# Integrating Best-effort Scheduling into a Real-time System

Scott Banachowski, Timothy Bisson and Scott A. Brandt
Computer Science Department, University of California, Santa Cruz
{sbanacho,tbisson,sbrandt}@cs.ucsc.edu

## Abstract

*Demand for real-time capability in general-purpose systems is rising and as systems are retrofitted with new scheduling features they become increasingly complex. To counter this trend we present the best-effort bandwidth server (BEBS), an aperiodic server for flexible and efficient support of best-effort applications in a real-time system. Recognizing that the responsiveness of a server depends on its period, and that not every best-effort task requires equal responsiveness, the algorithm adjusts its period based on run-time behavior of tasks. We created a prototype implementation of the system to demonstrate that it performs suitably as a general-purpose scheduler in comparison to Linux, and outperforms a common type of hierarchy used in existing general-purpose systems. The result is a system that integrates real-time scheduling with best-effort support, both simple and powerful enough to be used as the only scheduler in a general-purpose operating system.*

## 1. Introduction

Modern computer systems are expected to simultaneously execute a variety of applications, many with different types of timing requirements. For example, the proliferation of multimedia on desktop computers means that systems should support soft real-time applications in addition to general-purpose workloads. In many cases, even though media applications have deadlines, they run on best-effort systems that are unaware of their timing constraints. Other applications with similar soft real-time constraints include virtual reality simulators, virtual machines, games, soft modems, and hardware controllers.

Real-time systems are usually considered specialized because they are designed specifically to guarantee time constraints. Therefore real-time schedulers are not usually considered for executing best-effort workloads, which lack strict timing constraints. Nevertheless, we wish to design flexible systems that execute mixed time constraints: hard, soft and best-effort. While real-time systems handle hard and some soft real-time applications, they often treat best-effort applications in *ad hoc* fashion. This paper describes a system in which best-effort, time-share scheduling policies are integrated into a real-time scheduler, allowing tasks to coincide in the system without the need for complexities like partitioning or scheduling hierarchies.

In order to add real-time scheduling capability to a general-purpose system we do the opposite: start with a real-time scheduling algorithm (EDF) and add best-effort scheduling capability. Real-time scheduling algorithms are capable of scheduling applications without time constraints as long as conditions for guarantees are not violated [3]. Aperiodic servers are one way to run non-periodic tasks in EDF. When applied to general-purpose workloads, most aperiodic server approaches assume that best-effort tasks may be batch processed, and therefore make no distinction between interactive and compute-bound tasks. This conflicts with a long-standing design principle of time-share systems that gives interactive tasks increased priority for better responsiveness [13]. By combining a time-share strategy with an aperiodic server, the resulting system is, from a general-purpose user's standpoint, indistinguishable from a time-share scheduler, but also capable of meeting periodic deadline constraints for real-time applications.

We developed an aperiodic server called the Best-Effort Bandwidth Server (BEBS) to meet time-sharing goals. A major difference between BEBS and other aperiodic servers is that it dynamically adjusts periods as applications execute so that they perform similarly to other time-share schedulers. This unified approach provides general-purpose scheduling without sacrificing real-time capability, and has several advantages. It is simpler than many hierarchical systems, where several scheduling algorithms and their interactions at different levels must be understood in order to predict system behavior. Best-effort applications may be scheduled with better responsiveness than approaches that treat them as background tasks. And because it uses EDF, other EDF-based real-time scheduling algorithms may easily be integrated into the system. In this paper we will also show how BEBS provides better performance to periodic best-effort tasks, such as legacy multimedia applications, without any *a priori* knowledge about their processing characteristics.

This research demonstrates that using a strict real-time scheduler as the single, core scheduler for all tasks in an

open, general-purpose operating system is not only feasible, it is desirable. It offers performance similar to existing general purpose schedulers with the additional advantage of native EDF real-time scheduling. We have built a prototype implementation of the scheduler that efficiently and effectively schedules general-purpose workloads, automatically detects and schedules soft real-time workloads, and natively supports hard real-time workloads.

## 2. Related work

A predecessor to this research is the Rate-Based Earliest Deadline scheduler (RBED) [3]. The principle of RBED's operation is that resource allocation decisions are made distinctly from dispatching decisions. Jobs are dispatched using an EDF scheduler, while a resource allocation algorithm ensures that tasks meet specific performance criteria. RBED meets hard and soft real-time requirements by dynamically controlling the rate that applications consume CPU (by adjusting either the period or execution time). Rates changes occur within rules derived from a generalized periodic task model, such that an EDF schedule remains feasible. Although developed independently, the RBED model is similar to the Variable Rate Execution Model (VRE) [9], which also defines rules for changing application rates at any time. The elastic task model [5] similarly defines a method for sets of tasks to simultaneously change rate, using the novel approach of modeling task utilization as elastic springs; in contrast, RBED allows a single task to change rate independent of other tasks. BEBS expands RBED by integrating more robust and efficient best-effort support.

Because general-purpose systems do not provide guarantees or predictability, they are not typically used for real-time applications. However, there has been effort to add real-time capability. A common approach uses separate schedulers for different types of applications. The POSIX API [12] specifies fixed-priorities for real-time tasks, with other tasks in a lower priority scheduler. However performance degrades or fails when tasks misbehave [19]. RTLinux [25] takes this approach further, running Linux as a preemptable low-priority task in a small real-time executive. A downside of these approaches is that best-effort scheduling occurs in the slack of real-time programs (turning best-effort workload into background tasks with possibly poor response time). Linux-SRT [7] adds another scheduling class to the existing POSIX classes—we wish to instead treat each application the same when making scheduling decisions.

The new Linux 2.6 kernel [14] improved its scheduling performance by reducing the amount of non-interruptible code, allowing the kernel to be preempted, and improving scalability. Time-sensitive Linux (TSL) [10] and Real-time Enhanced (RTE) Linux [24] both add high-precision timing facilities so that scheduling allocations do not depend on coarse-grain periodic clocks. With all this emphasis on improving the real-time performance of modern operating systems, we pose the question: "Why not use a real-time scheduling algorithm as the only task scheduler?"

To simultaneously support mixes of applications, more flexible hierarchical approaches use dynamic schedulers to multiplex the CPU between schedulers, supporting multiple scheduling paradigms simultaneously [11, 8, 6, 15]. Although challenging, it is possible to analyze a system with multiple tiers of schedulers [20]. Nevertheless, hierarchical scheduling poses many engineering difficulties, and ultimately no matter how intricate the tree of schedulers, results in a one-dimensional schedule. Also, schedulers providing the weakest guarantees must be at lower levels, meaning that best-effort tasks are relegated to the lowest levels of the hierarchy, receiving only slack processing time. We chose to develop our scheduler without the extra complexity of multiple interacting schedulers.

The BEBS algorithm is similar to IRIS [18], which is based on the Constant Bandwidth Server (CBS) [1]. CBS is designed to provide CPU bandwidth reservations to continuous media applications. In our implementation, we model best-effort applications as aperiodic tasks so that we can serve them with a similar approach. IRIS enhances CBS with a fairer slack reclaiming strategy, and BEBS is similar to IRIS. However, BEBS differs from these and other aperiodic servers [16, 22] in that BEBS adapts its period and utilization according to the best-effort workload it is presented. We implemented our prototype system in Linux by developing an EDF scheduler with overrun protection (similar to R-EDF [26]).

## 3. General approach

Our scheduler uses the traditional periodic task model [17]: tasks consist of a series of sequential periodically-released jobs, and each job must complete before its deadline, which coincides with the release of the next job.[1] We use the earliest deadline first (EDF) scheduling algorithm because it is optimal for this workload, allows full processor utilization [17], and is not complicated to implement. We believe, as was recently argued [4], that although EDF introduces the overhead of dynamic priorities, its performance relative to a static priority system is better in terms of utilization and responsiveness.

Using EDF, the system naturally supports periodic real-time tasks; our implementation provides a system call and admission control for users to submit real-time jobs they wish to be scheduled. However, for general-purpose use,

[1] Note that making the deadline equal to the period is not a requirement, but a simplification.

much of the workload may not have specific periodic time constraints. Therefore, by default users do not specify any period or utilization parameters for these tasks, and the system transparently converts them to periodic tasks.

In our model, each non-real-time task is handled by an aperiodic server, an abstraction that converts tasks that are not necessarily periodic into a stream of periodic jobs schedulable by EDF. For tasks without deadlines, the goal of an aperiodic server is to minimize the service response time [23]. We distinguish between three different kinds of best-effort tasks:

- *Compute-bound tasks* spend most of their time ready to execute, with very little idle time.
- *Interactive tasks* spend much of their time blocked, waiting for input from users or devices.
- *Periodic tasks* have soft real-time periodic workloads, but are submitted as best-effort tasks, with their constraints unknown to the scheduler.

Best-effort systems do not make guarantees about the timing of resource delivery because such a system is not aware of timing constraints. However there are some assumptions about the responsiveness for different kinds of best-effort tasks: CPU bandwidth should be allocated fairly over the long-term, while interactive tasks should receive higher priority in the short term. This policy ensures that interactive tasks remain responsive, even in the presence of compute-intensive tasks. Our scheduler is designed to meet these goals, with the additional goal that it serves periodic best-effort applications so that they meet periodic deadlines.

In the literature, there is little indication of how to set up aperiodic servers for different kinds of applications. Common practice indicates that for a periodic task, the server's period should match the task's. For the unpredictable workload in a general-purpose scheduler, we use a heuristic to determine appropriate server parameters.

The responsiveness of a task running under a server is a function of the server period, because the period determines the granularity at which CPU bandwidth is allocated. For example, with a small period tasks receive small allocations of CPU frequently, while with a large period they receive CPU less frequently but in larger portions. For interactive tasks and those with time constraints, shorter periods generally provide faster responsiveness. If there were no scheduling overhead, periods could be set arbitrarily low, but there is another trade-off: the smaller the period, the more likely an increase in context switches and associated scheduler overhead.

This gives us the following heuristics:

- *Compute-bound tasks* should use large periods, because responsiveness is less critical, and increasing the period reduces overhead.
- *Interactive tasks* should use shorter periods, for better responsiveness.
- *Periodic tasks* should have a server period equal to their application period.

Managing periodic applications is not normally an explicit goal of general-purpose schedulers. Traditional time-share algorithms are not designed for periodic deadline processing and the performance of periodic best-effort applications degrades in the presence of other applications due to scheduling latency [2]. Because BEBS uses a real-time scheduling algorithm, it provides better responsiveness to workloads containing periodic deadlines with appropriate parameter settings. This means periodic tasks that do not specify their constraints to the system (such as legacy desktop multimedia programs) still realize the benefits of running in a real-time scheduler.

# 4. Supporting best-effort tasks

The Best-Effort Bandwidth Server (BEBS) sets the period and utilization parameters for best-effort tasks so that they do not violate any real-time constraints of other tasks, while providing the performance expected from a typical time-share scheduler. The novelty of this approach is that all tasks, real-time or not, execute together in the same scheduling algorithm. BEBS is composed of three components: a basic aperiodic server, a slack reclaiming procedure, and an algorithm for determining appropriate server parameters. The following subsections describe each component separately.

## 4.1. The aperiodic server algorithm

First we describe the aperiodic server. Each task is assigned a single server, and each server $i$ has a budget $b_i$ and period $p_i$. For now we assume that $b_i$ and $p_i$ are fixed, even though they will be changed on-the-fly, as explained in Section 4.3. The budget is the amount of time a server may execute per period, so utilization is $u_i = b_i/p_i$. The server's dynamic budget $c_i$ decreases as it executes; it represents the budget remaining in the current period. There are three states for a task: *executing*, *expired* or *blocked*. The rules for each server are:

1. A start of a period is called the release time $r_i$, at which time the server's dynamic budget $c_i$ is set $c_i = b_i$, its deadline is $d_i = r_i + p_i$, and its state is set to *executing*.
2. As a task executes, its server's budget is decremented by its execution time. If a task voluntarily sleeps before the budget expires, the state is set to *blocked*.
3. When $c_i = 0$, the task is *expired,* and suspended until its next release time $r_i = r_i + p_i$, at which time step 1 is repeated.

4. When a *blocked* job wakes at time $t$, the state is set to *executing*, and if $t \geq (r_i + p_i)$ or $(b_i - c_i) \leq (t - r_i)u_i$ then $r_i = t$, $d_i = r_i + p_i$ and $c_i = b_i$, otherwise resume with its current deadline and budget.

The algorithm is similar to CBS [1] (specifically, Step 4 is the same as CBS). The difference is that CBS does not include an *expired* state; in CBS when a budget expires it is immediately recharged, with the deadline incremented. In CBS, it is possible for the deadline to advance unbounded, which is potentially unfair to some workloads.

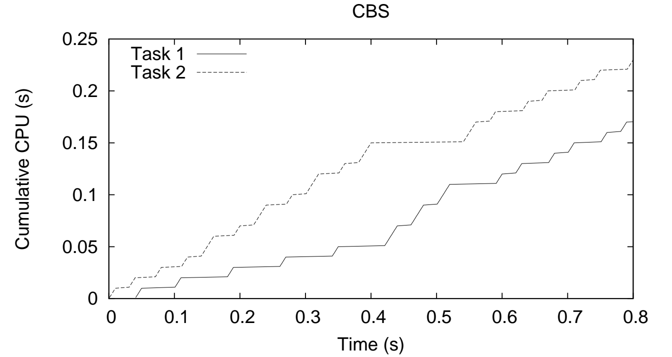
### 4.2. Slack reclamation

When a process does not consume all the CPU bandwidth it is allocated, we say that *dynamic slack* is added to the schedule. To fully utilize the processor, dynamic slack must be used by tasks that want to use it.

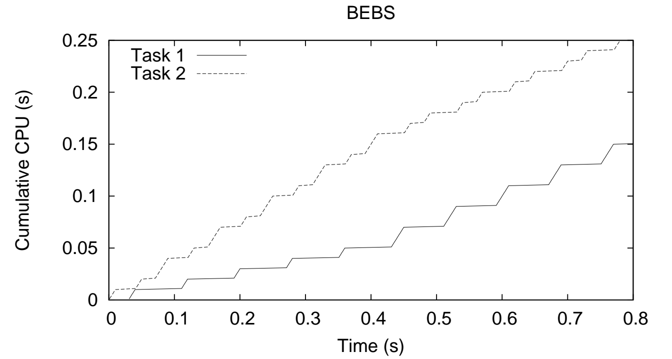
For dynamic slack reclamation, BEBS uses a method derived from the IRIS scheduler [18], although they differ in the way deadlines are set. When the system will idle because there is no task eligible to run, the next best-effort task to be released from the *expired* state is released immediately. To preserve fairness among other expired tasks, the release times of all other expired best-effort tasks are advanced by the same amount as the early-released task. The rule may be summarized as:

1. When a task reaches *expired* state, cache it's next scheduled release time $r_i$ in $r_{orig}$.
2. If the CPU idles at time $t$ and at least one task is in the *expired* state, find the *expired* task with earliest release $r_i$, and call this time $r_e$. For all *expired* tasks change their next release time to $r_i = r_i - (r_e - t)$. Note that at least one task will have a new $r_i = t$, and so is eligible to run.
3. When tasks with $r_i < r_{orig}$ are set to *executing*, set their respective budgets and deadlines to $c_i = b_i$ and $d_i = r_{orig} + p_i$, and set their next release to $r_i = t + p_i$, where $t$ is the current time.

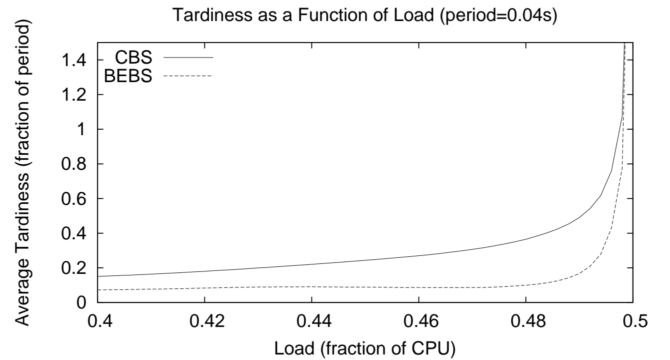
In CBS it possible for a task's deadline to exceed the current time by an unbounded amount, and this effect is pronounced when a task consumes slack, because as long as a task remains executing, its deadline advances. This has two negative effects: a task is penalized by reduced utilization in the future, and it is possible for it to temporarily starve. Suppose a compute-bound task runs during slack—when the slack goes away, its budget is always $\leq b$, but if its current deadline is far ahead, not only is its utilization in the current period diminished, but it may potentially wait a long time before becoming the earliest deadline. In other words, a task is penalized in the future for using slack. We do not wish to penalize tasks for using processing time that is essentially "free."




(a) Under CBS as Task 1 increases its CPU use, Task 2 starves.




(b) The same two tasks under BEBS without starvation.




(c) Tardiness of CBS and BEBS as the aperiodic share of bandwidth becomes saturated.

**Figure 1. A comparison of CBS and BEBS**

Figure 1(a) demonstrates the problem with CPU using two tasks assigned equal allocations of CPU bandwidth. Initially Task 1 does not use all of its bandwidth, and Task 2 consumes the slack. In CBS, as Task 1 begins to require more CPU, it temporarily starves Task 2 for an interval beginning at 0.4 seconds. This problem does not exist in BEBS (or IRIS). Figure 1(b) shows the same tasks served instead by the BEBS algorithm. As Task 1 demands its full share of CPU it receives it, without causing Task 2 to starve.

BEBS slack reclamation leads to better performance than CBS. In the experiment of Figure 1(c), a set of hard real-time tasks consumed half the processor bandwidth, while a soft periodic task that computes periodic, variable-time frames (like a multimedia application) ran in BEBS or CBS. The task should finish processing each frame at a constant rate, so it has periodic *frame deadlines* (which are independent of its server's deadlines). We measure the average tardiness as the load of the soft application saturates its available bandwidth. Tardiness is measured as the difference between the actual completion time and the frame deadline for late frames, or zero for on-time frames. We see that at less than full saturation, BEBS outperforms CBS, resulting in less than half the average tardiness. The reason is that because each frame is variable length, some take longer and require slack processing in order to complete. In BEBS, a task is not penalized for using slack as it is in CBS.

BEBS introduces an optimization for interactive tasks that differs from IRIS in Step 3. In IRIS, when a task is released earlier than its originally scheduled release, the next deadline is set to $d_i = r_i + p_i$. In BEBS, it is set to $d_i = r_{orig} + p_i$, which is exactly the deadline it would have received if it was not released early ($r_{orig} \geq r_i$). In IRIS, a task receiving slack is released with higher EDF priority, making it unfair to tasks that voluntarily block. Consider the following example with three tasks, each with $b = 10$ ms and $p = 30$ ms, depicted in Figure 2(a). At time 5 ms, Task 1 voluntarily blocks. At 25 ms, there is slack, so IRIS releases the two expired tasks, each with deadline $d = 25 + 30 = 55$. When Task 1 unblocks at 26 ms, its new deadline is set to $d = 26 + 30 = 56$. Therefore, it must wait to execute, even though it voluntarily released the CPU, because it has the latest deadline.

In keeping with time-share principles, we prefer that voluntarily blocking tasks remain responsive. Figure 2(b) shows the same scenario in the BEBS scheduler. When the tasks are released early at time 25 ms, their deadlines are set to $d = 30 + 30 = 60$. When Task 1 unblocks, it is assigned the earliest deadline ($d = 56$) and runs immediately. Note that in both algorithms the resources consumed and the subsequent release times of all tasks are equal.

Unlike CBS, in which deadlines may differ from the current time by an unbounded amount, BEBS bounds the difference between the actual time and the deadline by at most $2p_i - b_i$ (in IRIS, the bound is $p_i$). This prevents a task from starving for resources too long.

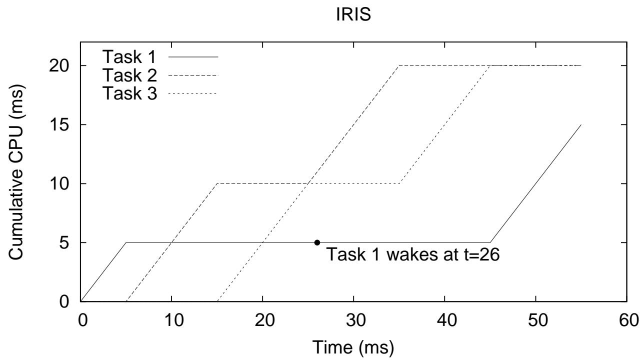
### 4.3. Setting server parameters

Period adaptation is an important aspect of handling workloads in which application properties are unknown *a priori*. For general-purpose workloads, it is not expected that users will want, or even know how, to properly assign server parameters for the applications they run. Nor is it incumbent upon application programmers to build such knowledge into their applications. In order to appropriately serve best-effort applications, BEBS observes applications as they execute and adjusts their server periods dynamically.

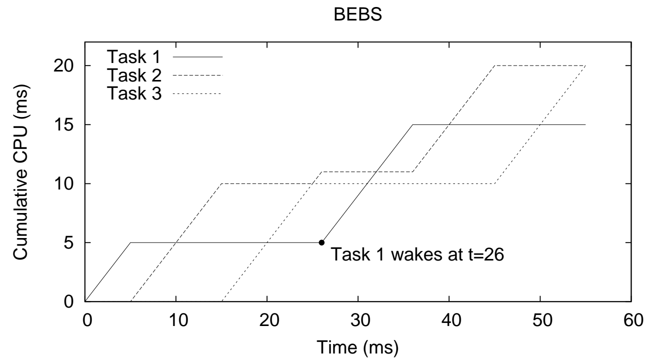
To test our assertion (in Section 3) that the period of the server impacts both the application response time and the scheduling overhead, we measured both the tardiness of a soft-real time task and the number of context switches as a function of server period. The workload is the same as in Figure 1(c) but we varied the period of the server while using a constant load. In Figure 3(a) we see that tardiness is generally better at lower periods (and as before BEBS outperformed CBS). We also see in Figure 3(b) that lower periods incur more scheduling overhead due to an increased number of context switches. (BEBS incurs more scheduling overhead than CBS because as CBS replenishes budgets of expired tasks it does not always need to preempt them, whereas BEBS always preempts expired tasks and waits until idle time before replenishing their budgets). Therefore, we wish to reduce overhead by using larger server periods for CPU-bound tasks that do not need fast response times, while using shorter periods for the interactive workload for better performance.

Period adaptation is a major difference between BEBS and other aperiodic servers. BEBS servers adjust their period to best suit best-effort workloads, and to provide the same performance expected from time-share scheduling algorithms. The algorithm adjusts periods by observing past execution rates, and inferring whether tasks are interactive, compute-bound, or soft real-time periodic.

Before describing the period adaption algorithm, we will motivate it with an example. Figure 4 shows what happens when we run 5 best-effort tasks in the BEBS server along with 5 real-time applications. Two of the best-effort tasks have soft periodic deadlines (with frame rates 25 and 12.5 frames/sec) that they should be able to meet within their share of CPU, and the other three are compute-bound. In the first run, the aperiodic server periods are not set appropriately. They are all set to the period of the lowest period application— the result is that the soft-real time tasks miss their deadlines, while the scheduler incurs more context switches. In the second run, the server period for all tasks is increased, resulting in less overhead but even worse real-time performance. In the third run, server periods are

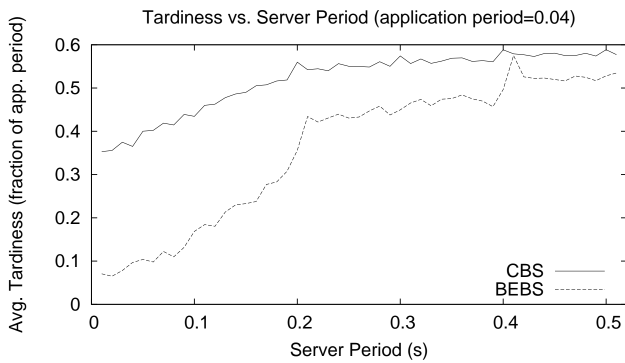



(a) Task 1 suffers poor response time after voluntarily blocking.

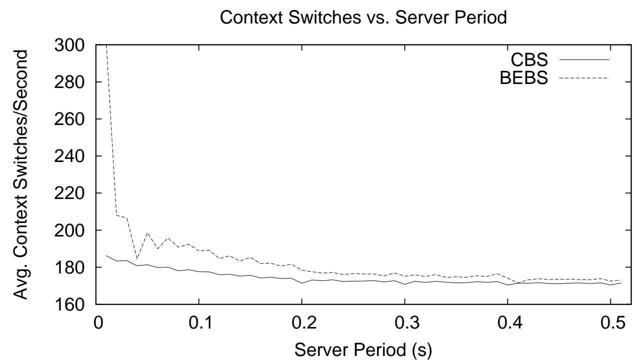



(b) After a task blocks it is more likely to be serviced quickly.

**Figure 2. A comparison of the IRIS and BEBS slack reclamation algorithms**




(a) The average tardiness of a periodic soft-real time application.




(b) The average number of context switches.

**Figure 3. Server performance as a function of server period. The server period of the application is constant.**

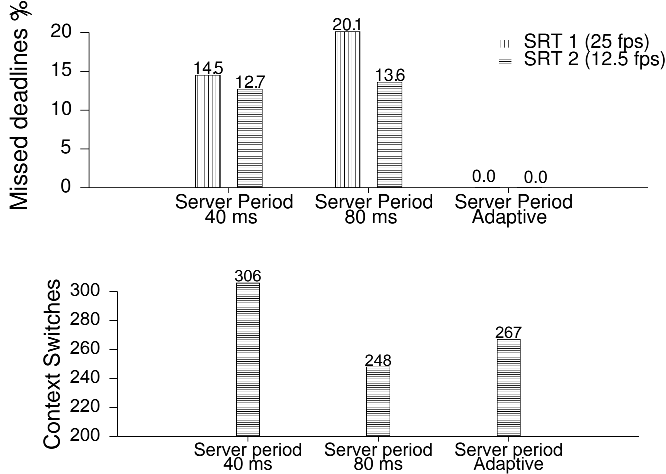



**Figure 4. Comparison of different fixed server periods vs. adaptive periods. When using fixed server periods, we miss more deadlines. The adaptive approach balances scheduler overhead for better performance in terms of missed deadlines.**

adjusted using the algorithm described in this section. The result is no deadline misses, with a balanced scheduling overhead.

BEBS assigns utilization to servers consistent with the fair-weight policy of time-share schedulers (in most UNIX-type schedulers, this is adjusted with the user-control *nice*). Each task has a priority $q$ based on the value of its *nice* priority. The utilization for each task is set to $u = \frac{q}{L} U_{BE}$, where the load $L = \sum q_i$ is the sum of all best-effort tasks' quanta, and $U_{BE}$ is the utilization available to best-effort tasks.

The period of each server is adjusted according to how its task executed in the past—in this way it distinguishes between interactive and compute-bound processes. For each process, the algorithm is:

1. Each time a task $i$ goes from state *executing* to either *blocked* or *expired*, sample the amount of CPU consumed $e_i$ since the last time the task entered *executing* state.
2. Average the current sample with the past average (using a weight $w$ for hysteresis): $e_{i,avg} = \frac{w e_{i,avg} + e_i}{w+1}$. In our prototype, $w = 3$ (making the divisor a bit-shift operation).
3. Set the budget so in the next period, the task consumes $b_i = \min(e_{i,avg} + C, e_{i,max})$. The value $C$ is added to the estimated duration to compensate for the fact that because $e_{i,avg}$ is an average, some durations may be more than the average. In our implementation, $C$ is set to $e_{i,avg}/2$. The upper bound on budget $e_{max}$ is set to 200 ms (equal to the maximum time-slice of Linux).
4. Set the appropriate period for a best-effort task, $p_i = \frac{b_i}{u_i}$.

Note that because each task's utilization based on weight, when tasks enter and leave the system, it may change. The following Section 4.4 discusses when these changes take place.

The BEBS algorithm automatically distinguishes between interactive and compute-bound tasks through its measure of past execution times, so the parameter assignments give each task the allocation it needs to preserve time-share goals. For example, a compute-bound process will have constant budget $e_{avg} = e_{max}$, and so period is a function of the system load $L$; as the load of the system increases, compute-bound tasks periods expand so that they run less frequently, but receive long budgets. In contrast, for an interactive bound process $e_{avg}$ depends on the rate and duration of CPU bursts, and will generally produce a short period so it is responsive relative to compute-bound tasks.

There is an additional benefit to this algorithm: if a periodic best-effort task meets its previous deadlines within its fair share of CPU, then its period is set so that it will likely meet future deadlines. Assume that a periodic task has an average runtime per job of $a$. If the task is to meet deadlines within its fair share, then $a/p_{actual} \leq u$, where $p_{actual}$ is the actual period unknown to the scheduler. As the task runs, it will block, on average, after consuming $a$ units of CPU, measured by the scheduler as $e_{avg} \approx a$. The server period is set to $p = e_{avg}/u$, which means that $p \leq p_{actual}$. The resulting deadline ensures that, on average, the task receives its processing on time. Because $a$ is an average, some frames will be expected to overrun their deadlines, nevertheless we previously showed that this technique provides much better best-effort support for periodic and multimedia applications [2].

## 4.4. Overhead discussion

The system combines an EDF scheduler with an adaptive aperiodic server algorithm. An obvious question is "How much overhead is imposed?" Although dynamic priorities incur more overhead than fixed priorities (used in many real-time systems), the actual overhead is similar to that of other general-purpose schedulers. In this section we consider this practical aspect of BEBS. To verify our approach, we developed an EDF scheduler in Linux, implemented the BEBS algorithm, and evaluated its performance (presented in Sections 5 and 6).

Our system uses a single server for each best-effort process. Maintaining a server requires an extra storage of 17 32-bit words in each process's state structure. A larger overhead is the periodic computation of server parameters, as these values may change due to either the system state or task behavior. These changes incur only simple math operations, as described above, all implemented as integer operations.

The RBED theory [3] allows parameters to be changed (within bounds) at any time, without violating EDF constraints. However in practice, they are calculated only at job releases, and not necessarily at every release; we lazily avoid recomputing them whenever possible. There are three variables affecting server parameters: the active best-effort load $L$ (which changes as best-effort tasks enter, leave or sleep), the best-effort bandwidth $U_{BE}$ (which changes as hard real-time tasks enter or leave), and the task's $e_{avg}$ (a function of execution behavior). The first two variables affect the available bandwidth. When the best-effort bandwidth is reduced, servers sharing this bandwidth reduce their utilization; this occurs aggressively to prevent overload. This also requires some care to prevent temporary overload, so the first instance of a newly introduced task may be delayed briefly if required. However, the inverse case occurs lazily, so that the schedule has some slack when the available bandwidth increases—this means load fluctuations do not always require parameters to be recomputed. Slack reclaiming ensures that even when the entire CPU is not allocated, contending tasks still fairly share the available CPU. Changes triggered by the average execution time of a task also occur with some hysteresis, to prevent recalculating parameters too frequently.

## 5. Prototype implementation

We implemented EDF in the Linux 2.6.8.1 kernel. We chose Linux because its source code is free and it includes device support, file-systems, and a wide assortment of tools and applications. It is important to note that many components, such as the virtual memory system, are not constraint-aware and do not provide the predictability required by hard real-time applications. It is also difficult to account for CPU consumed by system and interrupt processing, which steal cycles from running tasks. Nevertheless, the benefits of using the rich environment provided by Linux outweigh these issues for our proof-of-concept implementation.

To aid in EDF scheduling in Linux, we added a low-overhead high-resolution timer (using the Pentium's APIC timer, similarly to firm timers [10]) so that tasks cannot overrun CPU allocations that are finer than the kernel clock resolution. We also implemented several task queues for efficiently managing tasks of different states. Because the EDF queue is sorted, scheduler selection time is O(1), but queue insertion is linear with the number of active tasks. Section 6 shows that this overhead is not significantly worse than Linux.

By default, processes are best-effort upon creation and managed in EDF by a BEBS server. Real-time tasks may use the EDF scheduler directly; we modified the standard *sched_setscheduler* system trap as an interface for setting a periodic deadline and bandwidth reservation for real-time processes. The system prevents overload by rejecting a request to reserve a bandwidth $u_i$ if $(u_i + U_{RT} + U_{BE,min}) > 1$, where $U_{RT}$ is the total bandwidth reserved by real-time tasks, and $U_{BE,min}$ is the minimum bandwidth set aside for best-effort processing. If the real-time task is admitted, then $U_{RT} = U_{RT} + u_i$ and $U_{BE} = U_{BE} - u_i$, which will trigger a change to best-effort task parameters, as described in Section 4.4.

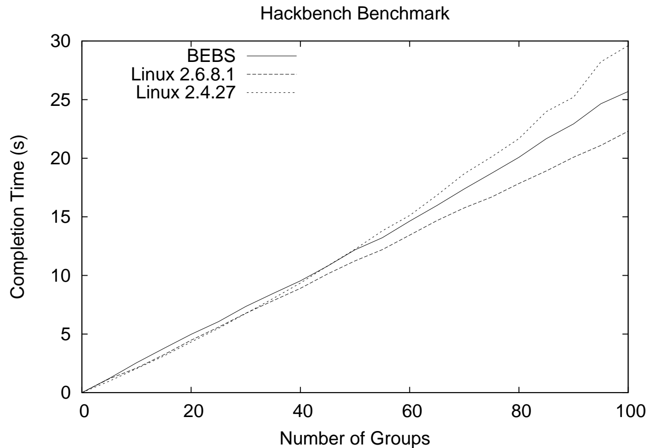



**Figure 5. Hackbench benchmark results. Hackbench provides an estimate of system overhead when comparing different kernels.**

## 6. Performance results

We tested BEBS to verify that it is practical for general-purpose workloads. Our test machine is an Intel P3 1 GHz processor. We used a combination of in-kernel traces and instrumented applications to make our measurements.

It is difficult to appraise best-effort scheduling algorithms because the meaning of "best-effort" is subjective. While a goal of time-share scheduling is to remain responsive, deciding quantitatively how responsive is difficult, especially when it trades-off with other qualities such as overhead and fairness. In this section, we use Linux as the control, so we implicitly assume that Linux provides "good" best-effort service. Our goal is to show that BEBS matches or exceeds Linux's performance. This assumption that Linux provides good performance is based on the fact that it has many users, some of whom must be reasonably satisfied.[2] Whenever appropriate, settable parameters in the BEBS matched those of Linux to make performance similar. For example, the maximum budget per period in BEBS

[2] A recent statistical estimate by the Linux Counter Organization places this number at 18 million users.

equals the maximum time-slice in Linux.

The Hackbench [21] macro-benchmark is typically used to evaluate the performance of Linux kernel development patches. The benchmark creates multiple groups of processes (each group with 20 clients and 20 servers), and measures the time for all groups to complete a series of send operations between clients and servers. As a result, it incurs many context switches. Hackbench is useful for comparing kernels because the workload is the same during each test and so performance differences must be attributed to differences in kernel implementations. Figure 5 compares BEBS to both the new and old versions of Linux. The only difference between BEBS and the Linux 2.6 kernel on which it is implemented is the scheduler, so any performance difference is attributed to scheduling algorithms. We see that BEBS incurs slight overheard compared to Linux 2.6. Both of BEBS and Linux 2.6 may have advantages over the older Linux 2.4, but we conclude that BEBS's overhead is in an acceptable range for general-purpose systems.

Figure 5 demonstrates that the overhead of BEBS with an EDF scheduler is sufficient for our general-purpose use. In keeping with general-purpose scheduling goals, we must also demonstrate that the scheduler is more responsive to interactive tasks than compute-bound tasks. In order to do this, we measure the time it takes different kinds of user tasks to respond to an event.

We created a task that sets the system hardware's real-time clock (RTC) to signal an interrupt in the future. We then measured responsiveness as the time between the occurrence of the hardware interrupt, and the time it takes the user task to discover that the interrupt occurred. We ran this task as either interactive or compute-intensive to compare the performance in the scheduler. By design, interactive tasks should exhibit better response times.

In interactive mode, the task sets the one-shot interrupt for a random time in the future, then does a blocking read on the RTC device. When the interrupt occurs, the task unblocks and is put into the running state to be scheduled. When scheduled, it records the time elapsed since the kernel received the interrupt. It repeats the measurement until the average converges (to 95% confidence of 10% half-width). In compute mode the task does the same, but instead of blocking, it polls the device for the event so that it consumes CPU as it waits for the event. If the task is scheduled when the interrupt occurs, polling discovers the event quickly. However, if the task is not running when the interrupt occurs, it must wait until it is scheduled again to discover it; since it does not have the high priority of an interactive task, it may take longer to discover. This allows us to measure the responsiveness of either interactive or compute-intensive tasks.

Figure 6(a) shows the response time of the compute-bound task as a function of the number of currently running extra tasks. As more tasks run concurrently, the average response time increases because the task must share the processor. The Linux policy for compute-bound tasks is that they share the processor fairly. BEBS performs similarly to Linux, also conforming to the time-share policy.

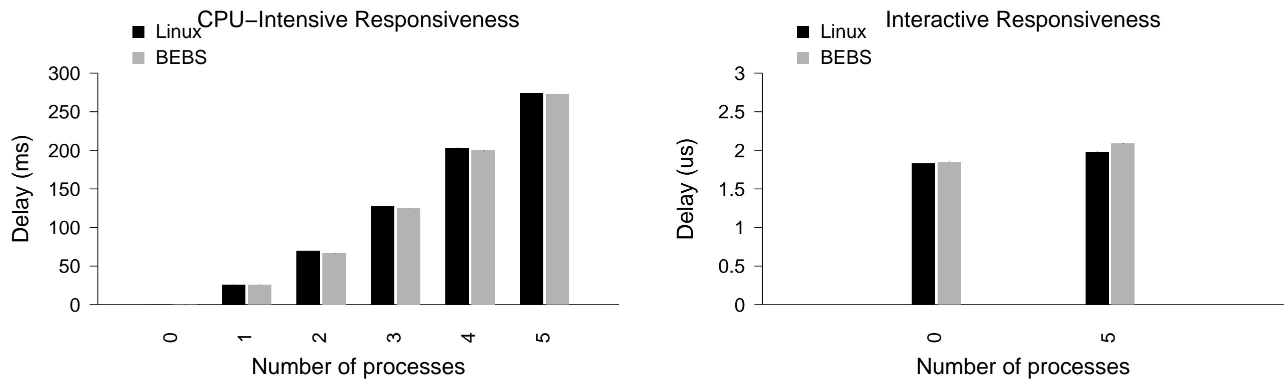
Figure 6(b) show the average response times of the interactive task running with no other tasks, and with 5 extra compute-intensive tasks. In keeping with the time-share principle, the compute-intensive load does not significantly hinder the responsiveness of the interactive task. BEBS performs the same as Linux.

In Figure 7 we repeated the response tests with periodic real-time tasks running in the background. In Linux, real-time tasks have fixed-priority, and always preempt best-effort tasks. In our scheduler, the real-time tasks run under EDF principle, along with the best-effort tasks which use BEBS. In each test, there was a also a compute-intensive task running. Real-time tasks were added with periods 15, 30, 45, and 60 ms, respectively, and each real-time task requires 15% CPU bandwidth. In Figure 7(a) the test task is CPU-intensive, sharing the CPU unused by real-time tasks with the other greedy tasks. In BEBS, the task are much more responsive. In Figure 7(b) the test task is interactive, and in BEBS responds much quicker than Linux, where the task is always preempted by real-time tasks.

It is interesting to note that the previous test is an indication of how any algorithm in which the scheduling of real-time tasks take precedence over non-real-time tasks will perform. RTLinux, POSIX, and many hierarchical schedulers take this approach, and it is our hypothesis that this approach is bad for general-purpose systems.

It is difficult to directly compare the best-effort performance of IRIS to BEBS because they treat non-real-time tasks differently. In IRIS, users must assign non real-time tasks a static bandwidth and reservation granularity. BEBS alleviates this step, letting the run-time behavior of a task determine its bandwidth settings. If the parameters of a task in IRIS are not set correctly, it may perform poorly; therefore it is relatively easy to create a straw-man IRIS that won't work as well as BEBS with some workloads. On the other hand, if IRIS parameters are set appropriately for the task, it should in theory perform similar to BEBS.

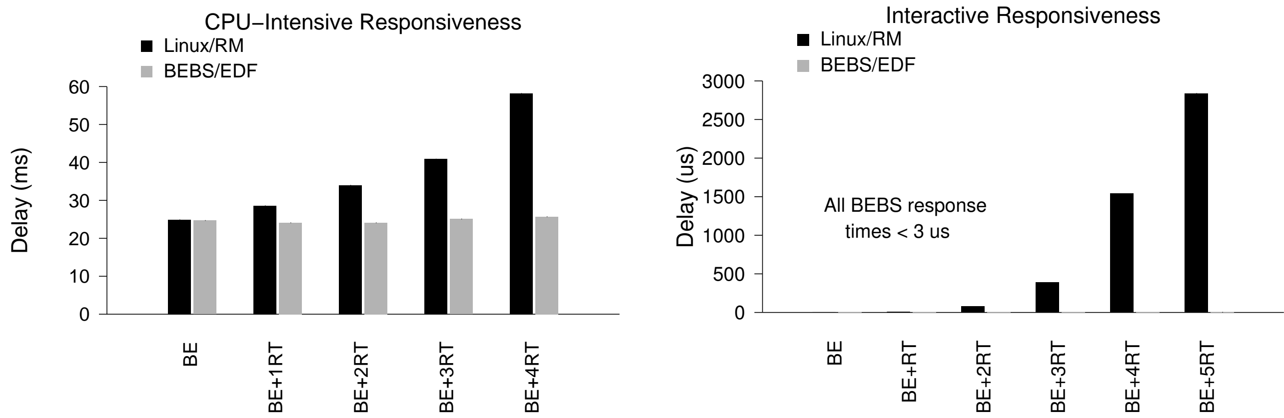
We repeated the responsiveness experiments using a naive implementation of IRIS. We assigned all tasks a static bandwidth and period by dividing the CPU equally and setting all servers to the same period (either 40 ms or 100 ms). In the compute-intensive case shown in Figure 8(a), the IRIS servers were more responsive than BEBS. However, compute-intensive tasks do not need such responsiveness under a time-share discipline. On the other hand, Figure 8(b) shows that the IRIS interactive task performed extremely poorly when competing with compute-bound tasks. This is not the fault of the IRIS algorithm, but of poorly as-




(a) Response time of a compute-intensive program as the number of other tasks running increases.

(b) Response time of an interactive task with no load and heavy load.

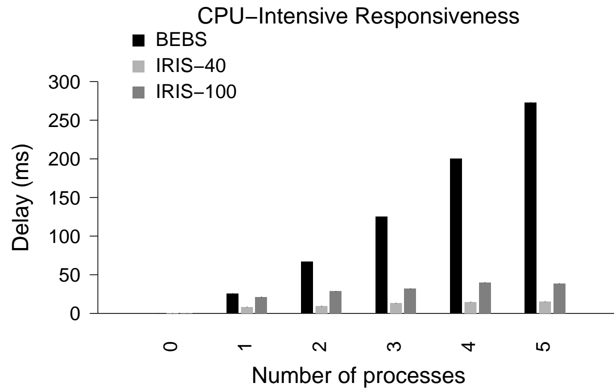
**Figure 6. Average response times of BEBS and Linux**



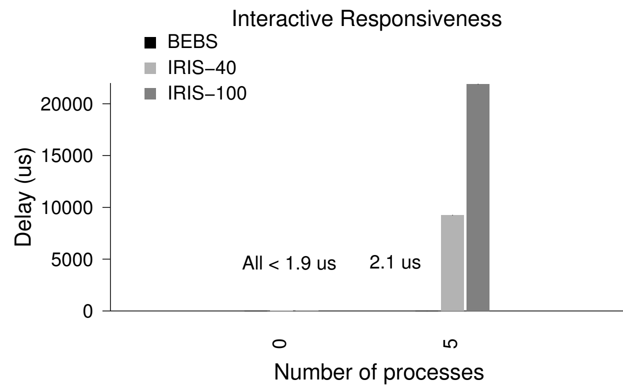

(a) Response time of a compute-intensive task as the number of periodic real-time tasks increase.

(b) Response time of an interactive task as the number of periodic real-time tasks increase.

**Figure 7. Average response times of BEBS and Linux in the presence of real-time workload running in EDF or RM, respectively.**




(a) Response time of a compute-intensive program as the number of other tasks running increases.




(b) Response time of an interactive task with no load and heavy load.

**Figure 8. Average response times for BEBS and IRIS**

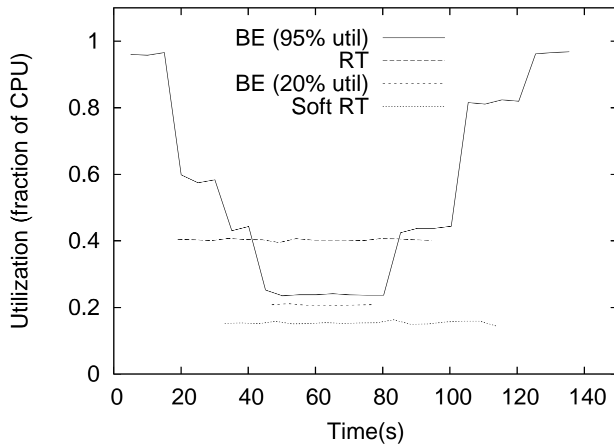



**Figure 9. A view of applications running in the integrated environment.**

signed server parameters. If an aperiodic server approach is applied to all tasks in a general-purpose system, as we propose, it is necessary for the system to automatically set the server parameters, or otherwise it will default to such a naive approach. This is the intuition behind BEBS's operation.

We evaluated BEBS while running a mix of real-time, best-effort periodic, and greedy best-effort processes simultaneously. Figure 9 shows the utilization of tasks over time. The real-time task, which enters at time 20, receives isolation from other tasks, maintaining constant utilization in accordance with its reservation of 40%. The best-effort tasks share the available processing, relative to how greedily they consume CPU. The periodic soft real-time task, which joins the system at time 30, consistently requires less than its share of CPU and it meets all its deadlines.

# 7. Conclusion

Real-time support in best-effort systems is generally cumbersome and ad hoc, often involving hierarchical schedulers, and generally yields poor best-effort performance, especially while real-time processes are executing. We present BEBS, an aperiodic server with a strong focus on best-effort performance, that integrates with a real-time scheduler. Performance in BEBS is shown to be comparable or better to standard best-effort schedulers such as Linux, and more appropriate at meeting time-share design goals than CBS and IRIS. Furthermore, BEBS also provides integral support for soft real-time processes, yielding SRT performance much better than that of typical best-effort schedulers without any *a priori* knowledge about process timing or resource usage requirements. Taken together, these capabilities provide a powerful argument for using a real-time scheduler as the heart of a best-effort system.

This system is part of an ongoing project to create scheduling frameworks that simultaneously handle a wide variety of different timing constraints. There is much future work: we would like provide native support for multi-threaded applications, include SMP processing, develop middleware support for other classes of applications, and add real-time support for the file, network, and memory systems.

**Acknowledgments** This research was funded in part by a DOE High-Performance Computer Science Fellowship, Intel, and the National Science Foundation. Thanks to the

review committee and Pau Martí for their meticulous comments and suggestions.

## References

[1] L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS 1998)*, pages 4–13, Dec. 1998.

[2] S. Banachowski and S. Brandt. Better real-time response for time-share scheduling. In *Proceedings of the Eleventh International Workshop on Parallel & Distributed Real-Time Systems (WPDRTS)*, Apr. 2003.

[3] S. A. Brandt, S. Banachowski, C. Lin, and T. Bisson. Dynamic integrated scheduling of hard real-time, soft real-time and non-real-time processes. In *Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS 2003)*, pages 396–407, Dec. 2003.

[4] G. C. Buttazzo. Rate monotonic vs. EDF: Judgement day. In *Embedded Systems*, volume 2855 of *Lecture Notes in Computer Science*, pages 67–83. Springer-Verlag, Sept. 2003.

[5] G. C. Buttazzo, G. Lipari, M. Caccamo, and L. Abeni. Elastic scheduling for flexible workload management. *IEEE Transactions on Computers*, 51(3):289–302, Mar. 2002.

[6] G. M. Candea and M. B. Jones. Vassal: Loadable scheduler support for multi-policy scheduling. In *Proceedings of the 2nd USENIX Windows NT Symposium*, pages 157–166, Aug. 1998.

[7] S. Childs and D. Ingram. The Linux-SRT integrated multimedia operating system: Bringing QoS to the desktop. In *Proceedings of the Real-Time Technology and Applications Symposium (RTAS01)*, May 2001.

[8] Z. Deng and J. W. Liu. Scheduling real-time applications in an open environment. In *Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS 1997)*, Dec. 1997.

[9] S. Goddard and L. Xu. A variable rate execution model. In *Proceedings of the 16th Euromicro Conference on Real-Time Systems*, pages 135–143, July 2004.

[10] A. Goel, L. Abeni, C. Krasic, J. Snow, and J. Walpole. Supporting time-sensitive applications on general-purpose operating systems. In *Proceedings of the 5rd Symposium on Operating Systems Design and Implementation (OSDI'02)*, Dec. 2002.

[11] P. Goyal, X. Guo, and H. M. Vin. A hierarchical CPU scheduler for multimedia operating systems. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI'96)*, Oct. 1996.

[12] The Institute of Electrical and Electronics Engineers. *IEEE Standard for Information Technology-Portable Operating System Interface (POSIX)-Part 1: System Application Programming Interface (API)-Amendment 1: Realtime Extension [C Language]*, Std1003.1b-1993 edition, 1994.

[13] J. Kay and P. Lauder. A fair share scheduler. *Communications of the ACM*, 31(1):44–55, Jan. 1988.

[14] The Linux kernel archives. http://www.kernel.org, Jan. 2004. A web site with the latest Linux kernel and information.

[15] G. Lipari and S. K. Baruah. Efficient scheduling of real-time multi-task applications in dynamic systems. In *Proceedings of the Real-Time Technology and Applications Symposium (RTAS00)*, pages 166–175, May 2000.

[16] G. Lipari and G. C. Buttazzo. Scheduling real-time multi-task applications in an open system. In *Proceedings of the 11th Euromicro Conference on Real-Time Systems*, June 1999.

[17] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the Association for Computing Machinery*, 20(1):46–61, Jan. 1973.

[18] L. Marzario, G. Lipari, P. Balbastre, and A. Crespo. IRIS: A new reclaiming algorithm for server-based real-time systems. In *10th IEEE Real-time and Embedded Technology and Applications Symposium (RTAS04)*, May 2004.

[19] J. Nieh, J. G. Hanko, J. D. Northcutt, and G. A. Wall. SVR4UNIX scheduler unacceptable for multimedia applications. In *Proceedings of the Fourth International Workshop on Network and Operating System Support for Digital Audio and Video*, 1993.

[20] J. Regehr and J. A. Stankovic. HLS: A framework for composing soft real-time schedulers. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS 2001)*, pages 3–14, London, UK, Dec. 2001. IEEE.

[21] R. Russell. Linux process scheduler improvements in version 2.6.0. http://developer.osdl.org/craiger/hackbench, 2004. A brief description of Hackbench with downloads.

[22] M. Spuri and G. Buttazzo. Scheduling aperiodic tasks in dynamic priority systems. *Real-Time Systems*, 10(2):179–210, 1996.

[23] M. Spuri, G. Buttazzo, and F. Sensini. Robust aperiodic scheduling under dynamic priority systems. In *Proceedings of the 16th IEEE Real-Time Systems Symposium (RTSS 1995)*, Dec. 1995.

[24] Y. Wang and K. Lin. Enhancing the real-time capability of the Linux kernel. In *Proceedings of the Real-Time Computing Systems and Applications*, Oct. 1998.

[25] V. Yodaiken and M. Barabanov. Real-time Linux. In *Proceedings of Linux Applications Development and Deployment Conference (USELINUX)*, Jan. 1997.

[26] W. Yuan, K. Nahrstedt, and K. Kim. R-EDF: A reservation-based EDF scheduling algorithm for multiple multimedia task classes. In *Proceedings of the Real-Time Technology and Applications Symposium (RTAS01)*, May 2001.