Proceedings of the 47th Italian Conference on Factory Automation (ANIPLA 2003), Brescia, Italy, November 21-22, 2003.

An Open-Source Real-Time Kernel for Control Applications

Paolo Gai
RETIS Lab
Scuola Superiore S. Anna, Pisa
pj@gandalf.sssup.it

Giorgio Buttazzo
Department of Computer Science
University of Pavia, Italy
buttazzo@unipv.it

Abstract

This paper presents Shark, an open source real-time kernel that has been successfully used in many application domains for implementing predictable control systems. Shark is developed using an innovative composable architecture explicitly designed for simplifying the implementation, the integration and the evaluation of novel scheduling algorithms and resource management protocols. The kernel can be configured at system initialization to conform functionality to specific application requirements. Most internal kernel policies related to scheduling, aperiodic service and concurrency control are fully modular and can easily be replaced independently. Similarly, application software is independent of a particular system configuration. The system is compliant with the POSIX 1003.13 PSE52 specifications to simplify porting real-time applications to other POSIX compliant kernels.

1 Introduction

Modern control systems must often rely not only on high computational power, but also on a predictable timing behavior, to satisfy stringent specification requirements imposed by today's applications. In fact, unfortunately, high speed computing is not sufficient to enforce predictability, which can only be achieved through suitable concurrency control algorithms and resource management policies. In addition, novel hardware architectures include complex low-level mechanisms that increase the difficulty of estimating the response times to external events. In small embedded systems with limited computational power, an efficient (if not optimal) exploitation of the available resources becomes still more important.

To provide efficient and predictable support for this kind of applications the operating system running the real-time software must have specific features and mechanisms for efficient resource exploitation. For instance, a high resolution time reference and the usage of deadline based scheduling are essential for implementing optimal scheduling policies that can achieve full processor utilization [13]. These features are also important for having a precise resource accounting, needed for estimating the actual system workload, enforce temporal isolation among tasks [1] and implement adaptive strategies for maximizing the control performance[16].

When industrial applications are distributed among various computing nodes, it is important to adopt advanced networking algorithms that are able to exploit the real-time characteristics of common bus architectures, like the CAN bus or the Ethernet bus[17].

Unfortunately, most of the new approaches proposed in the research community have not been accepted by the standard closed-source commercial operating systems typically used in industrial applications. Such systems are still focused on fixed priority paradigms that often do not fit well with the new control application requirements. The main reason that prevents novel algorithms from being used in industrial applications is due the fact that integrating new mechanisms in a kernel requires a big effort if the kernel does not offer the possibility to compose the different paradigms at the CPU/network scheduling level. In classical operating systems, in fact, often a policy is not confined in a few isolated functions, but it is spread on many kernel mechanisms located at different software layers.

In this paper we present Shark (Soft and Hard Real-time Kernel), a research kernel purposely designed to allow the composability of new approaches in processor/network scheduling that widely fits with the increasing requirements of modern control applications. The kernel can be dynamically configured with the desired algorithms and applications can be developed independently of a particular system configuration. As a consequence, new modules can be added or replaced to evaluate the effects of specific scheduling policies on the performance of the control algorithms. The kernel provides the basic mechanisms for queue management and dispatching and uses one or more external configurable modules to perform scheduling decisions. These external modules can implement periodic scheduling algorithms, soft real-time task management, aperiodic servers, protocols for semaphores and mutual exclusion. Most of the common algorithms (such as Rate Monotonic, EDF, Round Robin, and so on) are already provided with the kernel, and new modules can easily be develop according to a precise interface.

An important design guideline has been to enable the usage of open-source libraries widely available for other open-source kernels. Results have been satisfactory for several applications, such as MPEG players, a set of network drivers, the MESA 3D library, a FFT library, a frame grabber, and industrial data acquisition boards. Low-level drivers for the most typical hardware resources (like network cards, graphic cards, and hard disks) are also provided using the same approach, without imposing any form of device scheduling. To avoid the implementation of a new non-standard programming interface, Shark implements the standard POSIX 1003.13 PSE52 interface [26, 27].

The rest of the paper is organized as follows. Section 2 presents the related work. Section 3 illustrates the main features of the kernel. Section 4 introduces the overall architecture of the system. Section 5 illustrates how the system allows separating the Quality of Service (QoS) specification from the algorithms used to provide the requested service. Sections 6, 7, and 8 describe the approach for achieving modularity in task scheduling, resource access, and device management, respectively. Section 9 briefly presents some experiences with the kernel, and finally Section 10 states our conclusions and future work.

2 Related Work

A large number of real-time scheduling algorithms have been proposed in the real-time literature to deal with tasks with timing constraints. The most popular ones for periodic task handling are Rate Monotonic (RM) and Earliest Deadline First (EDF), both proposed by Liu and Layland [13]. Since these algorithms were originally analyzed under very restrictive assumptions (independent tasks, fixed execution times and periods, completely preemptive scheduling, and so on), a lot of work has been devoted to extend the original analysis to more realistic situations and present new algorithms for dealing with shared resources and aperiodic activities.

On the other hand, commercial real-time operating systems, such as QNX [28] and VxWorks [23], and other free projects, like RT-Linux [11], tend to minimize the kernel non-preemptable sections, in order to make the schedule more similar to the completely preemptive model assumed by Liu and Layland in [13]. This is usually done by reducing context switch times and interrupt latencies, and increasing the kernel efficiency. However, these kernels are still based on fixed priority scheduling, hence only RM and its derivates can easily be implemented.

RT-Mach [19] is a research system that provides real-time facilities and directly implements RM, EDF, Priority Inheritance, and CPU reservations [15], presenting a wide (but fixed) range of scheduling algorithms. Another research system developed to support time-sensitive application is Nemesis [21], based on an unconventional (vertical) OS structure. While RT-Mach is a micro kernel with the goal of minimizing the non-preemptable sections in the kernel, reducing the kernel size, and implementing the OS functions into external processes (called servers), Nemesis tends to implement all the OS functions in the application code. The Nemesis kernel only implements some support to access the hardware, and provides temporal protection through a reservation approach based on EDF. If an application needs some particular scheduling algorithm, it has to implement it at the user level.

Other authors [8, 29, 30] decided to modify a conventional OS, based on a monolithic kernel approach. Some of them [30] proposed modifications in the kernel in order to introduce preemption points, schedule the interrupt handlers, and provide some limited form of device scheduling. In general, all these works introduce a new scheduling algorithm in the kernel; however, since conventional kernels provide a quantum based resource allocation and are very difficult to modify, only a few algorithms can be implemented on them easily. Proportional Share algorithms [8, 29], being based on a per-quantum CPU allocation, are expressly designed to be implemented on a conventional kernel. Another interesting technology that is growing up recently is represented by Resource Kernels (RK). An RK is a resource-centric kernel that complements the OS kernel providing support for QoS, and enabling the use of reservation techniques in traditional OSs. As an example, this technology has been applied to Linux, implementing the Linux/RK [20].

Scheduling flexibility is becoming a hot topic in OS research, and some experimental kernels are beginning to provide support for implementing different scheduling policies (although the concept of separating the policy from the mechanism is not new). For example, the L4 μkernel [12] provides a mechanism based on *preempter threads* to implement flexible scheduling (however, the base scheduling mechanism uses a fixed priority scheme). A different solution is proposed by the ExoKernel [4], which does not introduce any abstraction, but requires the resource scheduling to be performed by user applications. From one hand, *User-level scheduling* represents a good solution for implementing different scheduling algorithms on the same kernel: for example, in [9] scheduling is performed by a privileged process running in user space. On the other hand, user-level scheduling can introduce a significant overhead, and is not general enough. In fact, some scheduling algorithms require a global view of all the tasks in the system, while a user-level scheduler can only manage a fraction of all the tasks.

RED-Linux [30] tries to solve these problems using a general scheduling framework composed by a Schedule Allocator, which creates jobs characterized by some common parameters, and a Schedule Dispatcher, which schedules the jobs inserted by the Allocator. A different approach is represented by the CPU Inheritance Scheduling [6], which provides some kernel mechanisms to "inherit" CPU time from one task to another. In this way, the kernel provides only the basic mechanisms used by application tasks to implement the scheduler.

Another approach that seems promising is the approach followed by Marte OS [22], that implements a modification to the POSIX standard to allow a modular specification of scheduling algorithms using an application independent interface, that is currently submitted to the POSIX Real-Time committee for standardization.

Other solutions can be found in the literature. For example, Vino [24] is an operating system that provides a collection of mechanisms; the applications dictate the policies applied to those mechanisms and all resources are accessed through a single, common interface. Spin [2] provides a core of extensible services that allow applications to safely change the operating system's interface and implementation. Finally, Rialto [10] is an architecture supporting coexisting independent real-time and non-real-time programs; it allows multiple independently authored real-time applications with varying timing and resource requirements to dynamically coexist and cooperate to share the limited physical resources available to them.

In conclusion, all the presented works are very interesting for their efficiency, preemptability and predictability, but most of them does not directly support the specification of the constraints that are typically found on modern control systems, forcing the application to adapt to the existing kernel, rather than viceversa.

3 Kernel features

The main features of the Shark Kernel are illustrated below:

Innovative scheduling algorithms. With respect to commercial kernels, Shark supports explicit timing constraints and deadline-based scheduling. This allows an efficient implementation of optimal scheduling algorithms, such as EDF [13], capable of achieving full processor utilization, and resource reservation mechanisms, such as the CBS server [1], capable of enforcing temporal isolation among

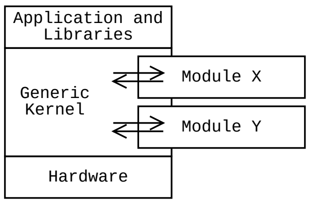

Figure 1: The Shark Architecture.

concurrent tasks. The use of EDF also allows the implementation of efficient scheduling policies for handling aperiodic activities, typical in event-driven systems.

CPU time monitoring. The kernel allows the user to monitor the CPU time consumed by each task. This information, typically unavailable in commercial systems, is essential for implementing temporal isolation among tasks and adaptive scheduling schemes (for coping with highly dynamic applications).

Composability. Shark allows the coexistence of different scheduling algorithms, which are handled according to a multi-level scheduling scheme. Hence, if a control application includes computational activities with different level of criticality and timing constraints, tasks can be partitioned into groups, and each group can be handled with the most appropriate scheduling scheme and share resource protocol.

Composability means not only modularity in the specification of the policies, but it also enables the usage of different parts of the system in a way which is independent of the others:

- applications do not have to rely on a particular scheduling algorithm, when they are implemented. Moreover, different applications can be mixed together still maintaining their temporal constraints;
- a scheduling algorithm does not have to rely on the particular implementation of other scheduling algorithms;
- resource access protocols do not have to rely on a particular implementation of other scheduling algorithms.

Open source Shark is an open source kernel distributed under the GPL license. Being open source gives us the possibility of inheriting low-level source code from existing GPLed systems for supporting common COTS hardware and powerful application libraries, thus simplifying the writing of the low-level code that is usually one of the most difficult tasks that have to be done when writing a new OS from scratch.

4 Scheduling Architecture

In order to realize independence between applications and scheduling algorithms (and between the schedulers and the kernel), Shark is based on a *Generic Kernel*, which does not implement any particular scheduling algorithm, but postpones scheduling decisions to external entities, the *scheduling modules*. In a similar fashion, the access to shared resources is coordinated by *resource modules*. A simplified scheme of the kernel architecture is depicted in Figure 1.

The Generic Kernel provides the mechanisms used by the modules to perform scheduling and resource management, thus allowing the system to abstract from the algorithms that can be implemented. The Generic Kernel simply provides the primitives without specifying any algorithm, whose implementation resides in external modules, configured at run-time (see Section 5).

Another important component of the Generic Kernel is the Job Execution Time (JET) estimator, which monitors the computation time actually consumed by each job. This is a generic mechanism, independent from the scheduling algorithms, that can be used for statistical measurements, for enforcing temporal protection[1], or for resource accounting.

The API is exported through the *Libraries*, which use the Generic Kernel to support some common hardware devices (i.e., keyboard, sound cards, network cards, graphic cards) and provide a compatibility layer with the POSIX Realtime Controller System Profile [27]. An *application* consists of a set of threads that share all the memory space (no memory protection is implemented).

Each Module provides a well defined interface to communicate with the Generic Kernel (user programs do not directly interact with the modules). The interface functions are called by the Generic Kernel to implement the kernel primitives. When modules need to interact with the hardware (for example, the timer), they can use the service calls provided by the Generic Kernel.

5 QoS Specification

A typical problem that arises when developing an industrial control application is that the application must comply with the features of the particular kernel used in the final product. This forces the designer to map the application requirements into a set of parameters, which often are not strictly related to what the user would like to do. For example, the task priority assignment strictly depends on the number of priority levels supported by the kernel and is influenced by the tasks that are present in the system. A good design methodology however, should prescind from the specific scheduling algorithm used in the system and rely only on the timing behavior expressed in the task parameters. This orthogonal approach is really useful when composing different applications together, and cannot be supported by most commercial kernels, that only export a fixed sets of scheduling algorithms.

To address the problem described above, Shark allows the user to easily specify the QoS of a task in a way independent of the real scheduling algorithm adopted in the final application. The user can decide the scheduling algorithm and the mutex access protocol at the integration phase, without modifying the application.

Independence between applications and scheduling algorithms is achieved by introducing the concept of *model*. Each task asks the system to be scheduled according to a given QoS specified by a model. In other words, a model is the entity used by Shark to separate the scheduling parameters from the QoS parameters required by each task. In this way, the kernel provides a common interface to isolate the task QoS requirements from the real scheduler implementation.

Models are descriptions of the scheduling requirements expressed by tasks. Shark provides two different kinds of models: Task Models and Resource Models. A task model expresses the QoS requirements of a task for the CPU scheduling. Requirements are specified through a set of parameters. A resource model is used to define the QoS parameters relative to a set of shared resources used by a task. For example, the resource model can be used to specify the semaphore protocol to be used for protecting critical sections (e.g., Priority Inheritance, Priority Ceiling, or Stack Resource Policy).

Models are required to make the generic kernel independent from the implemented scheduling algorithms: since the generic kernel does not implement any algorithm, it does not know how to serve a task, but invokes a service request to scheduling entities realized as external *modules*. Hence, the generic kernel does not interpret the models, but just passes them to the modules; each module, reading the common part of the model, can understand whether the task can be served or not.

Task creation works as follows: when an application issues a request to the kernel for creating a new task, it also sends the model describing the requested QoS. A kernel component passes the model to a module, selected according to an internal policy, and the module checks whether it can provide the requested QoS; if the selected module cannot serve the task, the generic kernel selects a different module. When a module accepts to manage the task described by the specified model, it converts the model's QOS parameters into the appropriate scheduling parameters. In general, a module can manage only a subset of the models, and the set of models is not limited by the kernel. This is possible

[1]The temporal protection is enforced evaluating the Job execution time and forcing the preemption of a thread if it executes more than the declared total execution time.

because the kernel does not handle the models, but it simply selects a module and passes the model to it. Currently, the kernel uses a simple strategy, according to which modules are selected based on the task models that they can handle. If they are more than one, it is a user responsibility to choose the right module that will manage the task.

6 Scheduling Modules

Scheduling Modules are used by the Generic Kernel to schedule tasks, or serve aperiodic requests using an aperiodic server. In general, the implementation of a scheduling algorithm should possibly be independent of resource access protocols, and handle only the scheduling behavior. Nevertheless, the implementation of an aperiodic server relies on the presence of another scheduling module, called the Host Module (for example, a Deferrable Server can be used if the base scheduling algorithm is RM or EDF, but not Round Robin). Such a design choice reflects the traditional approach followed in the literature, where most aperiodic servers insert their tasks directly into the scheduling queues of the base scheduling algorithm. Again, the modularity of the architecture hides this mechanism with the task models: an aperiodic server must use a task model to insert his tasks into the Host Module. In this way, the Guest Module have not to rely on the implementation of the Host Module.

The kernel distributes the tasks to the registered modules according to the task models the set of modules can handle. For this reason, the task descriptor includes an additional field (`task_level`), which points to the module that is handling the task.

When the Generic Kernel has to perform a scheduling decision, it asks the modules for the task to schedule, according to fixed priorities: first, it invokes a scheduling decision to the highest priority module, then (if the module does not manage any task ready to run), it asks the next high priority module, and so on. In this way, each module manages its private ready task list, and the Generic Kernel schedules the first task of the highest priority non empty module's queue. Due to lack of space, we cannot describe in detail the functions exported by a scheduling module. An interested reader can find plenty of documentation on the Shark website, that explains how to create, initialize and use a scheduling module.

7 Shared Resource Access Protocols

Shark is based on a shared memory programming paradigm, so communication among tasks is performed by accessing shared buffers. In this case, tasks that concurrently access the same shared resource must be synchronized through *mutual exclusion*: real-time theory [25] teaches that mutual exclusion through semaphores is prone to *priority inversion*. In order to avoid or limit priority inversion, suitable shared resource access protocols have to be used.

As for scheduling, Shark achieves modularity also in the implementation of shared resource access protocols. Resource modules are used to make resource protocols modular and almost independent from the scheduling policy and from the others resource protocols. Each resource module exports a common interface, similar to the one provided by POSIX for mutexes, and implements a specific resource access protocol. A task may also require to use a specified protocol through a resource model.

Some protocols (like Priority Inheritance or Priority Ceiling), directly interact with the scheduler (since a low-priority task can inherit the priority from a high-priority task), making the protocol dependent on the particular scheduling algorithm. Although a solution based on a direct interaction between the scheduler and the resource protocol is efficient in terms of runtime overhead, it limits the full modularity of the kernel, preventing the substitution of a scheduling algorithm with another one handling the same task models (for example, Rate Monotonic could be replaced by the more general Deadline Monotonic algorithm).

To achieve complete modularity, the Shark Generic Kernel supports a generic priority inheritance mechanism independent from the scheduling modules. Such a mechanism is based on the concept of *shadow tasks*. A shadow task is a task that is scheduled in place on another task chosen by the scheduler. When a task is blocked by the protocol, it is kept in the ready queue, and a shadow task

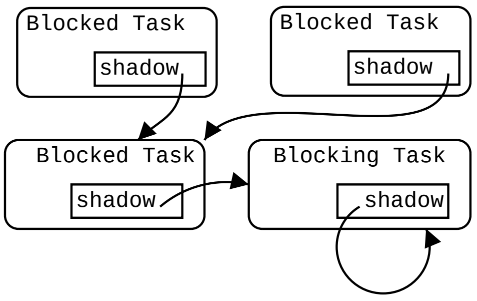

Figure 2: The shadow task mechanism.

is binded to it; when the blocked task becomes the first task in the ready queue, its binded shadow task is scheduled instead. In this way, the shadow task "inherits" the priority of the blocked task.

To implement this solution, a new field `shadow` is added to the generic part of the task descriptor. This field points to the shadow task. Initially, the shadow field is equal to the task ID (no substitution). When the task blocks, the shadow field is set to the task ID of the blocking task, or to the task that must inherit the blocked task priority. In general, a graph can grow from a blocking task (see Figure 2). In this way, when the blocked task is scheduled, the blocking (shadow) task is scheduled, thus allowing the schedulers to abstract from the resource protocols. This approach has also the benefit of allowing a classical deadlock detection strategy: cycles have to be searched in the shadow graph when a `shadow` field is set.

Using this approach a large number of shared resources protocols can be implemented in a way independent from the scheduling implementation. This independence is very important, since it allows to choose the mutual exclusion protocol and to change it in a second phase when integrating all the components that compose the final application.

8 Device management

One of the major problems when developing new operating systems is the lack of device drivers, that usually prevent open source kernels without a great community to be effectively used on COTS hardware.

The solution we adopted in our kernel is to inherit the source code for the most important drivers from other free OSs (namely Flux OSKit, Linux, FreeBSD and others) and to adapt their low level code using some glue code that remaps the other system calls to the Generic Kernel interface. In this way, it is possible to support almost all the devices supported by the free OS from which the driver is inherited (note that we can inherit code from Linux, that supports most of the current PC hardware).

In our experience, the driver code does not need special design techniques to be used in a real-time environment, thus inheriting legacy code at the device driver level does not cause major problems. We currently ported many different drivers using this technique. As an example, we implemented the glue code for using the Linux network drivers, and successfully inherited code for all the 3COM, NE network cards, and others. We also ported the graphics drivers from the SVGA project, the MESA 3D libraries, some MPEG decoders, FFT routines, framegrabbers and other libraries.

9 Experiences and successful results

Shark has been used in the context of university research for a few years, and it has got acceptance in a few universities as a kernel for testing the behavior of real-time scheduling algorithms, both in operating systems courses and in research laboratories for for implementing advanced control applications. For example, the following systems are currently controlled using Shark:

Inverted Pendulum Shark has been used to experiment advanced control algorithms to control a benchmark application of an inverted pendulum [16]. In the experiments, the pendulum was controlled by tracking the carrier position using a video camera. The tracking algorithm was scheduled using a resource reservation techniques due to its variable execution time.

Exapod A six-legged walking machine has been developed at the Robotic Lab of the University of Pavia [14]. Each leg is actuated by two servomotors, driven by PWM signals generated by a PIC subsystem connected to the main host using an RS232. The algorithm for coordinating the robot locomotion runs on a PC under Shark, which is used to acquire a joystick for modulating the robot trajectory, generate right and left motor set points, and guarantee a predictable timing behavior.

Web based ball balancing Shark has also been used in the development of a virtual laboratory environment aimed at the interaction with a real-time-system for running a two-degree-of-freedom plate balancing device for keeping a ball in a desired position. A remote user can connect through a standard Linux web server that interacts with the Shark target using a graphical interface [7].

Pointing system A two-degrees-of-freedom laser pointer is actuated by two servomotors to follow the trajectory of a moving target, based on the images produces by a CCD camera. The calibration of the visual system is performed using a back propagation neural network that is trained to convert coordinates in the image plane into motor rotations. A Kalman filter is used to predict future object positions [5].

Distributed networks A predictable communication protocol over the Ethernet, FTT Ethernet [17, 18], has been developed at the University of Aveiro using the Shark Kernel. The FTT protocol allows the achievement of real-time performances on COTS hardware, like Ethernet devices, and can schedule packets through the network with fault tolerance features and online reconfigurability. Another ongoing application under development at the University of Illinois involves a support for wireless networks and protocols for adaptive reconfiguration of the communication channel.

Elastic and adaptive schedulers The kernel has been used to investigate flexible scheduling algorithms that can react to overload conditions by changing the performance requirements of the application [3]. Load is typically controlled by changing period configurations or computation times, if multiple versions (with different service time and quality) are provided for each task.

Goalkeeper At the Robotic Lab of the University of Pavia the kernel has been used to implement a robot goalkeeper, where visual information is used to predict the trajectory of a ball and control the position of a moving cart to catch it.

Value scheduler competition In the real-time course at the Malardalen University in Sweden the kernel is used to implement a programming contest in which students are requested to design a best effort scheduling algorithm to handle overload conditions caused by aperiodic tasks (with given values) generated according to an unknown arrival pattern. The goal of the competition is to maximize the cumulative value of the tasks that complete execution before their deadline.

To simplify the development of real-time applications, a number of different modules have been implemented on Shark, including real-time scheduling algorithms, aperiodic servers and shared resource access protocols on real applications. A list of the implemented modules is reported in Table 1.

10 Conclusions

In this paper we presented Shark, a dynamic configurable research kernel architecture designed for supporting a simple implementation, integration and composition of control and industrial applications. The kernel is fully modular in terms of scheduling policies, aperiodic servers, and concurrency

• Scheduling modules
Earliest Deadline First, Rate Monotonic, POSIX scheduler, Round Robin, Slot Shifting, Value schedulers, Elastic Tasks, Feedback scheduling, Hierarchical scheduling
• Aperiodic servers
Polling Server, Deferrable Server, Sporadic Server, Total Bandwidth Server, Constant Bandwidth Server, CBS-FT, CASH
• Shared resource access protocols
Classic blocking protocol, non-preemptive protocol, Priority Inheritance, Priority Ceiling, Stack Resource Policy

Table 1: Modules Implemented in the Shark Kernel.

control protocols. The kernel has been successfully used in many control applications showing good performance results and flexibility in its usage.

The compliance with the POSIX standard allows to recycle existing code written and developed for other kernels, allowing testing it under novel scheduling algorithms. The Kernel is distributed under the GPL license, and it can be found at the URL `http://shark.sssup.it`.

References

[1] L. Abeni. Server mechanisms for multimedia applications. Technical Report RETIS TR98-01, Scuola Superiore S. Anna, 1998.

[2] B. Bershad, S. Savage, P. Pardyak, E. Sirer, M. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety and performance in the spin operating system, 1995.

[3] G. Buttazzo and L. Abeni. Adaptive workload management through elastic scheduling. *Real-Time Systems*, Vol. 23(n. 1-2):pages 7–24, July-September 2002.

[4] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 251–266, Copper Mountain Resort, Colorado, December 1995.

[5] T. Facchinetti. Realizzazione di un sistema robotico per il tiro a volo di mersagli mobili. Master's thesis, Università degli studi di Pavia, 2000.

[6] B. Ford and S. Susarla. Cpu inheritance scheduling. In *Proceedings of OSDI*, October 1996.

[7] C. Giuseppe and G. Buttazzo. A virtual laboratory environment for real-time experiments. In *Proceedings of the 5th SICICA Conference*, Aveiro, Portugal, July 2003.

[8] P. Goyal, X. Guo, and H. M. Vin. A hierarchical cpu scheduler for multimedia operating systems. In *2nd OSDI Symposium*, October 1996.

[9] H. hua Chu and K. Nahrstedt. CPU service classes for multimedia applications. In *Proceedings of the IEEE International Conference on Mutimedia Computing and Systems*, Florence, Italy, June 1999.

[10] M. Jones, P. Leach, R. Draves, and J. Barrera. Support for user-centric modular real-time resource management in the rialto operating system, 1995.

[11] F. Labs. Real-time linux home page. http://www.rtlinux.org/.

[12] J. Liedtke. On μ-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, Copper Mountain Resort, Colorado, December 1995.

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

[14] M. Mauro and G. Buttazzo. A computer architecture for small walking robots. In *IASTED Robotic Application 2003*, Salzburg, AT, July 2003.

[15] C. W. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves for multimedia operating systems. Technical Report CMU-CS-93-157, Carnegie Mellon University, Pittsburg, May 1993.

[16] L. Palopoli, L. Abeni, G. Bolognini, B. Allotta, and F. Conticelli. Novel scheduling policies in real-time multithread control system design. *Control Engineering Practice*, 10(10):1095–1110, November 2002.

[17] P. Pedreiras, L. Almeida, and P. Gai. The ftt-ethernet protocol: Merging flexibility, timeliness and efficiency. In *Proceedings of the 14th IEEE Euromicro Conference in Real-Time Systems*, Vienna, June 2002.

[18] P. Pedreiras, L. Almeida, P. Gai, and G. Buttazzo. Ftt-ethernet: A platform to implement the elastic task model over message streams. In *Proceedings of the WCFS '02*, Vasteras, August 2002.

[19] R. Rajkumar, K. Juvva, A. Molano, and S. 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.

[20] R. R. Rajkumar, L. Abeni, D. de Niz, S. Ghosh, A. Miyoshi, and S. Saewong. Recent developments with linux/rk. In *Proc. of the Second Real-Time Linux Workshop*, Orlando, Florida, november 2000.

[21] D. Reed and R. F. (eds.). Nemesis, the kernel – overview, May 1997.

[22] M. A. Rivas and M. G. Harbour. Posix-compatible application-defined scheduling in marte os. In *Proceedings of 14th Euromicro Conference on Real-Time Systems*, Vienna, June 2002.

[23] W. River. Vxworks 5.4. http://www.wrs.com /products/html/vxwks54.html.

[24] M. Seltzer, Y. Endo, C. Small, and K. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proc. of the Symposium on Operating System Design and Implementation (OSDI II)*, 1995.

[25] L. Sha, R. Rajkumar, and john P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE transaction on computers*, 39(9), september 1990.

[26] I. C. Society, editor. *International Standard ISO/IEC 9945-1: 1996 (E) - IEEE Std 1003.1, 1996 Edition - Information technology - Portable Operating System Interface (POSIX)*. IEEE, 1996.

[27] I. C. Society, editor. *IEEE Standard for Information Technology - Standardized Application Environment Profile - POSIX Realtime Application Support (AEP)*. IEEE, 1998.

[28] Q. Software Systems Ltd. Qnx neutrino real-time os. http://www.qnx.com/products/os/neutrino.html.

[29] C. A. Waldspurger and W. E. Weihl. Stride scheduling: Deterministic proportional-share resource mangement. Technical Report MIT/LCS/TM-528, Massachusetts Institute of Technology, June 1995.

[30] Y. Wang and K. Lin. Implementing a general real-time scheduling framework in the red-linux real-time kernel. In *Proceedings of IEEE Real-Time Systems Symposium*, Phoenix, December 1999.