Sáez Barona, S.; Crespo, A. (2013). Deferred setting of scheduling attributes in Ada 2012. Ada Letters. 33(1):93-100. doi:10.1145/2492312.2492322.

Deferred Setting of Scheduling Attributes in Ada 2012 *

Sergio Sáez, Alfons Crespo
Grupo de Informática Industrial y Sistemas de Tiempo Real
Universidad Politécnica de Valencia
{ssaez,acrespo}@disca.upv.es

Abstract

Some scheduling techniques, specially in multiprocessor systems, require changing several task attributes atomically to avoid scheduling errors and artifacts. This work proposes to incorporate the deferred attribute setting *mechanism to cope with this problem in the next Ada 2012.*

1 Introduction

A real-time system is usually composed by a set of concurrent task that collaborate to achieve a common goal. A real-time task has to produce its result within a temporal interval in order to be a valid result. To enforce this behaviour, the real-time system designer assigns a given priority to each system task. These priorities are used by the underlying real-time operating system (RTOS) to ensure a timeliness execution of the whole system.

In uniprocessor real-time systems using fixed priorities, the priority of a task tends to remain *fixed* during its entire lifespan[1]. However, in uniprocessor systems using dynamic priorities and in several scheduling approaches used in multiprocessor systems, the task has to update its scheduling attributes during the execution of the task's code. When several attributes have to be changed simultaneously some scheduling artifacts can arise if they are not updated atomically, giving rise to erroneous schedules.

In order to correctly implement these real-time systems with dynamic attribute setting, the underlying run-time system has to offer some support for atomically changing multiple attributes. This work proposes a flexible and scalable approach to incorporate this support to the Ada 2012 Run-Time System (RTS).

The rest of the paper is organised as follows: next section presents dynamic attribute changing in uniprocessor systems. Section 3 deals with multiprocessor scheduling and simultaneous setting of task attributes. Then, section 4 presents the proposed approach to solve the presented issues. Section 5 proposes to add CPU affinities to Timing_Events with deferred setting support. Finally, section 6 shows some conclusions.

2 Scheduling Attributes in Uniprocessor Systems

In Ada 2005 the priority of a task is represented by the task attribute Priority and, when the EDF_Across_Priorities policy is specified, by an absolute deadline. The Ada Run-Time System uses these *scheduling attributes* to choose which task or tasks have to be executed at a given instant. The initial value of these scheduling attributes can be specified within the task definition by means of the following pragmas:

```
pragma Priority ( expression );                          -- See RM D.1
pragma Relative_Deadline ( relative_deadline_expression ); -- See RM D.2.6
```

*This work was partially supported by the Vicerectorado de Investigación of the Universidad Politécnica de Valencia under grant PAID-06-10-2397 and European Project OVERSEE (ICT-2009 248333)

[1] Despite of priority changes due to priority inheritance protocols

Although on a uniprocessor system the priority of a task is not usually changed by the task itself, a periodic real-time task scheduled under the EDF policy has to change its absolute deadline on each activation. Ada 2005 offers two procedures to change the priority and the absolute deadline of a task that are shown bellow:

```
package Ada.Dynamic_Priorities is
   ...
   procedure Set_Priority ( Priority  : in System.Any_Priority;
                            T : in Ada.Task_Identification.Task_Id := Ada.Task_Identification.Current_Task );
end Ada.Dynamic_Priorities;
```

```
package Ada.Dispatching.EDF is
   ...
   procedure Set_Deadline (D : in Deadline;
                           T : in Ada.Task_Identification.Task_Id := Ada.Task_Identification.Current_Task );
end Ada.Dispatching.EDF;
```

However, as both procedures are dispatching points, when a task calls the Set_Deadline procedure to update its absolute deadline before getting suspended until its next release, it could be preempted by a task with a closer deadline, causing the scheduling artifacts shown in the Figure 1. It can be observed that when task `T0` changes its deadline to the next absolute deadline, the task `T1` becomes more urgent and does not allow task `T0` to execute its **delay until** statement until the task `T1` executes its own Set_Deadline procedure.

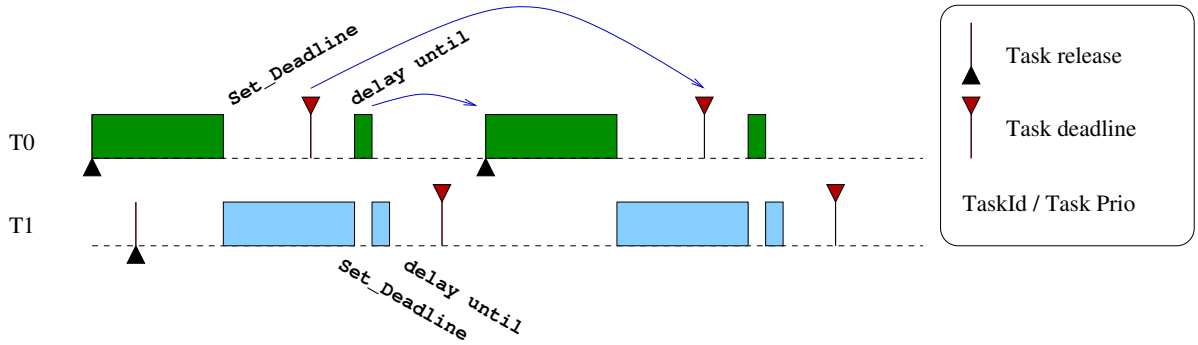

Figure 1. Scheduling artifact during Set_Deadline + delay until code sequence.

Although this anomalous behaviour does not cause a great damage in the scenario shown by Figure 1 (the only effect is that task `T0` executes its **delay until** after its deadline), Ada 2005 provides an additional procedure Delay_Until_And_Set_Deadline to avoid these artifacts. Its behaviour is defined as follows:

> The procedure Delay_Until_And_Set_Deadline delays the calling task until time Delay_Until_Time. When the task becomes runnable again it will have deadline Delay_Until_Time + Deadline_Offset. (RM D.2.6 15/2)

Using this procedure, the main loop of a periodic task scheduled under EDF policy will be as follows:

```
loop
   -- Task code
   -- Preparation code with scheduling artifacts
   Next_Time := Next_Time + Period;
   Set_Deadline(Next_Time + Relative_Deadline );
   delay until Next_Time;
end loop;
```

⇒

```
loop
   -- Task code
    ...
   -- It performs atomically Set_Deadline + delay until
   Next_Time := Next_Time + Period;
   Delay_Until_And_Set_Deadline(Next_Time, Relative_Deadline );
end loop;
```

3 Scheduling Attributes in Multiprocessor Systems

Real-Time and embedded systems are becoming more complex, and multiprocessor/multicore systems are becoming a common execution platform in these areas. In order to achieve a predictable schedule of a set of real-time tasks in a multiprocessor platform several approaches can be applied. Based on the capability of a task to migrate from one processor to another, the scheduling approach can be:

Global scheduling: All tasks can be executed on any processor and after a preemption the current job can be resumed in a different processor.

If the scheduling decisions are performed on-line, in a multiprocessor platform with M CPUs, the M active jobs with the highest priorities are the ones selected for execution. If the scheduling decisions are computed off-line, releases times, preemption instants and processors where tasks have to be executed are stored in a static scheduling plan.

Job partitioning: Each job activation of a given task can be executed on a different processor, but a given job cannot migrate during its execution.

The processor where each job is executed can be decided by an on-line global dispatcher upon the job activation, or it can be determined off-line by a scheduling analysis tool and stored in a processor plan for each task. The job execution order on each processor is determined on-line by its own scheduler using the scheduling attributes of each job.

Task partitioning: All job activations of a given task have to be executed in the same processor. No job migration is allowed.

The processor where a task is executed is part of the task's scheduling attributes. As in the previous approach, the order in which each job is executed on each processor is determined on-line by the scheduler of that processor.

In addition to these basic approaches, new techniques that mix task partitioning with task that migrate from one processor to another at specified times are already available in the literature. In this approach, known as *task splitting*, some works suggest to perform the processor migration of the split task at a given time after each job release [1] or when the job has performed a certain amount of execution [2]. It is worth noting that this approach normally requires the information about the processor migration instant to be somehow coded into the task behaviour.

To apply some of these scheduling approaches some specific support at kernel and user-space level is needed, e.g. system and CPU clock timers and dynamic scheduling attributes.

The forthcoming release of Ada 2012 is expected to offer explicit support for multiprocessor platforms through a comprehensive set of programming mechanisms shown in Listing 1 [3, 4]. Although these mechanisms have been shown adequate to apply task and job partitioning, and task splitting techniques [5, 6], they will suffer the same kind of artifact shown in Figure 1. Next subsection shows why the current proposal found in [3, 4] is still inadequate for real-time multiprocessor systems.

Listing 1. Ada 2012 facilities to set the target CPU

```
package System.Multiprocessors.Dispatching_Domains is
   ...
   type Dispatching_Domain (<>) is limited private;
   ...
   procedure Assign_Task(Domain : in out Dispatching_Domain;
                         CPU : in CPU_Range := Not_A_Specific_CPU;
                         T : in Task_Id := Current_Task);
   procedure Set_CPU(CPU : in CPU_Range; T : in Task_Id := Current_Task);
   function Get_CPU(T : in Task_Id := Current_Task) return CPU_Range;
   procedure Delay_Until_And_Set_CPU(Delay_Until_Time : in Ada.Real_Time.Time;
                                     CPU : in CPU_Range);
end System.Multiprocessors.Dispatching_Domains;
```

3.1 Multiprocessor scheduling requirements

Some multiprocessor scheduling approaches require to specify the target CPU for each real-time task. Additionally, in job partitioning and task splitting techniques the target CPU changes during the task lifespan. As each processor can have a

different set of tasks or to use an EDF scheduling policy, moving a task from one CPU to another could also require to change its priority or its deadline. If all these task attributes are not changed atomically, some scheduling artifacts could arise giving rise to incorrect schedules.

Figures 2 and 3 shown how a task can miss its deadline trying to change simultaneously its priority and its target CPU. Both scenarios try to change task `T0` from one CPU to another, but using a different priority into the new CPU. In Figure 2 the task `T0` losses its deadline while executing the Set_Priority + Set_CPU sequence. After `T0` changes its priority, the task `T1` has a greater priority and avoids the task `T0` to complete the Set_CPU statement until it is too late. Figure 3 shows a different scenario where the incorrect sequence is Set_CPU + Set_Priority . The only solution to these situations is to provide a mechanism to simultaneously change the priority and the target CPU. Similar requirements are needed when using dynamic priorities.

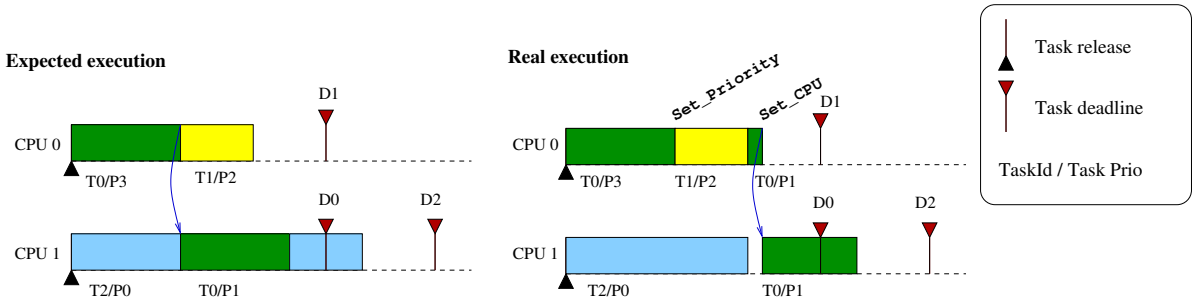

Figure 2. Expected and real execution for Set_Priority + Set_CPU code sequence.

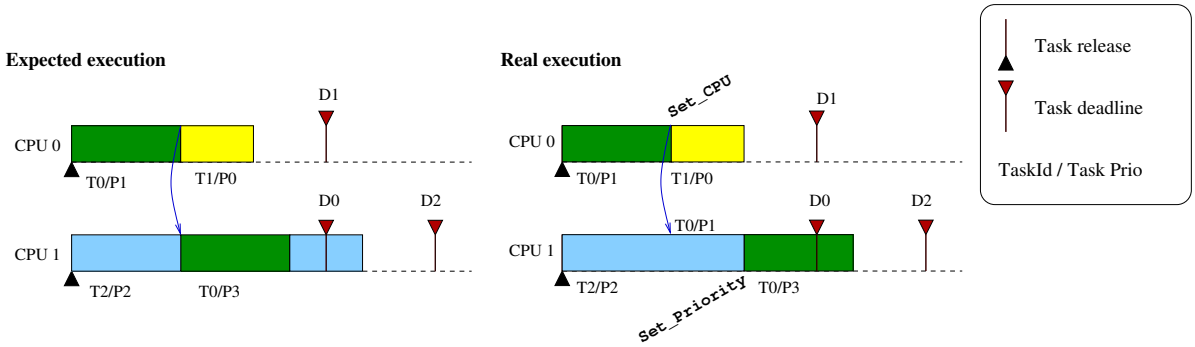

Figure 3. Expected and real execution for Set_CPU + Set_Priority code sequence.

Although the scenarios shown in Figures 2 and 3 can be solved by encapsulating both Set_CPU and Set_Priority inside a protected operation, this cannot be performed when the change of priority/deadline and target CPU is combined with a **delay until** statement. As shown in the following examples, no correct sequence of code is valid using the current multiprocessor support proposal.

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

These sequences of code are common in job partitioning schemes, in order to set the CPU where the next job is going to

be executed before the current job finishes, and in task splitting techniques, in order to reset the original CPU at the end of the job after one or more *splits*.

Next section presents a proposal to cope with these kind of scenarios using deferred attributes. This proposal was already briefly introduced in [6].

4 Deferred Attribute Setting

The Ada 2005 Reference Manual specifies the behaviour for a task that dynamically changes its priority or its deadline:

> On a system with a single processor, the setting of the base priority of a task T to the new value occurs immediately at the first point when T is outside the execution of a protected action. (RM D.5.1 10/2)
>
> On a multiprocessor, the implementation shall document any conditions that cause the completion of the setting of the priority of a task to be delayed later than what is specified for a single processor. (RM D.5.1 12.1/2)
>
> On a system with a single processor, the setting of the deadline of a task to the new value occurs immediately at the first point that is outside the execution of a protected action. If the task is currently on a ready queue it is removed and re-entered on to the ready queue determined by the rules defined below. (RM 2.6 16/2)

However, although the current Ada Issue AI05-167/11 [4] does not specify anything about protected objects, it seems reasonable to apply similar restrictions to the setting of the CPU. The text could be as follows:

> On a system with multiple processors, the setting of the target CPU of a task T to the new value occurs immediately at the first point when T is outside the execution of a protected action.

In order to follow the above mentioned restrictions, if a task invokes a procedure to change its attributes within a protected action, the Ada RTS has to defer the priority/deadline/CPU change until the task is outside the execution of the protected action. This deferred setting of task attributes can be used to solve the scheduling artifacts shown in the previous section.

This work proposes to add explicit support to perform a deferred setting of task attributes adding the following procedures:

```
package Ada.Dynamic_Priorities is
   ...
   -- Programs a deferred setting of the base priority
   procedure Set_Next_Priority ( Priority : in System.Any_Priority; T : in
   Task_Id := Current_Task );
   ...
end Ada.Dynamic_Priorities;
```

```
package Ada.Dispatching.EDF is
   ...
   -- Programs a deferred setting of the absolute deadline
   procedure Set_Next_Deadline (D : in Deadline; T : in Task_Id := Current_Task );
   ...
end Ada.Dispatching.EDF;
```

```
package System.Multiprocessors.Dispatching_Domains is
   ...
   -- Programs a deferred setting of the target CPU
   procedure Set_Next_CPU(CPU : in CPU_Range; T : in Task_Id := Current_Task );
   ...
end System.Multiprocessors.Dispatching_Domains;
```

The semantic of these procedures can sketched as:

> The deferred setting of a task attribute will delay the effective attribute setting until the next dispatching point. If the task T is inside the execution of a protected action, the setting of the new value occurs immediately at the first point when T is outside the execution of the protected action.

With the introduction of these new procedures, the erroneous code presented in section 3.1 can be rewritten as follows:

```
loop
   -- Task code
    ...
   Next_Time := Next_Time + Period;
   Set_Deadline(Next_Time + Relative_Deadline );
   Delay_Until_And_Set_CPU(Next_Time, Next_CPU);
end loop;
```

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

⇒

```
loop
   -- Task code
    ...
   Next_Time := Next_Time + Period;
   Set_Next_Deadline(Next_Time + Relative_Deadline );
   Set_Next_CPU(Next_CPU)
   delay until Next_Time;
end loop;
```

As **delay until** statement is a dispatching point, the deferred attributes set by Set_Next_Deadline and Set_Next_CPU take effect at that point. Also the code described in the scenarios of Figures 2 and 3 can be easily solved with the new procedures, as shown in the right side of the next listings.

```
loop
   -- Task code
    ...
   Set_Priority ( Next_Priority );
   Set_CPU(Next_CPU);
    ...
end loop;
```

```
loop
   -- Task code
    ...
   Set_CPU(Next_CPU);
   Set_Priority ( Next_Priority );
    ...
end loop;
```

⇒

```
loop
   -- Task code
    ...
   Set_Next_Priority ( Next_Priority );
   Set_CPU(Next_CPU);
    ...
end loop;
```

```
loop
   -- Task code
    ...
   Set_Next_CPU(Next_CPU);
   Set_Priority ( Next_Priority );
    ...
end loop;
```

The proposed procedures give rise to a more orthogonal and cleaner task code. It can also be extended to cover other existing task attributes or new proposed ones, if required. On the other side, the procedure Delay_Until_And_Set_CPU in package System.Multiprocessors.Dispatching_Domains will become unnecessary and the procedure Delay_Until_And_Set_Deadline deprecated.

5 CPU affinity of `Timing_Events`

Another feature of Ada 2005 highly required to support some of the multiprocessor scheduling approaches are the Timing_Events. They allow a task to execute protected actions at specific time instants, e,g. to program future changes to its task attributes, required to perform a task split.

However, when Timing_Events are used to wake up a real-time task, it should be taken into account that the processor where the task has to be executed can be different from the one used to program the TimingEvent. This scenario can introduce a unnecessary scheduling overhead: a TimingEvent produces a clock interrupt in a processor `P1`, this processor executes the TimingEvent handler and wakes up a task that has to be executed in a different processor `P2`; after the handler execution, the processor `P1` has to send an Inter-Processor Interrupt (IPI) to force the scheduler execution in processor `P2`. It will be

clearly more efficient, if the TimingEvent handler can be programmed to be executed directly in processor P2. However, implementation details have to be carefully studied in the case of using multiple timer queues. In this case, if the timer queue of the target processor P2 is empty, the above scenario may require also an IPI to program the clock interrupt in P2.

This work proposes a modification of the package Ada.Real_Time.Timing_Events to add the following procedures:

```
package Ada.Real_Time.Timing_Events is
   type Timing_Event is tagged limited private;
    ...
   -- Set the CPU where the current handler has to be executed
   procedure Set_CPU(Event : in out Timing_Event;
                     CPU : in CPU_Range);
   -- Set the CPU where the next handler established with Set_Handler has to be executed
   procedure Set_Next_CPU(Event : in out Timing_Event;
                          CPU : in CPU_Range);
    ...
end Ada.Real_Time.Timing_Events;
```

The procedure Set_CPU allows the programmer to specify where the handler that is already set has to be executed. It allows changing the target CPU of a programmed TimingEvent if the implicated task migrates from one CPU to another [2]. The procedure Set_Next_CPU establishes the target CPU to be used for the next handler when the procedure Set_Handler was used. If no handler is set, both procedures behave identical.

Depending on the available clock interrupt hardware, the Ada RTS can implement one or multiple queues to store the active Timing_Events. In the case of a global Timing_Events queue, the CPU information can be used to configure the interrupt controller and set the clock interrupt affinity to the target CPU of the closer event. In the case of a per-CPU clock interrupt hardware, multiple Timing_Events queues can be used. This could require to move Timing_Events from one to another when the Set_CPU is used. In both cases, the invocation of the procedure Set_CPU can require a reconfiguration of the clock interrupt hardware, while the invocation of the Set_Next_CPU is only used to configure the next event or to store it in a processor specific events queue.

6 Conclusions

A small set of modifications for the next Ada 2012 have been proposed to allow simultaneous setting of multiple task attributes. This behaviour is required to adequately support some of the multiprocessor scheduling approaches. The proposed solution is based on the introduction of *deferred attributes*.

The setting of deferred attributes allows the application to specify a set changes of task attributes that a task will apply in the next dispatching point. This gives rise to a simpler and scalable interface for simultaneously changing multiple task attributes. A similar interface has been proposed for the Timing_Events mechanism, allowing the Ada RTS to implement it efficiently in multiprocessor platforms.

References

[1] K. Lakshmanan, R. Rajkumar, and J. P. Lehoczky, "Partitioned fixed-priority preemptive scheduling for multi-core processors," in *21st Euromicro Conference on Real-Time Systems, ECRTS 2009*, pp. 239–248, IEEE Computer Society, 2009.

[2] S. Kato, N. Yamasaki, and Y. Ishikawa, "Semi-partitioned scheduling of sporadic task systems on multiprocessors," in *21st Euromicro Conference on Real-Time Systems, ECRTS 2009*, (Los Alamitos, CA, USA), pp. 249–258, IEEE Computer Society, 2009.

[3] A. Burns and A. J. Wellings, "Dispatching domains for multiprocessor platforms and their representation in Ada," in *15th International Conference on Reliable Software Technologies - Ada-Europe*, pp. 41–53, 2010.

[2] This action can produce a considerable overhead and its use must be carefully studied

[4] Ada 2005 Issues. AI05-0167-1/11, *Managing affinities for programs executing on multiprocessors*, 2011. Version: 1.17. Status: Amendment 2012.

[5] B. Andersson and L. M. Pinho, "Implementing multicore real-time scheduling algorithms based on task splitting using Ada 2012," in *15th International Conference on Reliable Software Technologies - Ada-Europe*, pp. 54–67, 2010.

[6] S. Sáez and A. Crespo, "Preliminary multiprocessor support of Ada 2012 in GNU/Linux systems," in *15th International Conference on Reliable Software Technologies - Ada-Europe*, pp. 68–82, Springer, 2010.