Implementation of Timing-Event Affinities in Ada/Linux *

Sergio Sáez, Jorge Real, Alfons Crespo
Universitat Politècnica de València

E-mail: {ssaez,jorge,alfons}@disca.upv.es

Abstract

Ada 2012 has introduced mechanisms for exploiting multiprocessor platforms at the application level. These include task affinity control and definition of dispatching domains. However, there are other executable entities defined in the language for which there is no such support to affinity control: event handlers. With event handlers *we mean both timing-event and interrupt handlers.*

This paper discusses the consequences of this lack of functionality and explores implementation issues related with this ability. We propose a working implementation for affinity of timing-event handlers on top of Linux.

1 Introduction

The Ada 2012 standard [1] introduced mechanisms that enable programmers to take advantage of multiprocessor platforms. In particular, package `System.Mutiprocessors` defines the `CPU_Range` type (a range between 0 and an implementation-defined value) and its subtype `CPU`, a range from 1 to `CPU_Range'Last`. The `CPU_Range` value 0 is reserved to mean no specific CPU, and it is represented by the constant `Not_A_Specific_CPU`. Finally, a function `Number_Of_CPUs` returns the number of CPUs available in the underlying hardware platform.

A second package, `System.Mutiprocessors.Dispatching_Domains`, allows programmers to partition their applications into distinct dispatching domains at startup. A dispatching domain is defined by a subrange of CPUs, and this package includes subprograms (`Assign_Task` and `Set_CPU`) to allocate a task to a particular CPU or to any CPU in its dispatching domain. The package is complemented with a function to query the CPU to which a task is currently allocated (`Get_CPU`) and to set a task's CPU after a given absolute delay (`Delay_Until_And_Set_CPU`). There are also functions for creating dispatching domains, obtaining the dispatching domain of a task, and to query the limits defining the range of CPUs contained in a dispatching domain.

These are indeed useful abstractions to balance the workload (*at the task level*) when there are two or more processors available in the underlying hardware. However, there are no similar mechanisms in the language to assign a CPU to the execution of a particular event handler. With *event handler* we mean more specifically timing-event handlers and interrupt handlers. If we don't have the ability to decide and determine *where* (on which CPUs) event handlers will be executed, then we cannot precisely

*This work has been partially supported by the Spanish Government's projects COBAMI (DPI2011-28507-C02-02) and Hi-PartES (TIN2011-28567-C03-01-02-03) and the European Commission's projects DREAMS (FP7-ICT-2013.3.4, Contract 610640) and EMC^2 (ARTEMIS-JU Call 2013 AIPP-5, Contract. 621429).

bound the effects of their execution at run time. If the runtime decides to execute an event handler on a particular CPU, then we should account for the corresponding interference on that CPU in the schedulability analysis. The `Get_CPU` function in package `Ada.Interrupts` returns the CPU on which an interrupt handler executes; but there is no equivalent setter subprogram. A programmer can query, once the system is running, which CPU is executing an interrupt handler; but this is of little help for a priori, off-line schedulability analysis: if we cannot precisely determine on which CPUs do all the tasks and event handlers execute, then our schedulability analysis cannot be realistic and precise.

In this paper, we want to bring elements for discussion of the convenience and feasibility of extending Ada to support affinity control for event handling. We also propose an implementation of timing-event handlers affinity on the GNAT GAP 2014 Ada compiler and runtime system (RTS) on top of the Linux Operating System.

2 Motivation

In a recent paper [3], we explored mechanisms to reliably changing a set of scheduling attributes of real-time Ada tasks (period, deadline, priority, CPU...). We showed that there are potential errors depending on the order of enforcement of the tasks' attributes and proposed several possible implementations to solve the errors and minimise scheduling artefacts (short periods of interference due to the need for the algorithms to temporarily use a very high priority). One of the approaches proposed in that paper was to use a timing event handler to make all those changes atomically to the target task, so that they are effective as soon as the task resumes. Although we found this approach to be a most convenient and efficient way of solving the issue, the lack of a mechanism to set the timing event handler affinity implies that this handler will execute (and produce interference) on an unknown CPU. No matter how short this interference may be, the fact is that we cannot realistically take it into account in the schedulability analysis.

Similarly, we need to know on which CPU an interrupt handler will be executed if we want to use realistic data in the schedulability analysis.

We note here that this issue affects not only the whole language, but also the Ravenscar profile. Ravenscar imposes a proper set of restrictions to avoid indeterminism, but nothing can be done to determine the affinity of event handlers on Ravenscar.

3 Affinity of Timing-Event Handlers

3.1 Preliminary considerations

The implementation event-handling affinity depends very much on the kind of underlying Real-Time Operating System (RTOS). A critical aspect is the particular relation between the Ada application, the Ada RTS and the RTOS. We can find the following scenarios:

Common Address Space. The application code, the Ada RTS and the RTOS kernel are all executed in a common address space with no protection by the MMU[1]. This approach is typical in embedded RTOS or executives where only one application is running and performance and resource management is of critical importance.

Separate Address Spaces. In this situation, the application code and the Ada RTS have no access to the RTOS kernel address space. This access restriction is enforced by proper configuration of

[1]Memory Management Unit

the MMU, and normally implies the use of different execution stacks depending on the execution mode: user mode or privileged/kernel mode. This configuration is typical in Unix-based platforms where multiple applications can be running concurrently.

In a *Common Address Space* configuration, the Ada RTS can have full access to the RTOS kernel resources (including e.g. setting interrupt service routines). But in a *Separate Address Spaces* configuration, the relation between the Ada RTS and the RTOS has to be strictly through the API exposed by the RTOS. In such case, the RTOS is not aware of the kind of application that is invoking its services, and it normally offers an API (e.g. POSIX) that is not tailored to any programming language in particular (apart from the C language [2]).

For this work, we have used the GNAT-GAP 2014 implementation of Ada 2012, running on top of Linux. This implementation falls in the second category (separate address spaces). We have therefore replaced standard library packages with our own modified version to test implementation alternatives to the issue at hand.

3.2 Proposed API

Setting the affinity of a timing-event handler requires changes in the programming interface, so that the affinity of the timing-event handler can be specified. We have considered two approaches:

A specific `Set_CPU` subprogram for timing events. The affinity is established for the timing event by means of a dedicated subprogram as specified in listing 1.

Adding a CPU parameter to `Set_Handler` In this case, the affinity can be specified every time the handler is set (details in listing 2).

The first option permits setting the affinity separately from setting the handler. But this appealing flexibility can produce a nasty performance drawback at run time. It is reasonable to think that the RTS will keep separate queues, one per processor –and possibly another one for any CPU on a particular dispatching domain,– to store the already set timing events for that CPU. Let's suppose an application calls `Set_Handler` and then `Set_CPU`. This would imply first reinserting the timing event in its current CPU queue (as a result of `Set_Handler`), and then removing the timing event from the CPU queue where it was and inserting it in the destination CPU's queue (as a result of `Set_CPU`). The insertion should be ordered if we want to keep the earliest timing event at the head of that queue, and ordered insertion has a cost. If we change the order (first `Set_CPU` and then `Set_Handler`), then if the timing event is already set, we would first remove the timing event from its current CPU's queue and insert it (orderly) in the destination CPU's queue (as a result of calling `Set_CPU`); and then `Set_Handler` will impose reordering the destination CPU's queue. However, a single `Set_Handler` procedure with a CPU parameter can do a single removal (if the timing event was already set) and insertion of the timing event with the correct expiration time, handler, and destination CPU.

We also prefer the second option because it helps writing more clear code. With a `Set_CPU` procedure, we can establish the affinity in one place of the program and then we can set the handler in another part of the program. This may degrade readability of the application code. The only potential advantage of a `Set_CPU` approach is that it would favour applications that are designed precisely that way: one part of the program is responsible for deciding affinities and another part is in charge of setting handlers for timing events. But even in this case, the second approach would accommodate that design: the part deciding affinities, does so by updating a shared variable containing the CPU to which the handler should be attached in the future. Then the CPU can be retrieved from such shared variable and used as the CPU parameter in `Set_Handler`.

Listing 1. Signature of a `Set_CPU` procedure to establish a timing event's affinity

```
procedure Set_CPU (Event : in out Timing_Event; CPU : CPU_Range := Get_CPU(Current_Task));
```

Basically, we need to add a CPU parameter to `Set_Handler` in `Ada.RealTime.Timing_Events`, so that the affinity can be specified for a handler at the time of setting it. The handler can therefore execute on a different CPU every time it is set. An additional `Get_CPU` function would return the CPU to which an event handler is currently assigned.

Listing 2 shows these changes in the specification of package `Ada.RealTime.Timing_Events`[2]. The changes with respect to the current Ada 2012 standard appear in the signatures of procedures `Set_Handler`, which include a new parameter `CPU` that allows the programmer to specify the handler's affinity. The affinity of the handler defaults to the affinity of the invoking task when the CPU parameter not specified. This listing includes also the specification of `Get_CPU`, to retrieve the current affinity of a *set* timing event.

Listing 2. Specification of `Ada_Real_Time.Timing_Events` with CPU affinity support

```
-- "with" clauses omitted
package Ada_Real_Time.Timing_Events is

   type Timing_Event is tagged limited private;
   type Timing_Event_Handler
      is access protected procedure (Event : in out Timing_Event);

   procedure Set_Handler (Event : in out Timing_Event;
                          At_Time : Time;
                          Handler : Timing_Event_Handler;
                          CPU     : CPU_Range := Get_CPU(Current_Task)); -- New parameter

   procedure Set_Handler (Event : in out Timing_Event;
                          In_Time : Time_Span;
                          Handler : Timing_Event_Handler;
                          CPU     : CPU_Range := Get_CPU(Current_Task)); -- New parameter

   function Current_Handler (Event : Timing_Event) return Timing_Event_Handler;

   procedure Cancel_Handler (Event : in out Timing_Event;
                             Cancelled : out Boolean);

   function Time_Of_Event (Event : Timing_Event) return Time;

   function Get_CPU (Event : Timing_Event) return CPU_Range;       -- New function

private
   type Timing_Event is new Ada.Finalization.Limited_Controlled with record
      Timeout : Time := Time_First;
      Handler : Timing_Event_Handler;
      CPU : CPU_Range := Not_A_Specific_CPU;
   end record;
   ...
end Ada_Real_Time.Timing_Events;
```

[2]Since `Ada.Real_Time.Timing_Events` is a predefined language library, we have used an alternative, similarly named `Ada_Real_Time.Timing_Events` which we import in our test programs, instead of the standard `Ada.RealTime.Timing_Events`

In the private part of the specification shown in Listing 2 we have added a new field, `CPU`, to the `Timing_Event` tagged type. It could be thought that this addition could also be done at the user side by extending the tagged type `Timing_Event`; but it is not that easy. Both `Set_Handler` and `Cancel_Handler` need access to the fields `Timeout` and `Handler` to implement their behaviour, therefore this type extension could only be implemented by a child package of `Ada.Real_Time.Timing_Events`. Since the implementation shown in listing 2 defaults to a behaviour that is coherent with the current Ada 2012 standard, the creation of a child package of `Ada.Real_Time.Timing_Events` with the extended functionality is considered not necessary for the purpose of this paper.

3.3 Implementation on top of Linux

As noted above, implementing event handlers on top of a RTOS with *Separate Address Spaces* is not straightforward, since it is the RTOS who takes the initiative to execute a piece of code in the *user address space* when the event occurs. In Unix-like OSs, events are normally associated with the *signal* abstraction, and handling a signal involves the execution of *signal handlers*, ultimately user functions implemented in C.

Although this basic support can be used to execute an application function when an event occurs (e.g. a timer expiration), in general, it does not allow the application to specify any kind of priority or affinity for the handler, because it is not possible to determine which thread is going to execute the handler. Although the Linux kernel does allow the programmer to specify an existing thread as the target for a signal, i.e., the thread that will execute the handler asynchronously[3], there is still an additional problem: it is not safe to use blocking operations within signal handlers (e.g `sem_wait`). Given that timing-event handlers are implemented in Ada by protected procedures, it is very likely that the Ada runtime system will rely on the same kind of blocking OS operations to gain access to the enclosing protected object. The execution of an OS blocking operation in the signal handler can cause a deadlock. For example, an ongoing protected action may be interrupted by the execution of a signal handler that tries to access the already locked protected object. In such case, the thread executing the handler gets blocked inside the signal handler, so the interrupted protected action can never resume and release the protected object.

Due to these constraints, we propose an initial implementation of timing events with affinities based on a set of server tasks, one per CPU, that will be in charge of executing the timing-event handlers. We are aware that this is not in line with Ada's implementation advice (ARM D.15(25/2)): *The protected handler procedure should be executed directly by the real-time clock interrupt mechanism.* However, this is the approach used to implement the execution of timing event and interrupt handlers in the Ada RTS of AdaCore's GNAT GPL 2014, in which our extension is based. The original GNAT version of timing-event handling is implemented using a single task that extracts expired timing events from an event queue on a pseudo-periodic basis. Listings 3 and 4 show how the proposed extension is implemented using multiple queues and a set of event-based server tasks, instead of pseudo-periodically polling the presence of expired timing events.

Listing 3. Timing event implementation: event Queues

```
package body Ada_Real_Time.Timing_Events is

   type Any_Timing_Event is access all Timing_Event'Class;
   package Events is new Ada.Containers.Doubly_Linked_Lists (Any_Timing_Event);
```

[3] Assuming that the corresponding signal is not blocked and has a signal handler set.

```
protected type Timing_Events_Queue is
   procedure Insert (This : Any_Timing_Event);
   procedure Remove (This : Any_Timing_Event);
   function Next_Timeout return Time;
   procedure Remove_Expired (Expired_Event : out Any_Timing_Event;
                             Is_Expired : out Boolean;
                             Handler : out Timing_Event_Handler;
                             Now : Time := Clock);
   entry Detect_First_Changed;
private
   Queue : Events.List;
   First_Changed : Boolean := False;
end Timing_Events_Queue;

protected body Timing_Events_Queue is
   -- Insert the specified event pointer into the queue of pending events
   procedure Insert (This : Any_Timing_Event) is
      function Sooner (Left, Right : Any_Timing_Event) return Boolean;
      package By_Timeout is new Events.Generic_Sorting (Sooner);
      function Sooner (Left, Right : Any_Timing_Event) return Boolean is
      begin
         return Left.Timeout < Right.Timeout;
      end Sooner;
      Next : Time;
   -- Start of processing for Insert
   begin
      Next := Next_Timeout;
      Queue.Append (This);
      -- All occurrences are in ascending order by Timeout
      By_Timeout.Sort(Queue);
      if Next > Next_Timeout then
         First_Changed := True;
      end if;
   end Insert;
   -- Remove the specified event pointer from the queue of pending events
   procedure Remove (This : Any_Timing_Event) is
      use Events;
      Location : Cursor;
   begin
      Location := Queue.Find (This);
      if Location /= No_Element then
         Queue.Delete (Location);
      end if;
   end Remove;
   -- Returns the Timeout of the closest Timing Event in the queue
   function Next_Timeout return Time is
   begin
      if Queue.Is_Empty then
         return Time_Last;
      else
         return Queue.First_Element.Timeout;
      end if;
   end Next_Timeout;
```

```
   -- Remove the closer event from the queue of pending events if it was expired
   procedure Remove_Expired (Expired_Event : out Any_Timing_Event;
                             Is_Expired : out Boolean;
                             Handler : out Timing_Event_Handler;
                             Now : Time := Clock) is
   begin
      if Next_Timeout <= Now then
         Expired_Event := Queue.First_Element;
         Queue.Delete_First;
         Is_Expired := True;
         -- From here on the handler is going to be executed even though the timing event
         -- was changed by the user once the server task exits from this protected action
         Handler := Expired_Event.Handler;
         Expired_Event.Handler := null;
      else
         Is_Expired := False;
      end if;
   end Remove_Expired;

   entry Detect_First_Changed when First_Changed is
   begin
      First_Changed := False;
   end Detect_First_Changed;
end Timing_Events_Queue;

-------------------------
-- Timing_Events Queues --
-------------------------
TE_Queues: array (0 .. Number_Of_CPUs) of Timing_Events_Queue;
```

Listing 3 shows the implementation of timing-event queues as an array of protected objects, one per CPU in the system. This protected object allows the application to `Insert` and `Remove` timing events to/from the queue corresponding to their affinities, as well as the function and protected procedures to determine the next expiration time (`Next_Timeout`) and to remove the information related to the first expired event (`Remove_Expired`). An additional entry `Detect_Firts_Changed` is also implemented to wake up the server task in the case the closer event has changed.

The server task structure is shown in listing 4. The task type `TE_Handling_Task` applies to timing-event handler executor tasks. The discriminant `CPU`, initialised by successive calls to `Next_CPU_Nr`, is applied as the value for aspect CPU to instances of this type, which ensures that each handler task is assigned to a different CPU. Tasks of this type will run at the highest priority, set by means of aspect `Interrupt_Priority`.

In the main loop of the server task body, a server task waits until the next event from the timing events queue associated with its CPU has expired. However, the wait occurs on a *timed entry call* that uses the entry `Detect_First_Changed` to wake up the task if the head of the queue has changed. This allows the server task to react to the insertion of timing events with closer or even already expired timeouts. If the event queue is empty, the server task waits forever (`Time_Last`) or until a new event is inserted.

Once a server task wakes up, it tries to execute all the expired events placed in its CPU queue, if any. This is done in the procedure `Process_Expired_Events`. The implementation of this procedure is shown in listing 5. The server tasks are all arranged in the array `TE_Server_Tasks`.

Listing 4. Server tasks for executing timing-event handlers.

```
CPU_Counter : CPU_Range := 0; -- Warning: global var updated by Next_CPU_Nr at elaboration
function Next_CPU_Nr return CPU_Range is
   CPU_Temp : CPU_Range;
begin
   CPU_Temp := CPU_Counter; CPU_Counter := CPU_Counter + 1;
   return CPU_Range(CPU_Temp);
end Next_CPU_Nr;

task type TE_Handling_Task (CPU : CPU_Range := Next_CPU_Nr)
   with CPU => CPU, Interrupt_Priority => System.Interrupt_Priority'Last;

procedure Process_Expired_Events(CPU : CPU_Range);

task body TE_Handling_Task is
   Next_Timeout : Time;
begin
   System.Tasking.Utilities.Make_Independent;
   System.Interrupt_Management.Operations.Setup_Interrupt_Mask;
   loop
      Next_Timeout := TE_Queues(CPU).Next_Timeout;
      select -- Wait until the next event expires or the head of the queue is changed
         TE_Queues(CPU).Detect_First_Changed;
      or
         delay until Next_Timeout;
      end select;
      Process_Expired_Events(CPU); -- Process expired events, if any
   end loop;
end TE_Handling_Task;

-- TE Server Tasks
type TE_Server_Tasks_Type is array (0 .. Number_Of_CPUs) of TE_Handling_Task;
TE_Server_Tasks : TE_Server_Tasks_Type;
```

Listing 5. Implementation of `Process_Expired_Events`

```
procedure Process_Expired_Events (CPU : CPU_Range) is
   Next_Event : Any_Timing_Event;
   Is_Expired : Boolean;
   Handler : Timing_Event_Handler;
begin
   loop
      TE_Queues(CPU).Remove_Expired(Next_Event, Is_Expired, Handler);
      exit when not Is_Expired;
      begin -- We have an expired event that has timed out so we will process it.
         if Handler /= null then
            Handler.all (Timing_Event (Next_Event.all));
         end if;
      exception -- Ignore exceptions propagated by handler, as required by RM D.15(21/2).
         when others => null;
      end;
   end loop;
end Process_Expired_Events;
```

Finally, listing 6 shows the new implementation of the `Timing_Event` operations. The implementation of these procedures and functions is analogous to the original GNAT ones, but we have added a new CPU parameter, used for the selection of the timing-event queue associated with that affinity. We have omitted here all the procedures and functions that remain unchanged with respect to GNAT's implementation.

Listing 6. `Timing_Event` **implementation: Operations**

```
   procedure Set_Handler (Event : in out Timing_Event;
                          At_Time : Time;
                          Handler : Timing_Event_Handler;
                          CPU     : CPU_Range := Get_CPU(Current_Task)) is
   begin
      TE_Queues(Event.CPU).Remove (Event'Unchecked_Access);
      Event.Handler := null;
      if Handler /= null then
         Event.Timeout := At_Time;
         Event.Handler := Handler;
         Event.CPU := CPU;
         TE_Queues(Event.CPU).Insert (Event'Unchecked_Access);
      end if;
   end Set_Handler;

   procedure Set_Handler (Event : in out Timing_Event;
                          In_Time : Time_Span;
                          Handler : Timing_Event_Handler;
                          CPU     : CPU_Range := Get_CPU(Current_Task)) is
   begin -- Exactly as the absolute time version, except for:
         -- Event.Timeout := Clock + In_Time;
   end Set_Handler;

   function Current_Handler (Event: Timing_Event)
      return Timing_Event_Handler is (Event.Handler);

   procedure Cancel_Handler (Event : in out Timing_Event;
                             Cancelled : out Boolean) is
   begin
      TE_Queues(Event.CPU).Remove (Event'Unchecked_Access);
      Cancelled := Event.Handler /= null;
      Event.Handler := null;
   end Cancel_Handler;

   function Get_CPU (Event : Timing_Event)
      return CPU_Range is (if Event.Handler = null then Not_A_Specific_CPU else Event.CPU);

   procedure Finalize (This : in out Timing_Event) is
   begin
      This.Handler := null;
      TE_Queues(This.CPU).Remove (This'Unchecked_Access);
   end Finalize;

end Ada_Real_Time.Timing_Events;
```

3.4 Testing the implementation

We have tried this experimental implementation with the code shown in listing 7. The first part shows the handler procedure for all the timing events (protected procedure `Handler_Procedure`). Procedure `TE_Test` is the main unit in this test program, and it defines the `TEWA` (for Timing Events With Affinities) array of timing events , containing as many timing events as CPUs are present in the underlying platform (the `Number_Of_CPUs` is 4 in our case).

After printing the number of processors available and setting the affinity of the main program to one of them, a nested loop sets 16 events to be handled cyclically on the 4 processors.

The execution of this program has produced the output presented in listing 8, which shows that each handler is executed on a different, preset CPU.

Listing 7. Test code for timing events with affinity

```
-- "with" clauses, specs and package headers omitted

protected body Handler_PO is

   procedure Handler_Procedure (Event: in out Timing_Event) is
   begin
      Handled_Count := Handled_Count + 1;
      Put_Line(" Handling event nr" & Natural'Image(Handled_Count) &
                " on CPU" & CPU_Range'Image(Get_CPU));
   end Handler_Procedure;

end Handler_PO;

...

procedure TE_Test is
   TEWA : array (1 .. Number_Of_CPUs) of Timing_Event;
   Next : Time;

begin
   Put_Line("Number of CPUs =" & CPU_Range'Image(Number_Of_CPUs));
   Set_CPU(Number_Of_CPUs - (if Number_Of_CPUs > 1 then 1 else 0));
   Put_Line("Main is running on CPU" & CPU_Range'Image(Get_CPU));
   for I in 1 .. Number_Of_CPUs loop
      for J in 1 .. Number_Of_CPUs loop
         Next := Clock + Milliseconds (1000 + Integer(J) * 250);
         Set_Handler(TEWA(J), Next, Handler_PO.Handler_Procedure'Access, CPU_Range(J));
      end loop;
      delay 3.0;
   end loop;
end TE_Test;
```

Listing 8. Output of the test program

```
Number of CPUs = 4
Main is running on CPU 3
 Handling event nr 1 on CPU 1
 Handling event nr 2 on CPU 2
 Handling event nr 3 on CPU 3
 Handling event nr 4 on CPU 4
 Handling event nr 5 on CPU 1
 Handling event nr 6 on CPU 2
 Handling event nr 7 on CPU 3
 Handling event nr 8 on CPU 4
 Handling event nr 9 on CPU 1
 Handling event nr 10 on CPU 2
 Handling event nr 11 on CPU 3
 Handling event nr 12 on CPU 4
 Handling event nr 13 on CPU 1
 Handling event nr 14 on CPU 2
 Handling event nr 15 on CPU 3
 Handling event nr 16 on CPU 4
```

4 Affinity of Interrupt Handlers

To achieve complete control on the execution of event handlers, it is also important that the programmer was able to determine in which CPU a given interrupt handler must be executed when the interrupt occurs. For example, this feature would allow the system designer to build a mixed-criticality system, where non-critical interrupts are bounded to a given non-critical dispatching domain while critical interrupts are bound to well-known CPUs, decided at design time, within the critical dispatching domain. This way, all the interference due to interrupt handling can be analysed *a priori*, and the temporal behaviour of the system will not be jeopardised by a large number of interrupts arriving at improper CPUs. In that vein, the current interrupt handling API should be extended to cope with interrupt handler affinities. In the following subsection we propose a simple extension to package `Ada.Interrupts` (ARM C.3.2).

4.1 Proposed API

Listing 9 shows a new procedure `Set_CPU` that allows the programmer to determine the CPU where a given interrupt handler has to be executed. In contrast to the case of timing events, this procedure will not incur inefficiencies if the affinity of the interrupt handler is set at a different time than the `Attach_Handler` procedure is invoked, since no queue management needs be done by the underlying implementation. In fact, we think that this `Set_CPU` approach is probably better suited to the expected use of this feature, since setting an interrupt handler's affinity is normally related to the application deployment or initialisation, more than a dynamic attribute to be frequently changed. Moreover, the preferable form for this feature would be by means of a pragma or aspect, e.g. `Interrupt_Affinity`.

Listing 9. Proposed extension to `Ada.Interrupts`

```
package Ada.Interrupts is
   type Interrupt_ID is new System.Interrupts.Ada_Interrupt_ID;
   type Parameterless_Handler is access protected procedure;

   function Is_Reserved (Interrupt : Interrupt_ID) return Boolean;

   function Is_Attached (Interrupt : Interrupt_ID) return Boolean;

   function Current_Handler (Interrupt : Interrupt_ID) return Parameterless_Handler;

   procedure Attach_Handler (New_Handler : Parameterless_Handler;
                             Interrupt  : Interrupt_ID);

   procedure Exchange_Handler (Old_Handler : out Parameterless_Handler;
                               New_Handler : Parameterless_Handler;
                               Interrupt  : Interrupt_ID);

   procedure Detach_Handler (Interrupt : Interrupt_ID);

   function Reference (Interrupt : Interrupt_ID) return System.Address;

   function Get_CPU (Interrupt : Interrupt_ID) return System.Multiprocessors.CPU_Range;

   -- New procedure
   function Set_CPU (Interrupt : Interrupt_ID;
                     CPU       : System.Multiprocessors.CPU_Range);
private
   ...
end Ada.Interrupts;
```

4.2 Implementation on top of Linux

Given that Linux follows a *Separate Address Space* model, the kernel does not allow an application to establish a function located at the user's address space as an interrupt handler[4]. In fact, the interrupt concept is replaced in UNIX-like OS by the signal abstraction that are used to inform an application about user-level *exceptions* and *interrupts*. Consequently, the current implementation of interrupt support in the Ada RTS of AdaCore's GNAT GPL 2014 for Linux is based on POSIX signals management.

Although the existing implementation is rather complex, due to all the POSIX signal management, the execution support for interrupt handlers is based on a similar approach to the server tasks for timing events. In this case, the important matter is that, apart from an *Interrupt Manager* task, every possible interrupt (represented by a POSIX signal) has a server task. This server task is initially created on demand (e.g. the first time `Attach_handler` procedure is invoked for a given `Interrupt_ID`), but it is never detached from its original interrupt. As shown in the following code excerpt from `System.Interrupts`, server tasks are kept in an array called `Server_ID`.

[4] As far as authors know

```
package body System.Interrupts is
   ...
   Server_ID : array (Interrupt_ID'Range) of Task_Id := (others => Null_Task);
   ...
end System.Interrupts;
```

A possible implementation of the new `Set_CPU` procedure could consist in storing the interrupt affinity in a new `Interrupt_CPU` array and setting the server task's affinity (if the task Id already exists in `Server_ID` array), or to modify the corresponding procedures and entries to set the server task's affinity when this task is dynamically created, extracting the interrupt's affinity from the `Interrupt_CPU` array. This new array is shown in listing 10, together with an implementation of the `Get_CPU` function.

Listing 10. Array with affinity of each interrupt handler.

```
package body System.Interrupts is

   ...
   Interrupt_CPU : array (Interrupt_ID'Range) of CPU_Range := (others => Not_A_Specific_CPU);
   ...

   function Get_CPU (Interrupt : Interrupt_ID) return CPU_Range is
   begin
      return Interrupt_CPU (Interrupt);
   end Get_CPU;

end System.Interrupts;
```

5 Conclusions

The ability to determine the affinity for event handlers is a missing feature in Ada (and Ravenscar) and it has an impact on the effectiveness of schedulability analysis: knowing which CPUs are affected by the interference of event handling is of crucial importance in order to use realistic data in such analyses.

In this paper, we have explored the problem in the context of timing-event handlers and, to a lesser extent, interrupt handlers. We have shown that the current implementation of GNAT on Linux can be extended to support timing event affinity. Even though this implementation does not follow the implementation advice that suggests to execute timing-event handlers in the context of the clock interrupt handler, we believe that the proposed implementation demonstrates the feasibility of adding this feature to Ada in the future.

Our next steps will aim at studying other implementations and see how this feature can be implemented for target platforms that are more adequate for hard real-time systems. We are also interested in studying the intersection of this feature with the concept of dispatching domains.

References

[1] ISO/IEC JTC1 SC22 WG 9 Ada Rapporteur Group. *Ada Reference Manual - Language and Standard Libraries - ISO/IEC 8652:2012(E).* `http://www.ada-europe.org/manuals/LRM-2012.pdf`, 2012.

[2] B. W. Kernighan and D. M. Ritchie. *The C Programming Language.* Prentice-Hall Inc., New Jersey, 1978.

[3] S. Sáez, J. Real, and A. Crespo. Reliable Handling of Real-Time Scheduling Attributes on Multiprocessor Platforms in Ada 2012. In L. George and T. Vardanega, editors, *Reliable Software Technologies – Ada-Europe 2014*, volume 8454 of *Lecture Notes in Computer Science*, pages 74–90. Springer, June 2014.