Execution-time control for interrupt handling

Kristoffer Nyborg Gregertsen, Amund Skavhaug
Department of Engineering Cybernetics, NTNU
N-7491 Trondheim, Norway
{gregerts,amund}@itk.ntnu.no

Abstract

This paper proposes that execution-time control features for interrupt handling should be added to the Ada standard library. By measuring the execution-time for interrupts separately the accuracy of task execution-time measurement will be also improved. It is described how the proposed features were implemented for the GNAT bare-board Ravenscar run-time environment on the Atmel AVR32 architecture. Test results for the implementation and an example of usage are presented.

1 Introduction

The execution-time clocks and timers standardized with Ada 2005 allow the execution-time of tasks to be monitored and overruns to be handled [7]. This feature is useful as the worst-case execution-time (WCET) of tasks may be hard to compute [12] and may also be significantly higher than the average execution-time. By utilizing the new execution-time control features it is possible to use less conservative budgets and instead handle execution-time overruns dynamically by methods such as asynchronous control [11] in order to avoid deadline misses. It is also possible to implement execution-time servers using these features [3].

The Ada 2005 standard does not specify which task, if any, that is to be charged the execution-time of interrupt handlers [7]. Most implementations charges the running task the execution-time of the interrupt handler [9]. This leads to inaccuracy in task execution-time measurement, and the worst-case overhead of interrupt handling has to be added to all task budgets as it is not possible to predict which task will be interrupted. This inaccuracy was raised as an issue when implementations of the new Ada 2005 features were evaluated [10].

By separately measuring the execution-time for interrupt priorities this inaccuracy in task execution-time measurement is addressed. This would also allow execution-time control for interrupt handling, thereby making it possible to protect the system from burst of interrupts that could otherwise result in tasks missing their deadlines. This additional safety may be very useful for high-integrity real-time systems. A demonstration of these feature is developed for the Atmel AVR32 architecture [2].

The AVR32 is a brand new architecture developed by Atmel Norway in cooperation with the Norwegian University of Science and Technology (NTNU). The GNU Ada Compiler (GNAT) and a bare-board Ravenscar run-time environment (GNATforLEON) were ported to AVR32 [1] at NTNU [4]. GNATforLEON is based on the Open Ravenscar Kernel developed for the ERC32 and LEON space application processors that was integrated into the GNU Ada Run-time Library (GNARL) by Josẽ F. Ruiz at AdaCore [8]. The version ported to the AVR32 did not include the Ada 2005 real-time features. These features were later implemented in GNATforLEON 1.3 [9]. This implementation charges the running task the execution-time of interrupt handlers.

The Ada 2005 real-time features were implemented from scratch for the AVR32 architecture and the UC3 core [5]. Improved accuracy for execution-time measurement compared to GNATforLEON 1.3 was achieved by using separate execution-time clocks for each interrupt level and accounting for the effects of executing entries by proxy. It should be noted that the Ravenscar profile does not allow execution-time timers. Therefore strict compliance with the profile is broken by including this feature.

In the following there is a short introduction to the AVR32 architecture before the proposed additions to Ada standard library are presented. Next follows a description of how these features were implemented on the AVR32 UC3 core and test results for this implementation. An example of how the features may be used is given. Finally there is a discussion of the value added by the proposed features, the additions to the standard, the AVR32 implementation, and the portability of the features.

2 The AVR32 architecture

The Atmel AVR32 [2] is a 32-bit RISC architecture optimized for high code density and high computational throughput with low power consumption. The register file consists of 13 general purpose registers (R0 to R12), the link register (LR) used for storing the routine return address, the program counter (PC) and the system register (SR). The AVR32 has four interrupt levels, a Non-Maskable Interrupt (NMI) and exceptions.

The UC3 core [1] used in this paper is the second implementation of the AVR32 architecture. It is primarily intended for embedded control applications where deterministic execution-time is paramount. It has a three-stage pipeline integrated with an internal SRAM that bypasses the system bus, allowing deterministic, single-cycle read/write memory access. The core is shown in Figure 1.

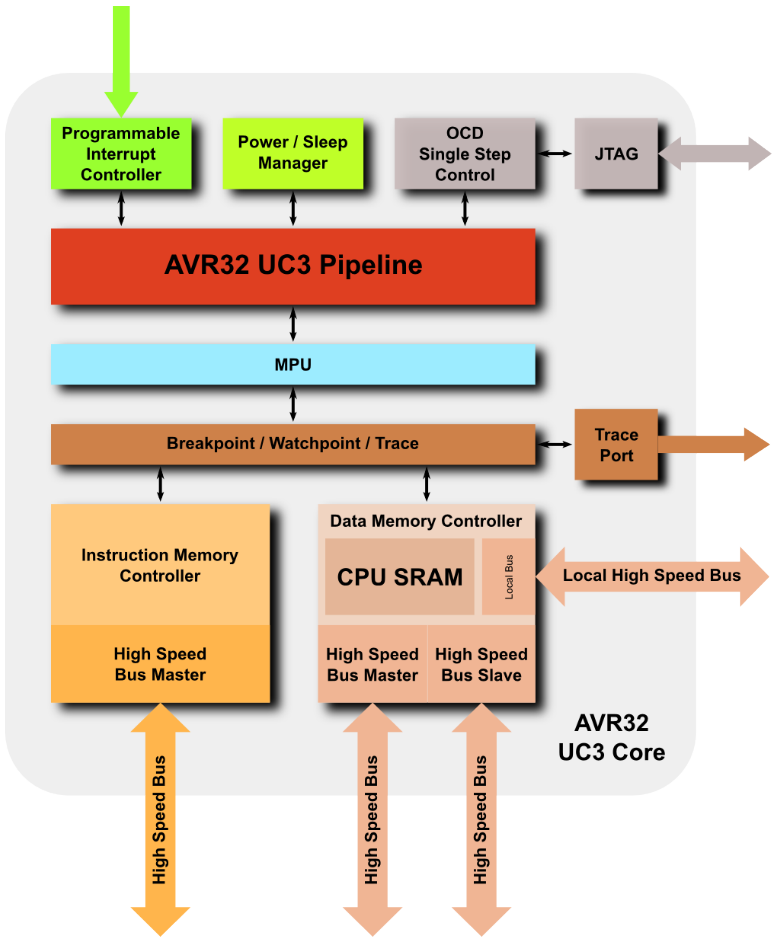

Figure 1: The AVR32 UC3 core

The UC3 has a programmable interrupt controller making it possible to configure the priority of interrupt groups. Prior to entering an interrupt level the UC3 automatically pushes registers R8 through R12, PC, LR and the system register on the stack.

The 32-bit COUNT/COMPARE system registers are used for execution-time measurement. The COUNT register has the value 0 at start-up. It is incremented on every clock cycle of the CPU and overflows silently. The COMPARE register is used for setting off the COMPARE interrupt when COUNT equals COMPARE. The interrupt is disabled when COMPARE has the value 0, which is the start-up value of the register. The interrupt is cleared by writing to the COMPARE register.

3 Standard library modifications

In order to support interrupt execution-time measurement and timer some additions had to be made to Annex D of the Ada 2005 standard [7]. It was chosen to make these changes in the existing execution-time packages specified in D.14 and D.14.1 instead of adding new packages to the standard library.

3.1 Execution-time measurement

In `Ada.Execution_Time` specified in section D.14 a function was added to support execution-time measurement for interrupt priorities:

```
function Interrupt_Clock
  (Priority : System.Interrupt_Priority)
  return CPU_Time;
```

This function returns the total execution-time spent by all interrupt handlers of the given interrupt priority since system start-up. It may be thought of as the execution-time clock of a pseudo task serving all interrupts of that interrupt priority.

3.2 Execution-time timers

In `Ada.Execution_Time.Timers` specified in section D.14.1 a tagged type `Interrupt_Timer` with the interrupt priority as discriminant that inherits `Timer` was defined for supporting execution-time timers for interrupt priorities:

```
Pseudo_Task_Id : aliased constant
  Ada.Task_Identification.Task_Id
  := Ada.Task_Identification.Null_Task_Id;

type Interrupt_Timer
  (I : System.Interrupt_Priority)
   is new Timer (Pseudo_Task_Id'Access) with private;
```

None of the operations of `Timer` are overridden as it is assumed that the same underlying mechanism will be used both for task and interrupt timers and that the only reason for having a separate type for interrupt timers is the difference in the discriminant.

An access to the constant `Pseudo_Task_Id` with the value `Null_Task_Id` is used as the discriminant `T` for all interrupt timers. This violates the Ada 2005 standard which states that a program error is to be raised for all operations on `Timer` if the value of `T.all` is `Null_Task_Id`. It was however needed to do this in order to define the `Interrupt_Timer` type.

4 AVR32 implementation

The execution-time control features were implemented in a new package `System.BB.TMU` (*Time Management Unit*) of the bare-board run-time environment.

The package defines the type `CPU_Time` representing execution-time as a 64-bit modular integer, a limited private type `Timer` used both for execution-time measurement and timers, an access type `Timer_Id` for the pointing to timers, and a procedure access type `Timer_Handler` taking an address as argument for low-level timer handlers. The type `Timer` is defined in the private part of the package as:

```
type Timer is
   record
      Active_TM : Timer_Id;
      Base_Time : CPU_Time;
      pragma Volatile (Base_Time);
      Timeout   : CPU_Time;
      Handler   : Timer_Handler;
      Data      : System.Address;
      Active    : Boolean;
      Acquired  : Boolean;
    end record;
```

Internally the TMU package may be though of as having two layers, a high-level layer managing timers, making sure that the correct timer is always *active*, and a low-level layer performing operations such as swapping timers and measuring execution-time.

4.1 Timer management

There are several possible active timers in the context of the running task since there are separate timers for the pseudo tasks such as the idle task and the interrupt server tasks, and the timer of another task is to be active when executing an entry by proxy. The possible timers in the context of a task and the relation among them are shown in Figure 2. The figure is simplified by only showing an arrow representing interruption of one level by the next one. When traversing the graph the history is kept so that the correct timers may be restored later.

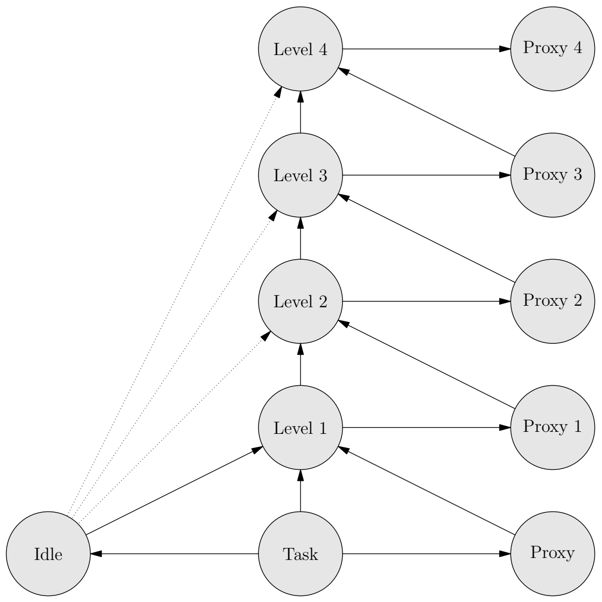

Figure 2: Possible timers of a task

After system initialization there is one and only one active timer at any time indicated by the field `Active` of the timer being set. The first active timer is that of the environment task. If the running task is not executing an interrupt, the idle loop, or an entry by proxy then the timer of the running task is active.

A stack of timer accesses is used for keeping track of the different interrupt level timers. The bottom element of the stack always points to the timer of the running task and is updated when a context switch is executed. The stack is pushed and popped when moving in the vertical direction of Figure 2, entering and leaving interrupt levels.

The field `Active_TM` is used for keeping track of horizontal movement in Figure 2 pointing to another timer when executing an entry by proxy or the idle loop.

4.2 Execution-time measurement

Execution-time is measured as the number of CPU clock cycles used since start-up. The `Base_Time` of a timer used for keeping track of absolute execution-time, also referred to as t_{base}, is initially zero. The execution-time for an active timer is the sum of the base time and the value of the COUNT system register, referred to as c:

$$t = t_{\text{base}} + c, \text{ where } c \in \{0, 1, \ldots, 2^{32} - 1\} \tag{1}$$

When a timer is deactivated its base time is updated:

$$t_{\text{base}} \leftarrow t_{\text{base}} + c \tag{2}$$

The execution-time measurement depends on the CPU counter never overflowing. To prevent this the value C written to COMPARE is never greater than a constant C_{max} chosen to be $2^{31} - 1$. When COUNT equals this value a COMPARE interrupt will be pending causing the timer to be inactivated and its base time updated when the interrupt is handled. The time-line for execution-time measurement is shown in Figure 3.

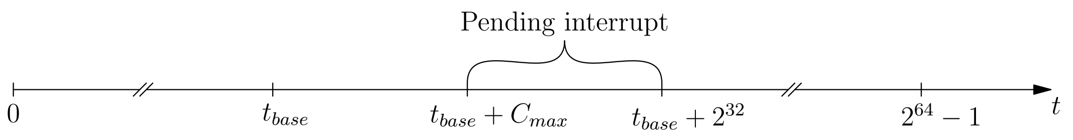

Figure 3: Execution-time for a timer from 0 to the final overflow.

4.3 Execution-time events

Execution-time events for timers are supported by having a `Timeout` field also referred to as t_{timeout} specifying when the timer is to expire, an access to a procedure `Handler` that is to be called when the timer expires, and an address `Data` to be given as an argument when calling the handler.

The COMPARE register is updated with the value C when timers are swapped, an active timer is set or an active and set timer is cancelled. If the timer is cleared C is set to C_{Max}, otherwise C is computed from the difference d between t_{timeout} and t_{base} by the following equation:

$$C = \begin{cases} 1 & \text{if } d < 1, \\ d & \text{if } 1 \leq d \text{ and } d \leq C_{max}, \\ C_{max} & \text{if } d > C_{max} \end{cases} \tag{3}$$

The handler for the COMPARE interrupt has the highest interrupt priority. When executed the timer of the highest interrupt priority is active and on the top of the stack, the timer causing the interrupt is pointed to by `Active_TM` of the timer below on the stack. The handler gets this timer and checks if it has expired as false interrupts may occur due. If expired the timer is cleared and the handler called with `Data` as argument. No further action is taken in the case of a false interrupt.

4.4 CPU counter primitives

Four primitive operations to manipulate the COUNT and COMPARE system registers were added to the architecture specific `CPU_Primitives` package:

- The function `Get_Count` that returns the value of COUNT system register.
- The procedure `Reset_Count` that resets COUNT and sets the COMPARE register to the value C given as argument without causing any false interrupts.
- The procedure `Swap_Count` that performs the same operation as `Reset_Count` but also returns the previous value of COUNT incremented in order to avoid execution-time leakage.
- The procedure `Adjust_Compare` that changes the value of COMPARE to C without losing an interrupt. If the procedure is called with a C that is less than COUNT, the COMPARE interrupt will be pending when leaving the procedure.

All these routines were written as inline assembler code for maximal control and efficiency.

4.5 Application Programming Interface

The tagged type `Timer` visible through the package `Ada.Execution_Time.Timers` includes a `Timer_Id` pointing to the associated kernel timer and a field holding the current handler.

The `Timer_Id` is initialized the first time `Set_Handler` is called. The initialization procedure acquires the kernel timer for an interrupt priority if the timer is in `Interrupt_Timer'Class` or else the timer for the task ID. If the timer is already acquired or not available a `Timer_Resource_Error` is raised.

When the kernel timer is set, the `Data` address argument is the address of the timer object and the `Handler` an access to an internal procedure. When called by the COMPARE handler this procedure type-casts the argument `Data` to an timer access before calling the handler provided by the programmer. This resembles the use of void pointers in C, and may be seen as a breach of Ada programming practice, but was chosen for reasons of simplicity and efficiency.

All operations performed on timers objects are done atomically by using the kernel protection package.

5 Tests

There is a non-zero overhead to the execution-time measurement of a task being interrupted or a task executing an entry by proxy. Test programs were made to measure this overhead.

The Atmel EVK1100 evaluation board with the UC3A0512 microcontroller [1] clocked at 60 MHz was used for the tests. Results were sent on the USART channel from the EVK1100 to a PC where it was retrieved and analyzed. The Atmel JTAG ICE Mk II was used for programming and debugging the UC3.

5.1 Cost of interruption

This program has a single loop executed by the environment task where the execution-time is first measured using `Clock`, then a busy wait for 50 ms is executed before the execution-time is measured again. The two measured execution-times are sent to the PC and the loop is repeated.

The only active interrupt in this program is the real-time clock overflow that will occur at most once between the two measurements of execution-time. The difference between the shortest and longest measured execution-time between the clock calls should give an indication on the overhead of interruption.

When the test was run and 5000 samples were gathered, there were only three unique values of the difference between the clock before and after executing the busy wait: 600050, 600182 and 600183.

This is a testament of the deterministic nature of the AVR32 UC3 microcontroller. It is inferred that the interruption overhead is 133 CPU clock cycles.

5.2 Cost of executing an entry by proxy

This test program has two tasks named *A* and *B* and a protected object. The protected object has an entry with a guard and a procedure that opens that guard. The protected object has the highest priority in order to avoid interruption while executing its operations.

Task *A* has higher priority than *B* and measures its execution-time before and after executing the protected procedure. After executing the procedure, *A* sends its execution-time measurements to the PC before it is delayed until next release. Task *B* will simply call the entry and then be delayed until next release. By releasing *A* after *B* half of the times, *A* will call the procedure both when *B* is blocked on the entry, in which case the entry will be executed by proxy, and when *B* is not blocked.

When running the test gathering 5000 samples, the execution-time between reading the clock was measured to be 387 CPU cycles when no entry by proxy was executed and 610 when the entry was executed, giving a cost of 223 CPU cycles. As with the previous test the unique values are due to the determinism of the AVR32 UC3 microcontroller. However 223 CPU cycles is the best-case cost, the worst-case situation is when task *B* has a handler set long into the future due to the calculation of the COMPARE value *C* for task *B* when swapping timers. When task *B* has a timer set to expire at `CPU_Time'Last`, the cost was measured to be 237 CPU cycles.

6 Application

Extensions were be made to the real-time framework by Andy Wellings and Alan Burns [11] to utilize the interrupt execution-time control features [6].

6.1 Interrupt handling

In order to support more flexible interrupt handling an interface for interrupt states was defined:

```
package Interrupt_States is
   type Interrupt_State is limited interface;
   procedure Handler
     (S : in out Interrupt_State) is abstract;
   procedure Enable
     (S : in out Interrupt_State) is abstract;
   procedure Disable
     (S : in out Interrupt_State) is abstract;
   type Any_Interrupt_State is
     access all Interrupt_State'Class;
end Interrupt_States;
```

Device drivers for peripherals associated with an interrupt may for instance implement procedures for enabling and disabling the interrupt while the application has a type that inherits the device driver and implements the interrupt handler itself.

A protected object for handling interrupts was defined as:

```
protected type Interrupt_Handler
  (Id  : Interrupt_Id;
   Pri : Interrupt_Priority;
   S   : Any_Interrupt_State) is
   pragma Interrupt_Priority (Pri);
private
   procedure Handler;
   pragma Attach_Handler (Handler, Id);
end Interrupt_Handler;
```

The private handler of the protected objects simply executes a dispatching call to the handler of the interrupt state.

6.2 Interrupt servers

Interrupt servers take advantage of the interrupt timers to control the execution-time spent on handling interrupts of a given interrupt priority. An interface for these servers was defined as:

```
package Interrupt_Servers is
   type Interrupt_Server_Parameters is
      record
         Pri : Interrupt_Priority;
         Budget : Time_Span;
         Period : Time_Span;
      end record;
   type Interrupt_Server is synchronized interface;
   procedure Register
     (S : in out Interrupt_Server;
      I : Any_Interrupt_State) is abstract;
   type Any_Interrupt_Server is
     access all Interrupt_Server;
end Interrupt_Servers;
```

The interrupt server parameters give the interrupt priority, budget and replenishing period of the interrupt server. The synchronized interface `Interrupt_Server` has a single operation `Register` for registering interrupts to be controlled.

The type `Deferrable_Interrupt_Server` implements the interface `Interrupt_Server` and was defined as:

```
protected type Deferrable_Interrupt_Server
  (Param : access Interrupt_Server_Parameters)
is new Interrupt_Server with
   procedure Register (I : Any_Interrupt_State);
   pragma Interrupt_Priority (Any_Priority'Last);
private
   procedure Replenish (TE : in out Timing_Event);
   procedure Overran (TM : in out Timer);
   Replenish_Event : Timing_Event;
   Execution_Timer : access Interrupt_Timer;
   Next : Time;
   Disabled : Boolean := True;
   Registered : Natural := 0;
   States : State_Array;
end Deferrable_Interrupt_Server;
```

The interrupt execution-time is controlled by using the interrupt timer of the given level and a timing event replenishing the budget. The first call to `Register` initializes the object and sets the first replenish event. All registered interrupts are disabled on overrun and enabled every time the budget is replenished:

```
procedure Replenish (TE : in out Timing_Event) is
begin
   Execution_Timer.Set_Handler
     (Param.Budget, Overran'Access);
   if Disabled then
      Disabled := False;
      for I in 1 .. Registered loop
         States (I).Enable;
      end loop;
   end if;
   Next := Next + Param.Period;
   TE.Set_Handler (Next, Replenish'Access);
end Replenish;
```

```
procedure Overran (TM : in out Timer) is
begin
   if not Disabled then
      Disabled := True;
      for I in 1 .. Registered loop
         States (I).Disable;
      end loop;
   end if;
end Overran;
```

Interrupts are assumed to be initially disabled and are enabled on the first replenish event.

6.3 Usage

In order to use the interrupt system the programmer could define a tagged type `My_Interrupt` extending the AVR32 device driver `External_Interrupt` which implements the interface `Interrupt_State`. The device driver implements the procedures to enable, disable and clear the external interrupt identified by its discriminant, so `My_Interrupt` only needs to implement the interrupt handler:

```
procedure Handler (S : in out My_Interrupt) is
begin
   S.Clear;
   --  Do work of handler...
end Handler;
```

The interrupt state, an interrupt handler and an interrupt server may be declared as:

```
S_I : aliased My_Interrupt (EIM_5);

H_I : Interrupt_Handler
  (EIM_5, EIM_5_Priority, S_I'Access);

P_I : aliased Interrupt_Server_Parameters :=
  (Pri    => EIM_5_Priority,
   Period => Milliseconds (25),
   Budget => Milliseconds (2));

E_I : Deferrable_Interrupt_Server (P_I'Access);
```

The interrupt state `S_I` needs to be registered to the interrupt server `E_I` when the application is initialized.

7 Discussion

7.1 Interrupt execution-time control

When charging the interrupted task the execution-time of interrupt handlers the WCET of all possible interrupt handler invocations have to be added to the execution-time budget of all tasks as there is no way of predicting how many times each task will be interrupted. This leads to reduced CPU utilization. By not charging the interrupted tasks the execution-time of interrupt handlers the accuracy of execution-time measurement is improved and the task budgets can be tighter. Ideally there should be zero execution-time cost to the interrupted task.

The approach taken by this paper is to charge the execution-time of interrupt handlers to a pseudo task though to execute all interrupt handlers of a given interrupt priority. This allows the execution-time spent on handling interrupts of a given task to be retrieved, which may be useful for debugging purposes, but more importantly it allows execution-time timers for interrupt handling.

By using interrupt timers it is possible to implement execution-time servers for interrupts priorities making it possible to protect the system against bursts of interrupts that would otherwise result in system failure. Such an execution-time server has been implemented in a real-time framework based on work by Andy Wellings and Alan Burns [6, 11]. It should be noted that system interrupts such as the clock interrupt, timing event interrupt and the interrupt for execution-time should never be disabled as this could cause run-time environments to malfunction.

An alternative approach would be to not charge any task the execution-time of interrupt handlers. This would give the same improvement in execution-time measurement for tasks without any additions to the Ada standard library, but would obviously not allow execution-time control for interrupt handling. The additional safety provided by the having execution-time timers for interrupt priorities seems to justify the modifications to the Ada standard library.

7.2 Standard library modifications

The proposed changes to the Ada standard library are in the existing packages `Ada.Execution_Time` and `Ada.Execution_Time.Timers`. Changes were made to existing packages instead of adding new ones since the additions are few and fairly simple. However it might be argued that the interrupt execution-time control features should be isolated in child packages so that architectures were these features are not implementable may simply leave these packages out.

The package `Ada.Execution_Time` is modified by adding a function `Interrupt_Clock` for retrieving the execution-time spent on handling interrupts of the given interrupt priority. The function works in the same way as `Clock` would for an interrupt pseudo task serving all interrupts of priority. An alternative would be to measure the execution-time spent handling each type of interrupt. This approach would require more run-time support, but would be more suitable for multiprocessors where several interrupts of the same priority may be handled at the same time in which case the pseudo task approach taken is this paper would not be possible.

The package `Ada.Execution_Time.Timers` is modified by adding the aliased constant `Pseudo_Task_Id`, and the tagged type `Interrupt_Timer` inheriting `Timer` taking the interrupt priority as discriminant, and using `Pseudo_Task_Id` as the discriminant `T` for `Timer` indicating that it is not bound to an ordinary task. This conflicts with the Ada 2005 standard as the `Pseudo_Task_Id` has the value `Null_Task_Id`. An alternative would be to have a special `Task_Id` defined for each pseudo interrupt server task, and use the type `Timer` for both ordinary task timers and interrupt timers. However this approach was not used as other parts of the run-time environment then would have to include checks for the special `Task_Id`.

7.3 AVR32 implementation

The Ada 2005 real-time features were implemented from scratch on the Atmel AVR32 architecture [5]. The addition of interrupt execution-time control and taking into account the effects of entry-by-proxy execution is the main difference between this implementation and GNATforLEON 1.3 [9].

The AVR32 implementation is not ideal with regards to interrupt execution-time measurement as there is an overhead to the interrupted task due to the timers being switched by software. The cost of interruption by the clock interrupt was found by testing to be 133 CPU cycles. This the best-case cost for general interrupts as the highest level interrupt timer is never set with a handler. The additional cost of being interrupted by an interrupt timer with a handler set long into the future is 14 CPU cycles as observed with execution of entries by proxy, giving a worst-case cost of 147 CPU cycles.

Execution by proxy improves system performance by reducing the number of context switches needed. This implementation always charges the blocked task the execution-time spent on the entry thereby improving the accuracy of the execution-time measurement as this execution-time does not have to be added to the budget of the task executing the entry by proxy. The usefulness of this feature is however highly dependent of the implementation overhead of changing timers compared to the execution-time of the entry. As seen from the test results the worst-case cost of executing an entry by proxy is 237 CPU cycles, which definitely is more than the cost of executing an entry with a null body. Still the benefit of having a known constant worst-case cost instead of an unknown cost when executing entries by proxy seems to justify the overhead.

7.4 Portability and hardware support

The implementation of the proposed interrupt execution-time control features were done on the AVR32 UC3 microcontroller. Unfortunately recent changes to the COUNT / COMPARE semantics [1] for the UC3 core has made the registers less useful for execution timing. The new semantics state that when COUNT equals COMPARE the interrupt line should be asserted and COUNT set to 0. The original semantics were used in this paper since only the engineering presample version was available with the EVK1100 at the time of development. The implementation could be adapted to the new semantics by adding more logic to the CPU primitives.

The bare-board GNAT implementation of the Ada 2005 execution-time features is portable to other architectures as long as it is possible to implement the CPU primitives with fairly low overhead. The features should also be portable to other embedded run-time systems that allow direct control of the timer hardware. In order to port the features to operating systems such as Linux, the kernel would have to be modified to support interrupt execution-time measurement. This would be harder, but fully possible.

8 Conclusion

The main contributions of this work is an extension to the Ada 2005 standard library allowing execution-time control for interrupt handling, and the implementations of these features on the Atmel AVR32 architecture. The benefits of this are twofold:

1. The accuracy for task execution-time measurement is improved as the execution-time of interrupt handlers is not charged the interrupted task. This allows task budgets to be tighter, and thereby higher CPU utilization.

2. It is possible to monitor and control the execution-time spent on handling interrupts of a given priority using execution-time servers. This makes it possible to protect the systems from bursts of interrupts that could otherwise result in tasks missing their deadline.

In the authors opinion the proposed features would be a valuable addition to the Ada programming language and should be particularly useful for high-integrity real-time systems. Therefore the authors recommend that the features are added to the next revision of the Ada standard.

Acknowledgment

Thanks to Alan Burns for giving valuable advice on the Ada 2005 execution-time control features, and to Atmel Norway for providing the needed development hardware.

References

[1] Atmel Corporation. *AT32UC3A Series - Preliminary Datasheet.*

[2] Atmel Corporation. *AVR32 - Architecture Document.*

[3] A. Burns and A.J. Wellings. Programming execution-time servers in Ada 2005. In *Proc. 27th IEEE International Real-Time Systems Symposium RTSS '06*, pages 47–56, Dec. 2006.

[4] K.N. Gregertsen and A. Skavhaug. An efficient and deterministic multi-tasking run-time environment for ada and the ravenscar profile on the atmel avr ®32 uc3 microcontroller. In *Design, Automation & Test in Europe Conference & Exhibition, 2009. DATE '09.*, pages 1572–1575, April 2009.

[5] Kristoffer Nyborg Gregertsen and Amund Skavhaug. Implementing the new Ada 2005 timing event and execution-time control features on the AVR32 architecture. Submitted to Journal of Systems Architecture, February 2009.

[6] Kristoffer Nyborg Gregertsen and Amund Skavhaug. A real-time framework for ada 2005 and the ravenscar profile. In *Software Engineering and Advanced Applications, 2009. SEAA '09. 35th Euromicro Conference on*, pages 515–522, Aug. 2009.

[7] ISO/IEC. *Ada Reference Manual - ISO/IEC 8652:1995(E) with Technical Corrigendum 1 and Amendment 1.*

[8] José F. Ruiz. GNAT pro for on-board misson-critical space applications. *Ada-Europe*, 2005.

[9] Santiago Urue na, José Pulido, José Redondo, and Juan Zamorano. Implementing the new Ada 2005 real-time features on a bare board kernel. *Ada Lett.*, XXVII(2):61–66, 2007.

[10] Andy Wellings. Implementation experience with Ada 2005. *Ada Lett.*, XXVII, no 2:59–60, 2007. session report.

[11] Andy Wellings and Alan Burns. *Ada-Europe 2007*, chapter Real-Time Utilities for Ada 2005, pages 1–14. Springer Berlin / Heidelberg, 2007.

[12] Reinhard Wilhelm et al. The worst-case execution-time problem—overview of methods and survey of tools. *Trans. on Embedded Computing Sys.*, 7(3):1–53, 2008.