Realtime Paradigms Needed Post Ada 2005

Stephen Michell Luke Wong Brad Moore
Maurya Software Inc CMCElectronics Inc. General Dynamics Canada
Ottawa, Ontario, Canada Ottawa Ontario, Canada Calgary, Alberts Canada
stephen.michell@maurya.on.ca Luke.wong@cmcelectronics.ca brad.moore@gdcanada.ca

Abstract
Ada 2005 significantly improved Ada's capabilities for handling real time systems by introducing scheduling and timing paradigms. To date, Ada's support for multiprocessor systems has not been as effective, even though Ada has concurrency directly in the language. This paper identifi es three missing paradigms from the language, mapping threads to processor\s, releasing many tasks from barriers, and Non-blocking_Delay, and proposes ways that they can be supported.

1. Introduction

The Ada Programming Language[Ada 2005] has made significant strides in implementing paradigms needed for real time systems for 2005. Technology keeps changing, adding additional processors, and more kinds of memory. Programs written in Ada need to be able to deal with these technologies in ways that let programs be as flexible and portable as possible. At IRTAW-13, one of us[Michell 2007] proposed ways that Ada could improve its posture by making it simpler to interface to POSIX, but also to other OS's such as Windows or embedded kernels.

We identify three necessary additions to Ada to better support concurrent systems, controlled mapping of threads to processors, releasing many tasks efficiently from a single barrier, and implementing a very small delay (Non_Blocking_Delay) capability.

ISO/IEC/JTC1/SC22/WG9 Ada Working Group is entertaining proposals for addition to the Ada programming language. Multiprocessor capabilities should be important additions. The approaches proposed in this paper should be considered for this revision.

This paper is organized as follows. Section 2 discusses the Set_Affinity capability. Section 3 discusses simultaneous release of tasks from barriers. Section 4 proposes a way to implement the “nanosleep” capability from POSIX. Section 5 gives conclusions.

2. Set Affinity Capability

Multiprocessor systems often have an non-uniform view of the hardware that is attached. Some processors may be uniquely attached to individual hardware, such as IO devices, interrupts or memory. Set_Affinity is a common paradigm in POSIX-based systems for allocating threads to processors, although is not in the POSIX standard [POSIX 2009].

We assume here a permissive view of the interrupt-processor model. Interrupts may be directed to all processors and handled by the first processor to acknowledge it, but we also consider the harder problem that interrupt – processor assignment is non-uniform and some (or all) may be handled by only a single processor.

[WB 2007] provides an excellent analysis of the problem areas. For the reasons given above, and identified in [WB 2007], protected objects and tasks that access such hardware must

- have methods to identify individual processors or sets of processors, and
- be mapped to the processor(s) in question.

We therefore propose a way of identifying and naming groups of processors, and a way to map threads to these processor groups. [WB 2007] propose a similar capability. This proposal goes further in that the role of protected objects in allocation of threads to processors is considered.

2.1. Name Processor Groups

Tasks and protected objects may be assigned to a single processor, or to a known group of processors with similar capabilities. Single processors could be identified by number, but groups of processors should have an identifying name.

We therefore propose to name a single processor, or a group of processors. A simple configuration pragma,

```
pragma Processor_Group(simple_name);
```

is reasonable to introduce a name for such a group. Similarly, the allocation of a processor to a processor group is also a configuration pragma,

```
pragma Set_Processor_To_Processor_Group(Processor_Id, Processor_Group_Name);
```

where Processor_Group_Name must be already declared at the point that such an assignment is done.

In order to permit every processor to be its own affinity group, we define that the name of the individual group for a single processor is the string equivalent of the processor number.

2.2 Set_Affinity - a per-object pragma.

Once processor groups have been declared, a single task's affinity for that processor group must be set. The most obvious way to do this is with a pragma, such as

```
task T is
  . . .
  pragma Processor_Affinity(task type | Protected type,
Processor_Group_Name);
end T;
```

If the processor group will change on a per-object basis, then a task discriminant or a protected type discriminant can be used to permit processor affinity to happen on a per-object basis.

An important part of this proposal is that the set_affinity pragma can be placed on a protected object, as well as a task object. There are a number of reasons for placing an affinity pragma on a protected object.

- If interrupts are enabled, then the interrupt procedures that form part of the protected object must execute on the named processor.
- A task that calls a protected operation on a protected object may not know the binding to a processor. The affinity statement in the protected object tells the runtime to transfer the execution of the task thread to the necessary processor as part of the call.
- A task may communicate with more than a single protected object that is managing hardware on different processors. In such cases, an affinity pragma on such a task would always bind it to only one processor.

For the reasons given above, we propose that *pragma Processor_Affinity* map *protected types* and protected objects (on a per-object basis) to processors.

In the case of protected objects mapped to a processor, this mapping will override any task-processor affinity while a relevant protected operation is being executed (by the task). This means that *task T* with *Processor_Affinity(PF1)* calling a protected object with *Processor_Affinity(PF2)* would execute on a *PF2* affinity processor while calling a protected operation of *P* with *Processor_Affinity(P, PF2)* ; This guarantees that potential execution of code on behalf of other tasks is execute on the correct processor.

2.3 Dynamic Affinities

There is also the question as to whether or not affinity should be static, or dynamic. It is our belief that affinity set at time of creation is sufficient, but dynamic affinities would be simple enough to achieve. For dynamic affinity, we propose a package *Processor_Affinity* as follows:

```
package System is
   ...
  type Processor_Affinity is -- implementation_defined -- but enumerable
   ...
 end System;
 -----------------------
```

```
with System;
package Ada.Processor_Affinity is
   exception Affinity_Error;

   function Affinity_Name(T : Task_ID; S : System.Processor_Affinity)
   return string;

   function Affinity_Of(S : String)
   return System.Processor_Affinity;

   procedure Set_Processor_Affinity(S : in out Affinity);
end Ada.Processor_Affinity;
```

Protected objects cannot be dynamically assigned Affinities.

So, to create per-object ***`Set_Affinity`***, a task discriminant and a pragma, or a call to ***`Set_Processor_Af-finity`*** will let multiple instances of one task be in different processor groups. For protected operations, using the pragma would guarantee that all tasks executing in that protected object execute on the correct processor to do the required work.

Errors in setting affinity would result in the predefined exception *Affinity_Error*. Errors in matching a string to *Affinity_Of*, or in the use of an unset Affinity result in the exception *Affinity_Error*;

Note that, if desired, additional subprograms to define *Processor_Group* names could be put into *Ada.Processor_Affinity* for a more dynamic connection, possibly using command line arguments.

3. Barrier Release

There are many situations where a group of tasks must wait for a single event and then be released together to process concurrently. This paradigm occurs naturally for tasks waiting on a single mailbox, and for tasks that are processing stripes in a CPU intensive problem such as vision processing or Fast Fourier Transforms. POSIX[POSIX 2009] provides a standardized way to create, initialize, wait on, and destroy barriers. The POSIX mechanism, however, is restricted to barrier release based upon the count of objects waiting on the barrier. We wish to permit more flexible ways to release waiting tasks, perhaps time-based or some other boolean condition.

There are two ways for an Ada task to wait for an event, either on a closed entry (in a protected object or in a task), and on a suspension object.

For a collection of tasks blocked on an entry, only the task at the head of an entry queue can be released, and then the entry conditions must be re-evaluated before the next task is released. This creates a sequential processing situation for releasing many tasks from a single entry. For a suspension object, Ada permits only a single task to wait on one suspension object, meaning that the releasing task would need to run a list of suspension objects to release a task collection, which again is a sequential operation.

Implementation-specific solutions can be provided, such as providing a pragma in an entry queue to have the runtime provide simultaneous release, but this hinders application portability as there can be a multitude of conditions reflecting if there is a maximum limit on queue length, if input parameters are allowed, if output parameters are allowed, and whether or not the entry itself can contain any code or access protected data. For these reasons, it is clear that a language-specified way of defining barriers is required.

There are two obvious choices for barriers: a specialized form of an entry, or a specialized form of a suspension object. We consider each in turn.

3.1. Entry approach to Barrier Release

This approach would provide a specialized entry via a pragma:

```
pragma Barrier_Entry(Entry_Name, Release_Count);
```

This pragma is applied to an entry or to an access-entry. The entry itself would be either a parameterless protected entry, or a protected entry with a specialized relationship between its in and out parameters. The body of the entry could be

- null for parameterless entries, or
- could calculate the *out* parameter(s) from existing protected state and the *in* parameters, but as for protected

functions, this entry would not be allowed to change the protected state.

This approach would let us extend the protected object definition so that multiple tasks can execute the same entry simultaneously, just as the definition permits multiple readers in functions.

The detection of assignment to PO state in the entry body would be a static check.

There are 3 outstanding issues in this approach.

1. If the barrier release condition is the number of waiting tasks, then there may never be a protected procedure caller that performs the release, and some optimization may be required to handle such cases.
2. The released tasks cannot run in parallel inside the protected object if protected data is set, so returning the boolean release condition to *False* must either be automatic, or a dedicated protected procedure call is required to set the barrier back to false.
3. If we wish to release based upon a count, then there will never be a release caller. For this reason, the optional *Release_Count* parameter is provided. When the number of queuing tasks reaches *Release_Count*, all tasks are released, as if a caller had called the entry.

   Note: This capability could be directly coded by having an initial always-open entry that every task calls, plus a requeue to the barrier entry. In this case, the boolean condition of

   ```
   when E'Count >= Release_Count-1 is ...
   ```

   would cause the correct behaviour.

   Note: The POSIX version of the release based on thread count makes one of the releasing threads unique. We could follow a similar approach, and have each task test for the unique attribute or value, and only the unique task may reset the Boolean guard.

3.2. Suspension Object approach to Barrier Release

The suspension object approach would see another object placed in package Synchronous_Control. We will call this object a Group_Suspension_Object. Each object declared has the maximum count of suspended tasks specified as it is declared, hence

```
package Ada.Synchronous_Control is
   type Suspension_Object is limited private;
   type Group_Suspension_Object(Count : Integer) is Limited_Private;
    -- a count of 0 means explicit release by a caller,
    -- non-zero means release when Count  tasks are suspended.
   procedure Set_True (S : in out Suspension_Object);
   procedure Set_True (S : in out Group_Suspension_Object);
   procedure Set_False(S : in out Suspension_Object);
   procedure Set_False(S : in out Group_Suspension_Object);
   procedure Current_Status(S : Suspension_Object) return Boolean;
   procedure Current_Status(S : Group_Suspension_Object) return Boolean;
   procedure Suspend_Until_True(S : in out Suspension_Object);
   procedure Suspend_Until_True(S      : in out Group_Suspension_Object;
                               Unique :    out Boolean;);

          . . .
end Ada.Synchronous_Control;
```

The actual object would be created as:

```
with  Ada.Synchronous_Control;
. . .
     -- create a barrier object for up to 10 tasks.
Local_Barrier : Ada.Synchronous_Control.Group_Suspension_Object(10);
```

And a task would

```
Suspend_Until_True(Local_Barrier);
```

Tasks would be released by a caller calling *Local_Barrier.Set_True* , or in the case of release-by-count, would be released when *Count* tasks are queued. In either case, somebody must reset the barrier to False. The *Unique* Boolean return value selects one task that may call *Local_Barrier.Set_False*;

The benefit of the suspension object approach is that this method can clearly be created to optimize the way that groups of tasks would interact. The downside is that there is no way to pass even a simple parameter, such as the stripe that a task is to process, upon release. Another issue is that more optimization would be required to handle automatic release of tasks when *Release_Count* is the defining release method.

Our preference is for the protected entry release mechanism, as it requires less optimizations and permits some level of protected entry processing to happen.

4. NonBlocking_Delay

Perhaps one of the trickiest situations for real time systems is trying to access hardware registers that have "settle time". This settle time is the time that a thread[1] must wait after it has accesses the register (read or write) before another access to that register or to registers controlled by the same hardware can be made. Such settle time is typically orders of magnitude slower than the timing for a single instruction, meaning that the thread must spin-wait or suspend until the register can be accessed.

If the task spin-waits, it consumes valuable CPU resources that may be needed for other tasks. The task also likely will not know how to calculate the number of "spins" to match the required waiting time.

If the thread suspends, then another thread may access the same protected object and may corrupt the object. Since the thread is executing inside a protected operation, we also have the Ada restriction that no potentially suspending operations are permitted.

It would seem reasonable on the surface that a controlled suspend on a protected object could solve the problem, especially since Ada already provides the "requeue" operation to let tasks executing in a PO suspend themselves and open the PO to other callers until conditions are right to resume execution.

The challenge is that the act of requeueing thread opens the protected object to be entered by another thread. The requeue operation and the setting of explicit protected state could be used to keep other callers on protected entries locked out, but callers of protected subprograms (such as an interrupt) would execute and may attempt to access the hardware that has not settled.

Therefore, to implement such a *NonBlocking_Delay*, the protected object must remain closed until the delaying task successfully exits the protected operation. This will mean that all protected operation locks must be true locks, not just priority-based locks. This, fortunately, is not a problem for multiprocessing systems, as priority-based locking is only effective for single CPU systems.

5. Conclusion

This paper provides some solutions to issues that interfere with real time paradigms and the ways that tasks and interrupts must interact. We have proposed a processor affinity operation, a way to release multiple blocked tasks simultaneously from a single barrier, and a proposal to permit miniscule blocking operations inside protected objects. We believe that it is appropriate to select reasonable approaches and alternatives from those proposed herein and given to the Ada Working Group as proposals for additional capability in a future revision of Ada.

1 The term "thread" is used here to refer to a task or an interrupt.

6. Bibliography

[Ada05] ISO/IEC 8652:2007, The Ada Programming Language

[BARNES2006] Barnes, John, Rationale for Ada 2005, available online from www.adaic.com

[Michell 2007] Michell, Stephen., "Interfacing Ada to Operating Systems", Proceedings of the 13 th Annual Real Time Ada Workshop, April 17-19 2007, ACM Ada Letters, Vol 27, Issue 2, July 2007, ACM, New York.

[POSIX2009] IEEE 1003.1 and IS9945-1:2009, The Portable Operating System Interface,

[WB 2007] Wellings A and Burns A., Beyond Ada 2005,
"Allocating Tasks to Processors in SMP Systems", Proceedings of the 13th Annual Real Time Ada Workshop, April 17-19 2007, ACM Ada Letters, Vol 27, Issue 2, July 2007, ACM, New York.

[WMM 2007]Wong, Luke, Michell, Stephen, Moore, Brad, Initial Work Scope Summary for updating Ada POSIX Binding IS 143519:2001 to the Ada Programming Language IS8652:2007, available from ISO/IEC/JTC1/SC22/WG9 Ada Working Group as document N477r.