Supporting Execution on Multiprocessor Platforms

A. Burns and A.J. 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 tasks to be executed on such platforms it provides no explicit support that allows programmers to map tasks to processors. If Ada is going to be an effective language for multiprocessor 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 a minimum set of such primitives, with the aim of initiating a debate that will lead to an agreed language change that can be accepted within the Ada community.

1 Introduction

One of the challenges facing real-time systems is how to analyse applications that execute on multiprocessor systems. Current schedulability analysis techniques are in their infancy, however, approaches are beginning to emerge. In this paper, we consider the support that Ada would need to give to allow applications to be able to benefit from these new techniques. We make no claims of whether this support can be implemented on current real-time multiprocessor operating systems.

A difficulty with this challenge is that there is more than one multiprocessor architecture. Moreover, there is more than one method of representing parallel code in languages. In this paper we consider symmetric multiprocessors – SMPs (e.g. homogeneous MPSoCs) and concurrency constructs such as the Ada task and protected object. The analysis builds upon that presented at the last IRTAW [9], but focusses on a minimum set of facilities needed to program schedulable real-time systems.

2 Basic Requirements

The primary requirement for supporting the execution of Ada tasks on a homogeneous SMPs is to manage the mapping of tasks to processors and to support inter-task communication. 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 of *jobs*. Between jobs the task is blocked waiting either for an event (typically an external interrupt) or for a future time instance. Jobs do not suspend themselves during their execution (they can of course be preempted). Inter-task communication is supported by protected objects.

To cater for the allocation/mapping of tasks/jobs to processors three basic approaches are possible, and should be supported at the language level:

1. **Task-Partitioning** – each task is allocated to a single processor on which all its jobs must run.
2. **Job-Partitioning** – all tasks can run on all processors; but once a job has started executing on one processor it must stay there.

3. **Global-Partitioning** – all tasks/jobs can run on all processors, jobs may migrate during execution.

Typically a task will loose any cache allocation between jobs, but on certain architectures the overhead of moving the execution of a job from one processor to another is prohibitive and hence the need to support job-partitioning. 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 all such schemes.

In the following discussions we will use the term *partitioning* to imply a mapping of a task to a CPU. This is the standard term used in multiprocessor scheduling work. Confusingly the Ada language uses the term *partition* as part of its support for distributed programming. If some form of partitioning is incorporated into the language definition then some reworking of the nomenclature will be needed.

3 Current Facilities

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 partition 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 above. However, currently the only way that an implementation can provide the mechanisms to allow the programmers to partition their tasks amongst the available processors is via implementation-defined pragmas, or non standard library packages.

For protected objects, the following points are made in the Annotated Ada Reference Manual[1].

- There is no language-defined ordering or queuing presumed for tasks competing to start a protected action. The intended implementation on a multi-processor is that tasks use spin locks (i.e. busy-wait). (par 19.a p355)
- In defining when modifications to shared variables that are outside a protected action become visible to tasks other than the modifier, Ada has the notion of signalling. Protected subprogram calls are not defined to signal one another, which means that such calls alone cannot be used to synchronize access to shared data outside of a protected object. "Reason: The point of this distinction is so that on multiprocessors with inconsistent caches, the caches only need to be refreshed at the beginning of an entry body, and forced out at the end of an entry body or protected procedure that leaves an entry open. Protected function calls, and protected subprogram calls for entryless protected objects do not require full cache consistency. Entryless protected objects are intended to be treated roughly like atomic objects each operation is indivisible with respect to other operations (unless both are reads), but such operations cannot be used to synchronize access to other nonvolatile shared variables." (par 9.c page 368)

[1] see ISO/IEC 8652:1995(E), http://www.adaic.com/standards/05aarm/html/AA-TTL.html

3.1 Facilities Provided by GNAT

GNAT uses a pragma called `Task_Info` and an associated package `System.Task_ Info` which provides platform specific information.

With a simple model the pragma is used to identify either global-partitioning (use of the parameter -1) or task-partitioning in which case the parameter designates the actual CPU to use – the CPUs are numbered from 1 to some constant maximum. The support package is of the following form.

```
package System.Task_Info is
   pragma Preelaborate;
   pragma Elaborate_Body;

   subtype CPU_Number is System.OS_name.ProcessorId;

   Any_CPU : constant CPU_Number := -1;

   Invalid_CPU_Number : exception;
   --  Raised when an invalid CPU number has been specified
   --  i.e. CPU > Number_Of_Processors.

   type Thread_Attributes is record
      CPU : CPU_Number := Any_CPU;
   end record;

   Default_Thread_Attributes : constant Thread_Attributes :=
                               (others => <>);

   type Task_Info_Type is access all Thread_Attributes;

   Unspecified_Task_Info : constant Task_Info_Type := null;

   function Number_Of_Processors return Positive;
   --  Returns the number of processors on the running host

end System.Task_Info;
```

Similar packages give support to the mapping of tasks to particular primitives such as threads or processes in specific OSs. In some situations the instruction in the pragma is deemed to be absolute (if it cannot be satisfied the program must be rejected) in others it is advisory:

> ... provides a way to define the ideal processor to use for a given thread. The ideal processor is not necessarily the one that will be used by the OS but the OS will always try to schedule this thread to the specified processor if it is available.

A more expressive interface is available in GNAT for some platforms in which a range of possible CPUs is identified via a set of boolean values and affinities:

```
package System.Task_Info is
  pragma Preelaborate;
  pragma Elaborate_Body;

  subtype CPU_Set is ...;

  Any_CPU : constant CPU_Set := (bits => (others => True));
  No_CPU  : constant CPU_Set := (bits => (others => False));

  Invalid_CPU_Number : exception;
  --  Raised when an invalid CPU mask has been specified
  --  i.e. An empty CPU set

  type Thread_Attributes is record
```

```
    CPU_Affinity : aliased CPU_Set := Any_CPU;
  end record;

  Default_Thread_Attributes : constant Thread_Attributes :=
                              (others => <>);

  type Task_Info_Type is access all Thread_Attributes;

  Unspecified_Task_Info : constant Task_Info_Type := null;

  function Number_Of_Processors return Positive;
  --  Returns the number of processors on the running host
end System.Task_Info;
```

For protected object accesses, GNAT tends to rely on the underlying OS facilities for implementing mutexes.

4 Language Requirements

Although GNAT does provide various levels of support, there is a need for uniformity in this provision, and for a portable scheme to be supported within the language definition.

In a paper at the previous workshop we presented [9, 5] an extensive set of facilities that could deal with various levels of OS autonomy. Here we concentrate on a simpler provision, where the programmer is in full control of all allocations. Our motivation is to support real-time applications that must manage allocation in the same way that they must manage priority assignment. For non-real-time applications, round robin scheduling and global partitioning under OS control is acceptable; for real-time systems behaviour must be constrained. This is especially true for hard real-time systems. With Ada this means embedding scheduling and allocation information within the program.

In this paper we restrict ourselves to identical processors; hence it is not necessary to identify the actual CPUs, a simple integer ordering is sufficient – we return to this issue again in Section 6. Also we assume that the available memory is homogeneously accessible from every task. We first consider task mapping issues and then protected object.

4.1 Mapping Tasks to Processors

The programmer's requirements of the implementation language is as follows (for some arbitrary task `T`):

1. Map `T` to a specific CPU – Task-Partitioning
2. Map `T` to one of a group of CPUs – Task-Partitioning
3. Map `T` to a group of CPUs – Global-Partitioning
4. Map `T` to a group of CPUs – Job-Partitioning
5. Map `T` to the same CPU as some other task `S` – Task-Partitioning

In general this mapping information can be held as a record:

```
type Partition_Mode is (Task_Partitioning, Global_Partitioning,
                        Job_Partitioning);
type CPU_Set is -- a set of boolean;

type Task_Map is record
  Mode : Partition_Mode;
  Affinity : CPU_Set;
end record;
```

Each task should hold this information as an attribute (as in the above GNAT examples).

These types could be declared in a support package (`Ada.System.Task_Partitioning`) that also contained other useful routines:

```
Any_CPU : constant CPU_Set := ((others => True));
No_CPU  : constant CPU_Set := ((others => False));

Invalid_CPU : exception;
  --  Raised when an invalid CPU mask has been specified
  --  i.e. an empty CPU set

function Number_Of_CPUs return Positive;
  --  Returns the number of processors on the running host

function CPU(T : Task_ID := Current_Task) return CPU_Set;
  -- Returns the actual allocation of a running task, or
  -- the allowable allocations for a blocked task
```

More generally the following schemes are required.

1. The facility to program mixed systems where different forms of partitioning are used on different CPUs in the same platform.
2. The facility to program changes in the partitioning mode and actual mapping to processors.
3. To be able to determine where a running task is allocated and which CPUs are active.

So, for example, on a quad-processor system, a set of tasks with hard real-time constraints could be statically allocated to the first processor, and all other tasks be allocated as global-partitioned over the other processors.

To give an initial assignment, a pragma is appropriate,

```
pragma Affinity(TM : Task_Map);
```

or possible two distinct pragmas:

```
pragma Partition_Mode(PM : Partition_Mode);
pragma CPU_Map(Map : CPU_Set);
```

The default for all tasks is `Global_Partitioning` and `Any_CPU`.

Although the definition of these types includes some redundancy (for example `Task_Partitioning`, `Global_Partitioning`, `Job_Partitioning` are all the same if only one CPU position has the value `True` in `CPU_Set`) they allow all the combinations identified earlier to be represented.

To alter an allocation during a task's execution requires a procedural interface. Note just changing a task's attributes will not be sufficient as this will not necessarily force the run-time to respond to the implied changes. A procedure such as the following could be defined in a support package (that has 'withed' `Ada.System.Task_Identification`).

```
procedure Set_Affinity(TM : Task_Map; T : Task_ID := Current_Task);
```

A call of this procedure would necessarily be a dispatching point.

The dual procedure would also naturally be provided in this package:

```
procedure Get_Affinity(TM : out Task_Map; T : Task_ID := Current_Task);
-- or
function Get_Affinity(T : Task_ID := Current_Task) return Task_Map;
```

To deal with processor failures, a handler needs to be executed that is invoked by the run-time. This would imply that it must be possible to identify those tasks (and jobs) that are currently statically mapped to any particular CPU.

It is however questionable if the basic level of support, motivated in this paper, should go as far as dealing with processor failure. The state of any task running on a failing processor may be difficult to ascertain, and hence recovered at the program level may not be programmable.

4.2 Protected Object Issues

There are several issues that need to be considered when accessing protected objects on multiprocessor systems. Here we focus on two: guaranteeing atomicity and increasing parallelism.

Guaranteeing atomicity

For single processor systems, the Ada use of the immediate ceiling priority protocol enables atomicity to be provided by the priority model itself, no additional lock is required. Clearly, on a multiprocessor system this no longer holds, and Ada suggests that an implementation should use spin-locks. However, in keeping with its global scheduling philosophy, it provides no mechanisms that allow the programmer to indicate whether a protected object will be accessed locally or globally. In this paper, the programmer may partition the task sets, and consequently, groups of tasks and protected objects may be local. Hence, there may be some value in allowing the programmer to indicate that a particular protected object could rely on the ceiling model. A pragma can be used for this purpose:

```
pragma Local_Ceilings;
-- to be used inside the specification of a PO.
-- The default would be globally accessible
```

In order to ensure ceilings work correctly when protected objects are shared between tasks executing on different processors, then the ceiling must be greater that all the tasks on all the processors from which the protected object can be accessed [4]. Hence, protected actions have to be effectively non preemptible.

Increasing Parallelism

Lock-based synchronization protocols are often criticised for inhibiting parallelism. Here we focus on two aspects. The first was raised at the last workshop by Ward and Audsley [8]. Here the idea is to have the notion of *entry functions*. An entry function is like a normal protected object function but it can have a barrier. However, we note here that in order to maintain the read-only status of the protected data during function execution, the `'Count` attribute should not apply to function entries.

The second aspects concerns whether alternative implementation approaches for protected objects should be specified to allow more optimistic concurrence control mechanisms to be used (for example transactional memory[6]). Given that Ada's protected actions are meant to be short, it is not clear whether the benefits here outweigh the additional complexity of the run-time system. However, as noted in the previous subsections, for global scheduling most protected actions have to be run non-preemptively.

5 Scheduling

The allocation schemes must be compatible with the scheduling schemes; which for Ada 2005 means the defined dispatching policies: fixed priority, EDF and round-robin, and the mixed dispatching approach. The Ada model (though not necessarily the algorithm to be followed by the implementation) is for each processor to have a set of *ready queues* of tasks that are waiting to execute on that CPU. One ready queue per priority. A globally mapped task would be on a ready queue for each of its CPU affinities. When a task is chosen for execution it is removed from all ready queues. A task-partitioned task would only ever be on one CPU's ready queue. A job-partitioned task would, when released from a blocked state, be placed on a ready queue for each of its potential CPUs (unless its affinity has changed). Once it starts executing however is it bound to that CPU; so if it is preempted it will be placed on the front of the ready queue for its priority on just that one CPU.

Full control of scheduling and partitioning is provided by the three languages features:

1. allocation of priority levels to dispatching policies,
2. allocation of each task to a priority level, and

3. allocation of each task to a CPU (or set of CPUs).

If tasks do not change their priority or CPU affinity then various static properties can be validated. For example, if one CPU is designated for EDF scheduling only then it is sufficient to check that no task mapped to that processor has a priority outside a `EDF_Across_Priority` range. It may be possible to build a more usage API to the language primitives discussed earlier, but they do seem to posses sufficient expressive power to be of real utility in the domain of real-time multiprocessor systems.

To illustrate the expressive power of the facilities advocated in this paper we will outline how a particular scheduling scheme could be programmed. This scheme, called *task-splitting*, has gained some attention recently[7, 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 (N-1 if there are N CPUs) 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.

Consider a dual-processor system with therefore just one split task. This task, `Split`, will be assumes 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).

The task would have the following outline. It uses a `Timer` to change the affinity and deadline of the task once it has executed for 1.7ms.

```
pragma Task_Dispatching_Policy(EDF_Across_Priorities);

with Ada.Execution_Time; use Ada.Execution_Time;
with Ada.Execution_Time_Timers; use Ada.Execution_Time_Timers;
with Ada.Task_Identification; use Ada.Task_Identification;
with Ada.Task_Partitioning; use Ada.Task_Partitioning;
with Ada.Real_Time; use Ada.Real_Time;
with Ada.Dispatching.EDF; use Ada.Dispatching.EDF;
with System; use System;

...

-- the following PO defined in a library package
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 from 5ms to 20ms
    Set_Affinity((Task_Partitioning,(False,True)),Client);
  end Handler;
end Switcher;
```

```
...

task Split is
  pragma Affinity((Task_Partitioning,(True,False))); -- or
  pragma Partition_Mode(Task_Partitioning);
  pragma CPU_Map((True, False));

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

task body Split is
  ID : aliased 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_Affinity((Task_Partitioning,(True,False)));
    Delay_Until_and_Set_Deadline(Next,First_Deadline);
  end loop;
end Split;
```

Interestingly this example indicated that a facility similar to `Delay_And_Set_ Deadline` is needed here. This delay statement allows a task to 'wake up' from a delay statement with a different deadline to that in effect when the task executed the delay statement. The above ideally needs a delay statement that allows the task to 'wake up' on a different processor to that on which the delay statement was executed. Alternatively, the semantics for a call of `Set_Affinity` could be 'switch when the task is next blocked or preempted'. To get an immediate switch a task would have to execute 'delay 0.0' immediately after the call to `Set_Affinity`. Unfortunately this delay statement could not be placed in a protected procedure – as needed above.

We can expand on this example to program another common scheme called *striping* or *slicing*. Here a task executes for a quantum of execution time and then switches to the next processor, after a further quantum the task moves on again [3]. Thus the task cycles though all the processors, but it can start on any processor. Assume the quantum is 0.1ms and that there are 8 processors:

```
type CPU_Range is 0 .. Number_Of_CPUs;

  -- code for Timer handler:
  procedure Handler(TM :in out Timer) is
    Temp : CPU_Set := No_CPU;
    No : CPU_Range;
  begin
    No := CPU(Client);
    if No = CPU_Range'last then
      No := 1;
    else
      No := No + 1;
    end if;
```

```
    Temp(No) := True;
    Set_Affinity((Task_Partitioning,Temp),Client);
    TM.Set_Handler(Microseconds(100),Switcher.Handler'Access);
  end Handler;

task Stripper is
  pragma Affinity((Task_Partitioning,Any_CPU)); - or
  pragma Partition_Mode(Task_Partitioning);
  pragma CPU_Map(Any_CPU);

  pragma Priority(Same_For_All_Tasks);
end Stripper.

task body Stripper is
  ...
begin
  Next := Ada.Real_Time.Clock;
  loop
    Switch.Set_Handler(Microseconds(100),Switcher.Handler'Access);
    -- code of application
    Next := Next + Period;
    Delay_Until_and_Set_Deadline(Next,First_Deadline);
  end loop;
end Stripper;
```

6 Beyond SMPs

The above discussion has assumed that the processors are identical (SMPs) and hence all tasks can run on all CPUs. Even if the latter property remains true, the multiprocessor architecture may not be fully homogeneous. For example, a 16 processors system may be composed of 4 clusters of 4 CPUs. Within a cluster there is cache coherence and so global partitioning, within the cluster, is allowed (and indeed encourage to improve scheduling) – but not between clusters. The CPU map for this architecture would just be a subtype 1..16. The cluster structure is not visible. To counter this problem some means of representing clusters should be provided – or a means of validating the run-time behaviour. For example an implementation should be allowed to raise an exception if a set of affinities are proposed for global-partitioning that are not compatible with run-time performance restrictions.

If one moved even further away from SMPs then many more issues become relevant. If not all memory is visible from all CPUs then the mapping of variables of all types as well as task and protected objects must be controlled. Memory pools would seem to have a role here – these issues go beyond the focus of this paper but are addressed in another submission [10].

7 Open Issues

If the definition of Ada is to incorporate the kind of support proposed in this paper then a number of issues need to addressed and a full proposal made. For example:

- Is it reasonable to ignore processor failures?
- Are the provisions for Task-, Job- and Global-Partitioning adequate?
- Is is useful to allow a task to be designated as Task-Partitioned but allow the run-time to choose which CPU to map it to – from amongst those specified.
- Is `Get_Affinity` sufficient? – or is the `CPU` routine also needed?
- What exactly are the semantics for `Get_Affinity` and `CPU`?

- How are group budgets to be managed on multiprocessor platforms?

8 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 minimum 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.

Acknowledgements

The authors gratefully acknowledge the input of Pat Rogers during the writing of this paper.

References

[1] B. Andersson and K. Bletsas. Sporadic multiprocessor scheduling with few preemptions. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 243–252, 2008.

[2] B. Andersson and J. Jonsson. 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] S. Baruah, N. Cohen, G. Plaxton, and D. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15(6):600–625, 1996.

[4] R. Rajkumar, L. Sha, and J.P. Lehoczky. Real-time synchronization protocols for multiprocessors. In *Proceedings 9th IEEE Real-Time Systems Symposium*, pages 259–269, 1988.

[5] J. Real and S. Mitchell. Beyond ada 2005 session report. In *Proceedings of IRTAW 13, Ada Letters, XXVII(2)*, pages 124–126, 2007.

[6] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, pages 204–213, 1995.

[7] K. Shinpei and Y. Nobuyuki. Portioned edf-based scheduling on multiprocessors. In *EMSOFT*, pages 139–148, 2008.

[8] M. Ward and N.C Audsley. Suggestions for stream based parallel systems in ada. In *Proceedings of IRTAW 13, Ada Letters, XXVII(2)*, pages 33–138, 2007.

[9] A.J. Wellings and A. Burns. Beyond ada 2005: allocating tasks to processors in smp systems. In *Proceedings of IRTAW 13, Ada Letters, XXVII(2)*, pages 75–81, 2007.

[10] A.J. Wellings, A.H. Malik, N.C. Audsley, and A. Burns. Ada and cc-NUMA architectures: What can be done with Ada 2005. In *submitted to IRTAW 14*, 2009.