HARTIK: A Real-Time Kernel for Robotics Applications

Giorgio C. Buttazzo
A.R.T.S. Lab, Scuola Superiore S. Anna
Via Carducci, 40 - 56100 Pisa, Italy
Email: giorgio@sssup1.sssup.it

Abstract

This paper presents a hard real-time kernel, called HARTIK, specifically designed to handle robotics applications with predictable response time.

The main relevant features of this kernel include: direct specification of time constraints, such as periods and deadlines; preemptive scheduling; coexistence of hard, soft, and non real-time tasks; separation between time constraints and importance; deadline tolerance; dynamic guarantee of critical tasks; and graceful degradation in overload conditions.

The functionality of the kernel is then shown by presenting a concrete example of a robot system that has to explore unknown objects by visual and force feadback. This and other examples of real-time tasks are available in videotape.

1. Introduction

Advanced robotic systems are going to become more complex, distributed, and capable of exhibiting intelligent, adaptive, and highly dynamic behavior. Next generation robotic system have to operate autonomously in unstructured environments, and perform critical tasks in many different conditions.

In order to perform critical operations in an unknown and dynamic environment, the robot control system must be equipped with a sophisticated sensory apparatus, and may have to satisfy different classes of timing constraints at different times. This precludes the use of static scheduling policies, commonly adopted in today's real-time systems.

Consider for example a robot arm equipped with an electrical gripper and a set of sensory systems, such as CCD cameras, proximity transducers and force/torque sensors. If the robot has to operate in unstructured environments, sensory information must be processed in real-time and used for closing servo loops at different control levels.

Controlling such a complex system requires the execution of different activities, each characterized by specific time constraints. Some constraints may be critical, in the sense that a missed deadline can cause serius damages to the system. For instance, sensory acquisition is a typical periodic activity that is performed at constant sampling rate. Low level servo loops, sensor-based control, and trajectory planning are also intrinsically periodic activities, that must execute under strict time constraints to guarantee the stability of the system. Other tasks involving graphics representations, status display, or monitoring activities, may not be critical, and can be delayed when a critical task has to complete the execution within its deadline.

Moreover, tasks can be activated dynamically. For instance, if a moving obstacle enters in the robot workspace while the arm is performing a critical action, the robot trajectory has to be replanned within the time constraints imposed by the current action to avoid catastrophic results. If this new activity cannot be executed in time, the system should not activate it, but execute instead a recovery action, which for example stops the robot in a safe location.

Conventional real-time kernels are not suitable for critical robot applications, since they do not manage time explicitly. This is because they are usually based on priority preemptive schedulers, which minimize the average response time of the system activities, but do not guarantee the meet the individual timing requirement of each task in all anticipated operating conditions [8].

In this paper we describe the characteristics of a hard real-time kernel, called HARTIK, specifically designed to develop predictable robotics applications. Although a number of HARTIK characteristics are common with other hard real-time systems [6][11][9][4], there are novel features that allow coexistence of hard, soft, and non real-time tasks, allow to define different types of time constraints, specify tolerant deadlines, separate dealines and importance, offer recovery strategies in overload conditions, and provide fast and asynchronous communication channels, called CABs, particularly suited for exchanging information among periodic control tasks.

1052-8725/93 $03.00 © 1993 IEEE

2. Characteristics of HARTIK

The HARTIK kernel has been designed with the following characteristics:

- Flexibility in expressing timing constraints. To cope with the different application requirements, each task can be declared as hard, soft, or non real-time; hard and soft tasks can be periodic or sporadic. Deadlines may have a tolerance value. See [2] for details.
- Dynamic preemptive scheduling and guarantee: when activating a critical task, the system performs a schedulability analysis to check whether all critical tasks will meet their timing constraints. If a task cannot be guaranteed, the system raises an exception to allow the programmer to take an alternative action.
- Separation between time constraints and importance of a task. Each task has an additional parameter, called VALUE, which reflects the relative importance of a task with respect to the other task in the set. The value is used to reject the least important tasks in overload conditions to make the task set schedulable. This feature allows graceful degradation in overload conditions.
- Asynchronous communication for periodic task interactions. A particular non blocking mechanism (CAB) is used to exchange data among periodic processes, so that unpredictable delays can be avoided during task execution.
- Interrupt mechanism integrated with the general scheduling policy of the kernel. Any I/O interrupt is treated as an instantiation of a new task, which is guaranteed and scheduled just like any other task.
- Time bounded primitives and short context switch time. This feature allows to develop processes with bounded execution and predictable response time.

These features cannot be found, all together, in current real-time systems. For example, the Spring kernel [9] is highly dynamic and flexible for specifing time constraints, but it uses non preemptive scheduling and does not provide special facilities (like CABs) for handling periodic activities, typical in robotics control applications. On the other hand, the MARS system [4], is intrinsically periodic, but it is based on static scheduling and off-line guarantee, which is not suited to work in dynamic environments.

2.1. Task scheduling and guarantee

To deal with all possible activities that can occur in a real-time control application, such as a multisensor robotics system, HARTIK handles four types of tasks:

- **HARD tasks.** A hard task is a periodic process with a critical deadline coincident with its period. As a hard task is activated, a guarantee routine verifies whether all time critical tasks will meet their deadline. A run time check on hard deadlines is also performed.
- **SPORADIC tasks.** A sporadic task is an aperiodic process with irregular arrival times, a maximum interarrival rate, a critical deadline, and guaranteed execution time. A missed deadline for a sporadic task is signalled by the system.
- **SOFT tasks.** A soft task is a process with non critical time constraints. Soft tasks are scheduled based on their deadlines, but they are not guaranteed by the system. Soft tasks are dispatched only when no HARD, nor SPORADIC processes are ready to execute. A missed deadline for a soft task is signalled by the kernel.
- **NRT tasks.** They are Non-Real-Time tasks with no time constraints at all. They are scheduled based on a static priority assigned by the user. NRT tasks are dispatched only when no HARD, SPORADIC, nor SOFT processes are ready to execute.

Time critical processes are scheduled based on their deadlines, according to the *Robust Earliest Deadline* (RED) scheduling policy [2]. The RED algorithm is an extension of the Earliest Deadline First (EDF) algorithm, which behaves as EDF in underload conditions, but also have a good performance in overload conditions. By separating task importance from time constraints, different rejecting and recovery strategies can be used during transient overloads to make the task set schedulable, minimizing the total loss value.

SPORADIC tasks are handled by using a sporadic server [10]. For a generic set of periodic and sporadic tasks in which all sporadic processes are handled by a server, the system guarantees the scheduling feasibility according to the following rule:

$$\sum_{i=1}^{n} \frac{C_i}{T_i} + \frac{C_s}{T_s} < 1 - U_{ov}$$

where C_i the estimated worst case execution time of a task, T_i is its period, C_s and T_s are the capacity and the period of the server, and U_{ov} is the utilization factor due to the overhead introduced by the timer handling routine.

The guarantee algorithm uses a set of timing parameters specified by the user, including an estimation of the worst-case execution time of the task, which can be computed by a dedicated tool, specifically developed for the processor used in the application.

2.2. Basic primitives

The kernel is implemented as a set of library functions, which extend the C language by introducing concurrency and hard real-time characteristics. The HARTIK environment is generated by the primitive *ini_system(quantum, unit)*, which allows the user to specifies the period of the interrupts from the real-time clock, ranging from 50 microseconds to 55 milliseconds.

An HARTIK process has the same structure as a C function, except that a process cannot have arguments, and its name has to be declared as PROCESS type. Once defined, a process can be created by the primitive:

```
create(pr_id, pr_name, mode, type, dline,
       ex_time, value);
```

where *pr_id* is an arbitrary string of characters; *pr_name* is the name used in the process definition; *mode* specifies the creation modality by which a process is inserted in the ready queue; *type* specifies the timing characteristics of the task (HARD, SPORADIC, SOFT, NRT); *dline* is the absolute time by which the process must complete its execution; *ex_time* is the estimated time required to the processor to execute the task without interruptions; and *value* is a parameter specifying the relative importance of the process within its class.

For periodic processes, the deadline parameter is interpreted as a period, whereas for sporadic tasks it represents the minimum interarrival time between two consecutive requests. For NRT tasks the execution time is ignored, and the deadline parameter is considered as a static priority from 0 to 255, being 0 the maximum priority.

Three types of activation modes are provided: NOW, SLEEP, and GROUP. Using the NOW modality, the task is created and immediately inserted in the ready queue. In SLEEP mode, all task data structures are allocated and initialized, but the process is maintained in a sleeping state, until an activation primitive, *activate*(process), is explicitally invocated by another task. When more tasks must be activated at the same time, they can be created with the GROUP mode, and then activated simultaneously by the primitive *start_group()*.

To terminate an instance of a periodic activity, HARTIK provides a primitive, called *end_period*, which inserts the process in a queue of suspended tasks. Since *end_period* does not modify the program counter and the process stack, the periodic code must be contained in a cyclic program structure, such as a *while* loop.

An aperiodic activity is simply terminated by calling the primitive *end_process*. The complete set of primitives with the evaluation of their computation time can be found in [1].

3. Process communication

Communication among critical tasks must be carefully considered in real-time systems. In fact, blocking over a shared resource or waiting for a message to arrive may introduce an unbounded delay that can cause a task to miss its deadline. For this reason, synchronous interactions should be avoided among critical tasks, unless they are time bounded. HARTIK provides both synchronous and asynchronous communication primitives to adapt to different task requirements.

3.1 Ports

The notion of *port* is widely used in real-time systems, since it provides the abstraction of "communication channel" as a data type [7][5]. HARTIK provides three types of ports: RECEIVE, BROADCAST and STICK.

A RECEIVE port is a port for one to one communication, where the owner task is the only task allowed to receive data from it. Sending messages to a receive port is non blocking, whereas receiving messages is always synchronous with timeout.

A BROADCAST port provides a one-to-many communication channel. It has a list of destination ports to which messages are to be forwarded. When a message is sent to a broadcast port, it is redirected to all ports specified in the list.

STICK ports can be seen as shared variables, because communicating tasks can block only in the critical section of the procedures send and receive, but never block for an empty or full buffer. Messages are not consumed in a stick port, and they are overwritten by a new message.

3.2 Cyclic Asyncronous Buffers (CAB)

HARTIK provides a particular communication mechanism, call CAB (Cyclic Asynchronous Buffer), purposely designed for the communication among periodic activities, such as control loops and sensory acquisition processes. A similar communication mechanism was proposed in [3].

A CAB provides a one-to-many communication channel, and contains at any instant the latest message inserted in its buffer. A message is not consumed by a receiving process but is maintained into the buffer until a new message is overwritten. In this way, a receiving process will always find a data in the buffer, so that unpredictable delays due to synchronization can be eliminated, making the system more reliable. Unlike a STICK port, a CAB does not require to copy the message in the data area of the receiving task, but it returns the message pointer to all tasks that make an explicit request.

Test programs showed that communicating through CABs is much faster than passing messages through STICK ports, and CAB primitives are time bounded [1]. CABs must be declared as cab type, and initialized with the primitive *open_cab*, having the following syntax:

```
cab      *cab_id;
cab_id = open_cab(name, num, dim, first);
```

where `name` is an arbitrary character string, `num` is the number of messages that the CAB may contain simultaneously, `dim` is the dimension (in bytes) of the message type for which that CAB is dedicated, and `first` is the pointer to the initial message contained in the CAB.

To insert a message in a CAB, a task must reserve a buffer from the CAB memory space, then it has to copy the message in that buffer, and finally put the buffer in the CAB structure, where it becomes the most recent data. Reserving and putting a CAB buffer is performed by two system calls, according to the following syntax:

```
buf_pointer = reserve(cab_id);
<copy message in *buf_pointer>
put_mes(buf_pointer, cab_id);
```

Similarly, to get a data from a CAB, a receiving process has to read the pointer to the most recent message in the CAB, use the data, and release the pointer to the CAB. This is done according to the following scheme:

```
mes_pointer = get_mes(cab_id);
<use message>
unget(mes_pointer, cab_id);
```

Notice that more tasks can access the same CAB simultaneously. If a task reserves a CAB for writing and the buffer is being used by another task, the CAB creates a new buffer, which will become the most recent one in that CAB. CABs are particularly suited for implementing sensor-based servo loops, in which a data acquisition process puts the sensory data in the CAB and one or more control tasks use the most recent data for monitoring activities or closing servo loops.

4. Other features and tools

The current version of HARTIK runs on a Intel 80486 microprocessor at 33 MHz, with 0.84 microseconds resolution hardware clock.

The interrupt mechanism is integrated in the general scheduling algorithm of the system, so that any I/O interrupt is treated as an instantiation of a new task which is subject to the guarantee algorithm and scheduled just like any other task.

To assist the programmer in estimating the worst case execution time of tasks, a specific tool (MET) has been developed for this purpose. It uses a table-driven model of the processor, where assembly instructions are translated into execution times depending on their operating code, operands and addressing mode.

Another tool, called *Real-Time Tracer* (RTT), has been developed to monitor the system evolution while an application is running. If active, the RTT records in main memory all context switches performed by the kernel, and, at system termination, it saves this information in a file. This file can then be interpreted by another tool, which produces a graphics representation of the system evolution in a desired time scale.

5. A robot control example

Consider a robot system that has to explore unknown objects by integrating visual and tactile information. To perform this task the robot has to exert desired forces on the object surface, and follow its contour by means of visual feadback. The software control architecture can be organized as two servo loops: the inner loop dedicated to image acquisition, force reading, and robot control, and the outer loop performing scene analysis and surface reconstruction. The task set consists of four processes.

- A sensory acquisition process periodically reads the force sensor and puts the data in a CAB named "force". This process is critical since a missed deadline could cause an unstable behavior of the robot. Hence, it is created as a HARD task with a period of 20 ms.
- A visual process periodically reads the memory filled by the frame grabber and computes the next exploring direction. Data are put in a CAB named "angle". This is also a HARD task with a period of 100 ms. A missed deadline could cause the robot to follow a wrong direction on the object surface.
- A robot control process, based on the force data and on the exploring direction suggested by the vision system, computes the cartesian set points to send to the position controller. The control process is a periodic HARD task with a period of 28 ms. Missing a deadline for this task could cause the robot to react too late and to exert too large forces on the explored surface, that could break the object or the robot itself.
- A representation task reconstructs the object surface based on the force/torque data and the exploring direction. Since this is a graphics activity which does not affect the robot motion, the representation process is created as a SOFT task with a period of 80 ms.

Here is the body of the main process, and of the visual process. HARTIK primitives are written in bold.

```
/*-----------------------------------*/
#include    "hartik.h"
#include    "param.h"

main() {

float       ini_vect[3] = {0.0,0.0,0.0};
float       ini_alfa = 0.0;

cab         ft_data, angle;
process     force, vision;
process     control, display;

   ini_system(1,MILLI);
   ft_data = open_cab("force",
                      SIZE1,2,ini_vect);
   angle = open_cab("angle",
                      SIZE2,2,&ini_alfa);
   create("sensor",force, GROUP, HARD,
                  20, MET1, VALUE1);
   create("camera",vision,GROUP, HARD,
                  100,MET2, VALUE2);
   create("robot",control,GROUP, HARD,
                  28, MET3, VALUE3);
   create("edge", display,GROUP, SOFT,
                  80, MET4, VALUE4);
   start_group();
   while (sys_clock() < LIFE_TIME);
   end_system();
}
/*-----------------------------------*/

PROCESS vision() {

float       image[256][256];
float       *alfa;

     while (1) {
           get_frame(image);
           alfa = reserve(angle);
           *alfa = compute_angle(image);
           put_mes(alfa,angle);
           end_period();
     }
}
/*-----------------------------------*/
```

6. Conclusions and work in progress

This paper presented a hard real-time kernel (HARTIK) specifically designed to provide facilities for programming robot tasks with explicit time constraints and predictable execution. For these characteristics, HARTIK is currently used as a platform for programming predictable real-time tasks in robotics applications, where control tasks and sensor acquisition processes have to be performed at different rates.

At the moment, all resources needed by critical tasks are considered available, so that resource constraints will not influence task scheduling. However, an integrated scheduling algorithm taking in account both time and resource constraints is under investigation.

Acknowledgements

The author wish to thank Jack Stankovic for his helpful discussions and suggestions for improving the paper. This work has been partially supported by ESPRIT III TRACS Project #6373, and by Italian MURST 40%.

References

[1] Buttazzo, G.C. et al: "HARTIK: A Hard Real-Time Kernel for Personal Computer", TR - ARTS Lab 92-01, Scuola Superiore S. Anna, Pisa, 1992.

[2] Buttazzo, G.C. and J.A. Stankovic: "RED: Robust Earliest Deadline Scheduling", Proc. of 3rd Int. Work. on Responsive Comp. Sys., Austin, 1993.

[3] Clark, D.: "HIC: An Operating System for Hierarchies of Servo Loops", Proc. of IEEE Int. Conf. on Rob. and Autom., pp. 1004-1009, 1989.

[4] Damm, A. et al: "The Real-Time Operating System of MARS", ACM Operating System Review, Vol. 23, No. 3, pp, 141-157, July 1989.

[5] Lee, I. and R. King: "Timix: A Distributed Real-Time Kernel for Multi-Sensor Robots", Proc. of Int. Conf. on Robotics and Automation, 1988.

[6] Levi, S., et al.: "The MARUTI Hard Real-Time Operating System". ACM Operating System Review, Vol. 23, No. 3, pp. 90-105, July 1989.

[7] Shin, K.G., and M.E. Epstein: "Intertask Communications in an Integrated Multirobot System", IEEE Jou. of Robotics and Automation, Vol. RA-3, No. 2, pp. 90-100, April 1987.

[8] Stankovic, J. A.: "A Serious Problem for Next-Generation Systems", IEEE Computer, Oct. 1988.

[9] Stankovic, J. A. and K. Ramamritham: "The Spring Kernel: A New Paradigm for Real-Time Systems", IEEE Software, 8 (3), pp. 62-72, May 1991.

[10] Sprunt, B., L. Sha, and J. P. Lehoczky: "Aperiodic Task Scheduling for Hard Real-Time Systems", The Jou. of Real-Time Systems, Vol. 1, No. 1, 1989.

[11] Tokuda, H. and C. W. Mercer: "ARTS: A Distributed Real-Time Kernel". ACM Operating System Review, 23 (3), pp. 29-52, July 1989.