Multiprocessor Systems
Session Summary

Chairs: A. Burns and A.J. Wellings
Rapporteurs: A.J. Wellings and A. Burns

Abstract

This report summarizes the discussion held at Fourteenth International Workshop on Real-Time Ada Issues (IRTAW 14) on how to provide better support for multiprocessor systems in Ada.

1 Introduction

The whole first day of the workshop was dedicated to a discussion of multiprocessor issues. The discussions were partitioned into two sub sessions. The first was chaired by Alan Burns and covered the following topics:

- scope of the issues to be addressed;
- current state of real-time multiprocessing scheduling;
- the development of a proposed model for Ada, including both task allocation, protected object access protocols and interrupt handling.

The second session was chaired by Andy Wellings and addressed the following issues:

- Execution-Time Clocks and Timers
- Group Budgets
- Ravenscar Issues
- Non SMP Architectures

Ada 2005 is about to undergo further updates and any proposals for changes have to be raised by the end of October 2009. Consequently, although the overall goal of the sessions was to consider a wide range of issues, it was agreed that the focus should be on developing better support for real-time scheduling on multiprocessor systems.

2 Scope of Issues

Early on in the discussions it was reaffirmed that traditionally the unit of concurrency in Ada had always been coarse grained and expressed via the task construct. Although for high-performance computing there may be some need to be able to express data-level parallelism, the workshop focus was real-time and it was agreed that we should only consider tasks.

It was further agreed that Ada 2005 currently does already support multiprocessor systems and that nothing we proposed should undermine the current specification, unless it was inherently broken. In the absence of any directives in the program, the behaviour of the program should be that currently defined in the Ada Language Reference Manual. The workshop took the Ada model to be essentially the following:

- all tasks executing in a partition must be able to access shared memory; and
- scheduling between tasks in a partition is global, hence when a processor becomes available it must be given to the highest priority runnable task.

For the above reason, the workshop decided to focus on multiprocessor systems that have access to shared memory and are symmetric, i.e. SMP architectures. There was some discussion on whether the IO space of an SMP architecture could be accessed from all processors. It was assumed that in general all memory locations/device registers could be assessed from all CPUs and that only interrupt may be constrained to certain CPUs.

No assumptions were made about the speed of the individual CPUs. Hence, hyperthreading architectures were in scope.

3 State of the Art in Multiprocessor Scheduling

Alan summarized the possibilities for real-time multiprocessor scheduling. The focus was on where tasks can run.

- Fully global partitioning – Any task can run on any of the available CPUs. The scheduler decides order and placement. Typically these require a global ready queue and there are some concerns about whether this can be implemented efficiently and whether it is scalable to large systems.
- Task partitioned – Each task can only run on one CPU, this being specified by the programmer. However, the specified CPU could be changed at run-time.
- Job partitioned – When a task is released it can run on any CPU, however preempted tasks return to the CPU on which they were initially allocated. Hence, the tasks can migrate only at release time.

It was agreed that the state of the art in multiprocessor scheduling was not mature enough to focus on supporting any particular approach. However, it was agreed that some form of both global and partition scheduling is needed. The reason for this is that better schedulability can be obtained by fixing CPU intensive tasks to a single processor and allowing the others to migrate. Also a task may need to be on a particular processor to service a device's interrupts.

4 The Proposed Model

During the next few hours, discussions were held on what was an appropriate model for Ada. The model that emerged can be summarized by the following points:

- The introduction of the notion of an **allocation domain**[1]. In the multiprocessor literature, allocation domains are sometimes called clusters. The key point about an allocation domain is that it defines a set of CPUs on which tasks are globally scheduled.
- There is a *default* allocation domain (the `System` domain) that is the set of all processors allocated to a program. *The assumption is that this is a fixed set of processors that does not change during the execution of the program.* The default behaviour of an Ada program if it does not specify anything is therefore global scheduling of all tasks across all processors. Depending on the run-time infrastructure, tasks may migrate between processors only at release time (called job-level allocation in Section 3) or within a release (called fully global allocation in Section 3)[2].

[1]We use this term as being short for scheduling or dispatching allocation domain

[2]During the workshop Björn Anderson considered the job partitioning approach and showed, from a scheduling point of view, that the approach could lead to very poor performance. The workshop therefore restricted its consideration to the task and global partitioning approaches.

- Allocation domains *cannot* overlap; that is no CPU can be in more than one allocation domain. The reason for this constraint is to ease schedulability analysis and to support efficient global run queues and scaling of the system to hundreds of processors.
- The model allows the programmer to constrain a task to execute on *only* one CPU in an allocation domain. This was seen as more useful than allowing a task to be fixed to a subset of the CPUs in the allocation domain. Also some current real-time scheduling practices require this.

The following issues were also discussed

- **Failure semantics:** Currently Ada has an implicit semantics of crash failures. It was decided that although the workshop would ideally like to support programs executing on platforms with partial failures, this was too big an issue to address within the current time constraints and the absence of any position papers on the topic.
- **Migration of tasks across allocation domain:** Should a task be able to automatically migrate across allocation domains? The main reasons for allowing this might be load shedding during a transient overload, or reconfiguration following a partial failure.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 will be allowed). This was not changed by the It was decided (mainly for simplicity of feasibility analysis) that migration between allocation domains should not be supported but that the domain of a task could be changed by the programmer at run time.

4.1 Protected Objects

There was some discussion on whether protected objects should be given allocation domains and potentially fixed to specific processors in those domains. The main requirement for this was device access. However, it was agreed that in the model under consideration all device registers were accessible for all CPUs and that interrupts, had “affinities” not protected objects.

The rules for setting the ceiling priorities were deemed by the workshop not to be part of the language. However, for completeness they were given.

- For fully global scheduling – setting the ceiling priority of a protected object that is *only* accessed within a single allocation 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 the global scheduling to work).
- Fixed tasks – Where tasks are fixed to a processor in the same allocation 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-pre-emptive execution of protected subprograms.
- If the underlying platform only supports job-level scheduling then all protected objects shared across processors should be accessed non-preemptively.
- For protected objects shared between allocation domains, the protected objects must run non pre-emptively. This is because there is no relationship between the priorities in one allocation domain and those in another.
- A lock is always required; using the priority model for locking is not sustainable with multiprocessors (unless it is possible to show that a protected object is only accessed from one processor).

It was also 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)
- Chain blocking is possible.
- In the absence of deadlock, blocking can be bounded.

4.2 Interrupt Handling

In the Ada model, interrupts are mapped to protected procedure calls. Typically these have ceiling set to the hardware priority of the interrupt. The workshop discussed at length how best to ensure mutual exclusive access to interrupt handling protected objects. Various models were considered, including migrating a task to the site where the interrupt is delivered and using the priority model, or disabling/masking the interrupt. In the end it was agreed that it is the run-time responsibility to ensure mutual exclusion of the protected object in the presence of user tasks calling the procedures.

The workshop did agree however that the 'affinity' of an interrupt should be available to the program so that a 'released' task can be co-located on the same CPU.

5 Execution-Time Clocks and Timers

The workshop discussed whether there were any multiprocessor issues with execution-time clocks and timers.

It was agreed that:

- Measuring execution times presented no additional problems on a multiprocessor as tasks can only be active on one CPU at a time.
- Timers similarly should be no problem. The interrupt from a timer may be constrained to be handled on one processor, but this simply required that the associated PO's subprograms execute non-preemptively.
- *Deciding what the values should be for the WCET on a multiprocessor system is problematic, particularly if processors run at different speeds.* However, this is not a language or a run-time issue. If was suggested that if the programmer was enforcing execution times of a task then its allocation domain should perhaps be set so that the task runs at a uniform speed.

6 Group Budgets

Supporting group budgets in a multiprocessor system is fraught with difficulties. Andy in his overview of the topic gave three approaches:

1. Group budgets can be active on many processors and processors may have variable speed.
2. Group budgets can be active on many processors but processors must have the same speed.
3. Group budgets can only be active on one processor at a time.

The first approach, the workshop felt, would be very difficult to implement accurately without hardware support.

The second approach had two possible implementation models. The first was that at every preemption and release point: the run-time had to look at all running tasks and the group budget for all these tasks. For each group budget, the run-time divides the remaining budget by the number of running tasks and sets timers for this time. When timers go off, the budget has expired.

The second implementation model tries to reduce the run-time cost of the above approach by considering only task release points. When a timer expires in this case, the budget needs to be checked and the timers reset if the remaining budget is greater than 0.

Given the complexity of a multiprocessor group budget, the workshop supported the single-processor group budget approach. It was felt that the use cases for the multiprocessor cases were not clear from a scheduling point of view.

It was noted by the workshop, that the current Ada Reference Manual supported implicitly multiprocessor budgets and that this had to be changed.

7 Non SMP Architectures

Ada was designed to support multiprocessor applications where there is memory shared between each processor. Although the workshop had hoped to discuss non-SMP architectures, it was decided that this was a big issue and left to another day.

8 Ravenscar Issues

For the Ada Ravenscar Profile, the workshop believed that the most restrictive model is probably the best:

- No creation of allocation domains.
- Each tasks fixed to a single processor (either by the programmer, or by the system).
- Interrupts are fixed by the run-time to one of the processors in the system allocation domain.

However, the workshop did feel that having multiple allocation domains was not out of the question.

8.1 Non-multiprocessor Ravenscar issues

Three other non-multiprocessor Ravenscar issues were discussed during the first day of the workshop. One concerned the programming of 'recovery' after an executing-time overrun or a deadline miss. It was felt that a new profile, Ravenscar+, that was still significantly smaller than the full language, was desirable. The definition of such a profile will be discussed at the next workshop.

The second, minor, issue concerned the fact that although relative delays are not permitted in Ravenscar, a relative delay via a timing event was possible. The workshop felt that this bug in the language definition should be fixed.

The final issue concerned timers which are currently excluded from Ravenscar. A number of people felt that as library-level timing events were permitted then timers (restricted to one per task) should also be allowed. There was some concern voiced as to the asynchronous nature of timer events – Ravenscar has eliminated most such events. A vote showed a majority in favour of a change to the definition of Ravenscar to include timers.