Implementing an Application-Defined Scheduling Framework for Ada Tasking

Mario Aldea[1], Javier Miranda[2], and Michael González Harbour[1]
aldeam@unican.es, jmiranda@iuma.ulpgc.es, mgh@unican.es

[1]*Departamento de Electrónica y Computadores, Universidad de Cantabria, 39005-Santander, SPAIN,*

[2]*Applied Microelectronics Research Institute, Univ. Las Palmas de Gran Canaria, 35017 Las Palmas de Gran Canaria, SPAIN*

Abstract: A framework for application-defined scheduling and its corresponding application program interface (API) were defined during the last International Real-Time Ada Workshop, and are being proposed for standardization in the future revision of the Ada language. The framework allows applications to install one or more task schedulers capable of implementing a large variety of scheduling algorithms. This paper describes the implementation of this framework, both at the compiler and the run-time system levels. The objective of this work is to serve as a reference implementation in which the API can be evaluated and tested, and its performance can be assessed. We show that the amount of changes to the compiler is relatively small, and that the application scheduling capability can be supported with a small level of complexity.

Keywords: Real-Time, Kernel, Scheduling, Compilers, Ada 95, POSIX

1 Introduction[1]

Real-time applications have evolved towards more complex systems in which there is a mixture of different kinds of timing requirements. It is common to see that in the same system the traditional hard real-time requirements, related for instance with the control part of the application, are mixed with more flexible requirements, related for instance with multimedia processing which requires the capability of reclaiming unused capacity to be able to fully utilize all the available system resources. A mixture of different quality of service requirements is therefore common, and these mixed requirements can only be met by using very flexible scheduling mechanisms.

The fixed priority scheduling policy currently specified in Ada 95's Real-Time annex is adequate for handling the traditional hard real-time requirements, or even soft real-time requirements when the application resources do not need to be fully utilized. However, it is well known that with dynamic priority mechanisms it is possible to achieve a higher degree of utilization of the system resources. But because the timing requirements in current applications are so diverse, it is not possible to pick a dynamic priority scheduling policy that satisfies all [2]. This makes it difficult to standardize on

1. This work has been funded by the *Comisión Interministerial de Ciencia y Tecnología* of the Spanish Government under grants TIC 2001-1586-C03-03 and TIC 2002-04123-C03 and by the *Commission of the European Communities* under contract IST-2001-34140 (FIRST project).

just one, or even a few, because whichever set is picked, it will not satisfy all application developers.

For this reason, we prefer a more general solution in which application developers could install their own schedulers for Ada tasks. The past International Real-Time Ada Workshop (IRTAW) discussed and approved a framework and associated application program interface (API) for being able to install one or more application-defined schedulers inside the Ada run-time system [8]. This framework is being proposed for the next revision of the Ada language [10].

Before the application-defined scheduling API described in [8] can be standardized, it is necessary to build a reference implementation in which the design can be evaluated and tested. In this paper we present the details of this implementation. The designed API requires compiler, run-time system, and underlying kernel support, so we will describe the implementation at these three levels. In the compiler level we need to implement a few pragmas that enable the application to specify the schedulers, associate tasks to these schedulers, and specify their timing parameters. In the run-time system level we need to implement a new support package and change some internal operations. The implementation will be based on the GNAT compiler and on our application-defined scheduling facilities [6] implemented on top of our MaRTE OS [1], which is an operating system for embedded real-time systems that conforms to the POSIX Minimal Real-Time System Profile [3][4]. The application-defined scheduling facilities available in MaRTE OS are also being proposed for standardization under the POSIX standard [3]. When this scheduling framework is not available, many kernels allow installation of kernel modules that could enable programming the required hooks to implement the functionality. It is also possible to implement the application-defined scheduling services for Ada in a run-time system running on a bare machine. The Real-Time Specification for Java [15] provides a framework in which an implementer, but not the application developer, can plug in a new scheduler.

The paper is organized as follows. Section 2 gives an overview of the application-defined scheduling framework that has been proposed for the next revision of the Ada language. Section 3 describes a high-level view of the architecture of the implementation of the application-defined scheduling framework using GNAT and MaRTE OS. Section 4 gives an overview of the changes to the compiler and run-time system that were needed for that implementation. Section 5 describes the `Ada.Application_Scheduling` support package, as well as the changes that we had to make to the underlying MaRTE OS kernel. Section 6 briefly describes the changes made to the compiler's front-end. Section 7 evaluates the implementation both from the point of view of its complexity and of its performance. Finally, Section 8 gives our conclusions.

2 Overview of the Application-Defined Scheduling Proposal

Fig. 1 shows the proposed approach for application-defined scheduling. Each application scheduler is a special software module that can be implemented inside the run-time system or as a special user task, and that is responsible of scheduling a set of tasks that have been attached to it. According to the way a task is scheduled, we can categorize the tasks as:

- *System-scheduled tasks*: these tasks are scheduled directly by the underlying run-time system and/or the operating system.
- *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 made ready by their application-defined scheduler.

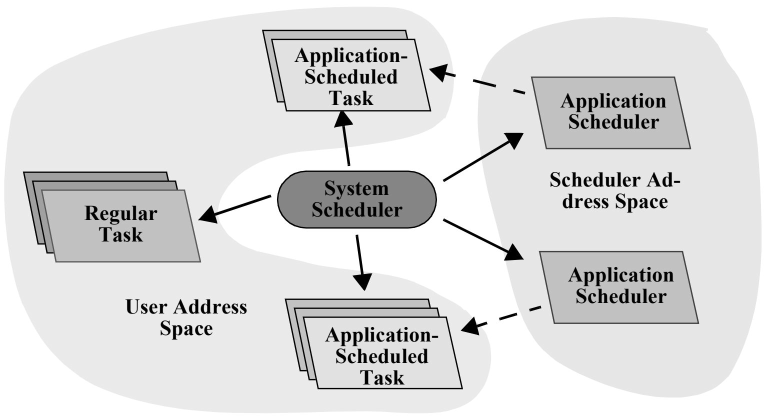

Fig. 1. Model for Application Scheduling

Because the scheduler may execute in an environment different than that of the application tasks, it is an error to share information between the scheduler and the rest of the application. An API is provided for exchanging information when needed. Application schedulers may share information among themselves.

Because the use of protected resources may cause priority inversions or similar delay effects, it is necessary to provide a general mechanism that can bound the delays caused by synchronization when using different kinds of scheduling techniques, and also when several schedulers are being used at the same time. We have chosen the Stack Resource Policy (SRP) [5] because it is applicable to a large variety of policies, including both fixed and deadline-based priority policies. The SRP is also being proposed for Ada 0Y with the name "preemption level locking policy" [13].

The scheduling API presented in [8] is designed to be compatible with the new `Round_Robin` policy described in [7] and proposed for Ada 0Y [14]. In that proposal, compatible scheduling policies are allowed in the system under the `Priority_Specific Task_Dispatching_Policy`, and the desired policy is assigned to each particular priority level, with the `Priority_Policy` pragma; two values are allowed: `Fifo_Within_Priorities`, or `Round_Robin`. At each priority level, only one policy is available, thus avoiding the potentially unpredictable effects of mixing tasks of different policies at the same level.

We propose adding one more value that could be used with the `Priority_Policy` pragma: `Application_Defined`; it represents tasks that are application scheduled. If the scheduler is implemented as a special task, its base priority must be at least equal to that of its scheduled tasks.

Application schedulers have the structure shown in Fig. 2 and are defined by extending the `Scheduler` abstract tagged type defined in the new package `Ada.Application_Scheduling`. This type contains primitive operations that are invoked by the system when a scheduling event occurs. The type is extended by adding the data structures required by the scheduler (for example, a ready queue and a delay queue), and by overriding the primitive operations of interest to perform the scheduling decisions required to implement the desired scheduling policy. As a result of their execution, each of these primitive operations returns an object containing a list of scheduling actions to be executed by the scheduler, such as requests to suspend or resume specific tasks

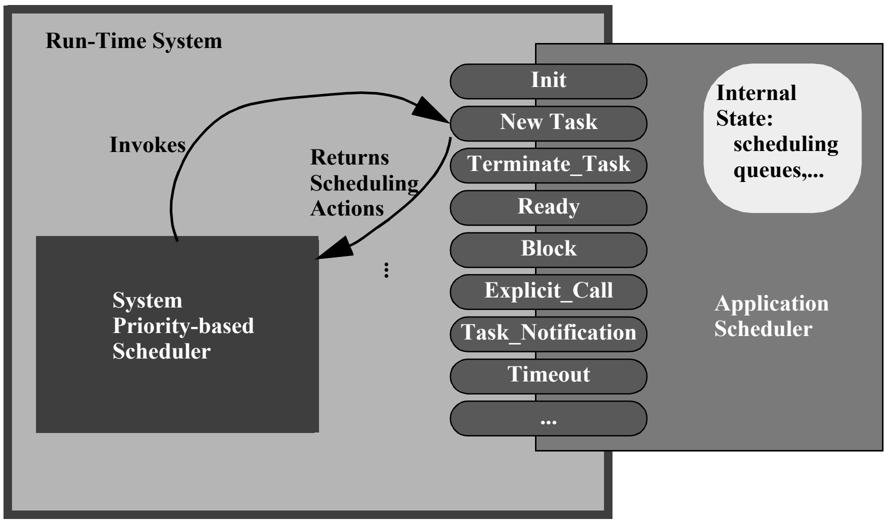

Fig. 2. Structure of an Application Scheduler

When defining an application scheduler we also need to extend the `Scheduling_Parameters` abstract tagged type defined in `Ada.Application_Scheduling`, to contain all the information that is necessary to specify the scheduling parameters of each task (such as its deadline, execution-time budget, period, and similar parameters).

3 Architecture of this Implementation

The implementation is based on the GNAT compiler together with the application-scheduling services described in [6], available in MaRTE OS [1], and extended with some new services that are necessary to support mutual exclusive synchronization using the SRP protocol [5]. Because group budget timers are not yet implemented in

MaRTE OS, we will leave this functionality unimplemented in this version. For a discussion on group budget timers see the Ada Issue generated at the last IRTAW [12].

Fig. 3 shows the high-level architecture of MaRTE OS from the perspective of an Ada application. Although in the future we plan on having implementations of application schedulers that do not require an additional task, the current kernel supports application scheduling by implementing the schedulers inside special user tasks. These tasks execute all the event handling actions at the priority specified by the application. This makes it possible to limit the overhead that a complex application scheduler could cause to critical tasks that might need to execute at a higher priority level. It also serializes the handling of scheduling events, which is a requirement imposed on the operations of the application scheduler to avoid having to program internal data synchronization.

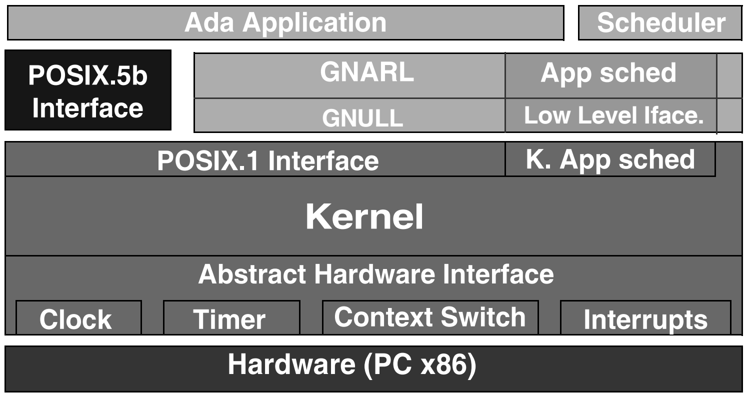

Fig. 3. Architecture of MaRTE OS from the perspective of Ada Applications

The application schedulers defined in [8] are specified as instances of an abstract tagged type. Our implementation uses a user task to implement the application scheduler. The scheduler task has a class-wide access discriminant that is used to pass the scheduler object to it. The task will serve as a driver for the scheduler object, by implementing the following pseudocode. Types `Scheduler` and `Scheduler_Actions` are defined in the `Ada.Application_Scheduling` package defined in [8]:

```
task type App_Sched_Task (Sched : access Scheduler'Class)
is
   Sched_Actions : Scheduling_Actions := Null_Actions;
begin
   Init(Sched);
   loop
      begin
         Execute_Scheduling_Actions (Sched_Actions);
         Wait_For_Sched_Event(Event);
         Sched_Actions:=Null_Actions;
```

```
        exception
           when Sched_Action_Error =>
               Error(Sched,Error_Code);
        end;
        case Event.Kind is
           when Ready =>
              Ready(Sched,Event.Task_Id,Sched_Actions);
           when Block =>
              Block(Sched,Event.Task_Id,Sched_Actions);
           ...
        end case;
     end loop;
  end App_Sched_Task;
```

All the scheduling events handled by the scheduler task are generated by the underlying kernel, except for one: `Abort_Task`. This event is generated by the Ada Run-Time System, which must be modified to notify it to the kernel. A POSIX real-time signal can be used to notify this event, because it can carry the task Id along with the signal.

4 Changes to the Compiler and the Run-Time System

The compiler needs modifications to add the various pragmas on which the application-defined scheduling API is based. The implementation of the changes to the compiler is described in Section 6.

4.1. Configuration Pragmas

To install an application scheduler using the priority-specific framework, we need to specify that the desired priority level, `P`, will be using an application-defined scheduler. We do so with the following configuration pragmas:

```
pragma Task_Dispatching_Policy (Priority_Specific);
pragma Priority_Policy (Application_Defined, P);
pragma Locking_Policy (Preemption_Level_Locking);
```

Therefore, we need to define a new value for the `Task_Dispatching_Policy` pragma. The effect of the pragma is to set the overall policy to the new value, and enable the use of the new pragmas defined to implement the application schedulers.

`Priority_Policy` is a new pragma that is only valid under the `Priority_Specific` dispatching policy. In our implementation there are two values allowed for the first argument of this pragma: `Application_Defined`, and `FIFO_Within_Priorities`; the latter is the default value and has the scheduling behaviour described in the Ada Reference Manual.

The effect of setting the `Priority_Policy` pragma to `Application_Defined` is to store this value associated with the specified priority level. This will later enable installation of the scheduler for that priority level, as described in subclause 4.2.

The `Locking_Policy` pragma is used to define the synchronization policy to be used for systems with application-defined schedulers. If the `Preemption_Level_Locking` value is chosen, this enables use of the `Preemption_Level` pragma for tasks and protected objects, as described in subclause 4.4. Under this locking policy, all protected objects will be implemented using preemption-level mutexes, instead of the regular priority ceiling mutexes.

4.2. Application Scheduler Pragma

We use the following pragma to attach the scheduler to the chosen priority level:

```
pragma Application_Scheduler (My_Scheduler,P,Parameters_Type);
```

where `My_Scheduler` is the scheduler type, specified as an extension of the type `Scheduler`, `P` is the desired priority level, and `Parameters_Type` is the type of the scheduling parameters, derived from the `Scheduling_Parameters` tagged type defined in package `Ada.Application_Scheduling`. This pragma should appear after the definition of the scheduler type, in a library-level unit. We have modified this pragma relative to the one proposed in [8], by adding the scheduling parameters type to enable us a run-time check that verifies the correctness of the specific scheduling parameters supplied for a given task when attached to this application scheduler. A new exception, `Wrong_Parameters_Type`, is added to notify failure of this check.

The effect of this pragma is to install, at elaboration time, a scheduler of the specified type for priority level `P`. A scheduler object is created and the corresponding scheduler task is also created with a pointer to the scheduler object as its discriminant, and with the tag for the specified scheduling parameters type. After the scheduler is created, all tasks that have their base priority equal to the value of `P` are attached to the new scheduler. Any task that is created in the future with that priority level, or that is switched to that priority level, will also become associated with the scheduler, after checking that its scheduling parameters are of the correct type.

4.3. Pragma for the Application-Scheduled Tasks

Application-scheduled tasks may choose to be scheduled by an application-defined scheduler just by setting their priority to the appropriate value. In addition, they must specify their own scheduling parameters, that will be used by the scheduler to schedule that task contending with the other tasks attached to the same application scheduler. We set the scheduling parameters through the following pragma:

```
pragma Application_Defined_Sched_Parameters
    (My_Parameters'Access);
```

where `My_Parameters` is an object of a type that extends the `Scheduling_Parameters` tagged type defined in package `Ada.Application_Scheduling`, and contains specific values for each parameter. These parameters may be changed dynamically with the `Set_Parameters` call. The application scheduling parameters have no effects in tasks with scheduling policies other than `Application_Defined`.

The effect of this pragma is to store a copy of the application-defined scheduling parameters in the task's control block, for future use by the corresponding application-defined scheduler.

4.4. Synchronization Pragma

As described above, a new `Locking_Policy` is defined for the SRP, identified with the `Preemption_Level_Locking` name. This is a configuration pragma that affects the whole partition. Under this locking policy, the priority ceilings of the protected objects continue to exist, and are assigned via the usual `pragma Priority`. In addition, a new pragma may be used to assign a preemption level to each task and protected object:

```
    pragma Preemption_Level (Level);
```

This pragma specifies the preemption level relative to tasks or protected objects of the same priority level.

The effect of this pragma when appearing in a protected object is to set the preemption level of the mutex associated with that protected object. Similarly, if the pragma appears within a task, the effect is to set the preemption level of the underlying thread in the kernel.

5 Support Package and Changes to the Underlying Operating System

Implementation of the `Ada.Application_Scheduling` package defined in [8] is trivial, because this is mostly a set of kernel data structures plus a nearly empty framework for the `Scheduler` tagged type. The main types defined in the package are:

- *Scheduler*. This is an abstract tagged type that represents the skeleton of a scheduler. The `Init` and `Error` primitive operations are abstract. The remaining primitive operations are all null, because if they are not overridden the default behaviour is to do nothing.
- *Event masks*. The type is the one provided by the kernel, and the operations are simple renames or wrappers of the corresponding kernel functions.
- *Scheduling parameters*. The type is abstract tagged, and the operations are wrappers for accessing the internal kernel operations that set and get the application-defined scheduling parameters.
- *Scheduling actions*. All of the scheduling actions are directly supported by the kernel, except for the use of timeouts and execution time timers. Because the single timeout and the execution-time timer actions do not need any special ordering relative to the rest of the actions, the scheduling actions type is designed as a record with three fields. The first one has the type of the kernel's internal scheduling actions. The second one is a single absolute time value (relative to the

clock defined in the `Ada.Real_Time` package). The third field is a fixed-length array of execution-time timer actions.

The behaviour of the execution-time timers is described in the "Execution-Time Timers" Ada Issue developed at the last IRTAW [11]. In [8] we proposed an interface to set a scheduling action corresponding to the expiration of an execution-time timer. The interface had a parameter that was a handler to a protected object that would be signalled by the timer upon expiration. The problem with this approach is that the application scheduler task cannot wait simultaneously for a scheduling event and for the protected object to be signalled. A service task would need to be created, thus increasing complexity and overhead.

In this paper we propose changing the approach, by defining a new interface for setting a execution-time timer expiration scheduling action, with versions for absolute and relative time:

```
procedure Add_Timer_Expiration
  (Sched_Actions : in out Scheduling_Actions;
   T             : in out Timer;
   Abs_Time      : in Ada.Real_Time.Execution_Time.CPU_Time);
procedure Add_Timer_Expiration
  (Sched_Actions : in out Scheduling_Actions;
   T             : in out Timer;
   Interval      : in Ada.Real_Time.Time_Span);
```

Each of these actions is implemented in the scheduler task by arming the kernel timer associated with `T` to expire at the desired time and send a signal to the application scheduler upon its expiration. With the current kernel interface for application-defined scheduling [6] it is possible to simultaneously wait for a scheduling event and a kernel signal, and thus the new interface allows a much more efficient implementation. A similar interface would be required to specify a scheduling action to wait for the expiration of the budget of a group of timers.

A second change that we want to propose for the API specified in [8] is derived from our further experience with implementing application-level schedulers. We found that for some particular scheduling policies it is useful to be able to add a task at the head of the queue corresponding to its priority in the system scheduler, in addition to the normal "ready" scheduling action that adds the task at the tail of its queue. This implies adding a new parameter to the operation that adds the "ready" action, to specify whether the task is added at the back or the front of the queue; it has a default value that causes the task to be added at the back of the queue:

```
type Queue_Place is (Tail, Head);
procedure Add_Ready
  (Sched_Actions : in out Scheduling_Actions;
   Tid           : in Ada.Task_Identification.Task_Id;
   Place         : in Queue_Place:=Tail);
```

To implement the application-defined scheduling framework specified in [8] we had to modify the kernel interface and implementation in MaRTE OS. First of all, we had to add the new SRP synchronization protocol for the mutexes, together with the introduction of preemption level attributes both for mutexes and threads. Associated with this synchronization protocol, we had to add the mask of scheduling events that are deferred while a protected operation is in progress. This mask allows an efficient implementation of the SRP rules within the application-defined scheduler.

In addition, we had to implement a new kind of scheduling action, to support the "timed task notification" scheduling actions. This notification is useful to let the kernel manage the scheduler delay queue, and thus allow integrating this management with the operation of SRP mutexes, by deferring the notification until the end of the protected operations.

6 Compiler support

This section presents the modifications carried out in the front-end to give support to the new pragmas. We will start with the compiler support for the basic pragmas required to install an application scheduler:

- `Pragma Task_Dispatching_Policy`: In case of the `Priority_Specific` policy, the semantic analyser enables the use of the new pragmas defined to implement application schedulers. In addition, it implicitly enables `Fifo_Within_Priorities` as the default dispatching policy to be used in priority levels without application schedulers (this is the default dispatching policy in MaRTE OS).
- `Pragma Priority_Policy`: In case of the `Application_Defined` policy, the semantic analyser checks that the `Priority_Specific` dispatching policy is enabled, evaluates the static expression found in the second argument (the priority level) and enables setting an application scheduler for that priority level.
- `Pragma Application_Scheduler`: The semantic analyser verifies that the `Priority_Specific` policy has been enabled, checks that the current compilation unit is a library-level unit, checks that the first argument is a valid extension of the abstract `Scheduler` type defined for this purpose in package `Ada.Application_Scheduling`, evaluates the static expression found in the second argument (the priority level), and checks that the third argument is a valid extension of the `Scheduling_Parameters` type and gets its tag (of the type `Ada.Tags.Tag)`. The expander uses this scheduler type to generate an object and activate it with the specified priority level and tag. The following code is generated inside the elaboration code of the library unit where the application scheduler type is defined:

```
   ... << default variables for the elaboration code>>
   I28b : aliased <<application scheduler type>>;
begin
   ... << default elaboration code >>
   GNARL.Install_Scheduler
      (I28b'access, <<Priority>>, <<Params'Tag>>);
end library_unit_name;
```

Note that with this approach that there is no need to dynamically create the scheduler.

After the scheduler is created, all tasks that have their base priority equal to the value of the application scheduler must be attached to it. For this purpose, the front-end adds code to the elaboration of all tasks that checks if some application scheduler has been enabled for its priority, and attaches the task to it.

Referring to the pragmas related with the scheduling parameters, the front-end provides the following support:

- `Pragma Application_Defined_Sched_Parameters`: The semantic analyser verifies that the `Priority_Specific` policy has been enabled, checks that the pragma has been found inside a task definition, and checks that the argument is an access to an object of a type that extends the `Scheduling_Parameters` tagged type defined in package `Ada.Application_Scheduling` for this purpose. The expander uses this argument to generate code in the elaboration of the task that calls a run-time subprogram that checks the tag of the argument and passes that argument to the corresponding application scheduler.

Finally, referring to the pragmas related with the preemption level, the front-end does the following work:

- `Pragma Locking_Policy`: In case of `Preemption_Level_Locking`, the semantic analyser checks that the `Priority_Specific` policy is enabled, and enables the use of the `Preemption_Level` pragma for tasks and protected objects. In addition, it implicitly enables the `Ceiling_Locking_Policy` as the default locking policy for tasks and protected objects without the `Preemption_Level_Locking` policy (this is the default locking policy in MaRTE OS).
- `Pragma Preemption_Level`: The actions carried out are: verify that the `Preemption_Level_Locking` policy has been enabled, check that the pragma is placed inside a task definition or a protected type definition, and evaluate the static expression found in the argument (the preemption level). The expander uses this argument to generate code in the elaboration of the task or protected object to set the preemption-level of its mutex.

All the pragmas mentioned above have been implemented as changes to the latest public version (3.15p) of the GNAT compiler, and are available in the MaRTE OS web page [9]. We still need to implement support in the front-end for dynamically changing the priority of a task, and therefore changing the application scheduler. We also want to explore in a future implementation the possibility of assigning the same application scheduler to two or more priority levels, thus making it possible to implement scheduling algorithms in which tasks need two priorities, such as the posix Sporadic Server policy. With the current implementation this is only possible if there are no tasks between the two priorities that are scheduled outside the application scheduler. It is expected that these changes will be available soon.

7 Evaluation

At the compiler level the implementation with GNAT has been quite immediate because:

1. At the semantics level, the current support for the analysis of pragmas is enough to analyse the new pragmas.
2. At the expansion level, the compiler must add some simple code to the elaboration code associated with tasks, protected objects, and library-level packages.
3. At the run-time level, if the POSIX services required for application schedulers are available, the implementation of the new package `Ada.Application_Scheduling` adds no special complexity.

The changes required in the run-time level are also rather small if an application-defined scheduling framework is supported by the underlying kernel, as was the case with our MaRTE OS implementation. When this support is not available, the amount of effort required to implement it is certainly more complex, but our experience with implementing this support in the kernel shows that it is relatively straightforward and does not require fundamental changes to the kernel data structures. We just need to add the necessary actions to notify the scheduling events, and then execute the scheduling actions with the functionality for suspending and resuming tasks already available in the kernel. This is possible in Ada implementations on a bare machine, or based on operating systems that have a modifiable kernel or that support installing kernel modules that could implement the required extensions.

In addition to this qualitative evaluation, we have made some experiments with the implementation to make a quantitative evaluation of its overhead. Table 1 shows some of the results of this evaluation as measured on a 1.1 GHz Pentium III processor.

Table 1. Results of overhead measurements

Metric	Time (μs)
Timed task notification event (from execution of a user task, until execution of the application scheduler)	1.3
Explicit scheduler invocation (from the call to the invoke operation, until the application scheduler executes)	0.9
Execute scheduling actions (one task suspended, one resumed, from scheduler execution until a new user task executes)	2.0
Delay execution (from invocation of delay, until a new user task is executing)	1.6
Rendezvous (from calling task invoking the entry, until it executes again after execution of a null accept)	6.9

The shaded metrics are for the implementation with no application scheduling, and serve as a comparison. We can see that a common context switch time with the current implementation is the sum of the scheduler invocation time (either the first or the second row) plus the execution time of the scheduling algorithm instructions (which varies with the algorithm chosen and is not shown in the table) plus the execution of the scheduling actions (row 3). So for the case of a timed task notification the overhead of the implementation (not counting the scheduling algorithm itself) would be 3.3 μs, just twice the time of a context switch due to a delay instruction, and half of the time needed to do a simple rendezvous. This kind of overhead is more than acceptable for common real-time applications which usually have timing requirements in the range of milliseconds or tens of milliseconds.

In the near future we plan to reduce this already small overhead by eliminating the need to have the application scheduler inside a user task. The proposed API allows implementing the scheduler operations as operations executed directly by the regular application tasks or by kernel.

8 Conclusion

The application-defined scheduling framework defined in [8] represents an opportunity for the Ada language to continue to be the reference language for real-time systems, by supporting the new application requirements for more flexible and resource-efficient scheduling. Before such a framework can be standardized it is necessary to build a reference implementation that can serve to test and evaluate the API and provide a guide for future implementations.

The implementation presented in this paper has shown that the API, with some minor changes described in the paper, is viable and provides an efficient and flexible way of installing and using application-defined schedulers for the Ada tasks. The integrated operation with preemption-level-based synchronization allows a large variety of scheduling policies to be implemented, and gives the application developers the degree of flexibility that is needed to support the requirements of today's applications with their evolving complexity.

The implementation has required a small amount of changes to the compiler and the run-time system. These changes are well localized and do not represent major changes to the existing parts of the implementation. If the underlying kernel does not support application-defined scheduling, our experience with its implementation in MaRTE shows that it is possible to make the implementation with only a moderate effort, without affecting the main kernel data structures and operations.

Performance measurements show moderate overheads that are perfectly acceptable for the majority of real-time requirements that can be found in common applications. In future implementations we plan to eliminate the need for the application scheduler to be implemented in a special user task. This will allow us to lower the overheads even further.

As a final conclusion, the results of the implementation presented in this paper allow us to recommend adoption of the application-defined scheduling services in the next revision of the Ada Language.

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] ARTIST. *Roadmap on Advanced Real-Time Systems for Quality of Service Management.* `http://artist-embedded.org/Roadmaps/A3-roadmap.pdf`

[3] IEEE Std 1003.1-2003. *Information Technology -Portable Operating System Interface (POSIX)*. Institute of Electrical and electronic Engineers.

[4] IEEE Std. 1003.13-2003. *Information Technology -Standardized Application Environment Profile- POSIX Realtime and Embedded Application Support (AEP)*. The Institute of Electrical and Electronics Engineers.

[5] Baker T.P., “Stack-Based Scheduling of Realtime Processes”, Journal of Real-Time Systems, Volume 3, Issue 1 (March 1991), pp. 67–99.

[6] Mario Aldea Rivas and Michael González Harbour. “POSIX-Compatible Application-Defined Scheduling in MaRTE OS” Proceedings of 14th Euromicro Conference on Real-Time Systems, Vienna, Austria, IEEE Computer Society Press, pp. 67-75, June 2002.

[7] A. Burns, M. González Harbour and A.J. Wellings. “A Round Robin Scheduling Policy for Ada”. Proceedings of the International Conference on Reliable Software Technologies, Ada-Europe-2003, Toulouse, France, in Lecture Notes in Computer Science, LNCS 2655, June, 2003, ISBN 3-540-40376-0.

[8] Mario Aldea Rivas and Michael González Harbour. “Application-Defined Scheduling in Ada”. Proceedings of the International Real-Time Ada Workshop (IRTAW-2003), Viana do Castelo, Portugal, September 2003.

[9] MaRTE OS home page: `http://marte.unican.es/`

[10] Pascal Leroy. “An Invitation to Ada 2005”. International Conference on Reliable Software TEchnologies, Toulouse, France, in Lecture Notes on Computer Science, LNCS 2655, Springer, June 2003.

[11] Ada Issue AI95-00307-01/05 “Execution-Time Clocks”. `http://www.ada-auth.org/~acats/AI-SUMMARY.HTML`

[12] Ada Issue AI95-00354-01/01 “Group Execution-Time Timers”. `http://www.ada-auth.org/~acats/AI-SUMMARY.HTML`

[13] Ada Issue AI95-00356-01/01 “Support for Preemption Level Locking Policy”. `http://www.ada-auth.org/~acats/AI-SUMMARY.HTML`

[14] Ada Issue AI95-00355-01/01 “Priority Specific Dispatching including Round Robin”. `http://www.ada-auth.org/~acats/AI-SUMMARY.HTML`

[15] The Real-Time specification for Java. `http://www.rtj.org/rtsj-V1.0.pdf`