Improving the responsiveness of Linux Applications in RTAI

Luca Marzario, Luca Abeni, Giuseppe Lipari
ReTiS Lab, Scuola Superiore S. Anna,
Piazza Martiri della Libertà
lukesky@gandalf.sssup.it {luca, lipari}@sssup.it

Abstract

When using RTAI for executing hard real-time tasks that require a large amount of processor bandwidth, Linux applications can experience a big delay. In fact, the real-time tasks, handled by RTAI, have always priority over Linux, and there are large intervals of time in which Linux is not executed at all. This approach makes it impossible to schedule soft real-time applications in Linux. In this paper, we propose to apply resource reservation techniques to the RTAI scheduler. We implemented the Constant Bandwidth Server (CBS) algorithm that permits to specify a maximum budget and a period for each task, Linux included. By using this approach, we are now able to reserve a minimum amount of processor bandwidth to Linux, with several advantages: during the debug phase of a real-time application, even if a high priority real-time task goes into a infinite cycle, Linux is still able to execute and we can examine the log files or issue shell commands. Second, we can guarantee soft real-time tasks scheduled inside Linux. Third, we can guarantee a maximum delay in the communication between hard real-time tasks, scheduled by RTAI, and soft real-time tasks inside Linux. After presenting our implementation, we show the advantages of our approach with some experiment.

1 Introduction and motivation

RTAI is a real-time kernel that permits to execute, on the same physical machine, both hard real-time applications and the Linux Operating System. RTAI is particularly useful when dealing with real-time processes that needs a guaranteed low response-time to interrupts. By using the fixed priority scheduler provided with RTAI, it is also possible to schedule periodic hard real-time, and to guarantee the schedulability of the system.

In RTAI Linux is assigned the lowest possible priority, and it is scheduled in background, i.e. when no other real-time task requires execution. This is the simplest way to guarantee that the behavior of the real-time tasks is not compromised by the behavior of Linux applications. However, under high real-time load, this method can be too restrictive for Linux applications.

In certain cases, it is useful to mix hard real-time and soft real tasks activities in the same system. Hard real-time tasks are critical activities that must finish before their assigned deadline, otherwise the system correctness is compromised. Soft real-time systems are non critical real-time activities: they are required to finish before their deadline, but occasional deadline misses do not compromise the correctness of the system. The most popular example of soft real-time application is a multi-media player: if some video frame is not displayed on time, or if it is skipped, nothing catastrophic happens. The ratio of deadline misses over a given interval of time can be taken as a measure of the Quality of Service (QoS) experienced by the final user: the lower the deadline miss ratio, the better the QoS. Soft real-time applications are usually executed on Linux because many drivers, libraries and tools for video and sound systems are already available on Linux.

When soft and hard real-time applications share the same physical machine, one possibility is to schedule hard real-time tasks as RTAI tasks, so guaranteeing a bounded response time; and soft real-time tasks as Linux processes. However, since in RTAI Linux is assigned the background priority, the amount of execution time assigned to Linux processes over a given time interval depends on the requirements of the RTAI tasks and can vary a lot from interval to interval. Therefore, the amount of execution time that multimedia streams will receive is quite irregular, and it is difficult to guarantee a-priori a certain level of QoS.

In this paper, we propose to apply a well-established

technique in the real-time system literature, called Resource Reservation, to RTAI. In particular, we changed the RTAI scheduler to implement the Earliest Deadline First (EDF) scheduling policy [5] and the Constant Bandwidth Server (CBS) [1]. With CBS, every task is assigned a capacity Q and a period P, with the meaning that a task is allowed to execute at least Q unites of time every P units of time. In addition, in our scheduler, Linux is served by a CBS: in this way, Linux cannot jeopardize the behavior of the hard real-time tasks, but it is guaranteed a minimum execution time of Q every period P. It is worth to point out that the proposed mechanism is very general, and it can be implemented with little effort in RT-Linux as well.

This mechanism has very little overhead at run-time. Moreover, it has another advantage in the debug phase. In fact, during the development of a hard real-time task, it can happen that the task goes into an infinite cycle. For example the task might actively wait for a condition that does never happen. With a pure fixed priority scheduler, it is quite difficult to see what happens, because Linux is scheduled in background and will never execute again. Instead, with our scheduler, Linux is guaranteed a minimum execution time every period: therefore, the programmer can still record the trace and see the logs showing the situation.

The paper is organized as follows: after introducing some terminology and assumption (Section 2), and a brief description of the Constant Bandwidth Server (CBS) (Section 3), we present our implementation in RTAI (Section 4). Then, we describe the experimental setup and show the results (Section 5). Finally, we draw the conclusion and discuss future improvements.

2 Terminology and System Model

A "Real-Time System" is a set of activities with timing requirements. The correctness of a Real-Time System depends not only on the correctness of the results but also on the time at which they are produced. A task is a finite or infinite sequence of requests for execution on a shared resource (e.g. the CPU). The i-th task in the system will be denoted by τ_i. A task request is also called *job* or *instance*: the j–th job of task τ_i will be denoted by $J_{i,j}$.

A job is characterized by at least an arrival time and computation time. The arrival time of job $J_{i,j}$ will be indicated by $a_{i,j}$ and its computation time by $c_{i,j}$. A real-time job has at least an additional parameter, the absolute deadline $d_{i,j}$.

Periodic tasks are tasks with periodic activations:$a_{i,j} = a_{i,j-1} + T_i$, where T_i is the period of the task. Sporadic tasks have a minimum interarrival time T_i such that $a_{i,j} \geq a_{i,j-1} + T_i$.

Real-time tasks are required to finish each job before its deadline. Usually, a periodic task has a relative deadline equal to its period: $d_{i,j} = a_{i,j} + T_i$. This means that each job must finish executing before the next job is activated. However, for very critical activities it is possible to specify relative deadline less than the period. For the sake of simplicity, in this paper we will only consider real-time tasks with relative deadline equal to period. A real-time task set is said to be schedulable under a given scheduling policy if every real-time job finishes before its deadline. In this respect, the periodic tasks of RTAI can be modeled by real-time periodic tasks, whereas Linux can be modeled as a tasks that consists of only one job that is always active.

Resource reservation in real-time systems. A general methodology for resource scheduling in real-time system is the resource reservation framework. The idea is not new [6, 7, 12]. However, it was first formally introduced by Rajkumar [11]. Each task is reserved a fraction of the processor available bandwidth: if the task tries to use more than it has been assigned, it is *slowed down.*

This framework allows a task to execute in a system as it were executing on a dedicated virtual processor, whose speed is a fraction of the speed of the processor. Thus, by using a resource reservation mechanism, the problem of schedulability analysis reduces to the problem of estimating the computation time of the task without considering the rest of the system. A formal definition of a reservation can be the following:

Definition 1 *A reservation for a resource $\mathcal{R}$ is an abstraction of a scheduling mechanism in which a task is guaranteed the use of the resource for a time Q every period P.*

In this paper, we consider only CPU reservations. A resource reservation is generally implemented by associating a pair (Q, P) to each task, where $\frac{Q}{P}$ is the fraction of processor utilization reserved to the task. When a task is activated for the first time in the system, with parameters Q_i and P_i, an admission test is run:

$$\sum_{j=1}^{n} \frac{Q_i}{P_i} \leq U_{lub}$$

where U_{lub} depends upon the underlying scheduling policy. For example, for EDF this limit is 1. If the admission test is passed, each task is guaranteed to execute for the reserved amount of time. If the test is

not passed, depending on the resource management policy, we can decide to reject the task or to schedule it with a lower QoS. In this paper, we will only address the problem of scheduling with a resource reservation algorithm. The problem of finding a resource management policies is beyond the scope of the paper: see [2, 9] for more details.

In the resource reservation framework, a hard real-time task is assigned a capacity Q greater or equal to its worst case computation time and a period P equal to the task's period. A soft real-time task, instead, can be assigned a maximum capacity Q less than the WCET, because even if some deadline is missed nothing catastrophic happens. By doing so, we can save precious resources for admitting other hard or soft real-time tasks, without compromising the schedulability of the hard real-time tasks.

A scheduling algorithm based on the resource reservation abstraction is usually implemented as follows: a remaining budget q_i is assigned to each task and it is initialized to Q_i. When the task executes, q_i is decreased accordingly. When the remaining budget goes to 0, the task is blocked until the end of the period. The budget is then replenished at the beginning of every period. Resource reservation techniques have been proposed both in fixed priority [3, 10] and in dynamic priority systems [1, 4].

Although resource reservations have been originally developed for supporting multimedia activities and for permitting to schedule real-time and non real-time activities on the same system, they have been proved to be very effective also in serving control tasks, and activities that are traditionally considered hard real-time [8]. Hence, we believe that the inclusion of a reservation technique in RTAI is important.

3 The Constant Bandwidth Server

The Constant Bandwidth Server (CBS) is a service mechanism providing CPU resource reservations on an EDF scheduler. The CBS extends the CPU reservation concept to make the algorithm work conserving (i.e., the CPU is never idle if there is at least a task ready to execute) by using dynamic priorities. The basic idea is that every task is associated a dynamic scheduling deadline by a *server*, and the ready queue is ordered according to tasks' scheduling deadlines.

The CBS algorithm is work conserving because when the budget q_i is depleted, the scheduling deadline is postponed by P, but the task is not blocked. The server assigns scheduling deadlines to jobs so that each task is reserved an amount of CPU time Q every *server period* P, according to the following rules:

Rule 1 When a new job $J_{i,j}$ arrives at time $r_{i,j}$, if $q_i \geq (d_i - r_{i,j})\frac{Q_i}{P_i}$, then a new scheduling deadline $r_{i,j} + P_i$ is generated, and q_i is recharged to the maximum value Q_i, otherwise the job is served with the last server deadline using the current budget.

Rule 2 Whenever a task τ_i, the budget q_i is decreased by the same amount.

Rule 3 When $q_i = 0$, the server budget is recharged at the maximum value Q_i and the server deadline is postponed by P_i. The EDF queue is then update accordingly.

Rule 4 the job with the earliest scheduling deadline is selected to execute, according to the EDF policy.

The CBS algorithm can correctly cope with aperiodic activations, and provides good performance, thanks to its work conserving nature [1].

In this paper, we are interested in resource reservations because of their capacity to guarantee that a reserved task will execute *at least* for a minimum fraction of the CPU time. We use this property for guaranteeing that Linux is not completely starved by RTAI real-time tasks. In fact, once the CBS algorithm has been implemented in RTAI, it is possible to schedule Linux with a dedicated CBS, instead of scheduling it in background. This approach has two advantages:

1. guarantees hard real-time tasks;
2. avoids starvation of Linux processes;
3. permits to increase the responsiveness of Linux applications.

Hence, the CBS scheduler permits to associate two parameters to Linux: the reserved bandwidth $B^L = Q/P$, representing the fraction of the CPU time that Linux processes are guaranteed to receive, and the CBS period P, which affects the responsiveness of Linux processes.

At run-time, Linux is treated exactly as any other hard real-time task: initially, it is assigned an absolute deadline $d = P$ and a remaining capacity $q = Q$. When the capacity of Linux is finished, its priority is decreased by postponing its absolute deadline to $d = d + P$, and its capacity is recharged to $q = Q$.

Since Linux is modeled as a non real-time pseudo-task, it is always backlogged: the Linux pseudo-task consists of a single job which arrives when the scheduler is initialized, and never finishes.

4 Implementation

We implemented the CBS in the RTAI UPscheduler. Our main goal was to minimize the number of modifications to the original RTAI code and the preserve the behavior of the fixed priority scheduler. In particular, the CBS tasks con coexist with fixed priority tasks: they are scheduled at a specified priority level. Currently, the CBS tasks are scheduled at the lowest priority level: therefore, a regular fixed priority task is not affected by CBS tasks. However, the priority level of the CBS tasks can be easily configured.

We added two different policies, `CBS_POL` and `EDF_POL`, which can be set with the `rt_set_sched_policy(RT_TASK *task, int policy, int rr_quantum_ns)` kernel primitive. If `policy = CBS_POL`, the `rr_quantum_ns` represents the CBS budget Q. The server period is set equal to the task period, which can be set with the `rt_make_periodic()`.

The CBS algorithm is implemented using the variables and functions already used by the round robin scheduler. We added two fields in the `RT_TASK` structure: `abs_dead` that contains the absolute deadline of the task, and that will be used for EDF scheduling; `start_time` that is used internally to compute the budget consumption. The server maximum budget Q_i is stored in `rr_quantum`, whereas the current remaining budget q_i is stored in `rr_remaining`.

In order to treat Linux as a CBS task, we added a global variable `task_min_prio`, which represents the task with the minimum priority and substitutes `rt_linux_task`. In fact, in the original RTAI scheduler, Linux was always the task with the minimum priority, while in our algorithm Linux is treated as any other CBS task.

The main modified functions are the ones used for en-queuing and dequeuing from the ready and timed queues, because they must implement the EDF policy.

Then, we modified the `RR_YIELD` to implement the budget handling (CBS rule 2) and the deadline postponing (CBS rule 3). We added the function `rt_task_assign_deadline()` that implement rule (CBS rule 1), and it is invoked by `wake_up_timed_task()`, `rt_task_resume()` and `rt_make_periodic()`. In the timer handler (`rt_timer_handler()`) we made some slight modification, because Linux can be scheduled as any other task, and hence can be selected also in the timer handler.

The budget and period for Linux can be assigned when inserting the RTAI scheduler module, through the module parameters, and by an RTAI program through functions `void set_linux_period(RTIME period_ns)` and `void set_linux_budget(RTIME budget_ns)`.

The patch to the RTAI scheduler is 444 lines long, for a total of 11 kbytes.

5 Experimental evaluation

We validated the effectiveness of the proposed solution through an extensive set of experiments. In this section, we present the most important results.

First of all, we verified the correctness of our CBS implementation by running a set of time-consuming CBS tasks and by checking that the generated schedule matches the expected one. Once verified that the scheduler is correctly implemented, we started to schedule the Linux pseudo-task using a CBS, to verify that it permits to avoid that real-time tasks starve Linux applications, and to decrease the latency experienced by Linux applications.

We run 3 time-consuming real-time tasks that overloads the system. Each one was served by a CBS with utilization equal to 0.3, for a total load of 0.9. Then we reserved a CPU bandwidth $B^L = 0.1$ to Linux and we verified that Linux processes can still execute (in particular, a shell is still active), whereas the 3 real-time tasks miss their deadline (because the system is overloaded). If the 3 tasks are scheduled using fixed priorities and Linux is scheduled in background, then Linux freezes.

To quantify the impact of the CBS on the latency of Linux processes, we performed two sets of experiments. In the first set, three periodic RTAI tasks τ_1, τ_2, and τ_3 periodically consume a fix amount of CPU time. The parameters of the 3 tasks are shown in the following Table. The total load of the three tasks is 0.8358. Since the load is less than 1, the system is schedulable with EDF.

Task	C_i	T_i
τ_1	6 msec	10 msec
τ_2	2 msec	17 msec
τ_3	3.9 msec	33 msec

Task τ_2 sends a message containing a time-stamp to a Linux process through a FIFO: we denote this time-stamp by $s_{i,j}$. The user process sleeps on the FIFO until a new message arrives, then collects the timestamps, and the instants in which they were received: we denote these instants by $r_{i,j}$. In Figures 1 and 2, we show the jitter between two consecutive arrivals minus the jitter between the two message timestamps: $(r_{i,j} - r_{i,j-1}) - (s_{i,j} - s_{i,j-1})$. This quantity is a measure of the "regularity" of the receiver: ideally, it should be equal to 0.

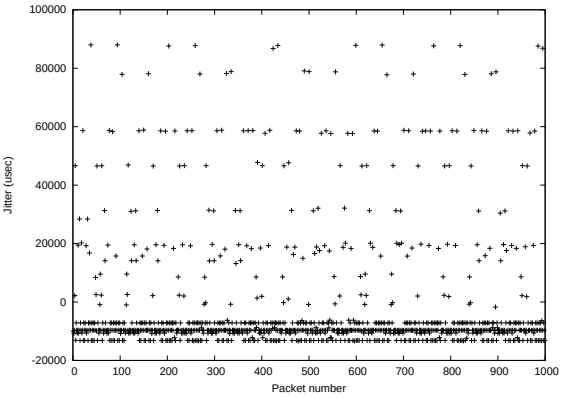

FIGURE 1: *User-Level Jitter when fixed priorities are used to schedule RTAI tasks, and Linux is scheduled in background.*

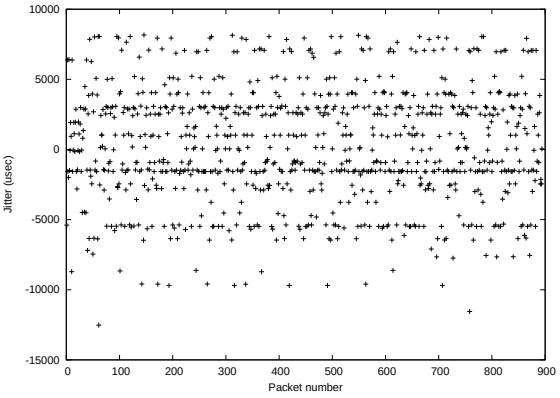

FIGURE 2: *User-Level Jitter when RTAI tasks and Linux are scheduled using CBS.*

In Figure 1, we show the results obtained by using the fixed priority scheduler, with Linux in background. As you can see, the figure is quite irregular. This happens because Linux is scheduled in background and it is difficult to know when it will be scheduled next and for how long.

In Figure 2, we show the results obtained by using our scheduler. Linux has been assigned a period $P^L = 2$ msec and a capacity $Q^L = 200$ usec, whereas each RTAI task is assigned a budget equal to its computation time. This means that Linux will be executed approximately 200 microseconds every 2 milliseconds. As a result, most of the values are clustered around 0, because Linux is scheduled more regularly.

It is important to underline that, by using our approach, all RTAI tasks finish before their next period and there is no deadline miss. If a real-time task needs a shorter deadline, it is possible to assign it a shorter CBS period, such that it is guaranteed that it will finish before its deadline.

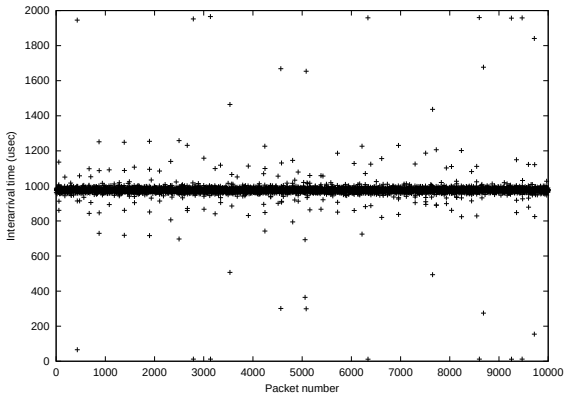

FIGURE 3: *Interarrival times of the UDP packets when no RTAI task is active*

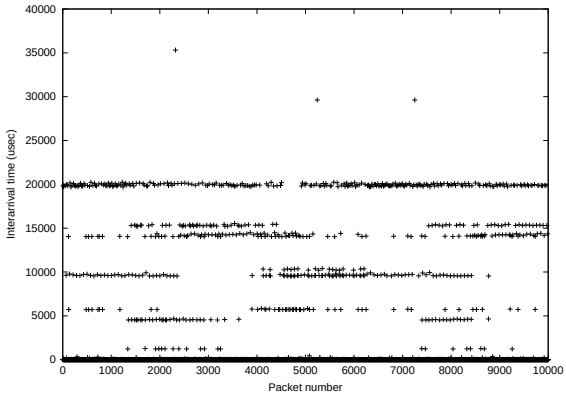

FIGURE 4: *Interarrival times of the UDP packets when RTAI tasks are scheduled using fixed priorities*

Thus, the better regularity of Linux applications comes at no cost for the real-time task. In a second set of experiments, we tried to see what happens when a Linux process needs to receive data from the network with a bounded jitter. Two computers are connected to a local network, one running pure Linux with a user process that periodically sends UDP packets; another computer running RTAI, and a Linux process receiving the packets. We verified that when no active RTAI tasks are present (Figure 4) the packets are received periodically (with the same period at which they are sent). This result matches the one that can be obtained by using a vanilla Linux kernel, or a patched kernel without any RTAI module inserted.

Then, we activated three RTAI tasks that increase the system load. Again, we raised to load to 90%, and we measured the jitter experienced by the packets. When the fixed priority scheduler is used, the results are shown in Figure 3. As you can see, the jitter

is quite large, between 0 and 20 msec. Moreover, we saw that some packet is missed. In fact, when Linux is not executing, the packets are stored in the Ethernet board buffer, and when the buffer fills-up the packets are discarded. In this experiments we used very short packets, therefore the number of packet that can fit in the hardware buffer is high: however, when some serious data-tranfer is involved, the buffer is filled-up by few packets, and the number of discarded packets can increase.
In Figure 5 we show what happens by using our algorithm. Since Linux is scheduled more regularly, the jitter is much lower, between 0 and 6 msec. Again, this comes at no cost for the real-time tasks: no RTAI task missed its deadline during the experiment.

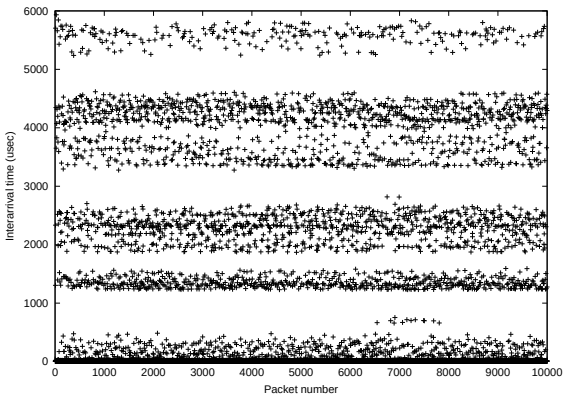

FIGURE 5: *Interarrival times of the UDP packets when RTAI tasks and Linux are scheduled by dedicated CBSs*

6 Conclusions and future work

In this paper we presented a mechanism for increasing the responsiveness of Linux processes in RTAI. The mechanism is based on a well-know paradigm for mixing hard and soft real-time activities: the resource reservation framework. In particular, we implemented the Constant Bandwidth Server (CBS) [1] in RTAI, and we assign a budget and a period to Linux, which is then scheduled as any other real-time task.
However, our implementation is not perfect, and there are problems that needs to be solved. One of the problems is the fact that the CBS algorithm is work conserving and does not suspend a task that has consumed its budget. This is very useful for reclaiming unused bandwidth: however it can cause problems to Linux. Suppose that the real-time tasks execute for much less than expected: since Linux is always active, it continuously exhaust its budget, and its deadline is postponed many times. As a consequence, soon Linux has a deadline very far in the future, and it basically scheduled in background. Hence, if the real-time tasks execute less than expected, sooner or later Linux goes in background.
To solve this problem, in the current implementation we "anticipate" the Linux scheduling deadline each time that this can be done without compromising the system's schedulability (this happens when all the other tasks do not use all their reserved time). A better solution, on which we are currently working, would be to deactivate the Linux pseudo-task when Linux schedules the idle task, and to reactivate it when an interrupt has to be delivered to Linux.

Acknoledgements

This work has been supported in part by the IST programme of the Commission of the European Communities, IST-2001-35102 (OCERA project).

References

[1] Luca Abeni and Giorgio Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, Madrid, Spain, December 1998.

[2] Chen Lee, Raj Rajkumar, John Lehoczky, and Dan Siewiorek. Pratical solutions for qos-based resource allocation. In *Proceedings of the IEEE Real Time Systems Symposium*, Madrid, Spain, December 1998.

[3] Chen Lee, Raj Rajkumar, John Lehoczky, and Dan Siewiorek. Pratical solutions for qos-based resource allocation. In *IEEE Real Time System Symposium*, Madrid, Spain, December 1998.

[4] G. Lipari. *Resource Reservation in Real-Time Systems.* PhD thesis, Scuola Superiore S.Anna, 2000.

[5] C. L. Liu and J. Layland. Scheduling alghorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1), 1973.

[6] Clifford W. Mercer, Raguanathan Rajkumar, and Hideyuki Tokuda. Applying hard real-time technology to multimedia systems. In *Workshop on the Role of Real-Time in Multimedia/Interactive Computing System*, 1993.

[7] Clifford W. Mercer, Stefan Savage, and Hideyuki Tokuda. Processor capacity reserves for multimedia operating systems. Technical Report CMU-CS-93-157, Carnegie Mellon University, Pittsburg, May 1993.

[8] Luigi Palopoli, Luca Abeni, Fabio Conticelli, Marco Di Natale, and Giorgio Buttazzo. Real-time control system analysis: An integrated approach. In *Proc. of the Real-Time Systems Symposium*, Orlando, Florida, November 2000.

[9] Ragunathan Rajkumar, Chen Lee, John Lehoczky, and Dan Siewiorek. A resource allocation model for qos management. In *Proceedings of the IEEE Real Time Systems Symposium*, 1997.

[10] Ragunathan (Raj) Rajkumar, Luca Abeni, Dionisio de Niz, Sourav Ghosh, Akihiko Miyoshi, and Saowanee Saewong. Recent developments with linux/rk. In *Proceedings of the Second Real-Time Linux Workshop*, Orlando, Florida, november 2000.

[11] Raj Rajkumar, Kanaka Juvva, Anastasio Molano, and Shuichi Oikawa. Resource kernels: A resource-centric approach to real-time and multimedia systems. In *Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking*, January 1998.

[12] Dickson Reed and Robin Fairbairns (eds.). Nemesis, the kernel – overview, May 1997.