Application-Defined Scheduling in Ada

Mario Aldea Rivas and Michael González Harbour

Departamento de Electrónica y Computadores
Universidad de Cantabria
39005-Santander, SPAIN
{aldeam,mgh}@unican.es

***Abstract**: This paper presents an application program interface (API) that enables Ada applications to use application-defined scheduling algorithms in a way compatible with the scheduling model of the Ada 95 Real-Time Systems Annex. Several application-defined schedulers, implemented as special user tasks, can coexist in the system in a predictable way. This API is currently implemented on our operating system MaRTE OS.*

Keywords: *Real-Time Systems, Kernel, Scheduling, Operating Systems, Ada 95, POSIX.*

1. Introduction[1]

The Real-Time Annex in Ada 95 defines only one scheduling policy: `FIFO_Within_Priorities`. Real-time POSIX [3][5] also defines a preemptive fixed priority scheduling policy similar to it, together with two other compatible scheduling policies: a round robin within priorities policy and the sporadic server policy. These policies are accessible to Ada tasks running on a POSIX-compliant real-time operating system. Although fixed priority scheduling is an excellent choice for real-time systems, there are application requirements that cannot be fully accomplished with these policies only. It is well known that with dynamic priority scheduling policies it is possible to achieve higher utilization levels of the system resources than with fixed priority policies. In addition, there are many systems for which their dynamic nature make it necessary to have very flexible scheduling mechanisms, such as multimedia systems, in which different quality of service properties need to be traded against one another.

It could be possible to incorporate into the Ada and POSIX standards new dynamic scheduling policies to be used in addition to the existing policies. The main problem is that the variety of these policies is so great that it would be difficult to standardize on just a few. Different applications needs would require different policies. Instead, in this paper we propose defining an interface for application-defined schedulers that could be used to implement a large variety of scheduling policies. This interface is integrated into the POSIX Ada Binding [5] and, together with its C language version, it is being submitted for consideration by the Real-Time POSIX Working Group.

The proposed interface is currently implemented in our operating system MaRTE OS (Minimal Real-Time Operating System for Embedded Applications) [1]. MaRTE OS is a real-time kernel for embedded applications that follows the Minimal Real-Time POSIX.13 subset [4], providing both the C and Ada language POSIX interfaces. It allows cross-development of Ada and C real-time applications. Mixed Ada-C applications can also be developed, with a globally consistent scheduling of Ada tasks and C threads. It is directly usable as the basis for the gnat run-time system (GNARL) [8].

The paper is organized as follows: Section 2 discusses some related work on application-defined scheduling and sets the justification for our proposal. Section 3 discusses our model for application-defined scheduling. In Section 4 the Ada API is described and in Section 5 an example of its use is shown. Section 6 presents some performance metrics showing the overhead of using our implementation. Section 7 gives our conclusions and future work.

2. Related Work and Motivation

The idea of application-defined scheduling has been used in many systems. A solution is proposed in RED-Linux [10], in which a two-level scheduler is used, where the upper level is implemented as a user process that maps several quality of service parameters into a low-level attributes object to be handled by the lower level scheduler. The parameters defined are the task priorities, start and finish

1. This work has been funded by the *Comisión Interministerial de Ciencia y Tecnología* of the Spanish Government under grants TIC99-1043-C03-03 and 1FD 1997-1799 (TAP)

times, and execution time budget. With that mechanism some scheduling algorithms can be implemented but there may be others that cannot be implemented if they are based on parameters different from those included in the aforementioned attributes object. In addition, this solution does not address the implementation of protocols for shared resources that could avoid priority inversion or similar effects.

A different approach is followed in the CPU Inheritance Scheduling [6], in which the kernel only implements task blocking, unblocking and CPU donation, and the application defined schedulers are tasks which donate the CPU to other tasks. In this approach the only method used to avoid priority inversion is the priority inheritance, which may be a limitation for special application-defined policies. In addition this approach is not designed for multiprocessors, because only one task can execute at a time.

Another common solution is to implement the application scheduling algorithms as modules to be included or linked with the kernel (S.Ha.R.K [7], RT-Linux [11], Vassal [2]). With this mechanism the functions exported by the modules are invoked from the kernel at every scheduling point. This is a very efficient and general method, but it has the drawback that the application scheduling algorithms can neither be isolated from each other nor from the kernel itself, so, a bug in one of them could affect the whole system.

In our approach the application scheduler is invoked at every scheduling point like with the kernel modules, so the scheduler can have complete control over its scheduled tasks. But in addition, our application scheduling algorithm is executed by a user task. This fact implies two important advantages from our point of view:

a) The system reliability can be improved by protecting the system from the actions of an erroneous application scheduler. For efficiency, our interface allows execution of the application-defined scheduler in an execution environment different than that of regular application tasks, for example inside the kernel. But alternatively, the interface allows the implementation to execute the scheduler in the environment of the application, to isolate it from the kernel.

b) The application scheduling code can use standard interfaces like those defined in the POSIX standard. In some systems part of these interfaces might not be accessible for invocation from inside the kernel.

We have designed our interface so that several application-defined schedulers can be defined, and so that they have a behaviour compatible with other existing scheduling policies in POSIX, both on single processor and multiprocessor platforms. In addition, the interface needs to take into account the implementation of application-defined synchronization protocols.

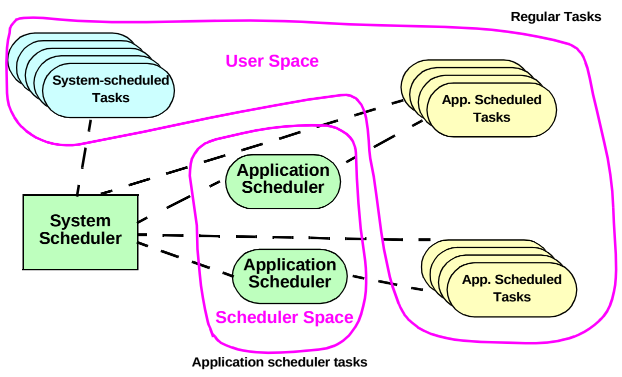

Figure 1. Model for Application Scheduling

One of the design criteria for our Ada interface has been to avoid requiring changes to existing run-time systems or to the compilers. This is important to make it easier to implement the interface and thus increase the chances that it is widely used.

The dynamic scheduling mechanism proposed for Real-Time CORBA 2.0 [9] represents an object-oriented interface to application-defined schedulers, but it does not attempt to define how that interface communicates with the operating system. The interface presented in this paper is the OS low-level interface, and thus an RT CORBA implementation could use it to support the proposed dynamic scheduling interface.

In summary, the motivation for this work is to provide developers of applications running on top of standard operating systems (POSIX) with a flexible scheduling mechanism, handling both task scheduling and synchronization, that enables them to schedule dynamic applications that would not meet their requirements using the more rigid fixed-priority scheduling provided in those operating systems. This mechanism allows isolation of the kernel from misbehaved application schedulers. In addition, we wish to provide this mechanism both for applications developed in C or Ada.

3. Model for Application-Defined Scheduling

Figure 1 shows the proposed approach for application-defined scheduling. Each application scheduler is a special kind of task, that is responsible of scheduling a set of tasks that have been attached to it. This leads to two classes of tasks in this context:

- *Application scheduler tasks*: special tasks used to run application schedulers.
- *Regular tasks*: regular application tasks

The application schedulers can run in the context of the kernel or in the context of the application. This allows implementations in which application tasks are not trusted, and therefore their schedulers run in the context of the application, as well as implementations for trusted application schedulers, which can run more efficiently inside the kernel. Because of this duality we will model the scheduler tasks as if they run in a separate context, which we call the scheduler space. The main implication of this separate space is that for portability purposes the application schedulers cannot directly share information with the kernel, nor with regular tasks, except by using the POSIX shared memory objects, which is the mechanism for sharing memory among entities with different address spaces.

According to the way a task is scheduled, we can categorize the tasks as:

- *System-scheduled tasks*: these tasks are scheduled directly by the run-time system and/or the operating system, without intervention of a scheduler task.
- *Application-scheduled tasks*: these tasks are also scheduled by the run-time system and/or the operating system, but before they can be scheduled, they need to be activated by their application-defined scheduler.

Although an application scheduler task can itself be application scheduled, implementations should not be required to support this feature, because usually these tasks will be system scheduled.

Because the use of protected resources may cause priority inversions or similar delay effects, it is necessary that the scheduler task knows about their use, to establish its own protocols adapted to the particular task scheduling policy. As we mentioned above, one of the goals of our interface is that it does not require changes to existing Ada run-time systems, and this means that we cannot change the priority ceiling locking defined in Ada's Real-Time Annex for protected objects. For this reason, we have designed our scheduling interface around the POSIX mutexes, which are accessible through the POSIX Ada Bindings. The application can still use protected objects if the `Ceiling_Locking` policy is acceptable for the chosen scheduling policy, and also for synchronization between system- and application-scheduled tasks. Figure 2 shows the basic model of these scheduling interfaces, in which two kinds of mutexes will be considered:

- *System-scheduled mutexes*. Those created with the current POSIX protocols: `No_Priority_Inheritance`, `Highest_Ceiling_Priority` (immediate priority ceiling), or `Highest_Blocked_Task` (basic priority inheritance). They can be used to access resource shared between application schedulers, between sets of application-scheduled threads attached to different schedulers, or even between an application scheduler and its scheduled threads (in this case the mutex and its protected state must be placed in a POSIX shared memory object).
- *Application-scheduled mutexes*: Those created with the protocol `Appsched_Protocol`. The behaviour of the protocol itself is defined by the application scheduler.

3.1. Relations with Other Tasks

Each task in the system, whether application- or system-scheduled, has a system priority:

- For system-scheduled tasks, the system priority is the priority assigned using `pragma Priority` or the subprogram `Ada.Dynamic_Priorities.Set_Priority`, possibly modified by the inheritance of other priorities through the use of mutexes or protected objects.
- For application-scheduled tasks, the system priority is lower than or equal to the system priority of their scheduler. The system priority of an application-scheduled task may change because of the inheritance of other system priorities through the use of mutexes or protected objects. In that case, its scheduler also inherits the same system priority (but this priority is not inherited by the rest of the tasks scheduled by that scheduler). In addition to the system priority, application-scheduled tasks have *application scheduling parameters* that are used to schedule that task contending with the other tasks attached to the same application scheduler. The system priority always takes precedence over any application scheduling parameters. Therefore,

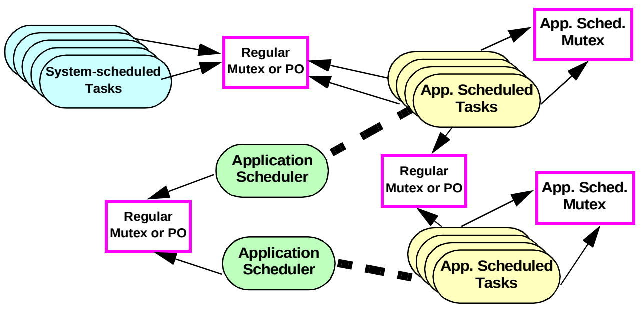

Figure 2. Model for Application-Defined Synchronization

application-scheduled tasks and their scheduler take precedence over tasks with lower system priority, and they are always preempted by tasks with higher system priority that become ready. The scheduler always takes precedence over its scheduled tasks.

If application-scheduled tasks coexist at the same priority level with other system-scheduled tasks, then the Real-Time Annex scheduling rules apply as if the application-scheduled tasks were scheduled under the `FIFO_Within_Priorities` policy; so a task runs until completion, until blocked, or until preempted, whatever happens earlier. Of course, in that case the interactions between the different policies may be difficult to analyse, and thus the normal use will be to have the scheduler task and its scheduled tasks running at an exclusive range of system priorities.

In the presence of priority inheritance, the scheduler inherits the same priorities as its scheduled tasks, to prevent priority inversions from occurring. This means that high priority tasks that share resources with lower system-priority application tasks must take into account the scheduler overhead when accounting for their blocking times.

3.2. Relations Between the Scheduler and its Attached tasks

When an application-defined task is attached to its application scheduler, the latter has to either accept it or reject it, based upon the current state and the scheduling attributes of the candidate task. Rejection of a task causes `POSIX_Error` to be raised during the attachment process.

Each application-defined scheduler may activate many application-scheduled tasks to run concurrently. The scheduler may also block previously activated tasks. Among themselves, concurrently scheduled tasks are activated like `FIFO_Within_Priorities` tasks. As mentioned previously, the scheduler always takes precedence over its scheduled tasks.

For an application-scheduled task to become ready it is necessary that its scheduler activates it. When the application task executes one of the following actions or experiences one of the following situations, a scheduling event is generated for the scheduler, unless the scheduling event to be generated is being filtered out (discarded).

- when a task requests attachment to the scheduler
- when a task blocks or gets ready
- when a task changes its scheduling parameters
- when a task invokes the *yield* operation (i.e., `delay 0.0`)
- when a task explicitly invokes the scheduler
- when a task inherits or uninherits a priority, due to the use of a system mutex or a protected object
- when a task does any operation on a application-scheduled mutex.

The application scheduler is a special task whose code is usually a loop where it waits for a scheduling event to be notified to it by the system, and then determines the next application task to be activated.

The scheduler being a single task implies that its actions are all sequential. For multiprocessor systems this may seem to be a limitation, but for these systems several schedulers could be running simultaneously on different processors, cooperating with each other. For single processor systems the sequential nature of the scheduler should be no problem. Again, it is possible to have several scheduler tasks running at the same time, and cooperating with each other by synchronizing through regular mutexes and condition variables, or protected objects.

4. Interface

The interface offered to the programmer is divided into three packages:

- `POSIX_Application_Scheduling`: a completely new package where the main part of the interface is defined. This package defines the scheduling events and actions and also operations to manage scheduler and scheduled tasks.
- `POSIX_Mutexes`: This is an existing package in the POSIX Ada Bindings, to which we have added the interface for management of application-scheduled mutexes.
- `POSIX_Timers`: This is also an existing package in the POSIX Ada Bindings, to which we have added the CPU-Time clocks. This functionality is not yet included in the POSIX Ada binding but it has already been standardized in the C family of POSIX standards [3]. It is very interesting for implementing scheduling algorithms based on the execution time consumed by the tasks, such as the sporadic server, round robin, and others.

The main elements of the interface are in package `POSIX_-Application_Scheduling`, and they are described in detail in the following subsections.

4.1. Interfaces for the Scheduler Tasks

4.1.1. Scheduling Events

The interface describes an abstract data type for the scheduling events that the system may report to a scheduler task. The system stores the scheduling events in a FIFO queue until processed by the scheduler. The information included in these events is:

- the event code,
- the identifier of the task that caused the event,
- and additional information associated with the event, and dependent on it; it can be an inherited (or uninherited) system priority, information related to an accepted signal, a pointer to an application-scheduled mutex, or application-specific information.

The specific events that may be notified to a scheduler task are shown in Table 1.:

Table 1: Scheduling Events

Event Code	Description	Additional info
NEW_TASK	A new task has requested attachment to the scheduler	none
TERMINATE_-TASK	A task has been terminated	none
READY	A task has become unblocked by the system	none
BLOCK	A task has blocked	none
YIELD	A task yields the CPU (due to a delay 0.0)	none
SIGNAL	A signal belonging to the requested set has been accepted by the scheduler task.	Signal-related information
CHANGE_-SCHED_PARAM	A task has changed its scheduling parameters	none
EXPLICIT_-CALL	A task has explicitly invoked the scheduler	Application message
TIMEOUT	A timeout has expired	none
PRIORITY_-INHERIT	A task has inherited a new system priority due to the use of system mutexes or protected objects	Inherited system priority
PRIORITY_-UNINHERIT	A task has finished the inheritance of a system priority	Uninherited system priority
INIT_MUTEX	A task has requested initialization of an application-scheduled mutex	Pointer to the mutex
DESTROY_-MUTEX	A task has destroyed an application-scheduled mutex	Pointer to the mutex
LOCK_MUTEX	A task has invoked a “lock” operation on an available application- scheduled mutex	Pointer to the mutex
TRY_LOCK_-MUTEX	A task has invoked a “try lock” operation on an available application- scheduled mutex	Pointer to the mutex
UNLOCK_-MUTEX	A task has released the lock of an application-scheduled mutex	Pointer to the mutex
BLOCK_AT_-MUTEX	A task has blocked at an application-scheduled mutex	Pointer to the mutex
CHANGE_-MUTEX_-SCHED_PARAM	A task has changed the scheduling parameters of an application-scheduled mutex	Pointer to the mutex

There are some Ada-specific events that might have been useful to a scheduler, but that are not available because the underlying POSIX standard does not know about them. For example, a task becoming completed, or a task blocked waiting for a child to elaborate. In the schedulers that we have built, these situations can be handled through the usual `BLOCK` event. If more accurate handling of these events is desired, cooperation from the application developer would be required, by establishing additional communication between the scheduled thread and its scheduler, via `Invoke_Scheduler` (see section 4.2) and/or by accessing shared data structures.

4.1.2. Executing Scheduling Actions

Another important abstract data type is the interface for storing a list of scheduling actions. This list will be prepared by the scheduler task and reported to the system via a call to one of the `Execute_Actions` operations. The possible actions that can be added to one of these lists are the following:

- Accept or reject a task that has requested attachment to this scheduler
- Activate or suspend an application scheduled task
- Accept or reject initialization of an application-scheduled mutex
- Grant the lock of an application-scheduled mutex

The main operations of our interface are the `Execute_Actions` family of procedures, which allow the application scheduler to execute a list of scheduling actions and then wait for the next scheduling event to be reported by the system. If desired, a timeout can be set as an additional return condition which will occur when there is no scheduling event available but the timeout expires. The system time measured immediately before the procedure returns can be requested if it is relevant for the algorithm.

The `Execute_Actions` procedures can also be programmed to return when a POSIX signal is generated for the task. This possibility eases the use of POSIX timers, including CPU-time timers, as sources of scheduling events for our scheduler tasks. Use of CPU-time timers allows the scheduler to impose limits on the execution time that a particular task may spend. The specification of the `Execute_Actions` procedures is:

```
procedure Execute_Actions
   (Sched_Actions : in  Scheduling_Actions;
    Set           : in  POSIX_Signals.Signal_Set;
    Event         : out Scheduling_Event;
    Current_Time  : out POSIX.Timespec);

procedure Execute_Actions_With_Timeout
   (Sched_Actions : in  Scheduling_Actions;
    Set           : in  POSIX_Signals.Signal_Set;
    Event         : out Scheduling_Event;
    Timeout       : in  POSIX.Timespec;
    Current_Time  : out POSIX.Timespec);
```

Other overloaded versions of these procedures with less parameters are available for the cases in which the application is not interested in obtaining the current time, or is not interested in waiting for a signal to be received.

4.1.3. Other Interfaces

The `Become_An_Application_Scheduler` procedure allows a regular task to become an application scheduler:

```
procedure Become_An_Application_Scheduler;
```

The ideal interface for this purpose would be a pragma that would enable the task to be created as an application scheduler from the beginning, but that would imply a modification to the compiler.

After the task has become an application scheduler it can set its attributes through different operations in the interface. The attributes are the kind of timeout that is supported (relative or absolute), the clock used to determine when the timeout expires, and the event mask that allows scheduling events to be discarded by the system, and thus are not reported to the application scheduler. A subset of the scheduler attributes interface is shown below:

```
procedure Set_Clock
   (Clock : in POSIX_Timers.Clock_Id);
procedure Set_Flags
   (Flags : in Scheduler_Flags);
type Event_Mask is private;
procedure Set_Event_Mask
   (Mask : in Event_Mask);
```

4.2. Interfaces for the Application-Scheduled Tasks

The `Change_Task_Policy_To_App_Sched` procedure allows a regular task to set its scheduling policy to application-scheduled, attaching itself to a particular scheduler. The scheduling parameters associated with the task can be set via the `Change_Task_Policy` procedure of the generic package `Application_Defined_Policy`. In this package the generic parameter is the scheduling `Parameters` type, which of course must be defined by the application.

```
procedure Change_Task_Policy_To_App_Sched
   (Scheduler : in Task_Identification.Task_Id);

generic
   type Parameters is private;
package Application_Defined_Policy is
   procedure Change_Task_Policy
     (Scheduler : in Task_Identification.Task_Id;
      Param     : in Parameters);
   procedure Get_Parameters
     (T     : in  Task_Identification.Task_Id;
      Param : out Parameters);
end Application_Defined_Policy;
```

The `Invoke_Scheduler` procedure allows a scheduled task to explicitly invoke its scheduler. A generic version of this procedure is also offered for the case in which the scheduled task needs to send a message with information to its scheduler. The type of this message is the generic parameter.

```
procedure Invoke_Scheduler;

generic
   type Message is private;
package Explicit_Scheduler_Invocation is
   procedure Invoke (Msg : in Message);
   function Get_Message
      (Event : in Scheduling_Event)
       return Message;
end Explicit_Scheduler_Invocation;
```

An interface for handling task-specific data has been added. At first we had planned using the task-specific data interface defined in package `Ada.Task_Parameters` defined in Ada 95 Annex C, but we found out that this package did not allow us handling tasks that had terminated, and it is usually an important part of the application scheduler to do cleanup operations after a task terminates. For this reason we have defined our own generic package for task-specific data that may be shared between the scheduler and its scheduled tasks.

5. Example of an Application-Defined Policy: EDF

The following example shows the pseudocode of a set of periodic tasks scheduled under an application-defined Earliest Deadline First (EDF) scheduling policy. In first place we define a data type for the scheduling parameters, and we instantiate the generic package `Application_Defined_Policy` using this type:

```
type EDF_Parameters is record
   Deadline, Period : POSIX.Timespec;
end record;
package EDF_Policy is new
   Application_Defined_Policy (EDF_Parameters);
```

The scheduler can be in one of the three following states: `Idle`, when there are no tasks registered to be scheduled; `Waiting`, when there are tasks registered but none active; and `Running`, when there are one or more active tasks:

```
type EDF_State is (Idle, Waiting, Running);
```

The EDF scheduler has a list of tasks that are registered for being scheduled under it. Each of these tasks can be active, blocked, or timed (when it has finished its current execution and is waiting for its next period):

```
type Scheduled_Task_State is
   (Active, Blocked, Timed);
```

The scheduler uses the following operations:

- The `Schedule_Next` procedure uses the current time and the list of registered tasks to update the list, calculate the current state, the next task to be executed, and the earliest start time of the set of active tasks (not including the next one to run).
- The `Add_To_List_Of_Tasks` procedure adds a new task to the list of scheduled tasks.
- The `Eliminate_From_List_of_tasks` procedure eliminates a terminated task from the list of scheduled tasks.
- The `Make_Active` procedure changes the state of a task in the list to `Active`.
- The `Make_Blocked` procedure changes the state of a task in the list to `Blocked`.
- The `Make_Timed` procedure changes the state of a task in the list to `Timed`.

The pseudocode of the scheduler is the following:

```
task body EDF_Scheduler is
   Sch_State : EDF_State;
   Current_Task : Task_Id;
   Now, Earliest_Start : POSIX.Timespec;
   Event : Scheduling_Event;
   Sched_Actions : Scheduling_Actions;
begin
   Become_An_Application_Scheduler;
   Set_Clock(Clock_Realtime);
   Set_Flags(Absolute_Timeout);
   Now:=Get_Time(Clock_Realtime);
   loop
      Schedule_Next(Sch_State, Current_Task,
                    Earliest_Start, Now);
      case Sch_State is
         when Idle =>
            Execute_Actions
               (Null_Sched_Actions, Event, Now);
         when Waiting =>
            Execute_Actions_with_Timeout
               (Null_Sched_Actions, Event,
                Earliest_Start,Now);
         when Running =>
            Add_Activate
               (Sched_Actions,Current_Task)
            Execute_Actions_with_Timeout
               (Sched_Actions, Event,
                Earliest_Start,Now);
      end case;
      -- process scheduling events
      case Get_Event_Code(Event) is
         when New_Task =>
            Add_To_List_of_Tasks
               (Get_Task(Event),now);
         when Terminate_Task =>
            Eliminate_From_List_of_Tasks
                (Get_Task(Event));
         when Ready =>
            Make_Active(Get_Task(Event));
         when Block =>
            Make_Blocked(Get_Task(Event));
         when Explicit_Call =>
            Make_Timed(Get_Task(Event));
         when Timeout => null;
         when Others => null;
      end case;
   end loop;
end EDF_Scheduler;
```

Finally, the pseudocode of one of the application-scheduled tasks is the following:

```
task body Periodic_Task is
   Param : EDF_Parameters:=(Deadline,Period);
begin
   EDF_Policy.Change_Task_Policy
         (Task_Id_of(EDF_Scheduler),Param);
   loop
      -- do useful work
      Invoke_Scheduler; -- task will wait
   end loop;          -- until the next period
end Periodic_Task;
```

6. Performance Metrics

The final paper will include performance metrics on the use of different application-defined scheduling policies. Preliminary results show quite promising results. For a round robin scheduler the context switch between two tasks was 2.9 microseconds on a 550 MHz Pentium III. This is about three times the context switch for regular fixed-priority tasks, and is a reasonable number if we take into account that now to execute one task the scheduler task also has to execute. The result is efficient enough to be able to run high frequency tasks that are scheduled by application defined policies.

7. Conclusions and Further Work

We have defined a new API for application-defined scheduling. There are two versions of the API, one in C and the other one in Ada. Both are designed in the context of a POSIX operating system. The main design requirements have been to have a compatible behaviour with other existing POSIX fixed priority scheduling policy, to be able to isolate the scheduler from the kernel and from other application schedulers, to be able to run both on single processor and multiprocessor systems, and to be able to describe application-defined synchronization protocols.

The proposed API has been implemented in MaRTE OS, which is a free software implementation of the POSIX minimal real-time operating system, intended for small embedded systems. It is written in Ada but provides both the POSIX Ada and the C interfaces. Using this implementation we have programmed and tested several scheduling policies, such as a priority-based round robin scheduler, EDF, and more complex dynamic scheduling policies. We have also tested some application-defined synchronization protocols, such as the full Priority Ceiling Protocol. Preliminary performance data show a nice level of efficiency.

The main limitations of our proposed interface from the point of view of an Ada application are:

- We cannot use protected objects for application-defined synchronization protocols. We can only use POSIX mutexes for this purpose.
- The scheduler task cannot be created as a scheduler task from the beginning; it has to be created dynamically. In addition, its scheduled tasks must be created with one of the POSIX fixed-priority scheduling policies, and then switched to become application scheduled tasks.

The reason for these limitations has been our desire to keep the implementation simple and not require changing the run-time system nor the compiler. In the future, it would be interesting to make these changes so that we could use protected objects for application-defined synchronization, and pragmas for setting the scheduler task and the scheduled tasks properties from the beginning.

MaRTE OS, including the application-defined scheduling services defined in this paper can be found at: `http://marte.unican.es`

References

[1] M. Aldea and M. González. “MaRTE OS: An Ada Kernel for Real-Time Embedded Applications”. Proceedings of the International Conference on Reliable Software Technologies, Ada-Europe-2001, Leuven, Belgium, *Lecture Notes in Computer Science, LNCS 2043*, May, 2001.

[2] G.M. Candea and M.B. Jones, “Vassal: Loadable Scheduler Support for Multi-Policy Scheduling”. *Proceedings of the Second USENIX Windows NT Symposium*, Seattle, Washington, August 1998.

[3] IEEE Std 1003.1-2001. *Information Technology -Portable Operating System Interface (POSIX)*. Institute of Electrical and electronic Engineers.

[4] IEEE Std. 1003.13-1998. *Information Technology - Standardized Application Environment Profile- POSIX Realtime Application Support (AEP)*. The Institute of Electrical and Electronics Engineers.

[5] IEEE Std 1003.5b-1996, *Information Technology—POSIX Ada Language Interfaces—Part 1: Binding for System Application Program Interface (API)—Amendment 1: Realtime Extensions*. The Institute of Electrical and Engineering Electronics.

[6] B. Ford and S. Susarla, “CPU Inheritance Scheduling”. *Proceedings of OSD*I, October 1996.

[7] P. Gai, L. Abeni, M. Giorgi, G. Buttazzo, "A New Kernel Approach for Modular Real-Time Systems Development", *IEEE Proceedings of the 13th Euromicro Conference on Real-Time Systems*, Delft, The Netherlands, June 2001.

[8] E.W. Giering and T.P. Baker (1994). The GNU Ada Runtime Library (GNARL): Design and Implementation. *Wadas’94 Proceedings*.

[9] OMG. *Real-Time CORBA 2.0: Dynamic Scheduling,* Joint Final Submission. OMG Document orbos/2001-06-09, June 2001.

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

[11]V. Yodaiken, “An RT-Linux Manifesto”. *Proceedings of the 5th Linux Expo*, Raleigh, North Carolina, USA, May 1999.