Programming Execution-Time Servers in Ada 2005*

A. Burns and A.J. Wellings
Real-Time Systems Research Group, Department of Computer Science
University of York, UK

Abstract

Much of the research on scheduling schemes is prevented from being used in practice by the lack of implementations that provide the necessary abstractions. An example of this is the support of execution-time servers. Apart for a single mechanism (the Sporadic Server), which is defined in the POSIX standard, these important building blocks are not available to the system developer. Over the last few years, we have been developing the mechanisms necessary to construct execution-time servers from within an Ada context. Versions of these have now been incorporated in the Ada 2005 standard. In this paper, we show how the mechanisms can be used to construct the Deferrable and Sporadic servers.

1. Introduction

Over the last two decade or more, a considerable volume of literature has been produced which addresses the issues and challenges of real-time scheduling. The focus of much of this research has been on how to effectively support a collection of distinct applications – each with a mixture of periodic and non-periodic, and hard and soft, activities. This work is seen as an enabling technology for a wide range of applications from multi-media to robust control. Notwithstanding the quality of individual contributions, it is unfortunately true that system implementors have not, in general, taken up these results. There are a number of reasons for this, including the normal inertia associated with technical change, but we would highlight the following.

- Inconsistency in the literature – no single approach has emerged as being widely applicable, indeed there is not even any consensus over the right simulation models to use for evaluation.
- Limitations of the schedulability analysis models – too many models have unrealistic assumptions such as ignoring run-time overheads or assuming that the WCET is known for all tasks (or not recognizing that actual execution times can be much smaller than WCET values).
- Difficulty of implementation – often the scheduling scheme will require operating system primitives that are not available on any commercial platform.
- Lack of computational model – the scheduling results are not tied back to realistic models that applications can use to construct systems.
- Lack of design patterns that applications programmers can adopt and adapt.

The work presented in this paper focuses on the final three problems of this group.

Over the last few years, we have been investigating the appropriate abstractions that are needed to support the development of flexible real-time systems [7, 8, 6, 18, 10, 9]. Working in conjunction with the Series of International Workshops on Real-Time Ada Issues - IRTAW[1], we have developed a set of mechanisms that allow a wide range of flexible real-time systems to be implemented. These have now been incorporated into the Ada 2005 standard [5]. In this paper, we will show how these facilities allow flexible scheduling schemes to be programmed and analysed, and code patterns developed.

The challenges of real-time scheduling are easy to (informally) define: hard deadlines must always be met, distinct applications (or subsystems of the same application) must be isolated from each other but any spare capacity (typically the CPU resource) must be used to maximise the overall utility of the applications. The 'extra' work that could be undertaken typically exceeds the spare capacity available, and hence the scheduler must decide which non-hard task to execute at any time. As processors become more complex and the analysis of worst-case execution time more pessimistic, the amount of spare capacity will increase. Indeed, the total utilisation of the hard tasks will often be much less than 50%. There is continuing pressure on applications to minimise their use of hard tasks and to exploit flexible schedul-

* This work has been undertaken within the context of the EU ARTIST2 and the UK EPSRC DIRC projects.

1 Proceeding published in Ada Letters, Vol XXI, March 2001, Vol XXII, December 2002 and Vol XXIII, December 2003.

ing to deliver dynamic behaviour. A real-time system with this mixture of hard and aperiodic tasks has to ensure that:

- all hard real-time tasks meet their deadlines, even when worst-case conditions are being experienced;
- failed components do not impact on non-failed applications by, for example, denial of service;
- all aperiodic tasks have a good response time – here, good should mean that the tasks meet their soft deadlines in most of their invocations;
- other soft tasks exhibit good quality of output by gaining access to the maximum computation time available before their soft/firm deadlines.

One of the key building blocks for delivering this form of flexible scheduling is the use of *execution-time servers* [14, 17, 13, 4, 3]. Servers are in some senses virtual processors, they provide their clients with a budget that has a defined 'shelf life', and a means of replenishing the budget in a predictable and analysable way. The servers collectively must sustain their budgets (i.e. allow the full budget to be available to clients), whilst the clients must ensure that the budget is sufficient. A number of papers have addressed the scheduling issues associated with server-based systems. In this paper we are concerned with programming servers – or more precisely defining the language primitives that allow server abstractions to be built.

The rest of the paper is structured as follows. The Ada 2005 language is introduced in the next section; section 3 shows a framework for programming execution-time servers and gives examples of its use. Related work is briefly considered in section 4, and the last section presents the future work and conclusions.

2. Ada 2005

Ada 2005[5] is the recent amendment to the Ada 95 standard[2]. It includes many additional features across a range of programming topics. Of particular interest here are the new features that have been added to support the programming of real-time systems with requirements that include high performance, flexibility and run-time protection.

The Ada 95 version of the language already defines a number of expressive concurrency features that are effective for the real-time domain. Features include:

- Tasks – the basic unit of concurrency; dynamic or static creation and flat or hieratical relationships between tasks are supported.
- Absolute and relative delay statements.
- Rendezvous – a synchronous means of communication between tasks.
- Asynchronous Transfer of Control (ATC) – an asynchronous means of affecting the behaviour of other tasks.
- Protected Types – a monitor-like object that enforces mutual exclusion over its operations, and which supports condition synchronisation via a form of guarded command.
- Requeue – a synchronisation primitive that allows a guarded command to be prematurely terminated with the calling task placed on another guarded operation; this significantly extends the expressive power of the synchronization primitive without jeopardizing efficiency.
- Exceptions – a means of abandoning the execution of a sequential program segment.
- Controlled objects – an OOP feature that allows the execution of code when objects enter and exit the scope of their declaration.
- Fixed priority dispatching – a standard implementation including ceiling priority protection against unbounded priority inversion.

What was missing from the Ada 95 model was explicit support for resource management and for other dispatching policies such as EDF and Round Robin. In this paper, we concentrate on the feature that have been included into Ada 2005 for resource control and management. However it should also be noted that Ada 2005 now supports: the EDF and Round Robin dispatching policies (together with Baker's algorithm[2] as a generalisation of the priority ceiling protocol); preemptive and non-preemptive dispatching; multiple dispatching policies (executing in a coherent way); the notion of *deadline* as a first class abstraction; the Ravenscar Profile (a subset of the tasking model) for high integrity real-time applications; and the use of task, protected and synchronized interfaces (which integrate the language's OOP model with the tasking model).

In the following subsection we introduce four important new features of Ada - *timing events*, *execution time clocks*, *timers* and *group budgets*.

2.1. Timing Events

Ada 2005 has introduced a new abstraction of a timing event to allow code to be executed at specified times without the need to employ a task/thread. These events are like interrupts but are generated by the progression of the system clock. Associated with a timing event is a handler that

2 The authors of this paper have been involved in the inclusion of real-time support into the Ada Standard. The features reported in this paper are the result of language design research aimed at improving the expressive power and ease of use of real-time primitives within procedural programming languages.

is executed at the allotted time. An implementation may execute this handler directly from the interrupt handler for the clock device. This leads to a very efficient scheme. The following standard library package is defined by Ada 2005:

```
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);
  procedure Set_Handler(Event : in out Timing_Event;
     In_Time: Time_Span;
     Handler: Timing_Event_Handler);
  function Is_Handler_Set(Event : Timing_Event)
     return Boolean;
  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;
private -- Not specified by the language.
end Ada.Real_Time.Timing_Events;
```

This package provides an API to the necessary abstraction for a timing event. The handler to be executed, when the associated time is reached, is a protected procedure that is passed the timing event as a parameter when the handler is called. This event is a tagged type and hence can be extended by the application (an example of this is given later). The term 'protected' implies it can be executed safely in a concurrent program (by the use of a ceiling protocol). Most execution-time server abstractions require budgets to be replenished at fixed points in time – timing events will be used to implement this requirement efficiently.

Many of the new real-time features introduced into Ada 2005 take the form of event handling and have a similar structure.

2.2. Execution Time Clocks

In hard real-time systems, it is essential to monitor the execution times of all tasks and detect situations in which the estimated WCET is exceeded. This detection was usually available in systems scheduled with cyclic executives, because the periodic nature of its cycle makes it easy to check that all initiated work had been completed by end of each cycle. In event-driven concurrent systems, the same capability should be available, and this can be accomplished with execution time clocks and timers. In addition, many flexible real-time scheduling algorithms require the capability to measure execution time and be able to perform scheduling actions when a certain amount of execution time has been consumed.

Ada 2005 directly supports execution time clocks for tasks, which includes timers that can be fired when tasks have used defined amounts of execution times:

```
with Ada.Task_Identification, Ada.Real_Time;
with Task_Identification, use Ada.Real_Time;
package Ada.Execution_Time is -- not all features shown
   type CPU_Time is private;
   CPU_Tick : constant Time_Span;

   function Clock(T : Task_ID := Current_Task)
     return CPU_Time;

   function "+"  (Left : CPU_Time; Right : Time_Span)
     return CPU_Time;
   function "+"  (Left : Time_Span; Right : CPU_Time)
     return CPU_Time;
   -- similar definitions for -, <=, > and >=, and
   -- other subprograms, not relevant here

   function Time_Of (SC : Seconds_Count;
     TS : Time_Span) return CPU_Time;
private -- Not specified by the language.
end Ada.Execution_Time;
```

The execution time of a task, or CPU time as it is commonly called, is the time spent by the system executing the task and services on its behalf. The clock is set to zero when the task is created and it then monotonically increases as the task executes. The accuracy of the measurement of execution time cannot be dictated by the language specification, it is heavily dependent on the run-time support software. On some implementation (perhaps on non real-time operating systems), it may not even be possible to support this package. But if the package is supported, the range of `CPU_Time` must be at least 50 years and the granularity of the clock (`CPU_Tick`) should be no greater than 1ms.

The following code shows how a periodic task is constructed. In it the task's deadline is used to interrupt the task, and on each iteration the execution time of the task is output. The predefined routines `Set_Deadline` and `Get_Deadline` allow a task to set and read its own deadline.

```
task body Periodic_Task is
  Interval : Time_Span := Milliseconds(30);
  -- the period of the task
  Rel_Deadline : Time_Span := Milliseconds(20);
  -- the relative deadline of the task
  Next : Ada.Real_Time.Time;
  CPU,CPU2 : Ada.Execution_Time.CPU_Time;
  Used : Ada.Real_Time.Time_Span;
begin
  Next := Ada.Real_Time.Clock;
  CPU := Ada.Execution_Time.Clock;
  Set_Deadline(Next+Rel_Deadline);
  loop
    select
      delay until Get_Deadline;
    then abort
      -- application code
    end select;
    CPU2 := Ada.Execution_Time.Clock;
    Used := CPU2-CPU;
    Print(Used);
    CPU := CPU2;
    Next := Next + Interval;
    Set_Deadline(Next+Rel_Deadline);
    delay until Next;
  end loop;
end Periodic_Task;
```

Note the use of 'select then abort' statements (called ATC – asynchronous transfer of control).

The application code executes until the deadline of the task is reached (obtained by calling `Get_Deadline`); the code is then aborted and the rest of the task is executed. In this example, the CPU time is calculated and output, the next release time is set and the next deadline computed before the task suspends itself. Note that the last two statements can be combined to remove a wasteful context switch (see the revision of this example in section 2.3).

This example illustrates termination; i.e. when the deadline is reached the code is abandoned. It is also possible, using a separate protected object, to catch the deadline miss but allow the task to continue to execute (perhaps with a lower priority, or new, longer, deadline).

2.3. Execution Time Timers

As well as monitoring a task's execution time, it is also useful to trigger an event if the clock gets to some specified value. Often this is an error condition, where the task has executed for longer than was anticipated. A child package of `Execution_Time` provides support for this type of event:

```
package Ada.Execution_Time.Timers is

   type Timer(T : not null access constant
      Ada.Task_Identification.Task_ID) is tagged
      limited private;
   type Timer_Handler is access protected
      procedure(TM : in out Timer);
   Min_Handler_Ceiling : constant
     System.Any_Priority := <Implementation Defined>;

   procedure Set_Handler(TM: in out Timer;
     In_Time : Time_Span; Handler : Timer_Handler);
   procedure Set_Handler(TM: in out Timer;
     At_Time : CPU_Time;  Handler : Timer_Handler);
   procedure Cancel_Handler(TM : in out Timer);
   function Current_Handler(TM : Timer)
     return Timer_Handler;

   function Time_Remaining(TM : Timer)
     return Time_Span;

   Timer_Resource_Error : exception;
private -- Not specified by the language.
end Ada.Execution_Time.Timers;
```

An execution time timer is a 'one-shot' event. It identifies code to be executed when the task CPU time clock reaches a specified value. Obviously it cannot reach that value again, so the handler needs to be reset for any further requirement.

To illustrate its use, the earlier code fragment is changed to stop the task if it executes for more than its WCET:

```
protected WCET_Protect is
  entry Overrun;
  procedure Overrun_Handler(T : in out Timer);
private
  Barrier : Boolean := False;
end WCET_Protect;

protected body Overrun_Handler is
  entry Overrun when Barrier is
  begin
    Barrier := False;
  end Barrier;

  procedure Overrun_Handler(T : in out Timer) is
  begin
    Barrier := True;
  end Overrun_Handler;
end Overrun_Handler;

task body Periodic_Task is
  Interval : Time_Span := Milliseconds(30);
  Rel_Deadline : Time_Span := Milliseconds(20);
  Next : Ada.Real_Time.Time;
  CPU : Ada.Execution_Time.CPU_Time;
  Me : access constant Task_Id :=
                          Current_Task'Access;
  WCET_Timer : Timer(Me);
  begin
    Next := Ada.Real_Time.Clock;
    CPU := Ada.Execution_Time.Clock;
    Set_Deadline(Next+Rel_Deadline);
    loop
      Set_Handler(WCET_Timer,WCET,
        WCET_Protect.Overrun_Handler'Access);
      select
        WCET_Protect.Overrun;
        -- action to deal with WCET overrun here
      then abort
        select
          delay until Get_Deadline;
        then abort
          -- application code
        end select;
      end select;
      Next := Next + Interval;
      Delay_and_Set_Deadline(Next,Rel_Deadline);
    end loop;
  end Periodic_Task;
```

Note, the outer 'select-then-abort' statement is triggered by the entry call becoming open, whereas the inner one is triggered by the passage of time.

2.4. Group Budgets

The support for execution time clocks allows the CPU resource usage of individual tasks to be monitored, and the use of timers allows certain control algorithms to be programmed but again for single tasks. There are, however, situations in which the resource usage of groups of tasks needs to be managed. Typically, this occurs when distinct subsystems are programmed together but need to be protected from one another – no failures in one subsystem should lead to failures in the others. So even if a high priority task gets into an infinite loop, tasks in other subsystems should still meet their deadlines. This is usually achieved with the use of execution-time servers.

Ada 2005 does not directly support servers. But it does provide the primitives from which servers can be programmed. Group budgets allows different servers to be

implemented. Note, servers can be used with fixed priority or EDF scheduling.

A typical server has a budget and a replenishment period. At the start of each period, the available budget is restored to its maximum amount. Unused budget at this time is discarded. To program a server requires timing events to trigger replenishment and a means of grouping tasks together and allocating them an amount of CPU resource. A standard package (a child of `Ada.Execution_Time`) is defined to accomplish this:

```
package Ada.Execution_Time.Group_Budgets is
  type Group_Budget is tagged limited private;

  type Group_Budget_Handler is access
       protected procedure(GB : in out Group_Budget);

  type Task_Array is array(Positive range <>) of
    Ada.Task_Identification.Task_ID;

  Min_Handler_Ceiling : constant
    System.Any_Priority := <Implementation Defined>;

  procedure Add_Task(GB: in out Group_Budget;
    T : Ada.Task_Identification.Task_ID);
  -- other subprograms not relevant here

  function Members(GB: Group_Budget)
    return Task_Array;

  procedure Replenish(GB: in out Group_Budget;
    To : Time_Span);
  procedure Add(GB: in out Group_Budget;
    Interval : Time_Span);
  function Budget_Remaining(GB: Group_Budget)
    return Time_Span;

  procedure Set_Handler(GB: in out Group_Budget;
    Handler : Group_Budget_Handler);

  Group_Budget_Error : exception;
private --  not specified by the language
end Ada.Execution_Time.Group_Budgets;
```

The type `Group_Budget` represents a CPU budget to be used by a group of tasks.

The budget decreases whenever a task from the associated set executes. The accuracy of this accounting is again implementation defined. To increase the amount of budget available, two routines are provided. The `Replenish` procedure sets the budget to the amount of 'real-time' given in the `To` parameter. It replaces the current value of the budget. By comparison, the `Add` procedure increases the budget by the `Interval` amount. But as this parameter can be negative, it can also be used to, in effect, reduce the budget.

The minimal budget that can be held in a `Group_Budget` is zero – represented by `Time_Span_Zero`. If a budget is exhausted, or if `Add` is used to reduce the budget by an amount greater than its current value then the lowest the budget can get is zero.

A handler is associated with a group budget by use of the `Set_Handler` procedure. There is an implicit event associated with a `Group_Budget` that occurs whenever the budget goes to zero. If at that time there is a non-null handler set for the budget, the handler will be executed[3]. Furthermore, the handler is permanently linked to the budget (unless it is changed or cancelled). So every time the budget goes to zero the handler is fired.

As with timers, there is a need to define the minimum ceiling priority level for the protected object linked to any group budget handler.

We will shortly give some examples to illustrate the use of group budgets. But it is important that one aspect of the provision is clear.

- When the budget is zero the associated tasks *continue* to execute.

If action should be taken when there is no budget, this has to be programmed (it must be instigated by the handler). So group budgets are not in themselves an execution-time server abstraction – but they allow these abstractions to be constructed.

3. Implementing Execution-Time Servers

In this section, we show a framework for implementing execution-time servers and then instantiate that framework to implement two well known servers: the deferrable server and the sporadic server.

The following package specification defines the common types and interface for all periodic servers.

```
with Ada.Real_Time, System, Ada.Task_Identification;
use  Ada.Real_Time, System, Ada.Task_Identification;
package Execution_Time_Servers is
  type Server_Parameters_Basic is tagged record
    Period : Time_Span;
    Budget : Time_Span;
  end record;

  type Server_Parameters is new
       Server_Parameters_Basic with record
    Foreground_Pri : Priority;
    Background_Pri : Priority;
  end record;

  type Server is synchronized interface;
  procedure Register(ES: in out Server; T :
                     Task_Id := Current_Task) is abstract;

  type Any_Server is access all Server'Class;
end Execution_Time_Servers;
```

All servers have parameters that determine the servers characteristics. There are many different types of servers, but they all have

- a budget – how much CPU time has been allocated;

3 This will also occur if the budget goes to zero as a result of a call to `Add` with a large enough negative parameter.

- a period – how often the server's budget is replenished.

This is represented by the `Server_Parameters_Basic` type.

Some servers allow their client tasks to have different priorities, most require them to have the same priority. Some servers, suspend their clients when their budgets expire[4], others set their priorities to a background value. In this paper, we will show the infrastructure for those with a single foreground priority and setting a background priority on budget exhaustion. This is represented by the `Server_Parameters` type.

All execution servers require their clients to register. Here, any task can register any other tasks.

3.1. Deferrable Server

In this section we show how the deferrable server can be constructed. The following package specification illustrates the approach.

```
with System, Ada.Real_Time, Ada.Task_Identification,
     Ada.Real_Time.Timing_Events,
     Ada.Execution_Time.Group_Budgets,
     Execution_Time_Servers;
-- use clauses for all the above omitted
package Deferrable_Servers is
   protected type Deferrable_Server
        (Params : access Server_Parameters) is new
        Server with
      pragma Interrupt_Priority(Interrupt_Priority'Last);
      overriding procedure Register(
         T : Task_Id := Current_Task);
   private
      procedure Timer_Handler(E : in out Timing_Event);
      procedure Group_Handler(G : in out Group_Budget);
      T_Event : Timing_Event;
      G_Budget : Group_Budget;
      First : Boolean := True;
   end Deferrable_Server;
end Deferrable_Servers;
```

A deferrable server in Ada is a passive entity that is constructed as a protected type, which implements the `Server` interface and overrides the `Register` subprogram. It uses the new Ada 2005 mechanisms: group budgets are used to keep track of the registered tasks' CPU time consumption and timing events are used to signal the replenishment periods. For these reasons, the ceiling priority of the type is set to the highest interrupt priority level (as the `Timer_Handler` procedure is potentially called from the clock interrupt).

To illustrate the use of the server, consider the following:

```
Control_Params : aliased Server_Parameters := (
  Period => Milliseconds(10), Budget => Microseconds(1750),
  Foreground_Pri => 12, Background_Pri => Priority'First);

Con : Deferrable_Server(Control_Params'Access);
```

4 This can be done in Ada using the operations `Hold` and `Continue` from the predefined package `Asynchronous_Task_Control`.

In the above, first some server parameters are declared, and then an instance of the deferrable server is declared that references these parameters. Now, a client aperiodic task can be included:

```
task Aperiodic_Task is
  pragma Priority(Default_Priority);
  -- only used to control initial execution
end Aperiodic_Task;

task body Aperiodic_Task is
  -- local data
begin
  Con.Register;
  loop
    -- wait for next invocation
    -- undertake the work of the task
  end loop;
end Aperiodic_Task;
```

In this simple example, the task is tied to a particular server object; however, it can have just as easily taken an access parameter in a task discriminant if the task needed to be parameterized. The body of the deferrable server follows:

```
with Ada.Dynamic_Priorities, Ada.Task_Identification;
use Ada.Dynamic_Priorities, Ada.Task_Identification;
package body Deferrable_Servers is
  protected body Deferrable_Server is
    procedure Register(T : Task_Id := Current_Task) is
    begin
      if First then
        First := False;
        G_Budget.Add(Params.Budget);
        T_Event.Set_Handler(Params.Period,
          Timer_Handler'Access);
        G_Budget.Set_Handler(Group_Handler'Access);
      end if;
      G_Budget.Add_Task(T);
      if G_Budget.Budget_Has_Expired then
        Set_Priority(Params.Background_Pri, T);
      else
        Set_Priority(Params.Foreground_Pri, T);
      end if;
    end Register;

    procedure Timer_Handler(E : in out Timing_Event) is
      T_Array : Task_Array := G_Budget.Members;
    begin
      G_Budget.Replenish(Params.Budget);
      for I in T_Array'range loop
        Set_Priority(Params.Foreground_Pri,T_Array(I));
      end loop;
      E.Set_Handler(Params.Period,Timer_Handler'Access);
    end Timer_Handler;

    procedure Group_Handler(G : in out Group_Budget) is
      T_Array : Task_Array := G_Budget.Members;
    begin
      for I in T_Array'range loop
        Set_Priority(Params.Background_Pri,T_Array(I));
      end loop;
    end Group_Handler;
  end Deferrable_Server;
end Deferrable_Servers;
```

The first client task to call the server assigns the required budget and sets up the two handlers: one for a group bud-

get and one for a timing event. The timing event is used to replenish the budget (and reset the priority of the client tasks), and a group budget is used to lower the priority of the client tasks. When a task registers (or is registered by another task) it is running outside the budget, so it is necessary to check if the budget is actually exhausted during registration. If it is, the priority of the task must be set to the low value.

One of the advantages of the deferrable server is that it need not have any detailed knowledge of its clients after they have registered. Other servers require to know explicitly when there clients are executing as they have different replenishment policies. The sporadic server is an example, and is now considered.

3.2. Sporadic Server

To programme the Sporadic Server also requires the use of timing events and group budgets. In the following there is a single server for each sporadic task, so strictly speaking it does not require a group budget and an execution-time timer could be used instead. However, the example can be expanded to support more than one task, and even for one task, the server is easier to construct with a group budget.

As the sporadic server replenishes at times related to when the sporadic task is release, the server can be combined with the release mechanism. The sporadic task, therefore, has the following structure:

```
task body Sporadic_Task is
begin
  Sporadic_Controller.Register;
  -- any necessary initialisations etc
  loop
    Sporadic_Controller.Wait_For_Next_Invocation;
    -- undertake the work of the task
  end loop;
end Sporadic_Task;
```

The rule for the Sporadic Server are as follows (following the POSIX standard).

- If there is adequate budget, a task that arrives at time t and executed for time c will result in capacity c being returned to the server at time $t + T$ — where T is the 'period' of the server.
- If there is no budget at time t then the calling task is delayed until budget becomes available, say at time s; this capacity is returned at time $s + T$.
- If there is some budget available x (with $x < c$) then the task will immediately use this budget (and it will be returned at time $t + T$); later when at time s, say, further adequate budget becomes available then the task will continue and $c - x$ will be returned at time $s + T$.
- If s is before task has used initial budget then when it finishes, all of c is returned at time $t + T$.

In addition, we assume that the client tasks do not suspend themselves during their execution.

Each time the task calls its server, the amount of computation time it used last time must be noted and replenished at the appropriate time. To do this (and block the task for its release event) requires the use of Ada's requeue mechanism. Although there is only one task, it may execute a number of times (using less than the budget each time), and hence there can be more than one timing event outstanding. To enable a single handler to deal with all of these requires the timing event to be extended to include the amount of budget that must be returned. We will use a dynamic algorithm that defines a new timing event every time a replenish event should occur. The full specification for the server is as follows.

```
with System, Ada.Real_Time, Ada.Task_Identification,
  Ada.Real_Time.Timing_Events, Execution_Time_Servers,
  Ada.Execution_Time.Group_Budgets;
-- use the above
package Sporadic_Servers is
  type Budget_Event is new Timing_Event with record
    Bud : Time_Span;
  end record;
  type Bud_Event is access Budget_Event;

  protected type Sporadic_Server
      (Params : access Server_Parameters) is new
        Server with
    pragma Interrupt_Priority (Interrupt_Priority'Last);
    overriding procedure Register(
               T : Task_Id := Current_Task);
    entry Wait_For_Next_Invocation;
    procedure Release_Sporadic;
  private
    procedure Timer_Handler(E : in out Timing_Event);
    procedure Group_Handler(G : in out Group_Budget);
    entry Wait_For;
    TB_Event : Bud_Event;
    G_Budget : Group_Budget;
    Start_Budget : Time_Span;
    Release_Time : Time;
    ID : Task_ID;
    Barrier : Boolean := False;
    Task_Executing : Boolean := True;
  end Sporadic_Server;
end Sporadic_Servers;
```

An instance can be easily declared:

```
Sporadic_Controller : Sporadic_Server(P'Access);
   -- for some appropriate parameter object P
```

The full implementation is shown below:

```
with Ada.Dynamic_Priorities, Ada.Task_Identification;
use Ada.Dynamic_Priorities, Ada.Task_Identification;
package body Sporadic_Servers is
  protected body Sporadic_Server is
    procedure Register(T : Task_Id := Current_Task) is
    begin
      ID := T;
      G_Budget.Add_Task(ID);
      G_Budget.Add(Params.Budget);
      G_Budget.Set_Handler(Group_Handler'Access);
      Release_Time := Clock;
      Start_Budget := Params.Budget;
    end Register;
```

```
    entry Wait_For_Next_Invocation when True is
    begin
      -- work out how much budget used, construct
      -- the timing event and set the handler
      Start_Budget := Start_Budget -
        G_Budget.Budget_Remaining;
      TB_Event := new Budget_Event;
      TB_Event.Bud := Start_Budget;
      TB_Event.Set_Handler(Release_Time+Params.Period,
        Timer_Handler'Access);
      Task_Executing := False;
      requeue Wait_For with abort;
    end Wait_For_Next_Invocation;

    entry Wait_For when Barrier is
    begin
      if not G_Budget.Budget_Has_Expired then
        Release_Time := Clock;
        Start_Budget := G_Budget.Budget_Remaining;
        Set_Priority(Params.Foreground_Pri,ID);
      end if;
      Barrier := False;
      Task_Executing := True;
    end Wait_For;

    procedure Release_Sporadic is
    begin
      Barrier := True;
    end Release_Sporadic;

    procedure Timer_Handler(E : in out Timing_Event) is
      Bud : Time_Span;
    begin
      Bud := Budget_Event(Timing_Event'Class(E)).Bud;
      if G_Budget.Budget_Has_Expired and
          Task_Executing then
        Release_Time := Clock;
        Start_Budget := Bud;
        G_Budget.Replenish(Bud);
        Set_Priority(Params.Foreground_Pri,ID);
      elsif not G_Budget.Budget_Has_Expired and
          Task_Executing then
        G_Budget.Add(Bud);
        Start_Budget := Start_Budget + Bud;
      else
        G_Budget.Add(Bud);
      end if;
    end Timer_Handler;

    procedure Group_Handler(G : in out Group_Budget) is
    begin
      -- a replenish event required for the used budget
      TB_Event := new Budget_Event;
      TB_Event.Bud := Start_Budget;
      TB_Event.Set_Handler(Release_Time+
        Params.Period,Timer_Handler'Access);
      Set_Priority(Params.Background_Pri,ID);
      Start_Budget := Time_Span_Zero;
    end Group_Handler;
  end Sporadic_Server;
end Sporadic_Servers;
```

To understand how this algorithm works, consider a sporadic server with budget of 4ms and a replenishment interval of 20ms. Figure 1 shows the execution of an example task with our implementation using AdaCore's evolving Ada 2005 compiler. The system currently runs in a simulated real-time mode, as group budgets are not implemented yet (our simulator works at the milliseconds level).

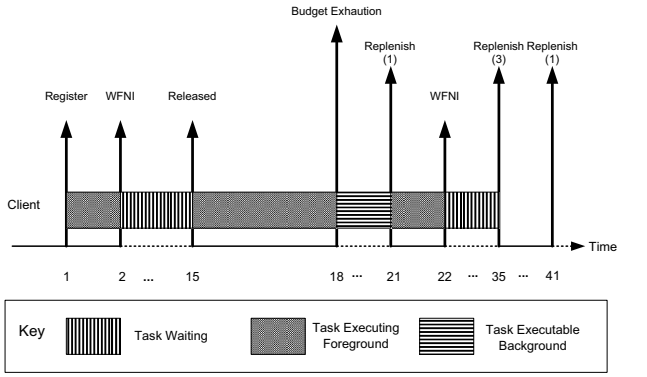

Figure 1. Sporadic Server Illustration 1

The task first calls `Register`; this sets up the group budget, adds the task to this group budget and notes the time of the registration (in our example, the registration takes place at time 1). It also notes that its starting budget is 4ms. After the execution of other initialisation activities, the task will call in to await its release event. This initial phase of execution takes 1ms. Within `Wait_For_Next_Invocation` (WFNI in the Figure 1), a timing event is constructed that triggers at time 21 with the budget parameter set at 1ms.

The external call to `Release_Sproadic` occurs at time 15. The client task is released and, as there is budget available, the release time of the task is noted (15) as is the current capacity of the budget (which is 3). The task executes immediately (i.e there are no higher priority tasks); it starts to use the budget. In this example, its CPU requirement is 4ms; it executes for 3ms and then, at time 18, the budget handler is triggered as the budget has been exhausted. In the handler, a timing event is constructed; its trigger time is 35 and its budget parameter is 3ms. The task is given a background priority. Our example has a lower priority task (above the background) that consumes CPU time, so the client task is not able to execute. The next event is the triggering of the first timing event at time 21. This adds 1ms to the budget and allows the task to continue to execute at its higher priority level. The time of release is noted (21) and the budget outstanding (1ms). The task finishes its invocation (and waits for the next one) with this 1ms capacity. Again, a timing event is created and triggered at time 41 (with parameter 1ms).

If the task while at the background priority was able to execute then this would not impact on the budget (which is zero). If it gets as far as completing its execution then when it calls `Wait_For_Next_Invocation`, a timing event with parameter 0ms will be constructed – this is inefficient, but a rare event and hence probably better to allow rather than test

for non-zero parameter.

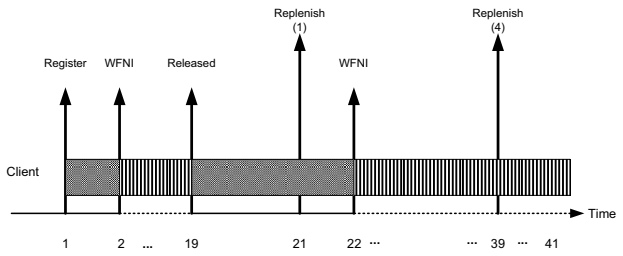

Figure 2. Sporadic Server Illustration 2

To illustrate just a further feature of the algorithm (see Figure 2), assume the call of `Release_Sproadic` occurred at time 19 (rather than 15). Now, the task will start executing at time 19 and run until it uses up the available budget at time 22. However, during that interval, the triggering of the timing event will occur at time 21. This will add 1ms to the budget but have no further effects as the budget is non-zero at that time. The task can now run to completion and, when it calls `Wait_For_Next_Invocation`, a timing event with a parameter of 4ms and a triggering time of 39 will be constructed. It may appear that this replenishment time is too soon (should it not be 40?) but the analysis of the Sporadic Server algorithm does allow this optimisation.

The above code has a single client and uses dynamically created timing objects. Adding more clients is relatively straightforward although it will require the use of execution time clocks to track budget usage of each client. The alternative to dynamically creating timing objects is to reuse a finite pool defined within the controller.

As CPU time monitoring (and hence budget monitoring) is not exact, the above algorithm is likely to suffer from drift – in the sense that the budget returned is unlikely to be exactly the equivalent of the amount used. To counter this, some form of re-asserting the maximum budget is required. For example the `Timer_Handler` routine could be of the form:

```
procedure Timer_Handler(E : in out Timing_Event) is
begin
  ...
  G_Budget.Add(Bud);
  if Budget_Remaining(G_Budget) > Params.Budget then
    Replenish(G_Budget, Params.Budget);
  end if;
  ...
end Timer_Handler;
```

This will prevent too much extra budget being created. To counter budget leakage, it would be necessary to identify when the task has not run for time `Period` and then make sure the budget was at it maximum level (using `Replenish`). This could be achieved with a further timing event that is set when the task is blocked on `Wait_For` and triggered at this time plus `Period` unless it is cancelled in `Release_Sporadic`.

4. Related Work

Work of real-time programming models for fixed priority systems can be divided into two categories. Those that use research-based languages (usually based on C – for example [15]) or real-time operating systems (such as the Shark kernel [12]) and those attempts to bring the result of research into international standards. Ada, of course, falls into the latter category. Of this class, the main technologies that can it can be compared to is the Real-Time Posix extensions [16] and the Real-Time Specification for Java [18].

Real-Time Posix, like Ada, attempts to provide low-level real-time mechanisms. It supports CPU-Time clocks that can be integrated into its other timing abstractions (e.g., timers that can take any clock type, and signals that can be generated when timers expire). However, it chooses to support a particular execution-time server: the sporadic server. There is no other notion of thread group budgets, and consequently constructing other servers is not possible.

The Real-Time Specification for Java provides higher level models than Ada or Posix. It directly supports periodic, aperiodic and sporadic programming abstractions. It also supports the notion of a processing group. Threads can be associated with a group and the group can be given a budget and a replenishment period. Moreover, unlike the sporadic and deferrable servers, the threads can have different parameters, and they are suspended when the budget is exhausted. By constraining the priorities, a deferrable server can be implemented but the more complicated sporadic sever is not possible.

5. Conclusion

Many of the new features of Ada 2005 take the form of events that are fired by the program's execution environment. Example events are:

- when a task's execution time reaches a defined value,
- when time reaches a defined value,
- when a group budget reaches zero,
- when an interrupt occurs (existing Ada 95 feature),
- when a task terminates.

All of these events trigger the execution of an application-defined handler programmed as a protected procedure with a ceiling priority that determines the priority at which the handler is executed. These provide the building blocks upon which it is possible to construct flexible real-time systems.

The focus of this paper has been on the construction of execution-time server abstractions using the facilities of Ada 2005. Simple servers such as the Deferrable Server are straightforward and need just a simple timing event and group budget. The Sporadic Server by contrast is quite complicated, and its implementation is a testament to the expressive power of the revised language. There are a number of other server algorithms for fixed priority scheduling and others, such as the bandwidth preserving server, that are defined to work with EDF[1, 11]. Space restrictions prevents any further examples in this paper, but they can be programmed following the general approach defined.

Our current work is focussing on extending the approach given in this paper to develop a range of real-time programming utilities that support a range of real-time task abstractions. The goal is to provide the Ada programmer with a high-level model on par with that provided by the RTSJ. However, the advantage of the Ada approach is that if the application requires a different abstraction then it can easily be provided from the low-level Ada mechanisms. In the RTSJ, the given abstractions are more difficult to tailor.

Ada has come along way since its initial version back in the early 1980s. That version was heavily criticised. Ada 95 responded to those criticisms and is an efficient language for high-reliable long-lived real-time applications. Yet arguably the damage to the language's reputation had already been done. Ada has for too long struggled to overcome those initial setbacks. Ada 2005 has continued the recovery, and the language now provides a comprehensive set of mechanisms that can deliver modern real-time scheduling theory to the systems engineer. It is time for those with prejudice against Ada to reconsider their position, and see Ada 2005 for what it really is – a flexible concurrent real-time object-oriented programming language.

6. Acknowledgements

The authors wish to thank member of ISO committee ARG and attendees of the IRTAW series for their input to the issues discussed in this paper. We also would like to thank AdaCore for allowing access to early releases of their Ada 2005 compiler, which was used to test the algorithms presented in this paper.

References

[1] L. Abeni, G. Lipari, and G. Buttazzo. Constant bandwidth vs proportional share resource allocation. In *Proceedings of the IEEE International Conference on Mutimedia Computing and Systems*, Florence, Italy, June 1999.

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

[3] G. Bernat, I. Broster, and A. Burns. Rewriting history to exploit gain time. In *Proceedings Real-time Systems Symposium*, pages 328–335, Lisbon, Portugal, 2004. Computer Society, IEEE.

[4] G. Bernat and A. Burns. New results on fixed priority aperiodic servers. In *Proceedings 20th IEEE Real-Time Systems Symposium*, pages 68–78, 1999.

[5] R. Brukardt(ed). Ada 2005 reference manual. Technical report, ISO, 2006.

[6] A. Burns, M. González Harbour, and A.J. Wellings. A round robin scheduling policy for Ada. In *Reliable Software Technologies, Proceedings of the Ada Europe Conference*, volume LNCS 2655, pages 334–343. Lecture Notes on Computer Science, Springer Verlag, 2003.

[7] A. Burns and A. J. Wellings. Accessing delay queues. In *Proceedings of IRTAW11, Ada Letters, Vol XX1I(4)*, pages 72–76, 2002.

[8] A. Burns and A.J. Wellings. Task attribute-based scheduling - extending Ada's support for scheduling. In T. Vardenega, editor, *Proceedings of the 12th International Real-Time Ada Workshop*, volume XXIII, pages 36–41. ACM Ada Letters, 2003.

[9] A. Burns, A.J. Wellings, and T.Taft. Supporting deadlines and EDF scheduling in Ada. In *Reliable Software Technologies, Proceedings of the Ada Europe Conference*, pages 156–165. Springer Verlag, LNCS 3063, 2004.

[10] A. Burns, A.J. Wellings, and T. Vardanega. Report of session: Flexible scheduling in Ada. In *Proceedings of IRTAW 12, Ada Letters, Vol XXIII(4)*, pages 32–25, 2003.

[11] M. Caccamo and L. Sha. Aperiodic servers with resource constraints. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 2001.

[12] P. Gai, L. Abeni, M. Giorgi, and G. Buttazzo. A new kernel approach for modular real-time systems development. In *Proceedings of the 13th IEEE Euromicro Conference on Real-Time Systems*, pages 199–207, 2001.

[13] J.P. Lehoczky and S. Ramos-Thuel. An optimal algorithm for scheduling soft-aperiodic tasks fixed-priority preemptive systems. In *Proceedings 13th IEEE Real-Time Systems Symposium*, pages 110–123, 1992.

[14] J.P. Lehoczky, L. Sha, and J.K. Strosnider. Enhanced aperiodic responsiveness in a hard real-time environment. In *Proceedings 8th IEEE Real-Time Systems Symposium*, pages 261–270, 1987.

[15] L. Palopoli, G. Buttazzo, and P. Ancilotti. A C language extension for programming real-time applications. In *Proceedings of the 6th IEEE International Conference on Real-Time Computing Systems and Applications*, pages 102–110, 1999.

[16] M. Aldea Rivas and M. Gonzalez Harbour. Evaluation of new POSIX real-time operating systems services for small embedded platforms. In *Proceedings of the 15th IEEE Euromicro Conference on Real-Time Systems*, pages 161–168, 2003.

[17] B. Sprunt, L. Sha, and J. P. Lehoczky. Aperiodic task scheduling for hard real-time systems. *Real-Time Systems*, 1:27–69, 1989.

[18] A.J. Wellings. Is Java augmented with the RTSJ a better real-time systems implementation technology than Ada 95? In *Proceedings of IRTAW 12, Ada Letters, Vol XXIII(4)*, pages 16–21, 2003.