Búrdalo Rapa, LA.; Terrasa Barrena, AM.; Espinosa Minguet, AR.; García Fornes, AM. (2012). Analyzing the effect of gain time on soft task scheduling policies in real-time systems. IEEE Transactions on Software Engineering. 38(6):1305-1318. doi:10.1109/TSE.2011.95.

Analyzing the Effect of Gain Time on Soft Task Scheduling Policies in Real-Time Systems

Luis Búrdalo, Andrés Terrasa, Agustín Espinosa and Ana García-Fornes

Abstract—In hard real-time systems, *gain time* is defined as the difference between the worst-case execution time of a hard task and its actual processor consumption at run time. This paper presents the results of an empirical study about how the presence of a significant amount of gain time in a hard real-time system questions the advantages of using the most representative scheduling algorithms or policies for aperiodic or soft tasks in fixed-priority preemptive systems. The work presented here refines and complements many other studies in this research area, in which such policies have been introduced and compared. This work has been performed by using the authors' testing framework for soft scheduling policies, which produces actual, synthetic, randomly-generated applications, executes them in an instrumented real-time operating system, and finally processes this information to obtain several statistical outcomes. The results show that, in general, the presence of a significant amount of gain time reduces the performance benefit of the scheduling policies under study when compared to serving the soft tasks in *background*, which is considered the theoretical worst case. In some cases, this performance benefit is so small that the use of a specific scheduling policy for soft tasks is questionable.

Index Terms—Real-Time systems, RT-Linux, scheduling policies.

1 Introduction

In the field of hard real-time systems, the main goal is to achieve that none of the so-called *hard tasks* in the system ever fails to meet its temporal requirements, usually defined in terms of deadlines. The current practice for achieving this goal is to adopt a certain scheduling paradigm in the development of the real-time system. The paradigm imposes both a particular task model at design time and a corresponding scheduling policy at run time, and then provides the system designer with a formal, *off-line* feasibility analysis by which it is possible to prove whether all hard tasks will be able to meet their deadlines before the system starts running. One of the most sound and widespread paradigms is fixed-priority preemptive scheduling. In this paradigm, the task model requires each hard task to have some known temporal attributes (release times, computation times, deadlines, etc.) and a fixed priority. At run time, the system always selects the ready task with the highest priority for execution in a preemptive manner.

Hard real-time systems may also include some other tasks without hard or strict deadlines, which are normally referred to as *soft tasks*. The scheduling paradigm typically considers that the execution of a soft task produces some utility value to the system if the task can be completed before some point in time (related to the task's arrival time), after which this value progressively decreases; in contrast, the utility value of a hard task instantly drops to zero after reaching its deadline. Soft tasks are by definition not included in the off-line guarantee analysis, resulting in two main consequences. First, the system is not a priori committed to run them in a given time. And second, soft tasks are less restricted by the task model; in particular, their worst-case execution times or their exact arrival patterns do not need to be determined at design time. Thus, the general way to deal with soft tasks in hard real-time systems is to try to run them as soon as possible when they arrive to the system (thereby maximizing their utility), without compromising the deadlines of hard tasks. In systems following the fixed-priority preemptive paradigm, the trivial solution for this is to assign soft tasks a lower priority than any hard task, which relegates them to running in the background. In order to improve the poor quality of service obtained by this *background policy*, many authors have proposed specific scheduling algorithms or policies for soft tasks. These policies are normally run-time algorithms that work in a compatible way with the fixed-priority preemptive scheme by which hard tasks are dispatched.

The off-line feasibility (or *schedulability*) analysis is based on comparing the temporal requirements of each hard task against its theoretical worst-case running scenario. In order to do so, one of the input parameters of the analysis is the worst-case execution time (WCET) of each hard task. This is probably the most difficult issue in the system design, since obtaining an accurate value of a task's WCET can be very complex, or even impossible, depending on the characteristics of both the task's code and the hardware on which the code is to be executed. An extensive study of different techniques and tools, as well as existing trends and open issues in the field of timing analysis in real-time systems can be found in [30].

• *L. Búrdalo, A. Terrasa, A. Espinosa and A. García-Fornes*
Departamento de Sistemas Informáticos y Computación
Universitat Politècnica de València
Camino de Vera SN, 46022, Valencia, Spain
E-Mail: {lburdalo,aterrasa,aespinos,agarcia}@dsic.upv.es

This work is partially funded by research projects PROMETEO/2008/051, CSD2007-022 and TIN2008-04446.

Digital Object Indentifier 10.1109/TSE.2011.95

Since an underestimated value of a hard task's WCET can make an apparently safe system crash at run time, the traditional approach has been to overestimate the WCETs of hard tasks. On the other hand, even if an accurate estimation of a task's WCET can be assumed, the worst-case behavior is actually very rare, since tasks do not always take the exact worst-case path within their code, and thus, it is often the case for tasks to consume only a fraction of this maximum time, as pointed out in [22]. Taking these two facts into account, it can be concluded that the usual case for hard tasks is to consume less processor time than their WCETs at run time, and often very significantly less. In real-time systems, the difference between a task's WCET and its actual processor consumption at run time is referred to as *gain time*.

When related to WCET overestimation, gain time has traditionally been considered as a design problem for hard tasks, but also as a benefit for soft tasks. The problem with WCET overestimation of hard tasks is that it restricts the ability of the system to be schedulable on a particular processor. This means that the system may be wrongly rejected by the off-line analysis, thereby forcing the system designer either to redesign the hard task set or to run the system in a faster (and more expensive) hardware than actually needed. However, from the viewpoint of soft tasks, gain time is considered an advantage, since it increases the expected amount of processor time available to their execution. In fact, some scheduling algorithms for soft tasks are designed to make an effective use of gain time, in order to further improve the running opportunities of soft tasks.

On the contrary, this paper presents the results of an empirical study about how the presence of a significant amount of gain time in a real-time system considerably *reduces* the advantages of using some of the most representative scheduling policies for soft tasks in fixed-priority preemptive systems. This work refines and complements many other previous studies in this research area, in which these scheduling policies have been introduced and compared, usually in a theoretical way or by means of simulations. For this reason, the study has considered some of the usual assumptions in the previous work regarding the specification of the experiment load and the evaluation of the scheduling algorithms. In particular, the most important assumptions are the following: soft tasks are assumed to have no deadlines, soft tasks are dispatched in FIFO order, and the performance of the algorithms is measured by means of the average response time of soft tasks.

The study presented here has been carried out by using the authors' testing framework for soft scheduling policies. The framework first generates synthetic test programs and then runs each program on an instrumented operating system (a modified version of Open Real-Time Linux) that implements the scheduling policies under study. As a result of each execution, the framework automatically produces a complete set of statistical data about the performance of the scheduling policy. The framework has been carefully designed in order to make the results of the different scheduling policies comparable, which basically involves two main aspects. First, the policies themselves have been implemented in order to run the applications on equal terms; in particular, all policies have a compatible interface of system calls, which allow application tasks to have exactly the same code regardless of the policy running the application. And second, the results of each execution are processed in order to make all experiments comparable with each other. Furthermore, the combination of experiments with different *factors* (such as amount of hard tasks, hard task utilization, soft task utilization, etc.) can be used to determine to what extent each of these factors *individually* affects the behavior of the different scheduling policies.

The main results of this paper show that, in general, the fact that hard tasks consume less execution time than their estimated WCETs (which in turn produces the availability of gain time) negatively affects the performance *benefit* of using any of the policies under study with respect to scheduling soft tasks in background. This is also true even for those policies that are specifically designed to efficiently reclaim and use gain time. In nearly all cases, this performance benefit is significantly reduced as the amount of gain time increases in the system. Under some conditions, this performance benefit is so small, or even negative, that the use of a specific scheduling policy for soft tasks becomes questionable. The final purpose of this work is for it to be used as a guide to determine which scheduling policies for soft tasks are more appropriate depending on the running conditions of the system and, specifically, the amount of gain time that is available at run time.

This paper is structured as follows: First, Section 2 describes the scheduling policies that take part in this work and some of the results obtained in previous comparative studies. Section 3 introduces the framework used to generate and run the experiments designed for this study, which are described in Section 4. The results of the experiments are presented in Section 5. Finally, Section 6 discusses the conclusions of the study.

2 Previous Work

2.1 Scheduling Policies

This study includes five of the most representative scheduling policies for aperiodic or soft tasks in fixed-priority preemptive real-time systems: *Deferrable Server* [21] (DS), *Sporadic Server* [2] (SS), *Extended Priority Exchange* [25] (EPE), *Dynamic Approximate Slack Stealing* [1], [13] (DASS), and *Dual Priorities* [11], [13] (DP). The execution of soft tasks in background, or *Background scheduling* (BG), is also included in the study as a lower bound in the performance of soft task scheduling.

Server-based scheduling policies are founded on the idea of reserving some execution bandwidth for soft tasks by means of adding a special task called "server"

to the hard task set which in turn runs the soft tasks. The priority and temporal parameters of the server (period and computation time, also called *budget* or *capacity*) are adjusted to off-line guarantee the entire task set. Both the Deferrable Server and the Sporadic Server work in a very similar way at run time. The main difference between them is the run-time strategy they use to replenish their budgets as soft tasks use them. This, in turn, limits the particular off-line equations that can be applied to analyze their schedulability. These conditions are more pessimistic for DS than for SS.

The Extended Priority Exchange algorithm uses a more complicated run-time strategy than the two previous algorithms. This strategy is based on the fact that there may be some available capacity for running soft tasks at each priority level as well as producing dynamic priority exchanges among tasks in order to preserve and use this capacity in an advantageous way for soft tasks. The initial capacity available at each priority level is computed off line in order to guarantee the schedulability of hard tasks, but it can be increased at run time if hard tasks consume less than their WCET.

Slack-based algorithms are based on delaying the execution of hard tasks in order to run soft tasks as soon as possible without missing any hard deadline. The family of slack scheduling algorithms includes some exact [20], [12], [23], [24] and approximate [13] versions. Among these algorithms, the Dynamic Approximate Slack Stealing algorithm is the only one that is feasible in practice, since the others present an excessive temporal or spatial overhead. The DASS is based on a fine-grained run-time supervision of the application tasks' execution, in order to keep track of the available *slack time* at each (hard task's) priority level. Then, soft tasks can safely run before hard tasks while there is slack time available in the system (that is, at all active hard priority levels) without compromising any hard deadline.

The Dual Priorities algorithm is based on assigning two priorities to each hard task, an *upper band* and a *lower band*, while soft tasks run in a *middle band*. The middle and lower bands have to be below the upper band of any hard task. At run time, every hard task starts its periodic activations in its lower band until a *promotion time* is reached; then, it runs the rest of the activation in its upper band. The system may assign any priority ranges to the middle and lower bands, as long as the hard task set is schedulable in the upper band. Compared to DASS, the main benefit of the DP algorithm is that it needs very little run-time supervision by the system, since promotion times can be calculated off-line.

Because of the purpose of this study, it is important to note that for both the DASS and DP algorithms, some extensions have been developed in order to *reclaim* gain time as it becomes available at run time, and then to use this gain time to run soft tasks. These extensions, originally defined in [20], [13] (for slack-based algorithms) and [13], [11] (for DP), are referred to as *Propagated Gain Time* and *Self Gain Time* in this paper:

- *Propagated Gain Time*. Both DASS and DP admit an extension by which the available gain time of any hard task i (g_i) is computed every time it ends an activation (g_i is calculated by subtracting the actual computation time spent by the task in the activation from the task's WCET). By definition, this time may be used to run soft tasks at task i's or any *lower* priority level (hence the name *propagated*).
  The implementation of the propagated gain time extension is different for each algorithm. In particular, DASS with this extension adds g_i to the slack time available at task i's and lower priority levels, thereby increasing the amount of time that all (active) hard tasks may be safely delayed to run soft tasks. On the other hand, DP with this extension may delay for g_i time units the promotion times of hard tasks with priorities lower than i, thereby increasing the opportunities for running soft tasks (in their middle band).
- *Self Gain Time*. This extension is exclusive for the DP algorithm. In DP, the promotion time of each hard task is computed off-line in such a way that the task can safely run (for its entire WCET) *after* reaching this promotion time. Thus, if at the beginning of an activation, the task is allowed to run for some time in its lower band, then it is safe to delay its promotion for that amount of time in the current activation, potentially increasing the amount of time soft tasks may be run in their middle-band priority.

2.2 Previous Comparative Studies

This section first presents the main conclusions of the simulation studies made by other authors. It must be noted that results from different simulation studies are difficult to compare because not all of them consider the same policies and they do not present comparable testing strategies. However, it is commonly accepted that the performance of the soft task scheduling policies is measured by means of the average response time of soft tasks, which are usually considered to have no deadlines and are served in FIFO order. The final part of the section concisely presents some general results derived from the authors' empirical testing framework, where all policies have been tested on equal terms.

In general, server-based policies improve the results obtained by scheduling soft tasks in background when the system's total utilization (including hard and soft tasks) is not too high; however, as the utilization of hard tasks grows, these policies tend to perform like background scheduling. When comparing the DS and SS policies, different studies do not come to the same conclusions. Studies in [17], [21], [2], [27] conclude that SS is better than DS because it allows for larger capacities and gets higher utilization values, while [18] shows larger response times for SS than for DS and. Finally, [5] concludes that both policies have similar response times and can get similar utilization values. Compared

to servers, [25] shows that EPE obtains better results than DS when the hard utilization is high.

Slack-based policies are taken into account in several studies. When compared with the server-based and EPE policies [18], [15], [13], [10], [11], the main conclusion is that slack-based algorithms outperform all of them. However, some of these studies also state that the main drawback of slack-based algorithms is that most of them are not practicable due to their high overhead. In particular, the *Dynamic Slack Stealing* (DSS) policy is commonly used as a reference to compare other policies since it has been proved to be optimal (see [12]) in the sense that it minimizes the response times of soft tasks without missing any hard deadlines (however, Tia et al. [29] showed some situations in which it is better not to use spare capacity immediately and therefore, optimality cannot be achieved). When compared with this optimal or *exact* version, the DASS exhibits a close performance with much less overhead. In particular, the study in [13] shows that the DASS presents a performance that is very near to DSS until the total load in the system (hard+soft+overhead) gets to 90%, at which point the system performance starts to degrade. The results in [15] show that DASS is very near to DSS in all cases, although in this study overhead is considered to be negligible.

The *Dual Priorities* scheduling policy (DP) is compared to other scheduling policies in [10], [11], [15], [16]. The results are very similar in all the studies: DP gets lower response times than BG, EPE and server-based policies. In fact, DP performs in a similar way to DASS when the system utilization is less than 90%, although response times are better for slack-stealing-based policies. [16] also shows that DP performs better than BG if and only if soft load is served in FIFO order.

Nearly all these studies do not consider the run-time overhead produced by the specific scheduling policies. Some studies do consider overhead, but in terms of theoretical worst-case costs (in orders of magnitude). The only studies in which run-time overhead is included are [18], which uses the number of context switches in the different simulations to approximate the total overhead produced by the scheduling policies, and [13], which includes extra CPU cycles in the simulations to approximate the cost of calculating the available amount of slack in the system. This study also includes the implementation of BG, DSS and DP in a real operating system and some overhead results, which show that DP presents a moderately higher amount of overhead than BG, while DSS incurs in such a great overhead penalty that makes it unfeasible in practice in systems with large numbers of hard tasks.

In contrast with these simulation studies, a general empirical study running real applications in a Real Time Operating System (RTOS) was presented in [8]. This study presented three general results: (1) in general, all the algorithms perform better than BG, even considering overhead; (2) all policies improve their performances compared to background as the *hard task utilization* grows; and (3), these performance improvements of all policies with respect to BG tend to disappear as the *number of hard tasks* increases, and the same happens as the *soft task utilization* grows. In addition, the study also presented some conclusions for each policy. The two server-based policies (DS and SS) perform much worse than DASS or DP, in spite of producing less overhead. In particular, differences in performance range from 15% in systems with low utilization up to 40% or more in heavy loaded systems. When compared, SS always performs better than DS, although the difference between the two policies is not very significant unless the system presents a high hard task utilization. This confirms the theoretical disadvantage of DS respect to SS about having a more restrictive feasibility test, which leads to lower server capacity and poorer performance. However, SS is more difficult to implement and produces more overhead than DS. Due to the overhead, the performance of the DASS policy is worse than expected (in the simulation studies) and most of the times it is outperformed by DP. This confirms the conclusions of [11]. However, DASS gets slightly better results than DP in heavy loaded systems, especially when the total utilization gets close to 100%.

Some of these scheduling policies have been subject to more recent studies in the field of multi-processor systems. For example, SS has been adapted and optimized to be effectively used in multi-processor systems [14]; DS has been shown to improve the performance of soft tasks when compared to BG in asymmetric multi-processor systems [9]; and both an optimal slack-based policy and DP have been used in order to globally allocate soft tasks among processors [3], [4], with DP outperforming the slack policy in heavy loaded systems.

3 The Testing Framework

This section summarizes the framework used to generate and run the experiments described later in the paper. As depicted in Figure 1, the framework is basically composed by four modules: Load Generator, Code Generator, the instrumented RTOS, and Result Extractor. These modules are now described, placing special emphasis on the main design ideas that support the validity of the results presented in the paper.

3.1 The Load Generator Module

The framework can be configured to generate tests for many different scenarios, depending on the particular goals of the experiment. This configuration is mainly carried out in the input file of the Load Generator module, or *load specification file*. According to the specification included in this file, the module generates the experiment's set of *task set specification* (TSS) files as an output. The load specification file contains the desired values of the parameters that the load generator will combine in order to create the task sets, as well as the number of task sets to generate for each parameter combination (or number of *replicas*). The main parameters that can

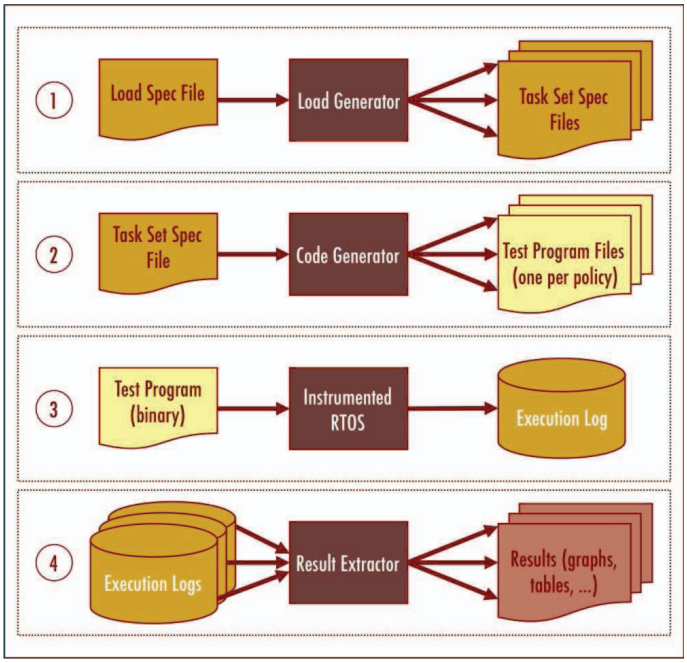

Fig. 1. The Testing Framework

be specified in the file include the number of hard and soft tasks, hard tasks utilization, soft tasks utilization, the priority levels for soft tasks, maximum hyperperiod, and hard tasks gain time. The Load Generator considers all possible combinations of the input parameters and then, for *each* combination, it generates as many task sets (TSS files) as the number of specified replicas. As a result, each TSS file contains a complete specification of a task set, including the number of tasks, their types (hard or soft) and their attributes (execution times, budgets, deadlines, periods, priority bands, etc.).

Each task set is randomly generated within the limits of its corresponding combination of input parameters, with two main restrictions: the set of hard tasks has to be schedulable, and the budgets assigned to soft tasks in server-based policies are maximized (while keeping the hard tasks schedulable). In particular, for each task set to generate, the Load Generator follows this procedure:

1) The period of each task (T_i) is randomly generated, according to three input parameters: the maximum hyperperiod of the task set, the type of random distribution to use (uniform or exponential) and the range of this distribution (maximum and minimum values). The set of generated periods is accepted if the resulting hyperperiod is not greater than the maximum specified value; otherwise, the process starts again.
2) According to the input specification, tasks are separated into two groups: hard and soft. Then, in each group, the individual utilization of each task (U_i) is computed by using the Uunifast algorithm [6] (which has been shown to produce unbiased random task sets) according to two input parameters per group: the number of tasks and the utilization.
3) The WCET of each task (C_i) is calculated as follows: $C_i = T_i U_i$. The input specification may establish a minimum WCET; if so, the task set is discarded if any C_i is below this value (and then the procedure returns to the first step).
4) The deadline for each task (D_i) is generated between C_i and T_i by using a uniform distribution.
5) The group of hard tasks is reordered by deadline (lowest deadline first). Then, hard tasks are assigned priorities following a deadline monotonic policy (the lower the deadline, the higher the priority). At run time, the actual priority of each task will depend on the particular scheduling policy, but in any case, the relative priority among hard tasks is maintained according to this initial assignment.
6) For each hard task i, its computation time ($Comp_i$) is calculated as a function of its WCET and the gain time percentage (G) specified in the input file:

$$Comp_i = \frac{(100 - G) * C_i}{100} \quad (1)$$

7) The budgets for the servers in DS and SS policies are calculated as the maximum values that keep the hard task set schedulable according to the exact schedulability test (based on the maximum response time of each task) required by each server policy. If the hard task set is not schedulable for any of the two tests, then the task set is discarded and the procedure returns to the first step.
8) The initial parameters of some policies, such as the promotion times of hard tasks in the dual-related policies or the initial aperiodic time available for hard tasks in the EPE algorithm, are calculated.
9) In order to better approach the usual run-time behavior of tasks, and also to ensure that different policies running the same task set will face exactly the same load, the framework also generates a list of *activations* per task. In the list, each activation corresponds to a task's release and it stores the run-time parameters that may vary in different releases of the same task. In particular, for each activation k of any task i, two values are calculated:
   a) The actual computation time of the release ($Comp_i[k]$). This value is computed by using a normal distribution with a mean equal to $Comp_i$ and a standard deviation value defined by the input specification.
   b) The time of the next release ($T_i[k]$). If task i is periodic, this value is always set to T_i. Otherwise, if task i is aperiodic, this value is randomly generated by using a uniform distribution with a range centered on T_i plus-minus some percentage value defined by the input specification.

After this procedure successfully generates a schedulable task set, all its relevant temporal parameters are written in the corresponding TSS file.

3.2 The Code Generator Module

The Code Generator module is responsible for producing actual *test programs* for each TSS file (task set), specifically, one test program per soft scheduling policy to be tested. Test programs are C source files that contain synthetic code generated for a particular TSS file and scheduling policy, in a compatible way to the system interface of the instrumented RTOS on which tests will be run, which is compatible with the POSIX standard. Each task set requires the generation of a different test program per policy because in POSIX, the selection of the scheduling parameters (priority and policy) for each task is performed by the application at run time.

The generation of test programs has been designed to make the running conditions of each test program both as close to its specification (TSS file) as possible, and as similar for all policies as possible, so that results are not biased for any particular policy. In order to achieve these goals, the test programs in the group corresponding to the *same* TSS file (one program per policy) are generated to meet the following four requirements:

- In every program in the group, execution starts at a *critical instant*, in which all tasks are simultaneously released. This is the worst scenario for soft tasks, but it is also a straightforward way of making all scheduling policies start on equal terms.
- In every program in the group, any given task i is generated in order to follow the sequence of release times ($T_i[k]$) specified in the TSS file. This ensures that the release pattern of each task will be the same for every policy.
- In every program in the group, any given task i is generated to have exactly the same code. This is possible because the policies under study have been designed with a compatible set of system calls, and the selection of the particular policy is performed at run time.
  For each task i, the code is generated in order to spend a computation time equal to $Comp_i[k]$ for each release k, as specified in the TSS file. In order to do so, the framework incorporates a calibration mechanism that adjusts the generation of code to the expected computation time for a particular processor. In practice, this calibration mechanism achieves actual execution times to be between 90% and 100% of the expected ones.
- In every program in the group, hard tasks are released for two consecutive hyperperiods, while soft tasks are released during the first hyperperiod only. This is further discussed below.

In the simulation studies mentioned in Section 2, each experiment is carried out by executing the simulated workload for a given period of time, which is normally the hyperperiod of the experiment's hard task set. The rationale for this is that the release pattern of the hard task set is repeated after each hyperperiod, and the same is true for the running conditions for the soft workload. However, with this approach, it may happen that some soft workload is pending (i.e., awaiting execution) at the end of the first hyperperiod, especially in heavily loaded systems. Furthermore, for any given task set, the amount of pending soft workload at the first hyperperiod may be notably different from one scheduling policy to another. The problem with this is that the measurements corresponding to pending soft workload are not included in the experiment's results.

In the framework described in this paper, the solution for this is to generate the soft workload during the first hyperperiod, in such a way that the specified soft utilization is met, but to run the experiment until there is no pending soft workload, even if this happens after reaching the first hyperperiod. In this way, for any given task set, the running conditions are equal to all the scheduling policies, and the results of each policy always include the entire soft workload. However, if any two policies running the same task set finish at different times, the results related to the system *overhead* are not directly comparable. Taking both restrictions into account, the strategy adopted by the framework is to generate the soft workload during the first hyperperiod of each experiment, but to run the experiment until exactly its second hyperperiod, in order to guarantee that both the performance results are complete for all policies and that their overhead results are comparable. This solution is valid as long as the experiment design (in particular, the sum of hard and soft utilizations, plus the overhead) allows the entire soft workload to be completed before the second hyperperiod.

3.3 The Instrumented RTOS

The instrumented Real-Time Operating System (RTOS) on which test programs are run is a modified version of Open Real-Time Linux [31], which is a small, hard real-time executive running under Linux. From the application's perspective, Open RT-Linux provides a programming interface that is compatible with the POSIX standard [19]. Internally, this RTOS has a simple run-time behavior, in which the system deals with each hardware interruption or system call invocation by means of a specific function and then, in all cases, the same scheduling function is called; this function is the one that is responsible for selecting the new task to be run, and then dispatching it, which provokes a context switch if the selected task is different from the running one. The original scheduling policy that Open RT-Linux applies by default for all tasks is fixed-priority preemptive (that is, POSIX's "SCHED_FIFO"), which has been used by the framework as the background (BG) policy. Because of its simple design and small size, Open RT-Linux produces very low and predictable overhead at run time. According to the experiments presented in this paper, carried out on a Pentium III 700Mhz computer, the average cost of the aforementioned scheduling function in RT-Linux ranges from 4.5μs (4 tasks) to 12μs (16 tasks),

while the maximum cost ranges from 12μs (4 tasks) to 30μs (16 tasks).

The framework has extended Open RT-Linux in two main ways: the implementation of the soft scheduling policies under study and the incorporation of a tracing mechanism by which it is possible to collect run-time information as applications are executed.

The soft scheduling policies under study have been implemented and provided as new scheduling choices for soft tasks at run time. To do this, some of these policies had to be redesigned in order to be provided with POSIX-like interfaces (as presented in [7]). The framework provides a specialized version of the system scheduler module for each policy, in order to avoid the implementation of any given policy to affect any other (in terms of overhead).

The tracing mechanism introduced to the RTOS includes a general, POSIX-like tracing system, and a particular instrumentation of the RTOS code which traces some system events (e.g., scheduling decisions, context switches, costs of the scheduler's functions, etc.) in order to analyze the behavior of the different scheduling policies. As a result, the execution of any real-time program on the instrumented RTOS can automatically produce a log file containing the events traced inside the RTOS during this execution. Both the instrumentation and the collection of events have been designed to make the overhead related to event tracing predictable and equivalent for all policies (the average cost of tracing each event is under 500 ns in a 700Mhz Pentium III processor, as shown in [28]). In particular, the set of system events and their instrumentation points are the same for all policies; at run time, events are traced to memory during the program execution, being dumped to the log file only after the application tasks have stopped running.

3.4 The Result Extractor Module

The Result Extractor module is responsible for extracting useful information from the logs generated by the instrumented RTOS when it runs the test programs. In particular, the extractor module works in three steps, which are presented below.

In the first step, the extractor module opens the log corresponding to an individual test program execution and then traverses it in order to calculate some predefined *metrics*. A metric is defined as a property of a program's execution. Examples of metrics are some temporal properties of application tasks (response times, execution times), costs of RTOS functions, number of context switches, etc. The output of this step is a basic statistical analysis of each metric (average, maximum, minimum and standard deviation). Among all the metrics, some of them are selected to be the *relevant results* of the tests (for example, the average response time of soft tasks), and then they are further processed by the module.

In the second step, the module combines the relevant results of all the executions corresponding to the *same* task set (one per policy). For each result, the module calculates the result *ratio* for each policy as the division of the policy's result value and the corresponding value obtained by the reference policy, which is BG. This ratio thus represents the result *difference* of using this scheduling policy with respect to using the BG policy for this particular task set.

In the third step, after the relevant result ratios have been computed for each task set, the extractor module combines the result ratios corresponding to *different* task sets in the experiment. The corresponding result ratios of different task sets are directly comparable with each other because ratios express the relative differences of results between a particular policy and the BG policy in each task set. For each relevant result, this third and final step obtains two types of outcomes: a *global value* per policy, expressing the average ratio for all the task sets in the experiment, and the *variation* of the result ratio for each policy as a function of the different input parameters of the experiment (or *factors*), such as the soft load utilization or the amount of gain time, for example. This variation is computed by grouping the result ratios of all tests according to the different values of this factor, and then calculating the average ratio value for each group. For example, if an experiment comprised task sets with four different values of gain time (0%, 25%, 50% and 75%), one possible outcome would be the variation of the soft task response time of each policy as a function of gain time. For each policy, the module would obtain four values: it would first classify all the experiment results according to the gain time (in four groups), and then, for each group, it would compute the average of the soft task response time ratios obtained by this policy in all the experiments in the group. These four values would show the variation (or evolution) of the result ratio for this policy as the factor varies.

4 Experiment Design

The main goal of this study is to determine to what extent the presence of gain time in real-time systems with hard and soft tasks influences the performance of the most representative scheduling algorithms for soft tasks in fixed-priority preemptive real-time systems. In order to better describe the experiments and their results, the following two concepts related to the hard task set in the system are defined: the *nominal* hard utilization, or nominal hard load, is the theoretical utilization of the hard task set, derived from the WCET values established at the schedulability analysis. On the other hand, the *real* hard utilization is the actual utilization of the hard task set at run time, derived from the real execution time consumed by hard tasks as the system runs. Thus, the difference between the nominal and the real utilization in a given task set will determine the amount of gain time that will be available for soft tasks at run time. Please note that according to the framework described above, the experiment sets as input parameters both the

TABLE 1
Parameter summary of the experiments

Experiment 1: 40% *Nominal* hard load	
# hard tasks	4, 8, 12, 16
Policies	BG, DS, SS, EPE, DASS, DASS-GAIN, DP, DP-GSELF, DP-GPROP, DP-GBOTH
% Gain time	Soft task utilization
0	10% to 60% (in steps of 10%)
25	10% to 70% (in steps of 10%)
50	10% to 80% (in steps of 10%)
75	10% to 90% (in steps of 10%)
# replicas	50

Experiment 2: 80% *Nominal* hard load	
# hard tasks	4, 8, 12, 16
Policies	BG, DS, SS, EPE, DASS, DASS-GAIN, DP, DP-GSELF, DP-GPROP, DP-GBOTH
% Gain time	Soft task utilization
0	10% to 20% (in steps of 10%)
25	10% to 40% (in steps of 10%)
50	10% to 60% (in steps of 10%)
75	10% to 80% (in steps of 10%)
# replicas	50

percentages of nominal hard load and gain time, and then the real hard utilization is derived from them.

The main decision about the experiment design was to determine the number of experiments to be carried out and to select the amount of nominal and real hard utilization in each experiment. Some preliminary experiments with the same framework proved that differences in performance between systems with and without gain time increased along with the nominal hard utilization for all the scheduling policies. For this reason, a total of four experiments were designed, with each of them fixing a particular value of *nominal* hard utilization: 20%, 40%, 60%, and 80%. Then, for each experiment, the framework generated four series of task sets with different percentage values of gain time (0%, 25%, 50%, and 75%), thereby producing different values of *real* hard utilization. Because of size limitations, this paper presents the results of the two experiments that rendered the most significant results: the ones with 40% and 80% of nominal hard utilization[1]. The rest of this section presents the parameter configuration of both experiments in full detail (summarized in Table 1).

The next design decision was to determine which input specification parameters to fix and which others to vary in order to generate the task sets for each experiment. In particular, periods of hard tasks were generated by following a uniform distribution between 50 and 2000 milliseconds, with a maximum hyperperiod of 10 seconds. This distribution, along with its limits, were chosen in order to produce task sets that are comparable with previous studies in the literature, such as [13], [18], [15] (some of these studies concluded that choosing a uniform or exponential distribution did not affect the results). On the other hand, the rest of the parameters in the specification were varied within certain limits in each experiment, in order to be able to study the influence of these parameters on the performances of the policies under study. In particular, for each experiment, the framework generated task sets for all the combinations of the following varying parameters: number of hard tasks (4, 8, 12, and 16), percentage of gain time (0%, 25%, 50%, and 75%) and soft load utilization (from 10% up to achieving 100% of total *real* utilization, in increments of 10%). For each possible combination of all input parameters, 50 replicas (different task sets) were generated. In the generation of each task set, the run-time variability parameters of hard and soft tasks were set in the following manner: the computation time of each (hard or soft) task was varied in an interval of $[-10\%, 10\%]$ of the task's real computation time (using a normal distribution), while the arrival pattern of each soft task was varied within an interval of $[-10\%, 10\%]$ of the task's period (using a uniform distribution). Please note that, as explained in Section 3, the framework always generates task sets in such a way that (1) hard task are schedulable according to the feasibility analysis, and (2) the budgets for the server (soft) tasks in server-based policies are as large as possible.

By combining all these different specification parameters, the total number of task sets for the experiments of 40% and 80% of nominal hard utilization were 6000 and 4000, respectively. For each task set, the framework generated a series of test programs, one for each of the ten scheduling policies considered in this study: Background (BG), Deferrable Server (DS), Sporadic Server (SS), Extended Priority Exchange (EPE), Dynamic Approximate Slack Scheduling (DASS), DASS with the propagated gain time extension (DASS-GAIN), Dual Priorities (DP), DP with the propagated gain time extension (DP-GPROP), DP with the self gain time extension (DP-GSELF) and DP with both types of gain time extensions (DP-GBOTH). As a result, the total number of test programs generated, compiled and executed for the two experiments were 60000 and 40000, respectively. All programs were run on a Pentium III 700Mhz computer, with 384Mb of RAM.

In every task set, the soft load was modeled (and generated) as a single aperiodic task configured to have the best running opportunities according to each scheduling policy under study: in BG, the soft task is always scheduled at the lowest priority. In SS and DS, the soft task is scheduled at the highest priority as long as there is some budget left, and it is otherwise relegated to running in the background. In EPE, the DASS-related and the DP-related policies, the soft task is always scheduled in a middle-band priority, while hard tasks start their activa-

1. Gain time had little effect on the performances of the scheduling algorithms in the experiment of 20% of nominal hard load, since the absolute amount of gain time was very small in this case, while the effect in the experiment of 60% of nominal hard load was intermediate between the results of the two experiments presented in the paper.

tions in their lower-band priorities (where the soft task can preempt them) and then they may change to their upper-band priorities if certain running conditions are reached (in particular, the available capacity is exhausted in EPE, or the available slack is exhausted in DASS, or the tasks' promotion times are reached in DP). According to some studies, this configuration may not be optimal in the case of DS and SS (in [5] it is shown that, by assigning the soft task the highest priority, the budget adjustment in server-based policies may not be optimal if task deadlines are lower than their periods). However, the selection of the optimal server parameters (budget, period and priority) is still an open research issue. For this reason, the experiments were set to schedule servers at the maximum priority, which is the most common approach in the literature.

In order to be able to compare the results with previous studies, the performances of the scheduling polices were measured by means of the average response time of soft tasks. In addition, the experiments also measured the overhead of the scheduling algorithms, in order to globally quantify the extra cost incurred by each algorithm and to relate it to its performance, if possible. In particular, the experiments measured two overhead indicators in each test: the number of context switches and the total scheduling time spent by the RTOS (the cumulated cost of the scheduling function inside the RTOS for the duration of each test). In this context, it has to be noted that the potential effect of the overhead on the performance of a scheduling policy depends on the proportion between the overhead values and the computation times of the application tasks (the effect increases as task computation times get closer to the scheduling costs). Taking this into account, the experiments were configured in order to generate task sets with task computation times in a reasonable range when compared to the average scheduling overhead of the reference policy (BG) measured on the same testing hardware. In particular, the experiments were configured for this *base overhead* to be around 10% of the total execution time, which is considered to be enough to influence the policy performance, but still within a reasonable range when compared to the overhead in real systems.

5 Results

The experiments considered three *relevant results* for each scheduling policy in each test: the soft task response time, the number of context switches, and the total scheduling cost, with the first one measuring the performance and the last two measuring the overhead of the scheduling algorithms. As explained in Section 3.4, these relevant results are expressed in relative terms (ratio values) with respect to the respective results obtained by BG, which is the reference policy. Thus, a ratio value of 1.0 expresses a result that is equal to the one obtained by BG, a ratio value of 0.9 expresses a result that is 10% lower than the result obtained by BG, and so on.

The following subsections analyze the results obtained in the two experiments: 40% and 80% of nominal hard utilization, named *Experiment 1* and *Experiment 2*.

5.1 Experiment 1: 40% of Nominal Hard Utilization

The global performance results of the experiment for each scheduling policy are presented in Table 2. This table shows the performance difference between each policy and the BG policy by means of a set of percentile values of the soft task response time ratios for all the 6000 task sets in the experiment (hence including all combinations of the varying parameters in the experiment: number of hard tasks, gain time, and soft load).

TABLE 2
Soft task response time ratios in Experiment 1 (percentile values)

	Percentiles				
Policy	5%	25%	50%	75%	95%
DS	0,8850	1,0007	1,0026	1,0058	1,0135
SS	0,8590	1,0011	1,0030	1,0061	1,0126
EPE	0,6721	0,9035	0,9692	0,9957	1,0073
DASS	0,5149	0,7568	0,8876	0,9594	0,9967
DASS-GAIN	0,5129	0,7403	0,8661	0,9384	0,9867
DP	0,5575	0,7837	0,8947	0,9551	0,9920
DP-GSELF	0,5416	0,7822	0,8928	0,9545	0,9919
DP-GPROP	0,5570	0,7845	0,8931	0,9552	0,9924
DP-GBOTH	0,5429	0,7822	0,8924	0,9540	0,9919

According to the values in the table, both DS and SS get better results than BG in a very low number of task sets only (the 5^{th} percentile values are 0.88 in DS and 0.85 in SS, meaning that in 5% of the task sets, these algorithms get 12% and 15% of improvement in the soft task response time over BG). However, all values from the 25^{th} percentile on are higher than 1.0 for both policies, which means that in at least 75% of the cases, they present slightly *negative* benefits when compared to BG, due to their extra overhead. DASS-related and DP-related policies perform better than BG in all cases, but only significantly better in a reduced number of task sets (their improvement over BG is around 25% in their 25^{th} percentile, but this figure is reduced to 5% of improvement in their 75^{th} percentile). In addition, the ratio values show almost no difference among these six policies, which implies that the gain time reclaiming extensions of DASS or DP do not improve the results obtained by these two algorithms in the experiment. Finally, the ratios of EPE show intermediate results between DS/SS and DASS/DP-related policies. The improvement of EPE over BG is 10% or higher in 25% of cases, it is negligible in at least 50% of the cases, and it is negative in at least 5% of cases, again due to its extra overhead.

The performance difference between each specific policy and BG can be further analyzed by considering the effect of gain time. In Figure 2, the average ratio of the soft task response time of each policy is represented as a

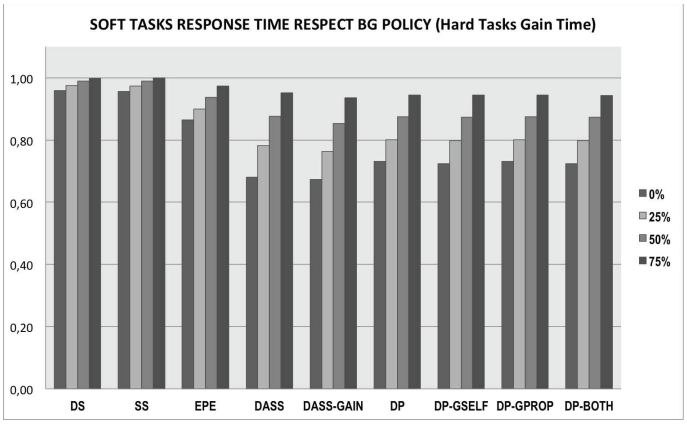

Fig. 2. Soft task response time ratios as a function of gain time in Experiment 1

function of the available gain time. For each policy, the graph presents the average of the ratio values obtained by the policy in all the task sets with a particular amount of gain time. As can be observed, the ratio values for *all* policies become closer to 1 as the amount of gain time grows, meaning that the performance difference between each policy and BG is reduced as the amount of available gain time increases. This effect is more acute in DASS-related and DP-related policies, since they obtain much better results (between 30% to 35% of benefit) in systems with no gain time than in systems with 75% of gain time (where the benefit is only around 5%).

Finally, Figure 3 shows the combined effect of gain time with the increase of the amount of soft load in the system. This is done by showing four graphs, each one depicting the average values of the soft task response time ratios as a function of the amount of soft load, in task sets with a specific amount of gain time (starting from the top left, graphs correspond to tests where hard tasks had 0%, 25%, 50%, and 75% of gain time, respectively). Please note that the number of points for every policy in each graph is different because task sets were produced in such a way that the soft task utilization was generated in increments of 10% until a 100% of total (hard plus soft) *real* utilization was reached. There are three relevant aspects to be pointed out about these graphs. First, in the four graphs, the results classify policies in three groups (the two server-based policies, EPE, and the DASS-related and DP-related policies) with little difference among the policies in each group. This is consistent with the conclusions derived from both Table 2 and Figure 2. Second, considering each graph separately, the performance benefits of all policies with respect to BG are reduced as the amount of soft load increases in the system (this confirms the conclusions of [8], independently of the amount of gain time available in the system). And third, considering the four graphs together, the performance benefits of *all* policies with respect to BG are reduced, some of them severely, as the amount of gain time increases in the system. In the last graph (75% of gain time), there is practically no benefit in using any of the policies, especially for a high amount of soft load.

TABLE 3
Average overhead ratios (and std. dev.) in Experiment 1 (values in % of increment with respect to BG)

Policy	# C. Switch	Total sched. Cost
DS	+0,90 (3,24)	+14,36 (13,25)
SS	+1,42 (4,42)	+18,58 (15,23)
EPE	+8,35 (6,04)	+53,36 (25,76)
DASS	-0,79 (3,06)	+43,15 (23,42)
DASS-GAIN	+2,74 (4,99)	+43,99 (24,35)
DP	+2,37 (3,95)	+24,82 (17,04)
DP-GSELF	+2,34 (3,92)	+22,06 (16,45)
DP-GPROP	+2,38 (3,97)	+25,96 (17,41)
DP-GBOTH	+2,34 (3,94)	+24,98 (17,19)

Regarding the overhead results of the experiment, Table 3 displays the global values, in terms of the *increment* percentage with respect to BG of two average ratios: the total number of context switches (second column) and the total scheduling cost of each execution (third column). For each increment value, the number in parenthesis expresses its standard deviation. The context switch values in the table show that, except for the EPE algorithm (with 8% of increment), there is a small general penalty in the number of context switches for using specific policies for soft tasks rather than using BG (less than 3% in all these policies). The DASS algorithm even presents a negative value, meaning that this policy actually produces fewer context switches (on average) than BG. On the other hand, considering the total scheduling cost of each test, it is clear that there is a significant penalty for using specific policies with respect to using BG, especially in some of the policies. In particular, the extra overhead is considerably higher than BG in DASS-related policies and EPE.

5.2 Experiment 2: 80% of Nominal Hard Utilization

The global performance results of the experiment are shown in Table 4. This table shows the performance difference between each policy and the BG policy by means of a set of percentile values of the soft task response time ratios for all the 4000 task sets in the experiment (including all combinations of the varying parameters).

Comparing the data in this table with the global results of Experiment 1 (in Table 2), the policies in this second experiment present the following performances: the two server-based policies again present the worst results, only slightly better than in the previous experiment (both policies now perform better than BG in at least 25% of the cases). EPE now performs considerably better than BG in a large number of cases (20% better in 50% of the cases, and 40% better in half of them); in fact, EPE now show ratios that are similar to DASS, or even slightly better (from the 75^{th} percentile on).

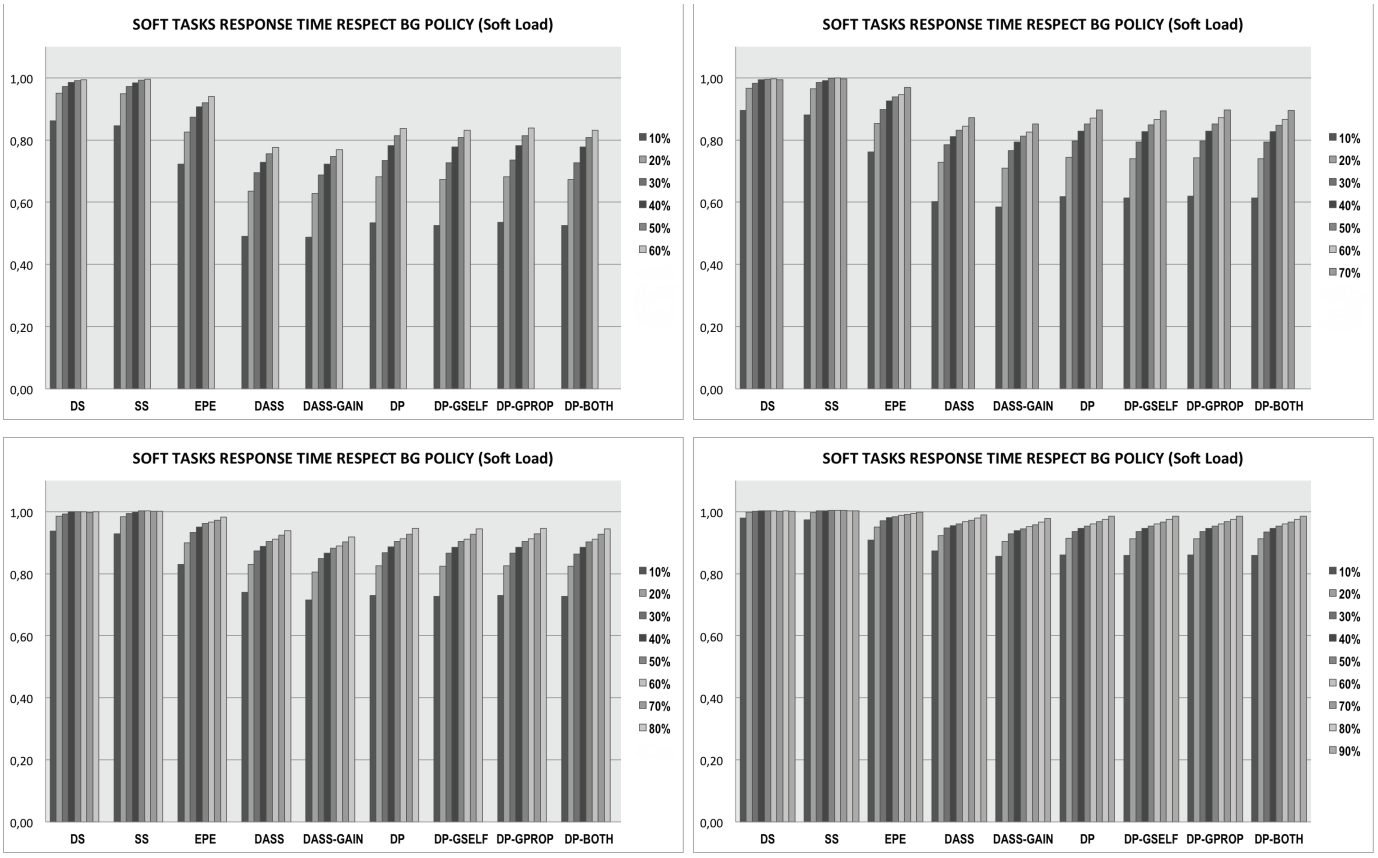

Fig. 3. Soft task response time ratios as a function of soft load in Experiment 1, divided by different values of gain time: 0% (upper left), 25% (upper right), 50% (lower left) and 75% (lower right)

TABLE 4
Soft task response time ratios in Experiment 2
(percentile values)

	Percentiles				
Policy	5%	25%	50%	75%	95%
DS	0,5672	0,9815	1,0028	1,0079	1,0196
SS	0,3549	0,9088	1,0007	1,0064	1,0162
EPE	0,2022	0,5872	0,8135	0,9318	1,0016
DASS	0,0761	0,5321	0,8137	0,9495	1,0018
DASS-GAIN	0,0632	0,2427	0,5147	0,7538	0,9068
DP	0,2076	0,4291	0,6334	0,8032	0,9543
DP-GSELF	0,1539	0,3674	0,6000	0,7958	0,9543
DP-GPROP	0,2083	0,4285	0,6333	0,8022	0,9535
DP-GBOTH	0,1539	0,3644	0,5983	0,7930	0,9532

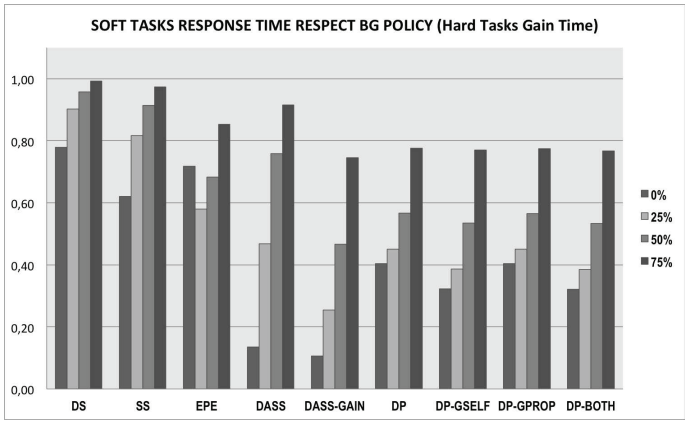

Fig. 4. Soft task response time ratios as a function of gain time in Experiment 2

In this experiment, there is a great difference between the two versions of DASS. For every percentile rank shown in the table, the ratio value of DASS is notably higher than the value of DASS-GAIN, meaning a better performance for the latter. Globally, DASS-GAIN obtains the best performance results in this experiment, while the performance of DASS is worse than all the dual-based policies, and sometimes worse than EPE. Finally, the performances of the four DP-related policies are quite homogeneous, with DP-GSELF and DP-GBOTH only moderately improving the results of the other two, and all of them being intermediate between DASS-GAIN and DASS. If compared with the previous experiment, the performances of all DP-related algorithms are now much better in all percentile ranks.

These global results are now refined by introducing the effect of gain time on the soft task response ratios, as shown in Figure 4. This graph again shows that gain time poses a negative effect on the performance benefit of all policies with respect to BG. In fact, comparing this graph with the one for the previous experiment,

the effect is now more severe for all policies. This general trend presents an exception, the EPE policy in systems with no gain time, which is further discussed below. When looking at specific policies, some relevant aspects may be pointed out: SS now outperforms DS, especially in systems with no gain time. EPE gets better performance than DASS in systems with large amounts of gain time (50% or higher). The effect of gain time on DASS is critical, where the performance benefit ranges from around 85% in systems without gain time to less than 10% in systems with a 75% of gain time); this effect is also evident in DASS-GAIN, but this latter policy only degrades up to an average benefit of 25% due to its ability to effectively use gain time for executing soft tasks. Finally, DP-GSELF and DP-GBOTH outperform both DP and DP-GPROP in systems with a low amount of gain time, but this difference tends to disappear as gain time augments.

The special case of the EPE policy is now discussed. According to the graph, EPE performs worse in systems without gain time than in systems where there is some gain time available (up to 50% of gain time). The reason for this can be related to (1) the extra overhead of this algorithm and (2) the inefficient way in which the algorithm computes off-line the initial aperiodic time available for hard tasks (which is more evident in this second experiment, where task sets have a high nominal hard utilization). The algorithm is designed to increase these aperiodic time values at run time by reclaiming gain time; however, in systems with no available gain time, EPE cannot compensate its poor initial configuration, and thus its performance gets closer to BG. Excluding this particular case, the effect of gain time on the EPE algorithm time shows the same trend than on any other policy.

Finally, Figure 5 shows the influence of the soft task utilization in the performance ratios of all policies in each of the four possible gain time values of the experiment. The conclusions that can be drawn from these graphs are similar to the ones in the previous experiment and are consistent with the general performance results of this experiment presented above: first, in systems with any particular value of gain time, the average performance benefits of all policies versus BG decrease as the amount of soft utilization grows. Second, for any given value of soft utilization, all policies exhibit less performance benefit with respect to BG as the amount of gain time increases in the system (except for the case of EPE in systems with no gain time, as discussed above). And third, all policies perform better now than in the previous experiment, for any given value of gain time. In this second experiment, there is a clear advantage of using some specific policies for soft tasks with respect to using BG, even in systems with a high percentage of gain time. Among such policies, the best results are rendered by DASS-GAIN and the four DP-related policies.

Table 5 presents the global overhead results for this experiment. When compared with the previous experiment, the values for the total scheduling costs are similar for each policy, while the values for the number of context switches are now higher in all policies except the DP-related ones. In particular, server-based policies have increased their context switch penalties with respect to BG around 4% (due to a higher number of times in which their budgets run out), the DASS-GAIN exhibits an increment of around 2% (due to the presence of a greater amount of absolute gain time, which allows more tasks to be run within the intervals of reclaimed gain time), and the EPE algorithm has incremented its penalty from around 8% to almost 12% (due to both more gain time available and a mechanism of reclaiming and using this time less efficiently than other algorithms, such as DASS-GAIN).

When analyzing the global overhead ratios as a function of the experiment parameters, the one with the greatest influence was, as expected, the number of hard tasks (since all the scheduling policies are based on certain computations to be performed over the entire list of hard tasks). In order to better show this, this second experiment was extended to incorporate task sets with more hard tasks (up to 32). The results are presented in Figure 6. The graph shows that the number of hard tasks in the system produces a linear increment of the total overhead ratio with respect to BG in all policies, but this effect is more pronounced in some policies than

TABLE 5
Average overhead ratios (and std. dev.) in Experiment 2 (in % of increment with respect to BG)

Policy	# C. Switch	Total sched. cost
DS	+4,48 (7,44)	+14,23 (12,98)
SS	+6,01 (9,75)	+18,05 (14,69)
EPE	+11,86 (8,02)	+51,04 (26,69)
DASS	+1,20 (5,85)	+43,09 (23,41)
DASS-GAIN	+4,14 (8,10)	+44,74 (24,79)
DP	+2,49 (5,40)	+23,32 (16,57)
DP-GSELF	+2,53 (5,53)	+21,84 (16,84)
DP-GPROP	+2,48 (5,41)	+23,52 (16,31)
DP-GBOTH	+2,52 (5,56)	+23,88 (17,51)

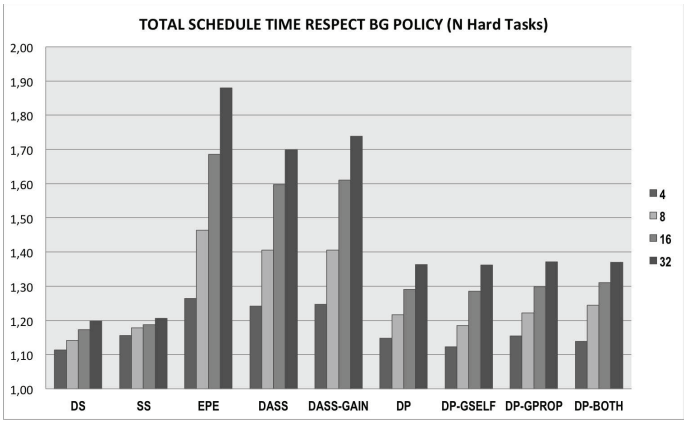

Fig. 6. Total overhead ratios as a function of the number of hard tasks in Experiment 2

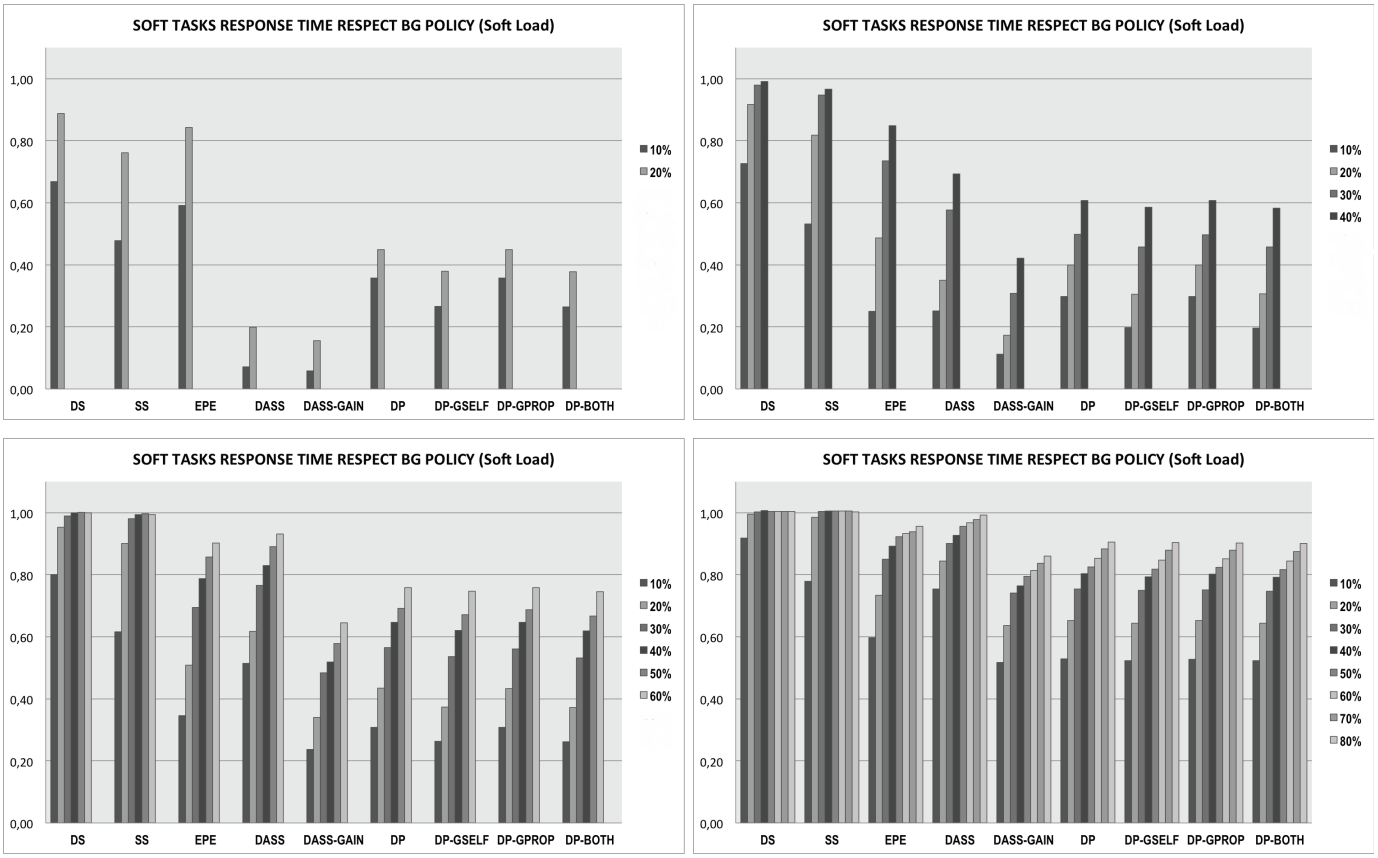

Fig. 5. Soft task response time ratios as a function of soft load in Experiment 2, divided by different values of gain time: 0% (upper left), 25% (upper right), 50% (lower left) and 75% (lower right)

in others. In particular, three different groups of policies can be observed: (1) the overhead values of DS and SS with respect to BG are barely affected; (2) for DASS, DASS-GAIN and EPE, the effect of this parameter is very significant (in EPE, the total scheduling cost is 26% higher than BG with 4 tasks, but it gets up to 87% higher with 32 tasks); and (3) the DP-related policies present an intermediate effect, in which the ratio worsens by around 7% every time the number of hard tasks doubles.

6 Conclusions

This paper has presented the results of an empirical study on the most relevant scheduling policies for soft tasks in fixed-priority, preemptive real-time systems. In particular, the goal of the study was to characterize the effect of gain time on the behavior of these scheduling policies. The existence of gain time, which is defined as the difference between the WCET of a hard task and its actual execution time, is typical in many real-time systems, for two main reasons. First, because the WCET overestimation is still a common practice in the design of many real-time systems in order to ensure the safety of the schedulability analysis. And second, because even if WCETs are accurately calculated, the typical case for tasks is to consume only a fraction of their WCETs at run time. Traditionally, gain time has been regarded as a design problem for hard tasks (when related to WCET overestimation), but also as an opportunity for soft tasks, which can use this spare time in order to improve their response times. Indeed, some scheduling policies for soft tasks have included specific extensions to make an effective use of this gain time.

The most general conclusion of the paper is that, other things being equal, the increase in gain time in the system significantly reduces the advantages of using any of the policies under study. More specifically, the relative performance *benefits* of all policies with respect to serving soft tasks in background (BG) are significantly *reduced* for all policies as gain time increases. This is consistent with the theoretical definition of these policies, where performance can be directly related to some policy variables that depend on the hard *nominal* load (such as the servers' budgets, the run-time available capacity/slack for EPE/DASS, or the promotion times for DP). Furthermore, the results presented in the paper have shown that this negative influence of gain time may affect policies differently, depending on some system parameters, as it is now summarized.

In systems with low hard nominal utilization, gain time produces a homogeneous negative effect on all policies with respect to BG. Although all policies still perform better than BG except in some particular cases (DS and SS actually perform worse than BG in systems

with high percentages of gain time, due to their extra overheads), adopting any of them becomes less worthwhile as the amount of gain time increases, especially for systems with high soft load utilization. Moreover, in the case of DASS or DP, their gain-time extensions have no effect on their respective performance benefits with respect to BG.

In systems with high hard nominal utilization, there is an even more pronounced negative impact of gain time on all policies (compared to BG), but this impact does not affect all policies in the same way. Both server-based policies provide good results when no gain time is available, but they rapidly degrade to BG as gain time augments, since their budgets become artificially small, and they cannot compensate this at run time. Moreover, as soft utilization grows, they end up performing worse than BG due to their extra overhead (especially due to the higher number of context switches, which is also a consequence of server budgets being very small). It has to be noted that SS has been implemented according to its official definition by POSIX; a recent study [26] claims that this definition has some defects which directly affect its performance and proposes some corrections, which have not been incorporated to the standard yet. EPE performs worse when there is no gain time available, because of being unable to compensate both its high overhead and its inefficiency at computing the initial capacity values of hard tasks. However, in systems with gain time, it presents much better results, and it ends up outperforming both server-based policies and DASS. Gain time has a very strong negative effect on DASS, which makes this policy degrade dramatically as gain time augments. In this case, its gain time extension becomes vital to compensate this degradation, to the extent that DASS-GAIN outperforms all other policies, even considering its extra overhead. Finally, the four DP policies (DP plus its three gain time extensions) present the most stable behavior in the performance results as gain time augments. In this case, the incorporation of gain time extensions does not produce a clear benefit, and all policies tend to perform equally (and equal to DASS-GAIN) with greater values of gain time. Also, since that DP has a straightforward implementation and produces little overhead, this policy is probably the best choice for these systems.

Acknowledgments

This work is partially funded by research projects PROMETEO/2008/051, CSD2007-022 and TIN2008-04446.

References

[1] N.C. Audsley, R.I. Davis, A. Burns, and A.J. Wellings. Appropriate mechanisms for the support of optional processing in hard real-time systems. In *11th IEEE Workshop on Real-Time Operating Systems and Software*, pages 23–27, 1994.

[2] L. Sha B. Sprunt and J. Lehozky. Aperiodic task scheduling for hard real-time systems. *The Journal of Real-Time Systems*, (1):27–60, 1989.

[3] J. M. Banus, A. Arenas, and J. Labarta. An efficient scheme to allocate soft-aperiodic tasks in multiprocessor hard real-time systems. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications - Volume 2*, pages 809–815, 2002.

[4] J. M. Banus, A. Arenas, and J. Labarta. Dual priority algorithm to schedule real-time tasks in a shared memory multiprocessor. *Parallel and Distributed Processing Symposium, International*, 2003.

[5] G. Bernat and A. Burns. New results on fixed priority aperiodic servers. In *IEEE Real-Time Systems Symposium*, pages 68–78, 1999.

[6] Enrico Bini and Giorgio C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30:129–154, 2005.

[7] L.A. Búrdalo, A. Espinosa, A. García-Fornes, and A. Terrasa. Framework for the development and evaluation of new scheduling policies in RT-Linux. In *OSPERT 2006*, pages 42–51.

[8] L.A. Búrdalo, A. Espinosa, A. Terrasa, and A. García-Fornes. Experimental results of aperiodic fixed-priority preemptive policies in RT-Linux. In *OSPERT 2007*, pages 10–19, 2007.

[9] John M. Calandrino, Dan P. Baumberger, Tong Li, Scott Hahn, and James H. Anderson. Soft real-time scheduling on performance asymmetric multicore platforms. In *IEEE Real Time Technology and Applications Symposium*, pages 101–112, 2007.

[10] R. Davis. Dual priority scheduling: A means of providing flexibility in hard real-time systems. Technical Report YCS230, University of York, UK, May 1994.

[11] R. Davis and A. Wellings. Dual priority scheduling. In *Proceedings of the 16th IEEE Real-Time Systems Symposium*, pages 100–109, 1995.

[12] R. I. Davis, K.W. Tindell, and A. Burns. Scheduling slack time in fixed priority preemptive systems. In *IEEE Real-Time Systems Symposium*, pages 222–231, Dec 1993.

[13] Robert Ian Davis. *On exploiting spare capacity in hard real-time systems*. PhD thesis, Department of Computer Science, University of York, 1995.

[14] D. Faggioli, M. Bertogna, and F. Checconi. Sporadic server revisited. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 340–345, 2010.

[15] A. García-Fornes, A. Terrasa, and V. Botti. Planificación de tareas aperiódicas en sistemas de tiempo real estricto. *NOVATICA*, Septiembre:22–30, 1997.

[16] B. Gaujal, N. Navet, and J. Migge. Dual-priority versus background scheduling: A path-wise comparison. *Real-Time Systems*, (25):39–66, 2003.

[17] T. M. Ghazalie and T. P. Baker. Aperiodic servers in a deadline scheduling environment. *Real-Time Syst.*, 9(1):31–67, 1995.

[18] J. Goossens and C. Macq. Performance analysis of various scheduling algorithms for real-time systems composed of aperiodic and periodic tasks. In *CISSAS'99*, 1999.

[19] IEEE. *1003.1, 2004 EDITION IEEE Standard for Information Technology Portable Operating System Interface (POSIX)*. 2004.

[20] J. Lehoczky and S. Ramos-Thuel. An optimal algorithm for scheduling soft-aperiodic tasks fixed-priority preemptive systems. In *Proc. IEEE Real-Time Systems Symposium*, pages 110–123, 1992.

[21] J. Lehoczky, L. Sha, and Jay K. Strosnider. Enhanced aperiodic responsiveness in hard real-time environments. In *Proc. IEEE Real-Time Systems Symposium*, pages 261–270, 1987.

[22] Stefan M. Petters. Execution-time profiles. Technical report, NICTA, Sydney, Australia, Jan 2007.

[23] S. Ramos-Thuel and J. P. Lehoczky. On-line scheduling of hard deadline aperiodic tasks in fixed priority systems. In *14th IEEE Real-Time Systems Symposium*, pages 160–171, 1993.

[24] S. Ramos-Thuel and J. P. Lehoczky. Algorithms for scheduling hard aperiodic tasks in fixed priority systems using slack stealing. In *15th IEEE Real-Time Systems Symposium*, pages 22–35, 1994.

[25] B. Sprunt, J. Lehoczky, and L. Sha. Exploiting unused periodic time for aperiodic service using the extended priority exchange algorithm. In *Proceedings of the Real-Time Systems Symposium*, pages 251–258, 1988.

[26] M. Stanovich, T.P. Baker, An-I Wang, and M.G. Harbour. Defects of the posix sporadic server and how to correct them. In *16th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 35–45, 2010.

[27] J. Strosnider, J. P. Lehoczky, and L. Sha. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. *IEEE Trans. on Computers*, 1(44):73–91, 1995.

[28] A. Terrasa, A. Espinosa, and A. García-Fornes. Lightweight POSIX Tracing. *Software Practice and Experience*, 38(5):447–469, 2008.

[29] Too-Seng Tia, Jane W.-S. Liu, and M. Shankar. Algorithms and optimality of scheduling soft aperiodic requests in fixed-priority preemptive systems. *Real-Time Syst.*, 10(1):23–43, 1996.

[30] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem – overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7:36:1–36:53, 2008.

[31] V. Yodaiken. The RTLinux manifesto. In *Proc. of The 5th Linux Expo, Raleigh, NC*, March 1999.

Luis Búrdalo studied computer science in the Universitat Politècnica de València (UPV), in Spain, where he received his B.S. degree in 2004. Currently, he works as a researcher at the Department of Information Systems and Computation (DSIC) of the UPV. Also, he is a member of the Group of Information Technology/Artificial Intelligence (GTI/IA) research group. His research interests mainly include real-time systems, real-time artificial intelligence, and multi-agent systems.

Andrés Terrasa studied computer science in the Universitat Politècnica de València (UPV), in Spain, where he received his B.S. degree in 1995, and his Ph.D. degree in 2001. He currently works as an Associate Professor at the Department of Information Systems and Computation (DSIC) of the UPV. Also, he is a member of the Group of Information Technology/Artificial Intelligence (GTI/IA) research group. His research interests mainly include real-time systems, real-time artificial intelligence, and multi-agent systems.

Agustín Espinosa studied computer science in the Universitat Politècnica de València (UPV), in Spain, where he received his B.S. degree in 1988, and his Ph.D. degree in 2003. He currently works as an Associate Professor at the Department of Information Systems and Computation (DSIC) of the UPV. Also, he is a member of the Group of Information Technology/Artificial Intelligence (GTI/IA) research group. His research interests mainly include real-time systems, real-time artificial intelligence, and multi-agent systems.

Ana García-Fornes received her B.S.(1986) and Ph.D.(1996) degrees in Computer Science from the Polytechnic University of Catalonia, Spain, and the Universitat Politècnica de València, Spain, respectively. She currently works as an Associate Professor at the Department of Information Systems and Computation at the Universitat Politcnica de Valencia, Spain. Her research interests focus on real-time scheduling, real-time operating systems, real-time agent/multi-agent systems, and multi-agent systems platforms.