Dispatching Domains for Multiprocessor Platforms and Their Representation in Ada

Alan Burns and Andy Wellings

Department of Computer Science
University of York, UK
{burns,andy}@cs.york.ac.uk

Abstract. Multiprocessor platforms are becoming the norm for more powerful embedded real-time systems. Although Ada allows its programs to be executed on such platforms it provides no explicit support. If Ada is going to be an effective language for multiprocessor real-time systems then it needs to address the mapping issue that will allow the programmer to express their requirements for task to processor affinity. A number of different mapping and scheduling approaches are advocated in the scheduling literature. The primitives in the language should allow these schemes to be directly supported. In this paper we propose extensions to Ada 2005 to introduce the notion of dispatching domains, and we show how these can be used to implement two example multiprocessor scheduling approaches.

1 Introduction

One of the challenges facing real-time systems is how to analyse applications that execute on multiprocessor systems. Although current schedulability analysis techniques are in their infancy, approaches are beginning to emerge. In this paper, we consider the support that Ada can give to allow applications to be able to benefit from these new techniques. The facilities described in this paper originated from ideas developed at the International Real-Time Ada Workshop (IRTAW 14) in September 2009[1].

An increasing number of embedded applications are now executed on multiprocessor and multicore platforms. For non-real-time programs, it is usually acceptable for the mapping of tasks to processor (we shall use the term CPU in this paper) to be implementation defined and hidden from the program. For real-time programs, this is not the case. *The control of affinities is as important as the control of priorities.*

A difficulty with targeting language abstractions to multiprocessor systems is that there is more than one multiprocessor architecture. In addition, the facilities that operating systems provide to exploit the parallelism in the architecture vary with no accepted standards. Moreover, there is more than one method of representing parallel code in languages. In this paper we consider symmetric multiprocessors – SMPs (homogeneous MPSoCs), we assume base-line operating system facilities such as those

[1] The IRTAW series of workshops are unlike most workshops in that ideas are developed from submitted position papers. One of the main themes of IRTAW 14 was to develop proposed extensions to Ada 2005 to allow programs to exploit multiprocessor systems. The related position papers are [6,9,11,15,13,14]. This paper refines and develops the initial proposals.

J. Real and T. Vardanega (Eds.): Ada-Europe 2010, LNCS 6106, pp. 41–53, 2010.

available with the Linux 2.6 kernel, and concurrency constructs such as the Ada task and protected object.

The paper is structured as follows. We first briefly review the current provisions within Linux and the Ada language for multiprocessor execution. We then consider the basic requirements that any change to the language must try and address. In Section 4 we propose a set of language modifications that are aimed at satisfying these requirements. Two illustrations of the usage of these facilities are provided in Section 5. Some conclusions are then given.

2 Current Facilities

2.1 Linux

Since kernel version 2.5.8, Linux has provided support for SMP systems [8] via the notion of CPU affinity. Each process in the system can have its CPU affinity set according to a CPU affinity mask. The CPU affinity mask of a process determines the set of CPUs on which it is eligible to run.

```
#include <sched.h>

int sched_setaffinity(pid_t pid, unsigned int cpusetsize,
    cpu_set_t *mask);

int sched_getaffinity(pid_t pid, unsigned int cpusetsize,
    cpu_set_t *mask);

void CPU_CLR(int cpu, cpu_set_t *set);
int CPU_ISSET(int cpu, cpu_set_t *set);
void CPU_SET(int cpu, cpu_set_t *set);
void CPU_ZERO(cpu_set_t *set);
```

A CPU affinity mask is represented by the `cpu_set_t` structure. Four macros are provided to manipulate CPU sets. `CPU_ZERO` clears a set. `CPU_SET` and `CPU_CLR` respectively add and remove a given CPU from a set. `CPU_ISSET` tests to see if a CPU is part of the set. The first available CPU on the system corresponds to a cpu value of 0, the next CPU corresponds to a cpu value of 1, and so on. A constant `CPU_SETSIZE` specifies a value one greater than the maximum CPU number that can be stored in a CPU set.

`sched_setaffinity` sets the CPU affinity mask of the process whose ID is `pid` to the value specified by the mask. If the process specified by `pid` is not currently running on one of the CPUs specified in `mask`, then that process is migrated to one of the CPUs specified in `mask`. This action is performed by a high priority kernel thread called the migration thread (there is one thread per CPU)[5]. If necessary, an interprocessor interrupt is sent to force a reschedule on another processor.

`sched_getaffinity` allows the current mask to be obtained.

An error is returned if the affinity mask contains no processors that are physically in the system, or if `cpusetsize` is smaller than the size of the affinity mask used by the kernel.

The affinity mask is actually a per-thread attribute that can be adjusted independently for each of the threads. The value returned from a call to `gettid` (get thread id) can be passed in the argument `pid`.

Two final points are worth making. The first is that unless the Real-Time Preemption patch is installed then preemptive priority-based scheduling across processor cannot be guaranteed. The second point is that the set of CPUs that are allocated to a process can be constrained externally to that process. Hence, the set passed to `sched_setaffinity` is filtered by the Linux kernel so that only the allowed CPUs are used. This allowed set can be changed asynchronously using the "cpuset virtual file system"[7].

Other operating systems provide slightly different facilities, but the Linux support is typical of what can be expected.

2.2 Ada

The Ada Reference Manual allows a program's implementation to be on a multiprocessor system. However, it provides no direct support that allows programmers to assign their tasks onto the processor in the given system. The following ARM quotes illustrate the approach.

> "NOTES 1 Concurrent task execution may be implemented on multicomputers, multiprocessors, or with interleaved execution on a single physical processor. On the other hand, whenever an implementation can determine that the required semantic effects can be achieved when parts of the execution of a given task are performed by different physical processors acting in parallel, it may choose to perform them in this way." ARM Section 9 par 11.

This simply allows multiprocessor execution and also allows parallel execution of a single task if it can be achieved, in effect, "as if executed sequentially".

> "In a multiprocessor system, a task can be on the ready queues of more than one processor. At the extreme, if several processors share the same set of ready tasks, the contents of their ready queues is identical, and so they can be viewed as sharing one ready queue, and can be implemented that way. Thus, the dispatching model covers multiprocessors where dispatching is implemented using a single ready queue, as well as those with separate *dispatching domains*." D.2.1 par 15.

This allows the full range of partitioning identified below. However, currently the only way that an implementation can provide the mechanisms to allow programmers to partition their tasks amongst the available processors is via implementation-defined pragmas, or non-standard library packages. For example, GNAT uses a pragma called `Task_Info` and an associated package `System.Task_Info` which provides platform specific information.

3 Basic Requirements

The primary real-time requirement for supporting the execution of Ada tasks on SMPs is to manage the mapping of tasks to processors. We assume that we are concerned with real-time code, in which case the execution of any task can be view as a sequence of invocations or *jobs*. Between jobs, the task is blocked, waiting either for an event (typically an external interrupt) or for a future time instance[2].

To cater for the allocation/mapping of tasks/jobs to processors, two basic approaches are possible:

1. **Fully Partitioned** – each task is allocated to a single processor on which all its jobs must run; and
2. **Global** – all tasks/jobs can run on all processors, jobs may migrate during execution.

There are many motivation for choosing either global or partitioned allocation, some of these motivations come from issues of scheduling [2]. These details are not significant here, what is important is that the Ada language should be able to support both schemes.

From these schemes, two further variants are commonly discussed: for global scheduling, tasks are restricted to a subset of the available CPUs; and for partitioned scheduling, the program can explicitly change a task's affinity and hence cause it to be moved at run-time.

Restricting the set of CPUs on which a task can be globally scheduled supports scalability – as platforms move to contain hundreds of CPUs, the overheads of allowing full task migration become excessive and outweigh any advantage that might accrue from global scheduling. Controlled changing of a task's affinity has been shown to lead to improved schedulability for certain types of application [12,1].

There may be other non-real time requirements for a mapping facility. For example, coscheduling where tasks *must* run in parallel is sometimes quoted as a requirement for high performance computing [10]. Alternatively, there is often a perceived need to have a group of tasks share access to a processor's data cache. The programmer doesn't care which processor they run on as long as it is the same processor. In both these cases, the tasks are tightly coupled and regularly exchange data. From a real-time perspective, the former can be achieved (approximately) by setting the priorities of the tasks to the same value. The later can only be achieved by fixing the tasks to a single processor (or a set of globally scheduled processors that have a common cache). We note that analysis models for these two use cases (with their strict definitions) have yet to be derived.

In the following discussions, in keeping with the terminology in the current Ada Standard, we will use the term *processor dispatching domain* (or just *dispatching domain*) to represent a group of processors across which global scheduling occurs. A task is said to be *assigned* to a dispatching domain; and if it is partitioned to execute on just one CPU, it said to be *set* to that CPU.

[2] Of course, this view of an Ada task is not explicitly enforced by the Ada language semantics. Rather, tasks should follow this convention to be amendable to schedulability analysis.

4 Language Modifications

We propose facilities to allow the processors allocated to an Ada program to be partitioned into a number of *non-overlapping* dispatching domains. Every task is scheduled within a dispatching domain. A task may also be assigned to execute on just one CPU from within its associated dispatching domain. We assume that the set of processors allocated to a program is not changed during its execution. Any external change to the set is outside the control of the Ada program and is likely to result in erroneous program execution. The following issues are addressed:

- representing CPUs and CPU sets,
- representing dispatching policies and dispatching domains,
- identifying interrupt affinities,
- ceiling priorities and locking.

Although the these proposals are designed to support SMP systems, the goal is that they should be applicable to more general systems such as those with different CPU speeds and/or those with non-uniform memory access times[3]. *All the facilities discussed here are on a per-partition basis.*

4.1 Representing CPUs and CPU Sets

First, a simple representation of the multiprocessor platform is required. A simple integer type is used to represent the range of CPUs (`CPU_Range`). These definitions are give here in a child package of System, although they could just be added to System:

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

Typically, operating systems provide a means of specifying a collection of CPUs using a bitset. Here, we provide a private type in a child package[4].

```
package System.Multiprocessors.CPU_Sets is

   type CPU_Set is private;
   Default_CPU_Set : constant CPU_Set;  -- includes all processors

   procedure Zero(Set: in out CPU_Set);
   procedure All_Set(Set: in out CPU_Set);
   procedure Set_One(Set: in out CPU_Set; Processor : CPU);
   procedure Clear(Set: in out CPU_Set; Processor : CPU);
   procedure Set_Many(Set: in out CPU_Set; Processors : CPU_Set);
```

[3] Platforms containing processors with different instruction sets are not considered here. The assumption is that they are best addressed using the Ada *partition* concept, and treated more like distributed systems than multiprocessor systems.

[4] As the Ada language evolves to support other architectures such as NUMA and cc-NUMA, further data structures may need to be introduced; for example, the processor domain and the node domains that are defined in Linux.

```
   procedure Clear(Set: in out CPU_Set; Processors : CPU_Set);
   function Is_Set(Set: in CPU_Set; Processor : CPU) return Boolean;

private
   ...
end System.Multiprocessors.CPU_Sets;
```

We note that for SMPs, a simpler representation is possible using just a subrange within `CPU_Range`.

4.2 Representing Dispatching Policies and Dispatching Domains

Dispatching Policies

Ada supports a range of dispatching options all from within a priority-based dispatching framework. These are defined in a hierarchy of packages rooted in the `Ada.Dispatching` package. Each of the policies can be applied across the whole priority range or within bands of priorities. The dispatching policy is set using pragmas and applies at the partition level. To extend this framework to allow the dynamic creation of dispatching domains, it is first necessary to have a more formal definition of a dispatching policy as a predefined type within the language. Here, we introduce this type (`Dispatching_Domain_Policy`) in the `Ada.Dispatching` package. We also define a subtype to represent the policies that can be used with priority-specific dispatching.

```
with System; use System;
package Ada.Dispatching is
   pragma Pure(Ada_Dispatching);

   Dispatching_Policy_Error : exception;

   type Dispatching_Domain_Policy is private.

   type Policy is (Priority_Specific_Dispatching,
                   Non_Preemptive_FIFO_Within_Priorities,
                   FIFO_Within_Priorities,
                   Round_Robin_Within_Priorities,
                   EDF_Across_Priorities);

   subtype Priority_Specific is Policy range
     FIFO_Within_Priorities .. EDF_Across_Priorities;

   procedure Set_Policy(DDP : in out Dispatching_Domain_Policy;
                        P : Policy);

   procedure Set_Priority_Specific_Policy(
               DDP : in out Dispatching_Domain_Policy;
               P : Priority_Specific; Low : Priority; High : Priority);
   -- raises Dispatching_Policy_Error if
   --   DDP has not been set to Priority_Specific_Dispatching, or
   --   High is not greater than Low, or
   --   any priority from Low to High had already been set
```

```
private
  -- not defined by language
end Ada.Dispatching;
```

A series of calls of the final procedure allows the program to construct the required priority-specific allocations.

We note that a more extendible representation of these policy types is possible.

Dispatching Domains

Although this proposal allows the dynamic creation of dispatching domains, there is one main restriction. This is the dispatching policy for the domain must be specified at the time the domain is created. Once specified it cannot be changed.

The following package (defined here as an extension to `Ada.Dispatching`) allows the group of CPUs to be partitioned into a finite set of non-overlapping `Dispatching_Domains`. One dispatching domain is defined to be the `System` dispatching domain; the environmental task and any derived from that task are allocated to the `System` dispatching domain by default. Subprograms are defined to allow new dispatching domains to be created and their scheduling policies defined.

Tasks can be assigned to a dispatching domain and be globally scheduled within that dispatching domain according to the defined scheduling policy for its priority level. Alternatively they can be assigned to a dispatching domain and set to a specific CPU within that dispatching domain. Tasks cannot be assigned to more than one dispatching domain, or set to more than one CPU.

```
with Ada.Task_Identification; use Ada.Task_Identification;
with Ada.Real_Time; use Ada.Real_Time;
with System.Multiprocessors; use System.Multiprocessors;
with Ada.Dispatching; use Ada.Dispatching;
with System; use System;
package Ada.Dispatching.Domains is

   type Dispatching_Domain is private;

   System_Dispatching_Domain : Dispatching_Domain;

   function Create(PS : in CPU_Set;
     DDP : in Dispatching_Domain_Policy) return Dispatching_Domain;
   -- Checks to see if the processors are in the system
   -- dispatching domain; if so, remove from system scheduling
   -- domain and add to the new domain, set the scheduling policy
   -- for the domain
   -- raise Dispatching_Policy_Error
   --    if the system cannot support global scheduling
   --      of the processors identified in PS, or
   --    if processors not in system dispatching domain, or
   --    if in system scheduling domain but has a task set, or
   --    if the allocation would leave the system dispatching domain
   --      empty, or
   --    if Dispatching_Domain_Policy has not been assigned
   function Get_CPU_Set(DD : Dispatching_Domain) return CPU_Set;
   function Get_Dispatching_Domain(T : Task_Id := Current_Task)
            return Dispatching_Domain;

   procedure Assign_Task(DD  : in out Dispatching_Domain;
                         T : in Task_Id := Current_Task);
   -- raises Dispatching_Domain_Error if T is already assigned
   -- to a dispatching domain

   procedure Assign_Task(DD  : in out Dispatching_Domain;
                         P : in CPU_Range;
                         T : in Task_Id := Current_Task);
   -- raises Dispatching_Domain_Error if P not in DD or if
   -- T is already assigned

   procedure Set_CPU(P : in CPU_Range; T : in Task_Id := Current_Task);
   -- raises Dispatching_Domain_Error if P not in current DD for T

   procedure Free_CPU(T : in Task_Id := Current_Task);

   function Get_CPU(T : in Task_Id := Current_Task) return CPU_Range;
   -- returns 0 if T is not set to a specific CPU

   procedure Delay_Until_And_Set_CPU(
       Delay_Until_Time : in Ada.Real_Time.Time; P : in CPU_Range);
   -- raises Dispatching_Domain_Error if P not in
   -- current DD for calling task

private
  -- not defined by the language
end Ada.Dispatching.Domains;
```

The required behaviour of each subprogram is as follows;

- `Create` – Creates a dispatching domain with a dispatching policy. The identified CPUs are moved from the `System` dispatching domain to this new domain. A CPU cannot be moved if it has a task assigned to it. The `System` dispatching domain must not be emptied of CPUs as it always contains the environmental task[5].
- `Get_CPU_Set` and `Get_Dispatching_Domain` – as their names imply.
- `Assign_Task` – There are two `Assign_Task` procedures. One allocates the task just to a dispatching domain (for global scheduling) the other allocates it to a dispatching domain and sets a specific CPU within that dispatching domain (for partitioned scheduling).
- `Set_CPU` – sets the task to a specified CPU. The task will now only execute on that CPU.
- `Free_CPU` – removes the CPU specific assignment. The task can now execute on any CPU within its dispatching domain.
- `Get_CPU` – returns the CPU on which the designated task is assigned.
- `Delay_Until_And_Set_CPU` – delays a task and then sets the task to the specified CPU when the delay expires. This is needed for some scheduling schemes.

[5] Note, there still is only one environmental task. There is no requirement for parallel elaboration of an Ada partition.

In addition to these two packages there are two new pragmas required to control the affinity of tasks during activation:

```
pragma Dispatching_Domain (DD : Dispatching_Domain);
pragma CPU (P : CPU_Range);
```

The following points should be emphasized.

– All dispatching domains have the same range of priorities (`System.Any_Priority`).
– The ‘`System`’ dispatching domain, `System_Dispatching_Domain`, is subject to the policies defined using the configuration pragmas: `Task_Dispatching_Policy` and `Priority_Specific_Dispatching`.
– A task has, by default, the dispatching domain of its parent. If the parent is assigned to a processor, then so is the child task.
– A task that wishes to execute, after a delay, on a different CPU and with a different deadline (for EDF scheduling) must use a Timing Event.

Finally, there are a number of implementation characteristic that must be documented, and there will be certain implementation advice useful to include in the ARM. For example the CPU(s) on which the clock interrupt is handled and hence where delay queue and ready queue manipulations (and user code - Timing Events) executed must be documented. As there is no scheduling between dispatching domains an implementation is recommended to have distinct queues per dispatching domain. If the Ada environment is being implemented on a system that has predefined dispatching domains, the details of these domains should also be documented.

4.3 Interrupt Affinities

Ada programs identify interrupt handlers using pragmas within protected objects. Although interrupts may be directed (on some architectures) to particular CPUs, the assumption here is that their Ada handlers (the associated protected objects) are accessible from all CPUs as they simply reside in main memory.

There is, however, a need to know on which CPU interrupt code will execute. Hence the following should be added:

```
function Get_CPU(I: Interrupt_Id) return CPU_Range;
-- returns 0 if interrupt is handled by more than one CPU
```

4.4 Ceiling Priorities and Locking

The current Ada mechanism for accessing protected objects from multiple CPUs is not fully defined by the language. Instead, implementation advice is given. In this, the assumption is that tasks will busy-wait (spin) at their active priorities for the lock (although other implementations are allowed). This is not changed by the proposals in this paper.

However, the non-lock optimization that is typically used in uniprocessor implementation is no longer viable on a multiprocessor platform. Generally, protected objects require a real lock in this environment unless all user tasks accessing the object are set to the same CPU.

The following guidelines on setting ceiling priorities can be given:

- For global scheduling – setting the ceiling priority of a protected object that is *only* accessed within a single dispatching domain can use the usual approach of setting ceilings to max priority of the accessing tasks plus 1 (note it must be plus 1 for global scheduling to work).
- Fixed tasks – Where tasks are fixed to a processor in the same dispatching domain, care must be taken and the interaction between tasks and protected object must be understood when setting the ceilings. It is probably safest to force non-preemptive execution of protected subprograms by setting the ceiling to the highest priority. It may also be advantageous to spin at this ceiling level as well.
- For protected objects shared between dispatching domains, the protected objects must run non-preemptively. This is because there is no relationship between the priorities in one allocation domain and those in another.

It should also be noted that on multiprocessor systems:

- Nested protected object locks can cause deadlock (there are some schemes in the literature to avoid this – for example for each chain another lock must be acquired first[4]).
- Chain blocking is possible.
- In the absence of deadlock, blocking can be bounded.

Programmer control over ceilings allows protocols such as non-preemptive execution of ‘shared’ POs to be programmed. No further language provision is required.

5 Examples

To illustrate the expressive power of the facilities advocated in this paper we will outline how two particular scheduling schemes can be programmed. Further examples of the use of the basic model is presented in [3], which can be found in the same proceedings as this paper.

5.1 Task-Splitting

Here we will use only the system dispatching domain but we will set tasks to execute on specific processors.

This scheme, called *task-splitting*, has gained some attention recently[12,1] as it attempts to combine the benefits of static and global partitioning. The scheme uses EDF scheduling on each CPU with task-partitioning for most tasks. A small number of tasks are however allowed to migrate at run-time, they execute for part of their execution time on one CPU and then complete on a different CPU. For N CPUs, there are N-1 split tasks.

Consider a dual-processor system with therefore just one split task. This task, `Split`, will be assumed to have a period of 20ms and a relative deadline equal to its period. The worst-case execution time of the task is 3.2ms. The splitting algorithm (the details are not relevant here) calculates that the task should execute on CPU 1 for 1.7ms (within a deadline of 5ms) and then switch to CPU 2 to execute its remaining 1.5ms (within its final relative deadline of 20ms).

It uses a `Timer` to change the task's processor allocation and deadline of the task once it has executed for 1.7ms. This is achieved by the following protected object:

```
with Ada.Dispatching.EDF; use Ada.Dispatching.EDF;
with Ada.Dispatching.Domains; use Ada.Dispatching.Domains;
...
protected Switcher is
  procedure Register(ID : Task_ID; E : Time_Span);
  procedure Handler(TM :in out Timer);
private
  Client : Task_ID;
  Extended_Deadline : Time_Span;
end Switcher;

protected body Switcher is
  procedure Register(ID : Task_ID; E : Time_Span) is
  begin
    Client := ID;
    Extended_Deadline := E;
  end Register;

  procedure Handler(TM :in out Timer) is
    New_Deadline : Deadline;
  begin
    New_Deadline := Get_Deadline(Client);
    Set_Deadline(New_Deadline + Extended_Deadline,Client);
      -- extends deadline by fixed amount passed in as E
    Set_CPU(2,Client);
  end Handler;
end Switcher;
```

The task itself would have the following outline.

```
pragma Task_Dispatching_Policy(EDF_Across_Priorities);
with Ada.Dispatching.EDF; use Ada.Dispatching.EDF;
with Ada.Dispatching.Domains; use Ada.Dispatching.Domains;
...

task Split is
  pragma Relative_Deadline(Milliseconds(5));
  pragma Priority (15); -- computed from deadline of task
  pragma CPU(1);
end Split.

task body Split is
  Id : Task_ID := Current_Task;
  Switch : Timer(ID'Access);
  Next : Time;
  First_Phase : Time_Span := Microseconds(1700);
  Period : Time_Span := Milliseconds(20); -- equal to full deadline
  First_Deadline : Time_Span := Milliseconds(5);
  Temp : Boolean;
```

```
begin
  Switcher.Register(ID,Period-First_Deadline);
  Next := Ada.Real_Time.Clock;
  loop
    Switch.Set_Handler(First_Phase,Switcher.Handler'Access);

    -- code of application

    Next := Next + Period;
    Switch.Cancel_Handler(Temp); -- to cope with task
                                 -- completing early (ie < 1.7ms)
    Set_Deadline(Next+First_Deadline);
    Delay_Until_And_Set_CPU(Next,1);
    -- a Timing Event could be used to combine the last two operations
  end loop
end Split;
```

5.2 Two Separate Domains

Here we consider a system with 8 or more processors, the first 4 are to be placed in one dispatching domain (the default) and will employ fixed priority dispatching (i.e. the default dispatching policy). The other processors will be placed into a second domain employing a partitioned scheme in which the lower 20 priorities are to be used for EDF scheduling and the others for fixed priority scheduling.

Here we provide the library package that will set up the second domain. The relevant predefined packages are assumed to be present.

```
procedure Set_Up(Second_Domain : out Dispatching_Domain);
  DP : Dispatching_Domain_Policy;
  CP : CPU_Set;
begin
  Set_Policy(PP, Priority_Specific_Dispatching);
  Set_Priority_Specific_Policy(PP, EDF_Across_Priorities,
    Priority'First, Priority'First + 20);
  Set_Priority_Specific_Policy(PP, FIFO_Within_Priority,
    Priority'First+21, Priority'Last);

  Zero(CP); -- clear mask
  for CPU in 5 .. Number_Of_CPUs loop
    Set_One(CP, CPU);
  end loop;

  Second_Domain := Create(CP,DP);
end Set_Up;
```

6 Conclusions

Historically, Ada has always taken a neutral position on multiprocessor implementations. On the one hand, it tries to define its semantics so that they are valid on a multiprocessor. On the other hand, it provides no direct support for allowing a task set to be partitioned. This paper has presented a set of facilities that could gain wide support and would help Ada system developers migrate their programs to what is becoming the default platform for embedded real-time systems.

The assumptions underlying the proposals made in this paper is that an Ada program has access to a fixed set of CPUs on the execution platform. Any external changes to the set of available processors is outside the control of the Ada program and is likely to result in erroneous program execution.

Acknowledgements

The ideas presented in this paper were initially suggested by the authors and then refined by the participants at the IRTAW 14 and in conjunction with the Ada Rapporteur Group. We gratefully acknowledge their contributions.

References

1. Andersson, B., Bletsas, K.: Sporadic multiprocessor scheduling with few preemptions. In: Euromicro Conference on Real-Time Systems (ECRTS), pp. 243–252 (2008)
2. Andersson, B., Jonsson, J.: Fixed-priority preemptive multiprocessor scheduling: to partition or not to partition. In: Proceedings of the International Conference on Real-Time Computing Systems and Applications (2000)
3. Andersson, B., Pinho, L.M.: Implementing multicore real-time scheduling algorithms based on task splitting using Ada 2012. In: Real, J., Vardanega, T. (eds.) Ada-Europe 2010. LNCS, vol. 6106, pp. 54–67. Springer, Heidelberg (2010)
4. Block, A., Leontyev, H., Brandenburg, B.B., Anderson, J.H.: A flexible real-time locking protocol for multiprocessors. In: Proceeding of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2007 (2007)
5. Bovet, D.P., Cesatí, M.: Understanding the Linux Kernel, 3rd edn. O'Reilly, Sebastopol (2006)
6. Burns, A., Wellings, A.J.: Supporting execution on multiprocessor platforms. In: Proceeding of the 14th International Workshop on Real-Time Ada Issues (IRTAW 14) (to appear)
7. Linux Kernel Documentations. CPUSets. Technical report (2008), `http://linux.mjmwired/kernel/Documentation/cpusets.txt`
8. Linux Manual Page. sched_setaffinity(). Technical report (2006), `http://linux.die.net/man/2/sched_setaffinity`
9. Michell, S., Wong, L., Moore, B.: Realtime paradigms needed post Ada 2005. In: Proceeding of the 14th International Workshop on Real-Time Ada Issues, IRTAW 14 (2005) (to appear)
10. Ousterhout, J.K.: Scheduling techniques for concurrent systems. In: Proceedings of Third International Conference on Distributed Computing Systems, pp. 22–30 (1982)
11. Ruiz, J.: Towards a ravenscar extension for multiprocessor systems. In: Proceeding of the 14th International Workshop on Real-Time Ada Issues (IRTAW 14) (to appear)
12. Shinpei, K., Nobuyuki, Y.: Portioned edf-based scheduling on multiprocessors. In: EMSOFT, pp. 139–148 (2008)
13. Wellings, A.J., Burns, A.: Beyond Ada 2005: allocating tasks to processors in smp systems. In: Proceedings of IRTAW 13, Ada Letters, vol. XXVII(2), pp. 75–81 (2007)
14. Wellings, A.J., Malik, A.H., Audsley, N.C., Burns, A.: Ada and cc-numa architectures. what can be achieved with Ada 2005? In: Proceeding of the 14th International Workshop on Real-Time Ada Issues (IRTAW 14) (to appear)
15. White, R.: Providing additional real-time capabilities and flexibility for Ada 2005. In: Proceeding of the 14th International Workshop on Real-Time Ada Issues (IRTAW 14) (to appear)