Sáez Barona, S.; Real Sáez, JV.; Crespo, A. (2014). Reliable Handling of Real-Time Scheduling Attributes on Multiprocessor Platforms in Ada 2012. En Reliable Software Technologies – Ada-Europe 2014. Springer. 74-90. doi:10.1007/978-3-319-08311-7_7.

Reliable Handling of Real-Time Scheduling Attributes on Multiprocessor Platforms in Ada 2012

Sergio Sáez, Jorge Real, and Alfons Crespo

Instituto de Automática e Informática Industrial
Universitat Politècnica de València
Camino de vera, s/n, 46022 Valencia, Spain
{ssaez,jorge,alfons}@disca.upv.es

Abstract. The real-time attributes of a concurrent task define the parameters that will determine when the task can be allocated the required resources. Typical examples are the task's priority, the deadline, and the CPU (or CPUs) on which it must be executed. Since the 2012 revision, Ada is prepared for handling all these attributes. But the handling is per-attribute: it is not possible to change several attributes at a time, in a single call. Instead, they have to be changed one by one, which poses scheduling issues especially in multiprocessor platforms.
This paper proposes and discusses approaches for implementing atomic changes of multiple scheduling attributes, thus mitigating or eliminating those issues.

Keywords: Real-time systems, multiprocessor scheduling, Ada 2012.

1 Introduction

The scheduling attributes of a concurrent, real-time task define how resources are allocated to that task. Typical scheduling attributes are *priority*, *deadline* (in EDF-scheduled systems) and *CPU* of the task (in multiprocessor systems). Using Ada 2012 [1], a programmer can access these attributes and modify them according to changing application needs.

The package `System.Multiprocessors.Dispatching_Domains` supports the concept of dispatching domains for multiprocessor platforms, i.e., the set of processors on which a task can be executed. The package offers subprograms for querying and setting the current CPU for a task. It also provides the subprogram `Delay_Until_And_Set_CPU` to perform an atomic delay and change of CPU. The calling task will be assigned the new processor when the delay expires. This avoids the task taking too long to move to the destination CPU, when it has a relatively low priority in the original CPU.

The package `Ada.Dispatching.EDF` supports the concept of deadline and provides subprograms to query and set a task's deadline. Similarly to the CPU case, the package also provides one subprogram, `Delay_Until_And_Set_Deadline`, for atomically executing an absolute delay and setting a new relative deadline for the task. This atomicity is

needed so that the calling task can wake up from the delay at the priority level dictated by its new deadline, and not the previous one.

The package `Ada.Dynamic_Priorities` provides subprograms for querying and setting a task's priority at run time. As opposed to the cases of deadline and CPU, there is no subprogram provided for atomically changing the priority at the end of a delay (a hypothetical `Delay_Until_And_Set_Priority`). Hence it is possible that scheduling anomalies occur when a task needs to wake up from a delay with a changed priority. For example, if the task wakes up with an old low priority and then it wants to raise its priority to high, then it can suffer interference from mid-priority tasks in the interim. This issue can however be worked around by using a timing event rather than a delay statement. The timing event handler would change the task's priority from a high interrupt priority, hence reducing the scheduling error to an interference glitch to higher-priority tasks during the execution of the timing event handler.

So CPU and deadline can be separately changed with immediate or deferred effect, and priority can be immediately changed. But there is no functionality in Ada, however, that allows the programmer to change several scheduling attributes at a time, either immediately or right after a delay. Such scheme would be very useful for applications using prevalent multiprocessor techniques such as:

Job partitioning which alternates *jobs* of a task (execution instances) in different CPUs at possibly different priorities and/or deadlines. Here the task needs to change several of its attributes, and have them enforced by the next job activation.

Task splitting and dual-priority systems where a task may need to change CPU, priority, deadline or a combination of them after a programmed amount of real time or execution time [2–4].

Multimoded systems that potentially require changes to several attributes of tasks after a mode change request [5, 6].

The need for language changes to support this functionality is a current subject of discussion in a part of the Ada community [7–9]. One argument against changes or additions to the language is that, presumably, the programmer could use timing events to obtain a sufficiently effective solution [9]. Since the timing event executes at a very high priority, and it acts upon another task, the handler can change the priority or deadline of the target task, and then awake it from suspension, possibly in a different CPU. In this paper, we explore and implement this idea as well as other approaches. We then analyse different aspects of the implementations obtained, from system requirements to run-time behaviour. Our conclusion is that, although effective to a certain extent, approaches such as timing events are not absent from certain scheduling artefacts. These artefacts however, are shown to disappear in an alternative scheme based on the use of a server task for enforcing the attribute changes. The server task approach is a working solution in Ada 2012, although less efficient than it could be if the programmer could enforce the affinity of timing event handlers.

The paper is organised as follows. Section 2 motivates our study by showing the issues around the operation of changing scheduling attributes at run time, especially on multiprocessor platforms. Section 3 discusses design alternatives of safe mechanisms for handling multiple scheduling attributes in multiprocessors, either with immediate or delayed effect. Section 4 describes and discusses implementations of those design alternatives. Finally, Section 5 summarises our conclusions.

2 Motivation

Before we propose alternatives for properly handling task scheduling attributes in multiprocessor platforms, we will visit some scenarios that justify the need for a controlled mechanism. We will show how the order in which several attributes are changed is relevant for proper execution of the real-time schedule at run time, especially when the CPU is one of the changing attributes. Failing to apply changes in the correct order leads to scheduling errors. Even if the order is correct, failing to change them atomically will cause scheduling artefacts (short, bounded situations of priority inversion).

Consider Scenario A in figure 1, where we want to change both the priority and CPU[1] of a task τ in a 2-processor system, with CPU1 and CPU2. Task τ is running at priority 20 on CPU1. We want to move task τ to processor CPU2 with priority 10. If the change of CPU and priority is not atomic, then they must be done in some sequential order.

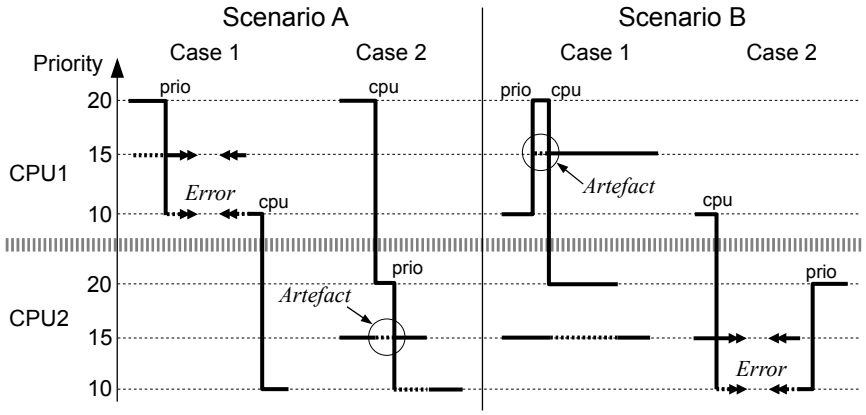

Fig. 1. Scenarios described in Section 2. Thick lines represent task execution. Dotted thick lines represent intervals of interference. Intervals between two double arrows are arbitrarily large. Errors affect the changing task τ. Artefacts affect tasks of priority 15.

Case 1 If the priority is changed first, then τ will have its priority lowered to 10 while in CPU1. This would make τ suffer a potentially large amount of interference from all tasks with priority higher than 10 in CPU1. The change to CPU2 will not occur until all tasks with a priority higher than 10 are idle on CPU1. This delay can be unacceptably large and it would break the assumptions of any static real-time scheduling analysis. The impact on the schedule would be especially notorious if the new priority of τ was the highest in CPU2.

Case 2 Let's now consider the reverse order. If the CPU is changed before the priority, then τ will move to CPU2 with priority 20, rather than 10. This will preempt any

[1] The same discussion applies for the case of changing deadline and CPU of deadline-scheduled tasks. We limit ourselves to priority and CPU for simplicity.

tasks with priority lower than 20, when τ should execute at priority as low as 10 on CPU2. The duration of this interference will however be bounded to the time between the change of CPU and the change of priority from 20 to 10. If that time is short and bounded, then this particular order (CPU, then priority) will cause only a scheduling artefact.

Consider now the reverse Scenario B: task τ has a priority 10 in CPU1 and wants to move to CPU2 with a priority 20. If we first change the priority, then we will cause interference in CPU1 to all tasks with priorities between 10 and 20. This interference will however last only the time it takes to move from CPU1 to CPU2. A situation similar to Case 2 above, but the artefact will occur in the original CPU1 rather than in the destination CPU2. In symmetry with the previous scenario, if we first change the CPU, then τ is moved to CPU2 with a priority 10 rather than 20, so it will not be able to raise its priority to 20 until all tasks with priorities between 20 and 10 are idle. This corresponds to Case 1 above, but the unbounded interference on τ would occur while in CPU2, rather than CPU1.

In the rest of this paper, we will explore design alternatives to remove the scheduling errors described in this Section. Note that if the scheduling artefacts can not be removed, at least they will only cause limited interference.

3 Design alternatives

The situations described above suggest that the change of several attributes of a target task, especially when the CPU attribute is involved, needs be done atomically. At an application level (i.e., not considering the implementation of the underlying system services to change task scheduling attributes) there are four Ada mechanisms we want to explore for pursuing the required atomicity:

- A protected object with the highest ceiling changes the target task attributes.
- The target task performs the changes of its own attributes using the highest priority to avoid interference from other tasks during the sequence of changes.
- A timing event is programmed to change the attributes of the target task.
- A server task changes the attributes of the target task using rendezvous at the highest priority.

In the following subsections, we explore the properties of these approaches to correctly solve the problem of changing several scheduling attributes at a time.

3.1 Using protected objects

Changing the priority, deadline or CPU of an Ada task are task dispatching points. However, all these operations are deferred if they occur within a protected action. This seems to give a chance to atomicity if all attribute changes are done within a protected action. By choosing the right ceiling priority for the protected object, the programmer can avoid priority inversions leading to scheduling errors such as those described in Section 2. In the most extreme case, the ceiling can be as high as `System.Interrupt_Priority'Last`, so that the attribute changes will never be preempted by any other application task or interrupt handler.

Taking Case 1 of Scenario A in Figure 1, if the change of priority and CPU was atomic, then task τ would travel to CPU2 without spending a potentially large amount of time suspended in CPU1, with a relatively low priority that would delay its migration. In Case 2 of Scenario B, the task would migrate to CPU2 with the correct priority 20, and hence it would not suffer priority inversion from lower-priority tasks in CPU2.

The deferred setting of multiple scheduling attributes (*à la* *Delay_Until_And_Set_Scheduling_Attributes*) can also be dealt with by requeuing the calling task in a private entry of the protected object, with a closed barrier, and programming a timing event for the required delay time. The timing event handler would then open the barrier of that private entry and release the waiting task with the new attributes.

Unfortunately, this analysis fails because we must also take into account how the underlying combination of operating system (OS) and run-time system actually implements the change of several attributes at the end of a protected action. When the task abandons the protected action it must have its priority and CPU changed. The original problem arises again: if the OS/runtime is requested to change the priority before the CPU, then the task will be inserted in the scheduling queue corresponding to its new priority. Since the change of CPU is still pending, then the error described for Scenario A Case 1 will happen again. If the OS/runtime is commanded to perform the changes in the reverse order, first CPU and then priority, then we would repeat the situation and scheduling error shown in Case 2 of Scenario B in Figure 1.

A solution to these problems can only come from the particular runtime implementation of how protected actions are completed. A safe design choice is to have the completion of the protected action executed at the maximum priority and follow the order: change CPU, then priority. But that falls out of the control of an application programmer.

3.2 Self change of attributes from the highest priority

A second approach is to have the changing task applying the changes to itself but after rising its priority to the highest possible (`Interrupt_Priority'Last`) so that it cannot be pre-empted in the middle of the changes. The sequence of operations in the task would be: (1) Set my own priority to the highest, where I cannot be pre-empted; (2) Call `Delay_Until_And_Set_CPU` to change to the target CPU at the specified absolute time (it could be immediately if `Set_CPU` is used instead); Finally (3) enforce my new scheduling attributes, in an order such that the last attribute changed is my priority, so that all changes are done from `Interrupt_Priority'Last`.

Figure 2 shows this approach and the impact it may have on other higher- and lower-priority tasks. On the left hand side, the task executes on CPU1 at priority 20 and wants to move to CPU2 with priority 15. At the time given in `Delay_Until_And_Set_CPU`, the task wakes up in CPU2, still with the highest priority. This may cause bounded interference on all tasks in CPU2. In particular, this will even preempt tasks with a priority higher than the destination priority of the changing task. This is marked as an artefact in the left-hand side of Figure 2.

On the right-hand side of Figure 2, the changing task executes on CPU1 at priority 15 and then migrates to CPU2 with priority 20. The impact on lower-priority tasks in CPU2 is just regular pre-emption.

We note however that any task running on CPU2 (below `Interrupt_Priority'Last`) could be subject to pre-emption bursts in the case of multiple tasks migrating to CPU2 in a short interval during its execution. And it would not matter whether the target

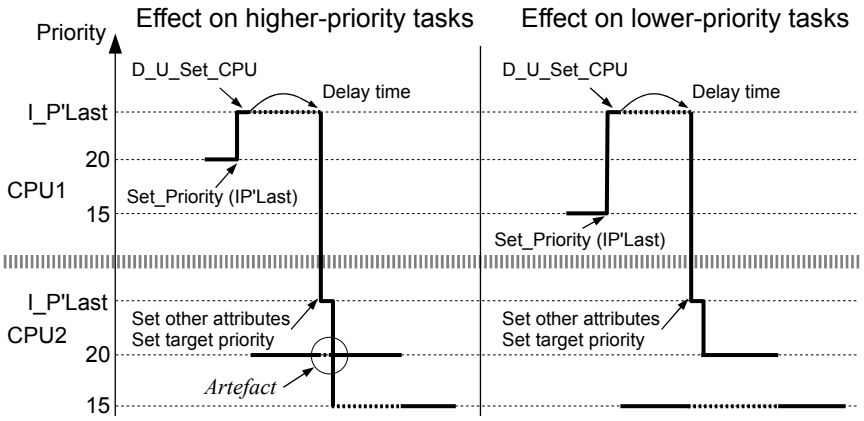

Fig. 2. Scenarios for the self-change approach and their impact on higher- and lower-priority tasks in the destination processor CPU2.

priorities of the pre-empting tasks were higher or lower than that of the pre-empted task (or tasks) in CPU2. This is an important drawback of this approach since it ruins the assumptions of schedulability analysis: any task on CPU2 could suffer interference from any other task potentially migrating to CPU2 from any other CPU.

3.3 Using timing events

For our purpose, and in terms of scheduling, the timing event mechanism has similar properties to using a protected action with the highest possible ceiling priority. Furthermore, timing events are amenable to more efficient implementations than protected objects, since they are simpler. An additional advantage is that the timing event handler can be programmed for the future. This is most suitable for the deferred setting of scheduling attributes, such as the Ada supported operations `Delay_Until_And_Set_CPU` and `Delay_Until_And_Set_Deadline` for individual attributes. Timing events are conceived for handling an event in the future, but the handler can be forced to execute immediately by programming the event for a time in the past.

But perhaps the principal advantage of using a timing event (*vs.* a protected object) is that the timing event handler will apply the attribute changes to *another task*. It can therefore change the task's priority without forcing a reschedule (since the handler is still executing at the highest priority) and then change the task's CPU before completing the execution of the handler. This eliminates the issue mentioned above with protected objects: even though the OS does not support the atomic change of several scheduling attributes of a task, since the changes are done from a timing event handler, there will be no other application tasks interfering the whole operation. Figure 3 shows how the scheduling errors described in Section 2 disappear and only the artefact glitches remain. Since the changes of priority and CPU are enforced from the timing event handler, with no possible pre-emption, the order in which they are performed is not relevant, hence the absence of cases 1 and 2 in Figure 3.

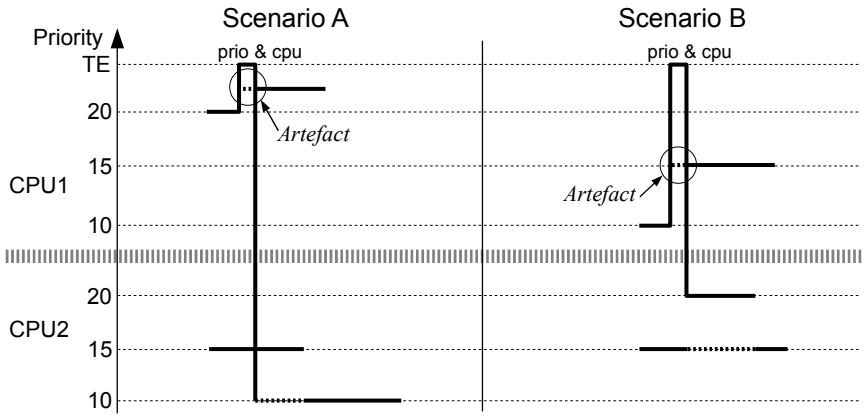

Fig. 3. An implementation based on a timing event prevents the scheduling errors shown in Figure 1. TE represents the priority at which the timing event handler is executed. In practice, under the Ceiling Locking policy, this priority is `Interrupt_Priority'Last`.

In summary, an implementation based on timing events effectively eliminates the scheduling errors described in section 2, but not the artefacts. We note that, unfortunately, it is unknown to the programmer in which CPU the artefact will occur, given that the underlying OS/runtime could choose any CPU to execute the timing event handler[2]. This poses a challenge to schedulability analysis and motivates us to explore how the hypothetical ability to set the affinity for timing events, would help solve this issue and determine precisely where the artefacts occur, which would provide invaluable information for schedulability analysis. We will develop this idea in Section 4.4.

3.4 Using rendezvous with a server task

We now explore an alternative approach whereby a server task is used for servicing requests of immediate and deferred changes of the scheduling attributes.

Assume one server task is in charge of applying the attribute changes to another calling (client) task. The server task has the highest priority, `Interrupt_Priority'Last`. A client task calls the appropriate server entry (for immediate or deferred change) and then it becomes blocked during the execution of the handled sequence of sentences of the corresponding accept statement on the server side. According to the Ada standard [1], the execution of the rendezvous occurs at the priority of the calling task. Since we do not want to reproduce the scheduling errors described above for the protected object scheme, we want the rendezvous to occur at the highest priority. To this end, the calling task will rise its priority to the highest before issuing the actual entry call to the server.

[2] In Figure 3 we have represented the timing event executing in the same CPU where the calling task executes, but nothing prevents the runtime system to execute the handler in CPU2 or even in a third CPU, if it exists.

Consider first the immediate setting of scheduling attributes by means of a procedure `Apply_Scheduling_Attributes`. Within this procedure, the calling task first rises its priority to `Interrupt_Priority'Last` and then calls an entry in the server task to enforce the new scheduling attributes. Upon completion of the entry call, the server task goes back to accepting new calls and the client task is released from the rendezvous blocking with the new attributes applied. The effect on the schedule is a bounded interference at the highest priority in the origin CPU, that can easily be accounted for in the schedulability analysis as blocking time for all tasks of a priority higher than the client task in the origin CPU. This is because the whole operation starts at the client's original priority in the origin CPU.

Figure 4 shows the process for both immediate and deferred changes. In the immediate case (left side of Figure 4), a task in CPU1 with priority 20 wants to change its priority to 10 and CPU to CPU2. The task first rises its priority to the highest and then calls the appropriate entry in the server task. During the rendezvous (represented by a grey box) the server task enforces the changes of priority and CPU with no possible interference from other tasks. Note that, as opposed to what occurs with timing events, the artefacts depicted in Figure 4 will always occur in the origin CPU, so although they exist, they can be predictably accounted for in the schedulability analysis. We have represented the continuous execution of a second task of priority 15 in CPU2 to show that there are no glitches caused by this approach.

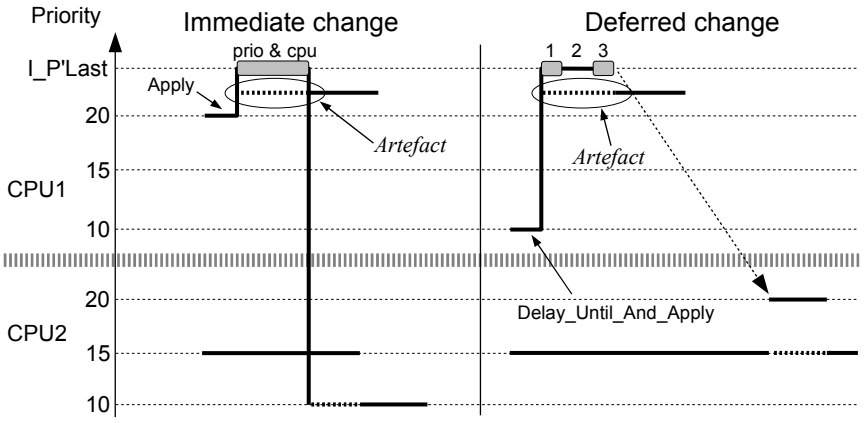

Fig. 4. Execution examples for immediate and deferred change of scheduling attributes using an implementation based on server tasks. Gray boxes represent execution of the server task. The explanation for marks 1, 2 and 3 is given in Section 3.4.

Consider now the deferred change of attributes, enabled by a procedure `Delay_Until_And_Apply_Scheduling_Attributes`. The right hand side of Figure 4 shows the case of a task running on CPU1 at priority 10, that wants to migrate to CPU2 with priority 20 at a given time in the future. As in the immediate case, the first thing to do is to rise the priority of the task to the highest. Then the entry call is issued and the rendezvous starts. In this case, the rendezvous only copies the parameters of the call to apply them at a later stage. This is marked as step 1 on the right hand side

of Figure 4. After step 1 the rendezvous completes and we have both server and client at the highest priority, since we have not yet changed any of the task's attributes. We now want to make sure that the client task goes on and executes a `delay until` sentence to suspend itself until the requested time. This is marked as step 2 in Figure 4. Having the client task suspended, the server task now executes step 3, where it simply enforces the new attributes to the (suspended) client task. The result is that no scheduling errors can occur, and the artefacts are actually only short blocking times. The whole operation of changing the priority and CPU of a task can be accounted as blocking time for tasks of a higher priority than the changing task in the origin CPU. And this situation of priority inversion always occurs in the origin CPU. This is a clear advantage with respect to the timing event alternative described in Section 3.3, where it can't be predicted in which CPU the priority inversion will occur.

To ensure that the steps will occur exactly in the order described here (1, then 2, then 3), the server task has to yield the processor after the rendezvous (step 1), so that the client task executes the delay statement (step 2) and is placed at the tail of the ready queue of `Interrupt_Priority'Last`. After that, the server task gains the CPU again to perform the change of attributes on the suspended client – this is because the only two tasks executing at the highest priority level are the client and the server. Indeed, this protocol causes two context switches between server and client, but the important benefit is that the associated overhead is bounded and predictable.

4 Implementation

This section discusses implementation details of the design alternatives described so far. After briefly restating the implementation goals, we will first look at the details of a data type to capture all the scheduling attributes of tasks, and its related primitive operations. We will then show the most relevant implementation aspects of the alternatives proposed in sections 3.2, 3.3 and 3.4, that is, having the attributes changed by the target task itself, or by a timing event or by a server task, respectively. We are not considering the implementation details of the protected object approach described in Section 3.1 because the way it behaves is heavily dependent on the underlying combination of OS and runtime support.

4.1 Goals

The goal of the software under design is to provide an abstraction to capture the set of scheduling attributes (priority, deadline, CPU, or other user-defined attributes...) individually associated to each task in the system. The programmer must be allowed to query these attributes, and to atomically change one or more individual attributes at a time. The change of these attributes can either have immediate or deferred effect, at a certain specified absolute time in the future. The type representing the set of attributes shall be extendable, so that application-specific scheduling attributes can be added at a later time.

4.2 Representation of scheduling attributes

Listing 1 shows the specification of the tagged type `Scheduling_Attributes`, that captures the scheduling attributes of a task. An instance of this type (or a type derived

from it) is associated with each task so that the whole set of attributes can be passed to a changer subprogram in a single call. The type was already proposed in [8], and extended with a derived type for including a deadline attribute for deadline-scheduled tasks. We show here only the two attributes defined in the root type, priority and CPU.

The implementation details in the private part show that the type is simply an extensible record with the proper fields, one for each attribute. The class-wide type `Any_Scheduling_Attributes` is needed for the class-wide operations `Apply_Scheduling_Attributes` and `Delay_Until_And_Apply_Scheduling_Attributes`. These subprograms internally use the private subprogram `Enforce_Scheduling_Attributes`, which is in charge of ultimately setting the attributes. It has to be implemented for each extension of the type, since the parameters to change will vary between those extensions. Listing 2 shows the implementation of the corresponding root operation.

The type is simple enough, and the operations are common setters and getters. The subprogram `Retrieve_Scheduling_Attributes` allows the programmer to read the attributes of a task. This is needed when we only need to change a subset of the attributes and leave the rest intact.

Listing 1. Data type for scheduling attributes and primitive operations

```
-- with clauses omitted
package Ada_Real_Time.Scheduling_Attributes is
   -- Data type to represent scheduling  attributes
   type Scheduling_Attributes is tagged private;
   procedure Set_Priority (SP : in out Scheduling_Attributes ; Prio: Any_Priority );
   function Get_Priority (SP : Scheduling_Attributes ) return Any_Priority ;
   procedure Set_CPU (SP : in out Scheduling_Attributes ; CPU_Nr: CPU_Range);
   function Get_CPU (SP : Scheduling_Attributes) return CPU_Range;
   procedure Retrieve_Scheduling_Attributes (SP : in out Scheduling_Attributes ;
                                             T_Id : Task_Id := Current_Task);
   type Any_Scheduling_Attributes is access all Scheduling_Attributes 'Class;

   -- Class-wide procedures
   procedure Apply_Scheduling_Attributes (SP : Any_Scheduling_Attributes ;
                                          T_Id : Task_Id := Current_Task);
   procedure Delay_Until_And_Apply_Scheduling_Attributes (SP : Any_Scheduling_Attributes ;
                                                         Delay_Until_Time : Time);
private
   type Scheduling_Attributes is tagged
      record
         Prio    : Any_Priority := Default_Priority ;
         CPU_Nr : CPU_Range    := Not_A_Specific_CPU;
      end record;

   procedure Enforce_Scheduling_Attributes (SP : Scheduling_Attributes ; T_Id : Task_Id );

end Ada_Real_Time.Scheduling_Attributes;
```

Listing 2. Root subprogram in charge of ultimately changing the attributes

```
   procedure Enforce_Scheduling_Attributes (SP : Scheduling_Attributes ; T_Id : Task_Id) is
   begin
      Set_Priority ( Priority => SP.Prio, T => T_Id);
      Set_CPU (CPU => SP.CPU_Nr, T => T_Id);
   end Enforce_Scheduling_Attributes ;
```

The implementation of a particular approach to handle the attributes will be fully contained in the subprograms `Apply_Scheduling_Attributes` and `Delay_Until_And_Apply_Scheduling_Attributes`. The following subsections illustrate the implementations of the three viable design alternatives discussed in Section 3.

4.3 Implementation based on self changing the attributes

Listing 3 shows the implementation of the deferred change of attributes contained in subprogram `Delay_Until_And_Apply_Scheduling_Attributes`. The key aspect of this approach (as described in Section 3.2) is that the changing task has the highest priority before it changes to the target CPU (line 5 of Listing 3). This ensures that it will continue to execute at the highest priority when it arrives in the destination CPU. Then we change the other attributes (all but CPU and priority, if any) and finally, we change the priority.

Note that the last two sentences (lines 8 and 9) will be executed on the target CPU, thus interfering with tasks that may have a higher priority than the final priority of the changing task.

Listing 3. Deferred change of attributes in the self-change approach

```
1  procedure Delay_Until_And_Apply_Scheduling_Attributes (SP : Any_Scheduling_Attributes;
2                                                         Delay_Until_Time : Time) is
3  begin
4      Set_Priority ( Interrupt_Priority 'Last); -- Rise caller's priority to highest
5     Delay_Until_And_Set_CPU(Delay_Until_Time,SP.CPU_Nr);
6     -- Caller wakes up from delay in the destination CPU and still with the highest priority
7     SP. Enforce_Scheduling_Attributes (Current_Task); -- Update other attributes
8      Set_Priority (SP.Prio); -- Decrease caller's priority down to the target priority
9  end Delay_Until_And_Apply_Scheduling_Attributes;
```

4.4 Implementation based on timing events

As discussed in Section 3.3, timing events could lead to a more efficient implementation than the self-changing and server task approaches. However, the interference caused by a timing event handler cannot be bound to a particular CPU, since the affinity of timing events cannot be enforced.

Listing 4 shows a hypothetical, extended specification of the `Set_Handler` subprogram (for programming a timing event handler). This extension adds the parameter `CPU_Nr` to set the handler's affinity. At the low level, the operation hides the complexity of adding timed events to the timer queue of a different CPU. If we had this in Ada, then we could remove the most important drawback of the timing-event approach to handling scheduling attributes.

Listing 4. A proposed signature for timing events with CPU affinity

```
procedure Set_Handler (Event : in out Timing_Event; At_Time : Time; Handler : Timing_Event_Handler;
                       CPU_Nr : CPU_Range := Get_CPU);
```

The use of this hypothetical feature could be as follows. Assume a *Scheduling Manager* abstraction is declared for each task whose scheduling attributes may be changed. This scheduling manager maintains the scheduling attributes of the task and enables their enforcement when the timing event expires. Listing 5 shows a part of the implementation of a protected object supporting the scheduling manager abstraction.

The immediate change of scheduling attributes, implemented within the entry body of `Apply_Scheduling_Parameters`, programs an already expired timing event to force the execution of the handler as soon as the calling task completes the entry call. In the case the calling task is also the target task, it also forces the task to wait until the handler is executed. The deferred setting of attributes is implemented in `Delay_Until_And_Apply_Scheduling_Attributes`. It is similar to the immediate case, but using an expiration time in the future.

Listing 5. Timing Event handler and protected operations

```
-- Scheduling Manager specification
protected type Scheduling_Manager with Interrupt_Priority => Interrupt_Priority'Last is
   entry Apply_Scheduling_Attributes (SP: Any_Scheduling_Attributes);
   entry Delay_Until_And_Apply_Scheduling_Attributes (SP : Any_Scheduling_Attributes;
                                                      Delay_Until_Time : Time);
private
   entry Wait;
   procedure Handler (Event : in out Timing_Event);

   Task_Waiting : Boolean := false;
   Sched_Params : Any_Scheduling_Attributes;
   Timing_Ev : Timing_Event;
end Scheduling_Manager;

-- Scheduling Manager body
protected body Scheduling_Manager is

   entry Wait when not Task_Waiting is
   begin
      null;
   end Wait;

   procedure Handler (Event : in out Timing_Event) is
   begin
      Sched_Params.Enforce_Scheduling_Attributes(Owner_Task);
      Task_Waiting := false;
   end Handler;

   entry Apply_Scheduling_Attributes (SP: Any_Scheduling_Attributes) when True is
   begin
      if not Task_Waiting then
         Sched_Params := SP;
         if Scheduling_Manager.Apply_Scheduling_Attributes'Caller /= Owner_Task then
            -- A task wants to change another task's attributes
            Timing_Ev.Set_Handler(Time_First, Handler'Access, SP.Get_CPU);
         else
            -- A task wants to change its own scheduling attributes
            Task_Waiting := True; -- Barrier for entry Wait
            -- An immediate timing event is programmed...
            Timing_Ev.Set_Handler(Time_First, Handler'Access, SP.Get_CPU);
            -- ... and the task is requeued to Wait until Handler updates its attributes
            requeue Wait;
         end if;
      end if;
   end Apply_Scheduling_Attributes;

   entry Delay_Until_And_Apply_Scheduling_Attributes (SP : Any_Scheduling_Attributes;
                                                      Delay_Until_Time : Time) when True is
   begin
      Sched_Params := SP;
      Task_Waiting := True;
      -- Program a TE for Delay_Until_Time ...
      Timing_Ev.Set_Handler(Delay_Until_Time, Handler'Access, SP.Get_CPU);
      -- ... and wait until Handler wakes me up with the new attributes
      requeue Wait;
   end Delay_Until_And_Apply_Scheduling_Attributes;

end Scheduling_Manager;
```

4.5 Implementation based on server tasks

In the approach described in Section 3.4, a server task is used to atomically change the scheduling attributes of a client task. Listing 6 shows an implementation of the server

task type, from which server objects can be instantiated. We may need up to one server task per CPU. The function `Next_CPU`, used as the initial value for the discriminant `CPU_Nr` of the task type, is just a global function that returns one distinct CPU number upon each call, all within the range of CPUs available in the execution platform.

Each server task offers two entries that will be used by the class-wide operations to implement the immediate and deferred changes. When a call is accepted to the entry `Apply_Attributes_Immediately`, the server simply calls the procedure that enforces the new attributes. As Listing 7 shows, the entry call is sent to the particular server task that is attached to the CPU where the changing task is executing. We have omitted the implementation of the function `Current_CPU`, which is dependent on the particular underlying operating system.

Listing 6. Server task that atomically enforces the scheduling attributes

```
task type Scheduling_Manager_Type(CPU_Nr : CPU := Next_CPU) with
   Interrupt_Priority   => Interrupt_Priority'Last,   CPU => CPU_Nr is
  -- entries omitted
end Scheduling_Manager_Type;

task body Scheduling_Manager_Type is
   Sched_Param : Any_Scheduling_Attributes;
   Target_Task: Task_Id;
begin
   loop
      select
         accept Apply_Attributes_Immediately (SP : in Any_Scheduling_Attributes;
                                              T_Id : Task_Id) do
            -- Change task's attributes
            SP.Enforce_Scheduling_Attributes(T_Id);
         end Apply_Attributes_Immediately;
      or
         accept Apply_Attributes_On_Suspend (SP : in Any_Scheduling_Attributes;
                                             T_Id : Task_Id) do
            -- Stores the target task and new attributes (Fig. 3 step 1)
            Target_Task := T_Id;
            Sched_Param := SP;
         end Apply_Attributes_On_Suspend;
         -- Forces client task to execute the delay until (Fig. 3 step 2)
         delay 0.0;
         -- Change the sched. attributes of the suspended client task (Fig. 3 step 3)
         Sched_Param.Enforce_Scheduling_Attributes(Target_Task);
      or
         terminate;
      end select;
   end loop;
end Scheduling_Manager_Type;
```

The second entry in Listing 6, `Apply_Attributes_On_Suspend`, implements steps 1 and 3 shown at the right-hand side of Figure 4. The sentence `delay 0.0` after the *accept* forces the server task to move to the tail of the `Interrupt_Priority'Last` ready queue[3]. This allows the client task, that has the same highest priority, to execute the statement `delay until Delay_Until_Time`, within subprogram `Delay_Until_And_Apply_Scheduling_Attributes` shown in Listing 7 (labelled as step 2 in Figure 4).

Once the client task is suspended, the server task resumes execution (it is the highest-priority active task) and changes the scheduling attributes of the suspended client task. When the client task wakes up, it will be inserted in the corresponding

[3] The `Yield` operation is not supported in our platform.

ready queue of the target CPU, but with the new scheduling attributes already applied. Therefore, no scheduling interferences at application level will occur[4].

Listing 7. Class-wide operations of the scheduling attributes

```
-- Class-wide procedures
procedure Apply_Scheduling_Attributes (SP : Any_Scheduling_Attributes;
                                       T_Id : Task_Id := Current_Task) is
begin
   Set_Priority ( Interrupt_Priority 'Last );
   Scheduling_Manager(Current_CPU).Apply_Parameters_Immediately(SP, T_Id);
end Apply_Scheduling_Attributes;

procedure Delay_Until_And_Apply_Scheduling_Attributes (SP : Any_Scheduling_Attributes;
                                                       Delay_Until_Time : Time) is
begin
   Set_Priority ( Interrupt_Priority 'Last ); -- Rise its priority to IP'Last
   Scheduling_Manager(Current_CPU).Apply_Parameters_On_Suspend(SP,
                                                              Current_Task);
   -- Sched. attributes will be change on suspension (Fig. 3 step 2)
   delay until Delay_Until_Time; -- It will wake up with new attributes applied
end Delay_Until_And_Apply_Scheduling_Attributes;
```

5 Conclusions

The ability to safely change several scheduling attributes of a task in a single operation, is a useful feature for real-time systems, especially on multiprocessor platforms. It is on these platforms that the operation poses the biggest challenges, since there are multiple opportunities for scheduling issues to occur at run time. This is especially true when the CPU is one of the changing attributes. In this paper, we have described those issues and explored four ways to solve them in Ada. We conclude that:

- A solution based on protected objects does not guarantee, at the language level, the required atomicity in the change of several scheduling attributes. Even if the protected action was executed at the highest possible ceiling. This is because we ultimately depend on how the OS/runtime implements the lower-level services to actually enforce the scheduling attributes.
- Self-changing the attributes from the highest priority level, introduces remote interference in the destination CPU. Although this interference is presumably short (just the time it takes to call the OS/runtime support to have the changes enforced, plus a task context switch in and out the destination CPU), a task in the destination CPU may suffer bursts of interference when many tasks are migrating to the CPU where it is running.
- Delegating on a timing event handler for the change of attributes, while the changing task is safely blocked awaiting for the handler to execute. This approach produces one interference glitch, presumably shorter than in the previous case since no task context switch is strictly necessary here. However, it is impossible for the programmer to determine in which CPU the interference will occur (unless it is very precisely documented in the language implementation). Moreover, at least one implementation that we know of uses a task for servicing timing events, hence the efficiency argument does not necessarily hold in all cases. We note that the

[4] The possible scheduling interferences that a task activation produces at kernel level depend ultimately on the implementation of the underlying operating system.

main weakness of this approach (namely, ignoring which CPU is affected by the handler glitch) could be overcome by adding a new language feature that enabled the programmer to set the affinity of timing event handlers.
- Using a server task to change the attributes of client tasks has proven to be the most reliable implementation in our experience. This approach requires up to one server task per CPU and produces interference only in the origin CPU, where it can be accounted for as interference from the highest priority level. Since all attribute changes are applied to a non-running task, they will be actually enforced at the next task activation. There is no need for further operations that are susceptible of causing additional interference. Our own criticism to this approach is that we need a double context switch between the server and the client tasks in order to apply all the changes in a controlled manner.

Acknowledgements

This work has been partially supported by the Spanish Government's projects COBAMI (DPI2011-28507-C02-02) and Hi-PartES (TIN2011-28567-C03-01-02-03) and the European Commission's MultiPARTES project (FP7-ICT-2011.3.4, Contract 287702).

References

1. ISO/IEC JTC1 SC22 WG9 Ada Rapporteur Group: Ada Reference Manual - Language and Standard Libraries - ISO/IEC 8652:2012(E), `http://www.ada-europe.org/manuals/LRM-2012.pdf`
2. Davis, R., Wellings, A.: Dual priority scheduling. In: Real-Time Systems Symposium, 1995. Proceedings., 16th IEEE. (1995) 100–109
3. Kato, S., Yamasaki, N., Ishikawa, Y.: Semi-partitioned scheduling of sporadic task systems on multiprocessors. In: 21st Euromicro Conference on Real-Time Systems, ECRTS 2009, Los Alamitos, CA, USA, IEEE Computer Society (2009) 249–258
4. Lakshmanan, K., Rajkumar, R., Lehoczky, J.P.: Partitioned fixed-priority preemptive scheduling for multi-core processors. In: 21st Euromicro Conference on Real-Time Systems, ECRTS 2009, IEEE Computer Society (2009) 239–248
5. Tindell, K., Burns, A., Wellings, A.: Mode changes in priority preemptively scheduled systems. In: Real-Time Systems Symposium, 1992. (1992) 100–109
6. Real, J., Crespo, A.: Mode Change Protocols for Real-Time Systems: A Survey and a new Proposal. Real-Time Systems **26**(2) (March 2004) 161–197
7. Sáez, S., Crespo, A.: Deferred setting of scheduling attributes in Ada 2012. Ada Letters **33**(1) (June 2013) 93–100
8. Sáez, S., Real, J., Crespo, A.: Deferred and atomic setting of scheduling attributes for Ada. Ada Letters **33**(2) (November 2013) 97–108
9. Vardanega, T., White, R.: Session summary: Improvements to Ada. Ada User Journal **34**(4) (December 2013) 239–241