set of n periodic tasks with processor utilization U_P and a TB server with processor utilization U_S, the whole set is schedulable* if *and* only if

$$U_P + U_S \leq 1.$$

□

3.2 Implementation Complexity

The implementation of the TB server is the simplest among those seen so far. In order to correctly assign the deadline to the new issued request, we only need to keep track of the deadline assigned to the last aperiodic request (d_{k-1}). Then, the request can be queued into the ready queue and treated by EDF as any other periodic instance. Hence, the overhead is only due to the increased length of the ready queue if several aperiodic requests are pending at the same time. However, this problem can be solved by managing a separate FIFO queue for the aperiodic requests, and inserting only the first one into the ready queue. In this way the overall overhead is practically negligible.

4 The EDL Algorithm

The Total Bandwidth algorithm is able to achieve good aperiodic response times with extreme simplicity. Still we could desire a better performance if we agree to pay something more. For example, looking at the schedule in Figure 2, we could argue that the second and the third aperiodic requests may be served as soon as they arrive, without compromising the schedulability of the system. The reason for this is that, when the requests arrive, the active periodic instances have enough effective laxity (*i.e.*, the interval between the completion time and the deadline) to be safely preempted. The main idea of the EDL algorithm is to take advantage of these laxities.

4.1 Definition of the EDL Server

The definition of the EDL server makes use of some results presented by Chetto and Chetto in [4]. In this

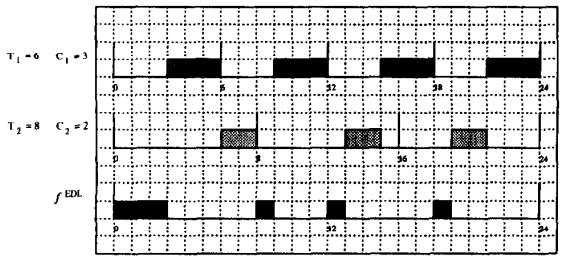

Figure 3: Availability function under EDL.

paper, two different implementations of EDF, namely EDS and EDL, are studied. Under EDS the active tasks are processed as soon as possible, while under EDL they are processed as late as possible. An accurate characterization of the idle times produced by the two algorithms is given. Moreover, a formal proof of the optimality, in the sense that it guarantees the maximum idle time in a given interval, is stated for EDL. In the original paper, this result is used to build an acceptance test for sporadic tasks (*i.e.*, aperiodics with hard deadlines) entering the system, while here it is used to build an optimal server mechanism for soft aperiodic activities.

Let us introduce the terminology used by the authors in [4]. With f_Y^X they denote the availability function

$$f_Y^X(t) = \begin{cases} 1 & \text{if the processor is idle at } t \\ 0 & \text{otherwise,} \end{cases}$$

defined with respect to a task set Y and a scheduling algorithm X. The function $f_{\mathcal{J}}^{\mathrm{EDL}}$, with $\mathcal{J} = \{\tau_1, \tau_2\}$, is depicted in Figure 3. The integral of f_Y^X on an interval of time $[t_1, t_2]$ is denoted by $\Omega_Y^X(t_1, t_2)$: it gives the total idle time in the specified interval.

The result of optimality addressed above is stated in Theorem 2 of [4], which we recall here.

Theorem 3 *Let $\mathcal{A}$ be any aperiodic task set and X any preemptive scheduling algorithm. For any instant t,*

$$\Omega_{\mathcal{A}}^{\mathrm{EDL}}(0,t) \geq \Omega_{\mathcal{A}}^{X}(0,t).$$

□

This result lets us build an optimal server using the idle times of an EDL scheduler. In particular, given the periodic task set, the function f_Y^X, which is periodic with *hiperperiod* $H = \mathrm{lcm}(T_1, \ldots, T_n)$, can be represented by means of two vectors. The first, $\mathcal{E} = (e_0, e_1, \ldots, e_p)$, represents the times at which idle times occur, while the second, $\mathcal{D}^* = (\Delta_0^*, \Delta_1^*, \ldots, \Delta_p^*)$,

i	0	1	2	3
e_i	0	8	12	18
Δ_i^*	3	1	1	1

Figure 4: Idle times under EDL.

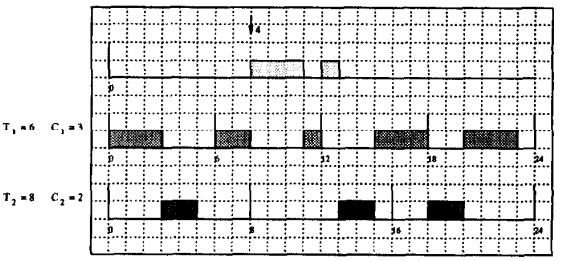

Figure 5: Example of schedule produced with an EDL server.

represents the lengths of these idle times. The two vectors for the example of Figure 3 are shown in Figure 4 (note that we can have idle times only after the arrival time of a periodic task instance).

The EDL server mechanism is based on the following idea: the idle times of an EDL scheduler are used to schedule aperiodic requests as soon as possible, postponing the execution of periodic activities, similarly to the effect of the "Slack Stealer" of [8]. The optimality stated in Theorem 3 will give us the optimality of the server built with this idea.

In particular, when there are no aperiodic activities in the system, the periodic tasks are scheduled according to the EDF algorithm. Whenever a new aperiodic request enters the system (and no previous aperiodic is still active) the set $\mathcal{J}(t)$ of the current active periodic tasks, plus the future periodic instances, is considered. The idle times of an EDL scheduler applied to $\mathcal{J}(t)$, that is, $f_{\mathcal{J}(t)}^{\mathrm{EDL}}$, are then computed and consequently used to schedule the current aperiodic requests. See Figure 5 for an example. Note that the response time of the aperiodic request is optimal.

The procedure to recompute at each new arrival the idle times of EDL applied to $\mathcal{J}(t)$ is described in [4] and is not reported here. The worst case complexity of the algorithm, which is $O(Nn)$, where N is the number of distinct periodic requests that occur in $[0, H[$, and n is the number of periodic tasks, is relatively high and can give the algorithm little practical interest. As for the "Slack Stealer", the EDL server will be used to provide a lower bound to the aperiodic response times, and to build a nearly optimal implementable

algorithm, described in the next section.

4.2 EDL Server Properties

The analysis of the EDL server schedulability is quite straightforward. In fact, the server allocates to the aperiodic activities only the idle times of a particular EDF schedule, without compromising the timeliness of the periodic tasks. This is more precisely stated in the following Theorem.

Theorem 4 *Given a set of n periodic tasks with processor utilization U_P and the corresponding EDL server (the behaviour of the server strictly depends on the characteristics of the periodic task set), the whole set is schedulable* if *and* only if

$$U_P \leq 1$$

(the server automatically allocates the bandwidth $1 - U_P$ to aperiodic requests). □

The property of optimality addressed above, that is, the minimization of the response times of the aperiodic requests, is stated in the following Lemma.

Lemma 2 *Let X be any on-line preemptive algorithm, $\mathcal{J}$ a periodic task set, and J an aperiodic request. If $c^X_{\mathcal{J}\cup\{J\}}(J)$ is the completion time of J when $\mathcal{J}\cup\{J\}$ is scheduled by X, then*

$$c^{\text{EDL server}}_{\mathcal{J}\cup\{J\}}(J) \leq c^X_{\mathcal{J}\cup\{J\}}(J).$$

□

5 The Improved Priority Exchange Algorithm

Although optimal, the algorithm described in the previous section has too much overhead to be considered practical. However, its main idea can be usefully adopted to develop an implementable algorithm, still maintaining a nearly optimal behaviour, as shown later in the discussion of the simulations.

What makes the EDL server not practical is the complexity of computing the idle times at each new aperiodic arrival. This computation must be done each time in order to take into account the periodic instances partially executed or already completed at the time of arrival.

We can avoid the heavy idle time computation using the mechanism of priority exchanges. With this mechanism, in fact, the system can easily keep track

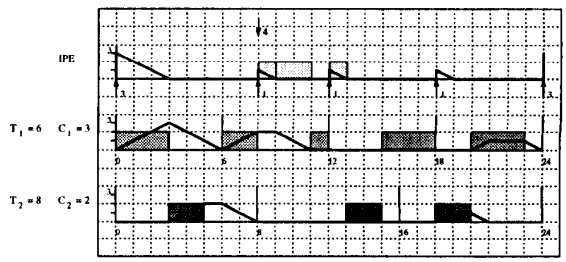

Figure 6: Improved Priority Exchange server example.

of the time advanced to periodic tasks and possibly reclaim it at the right priority level. The idle times of the EDL algorithm can be precomputed off-line. The server can use them to schedule aperiodic requests, when there are any, or to advance the execution of periodic tasks. In the latter case the idle time advanced can be saved as aperiodic capacity at the priority levels of the periodic tasks executed.

5.1 Definition of the IPE Server

To obtain the Improved Priority Exchange (IPE) algorithm, we modify the DPE server using the idle times of an EDL scheduler. First, we obtain a far more efficient replenishment policy for the server. Second, the resulting server is no longer periodic and it can always run at the highest priority in the system.

The IPE server is thus defined in the following way:

- the IPE server has an aperiodic capacity, initially set to 0;
- at each instant $t = e_i + kH$, with $0 \leq i \leq p$ and $k \geq 0$, a replenishment of Δ^*_i units of time is scheduled for the server capacity, that is, at time $t = e_0$ the server will receive Δ^*_0 units of time (the two vectors $\mathcal{E}$ and $\mathcal{D}^*$ have been defined in the previous section);
- the server priority is always the highest in the system, regardless of any other deadline;
- all other rules of IPE (aperiodic requests and periodic instances executions, exchange and consumption of capacities) are the same as for a DPE server.

The same task set of Figure 5 is scheduled with an IPE server in Figure 6. Note that the server replenishments are set according to the function $f^{\text{EDL}}_{\mathcal{J}}$, illustrated in Figure 3.

The IPE schedulability is stated in the following Theorem.

Theorem 5 *Given a set of n periodic tasks with processor utilization U_P and the corresponding IPE server (the parameters of the server depend on the periodic task set), the whole set is schedulable* if *and* only if

$$U_P \leq 1$$

(the server automatically allocates the bandwidth $1 - U_P$ to aperiodic requests). □

The reclaiming of unused periodic execution time can be done in the same way as for the DPE server. When a periodic task completes, its spare time is added to the corresponding aperiodic capacity. Again, this behaviour does not affect the schedulability of the system. The reason is of course the same as for the DPE server.

5.2 Implementation Complexity

As for the resource reclaiming, even the implementation complexity of IPE is similar to that of any other DPE server, at least from the time point of you. The two vectors $\mathcal{E}$ and $\mathcal{D}^*$ are in fact precomputed before the system is run. The replenishments of the server capacity are no longer periodic, but this does not change the complexity. Finally, all the rest is perfectly the same, hence even the consideration on the implementation complexity are comparable.

What can change dramatically is the memory requirement. If the periods of periodic tasks are not harmonically related, we could have a huge *hiperperiod* $H = \text{lcm}(T_1, \ldots, T_n)$, which would mean a great memory occupancy to store the two vectors $\mathcal{E}$ and $\mathcal{D}^*$, since the memory occupancy is $O(N)$, where N is the number of distinct periodic requests that occur in $[0, H[$.

6 Performance Results

DPE, TBS, EDL and IPE algorithms have been simulated to compare the average response times of soft aperiodic tasks with respect to the response times obtained with background scheduling. This form of aperiodic scheduling is the simplest possible: the aperiodic tasks are executed only when the processor would be otherwise idle, that is, no periodic task instances are ready to run.

For completeness, also a Polling server and a dynamic version of the Sporadic Server (DSS) [14, 15] have been compared with the proposed algorithms.

In all simulations, a set of ten periodic tasks with periods ranging from 100 and 1000 was chosen. Three periodic loads were simulated, by setting the processor utilization factor U_p at 40%, 65% and 90%, referred in the following as low, medium and high periodic load, respectively.

The aperiodic load for these simulations was varied across the range of processor utilization unused by the periodic tasks. The interarrival times (with average T_a) for the aperiodic tasks were modeled using a Poisson arrival pattern, whereas the aperiodic service times (with average T_s) were modeled using an exponential distribution.

Where applicable, the processor utilization of the servers was set to all the utilization left by the periodic tasks, that is, $U_S = 1 - U_P$. The period of the periodic servers, namely Polling, DPE and DSS, was set equal to the average aperiodic interarrival time (T_a) and, consequently, the capacity was set to $C_S = T_a U_S$.

Unless otherwise stated, the data plotted for each algorithm represent the ratio of the average aperiodic response time relative to the response time of background aperiodic service. The average is computed over ten simulations, in which a total of one hundred thousand aperiodic requests were generated. In this way, an average response time equivalent to background service has a value of 1.0 on all the graphs. Hence, a value less than 1.0 corresponds to an improvement in the average aperiodic response time over background service. The lower the response time curve lies on these graphs, the better the algorithm is for improving aperiodic responsiveness.

6.1 Experiment 1: IPE vs. EDL

In the first experiment, we have compared the performance of our IPE algorithm versus the optimal EDL server mechanism. The graph shown in Figure 7 corresponds to a high periodic load. The aperiodic load was generated using a mean interarrival time $T_a = 100$ and varying the average aperiodic service time T_s so that the total load covered, roughly, the range from U_p to the full processor utilization.

As can be clearly seen from the graph, the maximum difference between the performance of the two algorithms is less than 0.2%, *i.e.*, it is so small that can be reasonably considered negligible for any practical application.

Although IPE and EDL have very similar performances, they differ significantly in their implementation complexity. As mentioned in previous sections, the EDL algorithm needs to recompute the server parameters quite frequently (namely, when an aperiodic request enters the system and all previous aperiodics have been completely serviced). This overhead can be too expensive in terms of cpu time to use the algo-

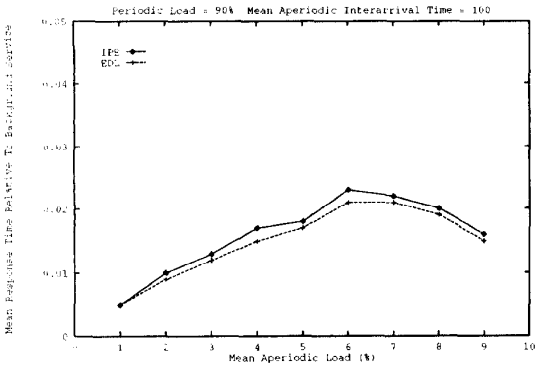

Figure 7: Comparison between IPE and EDL server.

rithm in practical applications. On the other hand, for the IPE algorithm we only have to compute off–line the parameters of the server. Then, at run–time, assuming we have enough memory, the implementation complexity is the same as for a DPE server, which is quite reasonable.

In summary, IPE has nearly the same performance of EDL, but with much less overhead. For this reason, the EDL server performance is not reported in all subsequent simulations. Moreover, the performance of the IPE server will be the reference in the following experiments.

6.2 Experiment 2: Response Time vs. Aperiodic Load

In the second experiment, we tested the performance of all algorithms as a function of the aperiodic load. The load was varied by changing the average aperiodic service time, while the average interarrival time was set at the value of $T_a = 100$.

Figure 8 presents the results of these simulations. In this figure, three graphs are presented, which correspond to the different periodic loads simulated, low, medium and high respectively. In each graph, the average aperiodic response time of each algorithm is plotted with respect to that of background service as a function of the mean aperiodic load $U_{ape} = \frac{T_s}{T_a}$.

As can be seen from each graph, the TBS and IPE algorithms can provide a significant reduction in average aperiodic response time compared to background or polling aperiodic service, whereas the performance of the DPE and DSS algorithms depends on the aperiodic load. For low aperiodic load, DPE and DSS perform as well as TBS and IPE, but as the aperiodic load increases their performance tends to be similar to

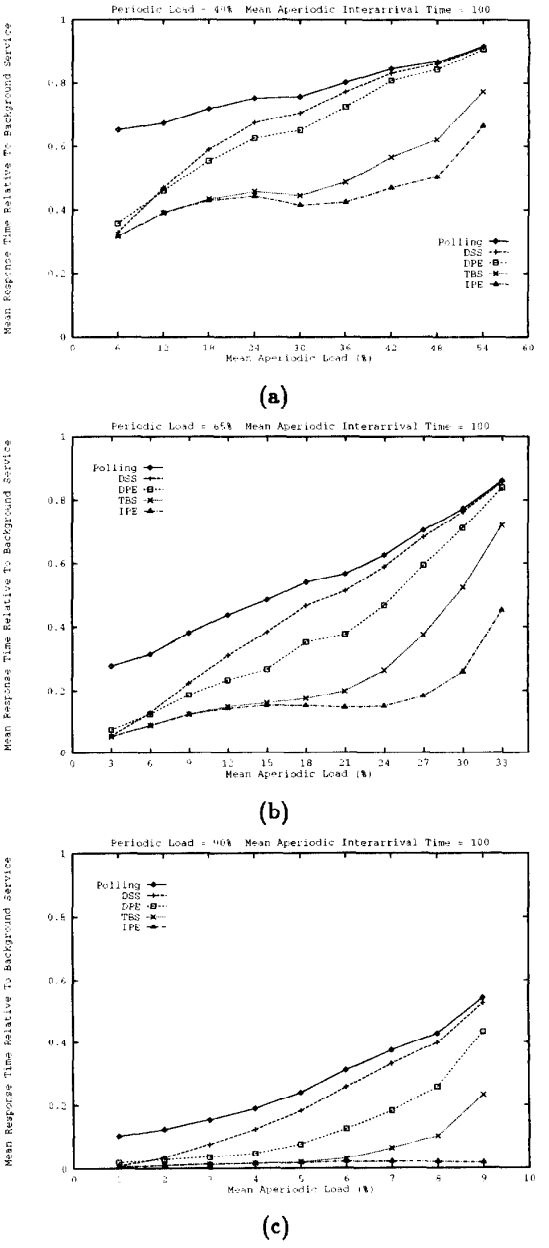

Figure 8: Algorithms performance with different processor loads.

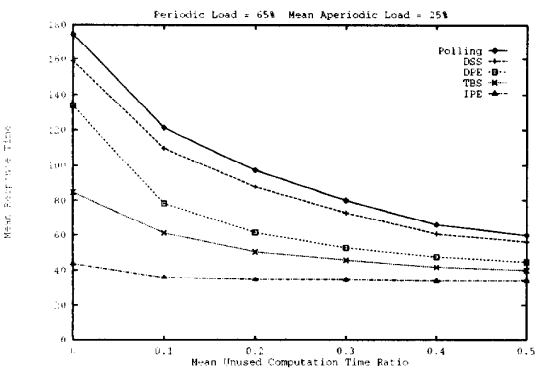

Figure 9: Response times vs. unused computation times.

that one shown by the Polling server.

Note that, in all graphs, TBS and IPE have about the same responsiveness when the aperiodic load is low, and they exhibit a slightly different behaviour for heavy aperiodic loads.

6.3 Experiment 3: Response Time vs. Unused Periodic Task Computation Time

The goal of this experiment was to verify the effectiveness of the resource reclaiming technique, described in Section 2.2, which can be used in the algorithms DPE and IPE. In order to do this, we have compared the performance of the five algorithms (Polling, DPE, DSS, TBS and IPE) on a number of task sets, in which the actual execution times of periodic tasks were less than the worst case ones. The estimated periodic load, computed using the worst case execution times, was set to 65%. The mean interarrival time of the aperiodic requests was set to 100 units, while the mean aperiodic service time was set to 25 units, thus giving a total estimated processor load of 90%. The actual execution time $aet_{i,j}$ of the j^{th} instance of the i^{th} periodic task was generated using the following formula:

$$aet_{i,j} = C_i \cdot \mathrm{rnd}(1 - 2\Delta, 1),$$

where C_i is the worst case execution time of the task, $\mathrm{rnd}(a, b)$ is a function that returns a random number in the interval $[a, b]$, using a uniform distribution, and the parameter Δ, which is $\frac{C_i - E[aet_{i,j}]}{C_i}$, represents the average ratio of the unused computation times.

The result of the simulation can be seen in the graph shown in Figure 9. In the vertical axis the average response time of each algorithm is represented as a function of the parameter Δ, which ranges from 0 to 0.5. The case $\Delta = 0$ corresponds to the situation in which the actual execution times are equal to the worst case ones. In this particular situation the result is equivalent to that shown in a previous experiment.

As soon as Δ becomes greater than zero, that is, the actual execution times become less than the worst case ones, the performance of the DPE server tends to be much better, and also tends to approach the performance of the TB server. This behaviour is confirmed for all other values of Δ, thus proving the effectiveness of the reclaiming technique used in the DPE and IPE algorithms.

From the graph, we can see that the TBS algorithm shows a good behaviour, too, although no explicit reclaiming has been designed for it. Finally, also the Polling and the Sporadic servers show good improvements, due to the lower actual periodic load. However, their performance is always significantly worse, compared to the others.

7 Discussion and Conclusions

In this paper we have introduced five novel on–line scheduling algorithms for real–time systems with dynamic priorities. Namely, all algorithms exploit the well known Earliest Deadline First policy to deal with both soft aperiodic and hard periodic tasks. All algorithms have been characterized in terms of schedulability and implementation complexity. For two of them, DPE and IPE, a simple resource reclaiming technique has been designed and proved to be effective. Finally, extensive comparisons have been carried out in different experiments.

The experimental simulations have established that, from a performance point of view, IPE and EDL show the best results. Although optimal, EDL is far from being reasonably practical, due to the overall complexity. On the other hand, IPE is able to achieve a comparable performance with much less computational overhead. Both algorithms may have significant memory demands when the periods of the periodic tasks are not harmonically related.

The Total Bandwidth algorithm has shown a very good performance, sometimes comparable to that of the nearly optimal of IPE. Observing that its implementation complexity is among the simplest, one could consider this to be a good candidate for practical systems.

Even though a bit more complex, the DPE and the DSS algorithms show slightly worse performance, al-

though they both provide better responsiveness than the Polling server and the naive background service.

With this work we have covered a wide spectrum of algorithms dealing with aperiodic service. Considering also other works in the literature, the real–time designer that wishes to build a system with dynamic priorities should now have a sufficient number of choices for designing an efficient aperiodic service mechanism. In particular, in all those applications in which the periodic load is fixed, the aperiodic service algorithm can be chosen to balance efficiency against complexity.

As future work, we are considering to use the algorithms presented in this paper as a basis for handling hard aperiodic tasks. The main goal will be to build a uniform solution in which hard aperiodic tasks can be dynamically guaranteed [2], while average response times of soft aperiodic tasks can be predicted with reasonable accuracy.

References

[1] T.P. Baker, "Stack–Based Scheduling of Real–Time Processes," *The Journal of Real–Time Systems* 3(1), 67–100, 1991.

[2] G. Buttazzo and J. Stankovic, "RED: A Robust Earliest Deadline Scheduling Algorithm," *Proc. of 3rd International Workshop on Responsive Computing Systems*, Austin, 1993.

[3] M. Chen and K. Lin, "Dynamic Priority Ceilings: A Concurrency Control Protocol for Real-Time Systems," *The Journal of Real-Time Systems*, 2, 1990.

[4] H. Chetto and M. Chetto, "Some Results of the Earliest Deadline Scheduling Algorithm," *IEEE Trans. on Software Engineering*, 15(10), 1261–1269, 1989.

[5] H. Chetto, M. Silly, T. Bouchentouf, "Dynamic Scheduling of Real–Time Tasks under Precedence Constraints," *The Journal of Real–Time Systems* 2, 181–194, 1990.

[6] R.I. Davis, K.W. Tindell, A. Burns, "Scheduling Slack Time in Fixed Priority Pre–emptive Systems," *Proc. of Real–Time Systems Symposium*, 222–231, 1993.

[7] T.M. Ghazalie and T.P. Baker, "Aperiodic Servers In A Deadline Scheduling Environment," *The Journal of Real–Time Systems*, to appear.

[8] J.P. Lehoczky and S. Ramos–Thuel, "An Optimal Algorithm for Scheduling Soft–Aperiodic Tasks in Fixed–Priority Preemptive Systems," *Proc. of Real–Time Systems Symposium*, 110–123, 1992.

[9] J.P. Lehoczky, L. Sha, Y. Ding, "The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behaviour," *Proc. of Real–Time Systems Symposium*, 166–171, 1989.

[10] J.P. Lehoczky, L. Sha, J.K. Strosnider, "Enhanced Aperiodic Responsiveness in Hard Real–Time Environments," *Proc. of Real–Time Systems Symposium*, 261–270, 1987.

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

[12] A.K. Mok, "Fundamental Design Problems of Distributed Systems for the Hard–Real–Time Environment," *Ph.D. Dissertation*, MIT, 1983.

[13] S. Ramos–Thuel and J.P. Lehoczky, "On–line Scheduling of Hard Deadline Aperiodic Tasks in Fixed–Priority Systems," *Proc. of Real–Time Systems Symposium*, 160–171, 1993.

[14] B. Sprunt, L. Sha, J. Lehoczky, "Aperiodic Task Scheduling for Hard–Real–Time Systems," *The Journal of Real–Time Systems* 1, 27–60, 1989.

[15] M. Spuri and G. Buttazzo, "Scheduling Aperiodic Tasks in Dynamic Priority Systems," *Technical Report* ARTS Lab TR 94–06, Scuola Superiore S. Anna, Pisa, Italy, April 1994.