A New Kernel Approach for Modular Real-Time systems Development

Paolo Gai, Luca Abeni, Massimiliano Giorgi
RETIS Lab
Scuola Superiore S. Anna, Pisa
{pj,luca,massy}@hartik.sssup.it

Giorgio Buttazzo
University of Pavia (Italy)
INFM - Pavia research unit
buttazzo@unipv.it

Abstract

This paper presents a dynamic configurable kernel architecture designed for supporting a simple implementation, integration and evaluation of scheduling algorithms. The main goal of the proposed architecture is to provide a platform for fast prototyping scheduling algorithms both for the CPU and for the devices. The kernel is fully modular in terms of scheduling policies, aperiodic servers, and concurrency control protocols, allowing applications to be developed independently from a particular system configuration. Finally, the system is compliant with the POSIX 1003.13 PSE52 specifications to simplify porting of application code developed for other POSIX compliant kernels.

1. Introduction

Real-time computing is required in many application domains, ranging from embedded process control to multimedia systems. Each application has peculiar characteristics in terms of timing constraints and computational requirements (such as periodicity, criticality of the deadlines, tolerance to jitter, and so on). For this reason, a lot of different scheduling algorithms and resource allocation protocols have been proposed to conform to such different application demands, from the classical fixed or dynamic priority allocation schemes to adaptive or feedback-based systems.

However, most of the new approaches have been only theoretically analyzed, and sometimes evaluated using a scheduling simulator. In this case, the algorithm performance is not evaluated on real examples, but only on a synthetic workload. This choice is often dictated from the fact that writing a kernel from scratch every time a new scheduling algorithm is proposed would be unrealistic and would not offer the availability of meaningful applications. A more effective approach is to modify an existing kernel (such as Linux), since most of the existing applications and device drivers written for the host OS can be used in a straightforward fashion. On the other hand, a general purpose kernel is designed aiming at specific goals and generally its architecture is not modular enough for replacing or modifying the scheduling policy. Moreover, classical OSs do not allow to easily define a scheduling policy for resources other than the CPU and this poses a further limitation for testing novel research solutions. This is mainly due to the fact that the classical OS structure does not permit a precise *device scheduling* (due to problems involving resource contention, priority inversion, interrupt accounting, long non-preemptive sections, and so on).

In this paper we present S.Ha.R.K. (Soft and Hard Real-time Kernel), a research kernel purposely designed to help the implementation and testing of new scheduling algorithms, both for the CPU and for other resources. The kernel can be used to perform early validation of the scheduling algorithms produced in the research labs, and to show the application of real-time scheduling in real-time systems courses. These goals are fulfilled by making a trade off between simplicity and flexibility of the programming interface on one hand and efficiency on the other. This approach allows a developer to focus his/her attention on the real algorithmic issues, thus saving significant time in the implementation of new solutions. Another important design guideline is the use of standard naming conventions for the support libraries in order to ease the porting of meaningful applications written for other platforms. The results have been satisfactory for applications such as an MPEG player, a set of network drivers and a FFT library.

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 task management through real-time servers, semaphore protocols, and resource management policies. The modules implementing the most common algorithms (such as RM, EDF, Round Robin, and so on) are already provided, and it is easy to develop new modules. Each new module can be created as a set of functions that *abstract* from the implementation of the other scheduling modules and from the resource handling functions. Also the applications can be developed independently from a particular system configuration, so that new modules can be added or replaced to evaluate the effects of specific scheduling policies in terms of predictability, overhead, and performance. Low-level drivers for the most typical hardware resources (like network cards, graphic cards, and hard disks) are also provided, without imposing any form of device scheduling. In this way, device scheduling can be implemented by the user to test new solutions. To avoid the implementation of a new non-standard programming interface, which would discourage people from using the kernel, S.Ha.R.K. implements the standard POSIX 1003.13 PSE52 interface [21, 22].

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

2. Related Work

A big number of real-time scheduling algorithms have been proposed in literature to deal with timing constraints, starting from the classical Rate Monotonic (RM) or Earliest Deadline First (EDF) algorithms [12]. Since the analysis originally proposed for these algorithms was performed under very restrictive assumptions (independent tasks, fixed execution times and periods, completely preemptive scheduling, and so on), a lot of successive papers have been devoted to extend the original analysis and present new algorithms for dealing with shared resources or aperiodic requests.

On the other hand, commercial real-time operating systems, such as QNX [23] and VX Works [18], and other free projects, like RT-Linux [10], 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 [12]. This is usually done by reducing context switch and interrupt handling times 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 [15] is a research system that provides real-time facilities and directly implements RM, EDF, Priority Inheritance, and CPU reservations [13], presenting a wide (but fixed) range of scheduling algorithms. Another research system developed to support time-sensitive application is Nemesis [17], 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 [7, 24, 25] decided to modify a conventional OS, based on a monolithic kernel approach. Some of them [25] 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 [7, 24], 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 [16].

In any case, 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 [11] 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 [8] 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 manage only a fraction of all the tasks).

RED-Linux [25] 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.

Finally, in the literature other solutions can be found, like for example Vino[19] (that is an operating system which provides a collection of mechanisms, and the applications dictate the policies applied to those mechanisms; all resources are accessed through a single, common interface), Spin [1] (that provides a core of extensible services, that allow applications to safely change the operating system's interface and implementation) and Rialto [9] (that 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 none of them are so flexible and simple to allow the user to fast prototype a new scheduling algorithm without carrying on limitations due to a complex system interface.

3. Scheduling Architecture

In order to realize independence between applications and scheduling algorithms (and between the schedulers and the kernel), S.Ha.R.K. 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.

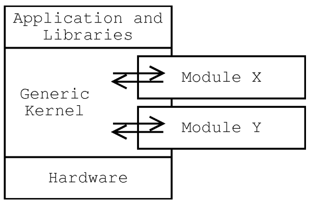

Figure 1. The S.Ha.R.K. Architecture.

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 runtime with the support of the *Model Mapper* (see Section 4).

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 (see Section 7).

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 [22]. An *application* consists of a set of threads that share all the memory space (no memory protection is implemented).

Each Module consists of a set of data and functions used for implementing a specific algorithm, whose implementation is independent from the one of the other modules in the system, thus realizing a trade-off between user-level and in-kernel schedulers. In this way, many different module configurations are possible. For example, a Polling Server can either work with a RM or an EDF scheduling Module without any modification.

Currently, S.Ha.R.K. provides two basic kind of

[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.

modules:

- modules that implement scheduling algorithms and aperiodic service policies (Scheduling Modules);
- modules that manage shared (hardware or software) resources (Resource Modules);

All resource access protocols, such as Priority Inheritance, are implemented as a mutex module whose interface is derived from the resource module interface. A POSIX mutex interface is also provided on top of the implemented protocols.

Each type of 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.

4. QoS Specification

One of the goals of the S.Ha.R.K. Kernel is to allow the user to easily implement and test novel scheduling algorithms. In particular, the kernel has been designed in order to achieve independence between the kernel mechanisms and the scheduling policies for tasks and resource management, allow to configure the system at run-time by specifying the algorithms to be used for task scheduling and resource access and achieve independence between applications and scheduling algorithms.

These requirements are useful when different algorithms need to be compared and tested on the same application. Such a module independence also allows the user to configure and test applications without recompiling them (only relinking is needed).

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 S.Ha.R.K. 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. S.Ha.R.K. 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 SRP).

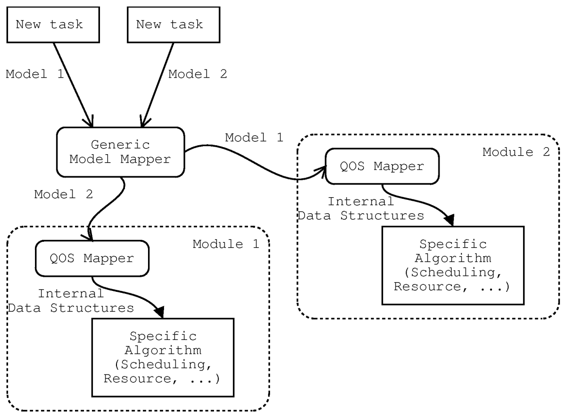

Figure 2. The interaction between the Model Mapper and the QOS Mapper.

Each task is characterized by a single mandatory QoS parameter, the *task criticality* (hard, soft, firm, non real-time, and so on). This parameter belongs to the common part of the task model, together with a *model identifier* and some other parameters, such as the stack size.

Each task model is implemented as a C structure, in which the first field is the model identifier, the following fields are the mandatory parameters, and the last field is a sequence of bytes containing the model-dependent parameters, that only the specific module can interpret. Resource models are completely generic and depend on the resource they describe: the only mandatory parameter is the model identifier.

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 (see Figure 2): 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, namely the *model mapper*, 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 model mapper 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. Such a conversion is performed by a module component, called the *QoS Mapper*. 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 because the kernel does not handle the models, but it simply passes them to the Model Mapper, that selects a module and passes the model to it. Currently, the Model Mapper 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.

5. 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 Model Mapper 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.

The interface functions provided by a scheduling module can be grouped in three classes: Level Calls, Task Calls and Guest Calls.

6. Shared Resource Access Protocols

S.Ha.R.K. 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 [20] 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, S.Ha.R.K. 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 S.Ha.R.K. 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 is binded to it; when the blocked task becomes the first task in the ready queue, its binded shadow task is

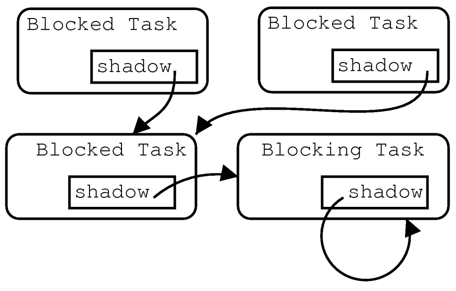

Figure 3. The shadow task mechanism.

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 3). 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 one of the objective of the architecture is to allow a simple implementation of a scheduling algorithm that tries to be independent from other implementations and from the shared resource policies. The classical approach, however, can also be pursued, but obviously it not reach the independence of the latter method.

7. Device Management

One of the goals of the S.Ha.R.K. design is to allow *device scheduling*, permitting to extend the scheduling techniques applied to the CPU to all the other hardware resources. To do that, device management has to be performed without affecting the other system activities guarantee. In particular, an hardware device has o be shared among applications respecting the real-time requirements expressed by the Resource Models.

Current research [17, 15] in real-time and multimedia operating systems suggests that QoS guarantee on hardware resource access, can be better achieved if device management is performed outside the kernel. This is important for ensuring that the device management code will not steal execution time to the application code. If, on the other hand, devices are managed in the traditional way, as in monolithic kernel, some problem like the receiving livelock [14] can happen.

In order to perform device scheduling allowing a real-time management and a precise resource accounting, S.Ha.R.K. makes a distinction between *device drivers* and *device managers*. A device driver is responsible for accessing hardware resources at a low level and it is generally composed of code accessing I/O ports and memory in response to user requests or hardware interrupts. Thus, device drivers are the hardware dependent part of the device management code. A resource manager uses the driver code to access the hardware and implements some real-time device management and resource allocation strategy. In this way, it is possible to perform any form of device scheduling.

7.1. Device Drivers

Device drivers are the hardware dependent part of the device management code, implementing the routines necessary to perform low-level accesses to the devices. Depending on the implemented management scheme, the driver code can be embedded in the Generic Kernel (obtaining a solution similar to that used in classical Unix systems), in a system or user dedicated thread (as in multithreaded kernels, like Solaris, or in μkernel architectures like Mach or L4), or in the user-level application code (as in the ExoKernel architectures or in the Nemesis vertical-structured kernel).

In our solution, the driver code can be inherited from other free OSs, and can be compiled in the S.Ha.R.K. environment using some glue code, remapping the other system calls to the Generic Kernel interface. In this way, it is possible to support 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. As an example, we implemented the glue code for using the Linux network drivers, and successfully inherited code for all the 3COM and NE network cards.

7.2. Device Managers

The device manager is responsible for using the driver level to share a device among all the applications. The manager is hardware independent and must only perform device scheduling, taking device management costs into account to provide some form of guarantee on hardware accesses.

For this purpose, the manager can be implemented as a dedicated thread, or as an application code: the first solution ensures that the device management will not influence the other system's activities (if an aperiodic server or an isolation technique is used to serve the device management task), whereas the second solution permits a better precision in accounting the CPU time used by the device manager to the application using the hardware resource.

As an example, we describe the S.Ha.R.K. file system: the device driver is a low level IDE code accessing the hard disk (exporting services such as block-read and block-write), while the manager is composed by the FAT file system code and by a disk scheduler. The scheduler selects the I/O requests to pass to the driver, according to some well known disk scheduling policy (such as SCAN), or to some real-time algorithm (derived from EDF). The device scheduling algorithm can be specified at the initialization of the device manager, and each task can specify its desired disk QoS using a resource model.

Currently, S.Ha.R.K. implements separation between resource drivers and managers, and the most important device drivers have been implemented (independently from the device scheduling algorithms) either inheriting them from other operating systems or writing them from scratch. We are currently working on the definition of a generic architecture for the device managers, to allow an easy implementation of device schedulers in proper independent modules.

8. Experiences and successful results

A number of different modules have been implemented on S.Ha.R.K. for experimenting 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.

A student of the OS course at the University of Pisa implemented the Sporadic Server (either static and dynamic) in about 2 weeks, as a project for the exam;

A visiting PhD student from Malardalens University (Sweden) implemented the Slot Shifting [5] algorithm in 2 weeks. Note that S.Ha.R.K. was designed without any knowledge of the Slot Shifting algorithm;

- Scheduling modules

  Earliest Deadline First, Rate Monotonic, POSIX scheduler, Round Robin, Slot Shifting

- 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 S.Ha.R.K. Kernel.

A group of students from Malardalens University (Sweden) successfully implemented a value-based scheduler as a scheduling module. The training period has been a few days to explain the interface to be used and a few hours to implement the scheduler;

A student from the University of Pavia implemented a new reservation algorithm for handling fault-tolerant real-time tasks in about 20 days;

A PhD student implemented the CBS-hd algorithm [3], and its evolution, the CASH algorithm [2], for testing performance results and overhead on a real kernel setting.

Note that when the kernel was developed, CBS-hd and CASH did not exist yet.

9. Conclusions and Future Work

In this paper we presented S.Ha.R.K., a dynamic configurable research kernel architecture designed for supporting a simple implementation, integration and comparison of scheduling algorithms. The kernel is fully modular in terms of scheduling policies, aperiodic servers, and concurrency control protocols.

Modularity has been achieved by a trade-off between efficiency and flexibility and by properly partitioning the system activities between a generic kernel and a set of modules which can be registered at initialization time to configure the kernel according to the specific application requirements.

The flexibility of this approach allows important benefits from three different points of view. At the user level, an application can be developed independently from a particular configuration of the system. At the research level, new modules can be added and tested on the same application to evaluate the impact in terms of predictability, overhead, and performance (in a way independent from the others modules). The development of new Modules is so simple that they can be implemented also by an undergraduate student from an Operating System course. Finally, the compliance with the POSIX standard allows to recycle existing code written and developed for other kernels, allowing in this way the testing of the newly created scheduling algorithms.

A lot of work has still to be done: in particular, we plan to extend the QoS specification provided by the models, and implement and test more complex strategies for the Model Mapper. We are currently working on the definition of a more generic structure (similar to the one used for the CPU scheduling) for performing device management. The Kernel is distributed under the GPL license, and it can be found at the URL `http://shark.sssup.it`.

Acknowledgements The authors would like to thank Luigi Palopoli and Giuseppe Lipari for their valuable suggestions to improve the paper.

References

[1] 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.

[2] M. Caccamo, G. Buttazzo, and L. Sha. Capacity sharing for overrun control. In *Proceedings of the IEEE Euromicro Conference on Real-Time*, Orlando, Florida, December 2000.

[3] M. Caccamo, G. Buttazzo, and L. Sha. Elastic feedback control. In *Proc. of the IEEE Euromicro Conference on Real-Time*, Stocolm, Sweden, June 2000.

[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] G. Fohler. Joint scheduling of distributed complex periodic and hard aperiodic tasks in statically scheduled systems. In *Proceedings of the 16th Real Time System Symposium, Pisa, Italy*, December 1995.

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

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

[8] 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.

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

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

[11] 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.

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

[13] 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.

[14] J. Mogul and K. Ramakrishnan. Eliminating receive livelock in an interuupt-driven kernel. *ACM Transactions on Computer Systems*, 15(3):217–252, August 1997.

[15] 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.

[16] 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.

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

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

[19] 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.

[20] 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.

[21] 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.

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

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

[24] 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.

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