Support for Dynamic QoS in the HARTIK Kernel

Luca Abeni
RETIS Lab
Scuola Superiore S. Anna, Pisa
luca@sssup.it
Phone: +39 050 883451
Fax: +39 050 883215

Giorgio Buttazzo
Dept. of Computer Science
University of Pavia
buttazzo@unipv.it
Phone: +39 0382 505755
Fax: +39 0382 505373

Abstract

HARTIK is a real-time kernel originally developed for supporting complex robot applications and hard real-time control applications, in general. However, in the last years, the kernel has been modified to introduce support for soft real-time and multimedia applications. Major changes include the development of a new service mechanism, which provides the abstraction of temporal protection, and the implementation of a QoS Manager capable of adapting the reserved resources to the actual tasks' requirements.

In this paper we report our experience with the new support for dynamic QoS applications, showing how our theoretical solutions have been applied on a real, stable, and freely available kernel.

1. Historical Background

In the first '90s, most of the research on real-time systems was focused on achieving highly predictable behavior from real-time kernels. The main goal was to manage tasks with explicit timing constraints, bound priority inversion, provide off-line guarantee of critical tasks, and also allow a certain degree of flexibility in integrating hard real-time tasks with soft real-time and non real-time activities. Unfortunately, commercial operating systems did not fulfill such characteristics, and research real-time kernels [13, 22, 21, 7] only partially conformed with the desired requirements.

The lack of flexible real-time kernels combined with the desire of experimenting novel scheduling approaches on concrete control applications were the main driving forces that pushed us to start the development of a new HArd Real-TIme Kernel (HARTIK), aimed at providing flexibility in expressing timing constraints, explicit management of periodic activities, preemptive scheduling and on-line guarantee, integration of hard and soft real-time tasks, and predictable interrupt management.

The first version of the HARTIK kernel, released in 1993 [5], implemented all these features and was targeted to Motorola 68000 machines and to Intel 80x86 based PCs running in real mode. As the name says, the kernel was aimed at managing hard real-time control applications and it resulted very effective for supporting robotics applications characterized by hard periodic activities running at different rates [6].

In the following years, the kernel evolved according to the research trends identified by the real-time community and, in 1997, the HARTIK 3.0 [11] kernel was released. Besides supporting some of the most advanced scheduling techniques based on EDF (such as the Total Bandwidth Server [20] for efficient aperiodic tasks management), the new version was characterized by a portable architecture, supporting the 80x86 Protected Mode and providing drivers for the most common peripherals. Moreover, HARTIK 3.0 was freely available (under GPL) and could be compiled using the free gcc compiler.

Using the 3.0 version, we started working on different application domains, such as multimedia and monitoring systems, characterized by periodic and semi-periodic data streams having soft timing constraints, but individual Quality of Service (QoS) requirements.

Working on this new area, we realized that classical real-time techniques are not very suited for supporting such kind of applications, but have to be significantly modified to deal with more dynamical and unpredictable execution environments.

To support this new kind of applications, we provided specific support for soft real-time, modifying the kernel and introducing a *temporal isolation* mechanism. Even introducing a bandwidth reservation mechanism, the system's performance resulted to be strongly dependent on the specific bandwidth assigned to each task, which could differ from the bandwidth actually requested. If a task was assigned less bandwidth than the requested one, the isolation mechanism slowed the task down, thus degrading its performance; if the assigned bandwidth was greater than the requested one, system's resources were wasted.

The problem of assigning tasks a correct bandwidth has been solved by introducing an on-line feedback mechanism to adapt the scheduling parameters to the actual task requirements. This novel mechanism, denoted as the *Adaptive Bandwidth Reservation* approach, has been implemented through a QoS management task realized as a user process (according to a middleware approach).

In this paper, we report our experience with the HARTIK kernel in the development of the dynamic QoS support.The paper is organized as follows: Section 2 presents some related work; Section 3 explains the principles that driven the design and the development of the kernel, and gives a general overview of the kernel; Section 4 presents the support for soft real-time and for fixed and dynamic QoS control; Section 5 reports our experimental experience; and Section 6 presents our conclusions and future directions.

2. Related Work

When HARTIK was developed, the existing real-time research kernels [13, 22, 21, 7, 23] were not suited for supporting highly dynamic soft real-time applications having highly variable computational demand. The approach most similar to ours was proposed in Real-Time Mach [23] (based on a μkernel approach), whose evolution [17, 15] has been similar to the one followed by HARTIK [1, 3].

In the last years, a number of new real-time kernels appeared either on the market (as QNX Neutrino [19] and VxWorks [18]) as well as in the research laboratories [12, 10]. QNX Neutrino is a real-time μkernel, based on fixed priorities, aimed at reducing the kernel nonpreemptable sections and minimizing the interrupt response time. The solution proposed by Neutrino to handle devices (based on out-of-kernel tasks) is similar to the one implemented in HARTIK. VxWorks is another very popular real-time kernel that provides a single address space for all the tasks (this solution is also similar to the HARTIK structure),

The Nemesis [12] and DROPS [10] operating systems use real-time techniques to support multimedia applications, proposing unconventional structures (the vertical kernel structure, or the multi-server OS design) to allow precise resource accounting and efficient interrupt handling. All these OSs and kernels are based on a new design, and not on the modification of a conventional kernel.

Other authors [9, 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 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 [9, 24], being based on a per-quantum CPU allocation, are expressly designed to be implemented on a conventional kernel.

3. Design objectives

The HARTIK kernel was originally developed according to the following design goals:

- Simplicity: the kernel is as *simple* as possible, in order to be easily modified, *easy to use*, and *small*, in terms of memory requirements.
- Theoretical soundness: all scheduling algorithms implemented in the kernel can be analyzed to verify the feasibility of the task set within the required timing constraints. A concurrency control protocol is used to bound priority inversions in accessing shared resources, and non-blocking state message buffers are available for time bounded inter-thread communication.
- Predictability: other sources of unpredictability are reduced as much as possible.
- Programming Interface: the kernel provides an Application Program Interface (API) that can be directly mapped to the real-time task model, so that the user does not have to worry about periodic activations, deadlines management, and so on.

3.1. Kernel Structure

The HARTIK kernel is implemented as a static library that can be linked with the user application program. In the 80x86 protected mode version, the executable file is an ELF or a COFF binary, which can be executed by a custom extender (X) running under DOS. The library and the program can be compiled and linked using the standard gcc environment (under DOS or Linux).

When the program starts, the standard `main()` function is called in a 32 bit monothreaded environment. This function can then initialize the real-time multiprogramming environment and create other real-time or non real-time threads. Since the standard C library provided with the compiler is not reentrant, we decided to provide a custom reduced version of the C library, which is multithread safe and predictable.

The concurrency model provided by HARTIK is *multithreading*: a program is composed of a set of concurrently executing *threads* that run in the same address space. Hence, different threads can communicate through shared memory buffers and use semaphores for synchronization. For efficiency reasons, no memory protection mechanisms are available at the moment.

All the devices are handled by user-level drivers that, being outside the kernel, are activated by the user program and can be linked in the executable only if they are really needed. The interrupt handling mechanism is similar to the one used in μkernel architectures.

3.2. Scheduling

The kernel directly supports different task models, which can be grouped based on two orthogonal dimensions: *criticality* and *periodicity*. Task criticality can be *HARD*, *SOFT*, or *NRT* (non real-time), whereas task periodicity can be *PERIODIC* or *APERIODIC*.

Task models characterized by different criticalities are handled by different scheduling algorithms. HARTIK has been historically based on the Earliest Deadline First (EDF) scheduling algorithm [14], so the main ready task queue is ordered by absolute deadlines.

Hard real-time tasks are directly put in the ready queue upon activation and are scheduled by their absolute deadline. To ensure that all hard deadlines are respected, an admission test (see [14]) is performed by the kernel on task creation, avoiding to create tasks that will overload the system. In order to perform the admission test, the WCET C_i of each hard tasks must be provided at creation time. To detect possible WCET violations (overruns), a runtime monitoring mechanism keeps track of the actual execution times and raises an exception whenever a thread τ_i tries to execute more than its declared C_i value. In this case, the program can decide either to abort (default option), to kill the misbehaved task, or to continue (at the risk of missing some deadlines).

Soft real-time tasks are put in the EDF queue after assigning them a suitable *scheduling deadline* through a proper mechanism. In the original kernel, all soft tasks were handled by a single Total Bandwidth Server [20]. This solution, however, prevented the possibility of controlling the performance of each single soft task. In the current version, each soft task is handled by a dedicated server, the Constant Bandwidth Server (CBS) [1], explicitly designed to provide isolation and control the QoS level of each task. Such a server mechanism will be briefly described in Section 4. Finally, non real-time tasks are scheduled in background with respect to hard and soft real-time tasks.

3.3. Interrupt Handling

One of the most important issues to be addressed in a real-time kernel is to respond to hardware interrupts in a short time without compromising the QoS experienced by the threads running in the system. In classical operating systems, when a hardware interrupt arrives, the kernel stops executing the current thread to serve the interrupt. Although such a solution minimizes interrupt latency, it introduces unpredictable delays in real-time threads' execution. On the other hand, if the system tries to preserve the real-time guarantee by delaying the interrupt request, the interrupt service time increases. Clearly, a trade-off between preserving the real-time guarantee and providing bounded interrupt response times is required.

HARTIK handles interrupts in a very flexible way: each handler is split into a *fast handler* and a *safe handler*. The fast handler is immediately executed in the context of the running thread as a non preemptable section, but with enabled interrupts. The safe handler is executed as a hard or soft real-time task, automatically activated by the interrupt. In this way, using a short (or even null) fast handler, the schedulability will not be affected by internal interrupts, whereas a short interrupt service time can be achieved using a safe handler with a short deadline (it is worth noting that the effects of the safe handler can easily be taken into account in the schedulability analysis). Section 4 discusses how to dimension the safe handler deadline.

This approach is very similar to the one used in μkernel architectures, where device drivers can be realized as kernel or user level processes (colocated servers or user servers). On the other hand, lots of variations of classical Unix kernels propose to schedule the bottom handlers, modifying the kernel such that it becomes similar to a multithreaded kernel (like Solaris), or to a μkernel with colocated servers. HARTIK is based on a cleaner structure, expressly designed for providing performance and high predictability.

3.4. Programming Interface

HARTIK was designed to provide an easy and direct mapping from the real-time task model to the kernel API. In HARTIK, a real-time application is a set $\Gamma = \{\tau_i\}$ of real-time and non real-time tasks[1]. A real-time task τ_i is a stream of jobs $J_{i,j}$, each of which released (activated) at time $r_{i,j}$, executing for a time $c_{i,j}$ and finishing at time $f_{i,j}$. Each job is also assigned an absolute deadline $d_{i,j}$, generally equal to $r_{i,j} + D_i$, being D_i the relative deadline of τ_i. A periodic task is a task such that $r_{i,j} = r_{i,j-1} + T_i$.

HARTIK directly supports the periodic and aperiodic task model, allowing the user to create periodic (time-driven) threads, that are automatically activated by the system at each period, and aperiodic (event-driven) threads, that must be explicitly activated, or connected to an interrupt. To our best knowledge, only few other real-time kernels (such as RT-Mach) directly support the concept of task

[1] In this paper the terms "task" is used to denote a generic concurrent activity, that can be a thread or a process.

```
TASK my_task(int parm)
{
      <local (private) variables>

      <Initialization>
      while (!finish) {
            <Job body>
            task_endcycle();
      }
      <Ending>
}
```

Figure 1. Structure of a typical HARTIK task.

instance. In this section, we are not interested in documenting the entire kernel API (for that, see the HARTIK manual [4]), but only in explaining how the API reflects the real-time task model.

The typical structure of a HARTIK real-time task is shown in Figure 1. After a (possibly empty) initialization phase, the task enters a (possibly infinite) loop, which represents the job body. The task notifies the kernel the end of an instance through the `task_endcycle()` system call. Periodic task instances are automatically activated at the beginning of each period, whereas aperiodic task instances can either be triggered by external interrupts or explicitly released through the `task_activate()` call. An aperiodic task can be bound to an external interrupt (as a safe handler) using the `handler_set()` system call.

We recall that such a powerful and simple programming interface can simplify the development and implementation of real-time applications, so that no design tools are required to develop complex applications: the programmer has only to design the application in terms of tasks, jobs activations, and relations between tasks (following the classical real-time model), and the implementation of such an application will result simple and natural. By freeing the programmer from implementing extra abstractions and mechanisms for dealing with periods, deadlines and periodic task activations, the proposed interface reduces the development time and decreases the number of programming errors.

4. Dynamic QoS Extensions

In many real-time applications dealing with multimedia and active monitoring systems, computational activities are characterized by timing constraints which are less stringent than those allowed by critical control systems. Nevertheless, the execution of each individual task needs to be controlled to keep the resulting QoS within a specified range. An additional problem introduced by multimedia systems is that task computation times, and even task interarrival times (if the network is involved), have usually large variations, thus any worst-case assumption would make the application very inefficient.

To cope with such more dynamic environments, we had to significantly modify the original kernel design, by introducing a temporal protection mechanism to isolate the behavior of each task. To provide temporal protection we decided to use a dedicated server for each soft task, so ensuring that each task received a specified fraction of the processor (a CPU bandwidth) according to its individual QoS requirement. Unfortunately, the TBS service mechanism was not suited for this purpose, since its behavior strongly relies on the knowledge of jobs' computation times, which cannot easily be predicted in multimedia systems.

The problem was solved by replacing the TBS with a new service mechanism, the Constant Bandwidth Server (CBS) [1], which has a performance similar to the TBS, but provides temporal protection against overruns. A more complete description of the algorithm and its properties can be found in [1]. In this context, we just recall a few basic concepts which can be useful for a better understanding of our feedback mechanism.

We recall that the HARTIK scheduler is based on a global EDF queue, where soft and hard real-time tasks coexist together: hard tasks are scheduled based on their absolute deadlines, whereas soft tasks are scheduled according to fictitious deadlines assigned by the server. A CBS is described by two parameters, Q_s and T_s, where Q_s is the *maximum server budget* and T_s is the *server period*. The ratio $U_s = Q_s/T_s$ is called the server *bandwidth*. The service mechanism can be summarized as follows: when a new request arrives at time $r_{i,j}$, the server checks whether it can be scheduled using the last assigned deadline, otherwise the request is assigned an initial deadline equal to $r_{i,j} + T_s$. If the job executes for more than Q_s units of time, its scheduling deadline is postponed by T_s. In this way, it is guaranteed that each active task can execute for at least Q_s units of time in every interval T_s, but cannot demand more than Q_s units in the same interval.

If the temporal parameters of a task are known, the Q_s and T_s parameters can be assigned to the server to provide a hard (see [1]) or a probabilistic (see [2]) guarantee for the task. If no a-priori information is available on the task temporal requirements, the server parameters Q_s and T_s can be adapted on line using a sort of feedback mechanism. This technique is often referred to as *adaptive scheduling* or *feedback scheduling*.

To introduce adaptive scheduling in the HARTIK kernel, we developed some task models whose parameters can easily be mapped to the adaptive scheduling framework [3] as explained below.

The task models defined for supporting adaptive scheduling describe three different execution behaviors:

- *PseudoPS* (PPS) tasks represent a uniform execution flow of instructions receiving a constant share of the CPU. Each PPS task τ_i is characterized by a weight w_i, defining the CPU share assigned to it.
- *MultiMedia* (MM) tasks describe periodic (time driven) activities with a variable (and unknown) execution time. They are characterized by a fixed activation period T_i and by a weight w_i, representing the task's importance.
- *Event Driven* (ED) tasks describe aperiodic (event driven) activities activated by external or internal aperiodic events. Each ED task is characterized by an *expected activation rate* $R_i = 1/T_i$ and by a weight w_i, representing the task's importance.

PPS tasks do not need any form of feedback to adapt their scheduling parameters. The maximum budget Q_s can be set equal to a fixed system constant Q, whereas the server period can be assigned such that the total PPS bandwidth B_{PPS} is partitioned according to the user defined weights.

MM tasks require a periodic execution and a feedback mechanism is used to automatically adapt the reserved bandwidth to properly schedule the tasks in a periodic fashion. ED tasks are similar to MM tasks, but they are activated by external or internal aperiodic events. A complete description of the adaptive task models can be found in [3].

Notice that, in the PPS model, the Q_s parameter remains constant (it is similar to a system "scheduling quantum") and the T_s parameter is adjusted to properly share the bandwidths when a PPS task is created or destroyed. On the contrary, in the ED and MM models, the T_s parameter is fixed and the Q_s parameter is adjusted using feedback. The goal of the feedback mechanism is to assign each task τ_i a budget Q_i such that the deadline assigned by the CBS will not be postponed. For each job $J_{i,j}$, the observed variable acting as a feedback is the *CBS scheduling error* $\epsilon_{i,j}$, defined as

$$\epsilon_{i,j} = d^s_{i,j} - r_{i,j} - T_s, \tag{1}$$

where $d^s_{i,j}$ is the last deadline assigned by the server to job $J_{i,j}$. By monitoring the CBS scheduling error, the budget Q_i assigned to a task τ_i can be increased when $\epsilon_{i,j} > 0$. Whenever a bandwidth B_i is increased, it is important to make sure that the total processor load is always kept below one or, in general, below a desired maximum value $B^{max} \leq 1$. In HARTIK, the total processor load is controlled through a *Bandwidth Compression* mechanism, which decreases the bandwidth assigned to each task when $\sum_i B_i > B^{max}$. In this case, each task bandwidth is compressed (by rescaling the Q_i parameter) proportionally to

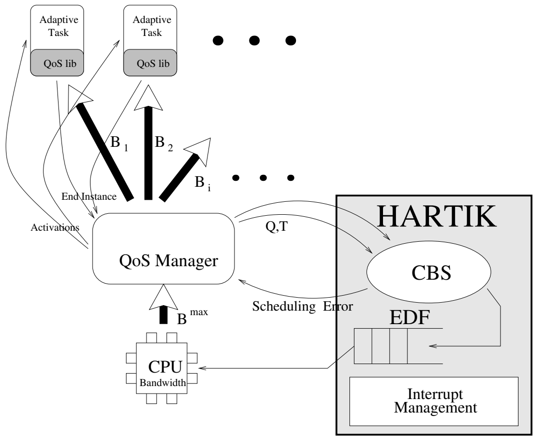

Figure 2. Scheme of the QoS Manager.

the task weight w_i (see the cited paper [3] for full explainations).

The QoS manager performing the bandwidth adaptation is implemented as a conventional HARTIK task, running on top of the kernel. When started, the `QoSman` task gets all the free CPU bandwidth B_{max}, preventing the kernel from creating new tasks. All new tasks are created by `QoSman` with a default small bandwidth, which will then be modified according to the feedback and compression mechanism. All tasks communicate with the `QoSman` using a QoS library, which is linked to the application code.

A scheme of the interactions between the QoS manager and the tasks is illustrated in Figure 2. An important advantage of the proposed approach is that it is not intrusive (no changes are required into the kernel) and, as a middleware component, the QoS manager results to be more portable and configurable than a direct in-kernel implementation. Moreover, the overhead introduced is small.

4.1. Adaptive interrupt handling

To improve predictability, as well as efficiency in device management, the bandwidth adaptation policy has been extended to the interrupt handling mechanism, for precisely tuning the parameters of the safe handlers. In fact, treating a safe handler as a hard aperiodic task would require the user to assign it a relative deadline equal to the maximum tolerable response time, which would degrade the average-case performance. Moreover, this approach would jeopardize schedulability if an interrupt service routine should require more time than its declared WCET, or if the interrupt requests should arrive more frequently than expected.

To prevent possible interrupt overruns to compromise the schedulability of the other tasks, each safe handler is treated as a soft aperiodic task served by a dedicated CBS. The server period T_s can be set equal to the service time desired for the interrupt, whereas the server budget Q_s has to be set according to the workload generated by the interrupt. This is especially important for the network driver, whose requests may have very irregular interarrival times.

In the present implementation, the user has the possibility to directly specify the Q_s parameter during the driver initialization, considering this problem as a program design issue. However, we believe that a more appropriate solution for assigning the server parameters to the safe handlers is to treat them as ED tasks: in this way, the same feedback mechanism can be used to reserve the correct fraction of the total CPU bandwidth to application tasks and to safe handlers.

By monitoring the driver and application tasks' executions, a feedback mechanism would be able to change the initial bandwidth reservation in order to properly balance interrupt versus application processing. Intuitively, an application task that blocks for too long, waiting for the driver, would cause the feedback mechanism to increase the bandwidth assigned to that safe handler. Conversely, a driver producing data that the application cannot consume, would cause an increase of the application task's bandwidth.

5. Experiences

In this section we show some performance experiments which prove the effectiveness of the proposed QoS adaptation mechanism. In a first experiment, we tested the isolation property introduced by the CBS on tasks' executions. To stress the protection mechanism, we created a set of soft tasks characterized by a very high variance in their execution times. A typical periodic task characterized by a large execution time variance is an MPEG [8] player (all experiments were performed using the Berkeley MPEG library [16], which was ported to HARTIK). Then, we performed a simple experiment with 2 tasks, τ_1 and τ_2, having periods $T_1 = 125ms$ (8 frames per second - FpS) and $T_2 = 30ms$ (33 FpS).

Figure 3 plots the number of decoded frames as a function of time when the two periodic tasks are scheduled by the plain EDF algorithm, without the CBS protection mechanism. We note that, when τ_2 is activated at time $t = 2000$, the system experiences an overload, and τ_1 is slowed down to 4.44 Fps. This happens because tasks are not isolated, so that the execution of τ_2 creates interferences on task τ_1.

Figure 4 shows the number of decoded frames for the same task set when the CBS protection mechanism is active. As clear from the plots, the CBS mechanism isolates the temporal behavior of the tasks, preventing τ_2 to interfere with the execution of τ_1.

Although the CBS provides isolation among tasks, enabling the programmer to guarantee each task independently from the others, some a-priori information on the WCETs is still needed to properly assign the Q_s parameters. If the maximum budget Q_s is enough for performing the required computations, the served task executes within its (soft) temporal constraints, otherwise it slows down according to the assigned bandwidth. We observed, however, that the task still receives its reserved fraction of the CPU, so the performance degradation is not unpredictable. To verify such a behavior, we performed a second experiment in which a task was assigned a budget less than the actual computational request. In particular, we tested the performance of a soft task with period $T_i = 30ms$ and mean execution time $\overline{c_i} > 5ms$, served by a CBS with parameters $Q_s = 5$ and $T_s = 30$, and verified that although the task does not execute at the requested rate (33 jobs per second), its execution rate is still stable.

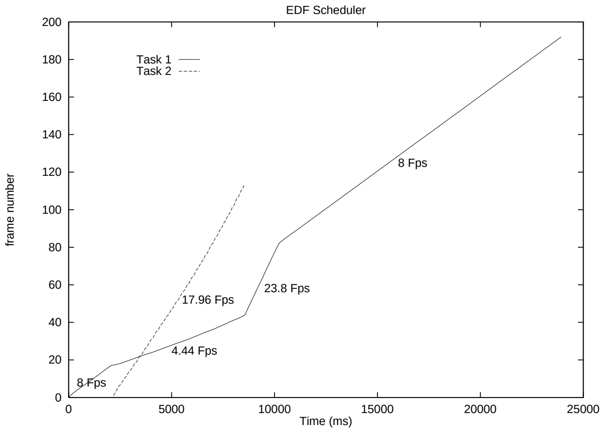

Figure 3. Two MPEG players scheduled by EDF.

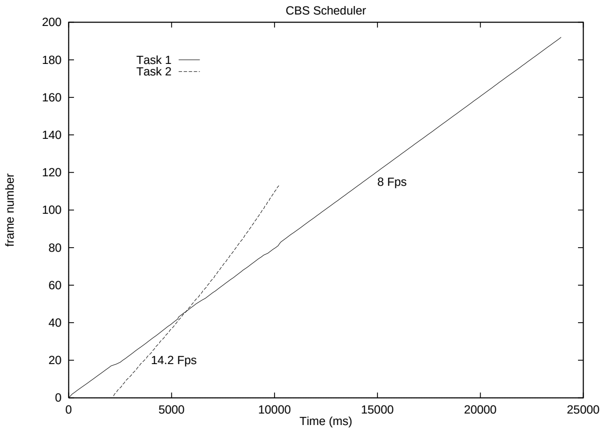

Figure 4. Two MPEG players scheduled by CBS.

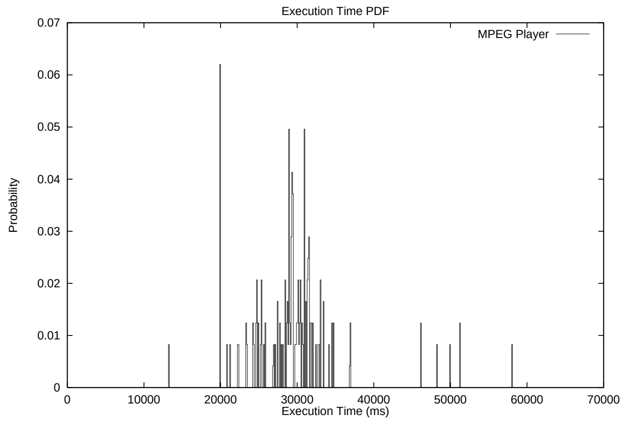

Figure 5. Execution time PDF for an MPEG player decoding a video.

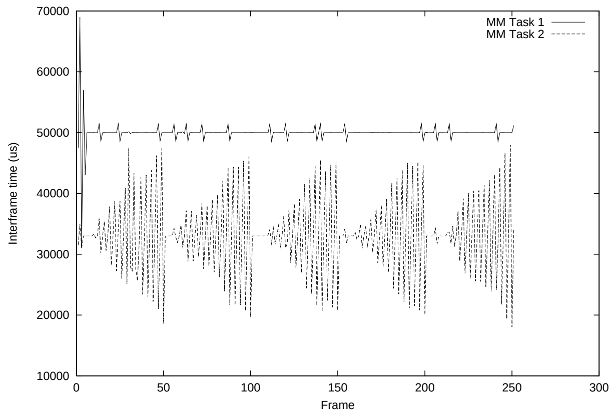

Figure 6. Interframe times with $w_1 = 10, w_2 = 5$.

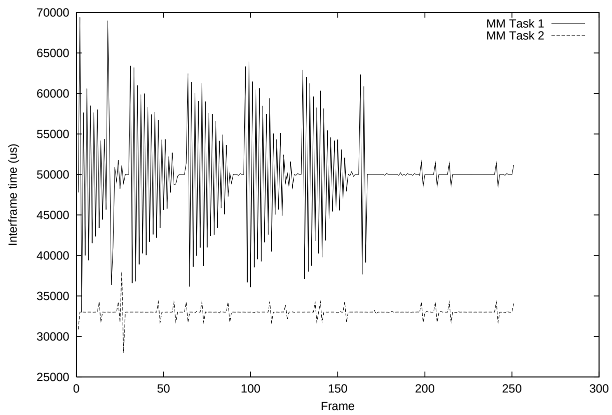

Figure 7. Interframe times with $w_1 = 5, w_2 = 10$.

Another set of experiments was aimed at testing a monitoring mechanism embedded in the kernel for on-line estimation of tasks' computation times. Our objective was to develop predictable soft real-time applications without forcing the user to provide WCET calculations, which are usually quite hard to achieve, unless they are very pessimistic.

To overcome the problems caused by a WCET-based design, we decided to developed an execution time estimator embedded in the kernel for collecting statistics on tasks execution times. Using this approach, a program can be run in advance for profiling and collecting all the jobs' execution times in an array, from which a distribution of the job execution times can be derived and used to perform a probabilistic guarantee, as described in [2]. Figure 5 reports an example of the Probability Distribution Function (PDF) derived from an MPEG player decoding a specific MPEG video.

To conclude, it is worth noting that the design approach based on two separate steps (profiling and parameters assignment) is not very practical for tasks whose execution times depend on the input data (such as, for example, compressed video/audio players). For this kind of tasks, the adaptive feedback scheduling enforced by the QoS manager is more effective. In addition, this approach makes applications easily portable on different hardware platforms, and allows to consider tasks' importance independently from task periods (see [3]). As an example, we report an experiment in which two video players are executed as two periodic tasks: τ_1 with $T_1 = 50ms$ and τ_2 with $T_2 = 33ms$. Figure 6 plots the interframe times when τ_1 is assigned an importance value greater than that of τ_2 ($w_1 = 10$ and $w_2 = 5$), whereas Figure 7 shows the interframe times when τ_2 is considered more important than τ_1 ($w_1 = 5$ and $w_2 = 10$). From the figures we see that the QoS experienced by the tasks can be controlled independently from their periods. Additional experiments which prove the effectiveness of the proposed approaches can be found in [3].

6. Conclusions and future work

In this paper, we described our experience in supporting multimedia applications with the HARTIK kernel. The most important lessons learned during this phase can be summarized as follows:

1. a temporal protection mechanism is essential for providing a desired QoS level to soft tasks, individually;

2. temporal protection can be easily and efficiently implemented using the CBS scheduling algorithm;

3. when tasks are isolated, a probabilistic guarantee can be performed on soft tasks;

4. an adaptive scheduling strategy based on a feedback mechanism can be efficiently used to assign server parameters when no a-priori information is available on the tasks;

5. the adaptive scheduling approach can be easily and efficiently implemented on top of the kernel, with a small runtime overhead.

Currently, we are using the QoS manager to test different feedback strategies and prove some theoretical properties, such as convergence and stability of our feedback control scheme. Another ongoing research deals with the problem of dynamically adapting the fraction of CPU time dedicated to interrupt handling. As discussed in Section 4, we are trying to use ED tasks for serving interrupts.

The HARTIK kernel is distributed under the GPL license and can be downloaded from

`http://hartik.sssup.it`

whereas the QoS manager will be released soon.

References

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

[2] L. Abeni and G. Buttazzo. Qos guarantee using probabilistic dealines. In *Proceedings of the IEEE Euromicro Conference on Real-Time*, York, England, June 1998.

[3] L. Abeni and G. Buttazzo. Adaptive bandwidth reservation for multimedia computing. In *Proceedings of the IEEE Real Time Computing Systems and Applications*, Hong Kong, December 1999.

[4] G. Buttazzo, L. Palopoli, L. Abeni, G. Lipari, G. Lamastra, A. Casile, M. Giorgi, and P. Gai. Hartik user manual. ftp://hartik.sssup.it/pub/hartik/3.3.1/man331ps.zip.

[5] G. C. Buttazzo. Hartik: A real-time kernel for robotics applications. In *IEEE Real-Time Systems Symposium*, December 1993.

[6] G. C. Buttazzo, B. Allotta, and F. P. Fanizza. Mousebuster: a robot for catching fast objects. *IEEE Control Systems Magazine*, 14(1):49–56, February 1994.

[7] A. Damm, W. Reisinger, W. Schwabl, and H. Kopetz. The real-time operating system of mars. *ACM Operating Systems Review*, 23(3):141–151, July 1989.

[8] D. Gall. Mpeg: A video compression standard for multimedia applications. *Communications of the ACM*, 34(4):46–58, April 1991.

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

[10] H. Hrtig, R. Baumgartl, M. Borriss, C.-J. Hamann, M. Hohmuth, F. Mehnert, L. Reuther, S. Schnberg, and J. Wolter. Drops - os support for distributed multimedia applications. Sintra, Portugal, September 1998.

[11] G. Lamastra, G. Lipari, G. Buttazzo, A. Casile, and F. Conticelli. Hartik 3.0: A portable system for developing real-time applications. In *Proceedings of the IEEE Conference on Real-Time Computing Systems and Applications*, October 1997.

[12] I. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Area on Communications*, June 1997.

[13] S.-T. Levi, S. K. Tripathi, S. D. Carson, and A. K. Agrawala. The maruti hard real-time operating system. *ACM Operating Systems Review, SIGOPS*, 23(3):90–105, 1989.

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

[15] T. Nakajima. Resource reservation for adaptive qos mapping in real-time mach. In *Sixth International Workshop on Parallel and Distributed Real-Time Systems (WPDRTS)*, April 1998.

[16] K. Patel, B. C. Smith, and L. A. Rowe. Performance of a software mpeg video decoder. In *Proceedings of ACM Multimedia 93*, Anaheim, CA, 1993.

[17] 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.

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

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

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

[21] J. Stankovic and K. Ramamritham. The spring kernel: A new paradigm for real-time systems. *IEEE Software*, 8(3):62–72, May 1991.

[22] H. Tokuda and C. Mercer. Arts: A distributed real-time kernel. *ACM Operating Systems Review*, 23(3):29–52, July 1989.

[23] H. Tokuda, T. Nakajima, and P. Rao. Real-time mach: Toward a predictable real-time system. In *USENIX Mach Workshop*, pages 73–82, October 1990.

[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. Enanching the real-time capability of the linux kernel. In *IEEE Real Time Computing Systems and Applications*, Hiroshima, Japan, October 1998.