Improving Soft Real-Time Performance Through Better Slack Reclaiming

Caixue Lin and Scott A. Brandt
Computer Science Department, University of California, Santa Cruz
{lcx,sbrandt}@cs.ucsc.edu

Abstract

Modern operating systems frequently support applications with a variety of timing constraints including hard real-time, soft real-time, and best-effort. To guarantee performance, critical applications typically over-reserve resources based on worst-case resource usage estimates, while others may reserve based on average-case or other estimates. When resources are fully subscribed, the performance of soft- and non-real-time applications depends upon the effective distribution of dynamic slack—reserved, but unused resources—from other tasks. Motivated by several representative examples, we derive four general principles for the effective management of slack. We have implemented these principles in four progressively better slack schedulers that demonstrate their effectiveness. BACKSLASH, which employs all four principles, misses fewer soft real-time deadlines than all of the other slack schedulers we examined.

1 Introduction

The increasing demand for more powerful computing platforms and applications requires modern operating systems capable of simultaneously supporting applications with a variety of different time constraints. The hierarchical HLS scheduler [16], the flat integrated RBED scheduler [4], the VRE model [9], the BEBS scheduler [2], and the two-level hierarchical scheme [8] are examples. Such systems simultaneously support (1) critical hard real-time applications such as external signal sampling and processing, (2) non-critical soft real-time applications such as desktop multimedia, and (3) best-effort applications such as compilers, word processors, *etc.* In such systems, hard real-time applications typically make worst-case resource reservations in order to guarantee that they meet all of their deadlines, soft real-time applications reserve less than worst-case to achieve a desired level of performance, and best-effort applications make no reservations beyond what is necessary to avoid starvation. Exacerbating the situation, the increasing complexity of modern processors makes it difficult to produce an accurate upper-bound of a task's execution time [7], requiring conservative estimates of resource usage.

Because actual application execution time often varies in data-, time-, or system-dependent ways, applications frequently use less resources than they have reserved, creating dynamic slack—reserved but unused resources[1]. Similarly, all except critical hard real-time applications may occasionally (or even frequently) need more resources than they have reserved. The efficient reclamation and redistribution of dynamic slack to processes whose current needs exceed their reservation can significantly improve the performance of both soft real-time and best-effort applications.

By its very nature, the availability of dynamic slack is unknown beforehand and can only be scheduled dynamically, when it is detected. Similarly, overrun situations are not known until a task has consumed all of the resources that have been reserved for it. The traditional solution is to schedule slack-consuming tasks when all real-time tasks are idle. This guarantees that all hard real-time deadlines will be met. In traditional priority-based real-time systems (such as RT-Linux [19]), this is accomplished by giving all hard real-time tasks higher priorities than all non-hard-real-time tasks. Assuming that the rate monotonic schedulability conditions are met [14], all real-time deadlines will be met and non-real-time tasks will only run, and consume slack, whenever all hard real-time tasks are idle. Modern general-purpose systems such as Linux and others implementing the POSIX standard [10] use similar mechanisms. Some recent systems have begun to address this problem, including CBS [1], CASH [5], GRUB [13], RBED [4], BEBS [2], IRIS [15], and HisReWri [3]. These systems present various ideas that improve the performance of non-hard-real-time tasks.

Based upon a careful study of the ways in which slack can be more effectively used to meet the deadlines of tasks

[1] In this paper we are concerned only with dynamic slack. Static slack—unreserved resources—may be consumed by changing the reservations of one or more applications that can vary their resource usage. Thus, whenever we use the term *slack*, we are referring solely to dynamic slack.

whose resource usage exceeds their reservations, including those in the systems above, we have derived four principles for effective slack scheduling. These principles represent a new understanding of the mechanisms underlying the effective use of dynamic slack and capture the key ideas behind the effective slack management implemented in these systems. Our principles are implemented in four progressively better slack schedulers: SRAND, SLAD, SLASH, and BACKSLASH. These schedulers demonstrate the increasing effectiveness of each of the principles. BACKSLASH, which implements all four, misses fewer soft real-time deadlines than all other slack schedulers we examined.

2 Related work

Many researchers have examined this problem in various contexts and have developed a number of effective techniques for improving slack scheduling. Far from contradicting them, our research borrows from them, distilling out and collecting the key principles that make for effective slack management. Some of the most relevant related projects are discussed below.

Slack stealing algorithms [12, 18] schedule aperiodic or periodic low-priority jobs whenever the execution of high priority jobs may be safely postponed without causing missed deadlines. The main drawbacks are the *a priori* knowledge of execution times that is required and the overhead incurred when computing the available slack during scheduling decisions.

The Constant Bandwidth Server (CBS) [1] immediately releases the next job of a task that has completed its current job. By doing this, a task may borrow against its future budget for current execution. CBS works for both periodic and aperiodic task models, but suffers in that the early-release effectively lowers the priority of slack-consuming tasks, punishing them for consuming slack. IRIS [15] enhances CBS with fairer slack reclaiming. BEBS [2] is similar to IRIS, but designed for time-share applications. In these algorithms, slack is not reclaimed until all current jobs have been serviced and the processor will otherwise idle. Our work focuses on improving the performance of soft real-time processes in terms of missed deadlines and tardiness by consuming slack as early (and not necessarily as fairly) as possible.

GRUB [13] is a CBS-like algorithm that dynamically allocates excess capacity to active servers. It requires a very fine granularity of time and must frequently compute the duration that slack is available. The available slack is dynamically re-allocated to servers by updating their reservations. These dynamic operations incur a large overhead. RBED [4] also provides CBS-like slack management for best-effort processes, which may consume the slack of hard real-time and soft real-time processes. Although slack is evenly distributed among the runnable best-effort processes, a newly-entered best-effort task can cause others to temporarily starve; it also does not allow other classes of processes (notably, soft real-time) to take advantage of dynamic slack. In HisReWri [3], designed for fixed priority schedulers, recent scheduling history is reviewed and consumed slack is retroactively allocated to tasks that executed; if slack was available, tasks' budgets are replenished by the amount of slack they consumed.

CASH [5] extends CBS to include a slack reclaiming algorithm. When a server becomes idle with residual budget, the slack is inserted onto the *cash queue* ordered by servers' deadlines. Whenever a new server is scheduled for execution it will first use any queued budget whose deadline is less than or equal to its own. CASH has the disadvantage of using deadline extension for servers: the earliest available slack is always used for the current task whose server has the earliest deadline, but a task that needs more budget to complete will have its deadline postponed before it completes. CASH performs poorly with aperiodic tasks since a task can't start to use its own server's budget until the server's previous slack in the cash queue is exhausted. Recent modifications to CASH improve its bandwidth sharing and allow it to work in the presense of shared resources [6].

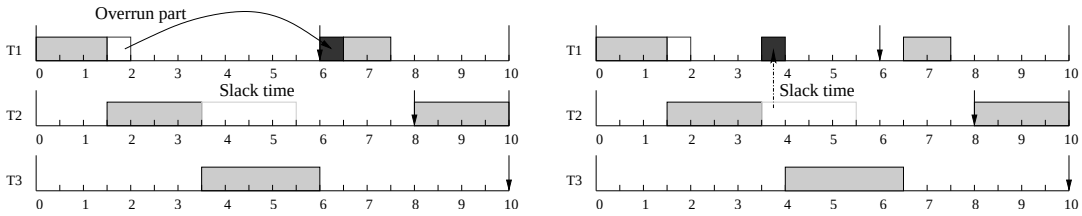
The BWI algorithm [11] applies the idea of priority inheritance to CPU resources in CBS, allowing a blocking low-priority process to steal resources from a higher-priority process that it has blocked. The CFA algorithm [17] improves the BWI algorithm by tracking stolen resources and allowing victimized tasks to reclaim them. Both of these mechanisms improve the performance of CBS in the presence of priority inversions caused by access to shared resources. Because our present research is concerned exclusively with independent processes, neither BWI nor CFA would affect the performance of CBS or any of the other algorithms we examined. In addition, neither of these algorithms directly manages slack, as ours do.

3 Slack scheduling principles

Our goal is to derive an effective EDF-based slack scheduler from first principles. We present a series of examples demonstrating potential slack management problems. For each problem, we present the solution to the problem, derive a general principle embodying that solution, and discuss how the principle can be applied in practice.

3.1 When to allocate slack

The first question we examine is that of when to allocate slack. The traditional approach is to allow slack-consuming applications to run when all real-time tasks are idle, as implemented in RT-Linux, Linux, and other systems based



(a) Problem 1 (b) Solution 1

Figure 1. When to allocate slack

the POSIX standard. This approach isolates the real-time tasks from the non-deterministic behavior of non-compliant or otherwise non-real-time tasks but, because it delays the use of the slack, it does not always allow it to be used for overruns by other tasks, reducing its utility.

Figure 1(a) illustrates this problem. All three tasks (labeled T1, T2, and T3) are released at time 0. Task 1 has a reservation of 1.5 time units, but needs 2 units to meet its deadline at time 6. Task 2 has a reservation of 4 time units, but only needs 2 units to meet its deadline at time 8. Task 3 has a reservation of 2.5 time units and needs all 2.5 units to meet its deadline at time 10. Although Task 2 has slack available at time 3.5 that could be used by Task 1, the system is not idle until time 6, at which time Task 1 misses its deadline.

The solution is to provide Task 2's slack to Task 1 at time 3.5, as illustrated in Figure 1(b). The system should not wait until all tasks are idle, nor should it use the next deadline of the overrun task to determine its priority (as in CBS, IRIS, RBED, and BEBS). Because they are reserved by Task 2, the 2 time units of dynamic slack generated by Task 2 can safely be used to meet the processing needs of Task 1, allowing Task 1 to meet its deadline, without causing Task 3 to miss its deadline. This leads to a potential principle:

Potential Principle. *Allocate slack as early as possible.*

Because our goal is to meet timeliness constraints, earlier allocation of slack should produce better results. Providing slack to a task earlier may allow it to meet a deadline that it would otherwise have missed, as shown in Figure 1(b), and should never cause it to miss a deadline that it would otherwise have met.

There is one problem with this potential principle. Although a task should allocate slack as soon as it is available, it cannot necessarily allocate all of its slack immediately. If a higher-priority task comes along, that task must be allowed to execute before the allocation of the slack proceeds. The solution is to execute the slack-consuming task with the priority of the donating task. In this way, the slack-consuming task will never delay the execution of any task that would not have been delayed by the execution of the donating task had the donating task consumed all of its reservation, preserving the correctness of the schedule. This leads to Principle 1:

Principle 1. *Allocate slack as early as possible, with the priority of the donating task.*

Implementing Principle 1 requires that the slack generated by all tasks is known at the time that they complete the processing for their current job. This is feasible in any scheduler that budgets resources to tasks and dynamically tracks their resource usage, as does our system and many of the other systems and schedulers discussed above.

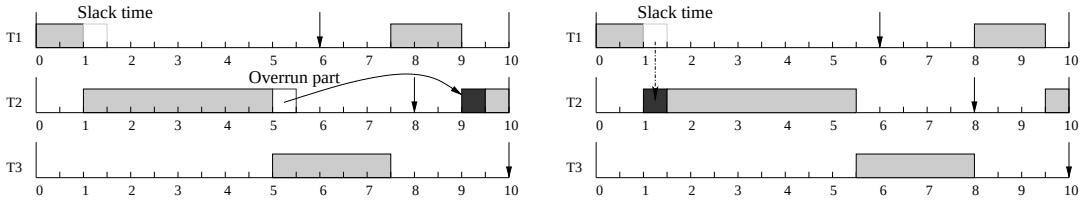
Our SRAND algorithm, described in Section 4.3, implements Principle 1. When a task completes, any remaining budget (slack) is used to schedule a randomly selected task, at the priority of the donating task. SRAND performs surprisingly well, providing fewer deadline misses than the traditional approach, demonstrating the effectiveness of early slack donation. However, randomly allocating slack to tasks is not optimal. The question of which task to donate slack to is addressed in the next section.

3.2 Who to allocate slack to

The next question we examine is that of which task to allocate the slack to. One possible solution is to just give it to tasks that have overrun their budget. However, a task that is going to overrun its budget may not have done so at the time that the slack is available, and thus may not yet know that it needs slack.

This problem is illustrated in Figure 2(a). All three tasks are again released at time 0 and the deadline and reservations are as in Figure 1: Task 1 has a period of 6 and a reservation of 1.5 time units, Task 2 has a period of 8 and a reservation of 4 times units, and Task 3 has a period of 10 and a reservation of 2.5 times units. This time, however, Task 1 only needs 1 time unit to complete its processing, generating .5 units of slack, while Task 2 overruns its budget, requiring 4.5 time units to complete. Principle 1 does not necessarily help in this situation because at the time that the slack is available from Task 1, Task 2 does not need any slack, and thus may not receive any.

The solution to this problem, as illustrated in Figure 2(b), is to provide the slack to Task 2 as soon as it is available,



(a) Problem 2 (b) Solution 2

Figure 2. Who to allocate slack to

even though it has not yet overrun its budget. This allows Task 2 to use both its allocated resources and the slack generated by Task 1 to meet its deadline at time 8. Again, because Task 2 uses the resources of Task 1 at the priority of Task 1, it does not interfere with Task 3 and all tasks meet their deadlines, preserving the correctness of the schedule. This leads to Principle 2.

Principle 2. *Allocate slack to the task with the highest priority (earliest deadline).*

Principle 2 says that not only should the system immediately make the slack available to other processes, as specified by Principle 1, but it should give it preemptively to the task with the earliest deadline. The reasoning behind this is that the server with the earliest deadline is the most critical, by virtue of being closest to its deadline, and thus the least likely to benefit from any later donations of slack. If any task will overrun its budget, the first one to do so is likely to be the one with the earliest deadline. In that case, the slack given to it will increase the likelihood that it will be able to meet its deadline. And if it does not overrun its budget, then it will complete sooner and it will itself have slack that can be allocated to the task with the next earliest deadline, and so on. This provides slack uniformly to many tasks, in order of deadline, carrying it forward until it is needed.

In a system based on EDF, the information needed to implement this principle is easily accessible. The task to receive the slack is always the next task to be scheduled. Before consuming its own budget, the task will consume any remaining budget of the donating task. Should a higher-priority task interrupt the task, it will consume any remaining slack from the donating process before consuming its own budgeted resources.

Our SLAD SLAck Donation algorithm, described in Section 4.3 implements Principles 1 and 2. It allocates slack as soon as it is available, to the process with the earliest deadline, with the priority of the donating task. SLAD always outperforms SRAND, demonstrating the effectiveness of Principle 2. SLAD also outperforms CBS and some other algorithms using the early-release mechanism, but the early-release idea has additional benefits, as discussed in the next section.

3.3 Borrowing from the future

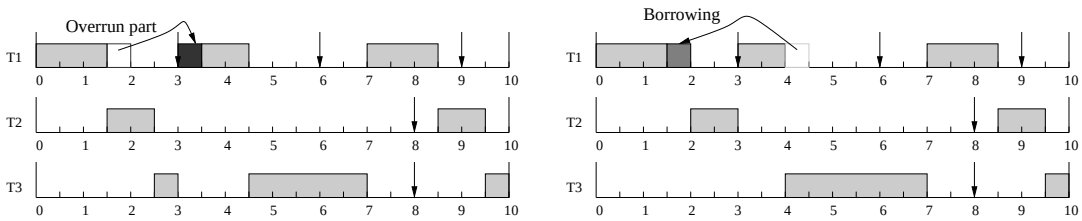
In CBS, IRIS, RBED, and BEBS, when a task overruns the resources allocated for its current job, its budget is replenished and its deadline is advanced one period. This allows the current job to borrow from the budget reserved for the task's next job. If the subsequent job underruns its budget by at least the amount borrowed for the current job, then this borrowing caused no problems. If the subsequent job needs more resources, it may itself borrow from the job after it. In this way, tasks whose actual execution varies around an average-case estimate may meet far more deadlines. Because the resources borrowed from future executions of the task are executed at the priority of the following job (i.e. with that job's deadline), this borrowing preserves the correctness of the schedule.

Although SLAD generally outperforms CBS, it can still be useful to allow a job that overruns its budget to borrow budget from a future job of the same task when no slack is currently available. This is illustrated in Figure 3(a). Task 1 has a budget of 1.5 and a period of 3, Task 2 has a budget of 1 and a period of 8, and Task 3 has a budget of 3 and a period of 8. Tasks 2 and 3 always use their budget, but job 1 of Task 1 needs 2 time units to complete, while job 2 needs only 1. Because no task has any slack to donate to job 1 of Task 1, Task 1 misses its first deadline.

The solution to this problem is illustrated in Figure 3(b). The budget of job 1 of Task 1 can be replenished with the budget of job 2 of Task 1, and the deadline extended to that of job 2. Because job 2 of Task 1 has an earlier deadline than the current jobs of Tasks 2 and 3, job 1 of Task 1 continues to execute and meets its deadline. Because Job 2 needs less than its full allocation, it also completes before its deadline. The correctness of the schedule is preserved because job 1 of Task 1 uses the resources of job 2 with the priority (i.e. deadline) of job 2. This gives us Principle 3:

Principle 3. *Allow tasks to borrow against their own future resource reservations (with the priority of the job from which the resources are borrowed) to complete their current job.*

Principle 3 says that in addition to donating slack to the task with the earliest deadline as soon as it is available, as



(a) Problem 3 (b) Solution 3

Figure 3. Borrowing from the future

specified by Principles 1 and 2, we should also allow tasks to borrow against their own future allocations, as in CBS, IRIS, RBED, and BEBS.

Strictly speaking, this principle is about the scheduling of potential future slack; resources are borrowed from future jobs of the same task in the hope that the resources will turn out to have been slack. This principle in effect says: always deal with the most critical job of a task first. When a job overruns its budget, we know that it needs more resources. Allowing it to use resources allocated to the next job of the same task, with the priority of that job, cannot hurt any other task, and may help the current job of the overrun task. Of course, borrowing resources from the future may cause future jobs to miss deadlines. But we know that if we don't borrow, the current job will miss its deadline, while if we do borrow, the next job may use less resources than it was allocated or get slack from another process—because its deadline is later, it is more likely to get slack than the current job.

This principle is straightforward to implement. Once a task's budget is depleted, it is immediately recharged and current deadline is extended to the next deadline. Once the deadline has been extended, the system continues to be scheduled with EDF. The deadline extension mechanism provides resource isolation and prevents any violation to other servers, preserving the feasibility of the schedule.

Principle 3 alone may cause two problems. The first is that a task whose deadline has been extended may miss an opportunity for slack donation, as occurs in CASH. According to Principle 2, slack is always allocated to the task with the highest priority (i.e. earliest deadline). A task whose deadline has been extended has effectively had its priority lowered, possibly below that of another task, in which case any slack that comes available will be given to that other task. This takes slack away from the very task that needs it the most—the one that already used up its original resource allocation.

This problem may be addressed by a slight modification to Principle 2:

Revised Principle 2. *Allocate slack to the task with the highest priority (earliest* **original** *deadline).*

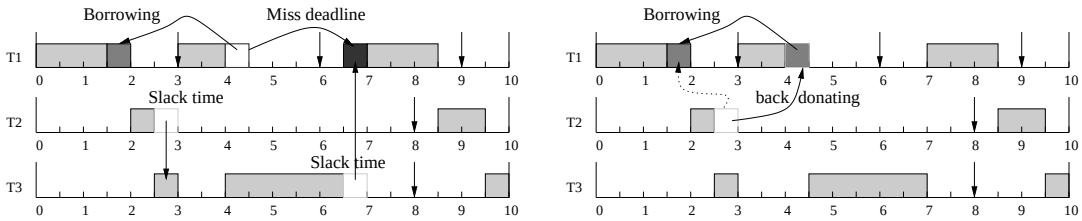
The revised Principle 2 says that slack should always be allocated to the task with the earliest original deadline regardless of whether or not that task has been dynamically assigned a new extended deadline. Implementing this revised Principle 2 requires a little more record-keeping to keep track of both the original and extended deadlines, but the overhead is negligible.

Our SLASH (SLAD + CASH) algorithm, described in Section 4.4, implements Principles 1, 2, and 3. It allocates slack as soon as it is available, to the task with the earliest original deadline, and allows jobs to borrow from future jobs of the same task. SLASH generally outperforms both SRAND and SLAD. However, it introduces another problem. This problem and its solution are discussed in the next section.

3.4 Donating to the past

Even with the revised Principle 2, Principle 3 introduces another problem. A task which has used up its budget, borrowed resources from a future job, and then completed the job may not be eligible for slack, even though it has already demonstrated the need for the slack (and, effectively, has already used it). The reason is that the job has completed and is no longer in the EDF task queue, so is not even considered for donation by the slack scheduling algorithm.

This problem is illustrated in Figure 4(a). The tasks are the same as in Figure 3: Task 1 has a budget of 1.5 and a period of 3, Task 2 has a budget of 1 and a period of 8, and Task 3 has a budget of 3 and a period of 8. This time, however, we assume that both jobs 1 and 2 of Task 1 need their full allocation of 2 time units and Task 2 requires only .5 time units to meets its deadline. At time 1.5 when Task 1 needs slack, no slack is yet available from Task 2 so, applying Principle 3, job 1 of Task 1 is allowed to borrow from job 2 of Task 1 (with the priority of job 2) to meet the deadline of job 1 at time 3. At time 2, job 1 of Task 1 completes and Task 1 idles until its next release time at time 3. At time 2.5, Task 2 completes with .5 units of slack time. As Task 3 is the only one that is active, it receives the slack from Task 2. Then, at time 3, job 2 of Task 1 is released with deadline of 6 and a budget of 1 time unit (depleted due to borrowing by job 1). Task 1 runs from time 3 to time 4, consuming



(a) Problem 4 (b) Solution 4

Figure 4. Donating to the past

its entire depleted budget. It then recharges its budget and extends its deadline to time 9. Because Task 3 has an earlier deadline, it runs until time 6.5, at which time it finishes with .5 units of slack (the slack carried forward from Task 2). It now gives the slack to Task 1, but it is too late to help—Task 1 missed its deadline at time 6.

The problem is that, although Task 1 needed slack, it completed the slack-consuming job by borrowing from its next job before the slack was available and the next job (with depleted budget) has not yet been released, making it ineligible to receive slack. The solution to this problem, as illustrated in Figure 4(b), is for Task 2 to give the slack generated at time 2.5 to Task 1, which has already demonstrated the need for slack. This allows job 2 of Task 1 to meet its deadline at time 6. This leads to Principle 4:

Principle 4. *Retroactively allocate slack to tasks that have borrowed from their current budget to complete a previous job.*

In effect, this principle says that a task that borrows from a future job to complete its current job should remain eligible to receive slack with the priority of its original (unextended) deadline. Having demonstrated the need for slack, this task should receive slack before currently executing jobs with deadlines later than its original deadline. This slack back-donation principle guarantees that any deadline extension of a task's job should not affect the slack donation mechanism.

This principle is an EDF version of the mechanism implemented in HisReWri. HisReWri operates on static priority rate monotonic schedules, but effectively does the same thing: tasks that previously consumed slack are eligible to receive future slack donations. Implementing this principle requires that the system maintain information about recently completed jobs that consumed slack. Alternatively, it can keep information about any depleted budgets, and the deadlines of the jobs that did the depleting. With this information, slack can be back-donated according to the EDF principle used for regular slack donation.

Our BACKSLASH algorithm, discussed in Section 4.4, implements this principle, in addition to Principles 1, 2, and 3. It allocates slack as early as possible, to the task with the earliest deadline, allows borrowing from future jobs of the same task, and allows jobs that have borrowed from future jobs of the same task to remain eligible for slack donations that occur before their original deadline. Our results show that BACKSLASH, which implements all four principles, generally outperforms all of the other slack scheduling algorithms we examined.

4 Implementation details

This section presents details of the implementation of our slack schedulers.

4.1 Task model

Like RBED, VRE, and BEBS, our real-time system uses an integrated scheduler for hard, soft, and non-real-time processes. The low-level scheduler is earliest deadline first (EDF [14]). Processes use the scheduler by associating their tasks with a rate-based server.

4.2 Rate-based server and EDF scheduling

Resource allocation is achieved using *rate-based servers*, which are conceptually similar to those of CBS, BEBS, and other bandwidth servers. A server is characterized by a reservation tuple (B_s, P_s), where B_s is the execution budget and P_s is the period (both in units of time). The server utilization is $U_s = \frac{B_s}{P_s}$. A dynamic deadline $d_{s,k}$ (where k is the period index) occurs at the end of each period.

Each hard or soft real-time task is associated with its own server. Periodic and aperiodic best-effort tasks are scheduled as soft real-time tasks. All other best-effort tasks are served by a single server (called BEServer) in a time-sharing fashion. The configurations of the servers for the different kinds of tasks are as follows:

- **Hard real-time**: B_s is the worst-case execution time (WCET) and P_s is the period (for periodic tasks) or inter-arrival rate (for aperiodic tasks). Every hard real-time task is guaranteed to receive its worst-case CPU

budget in every period. The total utilization of hard real-time servers is $U_{\mathbf{HRT}}$.

- **Soft real-time**: B_s may be less than the worst case execution time (e.g., average case execution time) and P_s is the period (for periodic tasks) or minimum inter-arrival time (for aperiodic tasks). Every task is guaranteed a minimum CPU budget per period, allowing good average performance, though some jobs may miss deadlines. A job whose deadline has been missed may be dropped or may continue to run in the next server period, depending upon the task. The total utilization of soft real-time servers is $U_{\mathbf{SRT}}$.
- **Best-effort**: P_{BE} is set to provide good responsiveness based on the system load, and $B_{BE} = U_{\mathbf{BE}} \times P_{BE}$, where $U_{\mathbf{BE}}$ is the unreserved utilization $(1 - U_{\mathbf{HRT}} - U_{\mathbf{SRT}})$.

The scheduling algorithm used in our system is earliest-deadline first (EDF). Since the system enforces $(U_{\mathbf{HRT}} + U_{\mathbf{SRT}} + U_{\mathbf{BE}}) \leq 1$ at all times, it always guarantees that all deadlines of all servers will be met. Because the budget B_s of the server for every admitted hard real-time task is always set to the task's WCET, this guarantees that all hard real-time deadlines will always be met. Other (e.g., soft real-time) tasks deadlines may or may not be met depending upon their servers' budgets and the availability and effective use of dynamic slack.

Under EDF, the server with the earliest deadline becomes eligible to pick its next pending job to run on the CPU. If a server has consumed less than its budget and has no pending tasks to execute, the remaining budget is slack and may be donated to another server.

To ensure a feasible schedule under EDF, a task must not overrun its server's reservation. Our system uses a one-shot timer to prevent task overrun. Upon task execution, if it reaches its server's reservation (by using up its budget), the one-shot timer interrupt handler will preempt the task and take the actions specified by the specific server scheduling algorithm (described in the following subsections).

4.3 SRAND and SLAD

SRAND implements Principle 1—it donates slack as early as possible. Whenever a server generates slack, SRAND randomly selects another active server (whose deadline is necessarily greater than that of the current tasks) to execute with the remaining budget of the donating server.

The **SLA**ck **D**onation (SLAD) slack scheduling algorithm implements Principles 1 and 2—it donates slack as early as possible, to the task with the earliest deadline. SLAD was designed for periodic tasks, although it also works for aperiodic tasks. In SLAD, a rate-based server has four states: *idle*, *waiting*, *running* and *expired*. The SLAD algorithm is as follows:

1. When a new task arrives, a new server is created for it and its parameters (B_s, P_s) are initialized as described in Section 4.2. The server state is set to *idle*.
2. When a new job arrives it is enqueued in the server queue. If the server is *idle*, its deadline $d_{s,k}$ is set to the current time and a new period starts.
3. At the beginning of a server period, the current budget of a server c_s is set to its reservation budget B_s, its dynamic deadline $d_{s,k+1}$ is set to $d_{s,k} + P_s$, and its state is set to *waiting*.
4. The *waiting* server with the earliest deadline becomes *running*. If there is no *waiting* or *running* server, the system becomes idle and the idle time is donated to the same task as described in step 5.
5. A *running* server executes its pending task on the CPU until it has finished its task or consumed its budget, and decreases its budget c_s by the actual amount of CPU consumed. If it has no pending task, it donates any remaining budget to:
   (a) the task of the server in the *waiting* or *expired* state with the earliest deadline; or, if none exist, to
   (b) the idle task
6. When c_s of a *running* server equals zero, if there is a pending job, its state is set to *expired* until the start of the next period, at $d_{s,k+1}$, at which time step 3 is repeated; otherwise, its state is set to *idle* until a new job arrives.
7. When a *running* server is preempted, its state is set to *waiting*.

4.4 SLASH and BACKSLASH

Ideally, we would like to donate slack to the soft real-time tasks that need it the most, and execute overrun tasks as early as possible (so that they improve their chance of meeting deadlines) by not forcing a server to remain inactive when its budget is consumed. SLASH and BACKSLASH address these two desires by improving the SLAD EDF-based slack donation mechanism with Principles 3 and 4 (as well as the Revised Principle 2).

Unlike SLAD, SLASH and BACKSLASH have no *expired* state. When a server consumes its budget, the budget is recharged, its deadline is advanced by one period, and its state is reset to *waiting*. To implement the revised Principle 2, SLASH and BACKSLASH use earliest virtual deadline first (EVDF) for slack scheduling decisions. The virtual deadline, which is the **original** deadline, of a server is calculated as follows:

$$vd_{s,k} = d_{s,k} - \left\lfloor \frac{d_{s,k} - t}{P_s} \right\rfloor P_s$$

where t is current time (note that the conditions $d_{s,k} > t$ and $vd_{s,k} \leq d_{s,k}$ always hold). While a server's deadline may be extended upon expiration, the virtual deadline remains unchanged until the server's job completes.

The detailed algorithm of SLASH differs from SLAD (described in Section 4.3) in the following manner:

1. In step 2, if the server is *idle* and $c_s \geq (d_{s,k} - t)U_s$, $d_{s,k}$ is set to current time t and a new period starts.
2. In step 5, for the *running* server s without a pending task: if $vd_{s,k} < d_{s,k}$, the server's state is set to *idle*; otherwise the server s donates slack using EVDF schedule, *i.e.* it donates any remaining budget to the task of another *waiting* server with the earliest **virtual** deadline (instead of earliest deadline).
3. In step 6, when c_s of a *running* server becomes zero, if there is a pending job, the server is recharged with full budget $c_s = B_s$, its deadline is incremented $d_{s,k+1} = d_{s,k} + P_s$, and its state is reset to *waiting* (or remains *running* if it still has the earliest deadline); otherwise, its state is set to *idle* until a new job arrives.

The modification to step 6 allows an expired server to borrow from its future budget and remain in the *waiting* or *running* state. However, under the modified rule, a task whose server has already borrowed some budget from the future would have been donated slack but missed the opportunity. The modifications of step 5, maintains the original slack donation mechanism (as in SLAD) regardless of whether the server has borrowed from its future budget. The modification to step 2 allows SLASH to support aperiodic task arrivals.

In order to support slack back-donation, BACKSLASH uses a back-donation queue Q which contains the servers that have borrowed budget from their future. The detailed algorithm of BACKSLASH is similar to SLASH except BACKSLASH guarantees the task that should have been donated the slack will be retroactively donated the slack, which is shown in a modification to **step 5** of the SLASH algorithm:

1. In step 5, for the *running* server s without pending task: if $vd_{s,k} < d_{s,k}$, the server's state is set to *idle* and if $c_s < B_s$ (when $vd_{s,k} < d_{s,k}$ is true), the server s is enqueued in the back-donation queue Q in **virtual** deadline order; otherwise the server s is scheduled as follows until its budget is gone:
   (a) if Q is empty, it donates slack using EVDF schedule as in the SLASH algorithm; otherwise
   (b) it starts back-donating: it runs the next task whose server is not slack server and has the earliest deadline (*i.e.* using a normal EDF but not slack schedule); when running, decreasing the budget of the slack server s and the budget of the server of the running task by the same amount of CPU consumed while increasing the budget (up to the same amount) of the first server s_{evd} in Q (If $c_{s_{evd}} == B_{s_{evd}}$, s_{evd} is dequeued from Q and it is assigned the next available server in Q).

5 Performance evaluation

We implemented SRAND, SLAD, SLASH and BACKSLASH in the Linux 2.6 kernel. For comparison, we also implemented CBS, CASH, IRIS, BEBS, and "EDF", a reservation-based EDF algorithm that, when all other tasks are idle, allocates slack to the task with earliest deadline. Our test machine was a 1 GHz Intel Pentium 3.

Our results showed that hard real-time tasks frequently missed their deadlines with IRIS. Even when all hard real-time tasks met their deadlines, the performance of soft real-time tasks was worse than in most of the other algorithms. BEBS performed better than IRIS, but similar to CASH. For clarity, BEBS and IRIS were left out of our performance figures. In addition, because we are concerned with the effect of slack allocation on the performance of the soft real-time tasks, our figures show only those tasks.

5.1 Performance metrics

Our metrics for soft real-time performance are *deadline miss ratio* (DMR)—the ratio of deadlines missed to the total number of periods (jobs)—and *tardiness* (TRD)—the ratio of the total accumulated lateness for all jobs to the total length of all periods. Lateness is 0 for jobs finishing on or before their deadlines, and the amount by which the job has missed its deadline for any job finishing after its deadline.

The deadline miss ratio and tardiness for a set of soft real-time tasks can be computed in two ways: averaged over the tasks, treating all *tasks* as equally important, or over the entire experiment, treating all *deadlines* as equally important. Which one is more important depends upon the goals of the system and the behavior of the algorithms differs somewhat under these two different set of metrics so we have chosen to present both. Interestingly, our algorithms do somewhat better using the second set of metrics.

Average Deadline Miss Ratio (ADMR) and Average Tardiness (ATRD) average DMR and TRD over the tasks and are defined as follows:

- $ADMR = \frac{\Sigma_1^i dmr_i}{n}$, where dmr_i is the deadline miss ratio of soft real-time task T_i.
- $ATRD = \frac{\Sigma_1^i trd_i}{n}$, where trd_i is the tardiness of soft real-time task T_i.

Overall Deadline Miss Ratio (ODMR) and Overall Tardiness (OTRD) average DMR and TRD over all periods and are defined as follows:

- $ODMR = \frac{\Sigma_1^i ndm_i}{\Sigma_1^i np_i}$, where ndm_i is the total number of deadlines missed (*i.e.* $ndm_i = dmr_i \cdot np_i$) and np_i is the total number of periods of task T_i.
- $OTRD = \frac{\Sigma_1^i (trd_i \cdot np_i)}{\Sigma_1^i np_i}$.

ODMR and OTRD treat all *deadlines* as equally important.

5.2 Workloads

Although we have conducted extensive comparisons, in the interest of brevity we discuss two representative workloads: fixed task sets and random tasks sets. We first investigate the effect of load and period on the performance of fixed sets of soft real-time tasks. Then we use extensive random workloads to demonstrate the overall and average performance of soft real-time tasks.

All of the real-time workloads for our tests were generated using a tool we developed for the purpose. Given a specification of a period (or minimum inter-arrival time) and execution time (worst-case execution time (WCET) or average-case execution time(ACET)), the tool generates periodic hard real-time or soft real-time tasks with constant or variable (with a normal distribution) execution times.

In all of our experiments, the actual execution time e of a task is a random value drawn from one of the following distributions:

1. $\text{NW}(\mu) = \begin{cases} \frac{1}{\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}, & 0 < x \leq \mu \\ 0, & x \leq 0 \text{ or } x > \mu \end{cases}$
   a normal distribution (with mean μ and standard deviation $\sigma = 0.1\mu$) except for the values that are non-positive or greater than μ.

2. $\text{NA}(\mu) = \begin{cases} \frac{1}{\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}, & 0 < x \\ 0, & x \leq 0 \end{cases}$
   a normal distribution with mean μ and standard deviation $\sigma = 0.1\mu$, except for the non-positive values.

Since we are focusing on the performance of real-time applications in a mixed environment we arbitrarily reserve a minimum of 2% of the CPU for best-effort tasks, enough to provide a functional interactive system for running command shells during the experiments.

5.3 Performance results with fixed workloads

In the fixed workload experiments, we change one or two task parameters (such as execution budget or period) while keeping the other parameters and workload characteristics fixed. This allows us to investigate the robustness of the slack scheduling algorithms to changes in these parameters.

5.3.1 Load effect on performance

The first experiment shows soft real-time performance as a function of system load. The workload consists of two periodic hard real-time tasks and one periodic soft real-time task, given in Table 1 (the time units for all task parameters in this section are in milliseconds).

Table 1. Workload 1

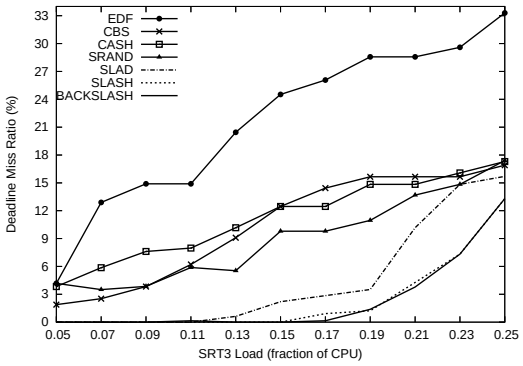
Task	Task Parameters		Server Parameters			Parameter Adjustment	
	$e = f(\bar{e})$	p	$B = \bar{e}$	$P = p$	$U = \frac{B}{P}$	$\Delta(\bar{e})$	$\Delta(U)$
HRT1	258	600	258	600	43%	+12	+2%
HRT2	NW(175)	350	175	350	50%	-14	−4%
SRT3	NA(15)	300	15	300	5%	+6	+2%

In this workload HRT1 has constant execution time equal to its server budget, HRT2 has normally distributed execution times with its server budget equal to its WCET, and SRT3 has normally distributed execution times with its server budget set to its ACET. SRT3 will often overrun its budget but should still be able to meet most of its deadlines by taking advantage of the slack from HRT2 and borrowing its own future budget as necessary.

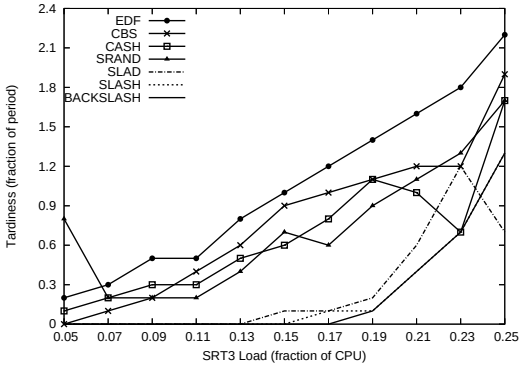
Figure 5 shows the deadline miss ratio and tardiness[2] of SRT3 as a function of its task load ranging from 5% to 25% under each of the slack scheduling algorithms, with total server load equal to 100% with 2% reserved for best-effort tasks. Figure 5(a) shows that (1) SRAND outperforms EDF, demonstrating the benefit of donating slack at the earliest possible time, (2) SLAD outperforms SRAND, demonstrating the additional benefit of donating slack to the task with the earliest deadline, (3) SLASH outperforms SLAD, demonstrating the effectiveness of allowing jobs to borrow from future jobs of the same task, (4) BACKSLASH slightly outperforms SLASH, demonstrating the effectiveness of slack back-donation, and (5) BACKSLASH outperforms all of the other algorithms we examined. It reduces the deadline miss ratio of SRT3 to zero until SRT3's load exceeds 17% and in the worst scenarios still reduced SRT3's deadline miss ratio by about 21% compared to CBS or CASH. Unsurprisingly, all of the algorithms perform better when the load of SRT3 is low (*i.e.* the load of the HRTs is high and the amount of available slack is large), and perform worse as the load of SRT3 increases.

Figure 5(b) shows the tardiness of SRT3 under each of the slack scheduling algorithms. The results are similar to the deadline miss ratio results, except that SLAD has less tardiness than BACKSLASH when SRT3's load is equal to 25%. This is because SLAD may cause a soft real-time task to miss more deadlines by smaller amounts, while BACKSLASH may cause it to miss fewer deadlines but by larger amounts.

[2] With only one soft real-time task ADMR=ODMR and ATRD=OTRD.

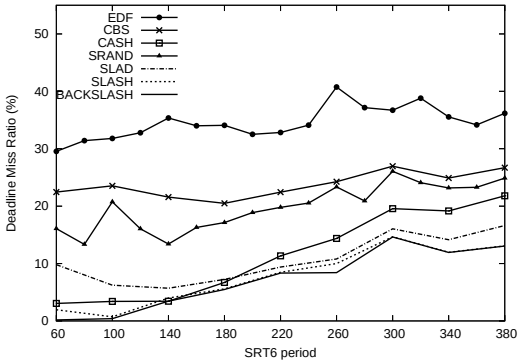


(a) Deadline Miss Ratio as a function of load

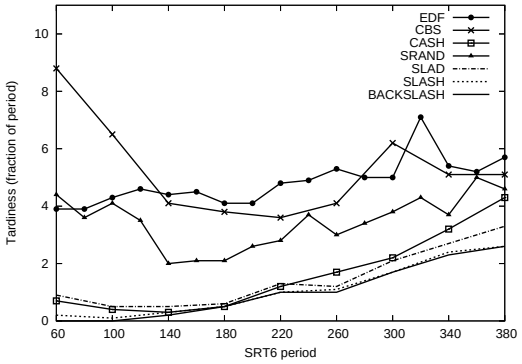


(b) Tardiness as a function of load

Figure 5. Load effect on performance (one soft real-time task, $p = 300$ms)



(a) Deadline Miss Ratio as a function of period



(b) Tardiness as a function of period

Figure 6. Period effect on performance (one soft real-time task, $u = 50\%$)

5.3.2 Period effect on performance

The second experiment shows soft real-time performance as a function of server period. The workload consists of five periodic hard real-time tasks and one periodic soft real-time task, given in Table 2.

Table 2. Workload 2

Task	Task Parameters		Server Parameters			Parameter Adjustment	
	$e = f(\vec{e})$	p	B	P	$U = \frac{B}{P}$	$\Delta(\vec{e})$	$\Delta(p)$
HRT1	NW(20)	200	20	200	10%	0	0
HRT2	NW(30)	300	30	300	10%	0	0
HRT3	NW(40)	400	40	400	10%	0	0
HRT4	NW(50)	500	50	500	10%	0	0
HRT5	NW(48)	600	48	600	8%	0	0
SRT6	NA(30)	60	30	60	50%	+20	+40

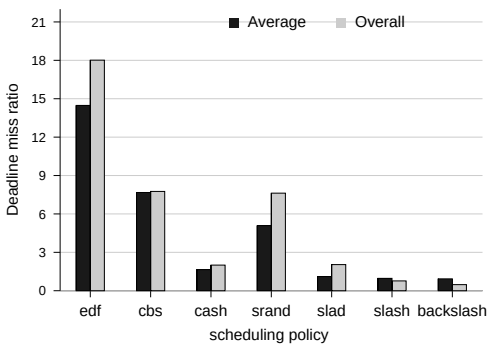
In this workload, every hard real-time task has normally distributed execution times with server budgets set to their WCET. SRT6 has normally distributed execution times with its server budget set to its ACET. The first four hard real-time tasks each reserve 10% of the CPU, the last one reserves 8%, and SRT6 reserves the remaining 50%.

Figure 6 shows SRT6's performance as a function of period ranging from 60ms to 380ms. The results again demonstrate that (1) SRAND outperforms EDF, verifying principle 1, (2) SLAD outperforms SRAND, verifying principle 2, (3) SLASH outperforms SLAD, verifying principle 3, (4) BACKSLASH always slightly outperforms SLASH, verifying principle 4, and (5) BACKSLASH always outperforms all of the other algorithms, again reducing the deadline miss ratio by 100% in the best case.

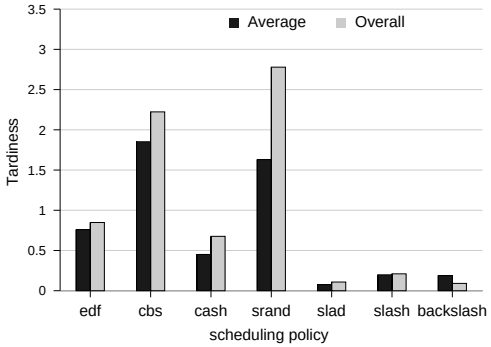
5.4 Performance results with random workloads

The purpose of our random workloads is to show how well each algorithm performs in different dynamic random scenarios in which the number of hard real-time and soft real-time tasks, the task model (periodic or aperiodic) and the tasks' parameters (periods or minimum inter-arrival time, and execution budget) all differ. The periods and execution budgets vary randomly from 1 ms to 1000 ms. These workloads allow us to study the average performance of different slack scheduling algorithms.

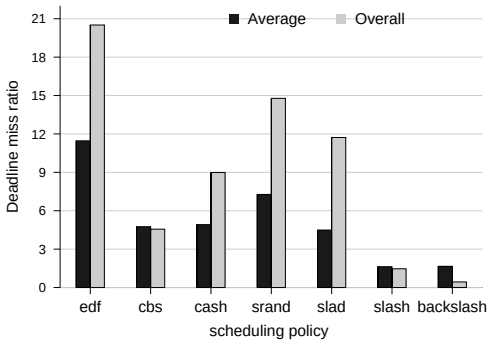
The first random workload consists of 12 task sets, each



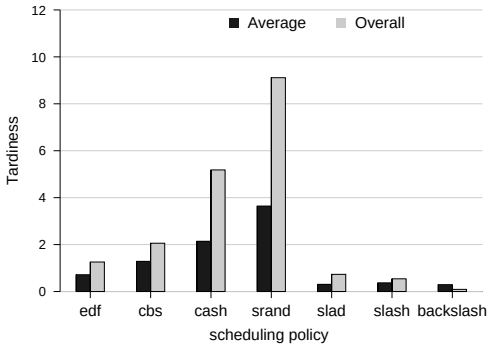
(a) Deadline miss ratio: periodic task sets



(b) Tardiness: periodic task sets



(c) Deadline miss ratio: aperiodic task sets



(d) Tardiness: aperiodic task sets

Figure 7. random workloads with 8 tasks

of which has 8 periodic tasks with a random distribution of hard and soft real-time tasks. Each task has random load and period such that the total load is high enough to overload the CPU. The second random workload is the same as the first one except that all the tasks are generated randomly in aperiodic or periodic mode (called aperiodic task sets for simplicity). We run each task set for 100 seconds and measure the performance on average using both metrics.

Figure 7 shows the performance of the different slack scheduling algorithms on the random workloads. The performance results for the first random workload (Figures 7(a) and (b)) are very similar to those for the fixed workloads. BACKSLASH again outperforms all other algorithms except that SLAD has lower average tardiness than SLASH and BACKSLASH. Compared to CBS and CASH, BACKSLASH achieves respectively 88% and 44% lower average deadline miss ratio and 90% and 58% lower average tardiness (BACKSLASH achieves even better performance in terms of overall deadline miss and overall tardiness).

The performance results on the aperiodic workload (Figures 7(c) and (d)) are similar to those for the periodic random workload, with a few notable differences. In general, BACKSLASH outperforms all the other algorithms in terms of both average/overall deadline miss ratio and average/overall tardiness. In the best scenarios it achieves respectively 65% and 66% lower average deadline miss ratio and 77% and 86% lower average tardiness compared to CBS and CASH. SLAD does not perform well in terms of deadline miss ratio for aperiodic task sets although it does well for periodic task sets; however it always has lower tardiness than CBS and CASH. Although CBS is designed for both periodic and aperiodic tasks, it achieves much better performance with aperiodic task sets. CASH is designed for periodic tasks, so it performs much better with them than with aperiodic tasks.

Additional experiments (not shown) are consistent with these results. The exact performance depends upon the load ratio of the slack-donating and slack-consuming tasks, but in general BACKSLASH outperforms all other algorithms.

We also measured kernel overhead in terms of total scheduling time spent in the schedule() function and total context switch time incurred, during a 100s run for a task set with nine tasks used in the last random workload (not shown). Our results indicated that the overhead for all algorithms was similarly acceptable, averaging around .06% of the CPU.

6 Conclusion

We have presented a set of principles for effective slack management in an EDF-based system that supports mixes of hard real-time, soft real-time and best-effort tasks:

Principle 1. *Allocate slack as early as possible, with the priority of the donating task.*

Principle 2. *Allocate slack to the task with the highest priority (earliest* **original** *deadline).*

Principle 3. *Allow tasks to borrow against future resource reservations (with the priority of the job from which the resources are borrowed) to complete their current job.*

Principle 4. *Retroactively allocate slack to tasks that have borrowed from their current budget to complete a previous job.*

We developed four slack scheduling algorithms, SRAND, SLAD, SLASH and BACKSLASH, each adding one principle to the previous algorithm. We implemented them in the Linux 2.6 kernel and compared them to CBS, CASH, (and IRIS and BEBS), and "EDF", a naive hierarchical EDF-based slack management algorithm.

Our results show that each of our progressively modified algorithms performs better than the previous ones, with BACKSLASH generally outperforming all others both in terms of deadline miss ratio and tardiness. Compared to CBS, and CASH, BACKSLASH reduces average deadline miss ratio by 100% and 100% respectively in the best scenarios of our fixed workloads; and by 88% and 66% in the best scenarios of our random workloads. Although designed for our system, these techniques should work equally well in any deadline-aware scheduler. In the future we plan to investigate the applicability of these principles to static priority schedulers.

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, T. Bisson, and S. A. Brandt. Integrating best-effort scheduling into a real-time system. In *Proceedings of the 25th IEEE Real-Time Systems Symposium (RTSS 2004)*, Dec. 2004.

[3] G. Bernat, I. Broster, and A. Burns. Rewriting history to exploit gain time. In *Proceedings of the 25th IEEE Real-Time Systems Symposium (RTSS 2004)*, pages 328–335, Dec. 2004.

[4] 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.

[5] M. Caccamo, G. Buttazzo, and L. Sha. Capacity sharing for overrun control. In *Proceedings of the 21th IEEE Real-Time Systems Symposium (RTSS 2000)*, pages 295–304, Dec. 2000.

[6] M. Caccamo, G. Buttazzo, and D. C. Thomas. Efficient reclaiming in reservation-based real-time systems with variable execution times. *IEEE Transactions on Computers*, 54(2):198–213, Feb. 2005.

[7] A. Colin and S. M. Petters. Experimental evaluation of code properties for wcet analysis. In *Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS 2003)*, pages 190–199, Dec. 2003.

[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] 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.

[11] G. Lamastra, G. Lipari, and L. Abeni. Bandwidth inheritance algorithm for real-time task synchronization in open systems. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS 2001)*, Dec. 2001.

[12] J. P. Lehoczky and S. Ramos-Thuel. An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority preemptive systems. In *Proceedings of the 13th IEEE Real-Time Systems Symposium (RTSS 1992)*, pages 110–123, Dec. 1992.

[13] G. Lipari and S. Baruah. Greedy reclamation of unused bandwidth in constant-bandwidth servers. In *Proceedings of the 12th Euromicro Conference on Real-Time Systems*, pages 193–200, June 2000.

[14] 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.

[15] 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.

[16] 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.

[17] R. Santos, G. Lipari, and J. Santos. Scheduling open dynamic systems: The clearing fund algorithm. In *Proceedings of the IEEE Real-Time Computing Systems and Applications*, 2004.

[18] T.-S. Tia, J. W. Liu, and M. Shankar. Algorithms and optimality of scheduling aperiodic requests in fixed-priority preemptive systems. *Real-Time Systems*, 10(1), January 1996.

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