Sáez Barona, S.; Real Sáez, JV.; Crespo, A. (2013). Deferred and atomic setting of scheduling attributes for ada. Ada Letters. 33(2):97-108. doi:10.1145/2552999.2553010.

Deferred and Atomic Setting of Scheduling Attributes for Ada

Sergio Sáez, Jorge Real, and Alfons Crespo
{ssaez|jorge}@disca.upv.es
Universitat Politècnica de València, Spain

Abstract

Deferred setting of scheduling attributes refers to a single operation that sets a new value for a scheduling attribute of a task at some future time. Although deferred setting of scheduling attributes is possible in Ada 2012, it is in a rather limited way: only deadline or CPU can be changed deferredly, either at a specified time or when the task is released from a suspension object. And only one of those two attributes at a time. Other scheduling attributes such as priority cannot have deferred setting by means of a single operation. This would be a convenient feature to have for schemes such as job partitioning, task splitting, or mode changes. Another issue is the absence of operations for atomically changing several parameters at a time, which would avoid scheduling issues specially on multiprocessors.

In this paper we explore a proposal aimed at correcting these two drawbacks. On one hand, we want to be able to change more attributes, not only deadlines, deferredly or immediately. On the other hand, we want to atomically change (now or later) a set of attributes, thereby avoiding scheduling artifacts that arise from sequentially changing several attributes, specially when the CPU is one of them.

Rather than providing a number of library operations for postponing the setting of a variety of scheduling attributes, we propose to encapsulate the scheduling attributes of each task in a single tagged type that can be extended with more attributes for specific applications if needed.

1 Introduction

Scheduling attributes refer to task attributes such as priority, CPU affinity, period, deadline, etc. Deferred setting of scheduling attributes refer to the ability for the programmer to specify that one or more scheduling attributes of a task need be set to a new value but not immediately, but at some point in time in the future, such as the next activation time of the task. One example of deferred setting of a scheduling attribute for a task is the procedure `Delay_Until_And_Set_Deadline` from the Ada 2005 standard package `Ada.Dispatching.EDF`. Similarly, sporadic tasks can also have their deadline changed upon their next activation if they use a suspension object for activation control. This can be achieved by means of `Suspend_Until_True_And_Set_Deadline` from package `Ada.Synchronous_Task_Control.EDF`.

At the 15th IRTAW, Mario Aldea, chairman of the workshop, summarized the discussion initiated around this topic as follows [**?**]:

> *The discussion about this [deferred attributes] topic was started in session 1.A and finished in this [closing] session. A presentation was made on the existing limitations of the current model of setting attributes (priority, deadline and affinity) that can cause undesirable effects when trying to change several of them simultaneously for the same task. There was some discussion about whether these changes could be performed atomically from inside a protected operation. The conclusion was that this is not a valid approach when changing other task's attributes. The group sentiment was that a mechanism is required to allow deferred attribute setting for the next*

dispatching point of a task. Two alternative implementations of the aforementioned mechanism were discussed: using an attributes object or using a set of procedures. It was agreed that this issue needs further investigation, modeling and trial implementations.

From the two alternative implementations mentioned at the end of this quote, we want to propose a model that uses the first approach and encloses the setting of several scheduling parameters in a single operation that can be executed atomically. By doing so, undesirable artifacts are avoided at run time, specially on multiprocessor platforms. A single *container* object for all relevant scheduling attributes has also advantages if it is tagged, as we will show.

The rest of this paper is structured as follows. Section 2 describes the problem context. Section 3 defines our proposal. Section 4 discusses implementation issues related to the proposal and that need be solved. A use example is given in Section 5. Finally, Section 6 concludes the paper.

2 Context and problem description

The problems we are trying to solve with this proposal were already described in [**?**]. For convenience, we give here a brief description. The following actions are sources for scheduling decisions and also for potential scheduling issues:

1. Changing a single task's scheduling parameter.
2. Changing a single scheduling parameter in the future.
3. Changing a set of scheduling parameters, either *now* or in the future.

By *scheduling parameters* we mean task parameters that have an impact on how the system schedules that task, including priority, deadline and CPU, and possibly other user-defined attributes.

The first case is solved in Ada by delaying the actual effect of the parameter change until the task's next dispatching point. The second case involves a parameter change plus the execution of a `delay until` statement. Ada allows the deferred setting of scheduling attributes for the cases of deadline and CPU by means of subprograms `Delay_Until_And_Set_Deadline` and `Delay_Until_And_Set_CPU`. The third case, however, cannot be cleanly solved in Ada. The case is particularly problematic in the context of multiprocessor platforms, when the CPU attribute is one of the parameters to be changed. In other words, tasks or jobs need to (dynamically) migrate to a different processor. This is the case of multi-moded systems and also in multiprocessor scheduling approaches such as job partitioning and task splitting [**?**, **?**], to give some examples.

Figures 1 and 2 (reproduced from [**?**]) show how a task can miss its deadline when it tries to simultaneously change both its priority and its target CPU. In both scenarios, the task `T0` migrates from one CPU to another, but uses a different priority in the target CPU. The expressions `TX/PY` in these figures denote *task X / priority Y*.

In Figure 1, task `T0` misses its deadline while executing the `Set_Priority + Set_CPU` sequence. The expected execution would be that task `T0` migrates to CPU1 and preempts `T2`, while `T1` becomes the highest priority task in CPU0, as shown in the *expected execution* side of the figure. However, right after `T0` changes its priority, the task `T1` has the highest priority on CPU0, preempts `T0` and therefore impedes it to execute the `Set_CPU` statement until it is too late (after deadline D0). Figure 2 shows a different situation where the incorrect behavior is caused by the sequence `Set_CPU + Set_Priority`. In this second case, `T0` migrates to CPU1 with the wrong priority, hence it can not preempt `T2` and is dispatched too late to meet its deadline. The only solution to these situations is to provide a mechanism to simultaneously change the priority and the target CPU. Similar issues arise under multiprocessor EDF dispatching with respect to `Set_Deadline` and `Set_CPU`.

Although the scenarios shown in Figures 1 and 2 can be solved by encapsulating both `Set_CPU` and `Set_Priority` within a protected operation, this cannot be done when the change of priority or deadline, and target CPU is combined with a `delay until` statement. As shown in the two following code examples, no correct sequence of code can be found using the current multiprocessor support in Ada. Note that these sequences of code are natural ways to implement job partitioning schemes, for setting the CPU where the next job is going to be executed before the current job finishes; and also task splitting, for resetting the original CPU at the end of the job.

```
loop
  -- Task code
  ...
  Next_Time := Next_Time + Period;
```

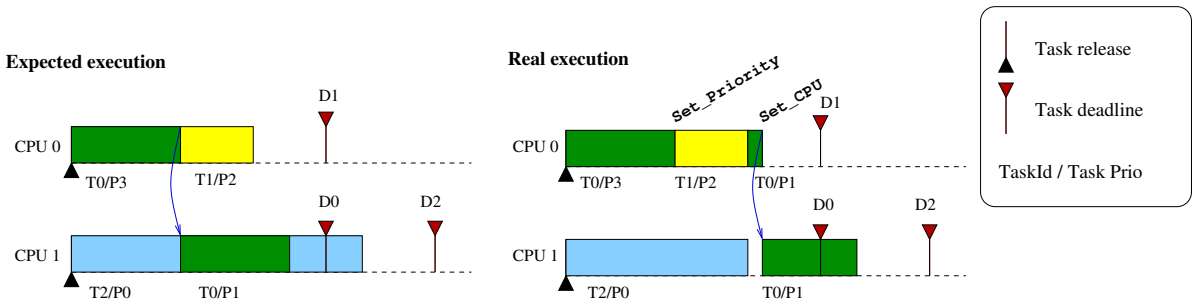

Figure 1. Expected and real executions of a `Set_Priority + Set_CPU` sequence.

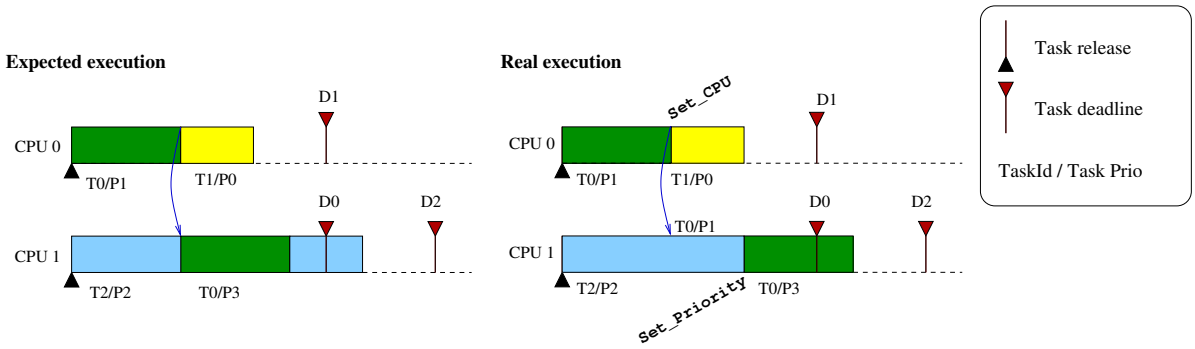

Figure 2. Expected and real executions of a `Set_CPU + Set_Priority` sequence.

```
  Set_Deadline(Next_Time + Relative_Deadline);
  Delay_Until_And_Set_CPU(Next_Time, Next_CPU);
  -- Similar to scenario with Set_Priority + Set_CPU
end loop;
```

```
loop
  -- Task code
  ...
  Next_Time := Next_Time + Period;
  Set_CPU(Next_CPU);
  Delay_Until_And_Set_Deadline(Next_Time, Relative_Deadline);
  -- Similar to scenario with Set_CPU + Set_Priority
end loop;
```

3 Proposal

The main ideas behind our proposal are:

- An object of a tagged type contains the relevant scheduling parameters (or attributes) for any given task. Let's call this type `Sched_Params`. In principle, `Sched_Params` contains only the CPU and priority of the task.
- The first natural extension to `Sched_Params` is to add a field for representing the deadline of a task. This is useful only for tasks scheduled under deadline-based policies such as EDF, hence we propose it as an extension to the root type.

- The Sched_Params type can also be extended by the user with other parameters that are relevant for a particular application. **Examples could be urgency level, offsets, capacity in server tasks, etc.**
- The following operations are possible over Sched_Params objects:

  Set_*Attribute* Sets the new value for a given attribute in the Sched_Params object. The *Attribute* part of this setter refers to the attributes priority or CPU for the root type. Derived types may define setters for additional parameters, such as deadline for the Sched_Params of an EDF task (see later).

  Get_*Attribute* Obtains the current value of a given attribute from the Sched_Params object.

  Apply_Sched_Params As the name indicates, makes the scheduling parameters effective *immediately* and atomically. This procedure can be applied to the currently executing task or to another given task.

  Delay_Until_And_Apply_Sched_Params This is to delay the task until a given time and atomically apply the set of scheduling parameters defined in a Sched_Params object.

  Suspend_Until_True_And_Apply_Sched_Params For symmetry with the existing procedure Suspend_Until_True_And_Set_Deadline, included in Ada.Synchronous_Task_Control.EDF.

The following listings give the profiles and location (new Ada library packages) for the proposed functionalities. We will first consider the type Sched_Params, a tagged record that holds a minimal set of scheduling attributes, and can be extended with more attributes if needed. In listing 1 we propose a new library package Ada.Scheduling_Parameters for the definition of this type.

Listing 1. Definition of the root type Sched_Params

```
with System, System.Multiprocessors, Ada.Task_Identification, Ada.Real_Time;
use System, System.Multiprocessors, Ada.Task_Identification, Ada.Real_Time;

package Ada.Scheduling_Parameters is

  type Sched_Params is tagged private;

  procedure Set_Priority (SP : in out Sched_Params; Prio: Any_Priority);
  function Get_Priority (SP : Sched_Params) return Any_Priority;

  procedure Set_CPU (SP : in out Sched_Params; CPU_Nr: CPU_Range);
  function Get_CPU (SP : Sched_Params) return CPU_Range;

  procedure Apply_Sched_Params (SP : Sched_Params; T_Id : Task_Id := Current_Task);
  procedure Delay_Until_And_Apply_Sched_Params (
    SP : Sched_Params;
    Delay_Until_Time : Time;
    T_Id : Task_Id := Current_Task);

private
  type Sched_Params is
    record
      Prio : Any_Priority := Default_Priority;
      CPU_Nr : CPU_Range := Not_A_Specific_CPU;
    end record;

end Ada.Scheduling_Parameters;
```

A first extension for the root type Sched_Params will include a deadline parameter, useful for EDF tasks. In listing 2 we propose a new child package of Ada.Scheduling_Parameters to include the new type Sched_Params_EDF, derived from Sched_Params, and setter and getter subprograms for the new deadline parameter. The package also provides new procedures to apply these extended scheduling parameters to EDF tasks. Note that we also include a new scheduling parameter At_Time that we explain below.

Listing 2. Extension of root scheduling parameters for EDF

```
with Ada.Real_Time; use Ada.Real_Time;

package Ada.Scheduling_Parameters.EDF is

   type Sched_Params_EDF is new Sched_Params with private;

   procedure Set_Deadline (SP : in out Sched_Params_EDF; D : Time_Span);
   function Get_Deadline (SP : Sched_Params_EDF) return Time_Span;

   procedure Set_At_Time (SP : in out Sched_Params_EDF; At_Time : Time);
   function Get_At_Time (SP : Sched_Params_EDF) return Time;

   procedure Apply_Sched_Params (SP : Sched_Params_EDF; T_Id: Task_Id := Current_Task);
   procedure Delay_Until_And_Apply_Sched_Params (
      SP : Sched_Params_EDF;
      Delay_Until_Time : Time;
      T_Id : Task_Id := Current_Task);
private
   type Sched_Params_EDF is new Sched_Params with
      record
         Relative_Deadline : Time_Span := Time_Span_Last;
         At_Time : Time := Time_Last;
      end record;

end Ada.Scheduling_Parameters.EDF;
```

We propose the deadline parameter to be of the type `Time_Span`. The semantics of `Delay_Until_And_Apply_Sched_Params` would be that the new absolute deadline is set for the time `Delay_Until_Time` plus the relative deadline given in the `Sched_Params_EDF` object. The absolute deadline is not so obvious in the case of `Apply_Sched_Params`. It could be the result of adding the relative deadline to the real-time clock value during the execution of `Apply_Sched_Params`. But that clock value is uncertain. We therefore propose to include the additional parameter `At_Time`, an absolute time taken as the reference to calculate the next absolute deadline for the task.

We considered the possibility of using such an absolute time reference as an additional parameter passed as a third parameter to `Apply_Sched_Params`. What makes that approach unattractive is that the profile for the primitive `Apply_Sched_Params` defined in package `Ada.Scheduling_Parameters` would no longer be valid for all cases, since that primitive does not include such parameter. Note that the functionality provided by such `At_Time` attribute is not achievable with `Delay_Until_And_Apply_Sched_Params`. For example, we may want to promote a task by shortening its absolute deadline after a certain time of the task's execution. Hence the task cannot delay until a certain time and then shorten its deadline, since it needs to be executing code meanwhile.

Finally, a link between Ada's synchronous task control and scheduling parameters would be useful for sporadic tasks whose activation is regulated by a suspension object. This is in line with the existing subprogram `Suspend_Until_True_And_Set_Deadline`, which is limited to setting only the deadline for the next activation of a task waiting on a suspension object. In listing 3 We propose a child package `Ada.Synchronous_Task_Control.Scheduling_Parameters` to contain the new functionality[1].

Listing 3. Synchronous task control and scheduling attributes

```
with Ada.Scheduling_Parameters; use Ada.Scheduling_Parameters;
package Ada.Synchronous_Task_Control.Scheduling_Parameters is
   procedure Suspend_Until_True_And_Apply_Sched_Params (
      S : in out Suspension_Object; -- We'll later introduce a new type of suspension object
      SP: Sched_Params'Class);
end Ada.Synchronous_Task_Control.Scheduling_Parameters;
```

[1]For clarity, we are using the existing type `Suspension_Object` in listing 3. In section 4.3 we will justify why we are proposing a new type of suspension object that implements modification of scheduling parameters.

Note that SP, the Sched_Params parameter for this procedure, is class-wide. Hence it can dispatch to root-type, EDF-extended or user-extended scheduling parameters.

4 Implementation issues

The operations presented above, Apply_Sched_Params and Delay_Until_And_Apply_Sched_Params, allow the application to atomically change several task scheduling parameters. The underlying Operating System (OS) has to provide specific support in order to allow the Ada Run-Time Support to implement these operations adequately. Although the required behaviour within the operating system kernel is simple, as it will be shown bellow, this support is not present in any POSIX-like operating system (to the best of our knowledge) including those that add non-portable extensions, such as the Linux kernel.

Any change in one scheduling parameter implies that the operating system removes the implied thread from the current run queue and inserts it again in a (possibly different) run queue in a different position. As one or more system run queues are modified, the system scheduler has to be invoked to determine the new highest priority thread. Furthermore, if the CPU of a thread is changed, then some kind of *Inter-Processor Interrupt* (IPI) has to be sent to inform the affected CPU or CPUs that they have to execute the scheduler. Therefore, the change of a scheduling parameter has to be considered always a thread dispatching point.

If an application wants to change several scheduling parameters at the same time, e.g. its priority and CPU, it has to invoke several system calls to change these parameters. For example, if the underlying OS is the Linux kernel, the application has to invoke sched_setparam and sched_setaffinity system calls. Each of these system calls is a thread dispatching point, since they may imply changes in the system run queues. The scheduling artifacts presented in section 2 are due to the sequential execution of these system calls and their corresponding thread dispatching points. Each time the application changes a single scheduling parameter, the scheduler can dispatch a different thread in one or more system CPUs, and therefore, the thread that is changing the parameters can lose the CPU or the thread with the new scheduling parameter can disturb temporarily other running threads. This undesired behaviour could be avoided if the scheduling parameters would be changed atomically.

The actions the kernel of the RTOS has to perform to support the simultaneous modification of several scheduling parameters are very simple:

1. Remove the thread from the run queue where it is currently located.
2. Change all the scheduling parameters specified by the application.
3. Insert the thread in a new task queue according to the new set of scheduling parameters. This queue could be a new priority queue in a new CPU or it could be the timer or mutex queue if the thread has to be suspended.

The main implementation issue is how to offer this kernel functionality to the application. We'll now explore the POSIX case and see what extensions would be needed.

4.1 Proposed POSIX extensions

Although POSIX standard does not provide support to simultaneously changing several scheduling parameters on a running process or thread (other than scheduling policy and priority using sched_setscheduler system call), it provides a similar funtionality for establishing all the scheduling parameters for the creation of a new thread. This functionality is offered through the structure pthread_attr_t and the following C functions that allow the application to specify the full set of thread attributes before creating it with pthread_create.

Listing 4. Thread attributes manipulation functions

```
pthread_attr_init/destroy // initialize and destroy thread attributes object
pthread_attr_set/getdetachstate // set/get detach state attribute in thread attributes object
pthread_attr_set/getinheritsched // set/get inherit scheduler attribute in thread attributes object
pthread_attr_set/getschedparam // set/get scheduling parameter attributes in thread attributes object
pthread_attr_set/getschedpolicy // set/get scheduling policy attribute in thread attributes object
...                     // Other attributes not related directly with scheduling
```

In the case of the Linux kernel, a small set of non-portable extensions are also available, mainly to support the CPU affinity of a thread. However, as it will be shown bellow, Linux also provides the function `pthread_getattr_np` that allows retrieval of the current attributes of an already created thread.

Listing 5. Linux specific non portable extension to thread attributes

```
pthread_attr_set/getaffinity_np // set/get CPU affinity attribute in thread attributes object

pthread_getattr_np       // get attributes of created thread
```

We proposes to extend this API with the corresponding `pthread_setattr_np` function that would allow the application to specify a set of scheduling attributes that has to be applied simultaneously over an already created thread. However, in order to support the operations that imply a possible suspension of the thread, i.e. `Delay_Until_And_Apply_Sched_Params`, it is required that the new API offers the possibility of deferring the setting of the attributes until the thread becomes suspended. Two different alternatives are proposed, similar to the ones presented in [**?**].

Listing 6. Proposed POSIX extensions

```
 #include <pthread.h>

int pthread_setattr_np(pthread_t thread, pthread_attr_t *attr);
int pthread_setattr_on_suspend_np(pthread_t thread, pthread_attr_t *attr);

// or alternatively ...

#define SCHED_SET_IMMEDIATE 1
#define SCHED_SET_ON_SUSPEND 2

int pthread_setattr_np(pthread_t thread, pthread_attr_t *attr, const long flag);
```

Next section will use the first approach to show how the Ada Run-Time Support could implement the main operations of the new `Sched_Params` type.

4.2 Implementation of `Sched_Params` operations

Based on these new OS functionalities and taking the source code GNAT GPL 2012 as a reference, the implementation of the new proposed operations could be as follows:

```
  procedure Apply_Sched_Params
    (SP:          : Sched_Params;
     T_Id         : Task_Id: Current_Task)
  is
    Attributes : aliased pthread_attr_t;
    Result   : Interfaces.C.int;

  begin
     -- Retrieve the current thread attributes
     Get_Task_Attributes (Attributes'Access, T_Id);

     -- Modify the task attributes
     Set_Attr_Priority(Attributes'Access, SP.Prio);
     Set_Attr_CPU(Attributes'Access, SP.CPU_Nr);

     -- Set the new thread attributes immediately
     Result := pthread_setattr_np (T_Id.Common.LL.Thread, Attributes'Access);
     pragma Assert (Result = 0);
  end Apply_Sched_Params;
```

```
procedure Delay_Until_And_Apply_Sched_Params
  (SP:          : Sched_Params;
   Delay_Until_Time : Ada.Real_Time.Time;
   T_Id         : Task_Id: Current_Task)
is
  Attributes : aliased pthread_attr_t;
  Result   : Interfaces.C.int;

begin
   -- Retrieve the current thread attributes
   Get_Task_Attributes (Attributes'Access, T_Id);

   -- Modify the task attributes
   Set_Attr_Priority(Attributes'Access, SP.Prio);
   Set_Attr_CPU(Attributes'Access, SP.CPU_Nr);

   -- Take note of the new thread attributes to be applied upon thread suspension
   Result := pthread_setattr_on_suspend_np (T_Id.Common.LL.Thread, Attributes'Access);
   pragma Assert (Result = 0);

   delay until Delay_Until_Time; -- New scheduling attributes take effect on wake up

end Delay_Until_And_Apply_Sched_Params;
```

In order to simplify the implementation, it is supposed that the Ada run-time system will provide procedures to retrieve and manipulate the `Attributes` type. In the current GNAT GPL 2012, the `Attributes` type is an opaque type that is manipulated using the POSIX interface only. The procedures used above (i.e. `Get_Task_Attributes`, `Set_Attr_Priority` and `Set_Attr_CPU`), will use these existing POSIX functions and the thread information maintained by the Ada run-time system, to prepare the `Attributes` object. This object represents the thread scheduling parameters at operating system level.

4.3 Implementation of Suspension Objects

The implementation of `Suspend_Until_True_And_Apply_Sched_Params` needs a different approach to `Delay_Until_And_Set_Sched_Params`. In this second case, it is clear in advance when the task will be awakened (at the specified absolute time) and hence have its new parameters applied. But in the case of using a suspension object, the task calling the suspension operation may either go through immediately (if the object's state is True) or it may have to wait for someone to set the object state to True. In the first case, the attributes need be changed as part of the call to the suspension operation; whereas in the second case, it is the call to `Set_True` that has the effect of enforcing the new scheduling parameters. So we need to store the task identification and new scheduling parameters to apply them at the proper time.

We therefore propose a new type of suspension object (`Suspension_Object_With_Sched_Params`) for sporadic tasks that use deferred setting of scheduling attributes. This new type contains, as part of its internal state, two fields to store the scheduling parameters (`SP`) and the task identification (`T_Id`).

```
type Suspension_Object_With_Sched_Params is record
   State : Boolean;
   pragma Atomic (State);
   -- Boolean that indicates whether the object is open.

   Waiting : Boolean;
   -- Flag showing if there is a task already suspended on this object

   L : aliased System.OS_Interface.pthread_mutex_t;
   -- Protection for ensuring mutual exclusion on the Suspension_Object
```

```
    CV : aliased System.OS_Interface.pthread_cond_t;
    -- Condition variable used to queue threads until condition is signaled

    T_Id : Task_Id;
    -- Task suspended within the Suspension Object

    SP : access all Sched_Params'Class;
    -- Scheduling Parameters to be applied to the suspended task
  end record;
```

When a sporadic task invokes `Suspend_Until_True_And_Apply_Sched_Params` with a new set of scheduling parameters, if the suspension object state is true, the scheduling parameters are applied immediately within the suspension object. Then the sporadic task continues with its *next activation* using the new scheduling parameters.

If the suspension object state is false, the sporadic task will be suspended until the state becomes true. In this case, `Suspend_Until_True` stores the task identifier of the sporadic task and scheduling parameters for its next activation. The task that invokes the `Set_True` procedure will apply the new scheduling parameters to the sporadic task before signaling the conditional variable within the suspension object, and therefore, before waking up the sporadic task. When the sporadic task wakes up, it already has its new scheduling parameters.

The `Suspension_Object_With_Sched_Params` type will also offer the `Suspend_Until_True` operation, that allows a task to be suspended until the suspension object state becomes true, but without modifying its scheduling parameters.

Based on the source code from GNAT GPL 2012, the new suspension object operations could be implemented as follows:

```
  procedure Suspend_Until_True_And_Apply_Sched_Params
     (S : in out Suspension_Object_With_Sched_Params;
      SP : access all Sched_Params'Class) is

     Result : Interfaces.C.int;
  begin
     SSL.Abort_Defer.all;

     Result := pthread_mutex_lock (S.L'Access);
     pragma Assert (Result = 0);

     if S.Waiting then
        Result := pthread_mutex_unlock (S.L'Access);
        pragma Assert (Result = 0);

        SSL.Abort_Undefer.all;

        raise Program_Error;
     else
        if S.State then
           S.State := False;
           S.Apply_Sched_Params;
        else
           S.Waiting := True;
           S.T_Id := Current_Task;
           S.SP := SP;
           loop
              Result := pthread_cond_wait (S.CV'Access, S.L'Access);
              pragma Assert (Result = 0 or else Result = EINTR);

              exit when not S.Waiting;
           end loop;
        end if;
```

```
      Result := pthread_mutex_unlock (S.L'Access);
      pragma Assert (Result = 0);

      SSL.Abort_Undefer.all;
    end if;
  end Suspend_Until_True;

  procedure Set_True
    (S : in out Suspension_Object_With_Sched_Params) is
    Result : Interfaces.C.int;

  begin
    SSL.Abort_Defer.all;

    Result := pthread_mutex_lock (S.L'Access);
    pragma Assert (Result = 0);

    if S.Waiting then
      S.Waiting := False;
      S.State := False;

      if S.SP /= null then
        S.SP.Apply_Sched_Params(S.T_Id);
      end if;

      Result := pthread_cond_signal (S.CV'Access);
      pragma Assert (Result = 0);

    else
      S.State := True;
    end if;

    Result := pthread_mutex_unlock (S.L'Access);
    pragma Assert (Result = 0);

    SSL.Abort_Undefer.all;
  end Set_True;
```

5 Use example

This section shows a brief example where this new funcionality is used to implement a task subject to job partitioning. With this scheduling scheme, a periodic task could decide to use a different CPU and priority for each job (i.e., each activation of the task). In the example below, this design decision is represented by a cyclic plan of *scheduling parameters*, `Params_List`. At the end of each job execution, the task retrieves the next set of scheduling parameters from its plan, and calls `Delay_Until_And_Apply_Sched_Params`. This allows the task to change the scheduling parameters for its next job atomically and avoids the scheduling artifacts mentioned in section 2.

Listing 7. Periodic task with job partitioning based on `delay until`

```
task body Periodic_With_Job_Partitioning is
  type List_Range is mod N;
  Params_List : array (List_Range) of Sched_Params := (...); -- Decided at design time
  Params_Iter : List_Range := List_Range'First;
  Next_Params : Sched_Params;
  Next_Release : Ada.Real_Time.Time;
  Period    : Time_Span := ...;
```

```
begin
   Task_Initialize;

   -- First job parameters
   Next_Release := Ada.Real_Time.Clock;
   Next_Params := Param_List(Param_Iter);
   Next_Params.Apply_Sched_Params(); -- Scheduling parameters for the first activation
   loop
      Task_Main_Loop;

      -- Next job preparation
      Params_Iter := Params_Iter'Succ;
      Next_Params := Params_List(Params_Iter);
      Next_Release := Next_Release + Period;

      -- Suspends the task until the next job activation
      Delay_Until_And_Apply_Sched_Params(Next_Params, Next_Release);
      -- Next job will wake up with the next scheduling parameters applied
   end loop;
end Periodic_With_Job_Partitioning;
```

6 Conclusion

The deferred, atomic setting of a set of scheduling attributes is a useful feature that is currently absent in Ada. It provides a clear semantics and avoids scheduling artifacts and wrong effects derived from sequentially applying one attribute after another, specially when the underlying hardware is a multiprocessor platform. In this paper we have proposed changes in the direction of including this feature in Ada.

All changes proposed are additions to the standard library, with no modification proposed to any other part of the language. The changes are also user-extensible since they are based on the use of tagged types. Perhaps the major change proposed is a new type of suspension object to give support to deferred, atomic setting of attributes of sporadic tasks. The fact that a sporadic task may have its parameters changed either immediately upon calling `Suspend_Until_True` (when the object's state is True) or deferredly when another task calls `Set_True` (in case the object state was False when the task called `Suspend_Until_True`) makes it necessary to provide a different type of suspension object, augmented with the capability of setting the waiting task's parameters.

These proposed extensions are mainly directed towards multiprocessor platforms, since the intended semantics is feasibly implementable on single-processors in Ada. However, some single-processor scheduling approaches could benefit from the changes proposed here, if only aesthetically (e.g., dual-priority scheduling, existing schemes for control tasks structured as Initial-Mandatory-Optional-Final, etc).

We want to finally note a gracious side effect of this proposal. With the proposed set of procedures, there would be strictly no need to use the existing procedure `Delay_Until_And_Set_CPU` from package `System.Multiprocessors_Dispatching_Domains`. The nice effect is that, if that procedure did not exist, then there would be no dependence with `Ada.Real_Time`, and therefore `System.Multiprocessors_Dispatching_Domains` could be preelaborable. But, unfortunately, changing the standard for this reason would introduce backward incompatibility.

References