Generalizing the EDF Scheduling Support in Ada 2005

A.J Wellings and A. Burns
Department of Computer Science, University of York
Heslington, York YO10 5DD, UK
(andy,burns)@cs.york.ac.uk

Abstract

Ada 2005 has introduced a version of Baker's Stack Resource Protocol called the Preemption Level Control Protocol in order to support EDF scheduling. Baker's protocol is, however, a general protocol that can be used with a range of scheduling algorithms, for example any static job-level algorithm such as static values or a modified least laxity algorithm. This paper investigates the changes that would be needed to Ada 2005 to allow it to support user-defined scheduling, where the user can implement any scheduling approach that is compatible with Baker's algorithm.

1 Introduction

Ada 2005 has added new dispatching mechanisms in order to allow Round Robin and EDF scheduling. To facilitate the latter, it has also added support for the Preemption Level Control Protocol (PLCP). The PLCP is a very general resource control protocol that is capable of supporting other scheduling approaches. This paper, therefore investigates what changes would be need to Ada 2005 to allow a wider class of scheduling algorithms to be supported within the Ada preemptive priority based approach.

We start by describing, in some detail, the Ada 2005 model[1], explaining how EDF scheduling has been introduced. This allows us to extract out the core areas that will need to be generalized. We then present the results of our study by defining the changes that would need to be made to the language to generalize the EDF support.

2 Hierarchical Scheduling in Ada 2005

Ada 2005 essentially supports two-level scheduling. At the outer level is the standard preemptive fixed priority based scheduling. Tasks with the same priority can either be scheduled in a FIFO or round-robin fashion. More flexibility is allowed by grouping priority levels into bands and supporting EDF scheduling across each band.

To support EDF dispatching, three language features are required:

- a formal representation of the deadline of a task,
- use of deadlines to control dispatching and
- a means of sharing data via protected objects that is compatible with EDF dispatching.

Each of these issues is considered in turn. However, it should be noted that Ada's support for deadline-based scheduling does not extend to any general notion of deadline inheritance (for example during a rendezvous).

[1] This material is based on that given in Burns and Wellings [2].

2.1 Representing deadlines

The EDF scheme uses absolute deadline – it requires that the task with the shortest (earliest) absolute deadline is the next to be executed. A direct representation is provided in Ada 2005:

```
with Ada.Real_Time;
with Ada.Task_Identification;
use Ada.Real_Time;
use Ada.Task_Identification;
package Ada.Dispatching.EDF is
  subtype Deadline is Time;
  Default_Deadline : constant Deadline :=
    Time_Last;
  procedure Set_Deadline(
    D : in Deadline;
    T : in Task_ID := Current_Task);
  procedure Delay_Until_And_Set_Deadline(
    Delay_Until_Time : in Time;
    TS : in Time_Span);
  function Get_Deadline(T : in Task_ID :=
    Current_Task) return Deadline;
end Ada.Dispatching.EDF;
```

With EDF, all dispatching decisions are based on deadlines, and hence it is necessary for a task to always have a deadline. However, a task must progress through activation before it can get to a position to call `Set_Deadline` and hence a default deadline value is given to all tasks (`Default_Deadline` defined in `Ada.Dispatching.EDF`). But this default value is well into the future, and hence activation will take place with very low urgency (all other task executions will occur before this task's activation). If more urgency is required, the following language-defined pragma is available for inclusion in a task's specification (only):

```
pragma Relative_Deadline(Relative_Time);
```

where the type of the parameter is `Ada.Real_Time.Time_Span`. The initial absolute deadline of a task containing this pragma is the value of `Ada.Real_Time.Clock + Relative_Time`. The call of the clock being made at sometime between task creation and the start of its activation. The actual timing of the call is implementation defined.

A final point to note with the deadline assignment routine concerns when it will take effect. The usual result of a call to `Set_Deadline` is for the associated deadline to be increased into the future and that a task switch is then likely (if EDF dispatching is in force). As this would not be appropriate if the task is currently executing within a protected object, *the setting of a task's deadline to the new value takes place as soon as is practical but not while the task is performing a protected action*.

2.2 Dispatching

To request EDF dispatching, the following use of the policy pragma is supported:

```
pragma Task_Dispatching_Policy(
       EDF_Across_Priorities);
```

The main result of employing this policy is that the ready queues are now ordered according to deadline (not FIFO). Each ready queue is still associated with a single priority but at the head of any ready queue is the runnable task with the earliest deadline. Whenever a task is added to a ready queue for its priority, it is placed in the position dictated by its current absolute deadline.

The active priority of a task, under EDF dispatching, *is no longer directly linked to the base priority of the task*. The rules for computing the active priority of a task are somewhat complex and are covered in the next subsection – they are derived from consideration of each task's use of protected objects. For a simple program with no protected objects, the following straightforward rules apply:

- any priorities set by the tasks are ignored;
- all tasks are always placed on the ready queue for the first priority.

Hence only one ready queue is used (and that queue is ordered by deadline).

Of course real programs require task interactions, and for real-time programs this usually means the use of protected objects. To complete the definition of the EDF dispatching policy, the rules for using protected objects must be considered in some detail.

2.3 EDF dispatching and the priority ceiling protocol

Ada supports immediate priority ceiling inheritance using the locking policy `Ceiling_Locking`. Ada 2005 has defined `EDF_Across_Priorities` to also work with `Ceiling_Locking`.

When fixed priorities and ceiling priorities are employed together, the notion of 'priority' is used to control dispatching and access to protected objects. Baker[1] showed that it is possible to abstract away from this particular model with a single use of 'priority' to one that has two measures: *urgency* and *preemption level*. He showed that if preemption levels are related to urgency in a specific way then the concurrent tasks can access shared objects (protected objects) in a way that has the important (ceiling protocol) properties, namely

- a task only suffers at most a single block per invocation from a lower priority task,
- deadlocks are prevented and
- mutual exclusion is provided by the protocol itself (on a single processor system).

This protocol is called the Preemption Level Control Protocol (PLCP) in Ada 2005, and is supported by using

- deadline to represent urgency,
- *base priority to represent each task's preemption level*,
- ceiling priorities to represent preemption levels for protected objects and
- standard `Ceiling_Locking` for access to protected objects.

So each task has a base priority but this is not used directly to control dispatching, EDF controls dispatching.

Before explaining in detail the dispatching policy, the following illustrates a straightforward application of the PCLP. Assume an application consists of five tasks, T1, ..., T5, and three protected objects, P1, P2, P3. The tasks are all periodic with periods given in Table 1 and deadlines equal to their associated periods (i.e. every task must always complete before it should next be released). The optimal way to assign preemption levels is *deadline monotonic* – so the shorter the relative deadline, the higher the preemption level. This is the identical (optimal) method of assigning priorities in a fixed priority scheme. Table 1 includes the preemption levels (PLs) of the tasks and the ceiling preemption levels for the protected objects. To execute this program with EDF dispatching requires the use of 'priority' to assign preemption levels to each task and protected object, and then the usual behaviour when a task calls a protected object – it will run with the ceiling value whilst inside the object. As its initial priority is `Priority'First` (because of EDF rules, and assumed to be 1 in this example) then the preemption levels are the same as priorities in terms of the behaviour required by `Ceiling_Locking`.

To return to the dispatching algorithm. The PCLP states that a newly released task, T1 say, preempts the currently running task, T2 say, if and only if:

- the deadline of T1 is earlier than the deadline of T2, and
- the preemption level of T1 is higher than the preemption level of any locked protected object (i.e. protected objects that are currently in use by any task in the system).

To keep track of all locked protected objects is an implementation overhead and hence the rules defined for Ada 2005 have the following form. Remember that if `EDF_Across_Priorities` is defined then all ready queues within the range `Priority'First .. Priority'Last` are ordered by deadline. Now rather than always place tasks in the queue for `Priority'First` the following rules apply:

Entity	Period	Usage	Task PL	Ceiling PL
T1	10	P1	6	
T2	12	P2	5	
T3	15	P3	4	
T4	20	P1, P3	3	
T5	30	P2	2	
P1		T1, T4		6
P2		T2, T5		5
P3		T3, T4		4

Table 1: An example task set

- Whenever a task T is added to a ready queue, other than when it is preempted, it is placed on the ready queue with the highest priority R, if one exists, such that:
  - another task, S, is executing within a protected object with ceiling priority R; and
  - task T has an earlier deadline than task S; and
  - the base priority of task T (its preemption level) is greater than R.

  If no such ready queue exists, the task is added to the ready queue for `Priority'First`.
- When a task is chosen for execution, it runs with the active priority of the ready queue from which the task was taken. If it inherits a higher active priority, it will return to its original active priority when it no longer inherits the higher level.

It follows that if no protected objects are in use at the time of the release of T then T will be placed in the ready queue at level `Priority'First` at the position dictated by its deadline.

A task dispatching point occurs for the currently running task T whenever:

- a change to the deadline of T takes effect; or
- a decrease to the deadline of any task on a ready queue for that processor takes effect and the new deadline is earlier than that of the running task; or
- there is a non-empty ready queue for that processor with a higher priority than the priority of the running task.

2.4 Mixed scheduling

All the above descriptions have assumed that a single dispatching policy is in effect for the whole program. The current language however goes further and allows mixed systems to be defined. To accomplish this, the system's priority range can be split into a number of distinct non-overlapping bands. In each band, a specified dispatching policy can be defined. So, for example, there could be a band of fixed priorities on top of a band of EDF with a single round robin level for non-real-time tasks at the bottom. To illustrate this, assume a priority range of 1..16:

```
pragma Priority_Specific_Dispatching
       (FIFO_Within_Priorities, 10, 16);

pragma Priority_Specific_Dispatching
       (EDF_Across_Priorities, 2, 9);

pragma Priority_Specific_Dispatching
       (Round_Robin_Within_Priorities, 1, 1);
```

Any task is assigned a dispatching policy by virtue of its base priority. If a task has base priority 12 it will be dispatched according to the fixed priority rules, if it has base priority 8 then it is under the control of the EDF rules. In a mixed system, all tasks have a preemption level (it is just their base priority) and all tasks have a deadline (it will be `Default_Deadline` if none is assigned in the program). But this deadline will have no effect on dispatching if the task in not in an EDF band.

To achieve any mixture including one or more EDF bands, the properties assigned in the definition of EDF dispatching to level `Priority'First` need to be redefined to use the minimum value of whatever range is used for the EDF tasks (in the above case, priority level 2). Also note that two adjacent EDF bands are not equivalent to one complete band. Runnable tasks in the upper bands will always take precedence over runnable tasks in the lower bands, even if the latter ones have earlier deadlines.

Tasks within different bands can communicate using protected objects (and rendezvous if required). The use of `Ceiling_Locking` for the entire partition ensures that protected objects behave as required. For example an 'EDF task' (preemption level 7) could share data with a 'round-robin' task and a 'fixed priority' task (priority 12). The protected object would have a ceiling value of (at least) 12. When the EDF task accesses the object its active priority will rise from 7 to 12, and while executing this protected action it will prevent any other task executing from within the EDF band. The 'round-robin' task will similarly execute with priority 12 – if its quantum is exhausted inside the object it will continue to execute until it has completed the protected action.

3 Generalizing the EDF Support

Given that Baker's protocol separates out the notion of urgency from preemption level, then it should be possible to generalize the support for PLCP so that the application can defined its own notion of urgency. Following the structure given in Section 2 this requires:

1. a mechanism of defining a general task attribute on which to base dispatching decisions (a user-defined notion of urgency that we will refer to as execution eligibility)
2. the use of this dispatching attribute to control dispatching within a priority-level
3. a means of sharing data via protected objects that is compatible with the dispatching policy.

Of course, if we can find a simple mapping between whatever dispatching criteria is desired and the Ada model of priority, then it is possible to just apply the translation and use the Ada model directly. In situation where this is not possible – perhaps the range of the attribute is much greater than the priority range or it has more than one component value – then the above approach is needed.

3.1 Representing general dispatching attributes

Although Ada supports task attributes via the package `Ada.Task_Attributes`, this package is generic and the `Attribute` type is not tagged. This makes it difficult to use subprograms with class-wide parameters. Instead, we choose to define a new type and introduce a child package of `Ada.Dispatching`.

```
with Ada.Task_Identification;
use Ada.Task_Identification;
with Ada.Real_Time; use Ada.Real_Time;
with System; use System;
package Ada.Dispatching.Attribute_Based is
   type Dispatching_Attribute is
        abstract tagged null record;
   type Any_Dispatching_Attribute is
        access all Dispatching_Attribute'Class;

   function ">" (LHS, RHS: Dispatching_Attribute)
        return Boolean is abstract;
   function Default_Attribute_Value
        return Dispatching_Attribute
```

```
            is abstract;

   procedure Set_Dispatching_Attribute(
              A: Any_Dispatching_Attribute;
              T : Task_Id := Current_Task);
   procedure Delay_Until_And_Set_Attribute(
              Delay_Until_Time : Time;
              A: Any_Dispatching_Attribute);
   function Get_Dispatching_Attribute(
             T : Task_Id := Current_Task)
                return Any_Dispatching_Attribute;
   procedure Register_Dispatching_Attribute(
             D: Any_Dispatching_Attribute;
             pri1, pri2 : Priority);
      -- Raises Dispatching_Policy_Error if pri1,
      -- pri2 is not set to User_Defined using
      -- pragma Priority_Specific_Dispatching.
      -- Raises Dispatching_Policy_Error if there
      -- is already an attribute registered
end Ada.Dispatching.Attribute_Based;
```

The specification of this package is modelled on that of the `Ada.Dispatching.EDF` package except now the $Deadline$ is generalised to a tagged type and called a `Dispatching_Attribute`. Also, as this is a general facility, it is necessary to provide a registration mechanisms and a function that will allow the Ada run-time support system to compare two attributes to determine their order. Hence a task with attribute $A1$ is more eligible for execution than a task with attribute $A2$ if $A1 > A2$.

The above allows us to be backward compatible with Ada 2005. An alternative is to modify `Ada.Dispatching` itself (and thereby removing its 'pure' status).

```
with Ada.Task_Identification;
use Ada.Task_Identification;
with Ada.Real_Time; use Ada.Real_Time;
with System; use System;
package Ada.Dispatching is
   Dispatching_Policy_Error : exception;

   type Dispatching_Attribute is
        abstract tagged null record;
   type Any_Dispatching_attribute is
        access all Dispatching_Attribute'Class;

   function ">" (LHS, RHS: Dispatching_Attribute)
            return Boolean is abstract;
   function Default_Attribute_Value
            return Dispatching_Attribute
            is abstract;

   procedure Set_Dispatching_Attribute(
              A: Any_Dispatching_Attribute;
              T : Task_Id := Current_Task);
   procedure Delay_Until_And_Set_Attribute(
              Delay_Until_Time : Time;
              A: Any_Dispatching_Attribute);
   function Get_Dispatching_Attribute
            (T : Task_Id := Current_Task)
              return Any_Dispatching_Attribute;
   procedure Register_Dispatching_Attribute
             (D: Any_Dispatching_Attribute;
             pri1, pri2 : Priority);
     -- Raises Dispatching_Policy_Error
end Ada.Dispatching;
```

Now of course, the EDF package would need to be rewritten.

```
package Ada.Dispatching.EDF is
   type Deadline is new
        Dispatching_Attribute with
      record
         D: Time;
      end record;
   overriding function ">" (LHS, RHS: Deadline)
      return Boolean;
   overriding function Default_Attribute_Value
      return Deadline;
end Ada.Dispatching.EDF;
```

A similar approach can obvious be taken for other dispatching criterial. For example, Value-Based scheduling (VBS) with static job-level *values*

```
package Ada.Dispatching.VBS is -- or
-- package Ada.Dispatching.Attribute_Based.VBS
   type Value is new Dispatching_Attribute with
      record
         V: Integer;
      end record;
   overriding function ">" (LHS, RHS: Value)
     return Boolean;
   overriding function Default_Attribute_Value
     return Value;
end Ada.Dispatching.VBS; -- or
-- end Ada.Dispatching.Attribute_Based.VBS;
```

Now all that is required is to be able to assign a user-defined dispatching attribute to a task. Here, we assume that all dispatching attributes can be set by a pragma

```
pragma Initial_Dispatching_Attribute(
       Attribute_Name, Value);
```

Where `Attribute_Name` must be a derived type of `Dispatching_Attribute` and `Value` an object of type `Attribute_Name`.

Hence in the above implementation of values, it would be necessary to instantiate the attributes at the library level instead of in the package body.

This would allow a task to be declared:

```
task Example is
  pragma Initial_Task_Attribute(
         Ada.Dispatching.VBS.Value,
         Some_Value);
end Example;
```

3.2 Dispatching

Firstly we assume the extensions to the `Task_ Dispatching_Policy` and the `Priority_ Specific_ Dispatching` pragmas:

```
pragma Task_Dispatching_Policy(
       User_Defined_Across_Priorities);
pragma Priority_Specific_Dispatching
       (User_Defined_Within_Priorities,
       pri1, pri2);
```

In order to generalize the EDF dispatching rules, it is necessary to define dispatching points other than those used for the general pre-emptive priority-based scheduling. Thus a dispatching point for the currently running task T occurs whenever:

1. T is executing within a user-defined scheduling band and the value of its dispatching attribute is changed (with the usual proviso that this does not have an immediately impact if called from within a protected action); or

2. any task executing on that processor within the same scheduling band has its dispatching attribute changed.

3.3 The Preemption Level Control Protocol

Baker's protocol states that a newly released task, T1 say, preempts the currently running task, T2 say, if and only if:

- execution eligibility of T1 is greater than the execution eligibility of T2, and
- the preemption level of T1 is higher than the preemption level of any locked protected object (i.e. protected objects that are currently in use by any task in the system).

Hence, we can generalize the Ada PLCP protocol so that

- Whenever a task T is added to a ready queue within a user-defined scheduling band, other than when it is preempted, it is placed on the ready queue with the highest priority R, if one exists, such that:
  - another task, S, is executing within a protected object with ceiling priority R; and
  - task T has a greater eligibility than task S; and
  - the base priority of task T (its preemption level) is greater than R.

  If no such ready queue exists, the task is added to the first ready queue of the band.
- When a task is chosen for execution, it runs with the active priority of the ready queue from which the task was taken. If it inherits a higher active priority, it will return to its original active priority when it no longer inherits the higher level.

Consequently, by providing a mechanisms that allows the application to control the order of the run queues, any scheduling approach that is appropriate to be used in conjunction with the PLCP can be programmed. Examples include any static job-level scheduling scheme such as static value-based scheduling or static least-slack time scheduling [1][3].

4 Discussion and Future Work

Over the last few Real-Time Ada Workshops we have strived to get more features into Ada to allow more flexibility scheduling approaches to be implemented. Whilst we have failed to convince the ARG of the need for complete user-defined scheduling, we have succeeded in getting round-robin and EDF scheduling. EDF scheduling required Baker's protocol to be supported. Baker's protocol has wider applicability than just EDF scheduling. This observation has motivated us to consider what mileage could be had by generalizing the EDF support.

Of course, this has only been a paper study, and we would welcome interaction with those who implement Ada compilers to test the efficacy and efficiency of the proposal made here. However, there is perhaps a more immediate discussion to have at the workshop. This concerns the generality of Baker's algorithm. All scheduling algorithms schedule their tasks in an order specified by some eligibility criterium; for example, highest priority, least slack time etc. The execution eligibility of a task can be *static*, *job-level static* or *dynamic*. A static eligibility does not change during the lifetime of the task, a job-level static eligibility is one that does not change during the release of the task. EDF is a good example: the eligibility is the absolute deadline and this remains fixed during each release. A dynamic eligibility is one where the eligibility of a task changes throughout the task's current release. Slack-time is a good example. The PLCP protocol can be used with any scheduler

that supports only static and job-level static execution eligibilities. The preemption level of a task is also a static parameter that specifies the resource usage eligibility of that task. If a task t1 has higher execution eligibility than another task t2, even if released later than t2, then it must also have a higher preemption level than t2. In other words, in situations where no locking takes place, preemption levels concur with execution eligibilities as to which task should run next.

The issue raised by this paper that need further discussion are

- whether other static job-level schedulers are worthwhile given the direct support for EDF?
- are they any other class of dynamic execution that can be constrained to be used with the PLCP?
- is there any other resource contention protocol that if supported by Ada would allow the easy addition of a wider range of user-defined schedulers?

Acknowledgements

The authors gratefully acknowledge the contributions of Pat Rogers to some of the ideas discussed in this paper.

References

[1] T. P. Baker. Stack-based scheduling of realtime processes. *Real Time Systems*, 3(1), 1991.

[2] A Burns and A. J. Wellings. *Concurrent and Real-Time Programming in Ada*. Cambridge University Press, 2007.

[3] J.W.S. Liu. *Real-Time Systems*. Prentice Hall, 2000.