Providing Additional Real-Time Capability and Flexibility for Ada 2005

Rod White
MBDA Missile Systems (UK) Ltd.
Bristol, UK
rod.white@mbda-systems.com

Abstract

Whilst Ada 2005 provides extensive support for the creation of real-time systems in general, and high-integrity ones in particular, there are a number of areas where the language could be both strengthened and made more flexible. This paper discusses a number of such areas, including scheduling approaches and the management of sporadic tasks. Following on from these, some small additions to the language are proposed with the goal of facilitating the wider adoption of certain approaches — namely non-preemptive and earliest deadline first dispatching. The paper also considers, more generally, the possible development of the language to address the issues inherent in emerging processing platforms, specifically "multi-core" devices.

1 Introduction

Ada 2005 [4] provides some useful additions, over those available in Ada 95 [3], for the construction of practical real-time systems. However there are some areas where the language could be further strengthened and made more flexible, and there are others where there would be a significant benefit in identifying coding "patterns" that fulfilled common needs. This paper discusses five such topic areas, spanning a significantly wide range of issues; these are:

- Adding flexibility to the non-preemptive dispatching model to provide both improved transparency and the ability to yield only to tasks of a higher priority than that of the current task;
- The provision of a variant of the Ravenscar profile [4, 1] to encourage the use of Earliest Deadline First (EDF) scheduling in high-integrity applications;
- Approaches to the release of sporadic (and aperiodic?) tasks executing under an EDF dispatching policy;
- An approach to managing the release of sporadic tasks in circumstances where it is essential that they are released at regular intervals, but in environments where the release event might be "unreliable";
- The exploitation of multi-core processors with multi-tasking systems — in particular, how to manage the association of particular tasks with particular processors or cores.

More detail on each of the above topics is provided in the sections below, starting with the issue of providing enhanced flexibility in the use of non-preemptive dispatching.

2 Adding flexibility to non-preemptive dispatching

When using non-preemptive dispatching, currently the only control over scheduling in an application is for tasks to execute a delay (or delay until) statement to either "now" or some time in the past, e.g. "`delay 0.0`", "`delay -1.0`", or "`delay until Clock_First`" — these are so-called "non-blocking delay statements" [4, Section D.2.4 9/2] The execution of such statements causes the current task to be moved to the back of the ready queue for its priority, thus allowing any other ready task at the same or higher priority to be scheduled. This results in the task that voluntarily relinquished the processor not being scheduled again until all of the ready tasks at the same and higher priorities have run and either been suspended, or have executed a similar non-blocking delay statement.

Whilst this is acceptable if the goal is to share the processing resources "fairly" between tasks, it can have limitations with respect to the responsiveness of various parts of an application in that higher priority tasks may be blocked for considerable periods by lower priority ones. An additional issue is that it also restricts the points at which a task may perform such non-preemptive dispatching operations — for example they cannot occur within a protected subprogram.

There are two approaches to addressing these limitations: the first is to assign (and maintain) unique priorities for all tasks — this ensures that only tasks of a higher priority will be scheduled when the current task performs such a non-blocking delay statement. Clearly this is only possible where the number of tasks does not exceed the number of priorities, which on some platforms can be quite limited due to the design of the underlying operating system. Additionally, this approach does not address the problem of being able to relinquish the processor in a protected subprogram.

The second approach is to introduce an additional runtime call that allows the current task only to relinquish the processor to tasks of a higher priority — a `Yield_To_Higher` operation. When this call is made the current task is suspended and placed at the head the ready queue for its priority — in much the same way as it would be were it to have been preempted under other dispatching policies. Thus the suspended task is only blocked until all of the ready, higher priority tasks have been run — after which the original task is resumed, much as it would have been had it been truly preempted. By only allowing higher priority tasks to be run the overhead of such an operation, in terms of disruption to the calling task, is potentially much lower than that associated with the use of the non-blocking delay statement.

The overhead of the call under conditions where no higher priority tasks are ready should be negligible — thus the addition of such calls in a significant density would introduce little overhead, but would make the application generally more responsive to higher priority events, and hence improve its overall schedulability. Finally, as this call only relinquishes the processor to tasks of strictly higher priority it could be used within a protected subprogram.

As an addition a general `Yield` operation could also be provided. This would have the same behaviour as the current non-blocking delay statement but would benefit the language by providing some clarity — i.e. the lack of need to interpret the meaning of a particular delay. The implementation of such a runtime facility could, of course, use the non-blocking delay approach.

The code fragment below suggests the general form of the specification for the additions the runtime:

```
package Ada.Dispatching.Non_Preemptive is

   procedure Yield_To_Higher;

   procedure Yield; --  Bounded error if executed within a
                    --  protected operation

end Ada.Dispatching.Non_Preemptive;
```

In common with a number of other operations there is a need to exclude the possibility of the execution of both `Yield` and `Yield_To_Higher` operations from the body of an entry.

An additional option that might be worthy of consideration is that of performing `Yield_To_Higher` operations "under the hood" at the points where the active priority of the current task is modified — for example the entry to, and exit from protected operations. This would ensure that higher priority tasks, which had been released from a protected entry, could be run in a timely manner without the need for the application to explicitly insert the appropriate "yield" operations. Perhaps a partition-wide pragma could be used to provide the control over such a feature; this would result in the solution being static with no ability to change the behaviour dynamically at run-time — probably not a bad restriction. A possible pragma, only applicable under *Non_Preemptive_FIFO_Within_Priorities* dispatching, would be:

```
pragma Cooperate_On_Priority_Change;
```

3 Extending the Ravenscar profile for EDF dispatching

In the high-integrity domain systems are often resource constrained with high levels of processor utilisation, conditions which can benefit from the use of EDF scheduling. Unfortunately the current definition of the high-integrity Ravenscar profile [1] explicitly states that the dispatching policy has to be `FIFO_Within_Priorities`, thus the possibility of using EDF dispatching with Ravenscar is currently excluded.

Given the origins of the Ravenscar profile this position is clearly understandable; the EDF approach to scheduling was not provided in Ada until Ada 2005, whereas the definition of the Ravenscar profile was developed somewhat earlier. However it appears that the profile only requires a small change to accommodate EDF scheduling; this relates to the restriction that limits the `Task_Dispatching_Policy` to be `FIFO_Within_Priorities`. If the restriction over the dispatching policy is simply relaxed to allow both the current `FIFO_Within_Priorities` and `EDF_Across_Priorities`, then it appears that EDF could then be used with the other Ravenscar restrictions.

Access to the profile would be through the `pragma Profile(..., ...);` statement; there is no need to introduce a new profile identifier, instead the profile can remain as being designated `Ravenscar`, with the profile argument being used to select which variant is to be used. Hence using such a *Ravenscar with EDF* profile would require the inclusion of the partition-wide pragma:

```
pragma Profile (Ravenscar, EDF_Across_Priorities);
```

For backwards compatibility, omitting the argument would result in the current Ravenscar profile, i.e. the `FIFO_Within_Priorities` version. It may also be useful, however, to allow the argument `FIFO_within_Priorities`; this would have the same effect as if there were no argument, but would have the advantage of making the selection of the variant of the profile explicit and visible. Taking the approach of using the argument to parametrize the policy opens the door to the simple inclusion of other variants of Ravenscar, for example one using the `Non_Preemptive_FIFO_Within_Priorities` dispatching approach.

A final question to consider here is: are there any additional restrictions that need to be enforced in a Ravenscar and EDF environment? At this point the answer appears to be no. Equally, the alternative of there being any scope for relaxing any of the current Ravenscar restrictions also seems to be unlikely.

4 Sporadic release under EDF dispatching

The current approach to EDF dispatching in Ada 2005 assumes that the tasks are generally periodic, and are scheduled using the "language" clock. Often in embedded systems there are tasks that run periodically, but whose release is not under the control of this language clock, instead these are scheduled using some external event related to some overall or common system clock — hence these tasks are what would more usually be considered to be sporadic in nature. Unfortunately, unlike for the

periodic tasks, where the `Delay_Until_And_Set_Deadline` procedure is provided, Ada 2005 does not give any assistance for setting the deadlines of sporadic tasks scheduled under the EDF dispatching policy at the point at which they are suspended.

Two possible approaches to addressing this limitation are described below: the first, using the existing language features, is based on protected objects; and a second, based on suspension objects, requiring an extension to a predefined package.

4.1 The protected object approach

Perhaps the simplest approach to addressing this shortcoming is to have the sporadic task waiting on a protected entry that, when released by some protected procedure attached to an interrupt handler, has its deadline set in the body of the entry. This poses the obvious question: how do we know what deadline to set?

Continuing with a simple approach the prospective waiter (sporadic task) can "register" itself with the "release manager" protected object (PO) and provide a relative deadline that may be used on each subsequent release. By only allowing a single waiter to be associated with any given PO, the task ID could also be recorded at the point of "registration" and stored in the PO, rather than being derived dynamically from the `'Caller` attribute in the body of `Wait` — the approach that is adopted in the example below.

An example for the code of such a sporadic release manager PO is:

```
protected type Sporadic_Release_Manager (Interrupt : Interrupt_Id) is
   procedure Release;
   pragma Attach_Handler (Release, Interrupt);
   procedure Register (RD : in Time_Span);
   entry Wait;
private
   Relative_Deadline : Time_Span := Time_Span_Last;
   Barrier_Open      : Boolean := False;
end Sporadic_Release_Manager;

protected body Sporadic_Release_Manager is

   procedure Release is
   begin
      Barrier_Open := True;
   end Release;

   procedure Register (Rd : in Time_Span) is
   begin
      Relative_Deadline := Rd;
   end Register;

   entry Wait when Barrier_Open is
   begin
      Set_Deadline (D => Clock + Relative_Deadline,
                    T => Wait'Caller); -- Task_ID of the waiter
      Barrier_Open := False;
   end Wait;

end Sporadic_Release_Manager;
```

Under conditions where it is required that the clock event is only responded to when "fresh" (i.e. the waiter is already present at the time of the event) the `Release` procedure may be replaced with:

```
procedure Release is
begin
   Barrier := Wait'Count /= 0;
end Release;
```

Where the deadline is a one-time fixed value the need to explicitly "register" the task with its instance of `Sporadic_Release_Manager` to provide the deadline can be removed through the use of an additional discriminant on the protected object. The relative deadline cannot be provided directly as its type is private (`Time_Span`), but it can be provided as a `Positive` value and be subsequently converted to the required type — in the case of the example below the discriminant `RD` is scaled to a number of microseconds. Taking this approach the specification of the `Sporadic_Release_Manager` becomes:

```
protected type Sporadic_Release_Manager (Interrupt : Interrupt_Id;
                                         RD        : Positive) is
   procedure Release;
   pragma Attach_Handler (Release, Interrupt);
   entry Wait;
private
   Relative_Deadline : Time_Span := Microseconds(RD);
   Barrier_Open      : Boolean := False;
end Sporadic_Release_Manager;
```

A more flexible solution is to provide the relative deadline as a parameter to the `Wait` entry call taking this approach the body of `Wait` becomes:

```
entry Wait (Relative_Deadline : in Time_Span) when Barrier_Open is
begin
   Set_Deadline (D => Clock + Relative_Deadline,
                 T => Wait'Caller); -- Task_ID of the waiter
   Barrier_Open := False;
end Wait;
```

This approach obviates the need to "register" the task and its deadline with the instance of the protected object, removes the `Relative_Deadline` state variable, and provides flexibility in allowing the deadline to be different for each call to `Wait`.

An alternative possible approach to providing the relative deadlines for each task would be to use task attributes. In common with the previous approaches this too removes the need to explicitly "register" the task with the `Sporadic_Release_Manager` PO, the deadline in this case being obtained in the body of `Wait` through the `Wait'Caller` attribute. Unfortunately this would not be compliant with a possible *Ravenscar with EDF* profile as the dependency on the package `Ada.Task_Attributes` is not allowed. An example taking this approach is given below. It assumes that there is an instance of `Ada.Task_Attributes` called `Sporadic_Deadlines`, which provides the relative deadlines, when in the body of the `Wait` entry; hence the body of the `Wait` entry becomes:

```
entry Wait when Barrier_Open is
begin
   Set_Deadline (D => Clock + Sporadic_Deadlines.Value (Wait'Caller),
                 T => Wait'Caller);
   Barrier_Open := False;
end Wait;
```

The above approaches, with their variants, are adequate for supporting the sporadic release under the EDF policy if there is a one-to-one association of sporadic waiters with events. Often however this is not the case and there may be several waiters on the same event, each with a potentially different relative deadline — the fan-out, or multicast event problem. Should a *Ravenscar with EDF* profile be agreed then a suitably compliant solution to this problem would require multiple instances of a release PO, one per sporadic task requiring release in response to the interrupt, with the interrupt attached protected procedure calling `Release` in each case. The code fragment below gives a generic solution to the multicast event problem for this case:

```
generic
   type Waiter_Range is range <>;
   Interrupt_ID : Interrupt_ID;
package Multicast is

   procedure Initialize (Waiter : in Waiter_Range;
                         RD     : in Time_Span);

   procedure Wait (Waiter : in Waiter_Range);

end Multicast;

package body Multicast is

   protected type Event_Handler (Interrupt : Interrupt_ID) is
      procedure Release;
      pragma Attach_Handler (Release, Interrupt);
   end Event_Handler;

   protected type SRM is
      entry Wait;
      procedure Release;
      procedure Initialize (RD : in Time_Span);
   private
      Barrier_Open : Boolean := False;
      Relative_Deadline : Time_Span := Time_Span_Last;
   end SRM;

   Interrupt_Handler : Event_Handler (Interrupt_ID);
   Multicast_Waiters : array (Waiter_Range) of SRM;

   protected body Event_Handler is
      procedure Release is
      begin
         for Index in Waiter_Range loop
            Multicast_Waiters (Index).Release;
         end loop;
      end Release;
   end Event_Handler;

   protected body SRM is

      procedure Release is
```

```
      begin
         Barrier_Open := True; -- "freshness" test could be applied
      end Release;

      procedure Initialize (RD : in Time_Span) is
      begin
         Relative_Deadline := RD;
      end Initialize;

      entry Wait when Barrier_Open is
      begin
         Set_Deadline (D => Clock + Relative_Deadline,
                       T => Wait'Caller);
         Barrier_Open := False;
      end Wait;
   end SRM;

   ----------
   -- Wait --
   ----------

   procedure Wait (Waiter : in Waiter_Range) is
   begin
      Multicast_Waiters (Waiter).Wait;
   end Wait;

   ----------------
   -- Initialize --
   ----------------

   procedure Initialize (Waiter : in Waiter_Range;
                         RD     : in Time_Span) is
   begin
      Multicast_Waiters (Waiter).Initialize(RD);
   end Initialize;

end Multicast;
```

Clearly "non-Ravenscar" compliant versions of this are also possible, but these do not add anything new and are, hence, not discussed further here.

4.2 The suspension object approach

All of the above approaches have used the Ada 2005 language as it exists today. An alternative approach requiring a language change, albeit simply the addition of a child of a predefined package, is to consider the use of the `Suspension_Object` (SO) in `Ada.Synchronous_Task_Control` as a trigger for sporadic tasks. Currently tasks call `Suspend_Until_True(...)` to wait on the SO until it is set `true`. A simple extension would be to provide the caller with the ability to provide a deadline as part of this suspension call, in a similar way to the `Delay_Until_and_Set_Deadline` provided for periodic tasks. This could be simply introduced as a child of the current `Ada.Synchronous_Task_Control package`, in something like:

```
with Ada.Real_Time;
package Ada.Synchronous_Task_Control.EDF is

   procedure Suspend_Until_True_And_Set_Deadline
                       (S : in out Suspension_Object;
                        D : in     Ada.Real_Time.Time_Span);

end Ada.Synchronous_Task_Control.EDF;
```

Of course, this package is only required, and would only have any meaning, when the task dispatching policy `EDF_Across_Priorities` is being used.

Tasks suspended on such a call would be released by the interrupt attached protected procedure calling the `Set_True` subprogram with the appropriate SO. At the point this was called the deadline would be applied thus ensuring that it was relative to the event and that the task was able to be scheduled in accordance with the policy. Of course the use of the suspension object does not allow the "freshness" test to be applied at the point of release, thus a task wishing to suspend until the arrival of the next event would have to call the `Set_False` subprogram prior to suspending otherwise it might be resumed immediately in response to a "latent" event.[1]

Whilst this approach does require a language change its simplicity and similarity with the facility provided for the periodic tasks makes it very appealing.

5 Ensuring reliable release of sporadic tasks in the presence of an "unreliable" event

Another aspect to consider, related to the release of sporadic tasks in response to periodic events derived from an external clock, is that where there is a need to ensure that a clock "tick" always arrives. There are two common circumstances where such a requirement exists:

- In the initial stages of development when the external clock, generated in another part of the system, is not available and there is the need to "stimulate" parts of the system for early development and testing;
- Operationally, in the completed system, where the clock maybe unreliable[2] and there is the need to ensure that some chain of processing always occurs at regular intervals, e.g. updating a control algorithm.

The provision of this reliable release of sporadic tasks can be achieved through the use of timing events initiated upon each "interrupt". If the interrupt (from the external clock) occurs then the timing event will be reset to the predicted time of the next interrupt[3] and the guard on `Wait` will be released; if alternatively the interrupt fails to occur then the timing event handler will be executed which too will release the guard on `Wait`, thus ensuring that any downstream processing can proceed as usual. When the normal clock event occurs the time for the timing event is moved forward to that predicted for the next external event.

The following code provides an example of such a mechanism:

[1] Perhaps another useful addition here would be a `Set_True_If_Waiting` procedure that would only set the SO `True` if these was a waiter.

[2] Here unreliable is restricted to mean those events that might not occur, or might be "late" — the possibilities of early or too rapid events are not considered.

[3] Actually slightly later than the time of the next interrupt to allow for the "jitter" in the external clock.

```
protected type Sporadic_Release_Manager (P  : Positive;
                                         M  : Positive;
                                         Id : Interrupt_ID) is
   procedure Release;
   procedure Timeout_Handler (Event : in out Timing_Event);
   pragma Attach_Handler (Release, Id);
   entry Wait;
   procedure Initialize;
private
   Period : Time_Span := Milliseconds (P); -- Period of the clock
   Margin : Time_Span := Microseconds (M); -- Clock jitter allowance
   Barrier_Open : Boolean := False;
   Timeout : Timing_Event;
end Sporadic_Release_Manager;

protected body Sporadic_Release_Manager is

   procedure Initialize is
   begin
      Set_Handler (Event   => Timeout,
                   In_Time => Period + Margin,  -- Relative to "now"
                   Handler => Timeout_Handler'Access);
   end Initialize;

   procedure Release is
   begin
      Barrier_Open := Wait'Count /= 0;
      Set_Handler (Event   => Timeout,
                   At_Time => Clock + Period + Margin, -- Absolute
                   Handler => Timeout_Handler'Access);
   end Release;

   procedure Timeout_Handler (Event : in out Timing_Event) is
   begin
      Barrier_Open := Wait'Count /= 0;
      Set_Handler (Event   => Timeout,
                   At_Time => Time_Of_Event (Event) +
                              Period, -- Absolute (margin built-in)
                   Handler => Timeout_Handler'Access);
   end Timeout_Handler;

   entry Wait when Barrier_Open is
   begin
      Barrier_Open := False;
   end Wait;

end Sporadic_Release_Manager;
```

The value `Margin`, set from the `M` discriminant, allows for the acceptable (worst-case) jitter inherent in the external clock and is used to prevent the timer event from firing prematurely — the value could be set to zero where the event is known to be permanently unavailable, e.g. during early development. The `Initialize` procedure is called prior to the first call to `Wait` to set-up the timing event to provide

an initial time-out; the time-out is propagated for the next event in `Release` or `Timeout_Handler`.

Finally, in an EDF context this approach could be combined with those described in Section 4, with the deadline being set in the body of `Wait`.

6 Associating tasks with cores in multi-core processors

The previous sections have identified concrete approaches to practical issues encountered in the development of real-time systems; in this section the approach is more of a discussion of the issues surrounding the use of multi-core processors in these systems, rather than being a set of suggested solutions.

In order to be able to exploit multi-core processors effectively it is essential that it is possible to define on which core(s) a particular task is able to be executed — or the ones with which it has an "affinity". There appear to be three aspects to the problem:

1. Identifying the processing resources — what they are, and how they might be referenced;
2. Defining how tasks are associated with processors — the possible modes of association;
3. Defining ways in which the interactions may be controlled — the dynamic aspects.

The identification of the resources requires that each processor be identified with a token or some description, and (probably) a `pragma` is required to associate the task with a set of these resources — this would provide the default association. An alternative to the `pragma` approach might be possible, but the question of where in the task's life its default affinity becomes binding is left rather open — for example it would be difficult to use a non-pragma approach if the affinity had to be defined before the beginning of the execution of the task body.

Considering the second aspect, that of the modes of association between processors and tasks, these appear to be able to take a number of forms; possible options being:

- Free-selection — any processor from the defined set may be chosen at any point where the task starts or resumes execution;
- Lock-to-first — the task will lock itself to the first processor on which it runs;
- Lock-on-preemption — when resumed after preemption the task will run on the same processor as it was executing when preempted. At other points of resumption it may run on any of the processors in its association set.

These three options appear to give adequate flexibility, though it may be that not all of these would be provided by an implementation, a possibility being that the preempted task will always be resumed on the processor from which it was preempted. To a large extent the extent of the capabilities will be constrained by the behaviour of the underlying operating system of the platform.

Moving on to consider the third aspect; it would appear that, from a flexibility viewpoint, it is desirable to be able to change the affinity of a task (and perhaps its mode of association). Assuming this is allowed, a fundamental question here is: when does the change take place? There are (at least) two options; it may:

- Occur immediately; or,
- Where the task who's affinity is being modified is currently executing, it may be deferred until the next dispatching point for that task.

The first case makes changing the affinity a dispatching point. Closely allied to this is the question of whether any affinity changing operation should be able to operate on any task, or simply on the current task?

Considering a different aspect of the multi-core problem: what happens when two (or more) simultaneously running tasks contend over access to an instance of a protected object? Clearly one will gain access and the other(s) will be blocked from entering,[4] thus the question is: what does the blocked task do? There are (at least?) two possibilities:

- The task continues to execute on its processor, at the priority of the PO, polling some form of "lock" on the PO until access is granted; or
- The task suspends itself and waits to be released once access is possible. This would require the PO to have a queue of possible accessors, from which tasks are made ready by the tasks completing access to the PO — the accessors would all be set to the priority of the PO before being queued.

The former is simple compared to the second, but the second might result in better use of resources.

All of the above discussions assume that the language allows for a number of processors to be associated with a task (a task-centric view); an alternative viewpoint would be to associate each processor with set of tasks (a processor-centric view). The question is: which of these is the more appropriate? As the concept of the task is firmly embedded in the language the task-centric view appears quite natural; however if the dynamics of resource scheduling are considered then possibly the processor-centric approach becomes the more natural. Even so, irrespective of the approach, there is the need to identify the processor (or processing resource) in some way.

This is clearly not an exhaustive consideration of the multi-core problem, other areas that perhaps need attention are:

- The meaning of priority — does it need to be refined for these environments?
- "Simultaneous" release of several waiters on a single entry (guard) on many processors — can this be "efficiently" supported?
- Heterogeneity amongst the processing cores — does this raise other issues or requirements?
- "Priority" of association — should it be possibly to state a preference for one core over another (applicable to the case where the cores are heterogeneous)?
- Wider applicability to multiple, more loosely coupled processors more generally — again does this expose other issues?
- Processor coupling — are restrictions imposed by dependencies between processors?
- Inter-task communications — is additional language support required? Especially for non-blocking interactions where the tasks are executing on separate processors (e.g. the 4-slot mechanism [2]).

7 Conclusions

The preceding sections of this paper have addressed a number of areas where the Ada language could be "improved" with respect to its real-time capabilities. In essence three main areas have been considered:

- Scheduling approaches — improving flexibility: non-preemptive dispatching features, and the combination of EDF and Ravenscar;
- Sporadic tasks — their use with EDF dispatching, and issues of event reliability;
- Multi-core/processors — identifying issues to address regarding exploitation.

[4] Though multiple simultaneous function calls should be allowed.

Whilst there is some cross-over between these (e.g. EDF, Ravenscar, and sporadic tasks) they can generally be seen as being largely independent.

The additions for non-preemptive scheduling provide additional flexibility and clarity, whilst the *Ravenscar with EDF* profile builds on the existing Ravenscar profile broadening it to bring EDF into the high-integrity arena. The approach suggested also simplifies the provision of future variants of the profile using other dispatching policies.

The various approaches to the release and management of sporadic tasks address a number of the practical issues with their use and the exploitation of EDF scheduling. It also identifies a small extension to the language that allows the deadlines for sporadic tasks to be treated in much the same way as those for periodic tasks.

Finally, the discussions regarding the support for multi-core processors (and multiple processors more generally) is a first step towards the identification of features that will allow these to be fully exploited.

References

[1] Information technology – Programming languages – guide for the use of the ada Ravenscar profile in high integrity systems. Technical Report TR24718:2005, ISO/IEC.

[2] H. R. Simpson. Four-slot fully asynchronous communication mechanism. *IEE Proceedings*, 1990.

[3] S. Tucker Taft and Robert A. Duff, editors. *Ada 95 Reference Manual. Language and Standard Libraries, International Standard ISO/IEC 8652:1995(E).* Springer-Verlag, 1997.

[4] S. Tucker Taft, Robert A. Duff, Randall L. Brukardt, Erhard Ploedereder, and Pascal Leroy, editors. *Ada 2005 Reference Manual. Language and Standard Libraries, International Standard ISO/IEC 8652:1995(E) with Technical Corrigendum 1 and Amendment 1.* Springer, 2006.