Technical Report

Implementing Multicore Real-Time Scheduling Algorithms Based on Task Splitting Using Ada 2012

Björn Andersson and Luís Miguel Pinho

HURRAY-TR-100301

Version: 1.0

Date: 10-03-2010

Implementing Multicore Real-Time Scheduling Algorithms Based on Task Splitting Using Ada 2012

Björn Andersson and Luís Miguel Pinho

IPP-HURRAY!

Polytechnic Institute of Porto (ISEP-IPP)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8340509

E-mail: bandersson@dei.isep.ipp.pt; lmp@isep.ipp.pt

http://www.hurray.isep.ipp.pt

Abstract

Multiprocessors, particularly in the form of multicores, are becoming standard building blocks for executing reliable software. But their use for applications with hard real-time requirements is non-trivial. Well-known real-time scheduling algorithms in the uniprocessor context (Rate-Monotonic [1] or Earliest-Deadline-First [1]) do not perform well on multiprocessors. For this reason the scientific community in the area of real-time systems has produced new algorithms specifically for multiprocessors. In the meanwhile, a proposal [2] exists for extending the Ada language with new basic constructs which can be used for implementing new algorithms for real-time scheduling; the family of task splitting algorithms is one of them which was emphasized in the proposal [2]. Consequently, assessing whether existing task splitting multiprocessor scheduling algorithms can be implemented with these constructs is paramount. In this paper we present a list of state-of-art task-splitting multiprocessor scheduling algorithms and, for each of them, we present detailed Ada code that uses the new constructs.

Implementing Multicore Real-Time Scheduling Algorithms Based on Task Splitting Using Ada 2012

Björn Andersson[1] and Luís Miguel Pinho[1],

[1] CISTER-ISEP Research Centre,
Polytechnic Institute of Porto, Portugal
bandersson@dei.isep.ipp.pt, lmp@isep.ipp.pt

Abstract. Multiprocessors, particularly in the form of multicores, are becoming standard building blocks for executing reliable software. But their use for applications with hard real-time requirements is non-trivial. Well-known real-time scheduling algorithms in the uniprocessor context (Rate-Monotonic [1] or Earliest-Deadline-First [1]) do not perform well on multiprocessors. For this reason the scientific community in the area of real-time systems has produced new algorithms specifically for multiprocessors. In the meanwhile, a proposal [2] exists for extending the Ada language with new basic constructs which can be used for implementing new algorithms for real-time scheduling; the family of task splitting algorithms is one of them which was emphasized in the proposal [2]. Consequently, assessing whether existing task splitting multiprocessor scheduling algorithms can be implemented with these constructs is paramount. In this paper we present a list of state-of-art task-splitting multiprocessor scheduling algorithms and, for each of them, we present detailed Ada code that uses the new constructs.

Keywords: Ada, multiprocessors, multicores, real-time scheduling.

1 Introduction

Despite multiprocessors, in the form of multicores, becoming the norm in current computer systems, their use for applications with real-time requirements is non-trivial. The reason is that although a comprehensive toolbox of scheduling theories is available for a computer with a single processor, such a well-established comprehensive toolbox is currently not available for multicores.

One of the emerging and most interesting classes of multiprocessor scheduling algorithms today is called task-splitting scheduling algorithms [3-9]. With such an algorithm, most tasks are assigned to just one processor, whilst a few tasks are assigned to two or more processors and may migrate in a controlled manner (the migration may be performed in the middle of the execution of a job) so that at every instant, such a task never executes on two or more processors simultaneously. This class is appealing because the algorithms in this class (i) cause few (and provably small number of) preemptions and (ii) can be proven to be able to schedule task sets to meet deadlines even at high processor utilizations.

The Ada community has shown an increasing interest in real-time scheduling on multicores [2,10,13] and a proposal exists [2] for extending the language for real-time scheduling on multicores. Our initial opinion was that the proposed extension seemed useful for implementing task splitting, but it is important to fully evaluate its appropriateness considering the task splitting scheduling algorithms that have already been published.

Therefore, in this paper, we present Ada code for implementing a subset of the current task splitting scheduling algorithm. From the extensive set of previously published algorithms, we have selected the ones [4,8,9] that perform best (in terms of being able to schedule tasks at high utilization and generating few preemptions), and that allow showing how different types of approaches can be programmed in Ada. We would like to note, nevertheless, that the algorithm in [3] could be also used, but it may require very small execution segments at highly precisely specified time intervals, something which is difficult to achieve in practice.

We find that the new proposal [2] is sufficient for implementing those task splitting algorithms [4,8,9] that we believe are useful to designers. Attaining efficient implementations of them may require a new timing construct however.

The remainder of this paper is organized as follows. Section 2 presents the system model and gives an overview of the algorithms. Section 3 presents the recently proposed language extension. Sections 4 to 6 present Ada programs for the dispatchers of the task splitting algorithms. Section 7 provides conclusions.

2 System model and algorithm overview

We consider the problem of scheduling a set of tasks $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$ on m processors. A task τ_i is characterized by T_i, D_i, and C_i with the interpretation that the task τ_i releases a (potentially infinite) sequence of jobs such that (i) the time between two consecutive jobs of the same task is at least T_i and (ii) each job must complete C_i units of execution within at most D_i time units from the release of the job. We assume that a job cannot execute on two or more processor simultaneously. We also assume that a processor cannot execute two or more jobs at the same instant. We assume that a job needs no resource (such as shared data structures) other than a processor for execution.

We distinguish between three types of task sets:

- In an implicit-deadline task set, each task τ_i has $D_i = T_i$;
- In a constrained-deadline task set, each task τ_i has $D_i \leq T_i$;
- In an arbitrary-deadline task set, each task τ_i is not constrained by the above ($D_i = T_i$ or $D_i \leq T_i$).

In this paper, we focus on algorithms for constrained-deadline task sets. In order to understand task splitting algorithms, let us consider the following example.

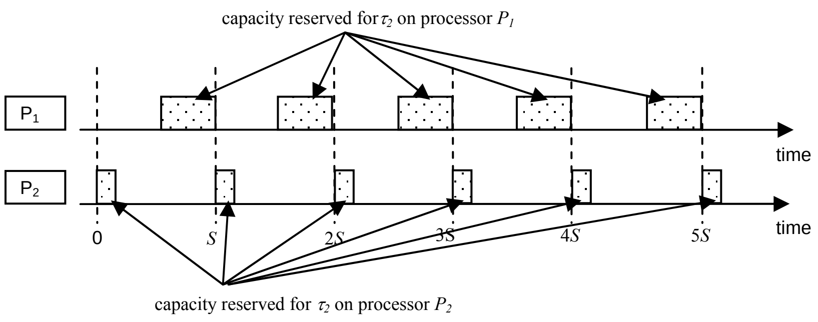
capacity reserved for τ_2 on processor P_1
P_1
time
P_2
0
S
$2S$
$3S$
$4S$
$5S$
time
capacity reserved for τ_2 on processor P_2

Figure 1. Slot-based split-task dispatching: How to perform run-time dispatching of a task that is assigned to two processors. A white rectangle with black dots indicates capacity reserved for task τ_2.

Example 1. Consider three tasks to be scheduled on two processors. Each task τ_i has T_i=D_i=1 and C_i=0.51. We can assign task τ_1 to processor 1 and task τ_3 to processor 2 and then let task τ_2 be assigned to both processors 1 and 2; we say that τ_2 is a *split task*. This splitting should be done in a controlled manner; for example do the splitting of τ_2 so that τ_2 requires 0.379 units of execution on processor 1 and 0.131 units of execution on processor 2. Since τ_2 is assigned to two processors, it is important that dispatchers on each processor ensure that τ_2 never executes on two or more processors simultaneously. □

In task splitting algorithms, there are three approaches for ensuring that a split task does not execute on two or more processors simultaneously:

- Slot-based split-task dispatching;
- Job-based split-task dispatching;
- Suspension-based split-task dispatching.

Slot-based split-task dispatching is used in [4, 5, 7]. Figure 1 shows the idea. Time is organized into timeslots of equal size and these timeslots are synchronized across all processors. The time interval of a timeslot is partitioned into three sub-time-intervals, one in the beginning of the timeslot, one in the middle of the timeslot and one in the end of the timeslot. A split task is assigned to the beginning sub-time-interval of one processor and the end sub-time-interval of another processor; these time intervals must be dimensioned so that the task meets its deadline and so that there is no overlap in time between the subintervals.

Job-based split-task dispatching is used in [8, 9]. Figure 2 shows the idea. There are no timeslots. Instead, when a job is released, a certain condition is setup specifying when the job should migrate to another processor. This condition can be that a certain amount of time has elapsed since the release of the job (used in [9]) or that a certain amount of execution has been performed by the job (used in [8]).

Suspension-based split-task dispatching is similar to job-based split-task dispatching but the default case is that all pieces of a job are ready all the time on all

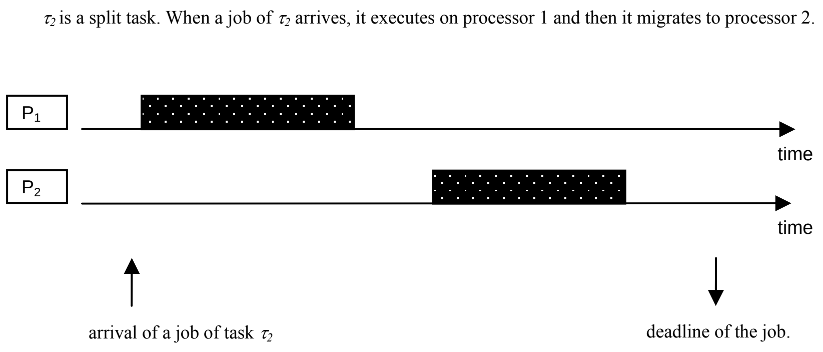
τ_2 is a split task. When a job of τ_2 arrives, it executes on processor 1 and then it migrates to processor 2.
P_1
time
P_2
time
arrival of a job of task τ_2
deadline of the job.

Figure 2. Job-based split-task dispatching: How to perform run-time dispatching of a task that is assigned to two processors. A dark rectangle with white dots indicates execution of the job of task τ_2.

processors to which the split task is assigned. But when the job executes on one processor, it suspends the job on the other processors.

Slot-based split-task dispatching and job-based split task dispatching are areas of active research in the real-time systems research community. The slot-based split-task dispatching offers higher utilization bounds whereas the job-based split task dispatching offers fewer preemptions.

Suspension-based split-task dispatching is not possible in the proposed Ada model for multiprocessors, since there is a single ready queue within the same allocation domain. It is also currently not explored in the real-time systems research community. The authors believe this is because suspension-based split-task dispatching provides utilization bounds and preemption bounds similar to the job-based split-task dispatching but with the suspension-based split-task dispatching there is the drawback that it can happen that an event (say a release of a job) on processor 1 causes a context switch on processor 2 which in turns causes a context switch on processor 3 and so on.

Hence, we will only discuss (i) slot-based split-task dispatching and (ii) job-based split-task dispatching, because we believe they are most relevant for software developers.

3 Language extensions

Burns and Wellings have proposed in [2, 10] language extensions for real-time scheduling on multicores, which after discussion in the International Real-Time Ada Workshop [13] have been proposed for the upcoming Ada revision [14]. This section presents the proposed extension, but limited to what is necessary for implementing the task splitting algorithms (more details on this proposal can be found in [16]).

The proposed extension defines appropriate packages for handling the set of CPUs available to the program, and the creation of non-overlapping dispatching domains:

```
package System.MultiProcessors is
   type CPU_Range is range 0..<implementation-defined>;
   function Number_Of_CPUs return CPU_Range;
end System.MultiProcessors;

package System.MultiProcessors.CPU_Sets is
   type CPU_Set is private;
   Default_CPU_Set : constant CPU_Set;
   procedure All_Set( Set: in out CPU_Set);
end System.MultiProcessors.CPU_Sets;

package Ada.Dispatching is
   type Dispatching_Domain_Policy is private;
   -- other declared types and subprograms not shown here
end Ada.Dispatching;

package Ada.Dispatching.Domains is
   type Dispatching_Domain is  private;
   System_Dispatching_Domain: Dispatching_Domain;

   -- other declared subprograms not shown here

   procedure Set_CPU(P : in CPU_Range;
                     T : in Task_Id := Current_Task);

   procedure Delay_Until_And_Set_CPU(
                     Delay_Until_Time : in Ada.Real_Time.Time;
                     P : in CPU_Range);
end Ada.Dispatching_Domains;
```

Procedure `Set_CPU` is fundamental for task splitting as it allows to dynamically change the allocation of tasks to specific CPUs.

Although not used in this paper, the capabilities for supporting more than one dispatching domain are very interesting for other approaches. For example, it is also important for some partitioned cluster approaches (such as in [7]) since it allows detecting incorrect assignment of tasks to processors. Also, it is useful for improving the performance of algorithms that do not use task splitting. For example, global scheduling with EDF suffers from poor ability to meet deadlines for certain task sets but this effect can be mitigated by subdividing processors into disjoint dispatching domains and applying global scheduling with EDF on each dispatching domain (such an approach is sometimes called *clustered-global EDF* [12]).

4 Slot-based split tasks dispatching

The algorithm described in this section is the one in [4], and it is shown in Figure 1. The algorithm is intended for implicit-deadline sporadic tasks. The left column of page 4 in [4] gives a good description of the dispatching algorithm. In this section, we reformulate it with the proposed Ada extensions.

To illustrate task splitting, we will consider a task set $\tau=\{\tau_1, \tau_2, \tau_3\}$ to be scheduled on two processors. The tasks are characterized as T_1=100 ms, T_2=200 ms, T_3=400 ms, D_1=*100* ms, D_2=*200* ms, D_3=400 ms and C_1=*51* ms, C_2=*102* ms, C_3=204 ms.

Recall that the algorithm depends on a timeslot; the size of the timeslot is $TMIN/\delta$, where $TMIN$ is the minimum of T_i of the task set and the parameters δ can be chosen by the user. We choose δ=4 and apply it to the example above and this gives us that the timeslot has a duration of 25 ms. Also, the algorithm depends on a parameter *SEP* which specifies how much we can fill-up a processor when we (i) assign tasks to processors and (ii) split tasks. Using Equation 27 in [4] tells us for δ=4 that *SEP*=0.889.

The task assignment/splitting algorithm in [4] gives us the following (for δ=4, *SEP*=0.889). Task τ_1 is assigned to processor 1; task τ_3 is assigned to processor 2; task τ_2 is assigned to both processor 1 and to processor 2 and the splitting of this task is specified by two variables, *hi_split* and *lo_split*, with values *hi_split*[1]=0.379 and *lo_split*[2]=0.131. Intuitively, this means that 37.9% of the processing capacity of processor 1 will be used for task τ_2 and analogously 13.1% of the processing capacity of processor 2 will be used for task τ_2. Together these figures (37.9% and 13.1%) give us the utilization of the task τ_2 (51%).

Recall that the timeslot duration is 25 ms. Due the unknown phasing of the task arrival related to the slot start, the algorithm specifies that a reserve on processor 1 for task τ_2 should occupy a fraction *hi_split*[1]+2α of the duration of the timeslot and a reserve on processor 2 for task τ_2 should occupy a fraction *lo_split*[2]+2α of the duration of the timeslot. (The value of α is computed based on δ; see Equation 9 in [4]; in this example α becomes 0.02786.) Therefore, the duration of the reserve for processor 1 becomes 4.668 ms and for processor 2 becomes 10.868 ms.

The code is as follows:

```
pragma Priority_Specific_Dispatching (EDF_Across_Priorities, 1, 10) ;
pragma Priority_Specific_Dispatching (FIFO_Within_Priorities, 11, 12);

with Ada.Real_Time.Timing_Events; use Ada.Real_Time.Timing_Events;
with Ada.Task_Identification; use Ada.Task_Identification;
with Ada.Dispatching.Domains; use Ada.Dispatching.Domains;
with System.Multiprocessors; use System.Multiprocessors;
with Ada.Real_Time; use Ada.Real_Time;
with Ada.Dispatching.EDF; use Ada.Dispatching.EDF;
with Ada.Asynchronous_Task_Control; use Ada.Asynchronous_Task_Control;

Period_Task_1             : constant Time_Span:=Milliseconds( 100);
Min_Inter_Arrival_Task_2 : constant Time_Span:=Milliseconds( 200);
Period_Task_3             : constant Time_Span:=Milliseconds( 400);

Deadline_Task_1           : constant Time_Span:=Milliseconds( 100);
Deadline_Task_2           : constant Time_Span:=Milliseconds( 200);
Deadline_Task_3           : constant Time_Span:=Milliseconds( 400);

Execution_Time_Task_1     : constant Time_Span:=Milliseconds(  51);
Execution_Time_Task_2     : constant Time_Span:=Milliseconds( 102);
Execution_Time_Task_3     : constant Time_Span:=Milliseconds( 204);

TMIN : constant Time_Span := Milliseconds( 100);
Time_Slot_Delta : constant integer := 4;
Time_Slot_Length : constant Time_Span := TMIN / Time_Slot_Delta;
Alpha : constant float := 0.02786; -- this is computed based on
                                   -- Time_Slot_Delta
```

```
CPU_1 : constant CPU_Range := 0;
CPU_2 : constant CPU_Range := 1;
Reserve_Phase_1_Task_2 : constant Time_Span:= Microseconds( 4668);
Reserve_Phase_2_Task_2 : constant Time_Span:= Microseconds(10868);

Start_Time : Time := Clock;

type Current_Phase is (Not_Released, Phase_1, Suspended, Phase_2);


protected type Sporadic_Switcher is
   pragma Priority(12);
   procedure Register(ID : Task_ID; Phase_1_CPU, Phase_2_CPU: CPU_Range;
                      Phase_1_Reserve, Phase_2_Reserve : Time_Span);
   procedure Handler(TM :in out Timing_Event);
   procedure Release_Task;
   procedure Finished;
   entry Wait;
private
   Released: Boolean := False;
   Switch_Timer: Timing_Event;

   Client_ID : Task_ID;
   Client_Phase_1_CPU, Client_Phase_2_CPU   : CPU_Range;
   Client_Phase_1_Reserve, Client_Phase_2_Reserve : Time_Span;
   Client_Current_Phase : Current_Phase;

   End_of_Phase_1, Start_of_Phase_2, End_of_Slot: Time;
end Sporadic_Switcher;

task Task_1 is
   pragma Priority (1);
end Task_1;

task Task_2 is
   pragma Priority (11);
end Task_2;

task Task_3 is
   pragma Priority (1);
end Task_3;

protected body Sporadic_Switcher is
   procedure Register(ID : Task_ID; Phase_1_CPU, Phase_2_CPU: CPU_Range;
                      Phase_1_Reserve, Phase_2_Reserve : Time_Span) is
   begin
      Client_ID := ID;
      Client_Phase_1_CPU := Phase_1_CPU;
      Client_Phase_2_CPU := Phase_2_CPU;
      Client_Phase_1_Reserve := Phase_1_Reserve;
      Client_Phase_2_Reserve := Phase_2_Reserve;
   end Register;

   procedure Handler(TM :in out Timing_Event) is
   begin
      case Client_Current_Phase is
         when Not_Released =>
            Set_CPU(Client_Phase_2_CPU, Client_ID);
            Switch_Timer.Set_Handler(End_of_Slot, Handler'Access);
            Client_Current_Phase := Phase_2;
            Released := True;
         when Phase_1 =>
            Client_Current_Phase := Suspended;
            Switch_Timer.Set_Handler(Start_of_Phase_2, Handler'Access);
            -- between slots - do nothing just set timer, alternative would
            -- be to lower priority to a "background" level priority
            -- more work conservative but we decided to maintain the algorithm as
            -- is in the original paper
            Hold(Client_ID); -- This call puts the task to sleep; it will not
                             -- execute on any CPU until “continue” has been
                             -- performed on it.
         when Suspended =>
            Set_CPU(Client_Phase_2_CPU, Client_ID);
            Switch_Timer.Set_Handler(End_of_Slot, Handler'Access);
            Client_Current_Phase := Phase_2;
            Continue(Client_ID);
         when Phase_2 =>
            Set_CPU(Client_Phase_1_CPU, Client_ID);
            Switch_Timer.Set_Handler(End_of_Phase_1, Handler'Access);
            Client_Current_Phase := Phase_1;
      end case;
   end Handler;


   procedure Release_Task is      -- called by someone else or by interrupt
      Number_of_Slots: Integer;
      Release_Time, Slot_Start: Time;
   begin
      -- calculate parameters

      Release_Time := Clock;
      Number_of_Slots := (Release_Time - Start_Time) / Time_Slot_Length;
      Slot_Start := Start_Time + Time_Slot_Length * Number_of_Slots;
      End_of_Phase_1 := Slot_Start + Client_Phase_1_Reserve;
      Start_of_Phase_2 := Slot_Start + Time_Slot_Length - Client_Phase_2_Reserve;
      End_of_Slot := Slot_Start + Time_Slot_Length;

      -- decide if release or not depending of phase
      if Release_Time >= Slot_Start and Release_Time < End_of_Phase_1 then
         Set_CPU(Client_Phase_1_CPU, Client_ID);
         Switch_Timer.Set_Handler(End_of_Phase_1, Handler'Access);
         Client_Current_Phase := Phase_1;
         Released := True;
      elsif Release_Time >= Start_of_Phase_2 and Release_Time < End_of_Slot then
         Set_CPU(Client_Phase_2_CPU, Client_ID);
         Switch_Timer.Set_Handler(End_of_Slot, Handler'Access);
         Client_Current_Phase := Phase_2;
         Released := True;
      else
         -- between slots - do nothing just set timer
         -- alternative would be to lower priority to a "background" level
         -- priority
         -- more work conservative but we decided to maintain the
         -- algorithm as is in the original paper
         Client_Current_Phase := Not_Released;
         Switch_Timer.Set_Handler(Start_of_Phase_2, Handler'Access);
      end if;
   end Release_Task;

   procedure Finished is
      Cancelled: Boolean;
   begin
      -- cancel the timer.
      Switch_Timer.Cancel_Handler(Cancelled);
   end Finished;

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

end Sporadic_Switcher;


task body Task_1 is
   Next : Time;
begin
```

```
   Next := Ada.Real_Time.Clock;
   Set_CPU( CPU_1 );
   loop
      Delay_Until_and_Set_Deadline( Next, Deadline_Task_1);
      -- Code of application
      Next := Next + Period_Task_1;
   end loop;
end Task_1;

My_Switcher: Sporadic_Switcher;

task body Task_2 is
begin
   My_Switcher.Register(Current_Task,
                        CPU_2, CPU_1,
                        Reserve_Phase_1_Task_2,
                        Reserve_Phase_2_Task_2);
   loop
      My_Switcher.Wait;
      -- Code of application
      My_Switcher.Finished;
   end loop;
end Task_2;

task body Task_3 is
   -- similar to Task 1
end Task_3;
```

We can make three observations. First, the non-split tasks, task 1 and task 3 have very simple code; they are basically programmed like we would have done if we wanted to implement partitioned EDF. Second, implementing task 2 requires some extra work. First of all, split tasks execute in the processor which they are currently allocated in preference to other tasks. Therefore, a priority level was created for the split task (priority 11), higher than the band for the regular EDF tasks.

Note also that the algorithm in [4] was designed for sporadic tasks, therefore a protected type is created to simultaneously control the release of the sporadic and to control the allocation of the task to the processors, depending on the phase within the slot. When the task is released (`procedure Release_Task`), first it is necessary to determine in what phase of the slot the release instant occurred. If it is within the interval reserved in a specific processor (Phase 1 – CPU 2; Phase 2 – CPU 1) then the task is allocated to that processor, and immediately released. Note that if the release instant is between the reserved slots, the task is not released. In all cases, a timer is armed for the next instant that the task attributes need to be changed.

When the timer handler is called, it changes the allocation of the task, or, if it is the end of the first phase, it needs to suspend the task with asynchronous task control. A better approach (for improving average responsiveness) would be to decrease the priority of the task to the EDF band (with a Deadline of `Time'Last`) or to create a background tasks lower priority band, which would allow the task to execute if the processor is idle. However, the task is suspended in order to maintain the equivalence to the algorithm of [4].

In the code-example above, we let tasks 1 and 3 arrive periodically and task 2 arrive sporadically. The algorithm allows any subset of tasks to arrive periodically and any subset of tasks to arrive sporadically however. For example, `Task_1` can easily be changed so that it arrives sporadically as well; changes needed for doing so are listed below:

```
protected PO_for_Task_1 is
   pragma Priority(1);
   procedure Release_Task;
   entry Wait;
private
   Released: Boolean := False;
end PO_for_Task_1;

protected body PO_for_Task_1 is
   procedure Release_Task is      -- called by someone else or by interrupt
   begin
      Released := True;
   end Release_Task;

   entry Wait when Released is
   begin
      Released := False;
   end Wait;
end PO_for_Task_1;

task body Task_1 is
begin
   Set_CPU( CPU_1 );
   loop
      PO_for_Task_1.Wait;
      -- Code of application
   end loop;
end Task_1;
```

Note that the protected object used for releasing Task 1 has the same priority (preemption level) of the task as we are assuming that the release event is only within CPU_1. If that was not the case, the preemption level would need to be higher [15] than the priority of Task 1, as in the case of the switcher protected object.

For arbitrary-deadline sporadic tasks, although different off-line scheduling algorithms are used [5], the algorithm for dispatching is the same as for implicit-deadline, with only the parameters being calculated differently.

5 Job-based split tasks dispatching for implicit-deadline sporadic tasks

The algorithm described in this section is the one in [9]. The text in the right column of page 3 in [9] describes the dispatching algorithm. The algorithm is based on configuring different priorities for each phase of the split task. The task starts to execute in one CPU, and after a certain clock time its affinity is changed to the second CPU, with a different priority.

To illustrate task splitting, we will consider the same task set as in Section 4; consider a task set $\tau=\{\tau_1, \tau_2, \tau_3\}$ to be scheduled on two processor. The tasks are characterized as T_1=100 ms, T_2=200 ms, T_3=400 ms, $D_1=100$ ms, $D_2=200$ ms, D_3=400 ms and $C_1=51$ ms, $C_2=102$ ms, C_3=204 ms. The approach in [9] uses a rule called HPTS (Highest Priority Task Splitting) and therefore, task τ_1 is split between processor 1 and processor 2. (Note that this is different from Section 4, where task τ_2 was split between two processors.). Task τ_2 is assigned to processor 1; task τ_3 is assigned to processor 2 and task τ_1 is split between processor 1 and processor 2.

The splitting of task τ_1 is done such that (i) the first piece of τ_1 has execution time 49 ms, relative deadline 49 ms and is assigned to processor 1 and (ii) the second piece of τ_1 has execution time 2 ms, relative deadline 51 ms and is assigned to processor 2. It is easy to see that the sum of the execution times of these pieces of task τ_1 is C_1 and the sum of the relative deadlines of the pieces of task τ_1 is D_1.

In this section, we formulate the algorithm with the new Ada constructs. For brevity we just show the main differences to the previous section.

The code is as follows:

```
pragma Task_Dispatching_Policy(FIFO_Within_Priorities);
with System; use System;
with Ada.Dynamic_Priorities; use Ada.Dynamic_Priorities;
-- includes and constants similar to the previous section
-- The constants are used for task 2
C_First_Phase   : constant Time_Span:=Milliseconds(49);
C_Second_Phase  : constant Time_Span:=Milliseconds( 2);
D_First_Phase   : constant Time_Span:=Milliseconds(49);
D_Second_Phase  : constant Time_Span:=Milliseconds(51);

Priority_Task1_First_Phase  : constant Priority := 20;
Priority_Task1_Second_Phase : constant Priority := 19;

Priority_Task2 : constant Priority := 18;
Priority_Task3 : constant Priority := 17;

protected body Job_Based_Switcher is
   procedure Register( ID : Task_ID; Phase_1_CPU, Phase_2_CPU: CPU_Range;
                       Phase_1_C, Phase_2_C, Phase_1_D, Phase_2_D: Time_Span;
                       Phase_1_Prio, Phase_2_Prio: Priority) is
   begin
      -- ... just update protected data
   end Register;

   procedure Handler(TM :in out Timing_Event) is
   begin
      -- in this algorithm, handler is just called in the end of phase 1
      Set_CPU(Client_Phase_2_CPU, Client_ID);
      Set_Priority(Client_Phase_2_Prio, Client_ID);
   end Handler;

   procedure Release_Task is
   begin
      -- calculate parameters

      Release_Time := Clock;
      End_of_Phase_1 := Release_Time + Client_Phase_1_D;

      -- set first phase parameters
      Set_CPU(Client_Phase_1_CPU, Client_ID);
      Set_Priority(Client_Phase_1_Prio, Client_ID);

      -- set timer
      Switch_Timer.Set_Handler(End_of_Phase_1, Handler'Access);

      -- release
      Released := True;
   end Release_Task;

   procedure Finished is
      Cancelled: Boolean;
   begin
      -- cancel the timer.
      Switch_Timer.Cancel_Handler(Cancelled);
   end Finished;
```

```
      entry Wait when Released is
      begin
         Released := False;
      end Wait;

   end Job_Based_Switcher;

   My_Switcher: Job_Based_Switcher;

   task body Task_1 is
   begin
      My_Switcher.Register( ... );
      loop
          My_Switcher.Wait;
          -- Code of application
          My_Switcher.Finished;
      end loop;
   end Task_1;
```

In this approach, the dispatching algorithm performs the migration at a certain time relative to the arrival of a job. Thus, both `Release_Task` and `Handler` procedures are much simpler. The first simply calculates the time to arm the timer, setting the parameters of the first phase (CPU and priority), whilst the second just changes these parameters. Note that we use fixed-priority scheduling of the task as proposed in [9].

6 Job-based split tasks dispatching for constrained-deadline sporadic tasks

The algorithm described in this section is the one in [8]. Figure 6 on page 6 in [8] gives a good description of the dispatching algorithm. The algorithm is very similar to the algorithm in Section 5 but differs in that (i) it uses EDF instead of RM on each processor and (ii) it performs migration when the job of a split task has performed a certain amount of execution. Therefore, there is no need for the mechanism to migrate the split task to know the arrival time of a job of a split task. Also, there are no timeslots.

The code is as follows:

```
   pragma Task_Dispatching_Policy(EDF_Across_Priorities);

   protected My_Job_Based_Switcher is
     pragma Priority(Ada.Execution_Time.Timers.Min_Handler_Ceiling);
     procedure Register(ID : Task_ID; Phase_2_CPU : CPU_Range);
     procedure Budget_Expired(T : in out Ada.Execution_Time.Timers.Timer);
   private
     Client_ID : Task_ID;
     Client_Phase_2_CPU : CPU_Range;
   end My_Job_Based_Switcher;


   protected body My_Job_Based_Switcher is
      procedure Register(ID : Task_ID; Phase_2_CPU: CPU_Range) is
      begin
         -- ... just update protected data
      end Register;

      procedure Budget_Expired(T : in out Ada.Execution_Time.Timers.Timer) is
      begin
         -- similarly to previous section,
         -- handler is just called in the end of phase 1
         Set_CPU(Client_Phase_2_CPU, Client_ID);
      end Budget_Expired;

   end My_Job_Based_Switcher;

   task body Task_2 is
      Next : Time;
      My_Id : aliased Task_Identification.Task_Id:= Task_2'Identity;
      The_Timer : Ada.Execution_Time.Timers.Timer(My_Id'Access);
      Cancelled: Boolean;
   begin
      My_Job_Based_Switcher.Register( ... );
      Next := Ada.Real_Time.Clock;
      --  note that we do not assign the task to any processor
      --  We will do it later in the loop below
      loop
         Delay_Until_and_Set_Deadline( Next, Deadline_Task_2);
         Set_CPU(Phase_1_CPU, My_ID);
         Ada.Execution_Time.Timers.Set_Handler(The_Timer, C_First_Phase,
                        My_Job_Based_Switcher.Budget_Expired'Access);
         -- Code of application
         Ada.Execution_Time.Timers.Cancel_Handler(The_Timer, Cancelled);
         Next := Next + Period_Task_2;
      end loop;
   end Task_2;
```

The code for execution-time monitoring that we use follows to some extent the idea on page 7 in [11]. But expiry of our handler for execution time monitoring does not need to notify the task (task 2) and this simplifies our code.

It should also be noted that the algorithm in [8] allows a task to be split between more than two processors. Our Ada code can be extended to that case by letting the handler `Budget_Expired` set up a new execution-time monitoring, with a call to `Set_Handler`.

7 Conclusions

We have seen details on how task splitting algorithms can be implemented using the recently proposed Ada extensions. In terms of efficiency, we expect this Ada code to be acceptable on multicores with a small number of cores. For very large multicores, the mechanism for migrating a task may impose a significant sequential bottleneck and for such platforms, a direct implementation of the task splitting algorithms in the Ada run-time may be needed. Also, if clocks and timers are available in just one (or in a reduced set of) processor(s), local timers are needed for better efficiency, particularly for more sophisticated algorithms such as the one in Section 4; an approach with user-defined clocks could be looked after. Nevertheless, multicore scheduling is still in its beginning, therefore it is still too soon to determine which algorithms to support.

Acknowledgments. This work was partially funded by the Portuguese Science and Technology Foundation (Fundação para a Ciência e a Tecnologia - FCT) and the European Commission through grant ArtistDesign ICT-NoE-214373.

References

1. Liu, C.L., Layland, J.W.: Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment", Journal of the ACM, vol. 20, pp. 46 - 61, 1973.
2. Burns, A., Wellings, A.: Supporting Execution on Multiprocessor Platforms". In Proc.of 14th International Real-Time Ada Workshop, 2009.
3. Andersson, B., Tovar E.: Multiprocessor Scheduling with Few Preemptions. In Proc.of The 12th IEEE International Conference on Embedded and Real-Time Computing and Application, pp. 322-334, 2006.
4. Andersson, B., Bletsas, K.: Sporadic Multiprocessor Scheduling with Few Preemptions. In Proc.of 20th Euromicro Conference on Real-Time Systems, pp. 243-252, 2008.
5. Andersson, B., Bletsas, K., Baruah, S.K.: Scheduling Arbitrary-Deadline Sporadic Task Systems on Multiprocessors". In Proc.of 29th IEEE Real-Time Systems Symposium, pp. 385-394, 2008.
6. Andersson, B., Bletsas, K.: Notional Processors: An Approach for Multiprocessor Scheduling". In Proc.of 15th IEEE Real-Time and Embedded Technology and Applications Symposium, pp. 3-12, 2009.
7. Bletsas, K. Andersson, K.: Preemption-light multiprocessor scheduling of sporadic tasks with high utilisation bound. In Proc.of 30th IEEE Real-Time Systems Symposium, 2009.
8. Kato, S., Yamasaki, N., Ishikawa, Y.: Semi-Partitioned Scheduling of Sporadic Task Systems on Multiprocessors. In Proc.of 21st Euromicro Conference on Real-Time Systems (ECRTS2009), pp. 249-258, 2009.
9. Lakshmanan, K., Rajkumar, R., Lehoczky, J.: Partitioned Fixed-Priority Preemptive Scheduling for Multi-Core Processors. In Proc.of 21st Euromicro Conference on Real-Time Systems, pp. 239-248, 2009.
10. Wellings, A., Burns, A.: Beyond Ada 2005: Allocating Tasks to Processors in SMP Systems; A. Wellings, A. Burns. In Proc.of 13th International Real-Time Ada Workshop, 2007.
11. "Ada Issue 307 Execution-Time Clocks", 2006, available at http://www.sigada.org/ada_letters/apr2006/AI-00307.pdf.
12. Brandenburg, B., Calandrino, J. Anderson, J.: On the Scalability of Real-Time Scheduling Algorithms on Multicore Platforms: A Case Study, Proceedings of the 29th IEEE Real-Time Systems Symposium, pp. 157-169, December 2008.
13. "Multiprocessor Systems Session Summary" at 14th International Real-Time Ada Workshop (IRTAW-14), Chairs: A. Burns and A.J. Wellings. Rapporteurs: A.J. Wellings and A. Burns.
14. "Managing affinities for programs executing on multiprocessor platforms", AI-167, 2009. Available at http://www.ada-auth.org/cgi-bin/cvsweb.cgi/ai05s/ai05-0167-1.txt?rev=1.1.
15. Rajkumar, R., Sha, L., Lehoczky, J.P: Real-time synchronization protocols for multiprocessors, In Proceedings 9th IEEE Real-Time Systems Symposium, pages 259–269, 1988.
16. Burns, A., Wellings, A.J.: Dispatching Domains for Multiprocessor Platforms and their Representation in Ada, 15th International Conference on Reliable Software Technologies - Ada-Europe 2010, Valencia, Spain, June 14-18, 2010.