Integrating Application-Defined Scheduling with the New Dispatching Policies for Ada Tasks

Mario Aldea Rivas[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: In previous papers we had presented an application program interface (API) that enabled applications to use application-defined scheduling algorithms for Ada tasks in a way compatible with the scheduling model defined in the real-Time Annex of the language. Each application scheduler was implemented with a special task. This paper presents a new implementation in which the application scheduler actions are executed as part of the kernel on which the run-time system is based, thus increasing the efficiency. This paper also presents modifications to the proposed API that align it with the evolution of the Ada Issues being considered in the Ada 200Y standardization. First, we use the new concept of deadline as an abstract notion of urgency, to order the tasks in the scheduling queue of the underlying kernel, freeing the application scheduler of the responsibility of keeping the desired ordering of tasks, and thus simplifying it and reducing its overhead. In second place, we also consider task synchronization through protected objects using the new Stack Resource Policy proposed for the EDF task dispatching policy in Ada 200Y, which can be used in a large variety of fixed and dynamic priority scheduling policies without explicit intervention of the application scheduler.

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

1 Introduction[1]

Although the Ada standard allows implementations to add their own task dispatching policies, most commercial run-time systems just offer the standard fixed-priority scheduling as the mechanism to support real-time concurrency. Scheduling theory and practice shows that fixed priorities can be used to build predictable and analyzable real-time applications, but it is well known that the use of dynamic priority scheduling algorithms in which priorities may vary depending on the passage of time or on other system parameters, allows a better usage of the available processing resources [16]. Current scheduling theory provides enough results to allow schedulability analysis in these systems.

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

As a consequence, it would be desirable to have dynamic priority scheduling policies[1] available in the Ada language. A major evolution in this direction is an Ada Issue being considered for the next revision of the Ada language [10], proposing an Earliest Deadline First (EDF) task dispatching policy [13]. However, this policy by itself is not enough for real-time applications. It is important, for instance, to have a server policy that can handle aperiodic activities in a predictable way; for example, the constant bandwidth server (CBS) [1] is an example of such server. In fact the number of task dispatching policies described in the literature is quite large because dynamic priority approaches are very flexible, and they can be tailored to adapt to many kinds of application requirements. It is impractical for Ada run-time system developers to provide all of these policies in their implementations.

In the past years we have been working on application program interfaces (APIs) and implementations of application-defined scheduling services that could be used for dispatching Ada tasks [8][9]. Those APIs allow the application to install one or more task schedulers, and assigning application tasks to these schedulers. The schedulers would receive the relevant scheduling events to make the appropriate scheduling decisions, and then suspend or resume their scheduled tasks. Other authors [17][18][19] have also worked in other application-defined scheduling services, but they did not provide a full solution covering both scheduling and the associated synchronization protocols.

A previous approach [9] was flexible enough to support many kinds of application defined scheduling policies, but it had several drawbacks that potentially limit its efficiency:

- Because the application scheduler is a special thread, every scheduling decision requires a double context switch, which in some OS architectures may be too expensive.
- The application scheduler has to keep the ordering of the threads that are ready for execution, instead of leaving this task to the underlying kernel where it can be made in a more efficient way.
- Mutual exclusive synchronization requires special support.

Although the overheads measured in our MaRTE OS [2] implementation were acceptable for common applications, we realized that they could be too high in other OS architectures, and thus we worked towards eliminating these sources of inefficiency. As a result we have created a fully new implementation, with similar capabilities, but which is much more efficient than the previous approach. This paper presents the details of this implementation model.

In addition, this paper also presents a new API that aligns the application-defined scheduling proposal with the evolution of the Ada Issues being considered in the Ada

1. Although Ada offers “dynamic priority” facilities that allow the application to explicitly change the priority of a task, the term “dynamic priority policies” is usually applied to those policies in which the priority may change without explicit intervention of the application.

200Y standardization [10]. First, we use the new concept of deadline defined in [13] as an indication of urgency that is used by the underlying kernel to order the tasks in the scheduling queue. By mapping application-defined scheduling parameters into this urgency concept in a way transparent to the remainder of the application, we can free the application scheduler of the responsibility of keeping the desired ordering of tasks, and thus we can simplify it and reduce its overhead. In second place, in this paper we also consider task synchronization through protected objects using the new Stack Resource Policy proposed for the EDF task dispatching policy [13] in Ada 200Y, which can be used in a large variety of fixed and dynamic priority scheduling policies without explicit intervention of the application scheduler.

This paper is organized as follows: Section 2 gives an overview of the application-defined scheduling proposal. Section 3 discusses the use of deadlines as an abstract notion of urgency to order the tasks in the ready queue. Section 4 provides an overview of the proposed API. Section 5 discusses the implementation details: inside the kernel, the run-time system, and the compiler. Section 6 shows an example of a non-trivial application-defined scheduler. Section 7 contains an evaluation of the new implementation with performance metrics that compare it to previous implementations. Finally, we give our conclusions in Section 8.

2 Overview of the New Application-Defined Scheduling Proposal

Fig. 1 shows the proposed approach for application-defined scheduling. Each application scheduler is a special software module 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 run-time system and/or the operating system, without intervention of an application scheduler.
- *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.

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 them.

The new scheduling API is designed to be compatible with the new task dispatching policies under the framework initially described in [6] that has evolved during the Ada 200Y standardization process [14]. In that proposal, compatible scheduling policies are allowed in the system by specifying the desired policy for each particular priority range, with the `Priority_Specific_Dispatching` pragma; three values are allowed: `Fifo_Within_Priorities`, `Round_Robin_Within_Priorities`, and `EDF_Across_Priorities` [13]. At each priority level or priority band, only one policy is available, thus avoiding the potentially unpredictable effects of mixing tasks of different policies at the same level.

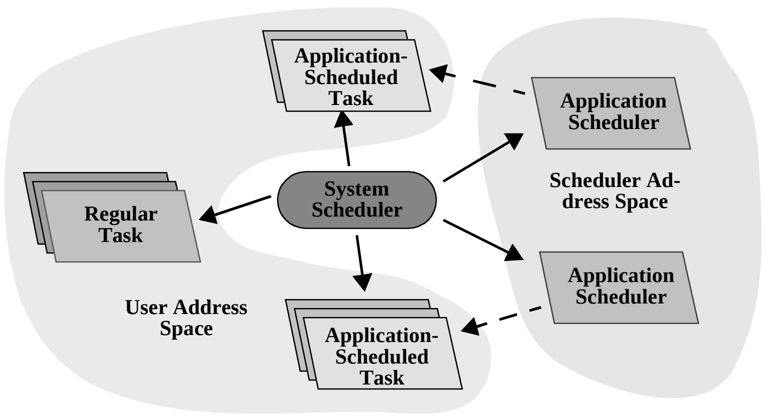

Fig. 1. Model for Application Scheduling

We propose adding one more value that could be used with the `Priority_Specific_Dispatching` pragma: `Application_Defined`; it represents tasks that are application scheduled, in a particular priority band.

The following language-defined library package serves as the parent of other language-defined library units concerned with dispatching, including the proposed application scheduling unit [14]:

```
package Ada.Dispatching is
   pragma Pure (Dispatching);
   Dispatching_Policy_Error : exception;
end Ada.Dispatching;
```

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.Dispatching.Application_Scheduling` [8]. This type contains primitive operations that are invoked by the system when a scheduling event occurs. To create an application-defined scheduler the type can be extended by adding the data structures required (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. Each of these primitive operations returns as a result an object containing a list of scheduling actions to be executed by the system, such as the requests to suspend, resume, or change the urgency of specific tasks

When defining an application scheduler we also need to extend the `Scheduling_Parameters` tagged type defined in `Ada.Dispatching.-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).

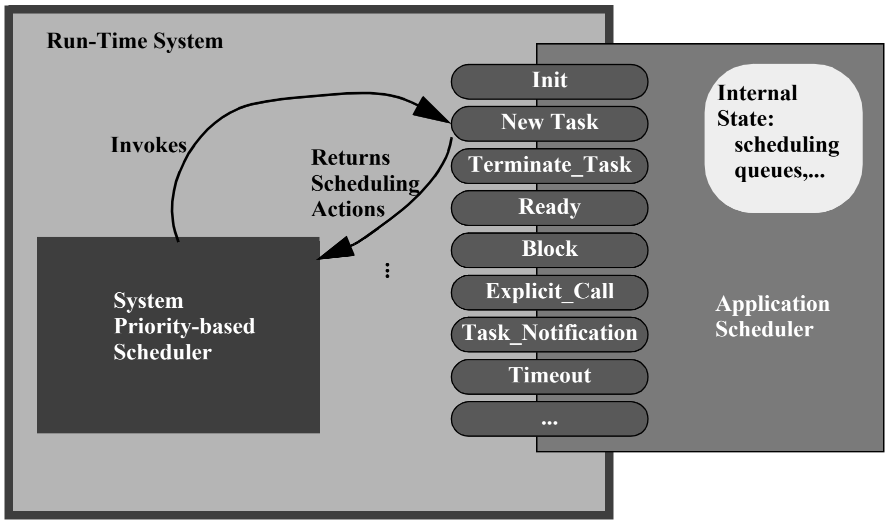

Fig. 2. Structure of an Application Scheduler

3 Abstract Ordering of the Ready Queue

In the framework proposed in [9] the application schedulers were responsible for ordering the tasks that were ready storing them in a specific queue inside the application scheduler, and required an extra overhead of invoking the scheduler for every scheduling decision. The run-time system's ready queue was not usually practical because for a given priority level it was just a FIFO queue, and dynamic priority scheduling policies usually require other orderings.

In this paper we propose taking advantage of the `EDF_Across_Priorities` policy defined in [13], using the deadline to order the tasks in the ready queue and interpreting it as an abstract notion of "urgency" on to which any particular scheduling parameter that the application scheduler chooses (e.g., deadline, value, quality of service, ...) can be mapped. This approach is especially suitable for scheduling policies in which the urgency or priority of the task only changes from one job to the next, but remains constant within a specific job. For example, for EDF scheduling the absolute deadline of the task does not change for a specific job, because it is defined as the release time plus the relative deadline. In any case, the approach continues to be valid for policies that do change the priority in the middle of a job, such as in the proportional time-sharing scheduler shown in the example of Section 6.

The scheduling deadline assigned to each job makes it easier to implement scheduling algorithms than in our previous application-defined scheduling proposal and, more important, the schedulers become more efficient because when a task finishes its current job it is not necessary to invoke the application scheduler again to determine the next task to execute. The system can choose the new task by itself. In this context only when a new job arrives, or when the relative priority or deadline of a job changes, it would be necessary to invoke the application scheduler.

The way chosen to inform the system about the “deadline” of a task is by adding a new parameter to the operation that adds the “ready” action:

```
procedure Add_Ready
  (Sched_Actions : in out Scheduling_Actions;
   Tid           : in Ada.Task_Identification.Task_Id;
   Urg           : in Ada.Dispatching.EDF.Deadline);
```

If this action is performed on a task that is already active, it changes the deadline of the task forcing the reordering of the ready queue.

As in the proposal for supporting deadlines and EDF scheduling in Ada 200Y [13], in our framework we have also chosen the Stack Resource Policy (SRP) [5] as the synchronization protocol for protected objects. We can use the same rules described in [13] to implement the SRP in the context of application-defined scheduling, by using the priorities in the EDF priority band as the actual preemption levels of the SRP. The priority ceilings in this band will have to be assigned according to the rules of the SRP, if we want to eliminate priority inversion effects.

4 Overview of the API

The package `Ada.Dispatching.Application_Scheduling` contains the proposed interface for the application scheduling operations.

In first place, there is an set of operations for the application scheduler to ask the system to execute scheduling decisions through what we call the scheduling “actions”. We add these actions to an object of the type `Scheduling_Actions`. One of these operations is `Add_Ready`, mentioned above. The scheduling action operations are shown next:

```
type Scheduling_Actions is private;
procedure Add_Accept
  (Sched_Actions : in out Scheduling_Actions;
   Tid           : in     Ada.Task_Identification.Task_Id);
procedure Add_Reject
  (Sched_Actions : in out Scheduling_Actions;
   Tid           : in     Ada.Task_Identification.Task_Id);
procedure Add_Ready
  (Sched_Actions : in out Scheduling_Actions;
   Tid           : in     Ada.Task_Identification.Task_Id);
procedure Add_Ready
  (Sched_Actions : in out Scheduling_Actions;
   Tid           : in     Ada.Task_Identification.Task_Id;
   Urg           : in     Ada.Dispatching.EDF.Deadline);
procedure Add_Timed_Task_Activation
  (Sched_Actions : in out Scheduling_Actions;
   Tid           : in     Ada.Task_Identification.Task_Id;
   Urg           : in     Ada.Dispatching.EDF.Deadline;
   At_Time       : in     Ada.Real_Time.Time);
```

```
procedure Add_Suspend
  (Sched_Actions : in out Scheduling_Actions;
   Tid           : in     Ada.Task_Identification.Task_Id);
procedure Add_Timeout
  (Sched_Actions : in out Scheduling_Actions;
   At_Time       : in     Ada.Real_Time.Time);
procedure Add_Timed_Task_Notification
  (Sched_Actions : in out Scheduling_Actions;
   Tid           : in     Ada.Task_Identification.Task_Id;
   At_Time       : in     Ada.Real_Time.Time);
procedure Add_Timer_Expiration
  (Sched_Actions : in out Scheduling_Actions;
   T             : in out Ada.Execution_Time.Timers.Timer;
   Abs_Time      : in     Ada.Execution_Time.CPU_Time);
procedure Add_Timer_Expiration
  (Sched_Actions : in out Scheduling_Actions;
   T             : in out Ada.Execution_Time.Timers.Timer;
   Interval      : in     Ada.Real_Time.Time_Span);
```

The application-defined scheduling package provides operations for the application to directly invoke the scheduler. For instance, the usual way in which an application task informs the scheduler that it has finished its current job is by calling these operations. Information can be passed from the application task to its scheduler, and viceversa, by extending the abstract types `Message_To_Scheduler` and `Reply_From_-Scheduler`, as needed. The explicit scheduler invocation types and operations are:

```
type Message_To_Scheduler is abstract tagged null record;
type Reply_From_Scheduler is abstract tagged null record;
procedure Invoke
  (Msg   : access Message_To_Scheduler'Class);
procedure Invoke
  (Msg   : access Message_To_Scheduler'Class;
   Reply : access Reply_From_Scheduler'Class);
```

The scheduler is declared as an abstract tagged type with primitive operations that represent the code to be executed when a specific scheduling event occurs:

```
type Scheduler is abstract tagged null record;
```

The actions to be executed by the system, described above, are passed as a parameter to each of the primitive operations. The specifications of the most important of these operations are:

```
type Error_Cause is
 (SRP_Rule_Violation, Invalid_Action_For_Task);

procedure Init
  (Sched   : out    Scheduler) is abstract;
procedure New_Task
  (Sched   : in out Scheduler;
   Tid     : in     Ada.Task_Identification.Task_Id;
   Actions : in out Scheduling_Actions);
procedure Terminate_Task
  (Sched   : in out Scheduler;
   Tid     : in     Ada.Task_Identification.Task_Id;
   Actions : in out Scheduling_Actions);
```

```
procedure Ready
  (Sched   : in out Scheduler;
   Tid     : in     Ada.Task_Identification.Task_Id;
   Actions : in out Scheduling_Actions);
procedure Block
  (Sched   : in out Scheduler;
   Tid     : in     Ada.Task_Identification.Task_Id;
   Actions : in out Scheduling_Actions);
procedure Yield
  (Sched   : in out Scheduler;
   Tid     : in     Ada.Task_Identification.Task_Id;
   Actions : in out Scheduling_Actions);
procedure Abort_Task
  (Sched   : in out Scheduler;
   Tid     : in     Ada.Task_Identification.Task_Id;
   Actions : in out Scheduling_Actions);
procedure Change_Sched_Param
  (Sched   : in out Scheduler;
   Tid     : in     Ada.Task_Identification.Task_Id;
   Actions : in out Scheduling_Actions);
procedure Explicit_Call
  (Sched   : in out Scheduler;
   Tid     : in     Ada.Task_Identification.Task_Id;
   Msg     : access Message_To_Scheduler'Class;
   Reply   : access Reply_From_Scheduler'Class;
   Actions : in out Scheduling_Actions);
procedure Task_Notification
  (Sched   : in out Scheduler;
   Tid     : in     Ada.Task_Identification.Task_Id;
   Actions : in out Scheduling_Actions);
procedure Timeout
  (Sched   : in out Scheduler;
   Actions : in out Scheduling_Actions);
procedure Execution_Timer_Expiration
  (Sched         : in out Scheduler;
   Expired_Timer : in out Ada.Execution_Time.Timers.Timer;
   Actions       : in out Scheduling_Actions);
procedure Error
  (Sched : in out Scheduler;
   Tid : in Ada.Task_Identification.Task_Id;
   Cause : in  Error_Cause;
   Actions : in out Scheduling_Actions) is abstract;
--  Non-abstract operations have a null body
```

The final part of the API contains operations to handle event masks, and an operation to set the mask of events that should be filtered out. The `Init` primitive operation of the example shown in Section 6 illustrates the usage of these operations to create a set of events and set the event mask. The set mask operation is:

```
procedure Set_Event_Mask (Mask : in Event_Mask);
```

For an in-depth description of the API shown in this section please refer to [8].

5 Implementation Details

5.1. Changes to the Compiler and the Run-Time System

The prototype implementation of this new proposal required the implementation of the new priority dispatching mechanism proposed for Ada 200Y [14]. For this purpose we modified the GNAT GAP 1.0 compiler to give support to the new configuration pragma `Priority_Specific_Dispatching`. This pragma allows one or more priority levels to be scheduled according to a given priority policy. In addition, we modified the semantic analyser to allow the programmers the use of the `Application_Defined` policy described in this paper. Hence, the form of this pragma as well as the supported values for the policy identifier in our prototype implementation is as follows:

```
pragma Priority_Specific_Dispatching
  (Policy_Identifier,
       -- Valid policy names:
       --    FIFO_Within_Priorities        (Ada 1995)
       --    Round_Robin_Within_Priorities (Ada 200Y)
       --    EDF_Across_Priorities         (Ada 200Y)
       --    Application_Defined           (new!)
   First_Priority_Expression, Last_Priority_Expression);
       -- Static expressions of type System.Any_Priority
```

The implementation of application-defined schedulers requires some additional mechanism that allows the programmer to register its scheduler with the run-time system. For this purpose we also give support to the following new pragma:

```
pragma Application_Scheduler
  (Application_Defined_Scheduler,
       -- Tagged type derived from
       -- Ada.Dispatching.Application_Scheduling.Scheduler
   First_Priority_Expression, Last_Priority_Expression);
       -- Static expressions of type System.Any_Priority
```

The first argument is the name of a type derived from the tagged type defined in package `Ada.Dispatching.Application_Scheduling`. In addition to the static check required to verify that all the priority levels specified in the range of priorities are allowed to have an application-defined scheduler (by means of pragma `Priority_Specific_Dispatching`), the compiler also checks that the first argument corresponds with a tagged-type derived from the correct type and that has been defined at the library level. This is required to ensure that the scheduler is available during the whole execution of the program. The implementation of pragma `Application_Defined_Sched_Parameters` is the same as was described in [9].

The information provided by these pragmas is also used by the compiler to generate additional code. For example, the arguments given in the pragma `Application_Scheduler` are used to modify the elaboration-code of the library-level package containing the definition of the application-defined scheduler: the corresponding call to the underlying operating system to create an application scheduler is performed and this scheduler is registered into the run-time for the specified range of priorities.

Concerning the run-time system, the code associated with the elaboration of the tasks has been modified. When the priority of a task is inside the range of priorities associated with an application-defined scheduler, the task is registered with it. In addition, the run-time must also take care of the re-allocation of a task to another scheduler when the priority of the task changes.

5.2. Changes to the Underlying Operating System

The implementation of the ordering of tasks by urgency in the ready queue required changing its enqueue operation. In the fixed-priority implementation, a new task was added at the tail of the queue for its priority. Now, the position depends on the urgency and the preemption level. This change is isolated to just one operation in the kernel.

In our previous implementation of application-level scheduling [9], all the scheduling operations were executed by a special task. Now, some of the scheduler operations are executed by the regular application tasks, namely `Explicit_Call`, `New_Task`, `Block`, `Yield`, and `Change_Sched_Param`. This simplification causes these operations to execute much faster because we can save the double context switch required to execute in the context of the special task, and return from it. Implementing this change has been very simple, because to accomplish the required mutual exclusion with the scheduler we just need to raise the urgency and preemption level of the task executing these operations to the same levels of that scheduler. This change of scheduling parameters does not imply reordering the queue, because the task was already executing; therefore, it is very fast.

In the new implementation, the scheduling operations triggered by asynchronous events and which cannot be executed in the context of the running task are handled by an auxiliary task created for each application scheduler. This is a kernel task that has a very simple, much faster context switch, compared to the regular Ada tasks, because it crosses less software layers. It has urgency and preemption level values higher than those of their associated application-scheduled tasks, to take precedence in execution. This auxiliary task can be eliminated completely once the "Timing events" [15] or a similar OS functionality are available. In the current implementation we don't yet have timing events available, and the only alternative to the auxiliary tasks would be the use of signal handlers, which would have more overhead than the auxiliary task.

6 Example: Proportional Share Scheduler

The following example shows the pseudocode of and application scheduler that implements a simple proportional share scheduling policy, which is useful in the context of soft real-time systems in which all tasks have to make progress but some tasks are more important than others. The policy can be easily adapted so that hard real-time tasks can coexist in the system at higher priority levels. We will make the following declaration to reduce the length of the code in this section:

```
package App_Sched renames
   Ada.Dispatching.Application_Scheduling;
```

Under the proportional share policy, each application-scheduled task has two specific scheduling parameters called “importance” and “period” that will be notified to the scheduler using the following types:

```
type Task_Importance is new Positive range 1..100;

type Share_Parameters is new
   App_Sched.Scheduling_Parameters with record
   Importance : Task_Importance;
   Period     : Ada.Real_Time.Time_Span;
end record;
```

The application scheduler will ensure that each task has a percentage of the CPU time allocated to it, proportionally to its importance according with the following expression:

$$W_t = \frac{I_t}{\sum_{\tau \in \{T\}} I_\tau}$$

That is, the percentage or CPU time (W_t) granted to a task t is equal to its importance (I_t) divided by the sum of the importances of all the tasks scheduled by the application scheduler. That means each task will be allowed to execute at most $W_t \cdot P_t$ units of time in each period (P_t). In order to enforce that constraint, the application scheduler will make use of the execution-time timers. As soon as a task reaches its allowed execution time for its current period the application scheduler lowers its “urgency” to a background level, so that it can only execute in the case that there are no other tasks with remaining execution time. When a new period for a task starts, the application scheduler raises the “urgency” of the task to a value proportional to its importance.

The application scheduler is defined as an extension of the abstract `Scheduler`.

```
type Share_Scheduler is
  new App_Sched.Scheduler with private;
```

And then it is defined in the private part of the package to contain the total importance and a list of items with the information associated with each task:

```
type Share_Task_Data is tagged record
   Importance : Task_Importance;
   Period     : Ada.Real_Time.Time_Span;
   Next_Period_Start : Ada.Real_Time.Time;
   Budget     : Ada.Real_Time.Time_Span;
   Task_Id    : aliased Ada.Task_Identification.Task_Id;
   Timer      : access Ada.Execution_Time.Timers.Timer;
end record;

package Task_Data_Lists is new
   Singly_Linked_Lists (Share_Task_Data);

subtype Task_Data_Ac is Task_Data_Lists.Element_Ac;
```

```
type Share_Scheduler is new App_Sched.Scheduler with
record
   List             : Task_Data_Lists.List;
   Total_Importance : Natural;
end record;
```

In the body of the proportional share scheduler a new task attribute is defined as an instance of `Ada.Task_Attributes`, so that each task can access its own scheduler information, by accessing the list element represented by a value of the type `Task_Data_Ac` .

```
package Share_Data is new
   Ada.Task_Attributes (Task_Data_Ac, null);
```

The primitive operations of `Share_Scheduler` perform the following actions, illustrated with the relevant calls to the application scheduling API:

- `Init`: initializes the list of tasks and sets the event mask (with `App_Sched.Set_Event_Mask`) to filter out all the events except *New_Task*, *Execution_Timer_Expiration*, and *Task_Notification.*

  ```
   App_Sched.Fill (Mask);
   App_Sched.Delete(App_Sched.New_Task, Mask);
  App_Sched.Delete(App_Sched.Execution_Timer_Expiration,Mask);
   App_Sched.Delete(App_Sched.Task_Notification, Mask);
   App_Sched.Set_Event_Mask (Mask);
  ```

- `New_Task`:
  - If there are no erroneous parameters accept the new task by adding an *Accept* action

    ```
    App_Sched.Add_Accept (Actions, Tid);
    ```

  - Create an execution-time timer

    ```
    T.Timer := new
       Ada.Execution_Time.Timers.Timer(T.Task_Id’Access);
    ```

  - Create a new item in the list of tasks, with information on the period, the importance, and the execution-time timer, and set the task’s attribute to point to that value

    ```
    Share_Data.Set_Value (T, Tid);
    ```

  - Update the total importance and recalculate the budgets of all tasks in the list
  - Activate the new task by adding a *Ready* event with an urgency equal to the importance plus the last value of importance (obtained with a trivial conversion function called `Urg_Foreground`)

    ```
      App_Sched.Add_Ready
         (Actions, Tid, Urg_Foreground(T.Importance));
    ```

  - Program a timed task notification, adding a *Timed_Task_Notification* action, to occur at the next period

    ```
      App_Sched.Add_Timed_Task_Notification
         (Actions, Tid, T.Next_Period_Start);
    ```

  - Program the execution-time timer

    ```
    App_Sched.Add_Timer_Expiration
       (Actions, T.Timer.all, T.Budget);
    ```

- `Execution_Timer_Expiration`: This function is called by the system when the task's execution-time budget gets exhausted for the current period. It has to lower the urgency to a value equal to the importance (obtained with the trivial conversion function `Urg_Background`)

  ```
  App_Sched.Add_Ready
     (Actions, T.Task_Id, Urg_Background(T.Importance));
  ```

- `Task_Notification`:

  - Obtain the task data using the corresponding task attribute

    ```
    T := Share_Data.Value (Tid);
    ```

  - Raise the task's urgency

    ```
    App_Sched.Add_Ready
      (Actions, Tid, Urg_Foreground (T.Importance));
    ```

  - Calculate the new ready time by adding the period to the previous one and program a task notification to occur at start of the next period

    ```
    App_Sched.Add_Timed_Task_Notification
      (Actions, Tid, T.Next_Period_Start);
    ```

  - Reprogram the execution-time timer to check the budget for the next period

    ```
    App_Sched.Add_Timer_Expiration
       (Actions, T.Timer.all, T.Budget);
    ```

The configuration pragmas in the system will be defined as follows:

```
pragma Priority_Specific_Dispatching
   (Application_Defined, 14, 18); -- for instance
pragma Locking_Policy (Ceiling_Locking);
```

And the pragma to define the application scheduler would be:

```
pragma Application_Scheduler
   (Share_Scheduling.Share_Scheduler,14,18);
```

The specification of an application-scheduler task type would be:

```
task type Share_Task_1
  (Param : access Share_Scheduling.Share_Parameters;
   Prio  : System.Any_Priority)
is
   pragma Priority (Prio);
   pragma Application_Defined_Sched_Parameters (Param);
end Share_Task_1;
```

And the body would just be the desired computation with no explicit relation with the scheduler:

```
task body Share_Task is
begin
   loop
      --  do useful work
   end loop;
end Share_Task;
```

To create a scheduler type task we would declare the importance and period inside an object of the `Share_Parameters` type, and then we would declare the task as:

```
 T1_Paramters : aliased Share_Scheduling.Share_Parameters :=
   (Importance => 60,
    Period     => Ada.Real_Time.To_Time_Span (4.0));
T1 : Share_Task (T1_Paramters'Access, Scheduler_Priority_Low);
```

7 Evaluation

We have made some experiments to compare the overheads of the new implementation of application-defined scheduling presented in this paper with the previous one [9]. Table 1 shows some of the average results of this evaluation as measured on a 1.1 GHz Pentium III processor.

Table 1. Comparison of overhead measurements

Metric	Time (μs) old approach	Time (μs) new approach
Timed task notification event (from the execution of a user task, until the execution of the application scheduler)	1.3	1.0
Explicit scheduler invocation (from the call to the invoke operation, until the application scheduler executes)	0.9	0.23
Execute scheduling actions (switch from a running user task to a new one)	2.0	1.3
Context switch for an EDF application-defined scheduler	3.3	1.0

We can see that the timed task notification (first row in Table 1) is now smaller mainly because of the use of a kernel task, instead of a regular Ada task. The explicit scheduler invocation (second row) is much smaller because it is now executed in the context of the application task, avoiding a double context switch. Execution of scheduling actions (third row) is also faster because in the new implementation, for the application scheduler to request a context switch it is no longer necessary to suspend a task and make the new one ready, but just to lower the urgency of the task that must leave the CPU.

We have measured a common context switch time for a specific policy with the old and the current implementation. The policy chosen was EDF, and the results appear in the fourth row 4 of Table 1. The scenario measured is a typical EDF context switch in which the most urgent task finishes its current job and explicitly invokes its scheduler to notify this situation and wait for the following activation. Once that task leaves the CPU the new most urgent task starts executing. We can see that the overhead drops to around one third of that of the old implementation, from 3.3 μs to just 1.0 μs. The reason is that now the explicit invocation is done as part of the calling task, and the new task is chosen by the operating system based on its urgency, and therefore without intervention of the application scheduler. The new context switch is under the time of a context switch due to a regular delay statement (1.6 μs for the same processor). 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.

8 Conclusion

In this paper we have presented the implementation of the application-defined scheduling framework defined in [8], augmented with the inclusion of an abstract notion of urgency defined to enhance efficiency and simplify the development of application schedulers. To implement this notion of urgency we take advantage of the new EDF dispatching policy and associated SRP synchronization protocol proposed for Ada 200Y.

The compiler and run-time system modifications presented in [9] have been adapted to the new version of the GNAT compiler (GAP 1.0) in a straightforward manner. Also the kernel support for urgency ordering of ready queues has been implemented with only a moderate effort in our operating system MaRTE OS (a standard priority-based operating system).

To prove the flexibility and usage simplicity of our framework an example of a non-trivial application scheduler has been presented. Performance measurements show more than acceptable overheads, moreover if the important advantages of having a framework as the proposed are taken into account.

We have also shown the usefulness of the EDF task dispatching policy proposed for Ada 200Y. Although this policy by itself may not be enough for all applications, thus motivating the presence of application defined schedulers, our proposal builds on top of it and takes advantage of the notion of deadline as well as the SRP implementation for protected objects using just the priority ceilings as preemption level control values.

We believe that an application-defined scheduling framework for Ada like the one presented in this paper represents an opportunity for this 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.

References

[1] L. Abeni and G. Buttazzo. “Integrating Multimedia Applications in Hard Real-Time Systems”. *Proceedings of the IEEE Real-Time Systems Symposium*, Madrid, Spain, December 1998

[2] 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.

[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] 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.

[7] Alan Burns, Andy J. Wellings and S. Tucker Taft. “Supporting Deadlines and EDF Scheduling in Ada”. 9th International Conference on Reliable Software Technologies, Ada-Europe, Palma de Mallorca (Spain), in Lecture Notes on Computer Science, Springer, LNCS 3063, June, 2004, ISBN:3-540-22011-9, pp. 156-165.

[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] Mario Aldea Rivas, J. Miranda and M. González Harbour. “Implementing an Application-Defined Scheduling Framework for Ada Tasking”. 9th International Conference on Reliable Software Technologies, Ada-Europe, Palma de Mallorca (Spain), in Lecture Notes on Computer Science, Springer, LNCS 3063, June, 2004, ISBN:3-540-22011-9, pp. 283-296.

[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 Rapporteur Group (ARG). “Execution-Time Clocks”, Ada Issue AI95-00307-1.13 . `http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00307.TXT`

[12] Ada Rapporteur Group (ARG). “Group Execution-Time Timers”. Ada Issue AI95-00354-1.8. `http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00354.TXT`

[13] Ada Rapporteur Group (ARG). “Support for Deadlines and Earliest Deadline First Scheduling”. Ada Issue AI95-00357-1.12. `http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00357.TXT`

[14] Ada Rapporteur Group (ARG). “Priority Specific Dispatching including Round Robin”. Ada Issue AI95-00355-1.9. `http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00355.TXT`

[15] Ada Rapporteur Group (ARG). “Timing events”. Ada Issue AI95-00297/10. `http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00297.TXT`

[16] Giorgio C. Buttazo. “Rate Monotonic vs. EDF: Judgment Day”. Journal of Real-Time Systems, Volume 29, Number 1, Jan 2005.

[17] 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.

[18] Bryan Ford and Sai Susarla, “CPU Inheritance Scheduling”. Proceedings of OSDI, October 1996.

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