Implementing a Flexible Scheduler in Ada

Guillem Bernat and Alan Burns
Real-Time Systems Research Group
Department of Computer Science
University of York

Abstract

Much of the research on flexible scheduling schemes is prevented from being used in practice by the lack of implementations that provide the necessary abstractions. In this paper we show how Ada's tasking facilities do enable such schedulers to be constructed. A case example is given that shows that the combination of existing language features is sufficient to program the required functionality. Only the lack of budget time management causes difficulty.

1 Introduction

Over the last decade a considerable volume of literature has been produced which addresses the issues and challenges of flexible scheduling. The focus of much of this research has been on how to effectively support a mixture of periodic and non-periodic, hard and non-hard 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 applications 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 technique 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 scheduling models - too many models have unrealistic assumptions (such as ignoring run-time overheads).
- Difficulty of implementation - often the scheduling scheme will require operating 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 computational algorithms - the seminal work on imprecise algorithms[15, 10, 12] (and anytime algorithms[7, 19, 13, 20]) has not lead to the development of application-specific instantiations of these generic schemes.

The focus of this paper is on the third of these problems. We will show how the facilities available in the Ada tasking model do allow implementations to be programmed. The aim of this, and related work, is to define a framework that will have wide applicability and utility. The framework is given the name *Jorvik*. Of course we would not argue that the framework will be effective in all situations. This is not the aim. Rather, we contend that it is possible to deliver, with today's technology, a general purpose platform that will enable flexible scheduling schemes to be exploited in a range of application domains.

The challenges of flexible scheduling are easy to (informally) define: hard deadlines must always be met, and any spare capacity (typically CPU resource) must be used to maximise the utility of the application. 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 utilisation of the hard task will often be less than 50%. There is continuing pressure on applications to minimise their use of hard tasks and to exploit flexible scheduling to deliver dynamic behaviour. The framework presented in this paper will facilitate these application requirements.

In this paper we assume that all non-hard jobs are *firm*. That is, they need not complete by their deadline, but there is no value in executing them beyond their deadline – no extra value will accrue. Firm jobs are assigned (statically) a 'value', V [11, 3, 1, 5, 8]. This is the utility that will be generated by the job if it completes by its deadline. Another important attribute of all jobs is their execution time, C. Minimum, average and worst-case measures may all be available.

2 Scheduling Firm Tasks

Given some mechanism for managing spare processing capacity, the resulting spare resource time must be assigned efficiently to the incoming firm tasks. A number of results in the literature form a boundary to this problem:

- if there is no overload then EDF is an optimal scheme, but if there is an overload EDF performs poorly due to the known domino effect [4],
- with a pure value based scheme, scheduling according to V/C is the most effective [11].

The entity V/C, which we denote as W, gives the value that will accrue per unit of execution. It is optimal if a system is heavily loaded. These two observations have lead to a general scheduling structure which consists of two mechanisms. An *admissions policy* (that keeps the load from exceeding the capacity of the resource) and a *scheduling policy* (for the jobs that are admitted). If the admissions policy is effective then overloads are prevented and hence EDF is an effective scheduling policy. The difficulty with the admissions policy is that any near optimal scheme has very high overheads [16, 14, 9].

The scheme that Jorvik adopts is based on the following observations:

- it is wasteful to start a task that has little chance of completing and that would therefore be aborted (as zero value is obtained after the deadline). However, some level of aborts can be tolerated if it leads to better performance overall,
- in a busy system the state of the available resources as predicted when a task arrives is a poor indicator of the actual state when the task first gets to run,

- the ‘optimal’ best effort scheme [11] has extremely high overheads [17, 18] and is only effective if exact execution times are known and delivered at run-time.

The algorithmic behaviour of Jorvik is as follows.

1. Each application task consists of a *mandatory* (hard) job and an *optional* (firm) job. Either can be omitted. The mandatory job always executes first. Both jobs **should** finish by the deadline - the hard one **must**.

2. When a firm job arrives it has known value V and minimum and required computation time.

3. Upon arrival all jobs are put into an EDF queue.

4. When a job gets to the front of the queue for the first time, a decision is made to either accept it (execute it) or revoke it.

5. If the job is revoked it is removed from the queue and never executes (for this invocation).

6. If the job is accepted it executes; a budget may be assigned and the job may be aborted at its deadline.

By postponing the decision to execute until this late stage, the data of relevance is as timely as possible; although it is still based on estimates of the resources needed by the jobs in the EDF queue.

As all hard and firm jobs are placed in the EDF queue it will become overloaded. Jorvik implements what might be called a *lazy* admissions policy; jobs are only revoked immediately before they would otherwise start executing, rather than when they arrive.

The decision embodied in stage 4 has two components. One concerns the jobs itself, the other the load in the EDF queue. In the following description non-preemptive execution for the EDF jobs is assumed.

To ensure that hard jobs always meet their deadlines, but that the maximum spare capacity is made available for the firm jobs, the dual priority scheme is employed[6, 12, 2]. With this scheme a hard job (task) starts its execution at a lower priority but has its priority raised ‘just in time’ so that it meets its deadline (by preempting the current EDF job). While at the lower priority firm tasks can execute in preference. With the algorithm used in this paper the hard jobs are initially placed in the EDF queue. They will either execute as they come to the front of this queue, or they will have their priorities raised so that they execute in preference to the current EDF job.

3 Programming the Scheduler in Ada

Ada’s task facilities provide an expressive set of facilities from which to construct a flexible scheduler. In this section these features are illustrated via the construction of a value-based scheduler for Jorvik.

Figure 1 gives an overview of the proposed scheme. In the figure a single client (there would of course be more in a complete system) calls the `Sch` protected object to either lodge a mandatory or optional execution request. All management of the dispatching process is undertaken by manipulating priorities.

- Clients call in at their normal priority but then have their priorities dropped.

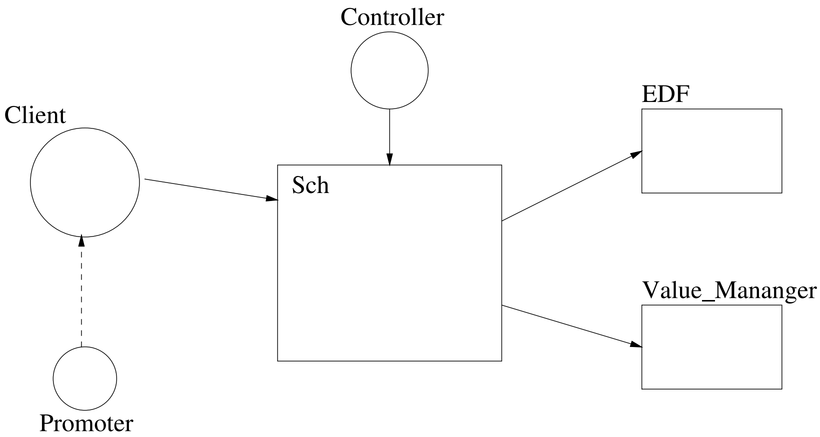

Figure 1: Overview of Proposed Scheme

- The promoter task raises the priority of the client to ensure the mandatory part is completed.
- When an execution request gets to the head of the EDF queue (and is accepted for execution) its priority is raised.
- The priority of ‘controller’ is chosen so that it only executes if the current ‘EDF’ job has completed.

These relationships are embedded in the following package:

```
package Program_Priorities is
  Hold : constant System.Priority := System.Priority'First + 1;
    -- priority of all task with jobs on EDF queue
  Controller_Pri : constant System.Priority := Hold + 1;
  Running : constant System.Priority := Hold + 2;
    -- priority of running task, i.e. job just removed from EDF queue
  Pro_Pri : constant System.Priority := System.Priority'Last;
    -- promoter tasks' priority
  subtype Promoted_Priorities is System.Priority range
     Hold + 3 .. System.Priority'Last - 1;
    -- priority of tasks executing hard jobs (after promotion)
end Program_Priorities;
```

The `Sch` protected object makes use of the two ‘queues’: one holds all jobs in EDF order, the other manages the optimal jobs using V and C.

First some general type definition are needed:

```
package Specific_Parameters is
  type Value is range 0 .. 10; -- say
  type Load is new Float range 0.0 .. 10.0; -- say
    -- load is a measure of current EDF work load
  Threshold : Load := 0.8; -- say
end Specific_Parameters;
```

The EDF package has a procedure for removing a mandatory job that has executed anyway (if it is found a Boolean flag is set). The `Controller` will call `Extract` to get the next job (i.e. job with shortest deadline). This routine also returns the current load on the EDF queue. If this load factor is greater than a threshold then an overload is assumed.

```
package Edf is
  procedure Add(Id : Task_Id);
  procedure Remove(Id : Task_Id; Found : out Boolean);
  procedure Extract(Id : out Task_Id; L : out Load);
end Edf;
```

The `Value_Manager` allows jobs to be added and removed. If there is an overload the routine will undertake a 'cut' on the outstanding jobs using the load factor. The 'current' job is this either admitted or not. Either way it is removed.

```
package Value_Manager is
  procedure Add(Id : Task_Id);
  procedure Remove(Id : Task_Id);
  procedure Cut(Id : Task_Id; Ld : Load; Admit : out Boolean);
end Value_Manager;
```

To pass scheduling parameters between the client task, its promoter and the scheduler, task-attributes are used:

```
package Task_Info is
  type Task_Information is
  record
    Deadline : Time := Time_Last;
    Val : Value := Value'First;
    Comp_Min, Comp_Req : Time_Span := Time_Span_Zero;
    Man_Fin : Boolean := False;
  end record;

  Default : Task_Information;
  -- default object is needed in definition of task attributed

  package Scheduling_Parameter is new
    Ada.Task_Attributes(Task_Information, Default);
end Task_Info;
```

The promoter task is of a general structure for all clients. A rendezvous is used to pass the static timing parameters to the promoter

```
task type Promoter is
  entry Set_Times(Start_T : Time; Hp : Priority;
                  Period,R_Deadline,R_Pro : Time_Span);
  pragma Priority(Pro_Pri);
end Promoter;

type Pro is access Promoter;

task body Promoter is
  My_Period : Time_Span;
  Pro_Time : Time_Span;
  Rel_Deadline : Time_Span;
  Tp : Task_Information;
  High : Priority;
  Client : Task_Id;
  Epoch : Time;
begin
  accept Set_Times(Start_T : Time; Hp : Priority;
```

```
                  Period,R_Deadline,R_Pro : Time_Span) do
    My_Period := Period;
    Epoch := Start_T;
    High := Hp;
    Pro_Time := R_Pro;
    Rel_Deadline := R_Deadline;
    Client := Set_Times'Caller;
  end Set_Times;
  loop
    delay until Epoch + Pro_Time;
    Tp := Scheduling_Parameter.Value(Client);
    if not Tp.Man_Fin then
      Set_Priority(High,Client);
    end if;
    delay until Epoch + Rel_Deadline;
    Set_Priority(High,Client);
    Epoch := Epoch + My_Period;
  end loop;
end Promoter;
```

The client task necessarily has a number of internal data definitions:

```
task type Example_Client(My_Priority : Promoted_Priorities) is
  pragma Priority(My_Priority);
end Example_Client;

task body Example_Client is
  Start : Time;
  Period : Time_Span := ...
  Rel_Deadline : Time_Span := ...
  Abs_Deadline : Time;
  Rel_Promotion_Time : Time_Span := ...
  Mandatory_Finished : Boolean;
  Self : Task_Id;
  Quality : Value;
  Timing_Error : exception;
  Minimum_Comp_Man : Time_Span := ...
  Required_Comp_Man : Time_Span := ...
  Minimum_Comp_Opp : Time_Span := ...
  Required_Comp_Opp : Time_Span := ...
  P : Pro := new Promoter;
begin
  ...
end Example_Client;
```

The body of the task uses a select-then-about construct to ensure that the deadline is not overrun. If it was, and the mandatory part had not completed, then this would be an error.

```
begin
  Start := Clock;
  Self := Current_Task;
  P.Set_Times(Start, My_Priority, Period, Rel_Deadline,
              Rel_Promotion_Time);
  loop
    Mandatory_Finished := False;
    Abs_Deadline := Start + Rel_Deadline;
    select
      delay until Abs_Deadline;
      if not Mandatory_Finished then
        raise Timing_Error;
      end if;
```

```
    then abort
      Quality := Value'Last;
      Scheduling_Parameter.Set_Value((Abs_Deadline, Quality,
        Minimum_Comp_Man, Required_Comp_Man, Mandatory_Finished));
      Sch.Register_Mandatory(Self);
      -- Mandatory Job
      Mandatory_Finished := True;
      Sch.Remove_Mandatory(Self);
      Quality := ... -- not Value'Last
      Scheduling_Parameter.Set_Value((Abs_Deadline, Quality,
        Minimum_Comp_Opp, Required_Comp_Opp, Mandatory_Finished));
      Sch.Register_Optional(Self);
      -- Optional Job
    end select;
    Start := Start + Period;
    delay until Start;
  end loop;
exception
  when ...
end Example_Client;
```

The scheduler protected object (`Sch`) is the more complex entity:

```
protected Sch is
  entry Release_New;
  procedure Register_Mandatory(Cl : Task_Id);
  procedure Remove_Mandatory(Cl : Task_Id);
  procedure Register_Optional(Cl : Task_Id);
  pragma Priority(Pro_Pri);
private
  Num_On_Edf : Natural := 0;
end Sch;
```

The entry is called by the controller task, which only executes when a job has finished (via the judicious setting of the priority levels):

```
task Controller is
  pragma Priority(Controller_Pri);
end Controller;

task body Controller is
begin
  loop
    Sch.Release_New;
  end loop;
end Controller;
```

If there are no current jobs then the controller will block on the entry.

For clarity of presentation two different register procedures are given (although it would be easy to roll them into one). The `Remove_Mandatory` routine is needed to cover for a job that has executed anyway (due to the dual priority being invoked). The protected procedures are straightforward.

```
protected body Sch is
  entry Release_New when ...
  end Release_New;

  procedure Register_Mandatory(Cl : Task_Id) is
  begin
    Edf.Add(Cl);
    Num_On_Edf := Num_On_Edf + 1;
```

```
    Set_Priority(Hold,Cl);
  end;

  procedure Remove_Mandatory(Cl : Task_Id) is
    Found : Boolean;
  begin
    Edf.Remove(Cl,Found);
    if Found then
      Num_On_Edf := Num_On_Edf - 1;
    end if;
  end;

  procedure Register_Optional(Cl : Task_Id) is
  begin
    Edf.Add(Cl);
    Num_On_Edf := Num_On_Edf + 1;
    Value_Manager.Add(Cl);
    Set_Priority(Hold,Cl);
  end;
end Sch;
```

The call of the entry is what causes a new EDF job to be selected:

```
entry Release_New when Num_On_Edf > 0 is
  Id : Task_Id;
  Ld : Load;
  Tp : Task_Information;
  Run_It : Boolean;
begin
  Edf.Extract(Id,Ld);
  Num_On_Edf := Num_On_Edf - 1;
  Tp := Scheduling_Parameter.Value(Id);
  if Tp.Val = Value'Last then -- mandatory part
    Set_Priority(Running,Id);
    return;
  end if;
  if Clock + Tp.Comp_Min > Tp.Deadline then
    Value_Manager.Remove(Id); -- not worth starting
    requeue Release_New;
  end if;
  if Ld <= Threshold then  -- no overload
    Value_Manager.Remove(Id);
    Set_Priority(Running,Id);
    return;
  end if;
  Value_Manager.Cut(Id,Ld,Run_It); -- overload
  if Run_It then
    Set_Priority(Running,Id);
  else
    requeue Release_New; -- find another task
  end if;
end Release_New;
```

Note the use of requeue to find another job (if the current one with the shortest deadline is not acceptable).

4 Conclusion

The focus of this paper has been on the construction of a flexible scheduler using the facilities of Ada. In doing this, the expressive power of the language has been demonstrated. As well as tasks and protected objects themselves, use is made of dynamic priorities, task attributes, select-then-about and requeue.

The one area in which Ada is deficient is in its control of execution time. It is not possible to set a computation time budget and then stop a computation when a budget is exhausted. It is to be hoped that a future version of the Real-Time Annex will rectify this shortcoming.

References

[1] S. Baruah, G. Koren, D. Mao, B. Mishra, A. Raghunathan, L. Rosier, D. Shasha, and F. Wang. On the competitiveness of online real-time task scheduling. *Real-Time Systems*, 4(2):124–144, 1992.

[2] G. Bernat and A. Burns. Combining (n m)-hard deadlines with dual priority scheduling. In *Proceedings of 18th IEEE Real-Time systems symposium. San Francisco, CA*, December 1997.

[3] S. Biyabani, J.A. Stankovic, and K. Ramamritham. The integration of criticalness and deadlines in scheduling real-time tasks. In *Proceedings of the 9th IEEE Real-Time Systems Symposium*, pages 152–169, 1988.

[4] G. Buttazzo, M. Spuri, and F. Sensini. Value vs. deadline scheduling in overload conditions. In *Proceedings of the IEEE Real-Time Systems Symposium*, 1995.

[5] G. Buttazzo and J. Stankovic. RED : A robust earliest deadline scheduling algorithm. In *Proceedings of the 3rd International Workshop on Responsive Computer Systems, Austin*, 1993.

[6] R.I. Davis and A. J. Wellings. Dual priority scheduling. In *Proceedings Real-Time Systems Symposium*, pages 100–109, 1995.

[7] T.L. Dean and M. Boddy. An analysis of time-dependent planning. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 49–54, 1988.

[8] E.D. Jensen, C.D. Locke, and H. Tokuda. A time driven scheduling model for real-time operating systems. In *Proceedings IEEE Real-Time Sytems Symposium*, pages 112–122, 1985.

[9] Y.S. Kim. An optimal scheduling algorithm for preemptive real-time tasks. *Information Processing Letters*, 50(1):43–48, 1994.

[10] J.W.S. Liu, K.J. Lin, W.K. Shih, A.C.S. Yu, J.Y. Chung, and W. Zhao. Algorithms for scheduling imprecise computations. *IEEE Computer*, pages 58–68, 1991.

[11] C.D. Locke. Best-effort decision making for real-time scheduling. CMU-CS-86-134 (PhD Thesis), Computer Science Department, CMU, 1986.

[12] R. Oliveira and J. Fraga. Scheduling imprecise computation tasks with intra-task/ inter-task dependence. In *Proceedings of the 21st IFAC/IFIP Workshop on Real-Time Programming, WRTP'96*, pages 51–56, 1996.

[13] S.J. Russell and S. Zilberstein. Composinng real-time systems. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, pages 212–217, 1991.

[14] K. Schwan and H. Zhou. Dynamic scheduling of hard real-time tasks and real-time threads. *IEEE Transactions on Software Engineering*, 18(8):736–748, 1992.

[15] W. K. Shih, J. W. S. Liu, and J. Y. Chung. Algorithms for scheduling imprecise computations with timing constraints. In *Proc. IEEE Real-Time Systems Symposium*, 1989.

[16] M. Silly, H. Chetto, and N. Elyounsi. An optimal algorithm for guaranteeing sporadic tasks in hard real-time systems. In *Proceedings 2nd IEEE Symposium on Parallel and Distributed Systems*, pages 578–585, 1990.

[17] H. Tokuda, J.W. Wendorf, and H.Y. Wang. Implementation of a time-driven scheduler for real-time operating systems. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 271–280, 1987.

[18] J.W. Wendorf. Implementation and evaluation of a time-driven scheduling processor. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 172–180, 1988.

[19] S. Zilberstein. Resource-bounded sensing and planning in autonomous systems. *Autonomous Robots*, 3:31–48, 1996.

[20] S. Zilberstein. Using anytime algorithms in intelligent systems. *AI Magazine*, 17(3):73–83, 1996.