Scheduling Aperiodic Tasks in Dynamic Priority Systems

Marco Spuri* Giorgio Buttazzo

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

Abstract

In this paper we present five 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

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

addition to their criticalness, tasks that require regular activations are called *periodic*, whereas tasks which have irregular arrival times are called *aperiodic*.

For example, in a robot control application, activities such as sensory acquisition, data processing, path planning, and low level control loops require periodic tasks that have to be executed at constant rates to insure robot stability. For this reason, periodic tasks often have hard deadlines. Aperiodic tasks are typically used to serve random processing requirements, such as operator requests or displaying activities, hence they usually have soft deadlines, or no deadlines at all. Aperiodic tasks with hard deadlines are called *sporadic* tasks.

Given a set of real–time tasks, a schedule is said to be feasible if all hard tasks complete within their deadlines. A critical task with a hard deadline is said to be guaranteed at its activation time if the system is able to find a feasible schedule for the newly arrived task and all previously guaranteed tasks. An operating system capable of guaranteeing and executing tasks with hard time constraints is called *Hard Real–Time* (HRT) system. In a critical application, the goal of an HRT system is not only to meet the deadlines of all hard tasks, but also to minimize the average response time for soft activities.

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. The basic idea was to use a special periodic task to efficiently serve possible aperiodic requests of execution. 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. Although it is not practical, because of its high overhead, the algorithm provides a lower bound on aperiodic response times, as well as the basis for nearly optimal implementable algorithms. In [13], the same authors extended the algorithm to deal also with hard aperiodic tasks, as well as a more complex task model based on serially executed subtasks. In [6], Davis *et al.* present a similar algorithm, in which the slack is computed at run–time, thus making the

algorithm applicable to 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 Deferrable 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 five algorithms having different implementation overheads and different performances. We first present two algorithms, namely the Dynamic Priority Exchange and the Dynamic Sporadic Server, which are extensions of previous work under Rate Monotonic (RM). Although much better than background and polling service, they do 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 per-

formance/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.

The rest of the paper is organized as follows. In section 2 we state our assumptions. Section 3 describes the Dynamic Priority Exchange (DPE) algorithm, which is an extension of the Priority Exchange algorithm proposed by Lehozcky *et al.* [10]. In section 4, an extension of the Sporadic Server [14] working under EDF is presented. A new simple and efficient algorithm, called Total Bandwidth Server, is introduced in section 5. In section 6, we describe an optimal algorithm and its main properties are shown. In section 7, a nearly optimal algorithm is derived from DPE, using the insights gained in section 6. Simulation results are discussed in section 8. Finally, considerations and conclusions are included in section 9.

2 Assumptions and Terminology

In the definition of our algorithms we will consider the following assumptions:

- all periodic tasks $\tau_i : i = 1, ..., n$ have hard deadlines;
- all aperiodic tasks $J_i : i = 1, ..., m$ do not have deadlines;
- each periodic task τ_i has a constant period T_i and a constant worst case execution time C_i, which is considered to be known, as it can be derived by a static analysis of the source code;
- all periodic tasks are simultaneously activated at time $t = 0$; i.e., the first instance of each periodic task has a request time $r_i(0) = 0$;
- the request time of the kth periodic instance is given by $r_i(k) = r_i(k-1) + T_i$;
- the deadline of the kth periodic instance is given by $d_i(k) = r_i(k) + T_i$;
- the arrival time of each aperiodic task is unknown;

- the worst case execution time of each aperiodic task is considered to be known at its arrival time.

For the sake of clarity, all properties of the proposed algorithms will be proven 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.

As a future work, we plan to treat also sporadic tasks and aperiodic tasks with firm deadlines, that is, tasks that can be rejected if not guaranteed to meet their deadlines [2].

3 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.

3.1 Definition of the DPE Server

In [10] Lehoczky *et al.* introduce the Priority Exchange (PE) algorithm, a server for aperiodic requests under the RM algorithm. The DPE server is an extension of the PE server adapted to work with the EDF algorithm.

The algorithm is defined in the following way:

- the DPE server has a specified period T_S and a capacity C_S;

- at the beginning of each period, the server's *aperiodic* capacity is set to C_S^d, where d is the deadline of the current server period;
- 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 server or one of the others) 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.

An example of schedule produced by the DPE algorithm is illustrated in Figure 1. Two periodic tasks, τ_1 and τ_2, with periods 8 and 12 and worst case execution times 2 and 3 respectively, and a DPE server, with period 6 and capacity 3, are present in the system. At time $t = 0$, the aperiodic capacities C_S^6 (with deadline 6), $C_{S_1}^8$ (with deadline 8) and $C_{S_2}^{12}$ (with deadline 12) are set to 3, 0 and 0. Since no aperiodic requests are pending the two first periodic instances of τ_1 and τ_2 are allowed to execute. Consequently, the 3 units of capacity C_S^6 are consumed in the first three units of time. In the same interval two units of time are accumulated in $C_{S_1}^8$ (during the execution of τ_1) and one unit in $C_{S_2}^{12}$ (at the beginning of the τ_2's first execution). At time $t = 3$, $C_{S_1}^8$ is the highest priority entity in the system. Again τ_2 is allowed to

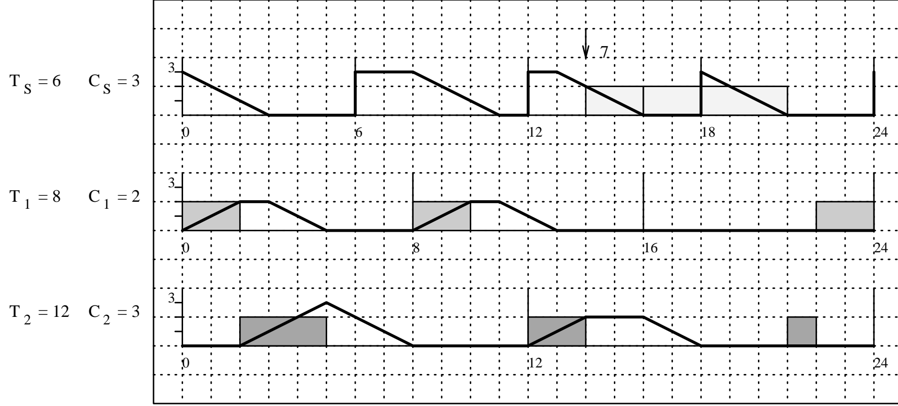


Figure 1: Dynamic Priority Exchange server example.

keep executing. The two units of $C_{S_1}^8$ are consumed and accumulated in $C_{S_2}^{12}$. In the following three units of time the processor is idle and $C_{S_2}^{12}$ is consequently consumed. Note that C_S^{12}, set at value 3 at time $t = 6$, is preserved until time $t = 8$, when it becomes the highest priority entity in the system (ties among aperiodic capacities are assumed to be broken in a FIFO order).

At time $t = 14$, an aperiodic request of 7 units of time enters the system. Since C_S^{18} is equal to 2, the first two units of time are served with deadline 18. The subsequent two units are served with the capacity $C_{S_2}^{24}$, *i.e.*, with deadline 24. Finally, the last three units are also served with deadline 24, because at time $t = 18$ the server capacity C_S^{24} is set to 3.

3.2 Dynamic Priority Exchange Schedulability

Let us now analyze the schedulability condition of a set of periodic tasks which are scheduled, together with a DPE server, with the algorithm illustrated above.

Intuitively, the 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.

As usual, let us define the periodic tasks utilization factor as

$$U_P = \sum_{i=1}^{n} \frac{C_i}{T_i}$$

and the server utilization factor as

$$U_S = \frac{C_S}{T_S}.$$

Our objective is to prove that the classical Liu and Layland result [11] for an EDF scheduler can be extended including the server utilization factor. In order to do this, given a schedule S produced using the DPE algorithm, let us consider a schedule S' built in the following way:

- the server is replaced with a periodic task of equal characteristics (*i.e.*, period T_S and worst case execution time C_S); in the new schedule, the task executes when the server capacity in S decreases;
- each execution of periodic instances during deadline exchanges (*i.e.*, increase in the corresponding aperiodic capacity) is postponed until the capacity decreases;
- all other executions of periodic instances are left as in S.

Note that, because of the definition of DPE, at any time there is at most only one aperiodic capacity decreasing in S, so S' is well defined. Also observe that, in each feasible schedule produced by the DPE algorithm, all the aperiodic capacities are exhausted before their respective deadlines (if one of these capacities would go beyond its deadline, introducing enough aperiodic requests we could build a schedule in which the execution of subsequent periodic instances would be delayed and some of them would miss their deadlines).

In Figure 2 the schedule S' built applying the definition to the schedule S in Figure 1 is shown. Note that all the periodic executions corresponding to increasing aperiodic capacities, have been moved to the corresponding intervals in which the same capacities decrease (of course the length of the corresponding intervals is the same). Also note that, even with a different schedule S, the schedule S' would not change. The reason for this is that the "actual" task execution scheduled by EDF would be always at the time when the capacity decreases, and not when increases.

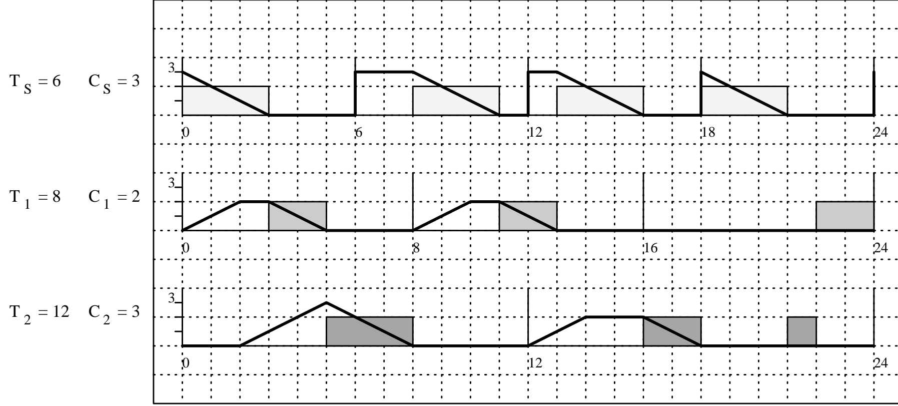


Figure 2: DPE server schedulability.

That is, the schedule S' is invariant, and it only depends on the characteristics of the server and on the periodic task set. This observation will let us prove the claimed result.

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

$$U_P + U_S \leq 1.$$

Proof. All the schedules produced by the DPE algorithm have a unique corresponding EDF schedule S', built following the definition above. Moreover, the task set in S' is periodic and has processor utilization $U = U_P + U_S$, that is, S' is feasible if and only if $U_P + U_S \leq 1$. Now, if $U_P + U_S \leq 1$, observing that in each schedule S the completion time of a periodic instance is less than or equal to the completion time of the corresponding instance in the schedule S', being S' feasible we can conclude that also S is feasible, that is, the set is schedulable by the DPE algorithm.

Viceversa, if the set is schedulable, observing that S' is a particular schedule produced by the DPE algorithm when there are enough aperiodic requests, we can conclude that $U_P + U_S \leq 1$. □

3.3 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

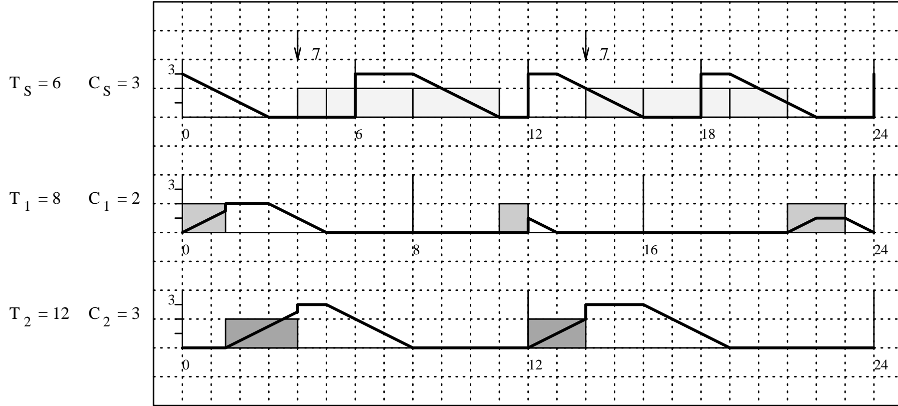


Figure 3: DPE server resource reclaiming.

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 3. 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 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.

3.4 Implementation Complexity

The Priority Exchange server has been said to be quite difficult to implement and to have a quite large run–time overhead [14]. Let us now evaluate the complexity of the implementation of a DPE server in a uniprocessor system using EDF as scheduling algorithm.

If we assume that the scheduler and the dispatcher are able to manipulate queues with two sorts of entities, tasks and aperiodic capacities, the server involves only some more operations in the book–keeping of these capacities. In particular, at each system tick it may be necessary, in case of deadline exchange, to update the values of two capacities and to check whether the "running" one is exhausted. The increase in complexity, with respect to EDF, is of course very low.

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). That is, the complexity of the routines that manipulate this queue can be at most doubled, if we use a linear list (using a binary heap the increase in complexity is practically negligible).

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.

4 The Dynamic Sporadic Server[1] Algorithm

In [14], another efficient algorithm, called Sporadic Server (SS), for servicing aperiodic requests in uniprocessor systems using a Rate Monotonic scheduler is introduced. Since this algorithm exhibits nice properties, namely efficiency and simplicity, we have studied how a similar policy could be extended to work under a dynamic EDF scheduler.

[1] A similar algorithm called Deadline Sporadic Server has been independently developed by Ghazalie and Baker, and has been recently described in [7].

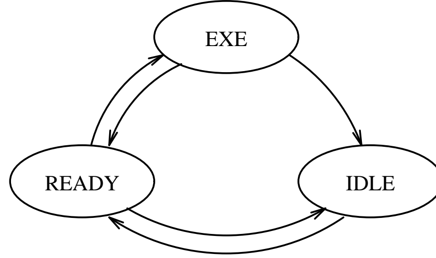


Figure 4: DSS state transition diagram.

4.1 Definition of the Dynamic Sporadic Server

Similarly to other servers, the DSS algorithm also has a specified capacity C_S and a period T_S. The main idea is always to preserve the server execution time (*i.e.*, its capacity) for possible aperiodic requests. The difference from previous server algorithms is that the capacity is not replenished at its full value at the beginning of each period of the server, but only when it has been consumed. The times at which the replenishments occur are chosen very carefully, according to a replenishment rule, which allows the system to achieve full processor utilization.

The main difference between the SS algorithm described in [14] and our dynamic version is that, whereas the SS has a fixed priority chosen according to the RM algorithm (that is, according to its period T_S), our version has a dynamic priority. This dynamic priority is assigned choosing a suitable deadline, whose value is set to the next defined replenishment time.

In order to describe the Dynamic SS algorithm (DSS), let us consider the three states in which the server, as any other task in the system, can be: IDLE, READY and EXE. Figure 4 illustrates the three states and all possible transitions among them. The DSS algorithm is described in terms of these transitions:

IDLE $\rightarrow$ READY : either an aperiodic request has entered the system and $C_S > 0$, or C_S has become greater than zero (a replenishment has occurred); in either case, the next replenishment time and the current deadline of the server are both set to $t + T_S$, where t is the current time[2];

[2] One way to improve the performance of the server would be, as in the definition of the Deadline Sporadic Server of [7], to allow this transition also when a periodic task instance with deadline $d \leq t + T_S$ starts to execute. This should let the server have a greater priority to serve forthcoming aperiodic requests.

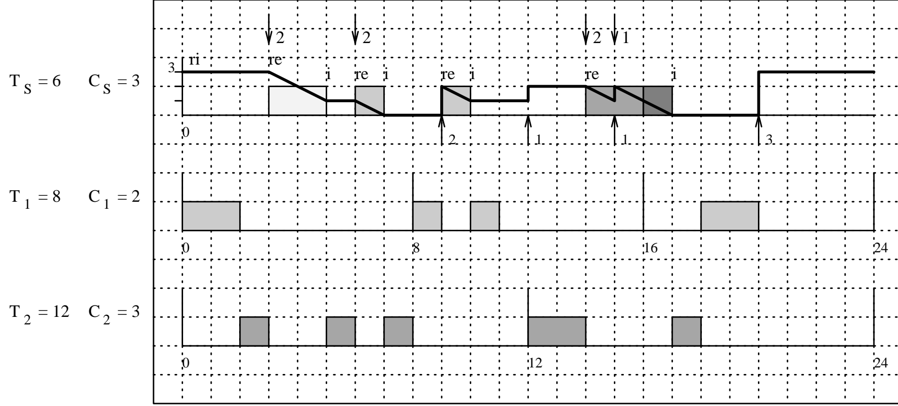


Figure 5: Dynamic Sporadic Server example.

READY → IDLE : the server has the highest priority in the system (*i.e.*, the shortest deadline), but there are no aperiodic requests to service[3];

READY → EXE : the server has the highest priority in the system and there are aperiodic requests to service; while requests are served, a capacity equal to their execution times is consumed accordingly;

EXE → READY : the server is preempted by a higher priority task;

EXE → IDLE : either an aperiodic request has been served and there are no other requests pending, or the server capacity has been exhausted; a replenishment of the consumed server execution time is scheduled to occur at the replenishment time set during the last transition IDLE → READY.

At the beginning, when the system is started, the server is in READY state, has full capacity C_S, and deadline and next replenishment time set to T_S. In Figure 5, an example of schedule produced by the DSS algorithm is shown.

At time $t = 0$, the server has the highest priority in the system (its deadline is 6). Since there are no aperiodic requests it immediately enters the IDLE state. At time $t = 3$, an aperiodic request with execution time 2 arrives. The server enters again the READY state setting the next replenishment time and its deadline to $t = 3 + T_S = 9$.

[3] Similarly to the previous situation, we could try to improve the performance of the server also checking whether the first task in the ready queue has a deadline d such that $d_S < d \leq t + T_S$. In this case we could assign the server a deadline $d_S = d$, leave the server in READY state and execute the task with deadline d.

That is, it becomes the highest priority task in the system and the request is serviced at once. At time $t = 5$, the request is completed, the server goes in IDLE state and a replenishment to C_S of two units of time is scheduled to occur at time $t = 9$.

At time $t = 6$, a second aperiodic requests arrives. Being $C_S > 0$ the server goes in READY state and the next replenishment time and its deadline are set to $t = 6 + T_S = 12$. Again the server becomes the highest priority task in the system (we assume that ties among tasks are always resolved in favour of the server) and the request is serviced immediately. This time, however, the server has only a capacity of one unit of time. Consequently, at time $t = 7$ the capacity is exhausted, the server goes in IDLE state, a replenishment of one unit of time is scheduled for $t = 12$, and the aperiodic request is delayed until C_S becomes again greater than zero. This is true at time $t = 9$, when a replenishment of two units of time occurs. The server goes consequently in READY state, setting the next replenishment time and its deadline to $t = 9 + T_S = 15$. Being the highest priority task, the pending aperiodic request is serviced until completion, which happens one unit of time later. A replenishment of one unit is then scheduled to occur at time $t = 15$.

Note that, if a second aperiodic request arrives while another one is being serviced, or, equivalently, when the server is in READY state, provided that enough capacity is available, even the second request is serviced with the same priority (deadline) of the first one. In Figure 5 this happens at time $t = 15$. The second aperiodic request is serviced with the same deadline (20) as the request arrived at time $t = 14$.

4.2 Dynamic Sporadic Server Schedulability

In order to prove the schedulability bound for the Dynamic Sporadic Server, we will first show that the server behaves, as intuitive, like a periodic task with period T_S and execution time C_S.

Lemma 1 *In each interval of time $[t_1, t_2]$, such that the Dynamic Sporadic Server is in IDLE state at t_1, if C_{ape} is the total execution time demanded for aperiodic requests in the same interval (that is, C_{ape} is the server time between t_1 and t_2 demanded with a deadline less than or equal to t_2), then*

$$C_{ape} \leq \left\lfloor \frac{t_2 - t_1}{T_S} \right\rfloor C_S.$$

Proof. First note that since replenishments are always equal to the time consumed, the server capacity is at any time less than or equal to C_S. Also, the replenishment policy establishes that the consumed capacity cannot be reclaimed before than T_S units of time after the instant at which the server has become ready. This means that after t_1, at most C_S time can be demanded in each subsequent interval of time of length T_S. The thesis follows. □

We are now able to show that not only the DSS behaves like a periodic task, but also that a full processor utilization is still achieved.

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

$$U_P + U_S \leq 1.$$

Proof. "If". Suppose there is an overflow at time t. The overflow is preceded by a period of continuous utilization of the processor. Furthermore, from a certain point t' on, only instances of tasks ready at t' or later and having deadlines less than or equal to t are run (the server may be one of these tasks). Let C be the total execution time demanded by these instances. Since there is an overflow at time t, we must have

$$t - t' < C.$$

We also know that

$$\begin{aligned} C &\leq \sum_{i=1}^{n} \left\lfloor \frac{t-t'}{T_i} \right\rfloor C_i + C_{ape} \\ &\leq \sum_{i=1}^{n} \left\lfloor \frac{t-t'}{T_i} \right\rfloor C_i + \left\lfloor \frac{t-t'}{T_S} \right\rfloor C_S \\ &\leq \sum_{i=1}^{n} \frac{t-t'}{T_i} C_i + \frac{t-t'}{T_S} C_S \\ &\leq (t-t')(U_P + U_S). \end{aligned}$$

It follows that

$$U_P + U_S > 1,$$

a contradiction.

"Only If". If there are enough aperiodic requests, the demanded server execution time is C_S for each subsequent T_S units of time. That is, the server behaves exactly as a periodic task with period T_S and execution time C_S. Being the processor utilization $U = U_P + U_S$, from Theorem 7 of [11] we can conclude that $U_P + U_S \leq 1$. $\square$

4.3 Implementation Complexity

In order to implement the DSS algorithm, the dispatcher and the scheduler, besides the usual task entries, must be able to manage the server entry in the system queues. This only means to manage a further queue (of aperiodic tasks) associated with the server.

Similarly to the DPE server, the system book–keeping is involved in the updates of the server capacity. This happens in two different situations. First, during aperiodic services the capacity must be decreased at each system tick in order to check whether it has been exhausted. Second, when a scheduled replenishment time is encountered, the capacity must be increased with the specified value.

Again, we can expect that the implementation of the Dynamic Sporadic Server is quite straightforward and that the run–time overhead is very low.

5 The Total Bandwidth Algorithm

Looking at the characteristics and the properties of the Sporadic Server, it can be easily seen that, when the server has a long period, the execution of the aperiodic requests can be delayed significantly. An example of this situation is illustrated in Figure 6. Here, the aperiodic request arrived at time $t = 6$ is served with a deadline equal to $6 + T_S = 18$, which causes the execution to be delayed until time $t = 11$. The second request at time $t = 13$ is treated in a similar way, while the third, at time $t = 18$, arrives when the server is in READY state and has enough capacity, so that it can be served with the same deadline as the second request.

Observing the schedule, we can conclude that the delays are mainly due to the fact that the server time, because of its long period, is always scheduled with a long deadline. And this is regardless of the aperiodic execution times.

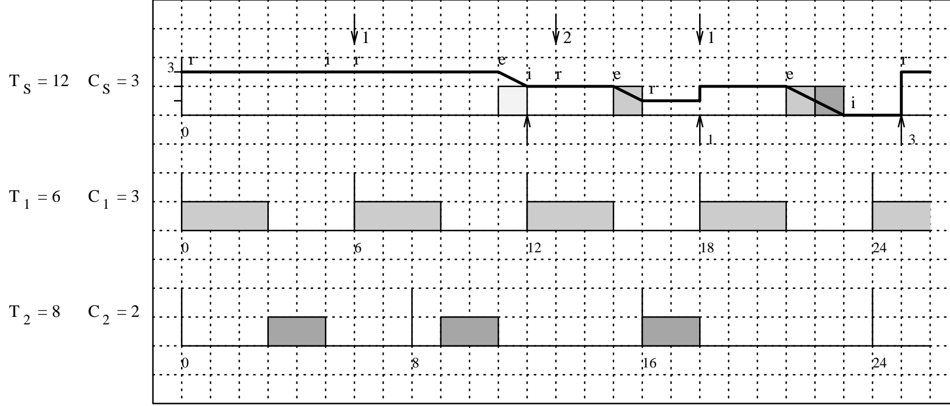


Figure 6: Delayed aperiodic executions with a Dynamic Sporadic Server.

There are two possible approaches that we can follow if we want to improve the aperiodic response times. The first is, of course, to use a Sporadic Server with a shorter period. The second, less obvious, is to assign a possible earlier deadline to each aperiodic request. The assignment must be done in such a way that the overall processor utilization of the aperiodic load never exceeds a specified maximum value U_S.

The second 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.

5.1 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 will be shown

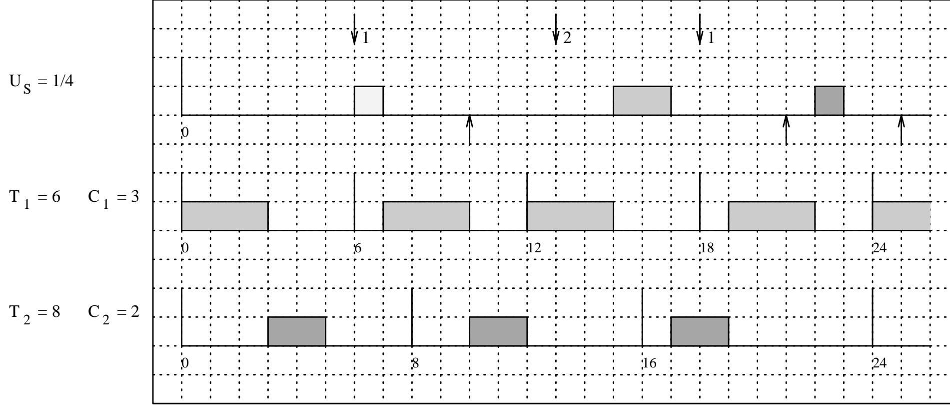


Figure 7: Total Bandwidth server example.

in Lemma 2, 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 7, the same situation of Figure 6 is treated with a TB server instead of a Sporadic Server. 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 $t = 22$. The response times of the first two requests are considerably improved, while for the third one we have no changes.

In Figure 8, a TB server with a high bandwidth is shown. Note that the response times of the aperiodic requests are very short. This is due to the high value of U_S, which lets the requests to demand their computation time with short deadlines, that is, with high priority.

5.2 Total Bandwidth Schedulability

Since we have defined the TB server in such a way that the aperiodic load never exceeds U_S, we expect to achieve a full processor utilization. As for the Sporadic Server, we first need to prove that the aperiodic processor utilization does not actually

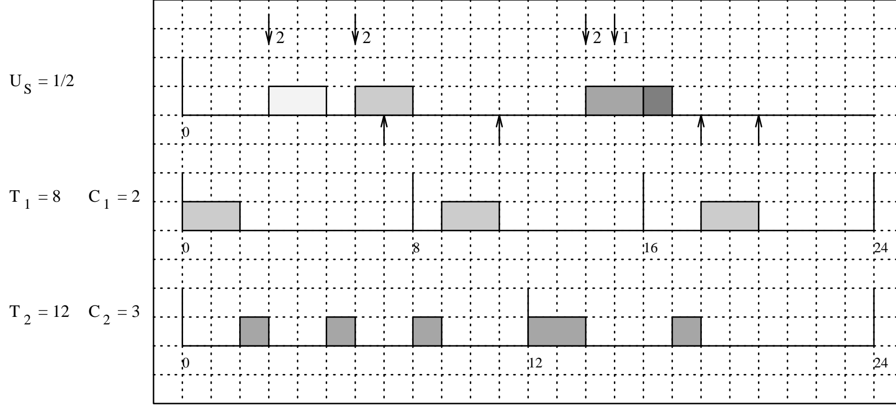


Figure 8: High capacity Total Bandwidth server example.

exceeds U_S.

Lemma 2 *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*

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

Proof. By definition

$$C_{ape} = \sum_{t_1 \leq r_k, d_k \leq t_2} C_k.$$

Given the deadline assignment of the TB server, there must be two indexes k_1 and k_2 such that

$$\sum_{t_1 \leq r_k, d_k \leq t_2} C_k = \sum_{k=k_1}^{k_2} C_k.$$

It follows that

$$\begin{aligned}
C_{ape} &= \sum_{k=k_1}^{k_2} C_k \\
&= \sum_{k=k_1}^{k_2} [d_k - \max(r_k, d_{k-1})]U_S \\
&= U_S \sum_{k=k_1}^{k_2} [d_k - \max(r_k, d_{k-1})] \\
&\leq U_S[d_{k_2} - \max(r_{k_1}, d_{k_1-1})] \\
&\leq U_S(t_2 - t_1).
\end{aligned}$$

□

Now we can prove the claimed result.

Theorem 3 *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

$$U_P + U_S \leq 1.$$

Proof. "If". Suppose there is an overflow at time t. The overflow is preceded by a period of continuous utilization of the processor. Furthermore, from a certain point t' on, only instances of tasks (periodic or aperiodic) ready at t' or later and having deadlines less than or equal to t are run. Let C be the total execution time demanded by these instances. Since there is an overflow at time t, we must have

$$t - t' < C.$$

We also know that

$$\begin{aligned} C &\leq \sum_{i=1}^{n} \left\lfloor \frac{t-t'}{T_i} \right\rfloor C_i + C_{ape} \\ &\leq \sum_{i=1}^{n} \frac{t-t'}{T_i} C_i + (t-t')U_S \\ &\leq (t-t')(U_P + U_S). \end{aligned}$$

It follows that

$$U_P + U_S > 1,$$

a contradiction.

"Only If". If an aperiodic request enters the system periodically, say each $T_S > 0$ units of time, and has execution time $C_S = T_S U_S$, the server behaves exactly as a periodic task with period T_S and execution time C_S. Being the processor utilization $U = U_P + U_S$, again from Theorem 7 of [11] we can conclude that $U_P + U_S \leq 1$. □

5.3 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 practically negligible.

6 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 7, 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. In order to do this, the idle times of a particular EDF schedule of the periodic task set are computed, and an optimal replenishment policy for the capacity of an aperiodic server is derived from these values.

6.1 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 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

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

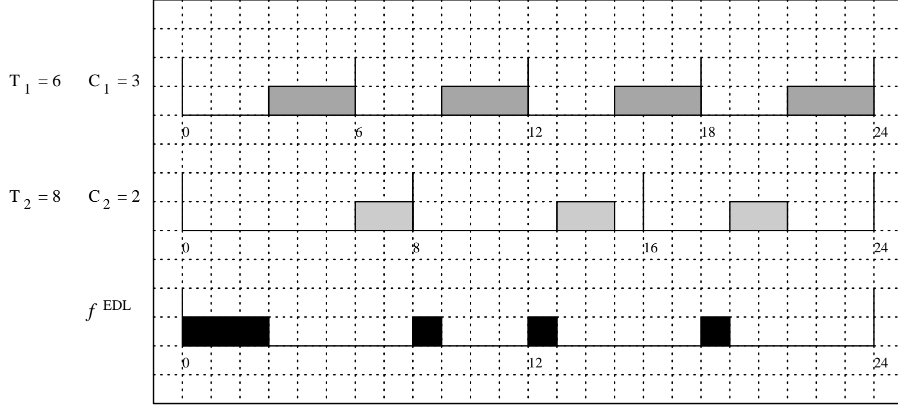


Figure 9: Availability function under EDL.

defined with respect to a task set Y and a scheduling algorithm X. The function $f_{\mathcal{J}}^{\text{EDL}}$, with $\mathcal{J} = \{\tau_1, \tau_2\}$, is depicted in Figure 9. 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 4 *Let $\mathcal{A}$ be any aperiodic task set and X any preemptive scheduling algorithm. For any instant t,*

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

□

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* $H = \text{lcm}(T_1, \ldots, T_n)$, 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, $\mathcal{D}^* = (\Delta_0^*, \Delta_1^*, \ldots, \Delta_p^*)$, represents the lengths of these idle times. The two vectors for the example of Figure 9 are shown in Figure 10 (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 4 will give us the optimality of the server built

i	0	1	2	3
e_i	0	8	12	18
Δ_i^*	3	1	1	1

Figure 10: Idle times under EDL.

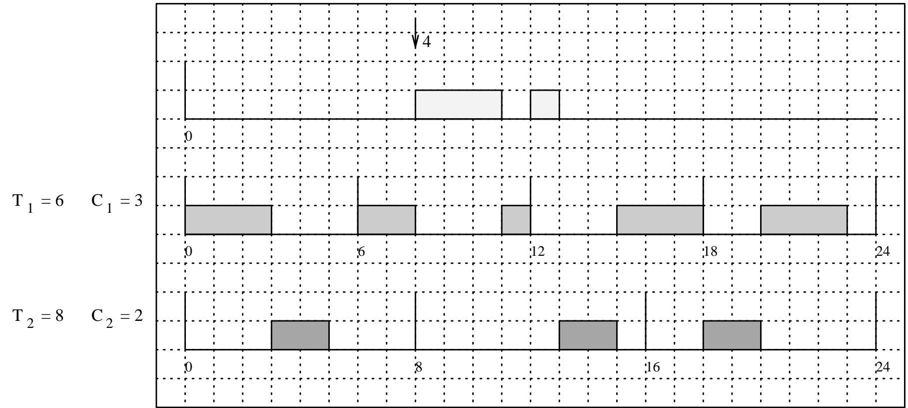
with this idea.

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 11 for an example. Here, an aperiodic request with an execution time of 4 units arrives at time $t = 8$. The idle times of an EDL scheduler are recomputed using the current periodic tasks, as shown in Figure 11b. The request is scheduled according to the newly computed idle times (Figure 11a). Note that the response time of the aperiodic request is optimal.

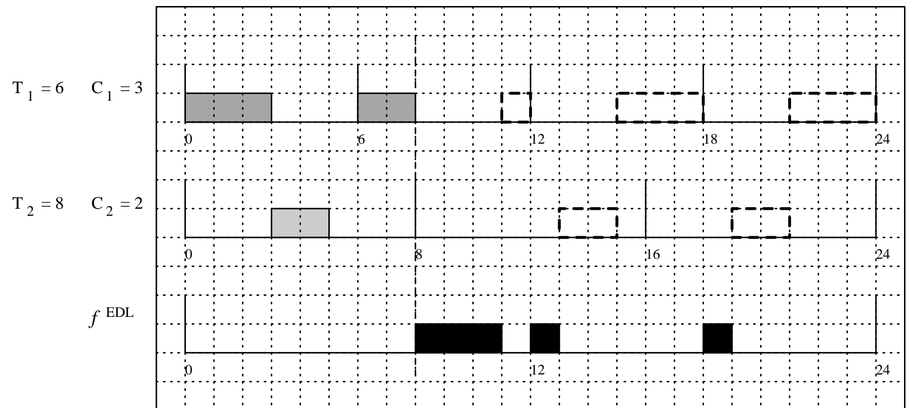
The procedure to recompute at each new arrival the idle times of EDL applied to $\mathcal{J}(t)$ is described in [4] and is not reported here. It is interesting to notice that not all the idle times have to be recomputed, but only those preceding the deadline of the current active periodic task with the longest deadline. However, 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.

6.2 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



(a)



(b)

Figure 11: a) Example of schedule produced with an EDL server. b) New EDL idle times at the aperiodic arrival.

schedule, without compromising the timeliness of the periodic tasks. This is more precisely stated in the following Theorem.

Theorem 5 *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).

Proof. "If". The condition is sufficient for the schedulability of the periodic task set under EDF (Theorem 7 of [11]), thus even under EDL, which is a particular implementation of EDF. The algorithm schedules the periodic tasks according to one or the other implementation, depending on the absence or the presence of aperiodic requests in the system. In the latter situation the executions of the aperiodic tasks are scheduled during the precomputed idle times of the periodic tasks. In both cases the timeliness of the periodic task set is unaffected, that is, no deadline is missed.

"Only If". Trivial, since the condition is necessary even for only the periodic task set (Theorem 7 of [11]). □

We now want to establish another nice property of the EDL server. In particular, we want to prove the property of optimality addressed above, that is, the response times of the aperiodic requests under the EDL algorithm are the best achievable. This is exactly what is stated by the following Lemma.

Lemma 3 *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*

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

Proof. Suppose J arrives at time t. Let $\mathcal{J}(t)$ be the set of the current active periodic instances (ready but not yet completed) and the future periodic instances. The new task J is scheduled together with the tasks in $\mathcal{J}(t)$. In particular, consider the schedule S of $\mathcal{J}\cup\{J\}$ under X. Let X' be another algorithm that schedules the

tasks in $\mathcal{J}(t)$ at the same time as in S, and S' be the corresponding schedule. J is executed during some idle periods of S'. Applying Theorem 4 with the origin of the time axis translated to t (this can be done since X is on–line), we know that for each $t' \geq t$

$$\Omega^{\mathrm{EDL}}_{\mathcal{J}(t)}(t, t') \geq \Omega^{X'}_{\mathcal{J}(t)}(t, t').$$

Recall now that, when there are aperiodic requests, the EDL server allocates their executions exactly during the idle times of EDL. Being

$$\Omega^{\mathrm{EDL}}_{\mathcal{J}(t)}(t, c^{\mathrm{EDL\ server}}_{\mathcal{J}\cup\{J\}}(J)) \geq \Omega^{X'}_{\mathcal{J}(t)}(t, c^{\mathrm{EDL\ server}}_{\mathcal{J}\cup\{J\}}(J))$$

it follows that

$$c^{\mathrm{EDL}}_{\mathcal{J}\cup\{J\}}(J) \leq c^{X}_{\mathcal{J}\cup\{J\}}(J).$$

That is, under the EDL server, J is never completed later than under X. □

7 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. The time “advanced” to the periodic instances becomes idle time that the server can use to schedule aperiodic requests, in addition to the idle time of an ideal EDL scheduler.

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 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.

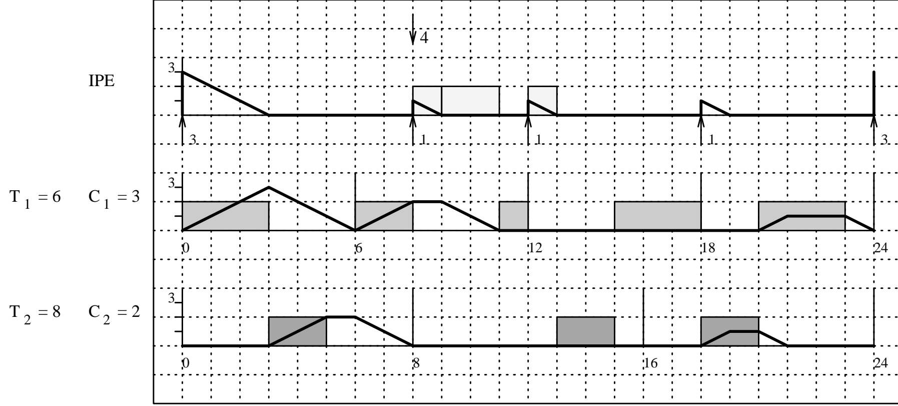


Figure 12: Improved Priority Exchange server example.

7.1 Definition of the IPE Server

The algorithm described here, called Improved Priority Exchange (IPE), is based on the idea pointed out above. In particular, 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 11 is scheduled with an IPE server in Figure 12. Note that the server replenishments are set according to the function $f_{\mathcal{J}}^{\mathrm{EDL}}$, illustrated in Figure 9.

When the aperiodic request arrives at time $t = 8$, one unit of time is immediately allocated to it by the server. However, other two units are available at the priority level corresponding to the deadline 12, due to previous deadline exchanges, and are allocated right after the first one. The last one is allocated later, at time $t = 12$, when the server receives a further unit of time. In this situation, the optimality of the response time is kept. As we will show later, there are only rare situations in which the optimal EDL server performs slightly better than IPE, that is, almost always IPE exhibits a nearly optimal behaviour.

7.2 IPE Server Schedulability

In order to analyze the schedulability of an IPE server, it is useful to define a transformation among schedules similar to that defined for the DPE server. In particular, given a schedule S produced by the IPE algorithm, we build the schedule S' in the following way:

- each execution of periodic instances during deadline exchanges (*i.e.*, increase in the corresponding aperiodic capacity) is postponed until the capacity decreases;
- all other executions of periodic instances are left as in S.

In this case, the server is not substituted with another task. Again S' is well defined and is invariant, that is, it does not depend on S, but only on the periodic task set. Moreover, S' is the schedule produced by EDL applied to the periodic task set (compare Figure 9 with Figure 12). The optimal schedulability is stated by the following Theorem.

Theorem 6 *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).

Proof. "If". The condition is sufficient for the schedulability of the periodic task set under EDF (Theorem 7 of [11]), thus even under EDL, which is a particular

implementation of EDF. Now, observe that in each schedule produced by the IPE algorithm the completion times of the periodic instances are never greater than the completion times of the corresponding instances in S', which is the schedule of the periodic task set under EDL. That is, no periodic instance can miss its deadline. The thesis follows.

"Only If". Trivial, since the condition is necessary even for the periodic task set only (Theorem 7 of [11]). □

7.3 Resource Reclaiming

The resource reclaiming, that is, 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.

7.4 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 $\mathcal{D}^*$.

8 Performance Results

DPE, DSS, 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 pos-

sible: 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 has been compared with the other algorithms. In this case, a periodic task for aperiodic service is created and, given its period and its maximum capacity, it is scheduled as any other periodic task. When the server is run, if aperiodic requests are pending they are served within the limit of the server capacity, otherwise the current periodic instance is completed.

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.

8.1 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 three graphs shown in Figure 13 correspond to three different periodic loads, low, medium and high, as addressed

above. 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 graphs, for small and medium periodic loads the two algorithms do not have significant differences in their performances. However, even for a high periodic load, the difference 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 algorithm 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.

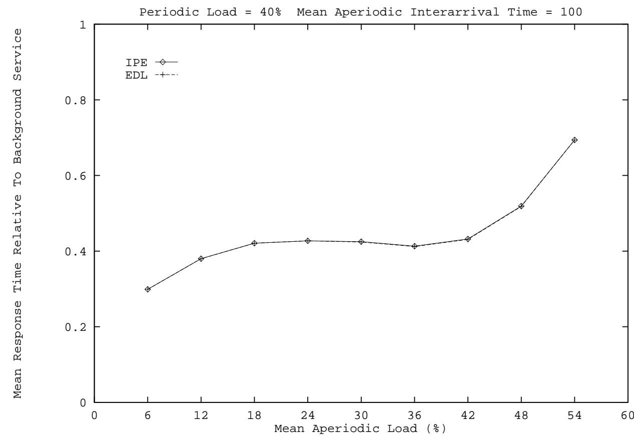
In summary, IPE has nearly the same performance of EDL, but with much less overhead. For this reason, the EDL server performance is not reported in all subsequent simulations. Moreover, the performance of the IPE server will be the reference in the following experiments.

8.2 Experiment 2: Response Time vs. Aperiodic Load

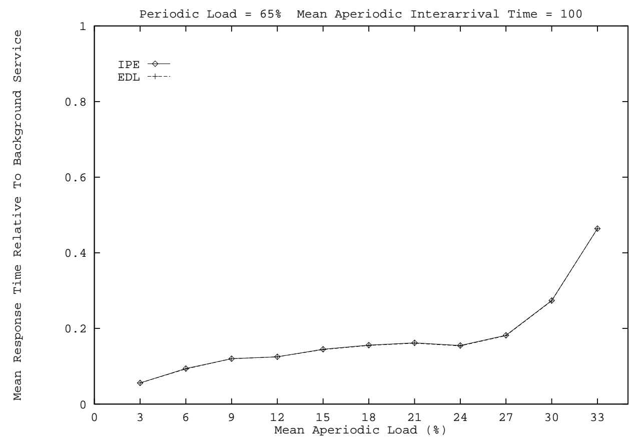
In the second experiment, we tested the performance of all algorithms as a function of the aperiodic load. The load was varied by changing the average aperiodic service time, while the average interarrival time was set at the value of $T_a = 100$.

Figure 14 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 $U_{ape} = \frac{T_s}{T_a}$.

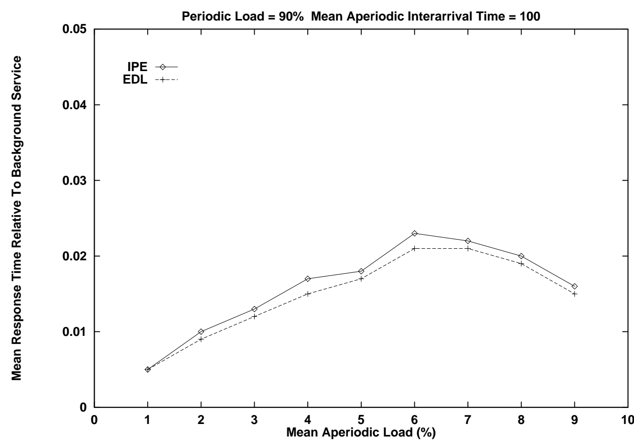
As can be seen from each graph, the TBS and IPE algorithms can provide a significant reduction in average aperiodic response time compared to background or



(a)

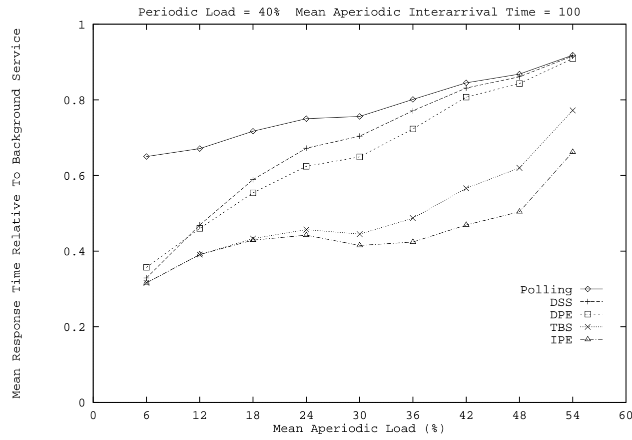


(b)

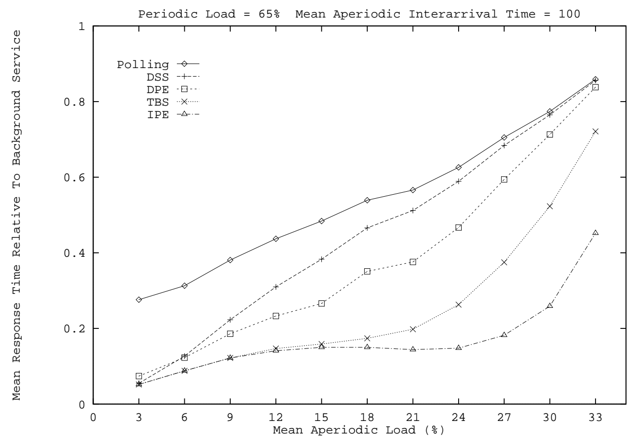


(c)

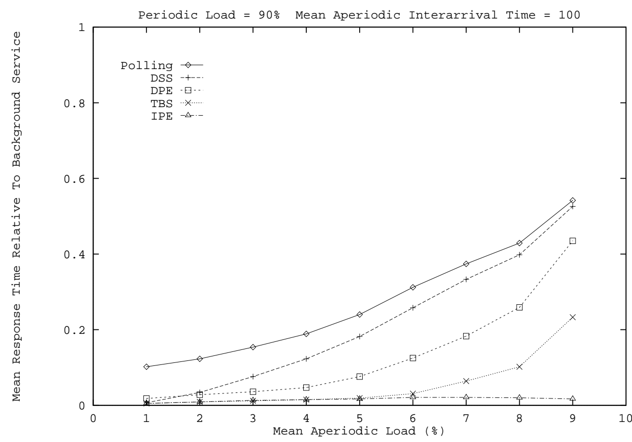
Figure 13: Comparison between IPE and EDL server.



(a)



(b)



(c)

Figure 14: Algorithms performance with different processor loads.

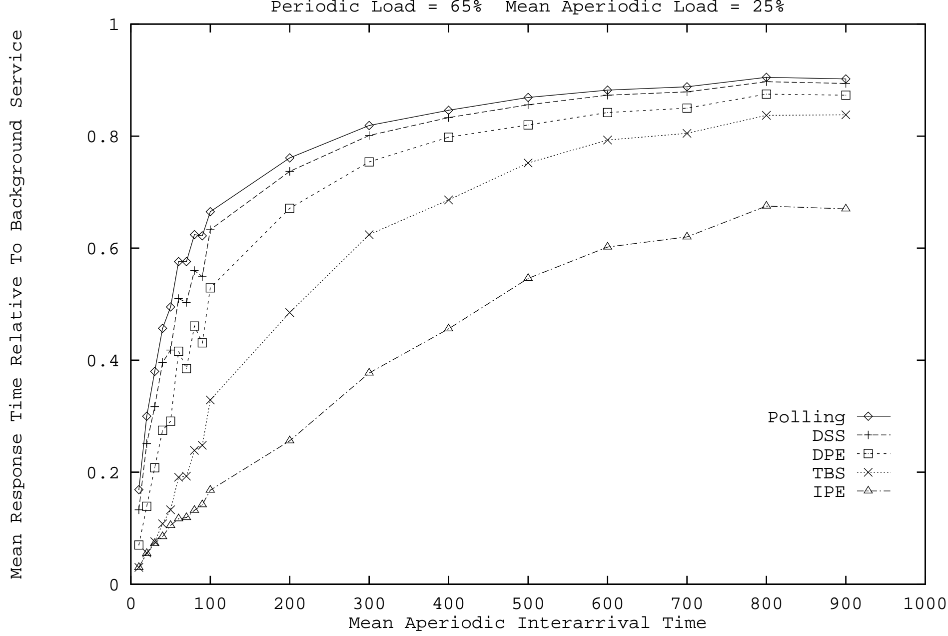


Figure 15: Algorithms performance with increasing aperiodic interarrival times.

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 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.

8.3 Experiment 3: Response Time vs. Interarrival Time

The performance of the proposed algorithms has also been compared as a function of the interarrival time T_a. Since the period of periodic tasks was chosen between 100 and 1000 units of time, the average interarrival time of aperiodic tasks was varied from 10 and 900 time units. In this experiment, the average periodic load was fixed at $U_p = 65\%$, and the average aperiodic load was set at $U_{ape} = 25\%$.

In order to maintain the aperiodic load constant, the average aperiodic service time T_s was computed as $T_s = T_a \cdot U_{ape}$. As a consequence, in the graph reported in

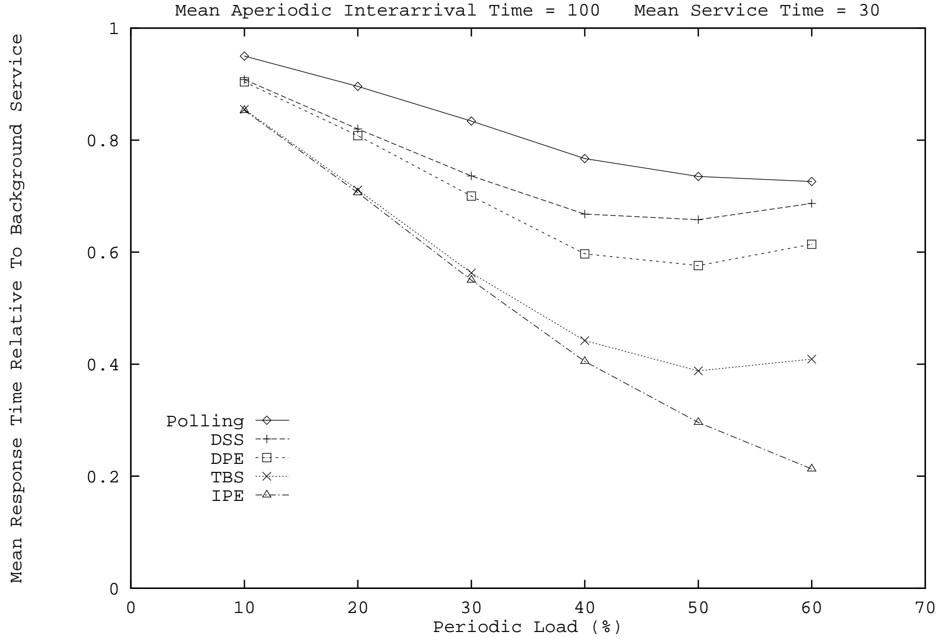


Figure 16: Algorithms performance with increasing periodic load.

Figure 15, the smaller the interarrival time on the x–axis, the smaller the aperiodic service time. This means that, for low values of T_a the aperiodic load is generated by a large number of small tasks, whereas for higher values of T_a the same aperiodic load is generated by a small number of long tasks.

As can be seen from the figure, all algorithms perform much better when the aperiodic load is generated by a large number of small tasks rather than a small number of long activities. Moreover, note that, as the interarrival time T_a increases, and the tasks' execution time becomes longer, the IDE algorithm shows its superiority with respect to the others, which tend to have about the same performance, instead.

8.4 Experiment 4: Response Time vs. U_p

In this experiment, the proposed algorithms have been compared with different periodic loads U_p. The graph shown in Figure 16 plots the average aperiodic response times when the processor utilization factor was varied from 10% to 60%. In this simulation, the aperiodic load was generated by setting $T_a = 100$ and $T_s = 30$, thus the total load was varied from 40% to 90%.

As can be seen from the graph, for very low periodic loads all aperiodic service algorithms show a behaviour similar to background service. As the periodic load increases, their performance improves substantially with respect to background service. In particular, DPE and DSS have a comparable performance, which tends to approach that of the Polling server for high periodic loads. On the other hand, TBS and IPE outperform all other algorithms in all situations. The improvement is particularly significant with medium and high workloads. With a very high workload, TBS is no more able to achieve the same good performance of IPE, even though it is much better than the other algorithms.

8.5 Experiment 5: 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 3.3, 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 jth instance of the ith 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 17. 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.

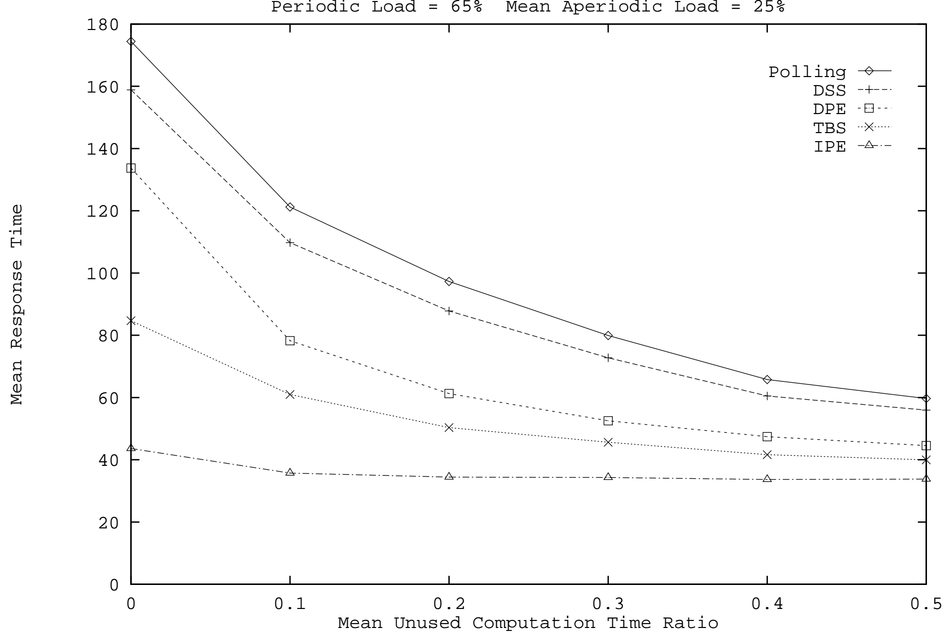


Figure 17: Response times vs. unused computation times.

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.

9 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 proven 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, although 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 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 Sys-*

tems, 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.