Implementing and Using Execution Time Clocks in Ada Hard Real-Time Applications

By: M. González Harbour, M. Aldea Rivas, J.J. Gutiérrez García, and J.C. Palencia Gutiérrez

Departamento de Electrónica y Computadores
Universidad de Cantabria
39005 - Santander
SPAIN
email: {mgh, aldeam, gutierjj, palencij}@ctr.unican.es
phone: +34 42 201483 - fax: +34 42 201402

Abstract[1]**.** Off-line analysis techniques for hard real-time systems are all based on the assumption that we can estimate the worst-case execution time of the different tasks executing in the system. In the traditional cyclic-executive schedulers, execution time limits were enforced for each task by the scheduler. Unfortunately, in concurrent hard real-time systems such as those using the tasking model defined in Ada, no bound on the execution time of tasks is enforced, which may result in a system timing malfunction not detected by the analysis techniques. In this paper we explore the implementation of execution time clocks within the task scheduler, and we describe methods to detect execution time overruns in the application, and to limit their effects. We also discuss the use of execution time clocks to enhance the performance of sporadic server schedulers implemented at the application level.

Keywords: Scheduling, Hard Real-Time, Ada 95, Execution Time, Sporadic Server

1 Introduction

All the hard real-time analysis techniques used to get off-line guarantees on the schedulability of the system, such as Rate Monotonic Analysis (RMA)[5][1], rely on the estimation of the worst-case execution times of the different tasks and actions that execute in the system. Although there are techniques to measure or calculate these execution times [9][10], this is always a difficult task because of the unpredictability of the different execution paths within the program. Today's computer architectures with superscalar processors [11] and caches [8] make the prediction of execution times even more difficult, specially in the context of concurrent programs in which cache misses are frequent after interrupt service routines or context switches. If the worst-case execution time is underestimated, severe timing errors may occur, causing the system to fail. These faults are not always detectable during testing, because they may only happen under particularly improbable circumstances. Besides, an overrun of the execution time of a particular task may not cause that task to miss its deadlines, but

1. This work has been funded in part by the *Comisión Interministerial de Ciencia y Tecnología* of the Spanish Government under grant number TAP97-892

perhaps it will be a lower priority task the one missing them. In systems with a large number of tasks, the problem of finding the task that overrun its execution time may be practically intractable.

Traditional real time systems were built (and still are) using cyclic executive schedulers [1]. In these systems, if a particular task or routine exceeded its budgeted execution time, the system could detect the situation. Basically, whenever the minor cycle interrupt came in, it could check whether the current action had completed or not. If not, that meant an overrun. Unfortunately, in concurrent real-time systems built with multitasking preemptive schedulers, there is no equivalent method to detect and handle execution time overruns. This is the case for systems built using the Ada tasking model and the associated Real-Time Annex.

As part of the development of the real-time extensions to POSIX [3], the standard for open operating system interfaces, execution time clocks and timers have been proposed as a new extension within the POSIX.1d standards project [2]. The purpose of this extension is precisely to provide a way to measure the execution time of real-time processes and threads, and to be able to detect and handle execution time overruns. If the POSIX.1d standard is approved, real-time systems built using a concurrent threads model will have the same capabilities that are available in traditional systems based on the cyclic executive approach.

In this paper we present an implementation of the POSIX execution time clocks and timers within an implementation of threads that follows the real-time POSIX standard very closely. We have used the Florida State University (FSU) threads implementation [7] because it is free software and the sources are available. The GNAT compilation system uses threads to implement the Ada tasks, and it can work on top of FSU threads. This means that we can access the thread/task execution time clocks and timers from the Ada application.

We will use execution time timers to detect execution time overruns. We have created a software package to encapsulate the use of the low-level POSIX primitives that are needed to handle the execution time timers and associated signals. Using this package, periodic or aperiodic hard real-time tasks written in Ada 95 can detect and limit the effects of execution time overruns in various ways, that will be described here.

In this paper we also show how to improve the throughput of sporadic server schedulers built at the application level [4]. If execution time clocks are not available, the scheduler must assume that each response to an event consumed an amount of execution capacity equal to the estimated worst-case execution time. By consuming only the actual execution time used, if the variability in execution time is large a fair amount of extra execution capacity is made available to schedule new events.

The paper is organized as follows. First, in Section 2 we present the POSIX model for execution time clocks and timers and we give details about their implementation. In Section 3 we present the design of the software package `CPU_Time` that provides operations to detect and limit execution time overruns. We also show four schemes for using this package from real-time application tasks. In section 4 we show how to take advantage of execution time clocks to implement application-level sporadic servers. In Section 5 we discuss implementation issues and we give performance metrics that help in evaluating the overhead associated with the execution time clocks. Finally, Section 6 gives our conclusions.

2 Execution Time Clocks and Timers

2.1 The Proposed POSIX Model

The execution time clocks interface defined in the proposed standard POSIX.1d [2] is based on the POSIX.1b [3] clocks and timers interface used for normal real time clocks. The new interface creates two functions to access the execution time clock identifier of the desired process or thread, respectively: *clock_getcpuclockid*() and *pthread_getcpuclockid*(). In addition, it defines a new thread-creation attribute, called *cpu_clock_requirement*, which allows the application to enable or disable the use of the execution time clock of a thread, at the time of its creation. Once the thread is created, this attribute cannot be modified. Therefore, if we want to use CPU-time clocks for threads, we must set the *cpu_clock_requirement* attribute to the value `CLOCK_REQUIRED_FOR_THREAD`.

An execution time clock “id” can be used to read or set the time using the same functions *clock_gettime*() and *clock_settime*() that are used for the standard `CLOCK_REALTIME` clock, which measures real time. In addition, timers may be created using either the `CLOCK_REALTIME` or a CPU-time clock as their time base. A POSIX timer is a logical object that measures time based upon a specified time base. The timer may be armed to expire when an absolute time is reached, or when a relative interval elapses. When the expiration time has been reached, a signal is sent to the process, to notify the timer expiration. The timer can be rearmed or disarmed at any time. In addition, it is possible to program the timer so that it expires periodically, after the first expiration.

If a timer is created using a CPU-time clock of a particular thread, and a relative expiration time is given, it can be used to notify that a certain budget of execution time has elapsed, for that thread. If the timer is armed each time a thread is activated, and the relative expiration time is set to the thread’s estimated worst-case execution time (plus some small amount to take into account the limited resolution and precision of the CPU-time clock), then the timer will only expire if the thread suffers an execution time overrun.

2.2 Implementation Details

The implementation of CPU time clocks and timers within the FSU threads requires on the one hand modification of the data structure that defines each thread, the thread control block, and on the other hand modifying the scheduler code to include the necessary steps to update each thread’s CPU-time clock and to operate the associated timers.

The information that must be added to the thread control block consists of:

- *cpu_clock_requirement*: a boolean that indicates whether the thread has a CPU-time clock enabled; it is set at the time of thread creation using the value specified in the thread attributes object used to create the thread.
- *cpuclk*: a structure with the information needed for the CPU time clock, including the clock identifier, the time of the last activation of the thread, and the total CPU-time consumed by that thread.
- *associated_timers*: an array with the information needed for each of the timers associated with that thread's CPU-time clock; this includes the timer identifier, a boolean indicating whether the timer is in use, a boolean indicating whether the timer is armed, and the timer's expiration time.

The FSU scheduler does not operate on a periodic basis, as it is invoked only at the points when the running task gets blocked or when a blocked task is activated. Thus, the modification required to support CPU time clocks and timers consists of adding code at the point where a new thread becomes the running thread. This code must: read the real-time clock storing the value as the "current time", perform the "actions for previous thread" if there was one running, and perform the "actions for new thread":

- *Actions for previous thread*: update the value of the CPU time clock by adding the difference between the current time and the activation time to the total CPU time of that thread; in addition, disarm any associated timers that were armed and had expired.
- *Actions for new thread*: store the current time as the activation time of the thread. In addition, if there are armed timers associated with this thread, calculate the time remaining until the nearest timer expiration as the difference between the minimum of the expiration times of the associated timers and the total CPU time of that thread. In this case, program an operating system timer to send a signal to the process when the calculated remaining time elapses. If there are no armed timers associated with the new thread, disarm the operating system timer. The operating system timer that we have used in our implementation is the "virtual time" timer that is accessible through the *setitimer*() OS function.

In addition, it is necessary to add the following actions at the point where a thread becomes blocked and there are no more active threads in the ready queue: read the real-time clock storing the value as the "current time", perform the "actions for previous thread", and disarming the operating system timer.

The implementation described above only works when there is just one active process in the system. It would be necessary to have a process cpu-time clock available (like the one defined in POSIX.1d) to create an implementation that would work with multiple processes.

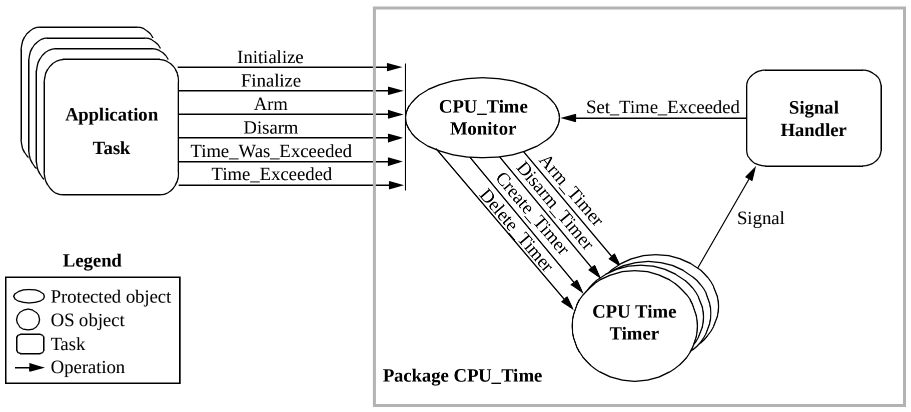

Fig. 1 Architecture of package *CPU_Time*

3 Execution Time Limits

3.1 Package CPU_Time

We have created an Ada package to encapsulate the internal aspects of the use of the POSIX interface for clocks and timers, including the use of signals to notify the occurrence of timer expirations caused by an execution time overrun. This package, called *CPU_Time*, contains the objects and operations that appear in Fig. 1.

The central part of package *CPU_Time* is a protected object called `Monitor`. This protected object has visible operations for the application tasks to initialize or finalize a CPU-time timer, to arm or disarm a timer, and to determine whether a timer has expired or not (*Time_Was_Exceeded*). In addition, *Monitor* has a family of entries (*Time_Exceeded*), one entry per timer, which can be used by application tasks to block until an execution time overrun is detected or, as we will see later, as an event that triggers the abortion of the instructions of a select statement with an abortable part.

Task *Signal_Handler* is a very high priority task that takes care of accepting all the signals generated by the different timers, identifying the particular timer, and notifying the *Monitor* protected object through operation *Set_Time_Exceeded*. This follows the recommended way of handling signals in multithreaded POSIX applications, using threads to accept signals with a *sigwait*() operation, instead of using signal handlers. The advantage is that the thread executes in a well-defined context, with well-defined scheduling behaviour, while a signal handler executes in a much more unspecified context.

3.2 Usage Schemes for CPU-Time Timers

Using package *CPU_Time*, described above, we can design different usage schemes, that depend on the particular needs of the application task whose execution time is being monitored. We have identified four major schemes:

- *Handled*: This is the case in which an execution time overrun is detected, but the task is allowed to complete its execution. This is applicable to systems under testing, or for tasks that have a high degree of criticality (and thus cannot be stopped) or for which an occasional execution time overrun can be tolerated, but needs to be reported.

  In this scheme, the application task arms the execution-time timer at the beginning of its regular execution (after initialization). At the end of its execution, it uses function *Time_Was_Exceeded* to determine whether the execution time was exceeded or not. If an overrun is detected, the error can be reported. Table 1 shows the pseudocode of the application task under this scheme.

Table 1. Periodic task under the “handled” scheme.

```
task body Periodic_Handled is
   Timer_Id : CPU_Time.Monitor_Id;
begin
   CPU_Time.Monitor.Initialize(Timer_Id);
   loop
      CPU_Time.Monitor.Arm(Timer_Id, Worst_Case_Exec_Time);
      Do Task’s Useful Work;
      if CPU_Time.Monitor.Time_Was_Exceeded(Timer_Id) then
         Handle the Error;
      end if;
      CPU_Time.Monitor.Disarm(Timer_Id);
      delay until Next_Start;
   end loop;
end Periodic_Handled;
```

- *Stopped*: This is the case in which if an execution time overrun is detected, the associated task execution is stopped, to allow lower priority tasks to execute within their deadlines. The whole instance of the stopped task is aborted and is never repeated. The task itself waits until its next activation and then proceeds normally.

  The implementation of this scheme consists of executing the regular instructions of the task inside the abortable part of an asynchronous select statement. The event that triggers abortion in this case is a call to the entry *Time_Exceeded* of *CPU_Time.Monitor.* As a consequence, if an execution time overrun occurs, the task instructions are aborted for that instance of the task execution. The pseudocode of the application task under this scheme is shown in Table 2.

- *Imprecise*: This scheme corresponds to the case in which the task is designed using the imprecise computation model [6], in which the task has a mandatory part (generally short and for which it is easier to estimate a worst-case execution time), and an optional part that refines the calculations made by the task. Since the worst-case execution time of this optional part is usually more difficult to estimate, this part will be aborted if an execution time overrun is detected. This allows us to use

Table 2. Application task under the "Stopped" scheme

```
task body Periodic_Stopped is
   Timer_Id : CPU_Time.Monitor_Id;
begin
   CPU_Time.Monitor.Initialize(Timer_Id);
   loop
      CPU_Time.Monitor.Arm(Timer_Id, Worst_Case_Exec_Time);
      select
         CPU_Time.Monitor.Time_Exceeded(Timer_Id);
         Handle the Error;
      then abort
         Do Task's Useful Work;
      end select;
      CPU_Time.Monitor.Disarm(Timer_Id);
      delay until Next_Start;
   end loop;
end Periodic_Stopped;
```

fixed priority scheduling in applications in which the optional part has an unpredictable execution time. The technique is also valid for cases in which the optional part continuously refines the quality of the results; we can let the optional part run until it exhausts its execution time budget, and then use the last valid result obtained. The implementation of this scheme consists of using the "handled" approach for the mandatory part of the task, and the "stopped" approach for the optional part. After the optional part, whether it is aborted or not, another mandatory part may exist to cause the outputs of the task to be generated. Table 3 shows the pseudocode of the application task under this scheme.

Table 3. Application task under the "Imprecise" scheme

```
task body Periodic_Imprecise is
   Timer_Id : CPU_Time.Monitor_Id;
begin
   CPU_Time.Monitor.Initialize(Timer_Id);
   loop
      CPU_Time.Monitor.Arm(Timer_Id,Worst_Case_Exec_Time_I);
      Do Task's Mandatory Part I;
      select
         CPU_Time.Monitor.Time_Exceeded(Timer_Id);
      then abort
         Do Task's Optional Part;
      end select;
      CPU_Time.Monitor.Disarm(Timer_Id);
      Mandatory Part II: Generate Task's Outputs;
      delay until Next_Start;
   end loop;
end Periodic_Imprecise;
```

- *Lowered*: This scheme can be used to limit the effects of an execution time overrun of a particular task, on lower priority tasks, when asynchronous select statements are not allowed or are not available for an application task. In this case, when the overrun is detected, the priority of the task is lowered to a background level, lower than the priorities of all real-time tasks. When the task that overrun its execution time has the opportunity to finish its execution, it can determine that it overrun by invoking *Time_Was_Exceeded*, and then it can take a corrective action or report the error; if it wishes so, it can raise its priority back to its normal level.

  This scheme requires a different implementation of package *CPU_Time*. In the new implementation, operation *Initialize* must store the task identifier of the calling task. This identifier is then used to lower the priority of the task when an execution time overrun is detected. This is done by the signal handler task (see Fig. 1) by using the facilities of package *Ada.Dynamic_Priorities*.

4 Enhancing Sporadic Server Schedulers

In [4] we presented a number of application-level implementations of the sporadic server scheduling algorithm. This algorithm is designed to schedule aperiodic activities in hard real-time systems, while bounding the effects of these activities on lower priority tasks. The sporadic server scheduler is based on keeping record of the amounts of execution time consumed by the aperiodic activities, and allowing them to consume only a certain execution time *capacity* during an interval of time called the *replenishment period*. In the implementations presented in [4] we always assumed that the consumed execution time for processing one event was equal to the worst-case execution time. In addition, we would only allow the aperiodic task to run at its normal priority level if the available execution capacity was at least equal to the worst-case execution time. Now, we can take advantage of execution time clocks and timers to enhance the performance of the sporadic server schedulers by eliminating both restrictions, as we describe in the following two subsections. Both enhancements increase the throughput of the sporadic server scheduler, while still preserving the schedulability of lower priority tasks in the system.

4.1 Accounting for the Execution Time Spent

In this first enhancement, we account for the actual execution time spent, instead of assuming that the worst-case execution time was spent. This makes sense in the sporadic server schedulers that allow *multiple* events per replenishment period. The sporadic server implementations do not need to change; we only need to change the application task to use a CPU-time clock to measure the execution time spent during the response to each event, and to pass the actual time spent to the sporadic server scheduler, as a parameter to the *Schedule_Next* operation. In tasks that have a high variability of their execution time, this approach allows the scheduler to save execution capacity for processing future events. The pseudocode of the application task for this case is shown in Table 4.

Table 4. Aperiodic task under a sporadic server scheduler

```
task body Application_Task is
   SS : Sporadic_Server.Scheduler;
   Last_Time, Now : POSIX.Timespec;
begin
   Sporadic_Server.Initialize(SS);
   Last_Time:=POSIX_Timers.Get_Time(pthread_getcpuclockid);
   loop
      Sporadic_Server.Prepare_To_Wait(SS);
      Wait for Event;
      Sporadic_Server.Prepare_To_Execute(SS);
      Do Task's Useful Work;
      Now:=POSIX_Timers.Get_Time(Thread_CPUtime_Clock);
      Sporadic_Server.Schedule_Next
            (SS, Spent=> Now-Last_Time);
      Last_Time:=Now;
   end loop;
end Application_Task;
```

4.2 Detecting When the Execution Capacity Gets Exhausted

In this second enhancement, we create one execution time timer per sporadic server scheduler, to measure the consumption of execution time, and to detect the case in which the sporadic server runs out of execution capacity. This is particularly useful in the schedulers using a replenishment manager like the *background* manager, that allows the application task to run at a background priority level when it does not have enough capacity available. Each time the aperiodic task is activated, we arm the timer with an expiration time equal to the available execution capacity. The aperiodic task is allowed to execute at its normal priority level. If the application task consumes all of the available capacity, the execution time timer expires, sending the associated signal. A signal handler task shared by all the schedulers, similar to the one shown in Fig. 1, accepts this signal and invokes the sporadic server protected manager to lower the priority of the application task, and schedule the corresponding replenishment operation. The advantage of using the execution time timer in this case is that we can use all of the available capacity; before, we could only use it if at the activation of the task the available capacity was larger than or equal to the worst-case execution time of the task.

Fig. 2 shows the basic architecture of the new replenishment manager called *CPU-time* manager. This manager is derived from the *Queued* manager and has the same attributes as the *Background* manager, except for a different *Protected_Manager* and with the addition of a new attribute to hold the CPU-time timer. The new *CPU-time* manager can be used by the *Simple*, *High_Priority*, and *High_Priority_Polled* sporadic server schedulers, giving way to three new implementations.

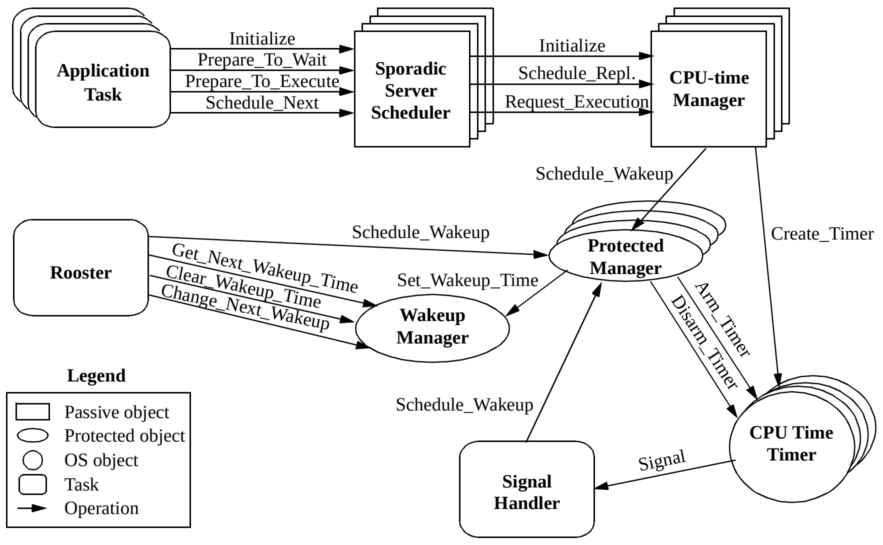

Fig. 2. Basic architecture of the new implementations using the *CPU-time* manager

5 Performance Considerations

The use of execution time clocks affects the performance of the system because it adds a fixed overhead to each context switch. There are two different classes of thread scheduler implementations that must be considered:

- *Ticker implementation*. In this implementation a periodic signal or interrupt activates the task scheduler. A simple way to count execution time in this case is to have an integer value counter for each thread and, each time the tick interrupt comes in, increment the counter associated to the task that was running. The resolution of such clock is one tick, but it is relatively cheap, since it only involves an addition operation during each tick.

- *Alarm clock implementation*. In this implementation, each task is allowed to run until it gives up the processor. A time-ordered delay queue is used to store the next activation time for timed tasks that are currently blocked. An alarm clock is set to invoke the task scheduler when the first task of the delay queue needs to be activated. In this case, the scheduler is not invoked periodically, and thus at each invocation it must read a hardware clock to determine how much time has elapsed since the last scheduling operation. This time must be added to the execution time of the task that was just running. Execution time clocks may be expensive in this implementation, depending on how much time it takes to read the hardware clock.

The FSU threads that we have used in our implementation of CPU-time clocks and timers follow the “alarm clock” model, and thus at each context switch we need to read the real-time clock and to program an operating system timer to take care of CPU-time timer expirations. We have measured the overhead associated with each context switch by creating a task that reads the clock continuously, and a second higher priority task that preempts the former task periodically, and finishes immediately. In this way, by measuring the difference between the two clock readings before and after the preemption from the higher priority task, we can determine the time required to perform two context switches (and thus dividing by two, we obtain the time for one context switch). The results of the time required for one context switch are shown in Table 5, in microseconds, for three experiments with different numbers of tasks. We can see that the overhead is between 13 to 19 microseconds per context switch, which represents an increase between 10% and 17%. This increase should be acceptable for most real-time applications, since tasks usually run at frequencies smaller than one kilohertz.

Table 5. Comparison of context switch times (μs)

Experiment	FSU Threads	FSU threads with CPU-time Clocks			FSU with CPU-time Clocks and Timers		
	C_s	C_s	Δ	%Δ	C_s	Δ	%Δ
2 tasks	110	128	18	14%	129	19	17%
2+10 tasks (low priority)	117	130	13	10%	132	15	13%
2+10 tasks (high priority)	115	130	15	12%	131	16	14%

Table 6 shows the results of average execution time measured for the different POSIX CPU-time clock services that we have implemented, as well as the times for the operations of package *CPU_Time*. All results have been obtained in a Pentium-133 CPU running under Linux.

Table 6. Execution times of the different operations (μs)

Posix_Timers	μs	CPU_Time	μs
Get_Time(pthread_getcpuclockid)	4	Monitor.Arm	62
Get_Time(CLOCK_REALTIME)	4	Monitor.Disarm	55
Arm_Timer	12	Monitor.Time_Was_Exceeded	12
Disarm_Timer	12	Monitor.Set_Time_Exceeded	35
		Monitor.Time_Exceeded	84

6 Conclusion

We have discussed the importance of enforcing the worst-case execution estimates in hard real-time systems. In an application designed under a concurrent tasking architecture, such as in the Ada tasking model, detecting and limiting execution time overruns may be achieved by using the proposed POSIX model for execution time clocks and timers. In this paper we describe how to implement these CPU-time clocks in the context of a POSIX threads implementation.

We have also described an implementation scheme that allows application tasks to use the POSIX execution time clocks to detect execution time overruns and limit their effects. We have also described how to take advantage of execution time clocks to enhance the behaviour of application-level sporadic servers. As a guide to users of these clocks, we have discussed some performance considerations that allow the application developer to determine the amount of overhead that he or she will suffer by using execution time clocks, for a particular application.

References

[1] A. Burns and A. Wellings. “Real-Time systems and Programming Languages”. 2nd. edition. Addison-Wesley, 1997.

[2] IEEE Standards Project P1003.1d, “Draft Standard for Information Technology -Portable Operating System Interface (POSIX)- Part 1: Additional Realtime Extensions”. Draft 10. The Institute of Electrical and Electronics Engineers, January 1997.

[3] ISO/IEC Standard 9945-1:1996. “Information Technology -Portable Operating System Interface (POSIX)- Part 1: System Application Program Interface (API) [C Language]”. The Institute of Electrical and Electronics Engineers, 1996.

[4] M. González Harbour, J.J. Gutiérrez García, and J.C Palencia Gutiérrez. “Implementing Application-Level Sporadic Server Schedulers in Ada 95”. Proceedings of the 1997 Ada-Europe International Conference on Reliable Software Technologies, in Lecture Notes in Computer Science, Vol. 1251, Springer, June 1997.

[5] M.H. Klein, T. Ralya, B. Pollak, R. Obenza, and M. González Harbour. “A Practitioner’s Handbook for Real-Time Analysis”. Kluwer Academic Pub., 1993.

[6] J. Liu, K.J. Lin, W.K. Shih, A. Chuang-Shi Yu, J.Y. Chung, and W. Zhao. “Algorithms for Scheduling Imprecise Computations”. IEEE Computer, pp. 58-68, May 1991.

[7] F. Mueller. “A Library Implementation of POSIX Threads under UNIX”. 1993 Winter USENIX, January, 1993, San Diego, CA, USA.

[8] F. Mueller. “Generalizing Timing Predictions to Set-Associative Caches”. Proceedings of the 9th Euromicro Workshop on Real-Time Systems, pp. 64-71, Toledo, Spain, June 1997.

[9] C.Y. Park. “Predicting program execution times by analyzing static and dynamic program paths”. Real-Time Systems Journal, Vol. 5, No. 1, pp. 31-62, 1993.

[10] P. Puschner and A.V. Schedl. “Computing Maximum Task Execution Times: A graph-based approach”. Real-Time Systems Journal, Vol. 13, No. 1, pp. 67-91, July 1997.

[11] N. Zhang, A. Burns, and M. Nicholson. “Pipelined Processors and Worst-Case Execution Times”. Real-Time Systems Journal, Vol. 5, No. 1, pp. 31-62, 1993.